Introduction to Programming Languages

Anthony A. Aaby

© 1996 by Anthony A. Aaby

Short Table of Contents

Preface

1. Introduction
2. Syntax
3. Semantics
4. Translation
5. Pragmatics
6. Abstraction and Generalization
7. Data and Data Structuring
8. Logic Programming
9. Functional Programming
10. Imperative Programming
11. Concurrent Programming
12. Object-Oriented Programming
13. Evaluation

Appendix
Stack machine
Unified Grammar
Logic
Bibliography
Definitions
Index
Long Table of Contents

Syntax, translation, semantics and pragmatics

1 Introduction
1.1 Data
1.2 Models of Computation
1.3 Syntax and Semantics
1.4 Pragmatics
1.5 Language Design Principles
1.6 Historical Perspectives and Further Reading
1.7 Exercises
2 Syntax
2.1 Context-free Grammars
2.1.1 Alphabets and Languages
2.1.2 Grammars and Languages
2.1.3 Abstract Syntax
2.1.4 Parsing
2.1.5 Table-driven and recursive descent parsing
2.2 Nondeterministic Pushdown Automata
2.2.1 Equivalence of pda and cfgs
2.3 Regular Expressions
2.4 Deterministic and Non-deterministic Finite State Machines
2.4.1 Equivalence of deterministic and non-deterministic fsa
2.4.2 Equivalence of fsa and regular expressions
2.4.3 Graphical Representation
2.4.4 Tabular Representation
2.4.5 Implementation of FSAs
2.5 Historical Perspectives and Further Reading
2.5.1 Backus-Naur Form
2.5.2 EBNF (extended BNF)
Introduction to Programming Languages

Declarative Semantics
Denotational Semantics
Pragmatics
Logic Programming and Software Engineering
The Logical Variable
Incomplete Data Structures
Arithmetic
Iteration vs Recursion
Backtracking
Exceptions
Logic Programming vs Functional Programming
Prolog and Logic
The Logic of Prolog
The Illogic of Prolog
Incompleteness
Unfairness
Unsoundness
Negation
Control Information
Extralogical Features
Multidirectionality
Rule Order
Historical Perspectives and Further Reading
Exercises

8 Functional Programming
8.1 The Lambda Calculus
8.1.1 Operational Semantics
8.1.2 Denotational Semantics
8.1.3 Translation Semantics and Combinators
8.2 Scheme
8.3 ML
8.4 Haskell
8.5 Historical Perspectives and Further Reading
8.6 Exercises
9 Imperative Programming
9. Historical Perspectives and Further Reading
9. Exercises
Pragmatics

10 Concurrent Programming
10.1 The Concurrent Nature of Systems
10.2 The Nature of Concurrent Systems
10.3 Concurrency in Programming Languages
10.4 The Engineering of Concurrent Programs
10.5 Historical Perspectives and Further Reading
10.6 Exercises

11 Object-Oriented Programming
11.1 Subtypes (subranges)
11.2 Objects
11.3 Classes
11.4 Inheritance
11.5 Types and Classes
11.6 Examples
11.7 Historical Perspectives and Further Reading
11.8 Exercises

12 Translation
12.1 Attribute Grammars and Static Semantics
12.2 Parsing
12.3 Scanning
12.4 Symbol Table
12.5 Virtual Computers
12.6 Optimization
12.7 Code Generation
12.8 Peephole Optimization
12.9 Historical Perspectives and Further Reading
12.10 Exercises

13 Evaluation
13. Historical Perspectives and Further Reading
13. Exercises Appendix

Logic
Bibliography
Definitions
Index
Supplementary Material

Code
Answers

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of Anthony A. Aaby. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee.

© 1998 Anthony A. Aaby. Send comments to aabyan@wwc.edu
Introduction

A complete description of a programming language includes the computational model, the syntax and semantics of programs, and the pragmatic considerations that shape the language.

Keywords and phrases: Computational model, computation, program, programming language, syntax, semantics, pragmatics, bound, free, scope, environment, block.

Suppose that we have the values $3.14$ and $5$, the operation of multiplication ($\times$) and we perform the computation specified by the following arithmetic expression

$$2 \times 3.14 \times 5$$

the result of which is the value:

$$31.4$$

If $3.14$ is an approximation for $\pi$, we can replace $3.14$ with $\pi$ abstracting the expression to:

$$2 \times \pi \times 5 \text{ where } \pi = 3.14$$

We say that $\pi$ is bound to $3.14$ and is a constant. The where introduces a local environment or block for local definitions. The scope of the definitions is just the expression. If $5$ is intended to be the value of a radius, then the expression can be generalized by introducing a variable for the radius:

$$2 \times \pi \times \text{radius} \text{ where } \pi = 3.14$$

Of course the value of the expression is the circumference of a circle so we may further abstract by assigning a name to the expression:

$$\text{Circumference} = 2 \times \pi \times \text{radius} \text{ where } \pi = 3.14$$

This last equation binds the name Circumference to the expression $2 \times \pi \times \text{radius} \text{ where } \pi = 3.14$. The variable radius is said to be free in the right hand side of the equation. It is a variable since its value is not determined. $\pi$ is not a variable, it is a constant, the name of a particular value. Any context (scope), in which this equation and the variable radius appears and radius is assigned
Introduction to a value, determines a value for Circumference. A further generalization is possible by parameterizing Circumference with the variable radius.

\[
\text{Circumference}(\text{radius}) = 2 \times \pi \times \text{radius} \text{ where } \pi = 3.14
\]

The variable radius appearing in the right hand side is no longer free. It is bound to the parameter radius. Circumference has a value (other than the right hand side) only when the parameter is replaced with an expression. For example, in

\[
\text{Circumference}(5) = 3.14
\]

The parameter radius is bound to the value 5 and, as a result, Circumference(5) is bound to 3.14. In this form, the definition is a recipe or program for computing the circumference of a circle from the radius of the circle. The mathematical notation (syntax) provides the programming language and arithmetic provides the computational model for the computation. The mapping from the syntax to the computational model provides the meaning (semantics) for the program. The notation employed in this example is based on the very pragmatic considerations of ease of use and understanding. It is so similar to the usual mathematical notation that most people have difficulty in distinguishing between the syntax and the computational model.

This example serves to illustrate several key ideas in the study of programming languages which are summarized in definition 1.1.

---

**Definition 1.1**

1. A **computational model** is a collection of values and operations.
2. A **computation** is the application of a sequence of operations to a value to yield another value.
3. A **program** is a specification of a computation.
4. A **programming language** is a notation for writing programs.
5. The **syntax** of a programming language refers to the structure or form of programs.
6. The **semantics** of a programming language describe the relationship between a program and the model of computation.
7. The **pragmatics** of a programming language describe the degree of success with which a programming language meets its goals both in its faithfulness to the underlying model of computation and in its utility for human programmers.
A program can be viewed as a function, the output data values are a function of the input data values.
Output = Program(Input)

Another view of a program is that it models a problem domain and the execution of the program is a simulation of the problem domain.

Program = Model of a problem domain
Execution of a program = simulation of the problem domain

In any case, data objects are central to programs.

The values can be separated into two groups, primitive and compound. The primitive values are usually numbers, boolean values, and characters. The composite values are usually arrays, records, and recursively defined values. Strings may occur as either primitive or composite values. Lists, stacks, trees, and queues are examples of recursively defined values.

Associated with the primitive values are the usual operations (e.g., arithmetic operations for the numbers). Associated with each composite value are operations to construct the values of that type and operations to access component elements of the type. A collection of values that share a common set of operations is called a data type.

The primitive types are implemented using the underlying hardware and, sometimes, special purpose software. So that only appropriate operations are applied to values, the value's type must be known. In assembly language programs it is up to the programmer to keep track of a datum's type. Type information is contained in a descriptor.

| Descriptor | Value |

When the type of a value is known at compile time the type descriptor is a part of the compiler's symbol table and the descriptor is not needed at run-time and therefore, the descriptor is discarded after compilation. When the type of a value is not known until run-time, the type descriptor must be associated with the value to permit type checking.

**Boolean values** are implemented using a single bit of storage. Since single bits are not usually addressable, the implementation is extended to be a single addressable unit of memory. In this case either a single bit within the addressable unit is used for the value or a zero value in the storage unit designates false while any non-zero value designates true. Operation on bits and boolean values are included in processor instruction sets.

**Integer values** are most often implemented using a hardware defined integer storage representation,
often 32-bits or four bytes with one bit for the sign.

\[
\begin{array}{c|cccc}
\text{sign} & 7\text{-bits} & \text{byte} & \text{byte} & \text{byte} \\
\end{array}
\]

The integer arithmetic and relational operations are implemented using the set of hardware operations. The storage unit is divided into a sign and a binary number. Since the integers form an infinite set, only a subrange of integers is provided. Some languages (for example Lisp and Scheme) provide for a greatly extended range by implementing integers in lists and providing the integer operations in software. This provides for ``infinite" precision arithmetic.

**Natural number** values are most often implemented using the hardware defined storage unit. The advantage of providing an natural number type is that an additional bit of storage is available thus providing larger positive values than are provided for integer values.

**Rational number** values may be implemented as pairs of integers. Rationals are provided when it is desired to avoid the problems of round off and truncation which occurs when floating point numbers are used to represent rational numbers.

**Real number** values are most often implemented using a hardware defined floating point representation. One such representation consists of 32-bits or four bytes where the first bit is the sign, the next seven bits the exponent and the remaining three bytes the mantissa.

\[
\begin{array}{c|cccc}
\text{sign} & \text{exponent} & \text{byte} & \text{byte} & \text{byte} \\
\end{array}
\]

The floating point arithmetic and relational operations are implemented using the set of hardware operations. Some floating point operations such as exponentiation are provided in software. The storage unit is divided into a mantissa and an exponent. Sometimes more than one storage unit is used to provide greater precision.

**Character** values are almost always supported by the underlying hardware and operating system, usually one byte per character. Characters are encoded using the 8-bit ASCII or EBCDIC encoding scheme or the emerging 16-bit Unicode encoding scheme.

**Enumeration** values are usually represented by a subsequence of the integers and as such inherit an appropriate subset of the integer operations.

Where **strings** are treated as a primitive type, they are usually of fixed length and their operations are implemented in hardware.
Compound (or structured) data types include arrays, records, and files.

**Abstract data types** are best implemented with pointers. The user program holds a pointer to a value of the abstract type. This use of pointers is quite safe since the pointer manipulation is restricted to the implementation module and the pointer is notationally hidden.

**Models of Computation**

There are three basic computational models -- functional, logic, and imperative. In addition to the set of values and associated operations, each of these computational models has a set of operations which are used to define computation. The functional model uses function application, the logic model uses logical inference and the imperative model uses sequences of state changes.

**The Functional Model**

The *functional model* of computation consists of a set of values, functions, and the operations of function application and function composition. In addition to the usual values, functions can take other functions as arguments and return functions as results (higher-order functions). A program is a collection of definitions of functions and a computation is function application (evaluation of an expression).

The initial example of the computation of the circumference of a circle is an example of functional programming. A more interesting example is a program to compute the standard deviation of a list of scores. The formula for standard deviation is:

\[
\sigma = \sqrt{\left( \frac{\sum_{i=1}^{N} x_i^2}{N} - \left[ \frac{\sum_{i=1}^{N} x_i}{N} \right]^2 \right)}
\]

where \( x_i \) is an individual score and \( N \) is the number of scores. The formula requires computing both the sum of the scores and the sum of the squares of the scores. The higher-order function `map` applies a function to each element of a list and the higher-order function `fold` reduces a list by applying a function to the first element of a list and the result of folding the rest of the list. Figure 1 illustrates what an implementation in a functional programming language might look like.

---

**Figure 1.1:** Standard deviation using higher-order functions

\[
sd(xs) = \sqrt{v} \\
\text{where} \\
\begin{align*}
    n &= \text{length}(xs) \\
    v &= \text{fold}(plus, \text{map}(sqr, xs))/n \\
    &\quad - \text{sqr}(\text{fold}(plus, xs)/n)
\end{align*}
\]
The functional model is important because it has been under development for hundreds of years and its notation and methods form the base upon which a large portion of our problem solving methodologies rest. The prime concern in functional programming is defining functional relationships.

The Logic Model

The logic model of computation consists of a set of values, definitions of relations and logical inference. Programs consist of definitions of relations and a computation is a proof (a sequence of inferences). For example the earlier circumference computation can be represented as:

\[ \text{circle}(R, C) \text{ if } \pi = 3.14 \text{ and } C = 2 \times \pi \times R. \]

The function is represented as a relation between \( R \) and \( C \).

A better illustration logic programming is a program to determine the mortality of Socrates and Penelope. We begin with the fact that Socrates and Penelope are human and the rule that all humans are mortal or equivalently for all \( X \), if \( X \) is human then \( X \) is mortal. An equivalent form of the fact and rule are:

\[
\begin{align*}
\text{human}(\text{Socrates}) \\
\text{human}(\text{Penelope}) \\
\text{mortal}(X) \text{ if human}(X)
\end{align*}
\]

To determine the mortality of Socrates or Penelope we make the assumption that there are no mortals i.e.

\[ \neg \text{mortal}(Y) \]
Figure 2 contains a computation (proof) that Socrates and Penelope are mortal.

**Figure 1.3: Socrates is mortal**

1a. `human(Socrates)`  Fact  
1b. `human(Penelope)`  Fact  
2. `mortal(X) if human(X)`  Rule  
3. `¬mortal(Y)`  Assumption  
4a. `X = Y`  from 2 & 3 by unification  
4b. `¬human(Y)` and modus tollens  
5a. `Y = Socrates`  from 1 and 4 by unification  
5b. `Y = Penelope`  
6. Contradiction 5a, 4b, and 1a; 5b, 4b and 1b

The first step in the computation is the deduction of line 4 from lines 2 and 3. It is justified by the inference rule *modus tollens* which states that if the conclusion of a rule is known to be false, then so is the hypothesis. The variables X and Y may be *unified* since they may have any value. By unification, Lines 5a, 4b, and 1a; 5b, 4b and 1b produce contradictions and identify both Socrates and Penelope as mortal.

*Resolution* is the an inference rule which looks for a contradiction and it is facilitated by *unification* which determines if there is a substitution which makes two terms the same. The logic model is important because it is a formalization of the reasoning process. It is related to relational data bases and expert systems. The prime concern in logic programming is defining relationships.

**Figure 1.4: Logic Programming**

<table>
<thead>
<tr>
<th>values</th>
<th>relations</th>
<th>logical inference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program = set of relation definitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computation = constructive proof (inference from definitions)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Inferences**

*program = set of axioms* -- the formalization of knowledge  
*computation = constructive proof of a goal statement from the program*
The Imperative Model

The imperative model of computation consists of a set of values including a state and the operation of assignment to modify the state. State is the set of name-value pairs of the constants and variables. Programs consist of sequences of assignments and a computation is a sequence of states. Each step in the computation is the result of an assignment operation (See Figure 3).

For example, an imperative implementation of the earlier circumference computation might be written as:

```plaintext
constant pi = 3.14

input (Radius) Circumference := 2 * pi * Radius Output (Circumference)
```

The computation requires the implementation to determine the value of Radius and pi in the state and then change the state by pairing Circumference with a new value.

It is easier to keep track of the state when state information is included with the code.

```plaintext
constant pi = 3.14

Radius _|_, Circumference = _|_, pi=3.14
input (Radius)
Radius x, Circumference = _|_, pi=3.14
Circumference := 2 * pi * Radius
Radius x, Circumference = 2 × x × pi, pi=3.14
Output (Circumference)
Radius x, Circumference = 2 × x × pi, pi=3.14
```

where _|_ designates an undefined value.
The imperative model is often called the *procedural model* because groups of operations are abstracted into *procedures*. The imperative-procedural model is important because it models change and changes are an integral part of our environment. It is the model of computation that is closest to the hardware on which programs are executed. Its closeness to hardware makes it the easiest to implement and imperative programs tend to make the least demands for system resources (time and space). The prime concern in imperative programming is defining a sequence of state changes.

**Figure 1.6: Imperative Programming**

<table>
<thead>
<tr>
<th>memory cells</th>
<th>Program = sequence of commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>Computation = sequence of state changes</td>
</tr>
</tbody>
</table>

**Computability**

The functional, logic and imperative models of computation are equivalent in the sense that any problem that has a solution in one model is solvable (in principle) each of the other models. Other models of computation have been proposed. The other models have been shown to be equivalent to these three models. These are said to be *universal* models of computation.

The method of computation provided in a programming language is dependent on the model of computation implemented by the programming language. Most programming languages utilize more than one model of computation but one model usually predominates. Lisp, Scheme, and ML are based on the functional model of computation but provide some imperative constructs while, Miranda and Haskell provide a nearly pure implementation of the functional model of computation. Prolog provides a partial implementation of the logic computational model but, for reasons of efficiency and practicality, fails in several areas and contains imperative constructs. The language Gödel is much closer to the ideal. The imperative model requires some functional and logical elements and languages such as Pascal, C/C++, Ada and Java emphasize assignments, methods of defining various computation sequences and provide minimal implementations of the functional and logic model of computation.

**Syntax and Semantics**

Syntax describes the structure of programs and semantics defines the relationship between the syntax and the computational model. To simplify the task of reasoning about programs, the syntax of a programming language should be closely related to the computational model. The key principle is the principle of clarity.
Principle of Clarity

The structure of a programming language should be well defined, and the outcome of a particular section of code easily predicted.

The notation used in the functional and logic models reflects common mathematical practice and exhibits the notational simplicity and regularity found in that discipline. Because the notation used for the imperative model must be able to specify both a variety of state sequences and expressions, it tends to be irregular and of greater complexity than the notation for functional and logic models. Because of this complexity and the wide spread use of imperative programming languages, the bulk of the work done in the area of programming language semantics deals with imperative programming languages.

Pragmatics

Pragmatics is concerned about the usability of the language, the application areas, ease of implementation and use, and the language's success in fulfilling its design goals. The forces that shape a programming language include computer architecture, software engineering practices (especially the software life cycle), computational models, and the application domain (e.g. user interfaces, systems programming, and expert systems).

For a language to have wide applicability it must make provision for abstraction, generalization and modularity. Abstraction (associating a name with an object and using the name to whenever the object is required) permits the suppression of detail and provides constructs which permit the extension of a programming language. These extensions are necessary to reduce the complexity of programs. Generalization (replacing a constant with a variable) permits the application of constructs to more objects and possibly to other classes of objects. Modularity is a partitioning of a program into sections usually for separate compilation and into libraries of reusable code. Abstraction, generalization and modularity ease the burden on a programmer by permitting the programmer to introduce levels of detail and logical partitioning of a program. The implementation of the programming language should be faithful to the underlying computational model and be an efficient implementation.

Concurrent programming involves the notations for expressing potential parallel execution of portions of a program and the techniques for solving the resulting synchronization and communication problems. The concurrent programming may be implemented within any of the computational models. Concurrency within the functional and logic model is particularly attractive since, subexpression evaluation and inferences may be performed concurrently and requires no additional syntax. Concurrency in the imperative model requires additional syntactic elements.

Object-oriented programming OOP involves the notations for structuring a program into a collection of objects which compute by exchanging messages. Each object is bound up with a value and a set of operations which determine the messages to which it can respond. The objects are organized
hierarchically and inherit operations from objects higher up in the hierarchy. Object-oriented programming may be implemented within any of the other computational models.

Programs are written and read by humans but are executed by computers. Since both humans and computers must be able to understand programs, it is necessary to understand the requirements of both classes of users.

The native programming languages of computers bear little resemblance to natural languages. Machine languages are unstructured and contain few, if any, constructs resembling the level at which humans think. The instructions typically include arithmetic and logical operations, memory modification instructions and branching instructions. For example, the circumference computation might be written in assembly language as:

```assembly
Load Radius R1
Mult R1  2 R1
Load Pi R2
Mult R1 R2 R1
Store R1 Circumference
```

Because the imperative model is closer to actual hardware, imperative programs have tended to be more efficient in their use of time and space than equivalent functional and logic programs.

Natural languages are not suitable for programming languages because humans themselves do not use natural languages when they construct precise formulations of concepts and principles of particular knowledge domains. Instead, they use a mix of natural language, formalized symbolic notations of mathematics and logic and diagrams. The most successful of these symbolic notations contain a few basic objects which may be combined through a few simple rules to produce objects of arbitrary levels of complexity. In these systems, humans reduce complexity by the use of definitions, abstractions, generalizations and analogies. Successful programming languages do the same by catering to the natural problem solving approaches used by humans. Ideally, programming languages should approach the level at which humans reason and should reflect the notational approaches that humans use in problem solving and must include ways of structuring programs to ease the tasks of program understanding, debugging and maintenance.

**Language Design Principles**

Programming languages are largely determined by the importance the language designers attach to the computational model, the intended application domain readability, write-ability and efficient execution. Some languages are largely determined by the necessity for efficient implementation and execution. Others are designed to be faithful to a computational model.

Research in computer architecture is producing...
producing more efficient implementations of programming languages for all models of computation.

Research in software engineering is producing a better understanding of the program structuring techniques that lead to programs that are easier to write, read (understand), and maintain.

All general purpose programming languages adhere to the following programming language design principle.

**Principle of Computational Completeness**

The computational model for a general purpose programming language must be *universal*.

**Principle of Implementation**

The implementation should be efficient in its use of space and time.

**Principle of Programming**

The program should be written in a language that reflects the problem domain.

The line of reasoning developed above may be summarized in the following principle.

**Principle of Programming Language Design**

A programming language must be designed to facilitate *readability* and *writ-ability* for its human users and efficient execution on the available hardware.

Readability and write-ability are facilitated by the following principles.

**Principle of Simplicity**

The language should be based upon as few

**Principle of Orthogonality**

Independent functions should be controlled by independent mechanisms.

**Principle of Regularity**

A set of objects is said to be regular with respect to some condition if, and only if, the condition is applicable to each element of the set.

**Principle of Extensibility**

New objects of each syntactic class may be constructed (defined) from the basic and defined constructs in a systematic way.

The principle of regularity and and extensibility require that the basic concepts of the language should be applied consistently and universally.

In the following pages we will study programming languages as the realization of computational models, semantics as the relationship between computational models and syntax, and associated pragmatic concerns.
Historical Perspectives and Further Reading

For a programming languages text which presents programming languages from the virtual machine point of view see Pratt, from the point of view of denotational semantics see Tennent, and from a programming methodology point of view see Hehner.

  *The Logic of Programming* Prentice-Hall International.
  *Programming Languages: Design and Implementation* 3rd ed. Prentice-Hall.
  *Principles of Programming Languages* Prentice-Hall International.

Exercises

1. Identify the applicable scope rules in Figure 2.
2. Construct a trace of the execution of the following program (i.e. complete the following proof).
   1. parentOf(john, mary).
   2. parentOf(kay, john).
   3. parentOf(bill, kay).
   4. ancestorOf(X,Y) if parentOf(X,Y).
   5. ancestorOf(X,Z) if parentOf(X,Y) and ancestorOf(Y,Z).
   6. not ancestorOf(bill,mary).
3. Construct a trace of the execution of \( \text{fac}(4) \) given the function definition

   \[
   \text{fac}(N) = \begin{cases} 
   1 & \text{if } N = 0 \\
   N \times \text{fac}(N-1) & \text{else}
   \end{cases}
   \]

4. Construct a trace of the execution of the following program

   \[
   \begin{align*}
   N & := 4; \\
   F & := 1; \\
   \text{While } N > 0 \text{ do} \\
   & \quad F := N \times F; \\
   & \quad N := N - 1;
   \end{align*}
   \]

5. Using the following definition of a list,

   \[
   \begin{align*}
   \text{list}([\ ] ) & -- \text{the empty list} \\
   \text{list}([X|L]) & -- \text{first element is } X \text{ the rest of}
   \end{align*}
   \]
the list is \( L \)

\[ X_0, \ldots X_n \] is an abbreviation for \( [X_0|\ldots[X_n|[ ]]|\ldots] \)

Complete the following computation (proof) and determine the result of concatenating the two lists.

1. \( \text{concat}([\ ], L, L) \quad \text{Fact} \)
2. \( \text{concat}([X|L_0], L_1, [X|L_2]) \text{ if } \text{concat}(L_0, L_1, L_2) \quad \text{Rule} \)
3. \( \neg\text{concat}([0,1], [a,b], L) \quad \text{Assumption} \)

6. Classify the following languages in terms of a computational model: Ada, APL, BASIC, C, COBOL, FORTRAN, Haskell, Icon, LISP, Pascal, Prolog, SNOBOL.

7. For the following applications, determine an appropriate computational model which might serve to provide a solution: automated teller machine, flight-control system, a legal advice service, nuclear power station monitoring system, and an industrial robot.

8. Compare the syntactical form of the if-command/expression as found in Ada, APL, BASIC, C, COBOL, FORTRAN, Haskell, Icon, LISP, Pascal, Prolog, SNOBOL.

9. An extensible language is a language which can be extended after language design time. Compare the extensibility features of C or Pascal with those of LISP or Scheme.

10. What programming language constructs of C are dependent on the local environment?

11. What languages provide for binding of type to a variable at run-time?

12. Discuss the advantages and disadvantages of early and late binding for the following language features. The type of a variable, the size of an array, the forms of expressions and commands.

13. Compare two programming languages from the same computational paradigm with respect to the programming language design principles.

14. Construct a program in your favorite language to do one of the following:
   a. Perform numerical integration where the function is passed as a parameter.
   b. Perform sorting where the less-than function is passed as a parameter.
Syntax

The syntax of a programming language describes the structure of programs without any consideration of their meaning.

Keywords and phrases: Regular expression, regular grammar, context-free grammar, parse tree, ambiguity, BNF, context sensitivity, attribute grammar, inherited and synthesized attributes, scanner, lexical analysis, parser, static semantics.

Syntax is concerned with the structure of programs and layout with their appearance. The syntactic elements of a programming language are determined by the computation model and pragmatic concerns. There are well developed tools (regular, context-free and attribute grammars) for the description of the syntax of programming languages. Grammars are rewriting rules and may be used for both recognition and generation of programs. Grammars are independent of computational models and are useful for the description of the structure of languages in general.

Context-free grammars are used to describe the bulk of the language's structure; regular expressions are used to describe the lexical units (tokens); attribute grammars are used to describe the context sensitive portions of the language. Attribute grammars are described in a later chapter.

Alphabets and Languages

A language is formally defined by its words and its sentence structure. In this section, we develop the basic notions of words and sentences as strings of words. The collection of words from which sentences are constructed is called an alphabet and a language is a collection of strings formed from the elements of the alphabet. A string with no words from the alphabet is called the empty string and is denoted by lambda. Definition 2.1 formalizes these notions.

**Definition 2.1: Alphabet and Language**

\[ \Sigma \]

An alphabet \( \Sigma \) is a nonempty, finite set of symbols.

\[ L \]

A language \( L \) over an alphabet \( \Sigma \) is a collection of strings of elements of \( \Sigma \).
empty string $\lambda$ is a string with no symbols at all.

The set of all possible finite strings of elements of $\Sigma$ is denoted by $\Sigma^*$. $\lambda$ is an element of $\Sigma^*$.

A string is a finite sequence of symbols from an alphabet, Sigma. The concatenation of two strings $v$ and $w$ is the string $wv$ obtained by appending the string $w$ to the right of string $v$.

Programming languages require two levels of description, the lowest level is that of a token. The tokens of a programming language are the keywords, identifiers, constants and other symbols appearing in the language. In the program

```c
void main()
{
  printf("Hello World\n");
}
```

the tokens are

```c
void, main, (, ), {, printf, (, "Hello World\n", ), ;, }
```

The alphabet for the language of the lexical tokens is the character set while the alphabet for a programming language is the set of lexical tokens;

A string in a language $L$ is called a sentence and is an element of $\Sigma^*$. Thus a language $L$ is a subset of $\Sigma^*$. $\Sigma^+$ is the set of all possible nonempty strings of $\Sigma$, so $\Sigma^+ = \Sigma^* - \{ \text{lambda} \}$. A token is a sentence in the language for tokens and a program is a sentence in the language of programs.

If $L_0$ and $L_1$ are languages, then $L_0L_1$ denotes the language $\{xy \mid x \text{ is in } L_0, \text{ and } y \text{ is in } L_1 \}$. That is $L_0L_1$ consists of all possible concatenations of a string from $L_0$ followed by a string from $L_1$.

Grammars and Languages

The ordering of symbols within a token are described by regular expressions. The ordering of symbols within a program are described by context-free grammars. In this section, we describe context-free grammars. A later section describes regular expressions. Context-free grammars describe how lexical units (tokens) are grouped into meaningful structures. The alphabet (the set of lexical units) consists of the keywords, identifiers, constants, punctuation symbols, and various operators. While context-free
grammars are sufficient to describe most programming language constructs, they cannot specify context-sensitive aspects of a language such as the requirements that a name must be declared before it is referenced, the order and number of actual parameters in a procedure call must match the order and number of formal arguments in a procedure declaration, and that types must be compatible on both sides of an assignment operator.

A grammar consists of a finite collection of grammatical categories (e.g. noun phrase, verb phrase, article, noun, verb etc), individual words (elements of the alphabet), rules for describing the order in which elements of the grammatical categories must appear and there must be a most general grammatical category. Figure 2.1 contains a context-free grammar for a fragment of English.

---

**Figure 2.1:** $G_0$ a grammar for a fragment of English

The grammatical categories are: S, NP, VP, D, N, V.
The words are: a, the, cat, mouse, ball, boy, girl, ran, bounced, caught.
The grammar rules are:

\[
\begin{align*}
S & \rightarrow NP \ VP \\
NP & \rightarrow N \\
NP & \rightarrow D \ N \\
VP & \rightarrow V \\
VP & \rightarrow V \ NP \\
V & \rightarrow ran \ | \ bounced \ | \ caught \\
D & \rightarrow a \ | \ the \\
N & \rightarrow cat \ | \ mouse \ | \ ball \ | \ boy \ | \ girl
\end{align*}
\]

The most general category is $S$, a sentence.

---

In a context-free grammar, the grammatical categories are called *variables*, the words (tokens) are called *terminals*, the grammar rules are rewriting rules called *productions*, and the most general grammatical category is called the *start symbol*. This terminology is restated in Definition 2.2.

---

**Definition 2.2:** Context-free grammar

Context-free grammar $G$ is a quadruple
\[ G = (V, T, P, S) \]

where

- \( V \) is a finite set of variable symbols,
- \( T \) is a finite set of terminal symbols disjoint from \( V \),
- \( P \) is a finite set of rewriting rules (productions) of the form
  \[ A \rightarrow w \text{ where } A \in V, \ w \in (V \cup T)^* \]
- \( S \) is an element of \( V \) called the **start** symbol.

Grammars may be used to generate the sentences of a language. Given a string \( w \) of the form

\[ w = uxv \]

the production \( x \rightarrow y \) is applicable to this string since \( x \) appears in the string. The production allows us to replace \( x \) with \( y \) obtaining the string \( z \)

\[ z = uyv \]

and say that \( w \) derives \( z \). This is written as

\[ w \Rightarrow z \]

If

\[ w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n \]

we say that \( w_1 \) derives \( w_n \) and write

\[ w_1 \Rightarrow^* w_n \]

The set of sentences of a language are derived from the start symbol of the grammar. Definition 2.3 formalizes these ideas.

**Definition 2.3:** Generation of a Language from the Grammar
Let $G$ be a grammar. Then the set

$$L(G) = \{ w \in T^* \mid S \Rightarrow^* w \}$$

is the language generated by $G$.

A language $L$ is context-free iff there is a context-free grammar $G$ such that $L = L(G)$.

If $w$ in $L(G)$, then the sequence

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots \Rightarrow w_n \Rightarrow w$$

is a derivation of the sentence $w$ and the $w_i$ are called sentential forms.

Using the grammar $G_0$ the sentence *the cat caught the mouse* can be generated as follows:

$S \Rightarrow NP \ VP$
$\Rightarrow D \ N \ VP$
$\Rightarrow the \ N \ VP$
$\Rightarrow the \ cat \ VP$
$\Rightarrow the \ cat \ V \ NP$
$\Rightarrow the \ cat \ caught \ NP$
$\Rightarrow the \ cat \ caught \ D \ N$
$\Rightarrow the \ cat \ caught \ the \ N$
$\Rightarrow the \ cat \ caught \ the \ mouse$

This derivation is performed in a *leftmost* manner. That is, in each step the leftmost variable in the sentential form is replaced.

Sometimes a derivation is more readable if it is displayed in the form of a derivation tree.
The notion of a tree based derivation is formalized in Definition 2.5.

**Definition 2.5: Derivation Tree**

Let $G = (V, T, P, S)$ be a context-free grammar. A *derivation tree* has the following properties.

1. The root is labeled $S$.
2. Every interior vertex has a label from $V$.
3. If a vertex has label $A$ in $V$, and its children are labeled (from left to right) $a_1, ..., a_n$, then $P$ must contain a production of the form
   
   $A \rightarrow a_1...a_n$

4. Every leaf has a label from $T$ union \{\texttt{lambda}\}.

In the generation example we chose to rewrite the left-most nonterminal first. When there are two or more left-most derivations of a string in a given grammar or, equivalently, there are two distinct derivation trees for the same sentence, the grammar is said to be *ambiguous*. In some instances, ambiguity may be eliminated by the selection of another grammar for the language or adding rules which may not be context-free rules. Definition 2.6 defines ambiguity in terms of derivation trees.

**Definition 2.6: Ambiguous Grammar**

A context-free grammar $G$ is said to be ambiguous if there exists some $w$ in $L(G)$ which has two distinct derivation trees.

**Abstract Syntax**

Programmers and compiler writers need to know the actual symbols used in programs -- the *concrete syntax*. A grammar defining the concrete syntax of arithmetic expressions is grammar $G_1$ in Figure 2.2.
We assume that \( c \) and \( id \) stand for any constants and identifiers respectively. Concrete syntax is concerned with the hierarchical relationships and the particular symbols used. The main point of abstract syntax is to omit the details of physical representation, specifying the pure structure of the language by specifying the logical relations between parts of the language. A grammar defining the abstract syntax of arithmetic expressions is grammar \( G_2 \) in Figure 2.3.

The terminal symbols are names for classes of objects.

An additional difference between concrete and abstract syntax appears. The key difference in the use of concrete and abstract grammars is best illustrated by comparing the derivation tree and the abstract syntax tree for the expression \( id + (id * id) \). The derivation tree for the concrete grammar is just what we would expect:

\[
E
\]
while the abstract syntax tree for the abstract grammar is quite different.

In a derivation tree for an abstract grammar, the internal nodes are labeled with the operator and the the operands are their children and there are no concrete symbols in the tree. Abstract syntax trees are used by compilers for an intermediate representation of the program.

Concrete syntax defines the way programs are written while abstract syntax describes the pure structure of a program by specifying the logical relation between parts of the program. Abstract syntax is important when we are interested in understanding the meaning of a program (its semantics) and when translating a program to machine code.

**Parsing**

Grammars may be used both for the generation and recognition (parsing) of sentences. Both generation and recognition requires finding a rewriting sequence consisting of applications of the rewriting rules which begins with the grammar's start symbol and ends with the sentence. The recognition of a program in terms of the grammar is called *parsing*. An algorithm which recognizes programs is called a *parser*. A parser either implicitly or explicitly builds a derivation tree for the sentence.

There are two approaches to parsing. The parser can begin with the start symbol of the grammar and attempt to generate the same sentence that it is attempting to recognize or it can try to match the input to the right-hand side of the productions building a derivation tree in reverse. The first approach is called *top-down parsing* and the second, *bottom-up* parsing.

Figure 2.4 illustrates top-down parsing by displaying both the parse tree and the remaining unrecognized input. The input is scanned from left to right one token at a time.
Each line in the figure represents a single step in the parse. Each nonterminal is replaced by the right-hand side defining it. Each time a terminal matches the input, the corresponding token is removed from the input.

Figure 2.4: Top-down Parse

Figure 2.5 illustrates bottom-up parsing by displaying both the parse tree and the remaining unrecognized input. Note that the parse tree is constructed up-side down, i.e., the parse tree is built in reverse.

Figure 2.5: Bottom-up Parse
Each line represents a step in the parsing sequence. The input tokens shifted from the input to the parse tree when the parser is unable to reduce branches of the tree to a variable.

**Table-driven and recursive descent parsing**

The simplest and most intuitive approach to constructing a parser is to translate the grammar into a collection of recursive routines. Such a parser is called a *recursive descent parser*. A procedure `parseN` is constructed for each variable `N`. The right-hand side of the productions determine the body of the parse procedure. Variables in the right-hand side become calls to a parse procedure. Terminals in the right-hand side are translated to a verification check to make sure the input corresponds to the terminal and a procedure call to get the next input token. Additional details and restrictions on grammars are
An alternate approach is to construct *top-down table-driven* parser which consists of a driver routine, a stack and the grammar (usually stored in tabular form). The driver routine follows the following algorithm:

1. Initialize the stack with the start symbol of the grammar.
2. Repeat until no further actions are possible
   a. If the top of the stack and the next input symbol are the same, pop the top of the stack and consume the input symbol.
   b. If the top of the stack is a nonterminal symbol, pop the stack and push the right hand side of the corresponding grammar rule onto the stack.
3. If both the stack and input are empty, accept the input otherwise, reject the input.

To illustrate this approach we use the grammar \( G_1 \) for expressions and parse the expression \( id + id \ast id \). Figure 2.6 contains a trace of the parse.

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>RULE/ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>[E]</td>
<td>id+id*id]</td>
<td>pop &amp; push using E ( \rightarrow ) E</td>
</tr>
<tr>
<td>+E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E+E]</td>
<td>id+id*id]</td>
<td>pop &amp; push using E ( \rightarrow ) id</td>
</tr>
<tr>
<td>id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>id+E]</td>
<td>id+id*id]</td>
<td>pop &amp; consume</td>
</tr>
<tr>
<td>+E]</td>
<td>+id*id]</td>
<td>pop &amp; consume</td>
</tr>
<tr>
<td>E]</td>
<td>id*id]</td>
<td>pop &amp; push using E ( \rightarrow )</td>
</tr>
<tr>
<td>E*E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E*E]</td>
<td>id*id]</td>
<td>pop &amp; push using E ( \rightarrow ) id</td>
</tr>
<tr>
<td>*E]</td>
<td>*id]</td>
<td>pop &amp; consume</td>
</tr>
<tr>
<td>E]</td>
<td>id]</td>
<td>pop &amp; push using E ( \rightarrow ) id</td>
</tr>
<tr>
<td>id</td>
<td>id]</td>
<td>pop &amp; consume</td>
</tr>
<tr>
<td>]</td>
<td>]</td>
<td>accept</td>
</tr>
</tbody>
</table>

The trace shows the contents of the stack and the remaining input at each step of the parse.
A third alternative is to construct a *bottom-up table driven* parser which consists of a driver routine, a stack and a grammar stored in tabular form. The driver routine follows the following algorithm:

1. Initially the stack is empty.
2. Repeat until no further actions are possible.
   a. If the top \( n \) stack symbols match the right hand side of a grammar rule in reverse, then reduce the stack by replacing the \( n \) symbols with the left hand symbol of the grammar rule.
   b. If no reduction is possible then shift the current input symbol to the stack.
3. If the input is empty and the stack contains only the start symbol of the grammar, then accept the input otherwise, reject the input.

To illustrate this approach we use the grammar \( G_1 \) for expressions and parse the expression \( id+id*id \).

Figure 2.7 contains a trace of the parse.

**Figure 2.7:** Bottom-up parse of \( id+id*id \)

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>RULE/ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id+id*id]</td>
<td>Shift</td>
</tr>
<tr>
<td>id]</td>
<td>+id*id]</td>
<td>Reduce using ( E \rightarrow id )</td>
</tr>
<tr>
<td>E]</td>
<td>+id*id]</td>
<td>Shift</td>
</tr>
<tr>
<td>+E]</td>
<td>id*id]</td>
<td>Shift</td>
</tr>
<tr>
<td>id+E]</td>
<td>*id]</td>
<td>Reduce using ( E \rightarrow id )</td>
</tr>
<tr>
<td>E+E]</td>
<td>*id]</td>
<td>Shift</td>
</tr>
<tr>
<td>*E+E]</td>
<td>id]</td>
<td>Shift</td>
</tr>
<tr>
<td>id*E+E]</td>
<td>}</td>
<td>Reduce using ( E \rightarrow id )</td>
</tr>
<tr>
<td>E*E+E]</td>
<td>}</td>
<td>Reduce using ( E \rightarrow E*E )</td>
</tr>
<tr>
<td>E+E]</td>
<td>}</td>
<td>Reduce using ( E \rightarrow E+E )</td>
</tr>
<tr>
<td>E]</td>
<td>}</td>
<td>Accept</td>
</tr>
</tbody>
</table>

The trace shows the contents of the stack and the remaining input at each step of the parse.

In these examples the choice of the which production to use may appear to be magical. In the case of a top-down parser, grammar \( G_1 \) should be rewritten to remove the ambiguity. For bottom up parsers, there are techniques for the analysis of the grammar to produce a set of unambiguous choices for productions. Such techniques are beyond the scope of this text.

**Nondeterministic pushdown automata**
A careful study of the parsing process reveals that whether the parse is top-down or bottom-up, the parser must hold some information on a stack. In the case of a recursive descent parser, the stack is implicit in the recursion. In the case of the top-down parser, it must pop variables off the stack and push the corresponding right-hand side on the stack and pop terminals off the stack when they match the input. In the case of the bottom-up parser, it must shift (push) terminals onto the stack from the input and reduce (pop) sequences of terminals and variables off the stack replacing them with a variable where the sequence of terminals and variables correspond to the right-hand side of some production.

This observation leads us to the notion of push-down automata. A push-down automata has an input that it scans from left to right, a stack, and a finite control to control the operations of reading the input and pushing data on and popping data off the stack. Definition 2.6 is a formal definition of a push-down automata.

**Definition 2.6: Push-down automaton**

A push-down automaton $M$ is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

- $Q$ is a finite set of states
- $\Sigma$ is a finite alphabet called the input alphabet
- $\Gamma$ is a finite alphabet called the stack alphabet
- $\delta$ is a transition function from $Q \times (\Sigma \cup \{e\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$
- $q_0$ in $Q$ is the initial state
- $Z_0$ in $\Gamma$ is called the start symbol
- $F$ a subset of $Q$; the set of accepting states

**PDA** = < States, StartState, FinalStates, InputAlphabet, Input, StackAlphabet, Stack, TransitionFunction, >
Configuration: $C = \text{State} \times \text{Stack} \times \text{Input}$; initial configuration $(\text{StartState}, [], \text{Input})$

$t : C --> C$

*Allowed transitions*
Syntax

\[
t(s, [], []) \rightarrow \text{accept (empty stack)}
\]
\[
t(s, [], S) \rightarrow \text{accept } s \text{ in FinalStates}
\]
\[
t(s, I, S) = (s', I, S) \rightarrow \text{epsilon move}
\]
\[
t(s, [i|I], S) = (s', I, S) \rightarrow \text{consume input}
\]
\[
t(s, I, [x|S]) = (s', I, S) \rightarrow \text{pop stack}
\]
\[
t(s, I, S) = (s', I, [x|S]) \rightarrow \text{push stack}
\]
\[
t(s, [i|I], [x|S]) = (s', I, S) \rightarrow \text{consume input and pop stack}
\]
\[
t(s, [i|I], S) = (s', I, [x|S]) \rightarrow \text{consume input and push stack}
\]

Example: palindroms program (StartState, Input, [])

\[
t(\text{push}, [], []) = \text{accept} \quad \text{// empty input}
\]
\[
t(\text{push}, [x|I], S) = (\text{pop}, I, S) \quad \text{// center, odd length palindrom}
\]
\[
t(\text{push}, [x|I], S) = (\text{pop}, I, [x|S]) \quad \text{// center, even length palindrom}
\]
\[
t(\text{push}, [x|I], S) = (\text{push}, I, [x|S]) \quad \text{// left side}
\]
\[
t(\text{pop}, [x|I], [x|S]) = (\text{pop}, I, S) \quad \text{// right side}
\]
\[
t(\text{pop}, [], []) = \text{accept}
\]

Applications of PDAs are explored in the exercises.

**Equivalence of PDA and CFGs**

Just as a grammar defines a language \(L(G)\), so a PDA M defines a language \(L(M)\), the set of strings that it accepts. The relationship between PDAs and CFG is interesting. Any language accepted by a PDA can be shown to be shown to have a context-free grammar. Also any context-free grammar defines a PDA. While the proof of these statements is beyond the scope of this text, the idea of the proof is this. The configurations of a PDA can be described in terms of a context-free grammar. All CFGs can be put into a special form (Greibach normal form) which can be used to describe the configurations of a PDA.

**Regular Expressions**

While CFGs can be used to describe the tokens of a programming languages, regular expressions (RE) are a more convenient notation for describing their simple structure. The alphabet consists of the character set chosen for the language and the notation includes

- `\cdot` to concatenate items (juxtaposition is used for the same purpose),
- `\|` to separate alternatives (often `+` is used for the same purpose),
- `\ast` to indicate that the previous item may be repeated zero or more times, and
- `( ` and `)` for grouping.
**Definition 2.7:** Regular expressions and Regular languages

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Language Denoted L(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td><em>lambda</em></td>
<td>{lambda}</td>
</tr>
<tr>
<td>a</td>
<td>{ a }</td>
</tr>
<tr>
<td>(E · F)</td>
<td>{uv</td>
</tr>
<tr>
<td>(E</td>
<td>F)</td>
</tr>
<tr>
<td>(E*)</td>
<td>{u_1u_2...u_n</td>
</tr>
</tbody>
</table>

Identifiers and real numbers may be defined using regular expressions as follows:

```
integer = D D*
identifier = A(A|D)*
```

A scanner is a program which groups the characters of an input stream into a sequence of tokens. Scanners based on regular expressions are easy to write. Lex is a scanner generator often packaged with the UNIX environment. A user creates a file containing regular expressions and Lex creates a program to search for tokens defined by those regular expressions. Text editors use regular expressions to search for and replace text. The UNIX grep command uses regular expressions to search for text.

**Deterministic and nondeterministic Finite State Machines**

Regular expressions are equivalent to *finite state machines*. A finite state machine consists of a set of states (one of which is a start state and one or more which are accepting states), a set of transitions from one state to another each labeled with an input symbol, and an input string. Each step of the finite state machine consists of comparing the current input symbol with the set of transitions corresponding to the current state and then consuming the input symbol and moving to the state corresponding to the selected
Definition 2.8: Finite State Automaton

A finite state automaton or fsa is defined by the quintuple

\[ M = (Q, \Sigma, \delta, q_0, F), \]

where

- **Q** is a finite set of *internal states*
- **\Sigma** is a finite set of symbols called the *input alphabet*
- **\delta**: \( Q \times \Sigma \rightarrow 2^Q \) is a total function called the *transition function*
- **q_0** in **Q** is the *initial state*
- **F** a subset of **Q** is the set of *final states*

**FSM** = <States, StartState, FinalStates, InputAlphabet, Input, TransitionFunction>

Configuration: \( C = \text{State} \times \text{Input} \); initial configuration (StartState, Input)

\[ t : C \rightarrow C \]

**Allowed transitions**

- \( t(s, []) = \text{accept } s \text{ in FinalStates} \)
- \( t(s, [x|I]) = (s', I) \) -- consume input
- \( t(s, I) = (s', I) \) -- epsilon move

Example: identifiers (StartState, Input)

\[
\begin{align*}
  t(\text{start}, [i|I]) &= (\text{ad}, I) \\
  t(\text{ad}, [i|I]) &= (\text{ad}, I) \\
  t(\text{ad}, [d|I]) &= (\text{ad}, I) \\
  t(\text{ad}, []) &= \text{accept}
\end{align*}
\]

The transition function \( \delta \) is defined on a state and an input symbol. It can be extended to a function \( \delta^* \) on strings of input symbols as follows:

1. \( \delta^*(q, -) = q \) for the empty string
2. \( \delta^*(q, wa) = \delta(\delta^*(q, w), a) \) for all strings \( w \) and input symbols \( a \)
A FSA is called \textit{deterministic} if there is at most one transition from one state to another for a given input and there are no \textit{lambda} transitions. A FSA is called \textit{nondeterministic} if there is one or more transitions from one state to another for a given input. A Moore machine is a FSA which associates an output with each state and a Mealy machine is a FSA which associates an output with each transition. The Moore and Mealy FSAs are important in applications of FSAs.

\section*{Equivalence of deterministic and nondeterministic FSA}

It might seem that a machine that could `guess' (nondeterministic) which move to make next would be more powerful than one that could not (deterministic). The following theorems show that in the case of FSAs, it is not the case.

\textbf{Theorem:} Every deterministic finite state automaton is a nondeterministic finite state automaton.
\textit{Proof:} The definition of a deterministic FSA is included in the definition of a nondeterministic FSA.

\textbf{Theorem:} For every nondeterministic finite state automaton there is a deterministic finite state automaton that accepts the same language.
\textit{Proof:}

Let $M=(S,A,t,q_0,F)$ be a nondeterministic FSA. Define $M'=(S',A,t',F')$ as follows:

- $S'$ is the set of all subsets of $S$; an element of $S'$ is denoted by $[q_1,...,q_m]$.
- $t'$: $t'([q_1,...,q_m],a) = [p_1,...,p_n]$ where $[p_1,...,p_n]$ is the union of the states of $S$ such that $t(q_i,a) = p_j$.
- $F'$ is the set of all states of $S'$ that contain an accepting state of $M$.

The proof is completed by induction on the length of the input string.

\section*{Equivalence of FSA and regular expressions}

Just as context-free grammars and PDAs are equivalent, so regular expressions and FSAs are equivalent as the following theorems show.

\textbf{Theorem:} If $r$ is a regular expression then there exists an NFA that accepts $L(r)$.
\textit{Proof (Sketch)} The formal proof is by induction on the number of operators in the expression. For the base cases (empty set, empty string and singleton set) the corresponding FSA is simple to define. The more complex cases are handled by merging states of simpler FSA.

\textbf{Theorem:} If $L$ is accepted by a DFA then $L$ is denoted by a regular expression.
\textit{Proof} beyond the scope of this text.
Graphical Representation

In a graphical representation, states are represented by circles, with final (or accepting) states indicated by two concentric circles. The start state is indicated by the word "Start". An arc from state \( s \) to state \( t \) labeled \( a \) indicates a transition from \( s \) to \( t \) on input \( a \). A label \( a/b \) indicates that this transition produces an output \( b \). A label \( a_1, a_2, \ldots, a_k \) indicates that the transition is made on any of the inputs \( a_1, a_2, \ldots, a_k \).

/* NEED A NICE DIAGRAM HERE */

Tabular Representation

In a tabular representation, states are one index and inputs the other. The entries in the table are the next state (and actions if required).

![Figure 2.8: Tabular representation for a transition function.](http://cs.wwc.edu/~aabyan/PLBook/HTML/Syntax.html)

| state/input | \( I_0 \) | \( \ldots \) | \( I_n \) |
|-------------|----------|-------------|
| \( S_0 \)   | \( S_{00} \) | \( \ldots \) | \( S_{0n} \) |
| \( \ldots \) | \( \ldots \) | \( \ldots \) | \( \ldots \) |
| \( S_m \)   | \( S_{m0} \) | \( \ldots \) | \( S_{mn} \) |

Implementation of FSAs

The transition function of a FSA can be implemented as a case statement, a collection of procedures and as a table. In a case based representation state is represented by the value of a variable, the case statement is placed in the body of a loop and on each iteration of the loop, the input is read and the state variable updated.

```
State := Start;
repeat
    get input I
    case State of
        ...
```
Si : case I of
    ...
    Ci : State := Sj;
    ...
end
...
end
until empty input and accepting state

In a procedural representation, each state is a procedure. Transitions to the next state occur when the
procedure representing the next state is "called".

procedure StateS(I : input)
    case I of
        ...
        Ci : get input I; StateT(I)
        ...
end

In the table-driven implementation, the transition function is encoded in a two dimensional array. One
index is the current state the other is the current input. The array element are states.

state := start;
while state != final do
    get input I;
    state := table[state,I]

The implementations are incomplete since they do not contain code to deal with the end of input.

Pragmatics

At the semantics level, concrete syntax does not matter. However, concrete syntax does matter to the
programmer and to the compiler writer. The programmer needs a language that is easy to read and write.
The compiler writer wants a language that is easy to parse. Simple details such as placement of
keywords, semicolons and case can complicate the life of the programmer or compiler writer.

Many languages are designed to designed to make compilation easy. The goal is to provide a syntax so
that the compiler need make only one pass over the program. This requirement means that with minor
exceptions, each constant, type, variable, procedure and function must be defined before it is referenced.
The trade-off is between slightly increased syntactic complexity of the language with some increased in
the burden on the programmer and a simpler compiler.
Some specific syntactical issues include:

- **Statement termination and/or separation.** In Pascal the semicolon is a statement separator while in C the semicolon is a statement terminator. Thus in Pascal a semicolon is not necessary after the last statement in a sequence of statements while it is required in C. If a language includes an empty statement, a misplaced semicolon can change the meaning of a program. For example, in the program fragment

  
  ```plaintext
  while C do; S;
  ```

  the first semicolon terminates the empty statement following the `do` and the while statement; `S` is not in the body of the while statement.

- **Case sensitivity.** Pascal is case insensitive while C is case sensitive.

- **Opening and closing keywords.** Algol-68 and Modula-2 require closing keywords. Modula-2 uses `end` while Algol-68 uses the reverse of the opening keyword for example,

  ```plaintext
  if C then S fi
  ```

- **The assignment operator.** The assignment operator varies among imperative programming languages.

  ```plaintext
  := Pascal and Ada
  = FORTRAN, C/C++/Java
  <-- APL
  ```

  The choice in FORTRAN and C/C++/Java is unfortunate since assignment is different from equality.

  ```plaintext
  .EQ. FORTRAN
  == C/C++/Java
  ```

- **Identification of function and procedure calls.** In C, procedure calls are distinguished by the presence of parentheses. Pascal does not require parentheses.

- **Return values.** In C, if a function is used as a command, its return value is ignored. In Pascal, a function cannot be used as a command. To ignore the returned value, Modula-3 requires the function call with the keyword `EVAL`.

A syntax directed editor can use color, font, and layout to assist the programmer in distinguishing between comments, reserved words, code, and can provide command completion.
Historical Perspectives and Further Reading

Backus-Naur Form

The BNF is a notation for describing the productions of a context-free grammar. The BNF uses the following symbols <, >, ::=, |. Variables are enclosed between < and >. The symbol --> is replaced with :: =. The symbol | is used to separate alternatives. Terminals are represented by themselves or are written in a type face different from the symbols of the BNF. The following is a BNF description of arithmetic expressions.

\[
\begin{align*}
\langle Expression \rangle & ::= \langle Identifier \rangle \mid \langle Number \rangle \\
& \mid \\
\langle Expression \rangle \ \langle Op \rangle \ \langle Expression \rangle \\
& \mid ( \langle Expression \rangle ) \\
\langle Op \rangle & ::= + \mid - \mid * \mid / \\
\langle Identifier \rangle & ::= \langle Letter \rangle \\
& \mid \langle Identifier \rangle \ \langle Letter \rangle \\
\langle Number \rangle & ::= \langle Digit \rangle \\
& \mid \langle Number \rangle \ \langle Digit \rangle \\
\langle Letter \rangle & ::= A \mid \ldots \mid Z \\
\langle Digit \rangle & ::= 0 \mid \ldots \mid 9
\end{align*}
\]

EBNF (extended BNF)

Several extensions to improve the readability of the BNF have been suggested. One such extension is to write the names of the variables in italics and without < and >. In addition, the EBNF (extended BNF) is a combination of the BNF and the notation of regular expressions. An EBNF production rule is of the form \( N ::= E \), where \( N \) is a nonterminal symbol and \( E \) is an extended regular expression. Like ordinary regular expressions, \( E \) may contain `|', `*', and parentheses for grouping but unlike ordinary regular expressions, it may contain variable symbols as well as terminal symbols.

Some additional extensions include the use of braces, \{E\}, or ellipses, E..., to indicate zero or more repetitions of an item and brackets, [E], to indicate an optional item.

Figure 2.8 contains a context-free grammar for a simple imperative programming language.

**Figure 2.8:** Context-free grammar for Simple

\[
\text{program ::= LET definitions IN command_sequence END}
\]


Syntax

\[
\text{syntax:} \quad \text{definitions ::= } e \mid \text{INTEGER id_seq IDENTIFIER .}
\]

\[
\text{id_seq ::= } e \mid \text{id_seq IDENTIFIER ,}
\]

\[
\text{command_sequence ::= } e \mid \text{command_sequence command ;}
\]

\[
\text{command ::= SKIP}
\]
\[
\quad \mid \text{READ IDENTIFIER}
\]
\[
\quad \mid \text{WRITE exp}
\]
\[
\quad \mid \text{IDENTIFIER ::= exp}
\]
\[
\quad \mid \text{IF exp THEN command_sequence ELSE command_sequence FI}
\]

\[
\text{command_sequence FI}
\]
\[
\quad \mid \text{WHILE bool_exp DO command_sequence END}
\]

\[
\text{exp ::= exp + term | exp - term | term}
\]

\[
\text{term ::= term * factor | term / factor | factor}
\]

\[
\text{factor ::= factor^primary | primary}
\]

\[
\text{primary ::= NUMBER | IDENT | ( exp )}
\]

\[
\text{bool_exp ::= exp = exp | exp < exp | exp > exp}
\]

---

Syntax

Slonneger & Kurts (1995)

*Formal Syntax and Semantics of Programming Languages* Addison Wesley

Watt, David A. (1991)

*Programming Language Syntax and Semantics* Prentice-Hall International.

Language Descriptions

It is instructive to read official language descriptions. The following are listed in historical order.

FORTRAN


LISP


ALGOL 60

Naur, P., ed (1963) *Revised Report on the Algorithmic Language ALGOL 60* Communications of
the ACM. 6, 1-17.

Algo 68
Pascal

Ada

C

C++
Java 1.02

Scheme
Haskell 1.3
Peterson, John., ed (1996)a *The Haskell Report 1.3*

Prolog
Gödel

**Parser (Compiler) Construction Tools**

Lex & Yacc (or Flex/Bison)
Eli Compiler Construction System
Purdue Compiler-Construction Tool Set (PCCTS)
Watt, David A. (1993)

*Programming Language Processors* Prentice-Hall International

JACK (Java parser and scanner construction tool)

**Formal Languages and Automata**

\[ TM = \langle States, InputAlphabet, TransitionFunction, FinalStates, StartState \rangle \]
Configuration: \( C = State \times Input \); initial configuration (StartState, Input)

\[ t : C \rightarrow C \]

Allowed transitions

\[ t(s, I) \rightarrow \text{accept s in FinalStates} \]
\[ t(s, I) = (s', I) \rightarrow \text{epsilon move} \]
\[ t(s, (\text{Left}, x, [y|\text{Right}])) = (s', ([x'|\text{Left}], y, \text{Right})) \rightarrow \text{move right} \]
\[ t(s, ([x|\text{Left}], y, \text{Right})) = (s', (\text{Left}, x, [y'|\text{Right}])) \rightarrow \text{move left} \]

For regular expressions and their relationship to finite automata and context-free grammars and their relationship to push-down automata see texts on formal languages and automata such as \cite{HU79}.
The original paper on attribute grammars was by Knuth\cite{Knuth68}. For a more recent source and their use in compiler construction and compiler generators see \cite{DJL88,PittPet92}

Hopcroft and Ullman (1979)

*Introduction to Automata Theory, Languages, and Computation* Addison-Wesley

Linz, Peter (1996)

*An Introduction to Formal Languages and Automata* D. C. Heath and Company

**Exercises**

1. [time/difficulty](cfg) What is the size of $L_0L_1$?
2. (cfg) Is $L_0L_1 = L_1L_0$?
3. (cfg) Show that the grammar $G_1$ is ambiguous by producing two distinct derivation trees for the sentence: $E + E * E$.
4. (cfg) Define a grammar for the if-then and if-then-else control structures. Is your grammar is ambiguous? Hint: try producing two distinct derivation trees for the sentence: if $C$ then if $C$ then $S$ else $S$.
5. (cfg, bnf, ebnf) Discuss the advantages and disadvantages of the following grammars for the if-then-else statements. Hint: consider the grammars from both the user and parser perspectives.
   a. $stmt --> begin stmts end$
   
   $stmt --> if expr then stmt$
   $stmt --> if expr then stmt$ else $stmt$
   
   $stmt --> if expr then stmts endif$
   
   $stmt --> if expr then stmts$ else $stmt$ sendif
   
   $stmt --> if expr then stmts$ else $stmts$ endif
   
   6. (cfg) Does the order in which production rules are applied matter? Can they be applied in an arbitrary order including in parallel or in some random order?
7. (cfg) Can a fully abstract grammar be ambiguous?
8. (parse) In a top-down parse, what is required of the grammar so that the parser will be able to pick the correct production?
9. (parse) In a bottom-up parse, ...
10. (parse) Construct a recursive descent parser for $G_0$, the grammar for a fragment of English (see figure 2.1).
11. (pda) Construct a PDA which checks for matching parentheses.
12. (pda) Construct a PDA which recognizes palindromes.
13. (pda) Construct a PDA which translates arithmetic expressions from infix to post-fix.
14. (pda) Show that a PDA can recognize the language $a^n b^n$.
15. (pda) Show that a PDA cannot recognize the language $a^n b^n c^n$.
16. (re) Define binary numbers using regular expressions.
17. (re) Define real numbers using regular expressions.
18. (re) Construct a scanner to recognize identifiers, numbers and arithmetic operators.
19. (re, parse) Using the following grammar for expressions:

\[
\begin{align*}
\text{exp} &::= \text{term exp}' \\
\text{exp}' &::= + \text{term exp}' | - \text{term exp}' | \text{epsilon} \\
\text{term} &::= \text{factor term}' \\
\text{term}' &::= \times \text{factor term}' | \div \text{factor term}' | \text{epsilon} \\
\text{factor} &::= \text{primary factor}' \\
\text{factor}' &::= ^\text{primary factor}' | \text{epsilon} \\
\text{primary} &::= \text{INT | IDENT | ( exp)}
\end{align*}
\]

a. Construct a trace of a top down parse for the expression \text{id+id*id}.

b. Construct a scanner and a recursive descent parser for the grammar.

20. (re, parse) Construct a scanner and a parser for the programming language Simple

21. (pragmatics) Discuss the advantages and disadvantages of Pascal or C style function calls (C requires empty parentheses for parameterless functions while Pascal does not).

22. (pragmatics) Discuss the advantages and disadvantages of case sensitivity for the programmer and compiler writer.

23. (pragmatics) Discuss the consequences of the number of reserved words in a programming language.

24. (pragmatics) Discuss the necessity separators and terminators.

25. (pragmatics) Discuss the advantages and disadvantages of requiring declarations before references for the compiler writer and for the programmer.
Semantics

The semantics of a programming language describe the relationship between the syntax and the model of computation.

Keywords and phrases: Algebraic semantics, axiomatic semantics, denotational semantics, operational semantics, semantic algebra, semantic axiom, semantic domain, semantic equation, semantic function, loop variant, loop invariant, valuation function, sort, signature, many-sorted algebra

Semantics is concerned with the interpretation or understanding of programs and how to predict the outcome of program execution. The semantics of a programming language describe the relation between the syntax and the model of computation. Semantics can be thought of as a function which maps syntactical constructs to the computational model.

semantics: syntax --> computational model

This approach is called syntax-directed semantics.

There are several widely used techniques (algebraic, axiomatic, denotational, operational, and translation) for the description of the semantics of programming languages.

- **Algebraic semantics** describe the meaning of a program by defining an algebra. The algebraic relationships and operations are described by axioms and equations.
- **Axiomatic semantics** defines the meaning of the program implicitly. It makes assertions about relationships that hold at each point in the execution of the program. Axioms define the properties of the control structures and state the properties that may be inferred. A property about a program is deduced by using the axioms. Each program has a pre-condition which describes the initial conditions required by the program prior to execution and a post-condition which describes, upon termination of the program, the desired program property.
- **Denotational semantics** tell what is computed by giving a mathematical object (typically a function) which is the meaning of the program. Denotational semantics are used in comparative studies of programming languages.
- **Operational semantics** tell how a computation is performed by defining how to simulate the execution of the program. Operational semantics may describe the syntactic transformations which mimic the execution of the program on an abstract machine or define a translation of the program into recursive functions. Operational semantics are used when learning a programming language and by compiler writers.
Translation semantics describe how to translate a program into another language usually the language of a machine. Translation semantics are used in compilers.

Much of the work in the semantics of programming languages is motivated by the problems encountered in trying to construct and understand imperative programs---programs with assignment commands. Since the assignment command reassigns values to variables, the assignment can have unexpected effects in distant portions of the program.

**Algebraic Semantics**

An algebraic definition of a language is a definition of an algebra. An algebra consists of a domain of values and a set of operations (functions) defined on the domain.

\[ \text{Algebra} = \langle \text{set of values; operations} \rangle \]

Figure N.1 contains an example of an algebraic definition. It is an algebraic definition of a fragment of Peano arithmetic.

---

**Figure N.1: Algebraic Definition of Peano Arithmetic**

**Domains:**

- \( \text{Bool} = \{ \text{true, false} \} \) (Boolean values)
- \( \text{N in Nat} \) (the natural numbers)
- \( \text{N ::= 0 | S(N)} \)

**Functions:**

- \( = : (\text{Nat, Nat}) \rightarrow \text{Bool} \)
- \( + : (\text{Nat, Nat}) \rightarrow \text{Nat} \)
- \( \times : (\text{Nat, Nat}) \rightarrow \text{Nat} \)

**Axioms and equations:**

- \( \text{not S(N)} = 0 \)
- \( \text{if S(M) = S(N) then M = N} \)
- \( (n + 0) = n \)
- \( (m + S(n)) = S(m + n) \)
- \( (n \times 0) = 0 \)
- \( (m \times S(n)) = ((m \times n) + m) \)
where \( m, n \) in Nat

The equations define equivalences between syntactic elements; they specify the transformations that are used to translate from one syntactic form to another. The domain is often called a sort and the domain and the function sections constitute the signature of the algebra. Functions with zero, one, and two operands are referred to as nullary, unary, and binary operations. Because there is more than one domain, the algebra is called a many sorted algebra. As in this example, abstract data types may require values from several different sorts. The signature of the algebra is a set of sorts and a set of functions taking arguments and returning values of different sorts. A stack of natural numbers may be modeled as a many-sorted algebra with three sorts (natural numbers, stacks and booleans) and four operations (newStack, push, pop, top, empty). Figure N.2 contains an algebraic definition of a stack.

**Figure N.2: Algebraic definition of an Integer Stack ADT**

Domains:

Nat (the natural numbers)
Stack (of natural numbers)
Bool (boolean values)

Functions:

newStack: () -> Stack
push : (Nat, Stack) -> Stack
pop: Stack -> Stack
top: Stack -> Nat
empty : Stack -> Bool

Axioms: or

Defining Equations:

pop(push(N,S)) = S
pop(push(N,S)) = N
empty(push(N,S)) = false
empty(newStack()) = true

Errors:

newStack() = []
push(N,S) = [N|S]
pop([N|S]) = S
top([N|S]) = N
In Figure N.1, the structure of the numbers is described. In Figure N.2 the structure of a stack is not defined. This means that we cannot use equations to describe syntactic transformations. Instead, we use axioms that describe the relationships between the operations. The axioms are more abstract than equations because the results of the operations are not described. To be more specific would require decisions to be made concerning the implementation of the stack data structure. Decisions which would tend to obscure the algebraic properties of stacks. The axioms impose constraints on the stack operations that are sound in the sense that they are consistent with the actual behavior of stacks regardless of the implementation. Finding axioms that are complete, in the sense that they completely specify the behavior of the operations of an ADT, is more difficult.

The goal of algebraic semantics is to capture the semantics of behavior by a set of axioms with purely syntactic properties. Algebraic definitions (semantic algebras) are the favored method for defining the properties of abstract data types.

### Axiomatic Semantics

The axiomatic semantics of a programming language are the assertions about relationships that remain the same each time the program executes. Axiomatic semantics are defined for each control structure and command. The axiomatic semantics of a programming language define a mathematical theory of programs written in the language. A mathematical theory has three components.

- **Syntactic rules**: These determine the structure of formulas which are the statements of interest.
- **Axioms**: These describe the basic properties of the system.
- **Inference rules**: These are the mechanisms for deducing new theorems from axioms and other theorems.

The semantic formulas are triples of the form:

\[ \{P\} \text{ } c \{Q\} \]

where \( c \) is a command or control structure in the programming language, \( P \) and \( Q \) are assertions or statements concerning the properties of program objects (often program variables) which may be true or false. \( P \) is called a pre-condition and \( Q \) is called a post-condition. The pre- and post-conditions are formulas in some arbitrary logic and summarize the progress of the computation.

The meaning of
{P} \ c \ {Q}

is that if \ c \ is executed in a state in which assertion P is satisfied and \ c \ terminates, then \ c \ terminates in a state in which assertion Q is satisfied. We illustrate axiomatic semantics with a program to compute the sum of the elements of an array (see Figure N.3).

**Figure N.3:** Program to compute \( S = \sum_{i=1}^{n} A[i] \)

```plaintext
S, I := 0, 0
while I < n do
    S, I := S + A[I+1], I+1
end
```

The assignment statements are *simultaneous* assignment statements. The expressions on the righthand side are evaluated simultaneously and assigned to the variables on the lefthand side in the order they appear.

Figure N.4 illustrates the use of axiomatic semantics to verify the program of Figure N.3.

**Figure N.4:** Verification of \( S = \sum_{i=1}^{n} A[i] \)

**Pre/Post-conditions**

1. \( \{ 0 = \sum_{i=1}^{0} A[i], 0 < |A| = n \} \)
2. \( \{ S = \sum_{i=1}^{I} A[i], I <= n \} \)
3. \( \{ S = \sum_{i=1}^{I} A[i], I < n \} \)
4. \( \{ S = \sum_{i=1}^{I+1} A[i], I+1 <= n \} \)
5. \( \{ S = \sum_{i=1}^{I+1} A[i], I+1 <= n \} \)
6. \( \{ S = \sum_{i=1}^{I+1} A[i], I+1 <= n \} \)
7. \( \{ S = \sum_{i=1}^{I+1} A[i], I+1 <= n \} \)
8. \( \{ S = \sum_{i=1}^{I+1} A[i], I+1 <= n \} \)
9. \( \text{end} \)
The program sums the values stored in an array and the program is decorated with the assertions which help to verify the correctness of the code. The pre-condition in line 1 and the post-condition in line 11 are the pre- and post-conditions respectively for the program. The pre-condition asserts that the array contains at least one element zero and that the sum of the first zero elements of an array is zero. The post-condition asserts that $S$ is sum of the values stored in the array. After the first assignment we know that the partial sum is the sum of the first $I$ elements of the array and that $I$ is less than or equal to the number of elements in the array.

The only way into the body of the while command is if the number of elements summed is less than the number of elements in the array. When this is the case, The sum of the first $I+1$ elements of the array is equal to the sum of the first $I$ elements plus the $I+1$st element and $I+1$ is less than or equal to $n$. After the assignment in the body of the loop, the loop entry assertion holds once more. Upon termination of the loop, the loop index is equal to $n$. To show that the program is correct, we must show that the assertions satisfy some verification scheme. To verify the assignment commands, we use the Assignment Axiom:

**Assignment Axiom**

\[
\{P[x:E]\} \ x := E \ \{P\}
\]

This axiom asserts that:

If after the execution of the assignment command the environment satisfies the condition $P$, then the environment prior to the execution of the assignment command also satisfies the condition $P$ but with $E$ substituted for $x$ (In this and the following axioms we assume that the evaluation of expressions does not produce side effects.).

An examination of the respective pre- and post-conditions for the assignment statements shows that the axiom is satisfied.

To verify the while command of lines 4, 7 and 9, we use the Loop Axiom:

**Loop Axiom:**

\[
\{I \land B \land V > 0 \} \ C \ \{I \land V > V' >= 0\}
\]

\[
\{I\} \text{ while } B \text{ do } C \text{ end } \{I \land \neg B\}
\]
The assertion above the bar is the condition that must be met before the axiom (below the bar) can hold. In this rule, \( \{I\} \) is called the loop invariant. This axiom asserts that:

To verify a loop, there must be a loop invariant \( I \) which is part of both the pre- and post-conditions of the body of the loop and the conditional expression of the loop must be true to execute the body of the loop and false upon exit from the loop.

The invariant for the loop is: \( S = \sum_{i=1}^{I} A[i] \), \( I \leq n \). Lines 6, 7, and 8 satisfy the condition for the application of the Loop Axiom. To prove termination requires the existence of a loop variant. The loop variant is an expression whose value is a natural number and whose value is decreased on each iteration of the loop. The loop variant provides an upper bound on the number of iterations of the loop.

A variant for a loop is a natural number valued expression \( V \) whose run-time values satisfy the following two conditions:

- The value of \( V \) greater than zero prior to each execution of the body of the loop.
- The execution of the body of the loop decreases the value of \( V \) by at least one.

The loop variant for this example is the expression \( n - I \). That it is non-negative is guaranteed by the loop continuation condition and its value is decreased by one in the assignment command found on line 7. More general loop variants may be used; loop variants may be expressions in any well-founded set (every decreasing sequence is finite). However, there is no loss in generality in requiring the variant expression to be an integer. Recursion is handled much like loops in that there must be an invariant and a variant. The correctness requirement for loops is stated in the following:

**Loop Correctness Principle:** Each loop must have both an invariant and a variant.

Lines 5 and 6 and lines 10 and 11 are justified by the Rule of Consequence.

**Rule of Consequence:**

\[
\begin{align*}
P \rightarrow Q, \ & \{Q\} \ C \{R\}, \ R \rightarrow S \\
\hline
\{P\} \ C \{S\}
\end{align*}
\]

The justification for the composition the assignment command in line 2 and the while command requires the following the Sequential Composition Axiom.

**Sequential Composition Axiom:**

\[
\begin{align*}
\{P\} \ C0 \{Q\}, \ & \{Q\} \ C1 \{R\} \\
\hline
\{P\} \ C0; C1 \{R\}
\end{align*}
\]

This axiom is read as follows:
The sequential composition of two commands is permitted when the post-condition of the first command is the pre-condition of the second command.

The following rules are required to complete the deductive system.

**Selection Axiom:**
\[ \{P \land B\} C_0 \{Q\}, \{P \land \neg B\} C_1 \{Q\} \]
\[ \{P\} \text{ if } B \text{ then } C_0 \text{ else } C_1 \text{ fi } \{Q\} \]

**Conjunction Axiom:**
\[ \{P\} C \{Q\}, \{P'\} C \{Q'\} \]
\[ \{P \land P'\} C \{Q \land Q'\} \]

**Disjunction Axiom:**
\[ \{P\} C \{Q\}, \{P'\} C \{Q'\} \]
\[ \{P \lor P'\} C \{Q \lor Q'\} \]

The axiomatic method is the most abstract of the semantic methods and yet, from the programmer's point of view, the most practical method. It is most abstract in that it does not try to determine the meaning of a program, but only what may be proved about the program. This makes it the most practical since the programmer is concerned with things like, whether the program will terminate and what kind of values will be computed.

Axiomatics semantics are appropriate for program verification and program derivation.

**Assertions for program construction**

The axiomatic techniques may be applied to the construction of software. Rather than proving the correctness of an existing program, the proof is integrated with the program construction process to insure correctness from the start. As the program and proof are developed together, the assertions themselves may provide suggestions which facilitate program construction.

Loops and recursion are two constructs that require invention on the part of the programmer. The loop correctness principle requires the programmer to come up with both a variant and an invariant. Recursion is a generalization of loops so proofs of correctness for recursive programs also require a loop variant and a loop invariant. In the summation example, a loop variant is readily apparent from an examination of the post-condition. Simply replace the summation upper limit, which is a constant, with a variable. Initializing the sum and index to zero establishes the invariant. Once the invariant is established, either the index is equal to the upper limit in which case there sum has been computed or the next value must be added to the sum and the index incremented reestablishing the loop invariant. The position of the loop invariants define a loop body and the second occurrence suggests a recursive call. A
recursive version of the summation program is given in Figure N.5.

**Figure N.5: Recursive version of summation**

\[
S, I := 0, 0
\]

\[
\text{loop: if } I < n \text{ then } S, I := S + A[I+1], I+1; \text{ loop}
\]

\[
\text{else skip fi}
\]

The advantage of using recursion is that the loop variant and invariant may be developed separately. First develop the invariant then the variant.

The summation program is developed from the post-condition by replacing a constant by a variable. The initialization assigns some trivial value to the variable to establish the invariant and each iteration of the loop moves the variable's value closer to the constant.

A program to perform integer division by repeated subtraction can be developed from the post-condition \{ 0 <= r < d, (a = q \times d + r) \} by deleting a conjunct. In this case the invariant is \{ 0 <= r, (a = q \times d + r) \} and is established by setting the the quotient to zero and the remainder to a.

Another technique is called for in the construction of programs with multiple loops. For example, the post condition of a sorting program might be specified as:

\[
\{ \text{for all } i. (0 < i < n \rightarrow A[i] <= A[i+1]), s = \text{perm}(A) \}
\]

or the post condition of an array search routine might be specifies as:

\[
\{ \text{if exists } i. (0 < i <= n \text{ and } t = A[i]) \text{ then location } = i \text{ else location } = 0 \}
\]

To develop an invariant in these cases requires that the assertion be strengthened by adding additional constraints. The additional constraints make assertions about different parts of the array.

**Denotational Semantics**

A *denotational definition* of a language consists of three parts: the abstract syntax of the language, a semantic algebra defining a computational model, and valuation functions. The valuation functions map the syntactic constructs of the language to the semantic algebra. Recursion and iteration are defined using the notion of a limit. the programming language constructs are in the *syntactic domain* while the mathematical entity is in the *semantic domain* and the mapping between the various domains is provided.
by valuation functions. Denotational semantics relies on defining an object in terms of its constituent parts. The Figure N.6 is an example of a denotational definition.

**Figure N.6:** Denotational definition of Peano Arithmetic

Abstract Syntax:

N in Nat (the Natural Numbers)
N ::= 0 | S(N) | (N + N) | (N × N)

Semantic Algebra:

Nat (the natural numbers (0, 1, ...)
+ : Nat -> Nat -> Nat

Valuation Function:

\( D : \text{Nat} \to \text{Nat} \)

\[
\begin{align*}
D[(n + 0)] &= D[n] \\
D[(m + S(n))] &= D[(m+n)] + 1 \\
D[(n × 0)] &= 0 \\
D[(m × S(n))] &= D[(m × n) + m] \\
\end{align*}
\]

where m,n in Nat

It is is a denotational definition of a fragment of Peano arithmetic. Notice the subtle distinction between the syntactic and semantic domains. The syntactic expressions are mapped into an algebra of the natural numbers by the valuation function. The denotational definition almost seems to be unnecessary. Since the syntax so closely resembles that of the semantic algebra.

Programming languages are not as close to their computational model. Figure N.7 is a denotational definition of the small imperative programming language Simple encountered in the previous chapter.
Figure N.7: Denotational semantics for Simple

Abstract Syntax:

C in Command  
E in Expression  
O in Operator  
N in Numeral  
V in Variable

\[
C ::= V := E \mid \text{if } E \text{ then } C_1 \text{ else } C_2 \text{ end } \mid \text{while } E \text{ do } C_3 \text{ end } \mid C_1;C_2 \mid \text{skip}
\]

\[
E ::= V \mid N \mid E_1 O E_2 \mid (E)
\]

O ::= + | - | * | / | = | < | > | <>

Semantic Algebra:

Domains:

\( \tau \) in \( T = \{ \text{true, false} \} \); the boolean values

\( \zeta \) in \( Z = \{ \ldots -1, 0, 1, \ldots \} \); the integers

\( + : Z \rightarrow Z \rightarrow Z \)

\( \ldots \)

\( = : Z \rightarrow Z \rightarrow T \)

\( \ldots \)

\( \sigma \) in \( S = \text{Variable} \rightarrow \text{Numeral}; \) the state

Valuation Functions:

\( C \) in \( C \rightarrow (S \rightarrow S) \)

\( E \) in \( E \rightarrow E \rightarrow (N \cup T) \)

\[
C[ \text{skip }] \quad \sigma = \sigma
\]

\[
C[ V := E ] \quad \sigma = \sigma[ V:E[ E ] \sigma]
\]

\[
C[ C_1; C_2 ] = C[ C_2 ] \cdot C[ C_1]
\]

\[
C[ \text{if } E \text{ then } C_1 \text{ else } C_2 \text{ end } ] \quad \sigma
\]

\[
= C[ C_1 ] \sigma \quad \text{if } E[ E ]\sigma = \text{true}
\]

\[
= C[ C_2 ] \sigma \quad \text{if } E[ E ]\sigma = \text{false}
\]

\[
C[ \text{while } E \text{ do } C \text{ end} ] \sigma
\]

\[
= \lim_{n \to \infty} C[ (\text{if } E \text{ then } C \text{ else } \text{skip} \text{ end})^n ] \sigma
\]

\[
E[ V ] \quad \sigma = \sigma(V)
\]

\[
E[ N ] = \zeta
\]
Denotational definitions are favored for theoretical and comparative programming language studies. Denotational definitions have been used for the automatic construction of compilers for the programming language. Denotations other than mathematical objects are possible. For example, a compiler writer would prefer that the object denoted would be appropriate object code. Systems have been developed for the automatic construction of compilers from the denotation specification of a programming language.

**Operational Semantics**

An operational definition of a language consists of two parts: an abstract syntax and an interpreter. An interpreter defines how to perform a computation. When the interpreter evaluates a program, it generates a sequence of machine configurations that define the program's operational semantics. The interpreter is an evaluation relation that is defined by rewriting rules. The interpreter may be an abstract machine or recursive functions. Figure N.8 is an example of an operational definition.

---

**Figure N.8:** Operational semantics for Peano arithmetic

**Abstract Syntax:**

\[
N \text{ in Nat (the natural numbers)}
\]

\[
N ::= 0 \mid S(N) \mid (N + N) \mid (N \times N)
\]

**Interpreter:**

\[
I: N \rightarrow N
\]

\[
I[ (n + 0)] \rightarrow n
\]

\[
I[ (m + S(n))] \rightarrow S(I[(m+n)])
\]

\[
I[ (n \times 0)] \rightarrow 0
\]

\[
I[ (m \times S(n))] \rightarrow I[(m \times n) + m]
\]

where \(m, n\) in Nat
It is an operational definition of a fragment of Peano arithmetic.

The interpreter is used to rewrite natural number expressions to a standard form (a form involving only S and 0) and the rewriting rules show how move the + and × operators inward toward the base cases. Operational definitions are favored by language implementors for the construction of compilers and by language tutorials because operational definitions describe how the actions take place.

The operational semantics of Simp is found in Figure N.9.

**Figure N.9: Operational semantics for Simple**

**Interpreter:**

\[
I: C \times \Sigma \rightarrow \Sigma \\
\{nu\} \text{ in } E \times \Sigma \} \rightarrow T \cup Z
\]

**Semantic Equations:**

\[
I(\text{skip}, \sigma) = \sigma \\
I(V := E, \sigma) = \sigma[V: nu(E, \sigma)] \\
I(C_1; C_2, \sigma) = E(C_2, E(C_1, \sigma)) \\
I(\text{if } E \text{ then } C_1 \text{ else } C_2 \text{ end}, \sigma) = I(C_1, \sigma) & \text{if } nu(E, \sigma) = \text{true} \\
& \text{if } nu(E, \sigma) = \text{false} \\
while E \text{ do } C \text{ end} = \text{if } E \text{ then } (C; \text{while } E \text{ do } C \text{ end}) \text{ else } \text{skip} \\
u(V, \sigma) = \sigma(V) \\
nu(N, \sigma) = N \\
nu(E_1 + E_2, \sigma) = nu(E_1, \sigma) + nu(E_2, \sigma) \\
... \\
nu(E_1 = E_2, \sigma) = \text{true if } nu(E, \sigma) = nu(E, \sigma) \\
& \text{false if } nu(E, \sigma) \neq nu(E, \sigma) \\
\text{otherwise} \\
...
\]
The operational semantics are defined by using two semantic functions, $I$ which interprets commands and $\nu$ which evaluates expressions. The interpreter is more complex since there is an environment associated with the program with does not appear as a syntactic element and the environment is the result of the computation. The environment (variously called the store or referencing environment) is an association between variables and the values to which they are assigned. Initially the environment is empty since no variable has been assigned to a value. During program execution each assignment updates the environment. The interpreter has an auxiliary function which is used to evaluate expressions. The while command is given a recursive definition but may be defined using the interpreter instead. Operational semantics are particularly useful in constructing an implementation of a programming language.

### Pragmatics

The use of formal semantic description techniques is playing an increasing role in software engineering. Algebraic semantics are useful for the specification of abstract data types. However, the lack of robust theorem provers has limited the effective use axiomatic semantics for program verification. Denotational semantics are beginning to play a role in compiler construction and a prescriptive rather than a descriptive role in the design of programming languages. Operational semantics have always proved helpful in the design of compilers.

### Historical Perspectives and Further Reading

An excellent short introduction to operational, denotational and axiomatic semantics is found in

Schmidt, David A.

Other references include:

- General texts: algebraic, axiomatic, denotational, and operational semantics
  
  Slonneger & Kurts (1995)
  
  *Formal Syntax and Semantics of Programming Languages* Addison Wesley
  
  Meyer, Bertrand Meyer. (1990)
  
  *Introduction to the Theory of Programming Languages* Prentice-Hall International.
  
  Watt, David A. (1991)
  
  *Programming Language Syntax and Semantics* Prentice-Hall International.

- Axiomatic semantics
  
  Gries, David (1981)
  
  *The Science of Programming* Springer-Verlag.

  
  *The Logic of Programming* Prentice-Hall International.
Denotational semantics

Exercises

1. (axiomatic) Give axiomatic semantics for the following:
   a. Multiple assignment command: \( x_0,\ldots,x_n := e_0,\ldots,e_n \)
   b. The following commands are a nondeterministic if and a nondeterministic loop. The IF command allows for a choice between alternatives while the DO command provides for iteration. In their simplest forms, an IF statement corresponds to an If condition then command and a LOOP statement corresponds to a While condition Do command.

   \[
   \text{IF } \text{guard } \rightarrow \text{command} \text{ FI } = \text{ if } \text{guard} \text{ then } \text{command}
   \]

   \[
   \text{LOOP } \text{guard } \rightarrow \text{command} \text{ POOL } = \text{ while } \text{guard} \text{ do } \text{command}
   \]

   A command proceeded by a guard can only be executed if the guard is true. In the general case, the semantics of the IF - FI and LOOP - POOL commands requires that only one command corresponding to a guard that is true be selected for execution. The selection is nondeterministic. Define the axiomatic semantics for the IF and LOOP commands:
   i. if \( c_0 \rightarrow s_0 \)
   ...
   c_n \rightarrow s_n
   fi
   ii. do \( c_0 \rightarrow s_0 \)
   ...
   c_n \rightarrow s_n
   od
   c. A for statement
   d. A repeat-until statement

2. (axiomatic) Use assertions to guide the construction of the following programs.
   a. Linear search
   b. Integer division implemented by repeated subtraction.
   c. Factorial function
   d. \( F_n \) the \( n \)-th Fibonacci number where \( F_0 = 0, F_1 = 1, \) and \( F_{i+2} = F_{i+1} + F_i \) for \( i \geq 0 \).
3. (algebraic) Construct algebraic semantics for the following:
   a. Stack
   b. List
   c. Queue
   d. Binary search tree
   e. Graph
   f. Grade book
   g. Complex numbers
   h. Rational numbers
   i. Floating point numbers
   j. Simple

4. (denotational) Construct denotational semantics for the following:
   a. Stack
   b. List
   c. Queue
   d. Binary search tree
   e. Graph
   f. Grade book
   g. Complex numbers
   h. Rational numbers
   i. Floating point numbers
   j. Simple
   k. Show that the following code denotes the same function.

```c
int f (int n)
{ if n > 1 then n*f(n-1)
   else 1
}
```

```c
int f (int n)
{ int t = 1;
   while n > 1 do {
      t := t*n;
      n := n-1
   }
}
```

5. (operational) Construct operational semantics for the following:
   a. Stack
   b. List
   c. Queue
   d. Binary search tree
6. (correctness) Construct an implementation of the following and show that your implementation is correct by showing that it satisfies a semantics.
   a. Stack
   b. List
   c. Queue
   d. Binary search tree
   e. Graph
   f. Grade book
   g. Complex numbers
   h. Rational numbers
   i. Floating point numbers
   j. Simple
Introduction to Compilers

A language translator is a program which translates programs from source language into an equivalent program in an object language.

Keywords and phrases: source-language, object-language, syntax-directed, compiler, assembler, linker, loader, parser, scanner, top-down, bottom-up, context-free grammar, regular expressions

Introduction

A computer constructed from actual physical devices is termed an actual computer or hardware computer. From the programming point of view, it is the instruction set of the hardware that defines a machine. An operating system is built on top of a machine to manage access to the machine and to provide additional services. The services provided by the operating system constitute another machine, a virtual machine.

A programming language provides a set of operations. Thus, for example, it is possible to speak of a Java computer or a Haskell computer. For the programmer, the programming language is the computer; the programming language defines a virtual computer. The virtual machine for Simple consists of a data area which contains the association between variables and values and the program which manipulates the data area.

Figure M.N: Simple's Virtual Machine and Runtime Environment

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program counter</td>
<td>Code Segment</td>
</tr>
<tr>
<td></td>
<td>Data Segment</td>
</tr>
</tbody>
</table>

Figure M.N: C's Virtual machine
Between the programmer's view of the program and the virtual machine provided by the operating system is another virtual machine. It consists of the data structures and algorithms necessary to support the execution of the program. This virtual machine is the run time system of the language. Its complexity may range in size from virtually nothing, as in the case of FORTRAN, to an extremely sophisticated system supporting memory management and inter process communication as in the case of a concurrent programming language like SR. The run time system for Simple as includes the processing unit capable of executing the code and a data area in which the values assigned to variables are accessed through an offset into the data area.

User programs constitute another class of virtual machines.

A language translator is a program which translates programs from source language into an equivalent program in an object language. The source language is usually a high-level programming language and the object language is usually the machine language of an actual computer. From the pragmatic point of view, the translator defines the semantics of the programming language, it transforms operations specified by the syntax into operations of the computational model---in this case, to some virtual machine. Context-free grammars are used in the construction of language translators. Since the translation is based on the syntax of the source language, the translation is said to be **syntax-directed**.
A **compiler** is a translator whose source language is a high-level language and whose object language is close to the machine language of an actual computer. The typical compiler consists of an analysis phase and a synthesis phase.

In contrast with compilers an **interpreter** is a program which simulates the execution of programs written in a source language. Interpreters may be used either at the source program level or an interpreter may be used it interpret an object code for an idealized machine. This is the case when a compiler generates code for an idealized machine whose architecture more closely resembles the source code.

There are several other types of translators that are often used in conjunction with a compiler to facilitate the execution of programs. An **assembler** is a translator whose source language (an assembly language) represents a one-to-one transliteration of the object machine code. Some compilers generate assembly code which is then assembled into machine code by an assembler. A **loader** is a translator whose source and object languages are machine language. The source language programs contain tables of data specifying points in the program which must be modified if the program is to be executed. A **link editor** takes collections of executable programs and links them together for actual execution. A **preprocessor** is a translator whose source language is an extended form of some high-level language and whose object language is the standard form of the high-level language.

The typical compiler consists of several phases each of which passes its output to the next phase

- The **lexical phase** (scanner) groups characters into lexical units or tokens. The input to the lexical phase is a character stream. The output is a stream of tokens. Regular expressions are used to define the tokens recognized by a scanner (or lexical analyzer). The scanner is implemented as a finite state machine.
- The **parser** groups tokens into syntactical units. The output of the parser is a parse tree representation of the program. Context-free grammars are used to define the program structure recognized by a parser. The parser is implemented as a push-down automata.
- The **contextual analysis phase** analyzes the parse tree for context-sensitive information often called the **static semantics**. The output of the contextual analysis phase is an annotated parse tree. Attribute grammars are used to describe the static semantics of a program.
- The **optimizer** applies semantics preserving transformation to the annotated parse tree to simplify the structure of the tree and to facilitate the generation of more efficient code.
- The **code generator** transforms the simplified annotated parse tree into object code using rules which denote the semantics of the source language.
- The **peep-hole** optimizer examines the object code, a few instructions at a time, and attempts to do machine dependent code improvements.

---

**Figure N.1:** Traditional Compiler Structure
The Scanner

The scanner groups the input stream (of characters) into a stream of tokens (lexeme) and constructs a symbol table which is used later for contextual analysis. The lexemes include

- Key words,
- identifiers,
- operators,
- constants: numeric, character, special, and
- comments.

The lexical phase (scanner) groups characters into lexical units or tokens. The input to the lexical phase is a character stream. The output is a stream of tokens. Regular expressions are used to define the tokens recognized by a scanner (or lexical analyzer). The scanner is implemented as a finite state machine.

Lex and Flex are tools for generating scanners in C. Flex is a faster version of Lex.

The Parser
The parser groups tokens into syntactical units. The output of the parser is a parse tree representation of the program. Context-free grammars are used to define the program structure recognized by a parser. The parser is implemented as a push-down automata.

Yacc and Bison are tools for generating bottom-up parsers in C. Bison is a faster version of Yacc. Jack is a tool for generating scanners and top-down parsers in Java.

**Symbol Tables and Error Handlers**

In addition to a data stream passing through the phases of the compiler, additional information acquired during a phase may be needed by a later phase. The symbol table is used to store the names encountered in the source program and relevant attributes. The information in the symbol table is used by the semantic checker when applying the context-sensitive rules and by the code generator. The error handler is used to report and recover from errors encountered in the source.

**Contextual Checkers**

Contextual checkers analyze the parse tree for context-sensitive information often called the static semantics. The output of the semantic analysis phase is an annotated parse tree. Attribute grammars are used to describe the static semantics of a program.

This phase is often combined with the parser. During the parse, information concerning variables and other objects is stored in a symbol table. The information is utilized to perform the context-sensitive checking.

**Intermediate Code Generator**

The data structure passed between the analysis and synthesis phases is called the intermediate representation (IR) of the program. A well-designed intermediate representation facilitates the independence of the analysis and syntheses (front- and back-end) phases. Intermediate representations may be

- assembly language like or
- be an abstract syntax tree.

**Code Optimizer**

Restructuring the parse tree to reduce its size or to present an equivalent tree from which the code generator can produce more efficient code is called optimization.
It may be possible to restructure the parse tree to reduce its size or to present a parse to the code generator from which the code generator is able to produce more efficient code. Some optimizations that can be applied to the parse tree are illustrated using source code rather than the parse tree.

- **Constant folding**

  \[
  I := 4 + J - 5; \quad \rightarrow \quad I := J - 1;
  \]
  or
  \[
  I := 3; \quad J := I + 2; \quad \rightarrow \quad I := 3; \quad J := 5
  \]

- **Loop-Constant code motion**

  From:
  
  ```
  while (count < limit) do
    INPUT SALES;
    VALUE := SALES * ( MARK_UP + TAX );
    OUTPUT := VALUE;
    COUNT := COUNT + 1;
  end;  -->
  ```

  to:
  
  ```
  TEMP :=  MARK_UP + TAX;
  while (COUNT < LIMIT) do
    INPUT SALES;
    VALUE := SALES * TEMP;
    OUTPUT := VALUE;
    COUNT := COUNT + 1;
  end;
  ```

- **Induction variable elimination** Most program time is spent in the body of loops so loop optimization can result in significant performance improvement. Often the induction variable of a for loop is used only within the loop. In this case, the induction variable may be stored in a register rather than in memory. And when the induction variable of a for loop is referenced only as an array subscript, it may be initialized to the initial address of the array and incremented by only used for address calculation. In such cases, its initial value may be set

  From:
  ```
  For I := 1 to 10 do
  ```
  
  to:
  ```
  For I := address of first element in A
  to address of last element in A
  increment by size of an element of A do
  ```

- **Common subexpression elimination**

  From:
  
  \[
  A := 6 \times (B+C); \\
  D := 3 + 7 \times (B+C); \\
  E := A \times (B+C); \\
  \]

  to:
  
  \[
  TEMP := B + C; \\
  A := 6 \times TEMP; \\
  D := 3 \times 7 \times TEMP; \\
  E := A \times TEMP; \\
  \]

- **Strength reduction**

  \[
  2 \times x \rightarrow x + x \\
  2 \times x \rightarrow \text{shift left } x \\
  \]

- **Mathematical identities**

  \[
  a \times b + a \times c \rightarrow a \times (b+c) \\
  a - b \rightarrow a + (\neg b) \\
  \]

We do not illustrate an optimizer in the parser for Simp.

**Code Generator**

The **code generator** transforms the intermediate representation into object code using rules which denote the semantics of the source language. These rules are define a *translation semantics*.

The **code generator**'s task is to translate the intermediate representation to the *native code* of the target machine. The native code may be an actual executable binary, assembly code or another high-level language. Producing low-level code requires familiarity with such machine level issues such as

- data handling
- machine instruction syntax
- variable allocation
- program layout
- registers
- instruction set
The code generator may be integrated with the parser.

As the source program is processed, it is converted to an internal form. The internal representation in the example is that of an implicit parse tree. Other internal forms may be used which resemble assembly code. The internal form is translated by the code generator into object code. Typically, the object code is a program for a virtual machine. The virtual machine chosen for Simp consists of three segments. A data segment, a code segment and an expression stack.

The data segment contains the values associated with the variables. Each variable is assigned to a location which holds the associated value. Thus, part of the activity of code generation is to associate an address with each variable. The code segment consists of a sequence of operations. Program constants are incorporated in the code segment since their values do not change. The expression stack is a stack which is used to hold intermediate values in the evaluation of expressions. The presence of the expression stack indicates that the virtual machine for Simp is a `stack machine".

### Declaration translation

Declarations define an environment. To reserve space for the data values, the `DATA` instruction is used.

\[
\text{integer } x, y, z. \quad \text{DATA 2}
\]

### Statement translation

The assignment, if, while, read and write statements are translated as follows:

- **Assignment**
  \[
  x := \text{expr} \\
  \text{code for expr} \\
  \text{STORE X}
  \]

- **Conditional**
  \[
  \text{if } C \text{ then} \\
  \text{S1} \\
  \text{else} \\
  \text{S2} \\
  \text{end} \\
  \text{L1: BR_FALSE} \\
  \text{L2: code for S1} \\
  \text{L1: BR L2} \\
  \text{L2: code for S2}
  \]

- **While-do**
  \[
  \text{while } C \text{ do } S \\
  \text{L1: code for C} \\
  \text{BR_FALSE L2} \\
  \text{L2: code for S} \\
  \text{BR L1}
  \]

```
If the code is placed in an array, then the label addresses must be *back-patched* into the code when they become available.

**Expression translation**

Expressions are evaluated on an expression stack. Expressions are translated as follows:

- constant LD_INT constant
- variable LD variable
- e1 op e2 code for e1
- code for e2
- code for op

**Peephole Optimizer**

*Peephole optimizers* scan small segments of the *target code* for standard replacement patterns of inefficient instruction sequences. The peephole optimizer produces machine dependent code improvements.

Figure N.1 contains a context-free grammar for a simple imperative programming language. It will be used to illustrate the concepts in this chapter.

---

**Figure N.2: Context-free grammar for Simple**

```
program ::= LET definitions IN command_sequence END

definitions ::= e | INTEGER id_seq IDENTIFIER .

id_seq ::= e | id_seq IDENTIFIER ,

command_sequence ::= e | command_sequence command ;
```
Systematic development of a recursive descent parser

A parser groups sequences of tokens into larger meaningful units described by a context-free grammar. The parser takes as input a stream of tokens where each token contains both the class and spelling of a token. The stream of tokens is processed sequentially and currentToken contains the token of immediate interest. The output of the parser is a syntax tree. The tree may or may not be built explicitly.

There are four steps in the systematic construction of a recursive descent parser.

1. Transform the grammar into proper form.
2. Determine the sets First[E] and Follow[N] for each right-hand side E and non-terminal N of the grammar.
3. Construct parsing procedures from the grammar.
4. Construct the parser.

Figure N.1 summarizes the grammar transformation rules.

---

**Figure N.3: Grammar Transformation Rules**

- Convert the grammar to EBNF
- Remove left-recursion: replace N ::= E | NF with N ::= E(F)*
- Left-factor the grammar: replace N ::= EFG | EF'G with N ::= E(F|F')G
- If N ::= E is not recursive, remove it and replace all occurrences of N in the grammar with E
First the grammar is converted to EBNF. The resulting grammar must have a single production rule for each non-terminal symbol. Next, rules containing left recursion are transformed to rules which do not contain left recursion. Left recursion occurs when the same non-terminal appears both at the head of the rule and as a left-most symbol on the right-hand side. The parser can enter an infinite loop if this transformation is not done. Mutual recursion must also be eliminated but it is more difficult. Next, the grammar is simplified by replacing non-terminals with their defining body. This should be done bottom up, stopping when recursion is encountered. Finally, simplify the grammar by factoring the right-hand sides. This makes it easier for the parser to select the correct grammar rule.

The first and follow sets are used by the parser to select the applicable grammar rule. Figure N.2 summarizes the rules for computing the First and Follow sets.

---

**Figure N.2:** First[E] and Follow[N]

| First[e]   | = empty set
| First[t]   | = \{t\} \ t is a terminal
| First[N]   | = First[E] \ where N ::= E
| First[E F] | = First[E] union First[F] \ if E generates lambda
|            | = First[E] \ otherwise
| First[E|F]   | = First[E] union First[F]
| First[E*]  | = First[E]
| Follow[N]  | = \{t\} \ in context Nt, t is terminal
|            | = First[F] \ in context NF, F is non-terminal

---

The First[E] is the set of terminal symbols that can start a string generated by E. The Follow[N] is the set of terminal symbols that can appear in strings that follow those strings generated by N. The importance of the first and follow sets becomes apparent when the grammar rules are converted to parsing procedures.

Figure N.3 summarizes the rules for converting the EBNF grammar to a collection of parsing procedures.
Figure N.3: EBNF to Parsing Procedures

- For each grammar rule \( N ::= E \), construct a parsing procedure

\[
\text{parseN} \{ \\
\quad \text{parse } E \\
\}
\]

- Refine \( \text{parse } E \)

If \( \text{parse } E \) is: then refine to:

- \( \text{parse lambda} \) skip
- \( \text{parse } t \) accept(\( t \)) where \( t \) is a terminal
- \( \text{parse } N \) parseN where \( N \) is a non-terminal
- \( \text{parse } E \ F \) parse \( E \); parse \( F \)
- \( \text{parse } E|F \) if currentToken.class in First[\( E \)] then
  parse \( E \)
  else if currentToken.class in First[\( F \)] then
  parse \( F \)
  else report a syntactic error
- \( \text{parse } E^* \) while currentToken.class in First[\( E \)] do
  parse \( E \)

If \( \text{parse } E \) is \( \text{parse lambda} \) (recall \( \text{lambda} \) is the empty string), then \( \text{parse } E \) is the skip command. If \( \text{parse } E \) is \( \text{parse } t \) (where \( t \) is a terminal symbol), then \( \text{parse } E \) is accept(\( t \)). If the current token is known to be \( t \), then acceptIt. If \( \text{parse } E \) is \( \text{parse } N \) (where \( N \) is a non-terminal), then \( \text{parse } E \) is the call parseN. If \( \text{parse } E \) is \( \text{parse } E \ F \), then \( \text{parse } E \) is \{parse \( E \); parse \( F \}\). If \( \text{parse } E \) is \( \text{parse } E|F \), then \( \text{parse } E \) is

\[
\text{if currentToken.class in First[\( E \)] then} \\
\quad \text{parse } E \\
\text{else if currentToken.class in First[\( F \)] then} \\
\quad \text{parse } F \\
\text{else} \\
\quad \text{report a syntactic error}
\]

where First[\( E \)] and First[\( F \)] are disjoint. If \( \text{parse } E \) is \( \text{parse } E^* \), then \( \text{parse } E \) is

\[
\text{while currentToken.class in First[\( E \)] do}
\]
where First[E] is disjoint from Follow[E*]

The parser consists of:

- a global variable currentToken;
- auxiliary procedures
  - scanToken obtains the next token from the scanner
  - accept(tc) which obtains the next token from the scanner if the current token is of the class tc else returns a syntactic error. In some instances, the current token is known and then a simplified procedure acceptIt may be used. It obtains the next token from the scanner.
- the parsing procedures developed from the grammar;
- a driver parse that calls parseS (where S is the start symbol of the grammar) after having called the scanner to store the first input token in currentToken;

```plaintext
parse() {
    getChar;
    scanToken;
    parseS;
}
```

Systematic development of a table-driven parser

Given a grammar which satisfies the restrictions specified in the recursive descent parser construction, a table-driven parser may be constructed using the top-down parsing algorithm.

Systematic development of a scanner

A scanner groups sequences of characters into tokens described by a regular grammar. The scanner takes as input a stream of characters. The stream of characters is processed sequentially and currentChar contains the character of immediate interest. The characters defining a token are collected into a string and the class of the token is identified. The output of the scanner is a stream of tokens. Each token contains information concerning its class and spelling.

There are three steps in the systematic construction of a scanner.

1. Transform the regular expressions into an EBNF grammar.
2. Transcribe each EBNF production rule  \textit{N} ::= \textit{E} to a scanning procedure scan\textit{N}, whose body is determined by \textit{E}.
3. Construct the scanner.

Figure N.M summarizes the rules for transforming the regular expressions to an EBNF grammar.

**Figure N.M: RE to EBNF**

- Each regular expression $RE_i$ defining a token class $T_i$ is put into the EBNF form:
  
  $T_i ::= RE_i$.

- A regular expression $Sep$ is constructed defining the symbols which separate tokens.

- The EBNF production $S ::= Sep*(T_0|...|T_n)$ is added to the grammar.

For each regular expression $RE$ defining a token $T$, the EBNF rule $T ::= RE$. A regular expression $sep*$ defining the strings that separate tokens is constructed. And the EBNF production $S ::= Sep*(T_0|...|T_n)$ is defined.

**Figure N.3: EBNF to Scanning Procedures**

- For each grammar rule $T_i ::= E_i$, construct a scanning procedure $scan_{T_i} \{ scan \ E_i \}$.

- Refine $scan \ E_i$

<table>
<thead>
<tr>
<th>$scan \ E_i$</th>
<th>Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$scan \ lambda$</td>
<td>skip</td>
</tr>
<tr>
<td>$scan \ ch$</td>
<td>takeIt(t) where ch is a character</td>
</tr>
<tr>
<td>$scan \ N$</td>
<td>scanN where $N$ is a non-terminal</td>
</tr>
<tr>
<td>$scan \ E \ F$</td>
<td>scan $E$; scan $F$</td>
</tr>
</tbody>
</table>
  | $scan \ E|F$ | if currentChar in First[$E$] then $scan \ E$
  | | else if currentChar in First[$F$] then
  | | $scan \ F$
  | | else report a syntactic error |
  | $scan \ E*$ | while currentChar in First[$E$] do $scan \ E$ |

The scanner is developed from an EBNF grammar (must be non-self embedding) as follows:
1. Objects
   - currentChar contains the current character.
   - currentToken contains the current token, its spelling and its class.
2. Convert the grammar to EBNF with a single production rule for each non-terminal symbol.
3. The scanner consists of the procedures developed in step (2) enhanced to record the token's class and spelling:
4. a procedure scanToken that scans 'separator*Token', and sets currentToken.spelling to the character string scanned and currentToken.class token.
5. the auxiliary procedures
   - start sets currentToken.spelling to the empty string.
   - getChar appends currentChar to currentToken.spelling and fetches the next character into currentChar.
   - finish sets currentToken.class to the identified class (used for simple disjoint classes)
   - screen sets currentToken.class to the identified class (used for complex classes that require additional analysis to determine class).

If currentChar is part of currentToken which is under construction, the procedure takeIt adds currentChar to currentToken and If currentChar is not part of currentToken which is under construction, the procedure leaveIt adds currentChar to currentToken.

Attribute Grammars and Contextual Constraints

Context-free grammars are not able to completely specify the structure of programming languages. For example, declaration of names before reference, number and type of parameters in procedures and functions, the correspondence between formal and actual parameters, name or structural equivalence, scope rules, and the distinction between identifiers and reserved words are all structural aspects of programming languages which cannot be specified using context-free grammars. These context-sensitive aspects of the grammar are often called the static semantics of the language. The term dynamic semantics is used to refer to semantics proper, that is, the relationship between the syntax and the computational model. Even in a simple language like Simp, context-free grammars are unable to specify that variables appearing in expressions must have an assigned value. Context-free descriptions of syntax are supplemented with natural language descriptions of the static semantics or are extended to become attribute grammars.

Attribute grammars are an extension of context-free grammars which permit the specification of context-sensitive properties of programming languages. Attribute grammars are actually much more powerful and are fully capable of specifying the semantics of programming languages as well.

For an example, the following partial syntax of an imperative programming language requires the declaration of variables before reference to the variables.
\[
P ::= D \ B \\
D ::= V \ldots \\
B ::= C \ldots \\
C ::= V ::= E \mid \ldots
\]

However, this context-free syntax does not indicate this restriction. The declarations define an environment in which the body of the program executes. Attribute grammars permit the explicit description of the environment and its interaction with the body of the program.

Since there is no generally accepted notation for attribute grammars, attribute grammars will be represented as context-free grammars which permit the parameterization of non-terminals and the addition of where statements which provide further restrictions on the parameters. Figure~\ref{ag:decl} is an attribute grammar for declarations.

\begin{figure}
\centering
\begin{align*}
P ::= D(\text{SymbolTable}) \ B(\text{SymbolTable}) \\
D(\text{SymbolTable}) & ::= \ldots V(\text{ insert(} \ V \text{ in SymbolTable)}\ldots \\
B(\text{SymbolTable}) & ::= C(\text{SymbolTable})\ldots \\
C(\text{SymbolTable}) & ::= V ::= E(\text{SymbolTable, Error(if} \ V \text{ not in SymbolTable)} \\
& \quad | \ldots
\end{align*}
\caption{An attribute grammar for declarations}
\end{figure}

The parameters marked with $\downarrow$ are called inherited attributes and denote attributes which are passed down the parse tree while the parameters marked with $\uparrow$ are called synthesized attributes and denote attributes which are passed up the parse tree. Attribute grammars have considerable expressive power beyond there use to specify context sensitive portions of the syntax and may be used to specify:

- context sensitive rules
- evaluation of expressions
- translation

**Historical Perspectives and Further Reading**

For information on compiler construction using Lex and Yacc see\cite{SchFre85}. Pratt \cite{Pratt84}
Translation emphasizes virtual machines. ELI, PCCTS, FLEX/BISON, LEX/YACC, Amsterdam Compiler Kit, Jack

Exercises

1. (translation) Construct a translation semantics for
   a. Simple
   b. HTML to TeX/LaTeX
   c. TeX/LaTeX to HTML
2. Construct a scanner and a parser for expressions (use a grammar from chapter 2)
3. Construct an attribute grammar for expressions
4. Construct a calculator using the attribute grammar for expressions.
5. Construct a scanner for Simple
6. Construct a parser for Simple
7. Construct a code generator for Simple
8. Construct an interpreter for Simple
9. Construct an interpreter for BASIC.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of Anthony A. Aaby. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee.
© 1998 Anthony A. Aaby. Last Modified -. Send comments to aabyan@wwc.edu
Pragmatics

The pragmatics of a programming language includes issues such as ease of implementation, efficiency in application, and programming methodology. -- Slonneger & Kurtz

Keywords and phrases: strict, non-strict, eager evaluation, lazy evaluation, normal-order evaluation, binding time, passing by value, passing by reference, passing by name, passing by value, passing by result, passing by value-result, aliasing

Heaps and Pointers

Treating procedure and function abstractions as first-class values is another potential cause of dangling references. (Watt)

A heap variable is one that can be created and deleted at any time. Heap variables are anonymous and are accessed through pointers. A heap is a block of storage within which pieces are allocated and freed in some relatively unstructured manner.

Initially the elements of the heap are linked together in some fashion to form a free-space list. The creation of a heap variable is requested by an operation called an allocator which returns a pointer to the newly created variable. To allocate an element, the first element on the list is removed from the list and a pointer to it is returned to the operation requesting the storage.

The heap variable's lifetime extends from the time it is created until it is no longer accessible.

Often there is an operation called a deallocator which forcibly deletes a given heap variable. When an element is deallocated (freed), it is simply linked back in at the head of the free-space list. If all references to heap variable are destroyed, the heap variable is inaccessible and becomes garbage. When a variable becomes garbage, its memory space is unusable by other variables since a means of referencing it must exist in order to return the space to the free-space list.

When deallocation is under the control of the programmer, it is a potential source of problems. If a programmer deallocates a variable, any remaining pointers to the deleted heap variable become dangling references.

Garbage and dangling references are potentially troublesome for the programmer. If garbage
accumulates, available storage is gradually reduced until the program may be unable to continue for lack of known free space (this is also called a memory leak). If a program attempts to modify through a dangling reference a structure that has been deallocated (destroyed), the contents of an element of free-space may be modified. This may cause the remainder of the free-space to become garbage or a portion of the program to become linked to free-space. The deallocated space could be reallocated to some other structure resulting in similar problems.

The problem of dangling references can be eliminated.

One solution is to restrict assignment so that references to local variables may not be assigned to variables with a longer lifetime. This restriction may require runtime checks and sometimes restrict the programmer.

Another solution is to maintain reference counts with each heap variable. An integer called the reference count is associated with each heap element. The reference count indicates the number of pointers to the element that exist. Initially the count is set to 1. Each time a pointer to the element is created the reference count is increased and each time a pointer to the element is destroyed the reference count is decreased. Its space is not deallocated until the reference count reaches zero. The method of reference counting results in substantial overhead in time and space.

Another solution is to provide garbage collection. The basic idea is to allow garbage to be generated in order to avoid dangling references. When the free-space list is exhausted and more storage is needed, computation is suspended and a special procedure called a garbage collector is started which identifies garbage and returns it to the free-space list.

There are two stages to garbage collection a marking phase and a collecting phase.

- Marking phase: The marking phase begins outside the heap with the pointers that point to active heap elements. The chains of pointers are followed and each heap element in the chain is marked to indicate that it is active. When this phase is finished only active heap elements are marked as active.
- Collecting phase: During the collecting phase the heap is scanned and each element which is not active is returned to the free-space list and the marked bits are reset to prepare for a later garbage collection.

This unuseable space may be reclaimed by a garbage collector. A heap variable is alive as long as any reference to it exists.

**Coroutines**

Coroutines are used in discrete simulation languages and, for some problems, provide a control structure that is more natural than the usual hierarchy of subprogram calls.
Coroutines may be thought of as subprograms which are not required to terminate before returning to the calling routine. At a later point the calling program may "resume" execution of the coroutine at the point from which execution was suspended. Coroutines then appear as equals with control passing from one to the other as necessary. From two coroutines it is natural to extend this to a set of coroutines.

From the description given of coroutines, it is apparent that coroutines should not be recursive. This permits us to use just one activation record for each coroutine and the address of each activation record can be statically maintained.

Each activation record is extended to include a location to store the CI for the corresponding coroutine. It is initialized with the location of the first instruction of the coroutine. When coroutine encounters a resume operation, it stores the address of its next instruction in its own activation record. The address of the CI for the resumed coroutine is obtained from the activation record of the resumed coroutine.

### Safety

The purpose of declarations is two fold. The requirement that all names be declared is essential to provide a check on spelling. It is not unusual for a programmer to misspell a name. When declarations are not required, there is no way to determine if a name is new or if it is a misspelling of a previous name.

The second purpose of declarations is assist the type checking algorithm. The type checker can determine if the intended type of a variable matches the use of the variable. This sort of type checking can be performed at compile time permitting the generation of more efficient code since run time type checks need not be performed.

Declarations and strong type checking facilitate safety by providing redundancy. When the programmer has to specify the type of every entity, and may declare only one entity with a given identifier within a given scope; the compiler then simply checks each the usage of each entity against rigid type rules. With overloading or type inference, the compiler must deduce information not supplied by the programmer. This is error prone since slight errors may radically affect what the compiler does.

Overloading and type inference lack redundancy.

### Historical Perspectives and Further Reading

Wadler, Philip (1996)
Exercises

1. [time/difficulty](section) Problem statement.
2. (compiler) Implement a virtual machine which provides ....

© 1996 by A. Aaby
Abstraction and Generalization

**Abstraction** is an emphasis on the idea, qualities and properties rather than the particulars (a suppression of detail).

**Generalization** is a broadening of application to encompass a larger domain of objects of the same or different type.

A parameter is a quantity whose value varies with the circumstances of its application.

Substitution---To put something in the place of another.

Encapsulate---To completely enclose.

Keywords and phrases: Name, binding, abstract, definition, declaration, variables, parameters, arguments, formals, actuals. Modularity, encapsulation, function, procedure, abstract type, generic, library, object, class, inheritance, partition, package, unit, separate compilation, linking, import, export, instance, scope. block, garbage collection, static and dynamic links, display, static and dynamic binding, activation record, environment, Static and Dynamic Scope, aliasing, variables, value, result, value-result, reference, name, unification, eager evaluation, lazy evaluation, strict, non-strict, Church-Rosser, overloading, polymorphism, monomorphism, coercion, transfer functions.

The ability to abstract and to generalize is an essential part of any intellectual activity. Abstraction and generalization are fundamental to mathematics and philosophy and are essential in computer science as well.

The importance of abstraction is derived from its ability to hide irrelevant details and from the use of names to reference objects. Programming languages provide abstraction through procedures, functions, and modules which permit the programmer to distinguish between what a program does and how it is implemented. The primary concern of the user of a program is with what it does. This is in contrast with the writer of the program whose primary concern is with how it is implemented. Abstraction is essential in the construction of programs. It places the emphasis on what an object is or does rather than how it is represented or how it works. Thus, it is the primary means of managing complexity in large programs.

Of no less importance is generalization. While abstraction reduces complexity by hiding irrelevant detail, generalization reduces complexity by replacing multiple entities which perform similar functions with a single construct. Programming languages provide generalization through variables,
Abstraction and generalization are often used together. Abstracts are generalized through parameterization to provide greater utility. In parameterization, one or more parts of an entity are replaced with a name which is new to the entity. The name is used as a parameter. When the parameterized abstract is invoked, it is invoked with a binding of the parameter to an argument. Figure N.1 summarizes the notation which will be used for abstraction and generalization.

**Figure N.1: Abstraction and Generalization**

| Abstraction | name : abstract |
| Invocation | name |
| Substitution | E[p:a] (a replaces p in E) |
| Generalization | lambda p.E |
| Specialization | (lambda p.E a) = E[p:a] |
| Abstraction and generalization | name : lambda p.E name(p) : E name p : E |
| Invocation and specialization | (name a) name(a) |

When an abstraction is fully parameterized (all free variables bound to parameters) the abstraction may be understood without looking beyond the abstraction.

Abstraction and generalization depend on the principle of referential transparency.

**Principle of Referential Transparency** The meaning of an entity is unchanged when a part of the entity is replaced with an equal part.

**Abstraction**

**Principle of Abstraction** An abstract is a named entity which may be invoked by mentioning the name.

Giving an object a name gives permission to substitute the name for the thing named (or vice versa) without changing the meaning. We use the notation

\[ \text{name : abstract} \]
to denote the *binding* of a name to an abstract. Declarations and definitions are all instances of the use of abstraction in programming languages.

In addition to naming there is a second aspect to abstraction. It is that the abstract is encapsulated, that is, the details of the abstract are hidden so that the name is sufficient to represent the entity. This aspect of abstraction is considered in more detail in a later chapter.

An object is said to be **fully abstract** if it can be understood without reference to any thing external to the object.

**Terminology.** The naming aspect of abstraction is captured in the concepts of *binding*, *definition* and *declaration* while the hiding of irrelevant details is captured by the concept of encapsulation. A binding is an association of two entities. A definition is a binding of a name to an entity, a declaration is a definition which binds a name to a variable, and an assignment is a binding of a value and a variable.

We could equally well say *identifier* instead of name. A *variable* is an entity whose value is not fixed but may vary. Names are bound to variables in declaration statements. Among the various terms for abstracts found in other texts are *module*, *package*, *library*, *unit*, *subprogram*, *subroutine*, *routine*, *function*, *procedure*, *abstract type*, *object*.

**Binding**

The concept of binding is common to all programming languages. The objects which may be bound to names are called the bindables of the language. The bindables may include: primitive values, compound values, references to variables, types, and executable abstractions. While binding occurs in definitions and declarations, it also occurs at the virtual and hardware machine levels between values and storage locations.

**Aside.** The imperative programming paradigm is characterized by permitting names to be bound successively to different objects, this is accomplished by the assignment statement (often of the form; name := object) which means ``let name stand for object until further notice."'' In other words, until it is reassigned. This is in contrast with functional and logic programming paradigms in which names may not be reassigned. Thus languages in these paradigms are often called single assignment languages.

Typically the text of a program contains a number of bindings between names and objects and the bindings may be composed collaterally, sequentially or recursively.

A **collateral binding** is to perform the bindings independently of each other and then to combine the bindings to produce the completed set of bindings. Nether binding can reference a name used in any other binding. Collateral bindings are not very common but occur in Scheme and ML.
The most common way of composing bindings is sequentially. A **sequential binding** is to perform the bindings in the sequence in which they occur. The effect is to allow later bindings to use bindings produced earlier in the sequence. It must be noted that sequential bindings do not permit mutually recursive definitions.

In C/C++ and Pascal, constant, variable, and procedure and function bindings are sequential. To provide for mutually recursive definitions of functions and procedures, C/C++ and Pascal provide for the separation of the signature of a function or procedure from the body by the means of function prototypes & forward declarations so that so that mutually recursive definitions may be constructed.

A **recursive binding** is one in which the name being bound is used (directly or indirectly) in its own binding.

Programming languages that require "declaration before reference" have to invent special mechanisms to handle forward references. For dynamic data types, the rule is relaxed to permit the definition of pointer types. For functions and procedures, there are separate declarations for the signature of the function or procedure and its body. Pascal with its "forward" declarations and C++ with its function prototypes are typical.

The "declaration before reference" is often chosen to simplify the construction of the compiler. In Modula-3 and Java the choice has been made to simplify the programmer's task rather than the compiler's and permit forward references.

**Encapsulation**

The abstract part of a binding often contains other bindings which are said to be local definitions. Such local definitions are not visible or available to be referenced outside of the abstract. Thus the abstract part of a binding involves "information hiding". This hidden information is sometimes made available by exporting the names.

A module system provides a way of writing large programs so that the various pieces of the program don't interfere with one another because of name clashes and also provides a way of hiding implementation details. ... A module generally consists of two parts, the export part and the local part. The export part of a module consists of language declarations for the symbols available for use in either part of the module and in other modules which import them and module declaration giving the symbols from other modules which are available for use in either part of the module and in other modules which import them. The local part of a module consists of language declarations for the symbols available for use only in this part. *TGPL-Hill and Lloyd*

The work of constructing large programs is divided among several people, each of whom must produce a part of the whole. Each part is called a module and each programmer must be able to construct his/her
module without knowing the internal details of the other parts. This is only possible when each module is separated into an interface part and an implementation part. The interface part describes all the information required to use the module while the implementation part describes the implementation. This idea is already present in most programming languages in the manner in which functions and procedures are defined. Function and procedure definitions usually are separated into two parts. The first part gives the subprogram's name and parameter requirements and the second part describes the implementation. A module is a generalization of the concept of abstraction in that a module is permitted to contain a collection of definitions. An additional goal of modules is to confine changes to a few modules rather than throughout the program.

While the concept of modules is a useful abstraction, the full advantages of modules are gained only when modules may be written, compiled and possibly executed separately. In many cases modules should be able to be tested independently of other modules.

**EXPANDTHIS!!! Advantages**

- Reduction in complexity
- Team programming
- Maintainability
- Reusability of code
- Project management

**Implementation**

- Common storage area -- Fortran
- Include directive -- C++
- Subroutine library

**Typical applications:**

- Subroutine packages -- mathematical, statistical etc
- ADTs

**examples from Ada, C++, etc**

**Generalization**

**Principle of Generalization** A *generic* is an entity which may be specialized (elaborated) upon invocation.
Generalization permits the use of a single pattern to represent each member of a group. We use the notation: \textbf{lambda }p.B' (called a lambda abstraction) to denote the generalization of \textit{B} where \textit{p} is called a \textit{parameter} and \textit{B'} is \textit{B} with \textit{p} replacing any number of occurrences of some part of \textit{B} by \textit{p}. The parameter \textit{p} is said to be \textit{bound} in the expression but \textit{free} in \textit{B'} and the \textit{scope} of \textit{p} is said to be \textit{B'}. 

The symbol \textbf{lambda} is a \textit{quantifier}. Quantifiers are used to replace constants with variables.

The specialization (elaboration) of a generic is called application and takes the form:

\[(\text{lambda }p.B \ a)\]

It denotes the expression \textit{B'} obtained from the lambda expression when the free occurrences of \textit{p} in \textit{B} are replaced by \textit{a}.

\textbf{Aside.} The symbol \textbf{lambda} was introduced by Church for variable introduction in the lambda calculus. It roughly corresponds to the symbol forall, the universal quantifier, of first-order logic. The appendix contains a brief introduction to first-order logic. The functional programming chapter contains a brief introduction to the lambda calculus.

Generalization is often combined with abstraction and takes the following form:

\[n(p) : B\]

where \textit{p} is the name, \textit{x} is the parameter, and \textit{B} is the abstract. The invocation of the abstract takes the form:

\[n(a)\]

or occasionally \[(n \ a)\] where \textit{n} is the name and \textit{a} is called the \textit{argument} whose value is substituted for the parameter. Upon invocation of the abstract, the argument is bound to the parameter. Figure N.1 summarizes the variety of notation that is used to denote the elaboration of a generalization.

Most programming languages permit an implicit form of generalization in which variables may be introduced without providing for an invocation procedure which replaces the parameter with an argument. For example, consider the following psudocode for a program which computes the circumference of a circle:
Abstraction and Generalization

\[ \pi : 3.14 \]
\[ c : 2\pi r \]

begin
  \[ r := 5 \]
  write c
  \[ r := 20 \]
  write c
end

The value of \( r \) depends on the context in which the function is defined. The variable \( r \) is a global name and is said to be \textit{free}. In the first write command, the circumference is computed for a circle of radius 5 while in the second write command the circumference is computed for a circle of radius 20. The write commands cannot be understood without reference to both the definition of \( c \) and to the environment (\( \pi \) is viewed as a constant). Therefore, this program is not \textit{``fully abstract''}. In contrast, the following program is fully abstract:

\[ \pi : 3.14 \]
\[ c(r) : 2\pi r \]

begin
  FirstRadius := 5
  write c(FirstRadius)
  SecondRadius := 20
  write c(SecondRadius)
end

The principle of generalization depends on the analogy principle.

**Analogy Principle** When there is a conformation in pattern between two different objects, the objects may be replaced with a single object parameterized to permit the reconstruction of the original objects.

It is the analogy principle which permits the introduction of a variable to represent an arbitrary element of a class.

The Principle of Generalization makes no restrictions on parameters or the parts of an entity that may be parameterized. Neither should programming languages. This is emphasized in the following principle:

**Principle of Parameterization** A parameter of a generic may be from any domain.
Terminology. The terms *formal parameters (formals)* and *actual parameters (actuas)* are sometimes used instead of the terms parameters and arguments respectively.

Substitution

The utility of both abstraction and generalization depend on substitution. The tie between the two is captured in the following principle:

**Principle of Correspondence** Parameter binding mechanisms and definition mechanisms are equivalent.

The Principle of Correspondence is a formalization of that aspect of the Principle of Abstraction that implies that definition and substitution are intimately related.

We use the notation

\[ E[p:a] \]

to denote the substitution of \( a \) for \( p \) in \( E \). The notation is read as "\( E[p:a] \) is the expression obtained from \( E \) by replacing all free occurrences of \( p \) with \( a \)."

Terminology. The notation for substitution was chosen to emphasize the relationship between abstraction and substitution. Other texts use the notation \( E[p:=a] \) for substitution. Their notation is motivated by the assignment operation which assigns the value \( a \) to \( p \). Other texts use the notation \( E[a/p] \) for substitution. This latter notation is motivated by the cancelation that occurs when a number is multiplied by its inverse ( \( p(a/p) = a \)).

Together, abstraction, invokation, generalization and specialization provide powerful mechanisms for program development. Generalization provides a mechanism for the construction of common subexpressions and abstraction a mechanism for the factoring out of the common subexpressions. In the following example, the factors are first generalized to contain common subexpressions and then abstracted out. The product

\[ (a+b-c)*(x+y-z) \]

is formed from two very similar factors. The factors generalize to a common expression

**lambda** i j k. i+j-k.

The lambda expression can use to rewrite the product as:

\[ (\text{lambda} \ i \ j \ k. \ i+j-k) \ a \ b \ c \ * \ (\text{lambda} \ i \ j \ k. \ i+j-k) \ x \ y \ z. \]
The lambda expression can be abstracted to a name with three arguments,

\[ \text{f}(i \ j \ k) : i+j-k, \]

which can be used to replace the lambda expressions with the name and we get the expression

\[ \text{f}(a \ b \ c) \times \text{f}(x \ y \ z) \text{ where } \text{f}(i \ j \ k) : i+j-k \]

which clearly indicates the similarity of the factors.

- Partitions
- Separate compilation
  - Linking
  - Name and Type consistency
- Scope rules
  - Import
  - Export
- Modules--collection of objects--definitions
- Package

**Block structure**

A *block* is a construct that delimits the scope of any definitions that it may contain. It provides a local environment i.e., a opportunity for local definitions. The *block structure* (the textual relationship between blocks) of a programming language has a great deal of influence over program structure and modularity. There are three basic block structures--monolithic, flat and nested. In the discussion that follows, we will refer to the block structures found in Figure N.2.

---

**Figure M.N: Block Syntax**

\[
\text{let } \text{Definitions in } \text{Body end}
\]

\[
\text{Body where } \text{Definitions}
\]

Figure N.2 presents two styles of blocks, the first requires the definitions to proceed the body and the second requires definitions to follow the body.
Abstraction and Generalization

A program has a *monolithic* block structure if it consists of just one block. This structure is typical of BASIC and early versions of COBOL. The monolithic structure is suitable only for small programs. The scope of every definition is the entire program. Typically all definitions are grouped in one place even if they are used in different parts of the program.

Figure M.N: **Monolithic Block Structure**

Global Data

Return Address$_1$

...  

Return Address$_n$

A program has a *flat* block structure if it is partitioned into distinct blocks, an outer all inclosing block one or more inner blocks i.e., the body may contain additional blocks but the inner blocks may not contain blocks. This structure is typical of FORTRAN and C. In these languages, all subprograms (procedures and functions) are separate, and each acts as a block. Variables can be declared inside a subprogram are then local to that subprogram. Subprogram names are part of the outer block and thus their scope is the entire program along with global variables. All subprogram names and global variables must be unique. If a variable cannot be local to a subprogram then it must be global and accessible by all subprograms even though it is used in only a couple of subprograms.

Figure M.N: Flat Block Structure

A:  

B:  

N:  

...  

A program has *nested* block structure if blocks may be nested inside other blocks i.e., there is no restriction on the nesting of blocks within the body. This is typical of the block structure of the Algol-
Abstraction and Generalization

like languages. A block can be located close to the point of use. In blocks visibility is controlled by nesting. All names are visible (implicitly exported) to internally nested blocks. No names are visible (exported) to enclosing blocks. In a block, the only names visible are those that are declared in all enclosing blocks or are declared in the block, but not those declared in nested blocks.

**Figure M.N:** Nested blocks

- **A:**
  - Can see all names in A including the names A and B.
  - Cannot see names in B, C, or D.

- **B:**
  - Can see all names in A and B including the names A, B, and C.
  - Cannot see names in C or D.

- **C:**
  - Can see all names in A, B, and C including the names A, B, and C.
  - Cannot see names in D.

- **D:**
  - Can see all names in A and D including the names A, B, and D.
  - Cannot see names in B or C.

A *local name* is one that is declared within a block for use only within that block.

A *global name* is a name that when referenced within a block refers to a name declared outside the block.

An *activation* of a block is a time interval during which that block is being executed.

The three basic block structures are sufficient for what is called *programming in the small* (PITS). These are programs which are comprehensible in their entirety by an individual programmer. However, they are not general enough for very large programs. Large programs which are written by many individuals and which must consist of modules that can be developed and tested independently of other modules. Such programming is called *programming in the large* (PITL).
Activation Records

Each block Storage for local variables.

Scope Rules

The act of partitioning a program raises the issue of the scope of names. Which objects with in the partition are to be visible outside the partition? The usual solution is to designate some names to be exported and others to be private or local to the partition and invisible to other partitions. In case there might be name conflict between exported names from partitions, partitions are often permitted to designate names that are to be imported from designated partitions or to qualify the name with the partition name.

The scope rules for modules define relationships among the names within the partitions. There are four choices.

- All local names visible globally.
- All external names visible locally.
- Only local explicitly exported names visible globally.
- Only external names explicitly imported are visible locally.

Name conflict is resolved via qualification with the partition name.

Dynamic scope rules

A dynamic scope rule defines the dynamic scope of each association in terms of the dynamic course of program execution. Lisp.

implementation ease, cheap generalization for parameterless functions.

Static scope rules

Terminology. Static scope rules are also called lexical scope rules.

Cobol, BASIC, FORTRAN, Prolog, Lambda calculus, Scheme, Miranda, Algol-60, Pascal

Environment

An environment is a set of bindings.
Scope has to do with the range of visibility of names. For example, a national boundary may encapsulate a natural language. However, some words used within the boundary are not native words. They are words borrowed from some other language and are defined in that foreign language. So it is in a program. A definition introduces a name and a boundary (the object). The object may contain names for which there is no local definition (assuming definitions may be nested). These names are said to be free. The meaning assigned to these names is to be found outside of the definition. The rules followed in determining the meaning of these free names are called scope rules.

Scope It is concerned with name control.

ADTs

An even more effective approach is to separate the signatures of the operations from the bodies of the operations and the type representation so that the operation bodies and type representation can be compiled separately. This facilitates the development of software in that when an abstract data type's representation is changed (e.g. to improve performance) the changes are localized to the abstract data type.

```plaintext
name : adt
operation signatures
...

name : adt body
type representation definition
operation bodies
...
```

Pragmatics

Bindings and Binding Times

Bindings may occur at various times from the point of language definition through program execution. The time at which the binding occurs is termed the binding time. Four distinct binding times may be distinguished.

1. **Language design time.** Much of the structure of a programming language is fixed and language design time. Data types, data structures, command and expression forms, and program structure are examples of language features that are fixed at language design time. Most programming languages make provision for extending the language by providing for programmer defined data types, expressions and commands.

2. **Language implementation time.** Some language features are determined by the implementation.
Programs that run on one computer may not run or give incorrect results when run on another machine. This occurs when the hardware differs in its representation of numbers and arithmetic operations. For example, the \textit{maxint} of Pascal is determined by the implementation. The C programming language provides access to the underlying machine and therefore programs which depend on the characteristics of the underlying machine may not perform as expected when moved to another machine.

3. \textit{Program translation time}. The binding between the source code and the object code occurs at program translation time. Programmer defined variables and types are another example of bindings that occur at program translation time.

4. \textit{Program execution time}. Binding of values to variables and formal parameters to actual parameters occur during program execution.

Early binding often permits more efficient execution of programs though translation time type checking while late binding permits more flexibility through program modification a run-time. The implementation of recursion may require allocation of memory at run-time in contrast a one time to allocation of memory at compile-time. An Example from Pratt 1984:

\begin{verbatim}
X := X + 10
\end{verbatim}

1. Set of possible data types for X (Language design time: Fortran; Translation time: C, Pascal (user defined))
2. Type of variable X (Translation time: C; Execution time: Lisp, APL)
3. Set of possible values for X (Language implementation time (often constrained by hardware))
4. Value of the variable X (Execution time (assignment))
5. Representation of the constant 10 (language definition time (base 10); language implementation time (base 2)).
6. Properties of the operator + (language definition time - addition operations; translation time - type of addition; execution-time - APL)

\section*{Procedures and Functions}

\textbf{THISSHOULDBEGENERALIZETOINCLUDEOTHERABSTRACTIONS!}

In the discussion which follows, the term \textit{subprogram} will be used to refer to whole programs, procedures and functions.

A program may be composed of a main program which during execution may call subprograms which in turn may call other subprograms and so on. When a subprogram is called, the calling subprogram waits for the called subprogram to terminate. Each subprogram is expected to eventually terminate and return control to the calling subprogram. The execution of the calling subprogram resumes at the point immediately following the point of call. Each subprogram may have its own local data which is found in an \textit{activation record}. An activation record consists of an association between variables and the value to which they are assigned. An activation record may be created each time a subprogram is called and
Abstraction and Generalization

destroyed when the subprogram terminates.

DYNAMIC VS. STATIC ALLOCATION

The runtime environment must keep track of the current instruction and the referencing environment for each active or waiting program so that when a subprogram terminates, the proper instruction and data environment may be selected for the calling subprogram.

The current instruction of the calling subprogram is maintained on a stack. When a subprogram is called, the address of the instruction following the call of the calling program is pushed on the stack. When a subprogram terminates, the instruction pointer is set to the address on the top of the stack and the address popped off the stack. The stack is often called the return address stack.

Figure M.N: Return Address Stack

Return Address₁
...
Return Addressₙ

The addresses of the current environment is also maintained on a stack. The top of the stack always points to the current environment. When a subprogram is called, the address of the new environment is pushed on the stack. When a subprogram terminates, the stack is popped revealing the previous environment. The stack is often called the dynamic links because the stack contains links (pointers) which reveal the dynamic history of the program.

When a programming language does not permit recursive procedures and data structure size is independent of computed or input data, the maximum storage requirements of the program can be determined at compile time. This simplifies the run time support required by the program and it is possible to statically allocate the storage used during program execution.

Parameters and Arguments

Strict, Non-strict, Eager and Lazy

An generic is said to be strict in a parameter if it is sure to need the value of the parameter and non-strict in a parameter if it may not require the value of the parameter. Arithmetic operators are strict
because their arguments must be evaluated to determine the value of the arithmetic expression but the conditional expression

\[ \text{if } B \text{ then } E_1 \text{ else } E_2 \]

is not strict in its second and third arguments since the selection of the second or third argument is dependent on the value of the boolean condition (the first argument).

Most programming languages assume that abstracts are strict in their parameters and, therefore, the parameters are evaluated when the function is called. This evaluation scheme is called \textit{eager evaluation}. This is not always desirable and so some languages provide a mechanism for the programmer to inform a function not to evaluate its parameters. Scheme provides for the quote operator to prevent the evaluation of an argument. Logic languages like Prolog and functional languages languages like Haskell and Miranda are non-strict languages and the arguments are evaluated only when the value is required. This evaluation scheme is called \textit{normal-order evaluation} and is often implemented using \textit{lazy evaluation} (the argument is evaluated only when it is first needed).

Most languages use strict evaluation because it is more efficient and simplifies the implementation of parameter passing for imperative programming languages. Normal-order evaluation coupled with side-effects found in imperative languages produces unexpected results. Algol-60's provides a parameter passing mechanism (pass by name) which is based on that does not provide the generality that is required in the imperative model as the following example shows.

\textbf{Figure M.N:} Algol-60, Jensen's device

```plaintext
procedure swap(x,y:sometype);
var t:sometype
begin
  t := x; x := y; y := t
end;
...
I := 1
a[I] := 3
swap(I,a[I])
```

Based on the code in the body of the procedure, it would seem that the values of the arguments would be swapped. That this is not the case is easily seen when the formal parameters are textually replaced with the actual parameters and the resulting code is executed in the context of the actual parameters. In this case, prior to the call to sort, \( I \) is 1 and \( A[i] \) is 3. Upon textual substitution, we have
Abstraction and Generalization

...  
I := 1  
{I = 1}  
a[I] := 3  
{I=1, a[I] = a[1] = 3}  
t := I; I := a[I]; a[I] := t -- replaces call to swap  
{T=1, I=3, a[i] = a[3] = 1, a[1] = 3}  

After execution, I is 3 and a[1] is still 3, but a[3] is now 1.

Argument Passing Mechanisms

In the previous chapter (Abstraction and Generalization), it appears that when an argument is passed to an abstract, it replaces the parameter, that is, it textually replaces the parameter. If the argument is large, the space and time requirements can be a significant overhead. Especially since the each time the argument is referenced, it must be evaluated not in the internal (local) environment of the abstract but in the environment external to (global) the abstract. This need not be the case and several mechanisms have been developed to make passing arguments simpler and more efficient.

The copy mechanism requires values to be copied into an generic when it is entered and copied out of the generic when the generic is exited. This form of parameter passing is often referred to as passing by value. The formal parameters are local variables and the argument is copied into the local variable on entry to the generic and copied out of the local variable to the argument on exit from the generic.

The value parameter of Pascal and the in parameter of Ada are examples of parameters which may be passed by using the copy mechanism. The value of the argument is copied into the parameter on entry but the value of the formal parameter is not copied to the actual parameter on exit. In imperative languages, copying is unnecessary if the language prohibits assignment to the formal parameter. In such a case, the parameter may be passed by reference.

Ada's out parameter and function results are examples of parameters which may be passed by using the copy mechanism. The value of the argument (actual parameter) is not copied into the formal parameter on entry but the value of the parameter is copied into the argument upon exit. In Pascal the function name is used as the parameter and assignments may be made to the function name. This form of parameter passing is often referred to as passing by result.

When the passing by value and result are combined, the passing mechanism is referred to as passing by value-result. Ada's in out parameter is an example of a parameter which may be passed by this form of the copy mechanism. The value of the actual parameter is copied into the formal parameter on entry and the value of the formal parameter is copied into the actual parameter upon exit.

The copy mechanism has some disadvantages. The copying of large composite values (arrays etc) is
expensive and the parameters must be assignable (e.g. expressions and file types in Pascal are not assignable).

The effect of a **definitional mechanism** is as if the abstract were surrounded by a block, in which there is a definition that binds the parameter to the argument.

Parameter : Argument

An parameter is said to be passed by **reference** if the argument is an address. References to the parameter are references to the argument. Assignments to the parameter are assignments to the argument. The **reference parameter** of Pascal and the array and structure parameters of C++ are passed using this mechanism. A toster provides an illustration of the effect of passing by reference.

If an argument --- A parameter is said to be passed by **name** if, in effect, the argument replaces parameter throughout the body of the subroutine (textual substitution with suitable renaming of local variables to avoid conflicts between local variables and variables occurring in the argument) i.e., in the subprogram, each reference to the parameter results in an evaluation of the argument in the calling environment.

In addition to the problems of the pass-by-name mechanism of Algol-60, imperative languages with reference parameters present the possibility of **aliasing**. Aliasing occurs when two or more names reference the same object. For example, the following procedure and call,

```plaintext
procedure confuse (var m, n : Integer );
begin
   n := 1; n := m + n
   end;
...
i := 5;
confuse(i,i)
```

both m and n are bound to the same variable, i, and i is initially 5, then after the call to the procedure, the value of i is 2 not 6. m and n are both bound to i and after the assignment n := 1, the value of m is also 1.

**Scope and Blocks**

A variables declared within a block have a **lifetime** which extends from the moment an activation record is created for the block until the activation record for the block is destroyed. A variable is bound to an offset within the activation record at compile time. It is bound to a specific storage location when the block is activated and becomes a part of storage.

**STATIC/DYNAMIC ALLOCATION**
**Data Access**

block structure, COMMON, ADT's, aliasing

**Scope Rules**

Conceptually the **dynamic scope rules** may be implemented as follows. Each variable is assigned a stack to hold the current values of the variable.

When a subprogram is called, a new uninitialized stack element is pushed on the stack corresponding to each variable in the block.

A reference to a variable involves the inspection or updating of the top element of the appropriate stack. This provides access to the variable in closest block with respect to the dynamic calling sequence.

When a subprogram terminates, the stacks corresponding to the variables of the block are popped, restoring the calling environment. The **static scope rules** may be implemented as follows. The data section of each procedure is associated with an *activation record*. The activation records are dynamically allocated space on a *runtime* stack. Each recursive call is associated with its own activation record. Associated with each activation record is a *dynamic link* which points to the previous activation records, a *return address* which is the address of the instruction to be executed upon return from the procedure and a *static link* which provides access to the referencing environment.

An activation record consists of storage for local variables, the static and dynamic links and the return address.

---

**Figure M.N: Activation Record**

<table>
<thead>
<tr>
<th>Static Link</th>
<th>Return Address</th>
<th>Dynamic Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The runtime stack of activation records (local data, static and dynamic links).
Abstraction and Generalization

**Figure M.N: Runtime Stack**

<table>
<thead>
<tr>
<th>Main</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Local data for M</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>SL</th>
<th>RA</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Local data for A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>SL</th>
<th>RA</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Local data for B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Main calls A which calls B.

Global data values are found by following the static chain to the appropriate activation record.

An alternative method for the implementation of static scope rules is the display. A *display* is a set of registers (in hardware or software) which contain pointers to the current environment. On procedure call, the current display is pushed onto the runtime stack and a new display is constructed containing the revised environment. On procedure exit, the display is restored from the copy on the stack.

**Partitions**

A *partition* of a set is a collection of disjoint sets whose union is the set.

There are a number of mechanisms for partitioning program text. Functions and procedures are among the most common. However, the result is still a single file. When the partitions of program text are arranged in separate files, the partitions are called modules. Here are several program partitioning mechanisms.

- Separate declaration of data and code
- Procedures
- Functions
- ADTs
Abstraction and Generalization

- Modules

Partitioning of program text is desirable to provide for separate compilation and for pipeline processing of data.

There are a number of mechanisms for combining the partitions into a single program for the purposes of compilation and execution. The include statement is provided in a number of languages. It is a compiler directive with directs the compiler to textually include the named file in the source program. In some systems the partitions may be separately compiled and there is a linking phase in which the compiled program modules are linked together for execution. In other systems, at run-time any missing function or procedure results in a run-time search for the missing module which if found is then executed or if not found results in a run-time error.

**Modules**

A *module* is a program unit which is an (more or less) independent entity. A module consists of a number of definitions (of types, variables, functions, procedures and so on), with a clearly defined interface stating what it exports to other modules which use it. Modules have a number of advantages for the construction of large programs.

- Modules facilitate parallel and independent development
- Modules facilitate separate compilation
- Modules facilitate code reuse

Modules are used to construct libraries, ADTs, classes, interfaces, and implementations. A module is the compilation unit. A module which contains only type abstractions is a specification or interface module.

In program construction the module designer must answer the following questions.

- *What* is the module's purpose?
- *How* does it achieve that purpose?

*Programming in the large* is concerned with programs that are not comprehensible by a single individual and are developed by teams of programmers. At this level programs must consist of modules that can be written, compiled, and tested independently of other modules. A module has a single purpose, and has a narrow interface to other modules. It is likely to be reusable (able to be incorporated into many programs) and modifiable without forcing changes in other modules.

Modules must provide answers to two questions:
Abstraction and Generalization

- **What** is the purpose of the module?
- **How** does it achieve that purpose?

The **what** is of concern to the user of the module while the **how** is of concern to the implementer of the module.

Functions and procedures are simple modules. Their signature is a description of **what** they do while their body describes **how** it is achieved. More typically a module **encapsulates** a group of components such as types, constants, variables, procedures, functions and so on.

To present a narrow interface to other modules, a module makes only a few components visible outside. Such components are said to be **exported** by the module. The other components are said to be **hidden** inside the module. The hidden components are used to implement the exported components.

Access to the components is often by a qualified name -- **module name**. **component name**. When strong safety considerations are important, modules using components of another module may be required to explicitly **import** the required module and the desired components.

**Historical Perspectives and Further Reading**

**Exercises**

1. [time/difficulty] (section) Problem statement
2. Extend the compiler to handle constant, type, variable, function and procedure definitions and references to the same.
3. What is the effect of providing lazy evaluation in an imperative programming language? % Ans: Due to the presence of side effects, the value of the actual % parameter may be different between the point of entry and the point % of evaluation.
4. Extend the compiler to handle parameterization of functions and procedures.
5. For a specific programming language, report on its provision for abstraction and generalization. Specifically, what entities may be named, what entities may be parameterized, what entities may be passed as parameters, and what entities may be returned as results (of functions). What irregularities do you find in the language?
6. Algebraic Semantics: stack, tree, queue, grade book etc
7. Lexical Scope Rules
8. Dynamic Scope Rules
9. Parameter Passing
10. Run-time Stack

© 1996 by A. Aaby
Values, Domains and Types

A value is any thing that may be evaluated, stored, incorporated in a data structure, passed as an argument or returned as a result.

What is a type?

- **Realist:** A type is a set of values.
- **Idealist:** No. A type is a conceptual entity whose values are accessible only through the interpretive filter of type.
- **Beginning Programmer:** Isn't a type a name for a set of values?
- **Intermediate Programmer:** A type is a set of values and operations.
- **Advanced Programmer:** A type is a way to classify values by their properties and behavior.
- **Algebraist:** Ah! So a type is an algebra, a set of values and operations defined on the values.
- **Type checker:** Types are more practical than that, they are constraints on expressions to ensure compatibility between operators and their operand(s).
- **Type Inference System:** Yes and more, since a type system is a set of rules for associating with every expression a unique and most general type that reflects the set of all meaningful contexts in which the expression may occur.
- **Program verifier:** Lets keep it simple, types are the behavioral invariants that instances of the type must satisfy.
- **Software engineer:** What is important to me is that types are a tool for managing software development and evolution.
- **Compiler:** All this talk confuses me, types specify the storage requirements for variables of the type.

**Keywords and phrases:** value, domain, type, type constructor, Cartesian product, disjoint union, map, power set, recursive type, binding, strong and weak typing, static and dynamic type checking, type inference, type equivalence, name and structural equivalence, abstract types, generic types, block, garbage collection, static and dynamic links, display, static and dynamic binding, activation record, environment, Static and Dynamic Scope, aliasing, variables, value, result, value-result, reference, name, unification, eager evaluation, lazy evaluation, strict, non-strict, Church-Rosser, overloading, polymorphism, monomorphism, coercion, transfer functions.

A computation is a sequence of operations applied to a value to yield a value. Thus values and operations are fundamental to computation. Values are the subject of this chapter and operations are the subject of later chapters.
Types

In mathematical terminology, the sets from which the arguments and results of a function are taken are known as the function's ``domain'' and ``codomain'', respectively. Consequently, the term domain will denote any set of values that can be passed as arguments or returned as results. Associated with every domain are certain ``essential'' operations. For example, the domain of natural numbers is equipped with an the ``constant'' operation which produces the number zero and the operation that constructs the successor of any number. Additional operations (such as addition and multiplication) on the natural numbers may be defined using these basic operations.

Programming languages utilize a rich set of domains. Truth values, characters, integers, reals, records, arrays, sets, files, pointers, procedure and function abstractions, environments, commands, and definitions are but some of the domains that are found in programming languages. There are two approaches to domains. One approach is to assume the existence of a universal domain. It contains all those objects which are of computational interest. The second approach is to begin with a small set of values and some rules for combining the values and then to construct the universe of values. Programming languages follow the second approach by providing several basic sets of values and a set of domain constructors from which additional domains may be constructed.

Domains are categorized as primitive or compound. A primitive domain is a set that is fundamental to the application being studied. Its elements are atomic. A compound domain is a set whose values are constructed from existing domains by one or more domain constructors.

Aside. It is common in mathematics to define a set but fail to give an effective method for determining membership in the set. Computer science on the other hand is concerned with determining membership with in a finite number of steps. In addition, a program is often constrained by requirements to complete its work within bounds of time and space. % In computer science ... streams ... infinite sequences ... % halting problem ... robust

Terminology. Domain theory is the study of structured sets and their operations. A domain is a set of elements and an accompanying set of operations defined on the domain.

The terms domain, type, and data type may be used interchangeably.

The term data refers to either an element of a domain or a collection of elements from one or more domains.

The terms compound, composite and structured when applied to values, data, domains, types are used interchangeably.

Elements of Domain Theory

There are many compound domains that are useful in computer science: arrays, tuples, records, variants, unions, sets, lists, trees, files, relations, definitions, mappings, etc., are all examples of compound
domains. Each of these domains may be constructed from simpler domains by one or more applications of domain constructors.

Compound domains are constructed by a domain builder. A domain builder consists of a set of operations for assembling and disassembling elements of a compound domain. The domain builders are:

- Product Domains
- Sum Domains
- Function Domains
- Power Domains
- Recursive Domains

**Product Domain**

The domains constructed by the *product* domain builder are called *tuples* in ML, *records* in Cobol, Pascal and Ada, and *structures* in C and C++. Product domains form the basis for relational databases and logic programming.

In the binary case, the product domain builder, $\times$, builds the domain $A \times B$ from domains $A$ and $B$. The domain builder includes the assembly operation, ordered pair builder, and a set of disassembly operations called projection functions. The assembly operation, ordered pair builder, is defined as follows:

if $a$ is an element of $A$ and $b$ is an element of $B$ then $(a, b)$ is an element of $A \times B$. That is,

$$A \times B = \{ (a,b) \mid a \in A, b \in B \}$$

The disassembly operations $fst$ and $snd$ are projection functions which extract elements from tuples. For example, $fst$ extracts the first component and $snd$ extracts the second element.

$$nsnd(a,b) = b$$

The product domain is easily generalized (see Figure N.1) to construct the product of an arbitrary number of domains.

**Figure N.1: Product Domain: $D_0 \times \ldots \times D_n$**

Assembly operation: $(a_0,\ldots,a_n)$ in $D_0 \times \ldots \times D_n$ where $a_i$ in $D_i$ and

$$D_0 \times \ldots \times D_n = \{ (a_0,\ldots,a_n) \mid a_i \in D_i \}$$

Disassembly operation: $(a_0,\ldots,a_n)|_i = a_i$ for $0 \leq i \leq n$
Both relational data bases and logic programming paradigm (Prolog) are based on programming with tuples.

Elements of product domains are usually implemented as a contiguous block of storage in which the components are stored in sequence. Component selection is determined by an offset from the address of the first storage unit of the storage block. An alternate implementation (possibly required in functional or logic programming languages) is to implement the value as a list of values. Component selection utilizes the available list operations.

**Terminology.** The product domain is also called the "Cartesian" or "cross" product. In Pascal it is called a record and in C a structure.

**Implementation.**

Product domain elements are usually implemented as contiguous locations in memory. Using the notation introduced in the Introduction,

<table>
<thead>
<tr>
<th>Product Descriptor</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Descriptor₁</td>
<td>value₁</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Descriptorₙ</td>
<td>valueₙ</td>
</tr>
</tbody>
</table>

**Sum Domain**

Domains constructed by the sum domain builder are called *variant records* in Pascal and Ada, *unions* in Algol-68, *constructions* in ML and *algebraic types* in Miranda.

In the binary case, the sum domain builder, +, builds the domain $A + B$ from domains $A$ and $B$. The domain builder includes a pair of assembly operations and a disassembly operation. The two assembly operations of the sum builder are defined as follows:

if $a$ is an element of $A$ and $b$ is an element of $B$ then $(A,a)$ and $(B, b)$ are elements of $A + B$
B. That is,
\[ A + B = \{ (A,a) \mid a \text{ in } A \} \cup \{ (B,b) \mid b \text{ in } B \} \]

where the A and B are called tags and are used to distinguish between the elements contributed by A and the elements contributed by B.

The disassembly operation returns the element iff the tag matches the request.
\[ A(A,a) = a \]

The sum domain differs from ordinary set union in that the elements of the union are labeled with the parent set. Thus even when two sets contain the same element, the sum domain builder tags them differently.

The sum domain generalizes (see Figure N.2) to sums of an arbitrary number of domains.

---

**Figure N.2: Sum Domain:** $D_0 + ... + D_n$

Assembly operations: $(D_i, d_i)$ in $D_0 + ... + D_n$ and $D_0 + ... + D_n = \text{Union}_{i=0}^n \{ (D_i,d) \mid d \text{ in } D_i \}$

Disassembly operations: $D_i(D_i, d_i) = d_i$

---

**Terminology.** The *sum* domain is also called the *disjoint union* or *co-product* domains. In Pascal it is called a variant record and in C a union.

Pascal it is called a record and in C a structure.

**Implementation.**

Sum domain elements are usually implemented as a contiguous piece of memory large enough to hold a value of any of the domains and a tag which is used to determine the domain to which the value belongs. Using the notation introduced in the Introduction,
Function Domain

The domains constructed by the function domain builder are called functions in Haskell, procedures in Modula-3, and procs in SR. Although their syntax often differs from that of functions, arrays are also examples of domains constructed by the function domain builder.

The function domain builder creates the domain $A \rightarrow B$ from the domains $A$ and $B$. The domain $A \rightarrow B$ consists of all the functions from $A$ to $B$. $A$ is called the domain and $B$ is called the co-domain.

The assembly operation is:

$$(\text{lambda } x.e) \text{ is an element in } A \rightarrow B \text{ whenever } e \text{ is an expression containing occurrences of an identifier } x, \text{ such that whenever a value } a \text{ in } A \text{ replaces the occurrences of } x \text{ in } e, \text{ the value } e[a:x] \text{ in } B \text{ results, then.}$$

The disassembly operation is function application. It takes two arguments, an element $f$ of $A \rightarrow B$ and an element $a$ of $A$ and produces $f(a)$ an element of $B$. In the case of arrays, the disassembly operation is called subscripting.

The function domain is summarized in Figure N.3.

Figure N.3: Function Domain: $A \rightarrow B$

Assembly operation: $(\text{lambda } x.E) \text{ in } A \rightarrow B \text{ where for all } a \text{ in } A, E[x:a] \text{ is a unique value in } B.$

Disassembly operation: $(g \ a) \text{ in } B, \text{ for } g \text{ in } A \rightarrow B \text{ and } a\text{ in } A.$

Mappings (or functions) from one set to another are an extremely important compositional method. The map $m$ from a element $x$ of $S$ (called the domain) to the corresponding element $m(x)$ of $T$ (called the range) is written as:

$$m : S \rightarrow T$$
where if \( m(x) = a \) and \( m(y) = a \) then \( x = y \). Mappings are more restricted than the Cartesian product since, for each element of the domain there is a unique range element. Often it is either difficult to specify the domain of a function or an implementation does not support the full domain or range of a function. In such cases the function is said to be a *partial function*. It is for efficiency purposes that partial functions are permitted and it becomes the programmer's responsibility to inform the users of the program of the nature of the unreliability.

Arrays are mappings from an index set to an array element type. An array is a finite mapping. Apart from arrays, mappings occur as operations and function abstractions. Array values are implemented by allocating a contiguous block of storage where the size of the block is based on the product of the size of an element of the array and the number of elements in the array.

The operations provided for the primitive types are maps. For example, the addition operation is a mapping from the Cartesian product of numbers to numbers.

\[ +: \text{number } \times \text{number} \rightarrow \text{number} \]

The functional programming paradigm is based on programming with maps.

**Terminology.** The function domain is also called the *function space*.

**Implementation.**

Function domain elements are usually implemented in code. However, arrays are a special case of function domain and they are usually implemented in contiguous memory elements. Using the notation introduced in the Introduction,

![Function Descriptor](image)

**Power Domain**

Set theory provides an elegant notation for the description of computation. However, it is difficult to provide efficient implementation of the set operations. SetL is a programming language based on sets and was used to provide an early compiler for Ada. The Pascal family of languages provide for set union and intersection and set membership. Set variables represent subsets of user defined sets.

The set of all subsets of a set is the power set and is defined:
Subtypes and subranges are examples of the power set constructor.

Functions are subsets of product domains. For example, the square function can be represented as a subset of the product domain $\text{Nat} \times \text{Nat}$.

\[
sqr = \{(0,0),(1,1),(2,4),(3,9),\ldots\}
\]

Generalization helps to simplify this infinite list to:

\[
sqr = \{(x,x^2) \mid x \in \text{Nat}\}
\]

The programming language SetL is based on computing with sets.

Set values may be implemented by using the underlying hardware for bit-strings. This makes set operations efficient but constrains the size of sets to the number of bits (typically) in a word of storage. Alternatively, set values may be implemented using software, in which case, hash-coding or lists may be used.

Some languages provide mechanisms for decomposing a type into subtypes

- one is the enumeration of the elements of the subtype.
- another is subranges since, enumeration is tedious for large sub-domains and many types have a natural ordering.

The power domain construction builds a domain of sets of elements. For a domain $A$, the power domain builder $P()$ creates the domain $P(A)$, a collection whose members are subsets of $A$.

---

Figure N.4: **Power Domain: $P^D$**

Assembly operations: $\emptyset$ in $P^D$, $\{a\}$ in $P^D$ for $a$ in $D$, and $S_i$ union $S_j$ in $P^D$ for $S_i$, $S_j$ in $P^D$
The definition is called recursive because the name of the domain "recurs" on the right hand side of the definition. Recursively defined domains depend on abstraction since the name of the domain is an essential part the definition of the domain. The context-free grammars used in the definition of programming languages contain recursive definitions so programming languages are examples of recursive types.

More than one set may satisfy a recursive definition. However, it may be shown that a recursive definition always has a least solution. The least solution is a subset of every other solution.

The least solution of a recursively defined domain is obtained through a sequence of approximations \((D_0, D_1, \ldots)\) to the domain with the domain being the limit of the sequence of approximations \(D = \lim_{i \to \infty} D_i\). The limit is the smallest solution to the recursive domain definition.

We illustrate the limit construction (see Figure N.5) with three examples.

---

**Figure N.5: Limit Construction**

- \(D_0 = \text{null}\)
- \(D_{i+1} = e[D;D_i]\) for \(i=0,\ldots\)
- \(D = \lim_{i \to \infty} D_i\)

---

**The Natural Numbers**

A representation of the natural numbers given earlier in the text was:

\[ N ::= 0 \mid S(N) \]

The defining sequence for the natural numbers is:

\[ N_0 = \text{Null} \]
\[ N_{i+1} = 0 \mid S(N_i) \text{ for } i = 0,\ldots \]

The definition results in the following:
The factorial function

For functions, Null can be replaced with \( \_|_ \) which means undefined.

The factorial function is often recursively defined as:

\[
\text{fac}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
n \times \text{fac}(n-1) & \text{otherwise}
\end{cases}
\]

The factorial function is approximated by a sequence of functions where the function fac\(_0\) is defined as

\[
\text{fac}_0(n) = \_|_
\]

And the function fac\(_{i+1}\) is defined as

\[
\text{fac}_{i+1}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
n \times \text{fac}_i(n-1) & \text{otherwise}
\end{cases}
\]

Writing the functions as sets of ordered pairs helps us to understand the limit construction.

\[
\text{fac}_0 = \{ \}
\]
\[
\text{fac}_1 = \{ (0,1) \}
\]
\[
\text{fac}_2 = \{ (0,1), (1,1) \}
\]
\[
\text{fac}_3 = \{ (0,1), (1,1), (2,2) \}
\]
\[
\text{fac}_4 = \{ (0,1), (1,1), (2,2), (3,6) \}
\]

Note that each function in the sequence includes the previously defined function and the sequence suggests that
The proof of this last equation is beyond the scope of this text. This construction suggests that recursive definitions can be understood in terms of a family of non-recursive definitions and in format common to each member of the family.

**Ancestors**

For logical predicates, Null can be replaced with \textit{false}. A recursive definition of the ancestor relation is:

\[
\text{ancestor}(A,D), \text{ if } \text{parent}(A,D) \text{ or } \\
\text{parent}(A,I) \text{ and ancestor}(I,D)
\]

The ancestor relation is approximated by a sequence of relations:

\[
\text{ancestor}_0(A,D) = \text{false}
\]

And the relation \text{ancestor}_i is defined as

\[
\text{ancestor}_{i+1}(A,D), \text{ if } \text{parent}(A,D) \text{ or } \\
\text{parent}(A,I) \text{ and ancestor}_i(I,D)
\]

Writing the relations as sets of order pairs helps us to understand the limit construction. An example will help. Suppose we have the following:

\[
\text{parent}( \text{John}, \text{Mary} ) \\
\text{parent}( \text{Mary}, \text{James} ) \\
\text{parent}( \text{James}, \text{Alice} )
\]

then we have:

\[
\text{ancestor}_0 = \{ \} \\
\text{ancestor}_1 = \{ (\text{John}, \text{Mary}), (\text{Mary}, \text{James}), (\text{James}, \text{Alice}) \} \\
\text{ancestor}_2 = \text{ancestor}_1 \text{ union } \{ (\text{John}, \text{James}), (\text{Mary}, \text{Alice}) \} \\
\text{ancestor}_3 = \text{ancestor}_2 \text{ union } \{ (\text{John,Alice}) \}
\]

Again note that each predicate in the sequence includes the previously defined predicate and the sequence suggests that

\[
\text{ancestor} = \lim_{i \rightarrow \infty} \text{ancestor}_i
\]

**Linear Search**
The final example of domain construction is a recursive variant of linear search.

\begin{verbatim}
Loop : if i < n --> if a[i] != target --> i := i + 1; Loop
   fi
fi
\end{verbatim}

Loop₀ is defined as:

```
Loop₀ = _|_
```

and Loopᵢ₊₁ is defined as:

```
Loopᵢ₊₁ : if i < n --> if a[i] != target --> i := i + 1;
   Loopᵢ
   fi
fi
```

with the result of unrolling the recursion into a sequence of if-commands.

**Implementation**

Since recursively defined domains like lists, stacks and trees are unbounded (in general may be infinite objects) they are implemented using a product domain where one domain is a node and one or more are address domains. In Pascal, Ada and C such domains are defined in terms of pointers while Prolog and functional languages like ML and Miranda allow recursive types to be defined directly.

**Type Systems**

A large percentage of errors in programs is due to the application of operations to objects of incompatible types. Type systems have been developed to assist the programmer in the detection of these errors. A *type system* is a set of rules for defining types and associating a type with expression in the language. A type system *rejects* an expression if it does not associate a type with the expression. Type checking may performed at compile time or run time or both.

---

**Definition N.M: Type System**

A *type system* is a set of rules for defining types and associating a type with expression in the language. A type system *rejects* an expression if it does not associate a type with the expression.
If the errors are to be detected at compile time then a *static* type checking system is required. One approach to static type checking is to require the programmer to specify the type of each object in the program. This permits the compiler to perform type checking before the execution of the program and this is the approach taken by languages like Pascal, Ada, C++, and Java. Another approach to static type checking is to add type inference capabilities to the compiler. In such a system, the compiler performs type checking by means of a set of type inference rules and is able to flag type errors prior to runtime. This is the approach taken by Miranda and Haskell.

If the error detection is to be delayed until execution time, then *dynamic* type checking is required. In dynamic type checking, each data value is tagged with type information so that the run time environment can check type compatibility and possibly perform type conversions if necessary. The programming languages Lisp, Scheme and Small-talk are examples of dynamically typed languages.

**Type Checking**

Machine operations manipulate bit patterns. Whether a bit pattern represents a character, an integer, a real, an address, or an instruction, any machine operation may be applied to any data item. There is no type checking at the assembly language level. Languages which permit operations to be applied to data of any type are called *untyped*. Prolog is one of the few high-level languages that is an untyped language. In Prolog, lists can consist of elements of any type and different sorts of values may be compared with the equality relation `=' but such comparison will always yield *false*.

**Example.** In C, the decision portion of any control structure can be any expression that produces a value. If the value is 0, it is treated as false and any nonzero values is treated as true. Since the value of an assignment command is the value of its right-hand side, the command `if x = 4 ...` any else clause will be ignored. In C characters are treated as integers and thus may occur in arithmetic expressions. C's type system is not robust enough to protect novice programmers these and other errors.

The advantage of untyped languages is their flexibility. The programmer has complete control over how a data value is used but must assume full responsibility for detecting the application of operations to objects of incompatible type.

**Figure N.M: Type Checking Definitions**

A language is said to be
Types

- **untyped** if no type abstractions are inforced,
- **strongly typed** if it enforces type abstractions (operations may be applied only to objects of the appropriate type),
- **statically typed** if the type of each expression can be determined from the program text,
- **dynamically typed** if the determination of the type of some expression depends on the runtime behavior of the program.

A **strongly typed** language enforces type abstractions. Most languages are strongly typed with respect to the primitive types supported by the language. So, for example, the mixing of numeric and character types that is permissible in C is not permitted in Pascal or Ada.

Strong typing helps to insure the security and portability of the code and it often requires the programmer to explicitly define the types of each object in a program. It is also important in compilation for picking appropriate operations and for optimization.

If the types of all variables can be known from an examination of the text (i.e. at compile time), then the language is said to be **statically typed**. Pascal, Ada, and Haskell are examples of statically typed languages.

Static typing is widely recognized as a requirement for the production of safe and reliable software. Static type checking implies that the types are checked at compile time. Static typing is chosen when efficiency in execution time is important and compiler support is used to support good software engineering practices.

If the type of a variable can only be known at run-time, then the language is said to be **dynamically typed**. Lisp and Smalltalk are examples of dynamically typed languages.

Dynamic type checking implies that the types are checked at execution time and that every value is tagged to identify its type in order to make the type checking possible. The penalty for dynamic type checking is additional space and time overheads.

Dynamic typing is often justified on the assumption that its flexibility permits the rapid prototyping of software.

Prolog relies on pattern matching to provide a semblance of type checking. There is active research on adapting type checking systems for Prolog.

Modern functional programming languages such as Miranda and Haskell and object-oriented languages combine the safety of static type checking with the flexibility of dynamic type checking through...
polymorphic types.

**Type Equivalence**

Two unnamed types (sets of objects) are the same if they contain the same elements. The same cannot be said of named types for if they were, then there would be no need for the disjoint union type. When types are named, there are two major approaches to determining whether two types are equal.

**Name Equivalence**

In name equivalence two types are the same if they have the same name. Types that are given different names are treated as distinct and cannot be accidentally mixed just because their structure happens to be the same. Name equivalence requires type definitions to be global.

Name equivalence was chosen for Modula-2, Ada, C (for records), and Miranda. The predecessor of Modula-2, Pascal violates name equivalence since file type names are not required to be shared by different programs accessing the same file.

**Structural Equivalence**

In structural equivalence, the names of the types are ignored and the elements of the types are compared for equality. It is possible that two logically different types may turn out to be the same by coincidence and may be mixed. Type definitions are not required to be global. Structural equivalence is important in programming distributed systems, in which separate programs must communicate typed data.

---

**Definition N.1:**

Two types $T$, $T'$ are *name equivalent* iff $T$ and $T'$ are the same name.

Two types $T$, $T'$ are *structurally equivalent* iff $T$ and $T'$ have the same set of values.

---

The following three rules may be used to determine if two types are structurally equivalent.

- A type name is structurally equivalent to its self.
- Two types are structurally equivalent if they are formed by applying the same type constructor (recursively) to structurally equivalent types.
- After a type declaration, type $n = T$, the type name $n$ is structurally equivalent to $T$. 
Structural equivalence was chosen by Algol-68 and C (except for records) because it is easy to implement.

**Type Inference**

Type inference is the general problem of transforming untyped or partially typed syntax into well-typed terms. Pascal constant declarations are an example of type inference, the type of the name is inferred from the type of the constant. In Pascal’s for loop the type of the loop index can be inferred from the types of the loop limits and thus the loop index should be a variable local to the loop. The programming languages Miranda and Haskell are statically types and provide powerful type inference systems so that a programmer need not declare any types. The languages also permit programmers to provide explicit type specifications.

A type checker must be able to

- determine if a program is well typed and
- if the program is well typed, determine the type of any expression in the program.

**Type inference axioms**

Axiom

given that: f is of type A --> B and x is of type A
infer that: f(x) is type correct and has type B

**Type declarations**

Even languages that provide a type inference system permit programmers to make explicit declarations of type. Even if the compiler can correctly infer types, human readers may have to scan several pages of code to determine the type of a function. Slight errors by the programmer can cause the compiler to emit obscure error messages or to infer a different type than intended. For these reasons it is good programming practice to explicitly state types on all but the most obvious cases.

**Examples.** In Miranda (a functional language) the types for the arithmetic + operation are declared as follows:

    + :: num --> num --> num
In Pascal the type of a function for computing the circumference of a circle is declared as follows:

\[
\text{function circumference( radius : real ) : real;}
\]

**Polymorphism**

A type system is *monomorphic* if each constant, variable, parameter, and function result has a unique type. Type checking in a monomorphic system is straightforward. But purely monomorphic type systems are unsatisfactory for writing reusable software. Many algorithms such as sorting and list and tree manipulation routines are *generic* in the sense that they depend very little on the type of the values being manipulated. For example, a general purpose array sorting routine cannot be written in Pascal. Pascal requires that the element type of the array be part of the declaration of the routine. This means that different sorting routines must be written for each element type and array size.

Completely monomorphic systems are rare. Most programming languages contain some operators or procedures which permit arguments of more than one type. For example, Pascal's input and output procedures permit variation both in type and in number of arguments. This is an example of *overloading*.

---

**Definition N.2:**

- **Monomorphism**: every constant, variable, parameter, operator and function has a unique type.
- **Overloading** refers to the use of a single syntactic identifier to refer to several different operations discriminated by the type and number of the arguments to the operation.
- **Polymorphism**: an operator, function or procedure that has a family of related types and operates uniformly on its arguments regardless of type.
- A *polymorphic* operation is one that can be applied to different but related types of arguments.

The type of the plus operation defined for integer addition is

\[
+: \text{int} \times \text{int} \rightarrow \text{int}
\]

When the same operation symbol is used for the plus operation for rational numbers and for set union, the symbol as in Pascal it is overloaded. Most programming languages provide for the overloading of the
Types

arithmetic operators. A few programming languages (Ada among others) provide for programmer defined overloading of both built-in and programmer defined operators.

When overloaded operators are applied to mixed expressions such as plus to an integer and a rational number there are two possible choices. Either the evaluation of the expression fails or one or more of the subexpressions are coerced into a corresponding object of another type. Integers are often coerced into the corresponding rational number. This type of coercion is called widening. When a language permits the coercion of a real number into an integer (by truncation for example) the coercion is called narrowing. Narrowing is not usually permitted in a programming language since information is usually lost. Coercion is an issue in programming languages because numbers do not have a uniform representation. This type of overloading is called context-dependent overloading.

Many languages provide type transfer functions so that the programmer can control where and when the type coercion is performed. Truncate and round are examples of type transfer functions.

Overloading is sometimes called ad-hoc polymorphism.

Most sorting algorithms can be explained without referring to the kind (type) of data being sorted. Typically, the data is an array of pointers to records each with an associated key. The type of the key does not matter as long as there is a "comparison" procedure which finds the minimum of a pair of keys. The sorting procedures use the compare two keys using the comparison procedure and swap the records by resetting pointers accordingly. However, in a strongly typed language this is not possible since the pointer type depends on the record type. This forces us to write a separate procedure for each type of data.

Stacks, queues, lists and trees also are largely type independent and yet in a strongly typed language, separate code must be written for each element type. Some languages permit type variables and these data structures can be defined with a type variable which then allows the user

A type system is polymorphic if abstractions operate uniformly on arguments of a family of related types.

This type of polymorphism is sometimes called parametric polymorphism.

Generalization can be applied to may aspects of programming languages.

Sometimes there are several domains which share a common operation. For example, the natural numbers, the integers, the rationals, and the reals all share the operation of addition. So, most programming languages use the same addition operator to denote addition in all of these domains. Pascal extends the use of the addition operator to represent set union. The multiple use of a name in different domains is called overloading. Ada permits user defined overloading of built in operators.
Prolog permits the programmer to use the same functor name for predicates of different arity thus permitting the overloading of functor names. This is an example of data generalization or polymorphism.

While the parameterization of an object gives the ability to deal with more than one particular object, polymorphism is the ability of an operation to deal with objects of more than a single type.

Generalization of control has focused on advanced control structures (RAM): iterators, generators, backtracking, exception handling, coroutines, and parallel execution (processes).

**Type Completeness**

**Principle of Type Completeness** No operation should be arbitrarily restricted in the types of the values involved.

**Pragmatics**

type declarations: spelling, type checking, type inference vs type declaration

**Historical Perspectives and Further Reading**

**Declarations**

**Constants**

literals

**User Defined Types**

Miranda's universe of values is numbers, characters and boolean values while Pascal provides boolean, integer, real, and char.

Declarations of variables of the primitive types have the following form in the imperative languages.

```
I : T;    { Modula-2: item I of type T}
```

```
T I;      // C++: item I of type T
```

Declarations of enumeration types involve listing of the values in the type.
Here are the enumerations of the items $I_1,...,I_n$ of type $T$.

\[ T = \{ I_1, ..., I_n \}; \quad \{ \text{Modula-2} \} \]
\[ \text{enum } T \{ I_1, ..., I_n \}; \quad // \ C++ \quad \]
\[ T ::= I_1 | ... | I_n \quad || \quad \text{Miranda} \]

**Modula-2, Ada, C++, Prolog, Scheme, Miranda -- list primitive types** Haskell provides the built in functions `fst` and `snd` to extract the first and second elements from binary tuples.

Imperative languages require that the elements of a tuple be named. Modula-2 is typical; product domains are defined by record types:

```plaintext
record
    I_1 : T_1;
    ...
    I_n : T_n;
end
```

The $I_i$s are names for the component of the tuple. The individual components of the record are accessed by the use of qualified names. for example, if MyRec is a element of the above type, then the first component is referenced by `MyRec.I_1` and the last component is referenced by `MyRec.I_n`.

C and C++ calls a product domain a structure and uses the following type declaration:

```plaintext
struct name {
    T_1 I_1;
    ...
    T_n : I_n;
};
```

The $I_i$s are names for the entries in the tuple.

Prolog does not require type declaration and elements of a product domain may be represented in a number of ways, one way is by a term of the form:

\[ name(I_1, \ldots I_n) \]

The $I_i$s are the entries in the tuple. The entries are accessed by pattern matching. Miranda does not require type declaration and the elements of a product domain are represented by tuples.
Types

\[(I_1, \ldots, I_n)\]

The \(I_i\)s are the entries in the tuple.

Here is an example of a variant record in Pascal.

```pascal
% From condensed pascal

type Shape = (Square, Rectangle, Rhomboid, Trapezoid, Parallelogram);

Dimensions = record
  case WhatShape : Shape of
    Square : (Side1: real);
    Rectangle : (Length, Width : real);
    Rhomboid : (Side2: real; AcuteAngle: 0..360);
    Trapezoid : (Top1, Bottom, Height: real);
    Parallelogram : (\=Top2, Side3: real;
                   ObtuseAngle: 0..360)
  end;

var FourSidedObject : Dimensions;
```

The initialization of the record should follow the sequence of assigning a value to the tag and then to the appropriate subfields.

```pascal
FourSidedObject.WhatShape := Rectangle;
FourSidedObject.Length := 4.3;
FourSidedObject.Width := 7.5;
```

The corresponding definition in Miranda is

```miranda
Dimensions :: Square num | Rectangle num num | Rhomboid num num | Trapezoid num num num | Parallelogram num num num

area Square S = S*S
area Rectangle L W = L * W
```

**Modula-2, Ada, C++, Prolog, Scheme, Miranda** Disjoint union values are implemented by allocating storage based on the largest possible value and additional storage for the tag.
Modula-2

```
array[domain_type] of range_type \{Modula-2\}
range_type identifier \{natural number\} // C++
```

Prolog and Miranda do not provide for an array type and while Scheme does, it is not a part of the purely functional part of Scheme. **Modula-2, Ada, C++, Prolog, Scheme, Miranda -- mapping type** In Pascal the notation \([i..j]\) indicates the subset of an ordinal type from element \(i\) to element \(j\) inclusive. In addition to subranges, Miranda provides infinite lists \([i..]\) and finite and infinite arithmetic series \([a,b..c], [a,b..]\) (the interval is \((b-a)\)). Miranda also provides list comprehensions which are used to construct lists (sets). A list comprehension has the form \([\text{exp} | \text{qualifier}]\)

```
sqs = \{ n*n | n <-[1..] \}
```

```
factors n = \{ r | r <-[1..n div 2]; n mod r = 0 \}
```

```
knight_moves [i,j] = \{ [i+a,j+b] | a,b <-[-2..2]; a^2+b^2=5 \}
```

**Modula-2, Ada, C++, Prolog, Scheme, Miranda -- power set**

```
[]
[I_0,...I_n]
[H | T]
```

The Miranda syntax for lists is similar to that of Prolog however, elements of lists must be all of the same type.

```
[*]
[I_0,..I_n]
[H | T]
```

Recursive types in imperative programming languages are usually defined using a pointer type. Pointer types are an additional primitive type. Pointers are addresses.

```
{Modula-2: the pointer and the list}
**type** NextItem = \verb+^+ListType

ListType = \verb+record\n    item : Itemtype;
    next : NextItem
\verb+end;\n```
// C++: the list type
struct list {
    ItemType Item;
    list* Next; // pointer to list
};

|| Miranda: list of objects of type T and
|| a binary tree of type T
[T]
tree ::= Niltree | Node T tree tree

Referencing/Dereferencing

type ListType = record
    item : ItemType;
    next : ListType
end;

Recursive values are implemented using pointers. The run-time support system for the functional and logic programming languages, provides for automatic allocation and recovery of storage (garbage collection). The alternative is for the language to provide access to the run-time system so that the programmer can explicitly allocate and recover the linked structures.

Variables

marginpar{state:store} It is frequently necessary to refer to an arbitrary element of a type. Such a reference is provided through the use of variables. A variable is a name for an arbitrary element of a type and it is a generalization of a value since it can be the name of any element.

Functions and Procedures

Persistent Types

A persistent variable is one whose lifetime transcends an activation of any particular program. In contrast, a transient variable is one whose lifetime is bounded by the activation of the program that created it. Local and heap variables are transient variables.

Most programming languages provide different types for persistent and transient variables. Typically files are the only type of persistent variable permitted in a programming language. Files are not usually permitted to be transient variables.
Most programming languages provide different types for persistent and transient. The type completeness principle suggests that all the types of the programming language should be allowed both for transient variables and persistent variables. A language applying this principle would be simplified by having no special types, commands or procedures for input/output. The programmer would be spared the effort of converting data from a persistent data type to a transient data type on input and vice versa on output.

A persistent variable of array type corresponds to a direct file. If heap variables were persistent then the storage of arbitrary data structures would be possible.

**Exercises**

1. Extend the compiler to handle constant, type, variable, function and procedure definitions and references to the same.
2. Static and dynamic scope
3. Define algebraic semantics for the following data types.
   1. Boolean

   ADT Boolean
   Operations
   \[\text{and}(\text{boolean}, \text{boolean}) \rightarrow \text{boolean}\]
   \[\text{or}(\text{boolean}, \text{boolean}) \rightarrow \text{boolean}\]
   \[\text{not}(\text{boolean}) \rightarrow \text{boolean}\]
   Semantic Equations
   \[\text{and}(\text{true}, \text{true}) = \text{true}\]
   \[\text{or}(\text{true}, \text{true}) = \text{true}\]
   \[\text{not}(\text{true}) = \text{false}\]
   \[\text{not}(\text{false}) = \text{true}\]
   Restrictions

   2. Integer
   3. Real
   4. Character
   5. String
4. Name or Structure equivalence (type checking)
5. Algebraic Semantics: stack, tree, queue, grade book etc.
6. Abstraction
7. Generalization
8. Name or Structure equivalence (type checking)
9. Extend the compiler to handle additional types. This requires modifications to the syntax of the language with extensions of the scanner, parser, symbol table and code generators.
Logic Programming

N. Wirth: *Program = data structure + algorithm*
R. Kowalski: *Algorithm = logic + control*
J. A. Robinson: *A program is a theory (in some logic) and computation is deduction from the theory.*

Logic programming is characterized by programming with relations and inference.

*Keywords and phrases:* Horn clause, Logic programming, inference, modus ponens, modus tollens, logic variable, unification, unifier, most general unifier, occurs-check, backtracking, closed world assumption, meta programming, pattern matching, set, relation, tuple, atom, constant, variable, predicate, functor, arity, term, compound term, ground, nonground, substitution, instance, instantiation, existential quantification, universal quantification, unification, modus ponens, proof tree, goal, resolvent.

A logic program consists of a set of axioms and a goal statement. The rules of inference are applied to determine whether the axioms are sufficient to ensure the truth of the goal statement. The execution of a logic program corresponds to the construction of a proof of the goal statement from the axioms.

In the logic programming model the programmer is responsible for specifying the basic logical relationships and does not specify the manner in which the inference rules are applied. Thus

Logic + Control = Algorithms

Logic programming is based on tuples. Predicates are abstractions and generalization of the data type of tuples. Recall, a tuple is an element of

$$S_0 \times S_1 \times \ldots \times S_n$$

The squaring function for natural numbers may be written as a set of tuples as follows:

$$\{(0,0), (1,1), (2,4) \ldots\}$$

Such a set of tuples is called a relation and in this case the tuples define the squaring relation.

$$\text{sqr} = \{(0,0), (1,1), (2,4) \ldots\}$$
Abstracting to the name sqr and generalizing an individual tuple we can define the squaring relation as:

\[ \text{sqr} = (x, x^2) \]

Parameterizing the name gives:

\[ \text{sqr}(X, Y) \leftarrow Y \text{ is } X \times X \]

In the logic programming language Prolog this would be written as:

\[ \text{sqr}(X, Y) \leftarrow Y \text{ is } X \times X. \]

Note that the set of tuples is named sqr and that the parameters are X and Y. Prolog does not evaluate its arguments unless required, so the expression \( Y \text{ is } X \times X \) forces the evaluation of \( X \times X \) and unifies the answer with \( Y \). The Prolog code

\[ \text{P} \leftarrow \text{Q}. \]

may be read in a number of ways; it could be read \( \text{P} \leftarrow \text{Q} \) or \( \text{P} \leftarrow \text{Q} \). In this latter form it is a variant of the first-order predicate calculus known as Horn clause logic. A complete reading of the sqr predicate the point of view of logic is: for every \( X \) and \( Y \), \( Y \) is the sqr of \( X \) if \( Y \) is \( X \times X \). From the point of view of logic, we say that the variables are universally quantified. Horn clause logic has a particularly simple inference rule which permits its use as a model of computation. This computational paradigm is called Logic programming and deals with relations rather than functions or assignments. It uses facts and rules to represent information and deduction to answer queries. Prolog is the most widely available programming language to implement this computational paradigm.

Relations may be composed. For example, suppose we have the predicates, \( \text{male}(X), \text{siblingof}(X,Y), \) and \( \text{parentof}(Y,Z) \) which define the obvious relations, then we can define the predicate uncleof(X,Z) which implements the obvious relation as follows:

\[ \text{uncleof}(X, Z) \leftarrow \text{male}(X), \text{siblingof}(X,Y), \text{parentof}(Y,Z). \]

The logical reading of this rule is as follows: ``for every \( X, Y \) and \( Z \), \( X \) is the uncle of \( Z \), if \( X \) is a male who has a sibling \( Y \) which is the parent of \( Z \).'' Alternately, ``\( X \) is the uncle of \( Z \), if \( X \) is a male and \( X \) is a sibing of \( Y \) and \( Y \) is a parent of \( Z \).'' \%fatherof(X,Y),fatherof(Y,Z) defines paternalgrandfather(X,Z)

The difference between logic programming and functional programming may be illustrated as follows. The logic program

\[ \text{f}(X, Y) \leftarrow Y = X \times 3 + 4 \]
is an abbreviation for
\[ \forall X,Y \ (f(X,Y) \leftarrow Y = X*3+4) \]

which asserts a condition that must hold between the corresponding domain and range elements of the function. In contrast, a functional definition introduces a functional object to which functional operations such as functional composition may be applied.

Logic programming has many application areas:

- Relational Data Bases
- Natural Language Interfaces
- Expert Systems
- Symbolic Equation solving
- Planning
- Prototyping
- Simulation
- Programming Language Implementation

**Syntax**

There are just four constructs: constants, variables, function symbols, predicate symbols, and two logical connectives, the comma (and) and the implication symbol.

Core Prolog

- \texttt{P} in Programs
- \texttt{C} in Clauses
- \texttt{Q} in Queries
- \texttt{A} in Atoms
- \texttt{T} in Terms
- \texttt{X} in Variables

Program ::= Clause... Query | Query
Clause ::= Predicate . | Predicate :- PredicateList .
PredicateList ::= Predicate | PredicateList , Predicate
Predicate ::= Atom | Atom( TermList )
TermList ::= Term | TermList , Term
Term ::= Numeral | Atom | Variable | Structure
Structure ::= Atom ( TermList )
Query ::= ?- PredicateList .
Numeral ::= an integer or real number
Atom ::= string of characters beginning with a lowercase letter or enclosed in apostrophes.
Variable ::= string of characters beginning with an uppercase letter or underscore

Terminals = {Numeral, Atom, Variable, :-, ?, -, comma, period, left and right parentheses }

While there is no standard syntax for Prolog, most implementations recognize the grammar in Figure M.N.

---

**Figure M.N:** Prolog grammar

P in Programs
C in Clauses
Q in Query
H in Head
B in Body
A in Atoms
T in Terms
X in Variable

\[
P ::= C \ldots Q \ldots \\
C ::= H [ :- B ] . \\
H ::= A [ ( T [ , T ] \ldots ) ] \\
B ::= G [, G] \ldots \\
G ::= A [ ( [ X | T ] \ldots ) ] \\
T ::= X | A [ ( T \ldots ) ] \\
Q ::= ?- B .
\]

---

CLAUSE, FACT, RULE, QUERY, FUNCTOR, ARITY, ORDER, UNIVERSAL QUANTIFICATION, EXISTENTIAL QUANTIFICATION, RELATIONS

In logic, relations are named by predicate symbols chosen from a prescribed vocabulary. Knowledge about the relations is then expressed by sentences constructed from predicates, connectives, and formulas. An n-ary predicate is constructed from prefixing an n-tuple with an n-ary predicate symbol.

A logic program is a set of axioms, or rules, defining relationships between objects. A computation of a logic program is a deduction of consequences of the program. A program defines a set of consequences, which is its meaning. The art of logic programming is constructing concise and elegant programs that...
have the desired meaning.

The basic constructs of logic programming, terms and statements are inherited from logic. There are three basic statements: facts, rules and queries. There is a single data structure: the logical term.

**Facts, Predicates and Atoms**

Facts are a means of stating that a relationship holds between objects.

```prolog
father(bill,mary).
plus(2,3,5).
...
```

This fact states that the relation father holds between bill and mary. Another name for a relationship is predicate.

**Queries**

A query is the means of retrieving information from a logic program.

```prolog
?- father(bill,mary).
?- father(bill,jim).
```

Note that the text terminates queries with a question mark rather than preceding.

**Semantics**

The operational semantics of logic programs correspond to logical inference. The declarative semantics of logic programs are derived from the term model commonly referred to as the Herbrand base. The denotational semantics of logic programs are defined in terms of a function which assigns meaning to the program.

There is a close relation between the axiomatic semantics of imperative programs and logic programs. A logic program to sum the elements of a list could be written as follows.

```prolog
sum([Nth],Nth).
sum([Ith|Rest],Ith + Sum_Rest) <-- sum(Rest,Sum_Rest).
```

A proof of its correctness is trivial since the logic program is but a statement of the mathematical properties of the sum.
Logic Programming

\[ A[N] = \sum_{i=N}^N A[i] \]
\[ \sum([A[N]], A[N]) \].
\[ \sum_{i=1}^N A[i] = A[I] + S \text{ if } 0 < I, \sum_{i=I+1}^N A[i] = S \]

Operational Semantics

Definition: The meaning of a logic program P, M(P), is the set of unit goals deducible from P.

- Logic Program: A logic program is a finite set of facts and rules.
- Interpretation and meaning of logic programs.

The rule of instantiation (P(X) deduce P(c)). The rule of deduction is modus ponens. From A :- B1, B2, ..., Bn. and B1', B2', ..., Bn' infer A'. Primes indicate instances of the corresponding term.

The meaning M(P) of a logical program P is the set of unit goals deducible from the program.

A program P is correct with respect to some intended meaning M iff the meaning of P M(P) is a subset of M (the program does not say things that were not intended).

A program P is complete with respect to some intended meaning M iff M is a subset of M(P) (the program says everything that was intended).

A program P is correct and complete with respect to some intended meaning M iff M = M(P).

The operational semantics of a logic program can be described in terms of logical inference using unification and the inference rule resolution. The following logic program illustrates logical inference.

\[ a. \]
\[ b \leftarrow a. \]
\[ b? \]

We can conclude b by modus ponens given that b \leftarrow a and a. Alternatively, if b is assume to be false then from b \leftarrow a and modus tollens we infer \neg a but since a is given we have a contradiction and b must hold. The following program illustrates unification.

\[ \text{parent_of}(a,b). \]
\[ \text{parent_of}(b,c). \]
\[ \text{ancestor_of}(\text{Anc}, \text{Desc}) \leftarrow \text{parent_of}(\text{Anc}, \text{Desc}). \]
\[ \text{ancestor_of}(\text{Anc}, \text{Desc}) \leftarrow \text{parent_of}(\text{Anc}, \text{Interm}) \wedge \]

http://cs.wwc.edu/~aabyan/PLBook/HTML/Logic.html (6 of 30)8/10/2006 8:27:32 PM
ancestor_of(Interm, Desc).
parent_of(a, b)?
ancestor_of(a, b)?
ancestor_of(a, c)?
ancestor_of(X, Y)?

Consider the query `ancestor_of(a,b)?'. To answer the question ``is a an ancestor of b'', we must select the second rule for the ancestor relation and unify a with Anc and b with Desc. Interm then unifies with c in the relation parent_of(b,c). The query, ancestor_of(b,c)? is answered by the first rule for the ancestor_of relation. The last query is asking the question, ``Are there two persons such that the first is an ancestor of the second.'' The variables in queries are said to be existentially quantified. In this case the X unifies with a and the Y unifies with b through the parent_of relation. Formally,

Definition M.N:

A unifier of two terms is a substitution making the terms identical. If two terms have a unifier, we say they unify.

For example, two identical terms unify with the identity substitution. concat([1,2,3],[3,4],List) and concat ([X|Xs],Ys,[X|Zs]) unify with the substitutions \{X = 1, Xs = [2,3], Ys = [3,4], List = [1|Zs]\}

There is just one rule of inference which is resolution. Resolution is much like proof by contradiction. An instance of a relation is ```computed'' by constructing a refutation. During the course of the proof, a tree is constructed with the statement to be proved at the root. When we construct proofs we will use the symbol \(\neg\) to mark formulas which we either assume are false or infer are false and the symbol [] for contradiction. Resolution is based on the inference rule modus tollens and unification. This is the modus tollens inference rule.

\[
\text{From } \neg B \\
\text{and } B \leftarrow A_0,\ldots,A_n \\
\text{infer } \neg A_0 \text{ or...or } \neg A_n
\]

Notice that as a result of the inference there are several choices. Each \(\neg A_\{i\}\) is a formula marking a new branch in the proof tree. A contradiction occurs when both a formula and its negation appear on the same path through the proof tree. A path is said to be closed when it contains a contradiction otherwise a path is said to be open. A formula has a proof if and only if each path in the proof tree is closed. The following is a proof tree for the formula B under the hypotheses A0 and B \leftarrow A0,A_\{1\}. 

http://cs.wwc.edu/~aabyan/PLBook/HTML/Logic.html (7 of 30)8/10/2006 8:27:32 PM
There are two paths through the proof tree, 1-4, 5, 6 and 1-4, 7, 8. The first path contains a contradiction while the second does not. The contradiction is marked with [].

As an example of computing in this system of logic suppose we have defined the relations parent and ancestor as follows:

1. parent_of(ogden, anthony)
2. parent_of(anthony, mikko)
3. parent_of(anthony, andra)
4. ancestor_of(A,D) <- parent_of(A,D)
5. ancestor_of(A,D) <- parent_of(A,X)
6. ancestor_of(X,D)

where identifiers beginning with lower case letters designate constants and identifiers beginning with an upper case letter designate variables. We can infer that ogden is an ancestor of mikko as follows.

¬ ancestor(ogden, mikko) the assumption
¬ parent(ogden, X) or} ¬ ancestor(X, mikko)
res)
¬ parent(ogden, X) first choice
¬ parent(ogden, anthony) unification with first entry
[] produces a contradiction
¬ ancestor(anthony, mikko) second choice
¬ parent(anthony, mikko) resolution
[] A contradiction of a fact.)

Notice that all choices result in contradictions and so this proof tree is a proof of the proposition that ogden is an ancestor of mikko. In a proof, when unification occurs, the result is a substitution. In the first branch of the previous example, the term antonony is unified with the variable X and anthony is substituted for all occurences of the variable X.

UNIVERSAL QUANTIFICATION, EXISTENTIAL QUANTIFICATION
The unification algorithm can be defined in Prolog. Figure~\ref{lp:unify} contains a formal definition of unification in Prolog

Figure MN: Unification Algorithm

\begin{verbatim}
unify(X,Y) <-- X == Y.
unify(X,Y) <-- var(X), var(Y), X=Y.
unify(X,Y) <-- var(X), nonvar(Y), \+ occurs(X,Y), X=Y.
unify(X,Y) <-- var(Y), nonvar(X), \+ occurs(Y,X), Y=X.
unify(X,Y) <-- nonvar(X), nonvar(Y), functor(X,F,N), functor(Y,F,N),
X =..[F|R], Y =..[F|T], unify_lists(R,T).

unify_lists([],[]).
unify_lists([X|R],[H|T]) <-- unify(X,H), unify_lists(R,T).

occurs(X,Y) <-- X==Y.
occurs(X,T) <-- functor(T,F,N), T =..[F|Ts], occurs_list(X,Ts).

occurs_list(X,[Y|R]) <-- occurs(X,Y).
occurs_list(X,[Y|R]) <-- occurs_list(X,R).
\end{verbatim}

Unification subsumes

- single assignment
- parameter passing
- record allocation
- read/write-once field-access in records

\begin{array}{l}
\frac{A1 <-- B.}, \quad \text{?- A1, A2,\ldots, An.}}{\text{?- B, A2,\ldots, An.}}
\frac{?\text{- true, A1, A2,\ldots, An.}}{\text{?- A1, A2,\ldots, An.}}
\caption{Inference Rules \label{lp:ir}}
\end{array}

To illustrate the inference rules, consider the following program consisting of a rule, two facts and a query:

a <-- b \wedge c . b <-- d . b <-- e . ?- a .

By applying the inference rules to the program we derive the following additional queries:
Among the queries is an empty query. The presence of the empty query indicates that the original query is satisfiable, that is, the answer to the query is yes. Alternatively, the query is a theorem, provable from the given facts and rules.

**Inference and Unification**

**Definition:** The law of universal modus ponens says that from

\[ R = (A :- B_1, \ldots, B_n) \text{ and } B'_1. \ldots B'_n. \]

\[ A' \text{ can be deduced, if } A' :- B'_1, \ldots, B'_n \text{ is an instance of } R. \]

**Definition:** A logic program is a finite set of rules.

**Definition:** An existentially quantified goal \( G \) is a logical consequence of a program \( P \) iff there is a clause in \( P \) with an instance \( A :- B_1, \ldots, B_n, n \geq 0 \) such that \( B_1, \ldots, B_n \) are logical consequences of \( P \) and \( A \) is an instance of \( G \).

The control portion of the the equation is provide by an inference engine whose role is to derive theorems based on the set of axioms provided by the programmer. The inference engine uses the operations of resolution and unification to construct proofs.

**Resolution** says that given the axioms

\[ f \text{ if } a_0, \ldots, a_m. \]
\[ g \text{ if } f, b_0, \ldots, b_n. \]

the fact

\[ g \text{ if } a_0, \ldots, a_m, b_0, \ldots, b_n. \]

can be derived.

**Unification** is the binding of variables. For example
A query containing a variable asks whether there is a value for the variable that makes the query a logical consequence of the program.

?- father(bill,X).
?- father(X,mary).
?- father(X,Y).

Note that variables do not denote a specified storage location, but denote an unspecified but single entity.

Definition: Constants and variables are terms. A compound term is comprised of a functor and a sequence of terms. A functor is characterized by its name, which is an atom and its arity, or number of arguments.

X, 3, mary, fatherof(F,mary), ...

Definition: Queries, facts and terms which do not contain variables are called ground. Where variables do occur they are called nonground.

Definition: A substitution is a finite set (possibly empty) of pairs of the form Xi=ti, where Xi is a variable and ti is a term, and Xi\neq Xj for every i \neq j, and Xi does not occur in tj, for any i and j.

\[ p(a, X, t(Z)), p(Y, m, Q); \theta = \{ X=m, Y=a, Q=t(Z) \} \]

Definition: A is an instance of B if there is a substitution \( \theta \) such that \( A = B\theta \).

Definition: Two terms A and B are said to have a common instance C iff there are substitutions \( \theta_1 \) and \( \theta_2 \) such that \( C = A\theta_1 \) and \( C = B\theta_2 \).

\[ A = \text{plus}(0, 3, Y), B = \text{plus}(0, X, X). C = \text{plus}(0, 3, 3) \]
since \( C = A\{ Y=3 \} \) and \( C = B\{ X=3 \} \).

Definition: A unifier of two terms A and B is a substitution making the two terms identical. If two terms have a unifier they are said to unify.

\[ p(a, X, t(Z))\theta = p(Y, m, Q)\theta \quad \text{where} \quad \theta = \{ X=m, Y=a, Q=t(Z) \} \]

Definition: A most general unifier or mgu of two terms is a unifier such that the associated common instance is most general.
unify(A,B) :- unify1(A,B).

unify1(X,Y) :- X == Y.
unify1(X,Y) :- var(X), var(Y), X=Y. % The substitution
unify1(X,Y) :- var(X), nonvar(Y), \+ occurs(X,Y), X=Y. % The substitution
unify1(X,Y) :- var(Y), nonvar(X), \+ occurs(Y,X), Y=X. % The substitution
unify1(X,Y) :- nonvar(X), nonvar(Y), functor(X,F,N), functor(Y,F,N),
X =..[F|R], Y =..[F|T], match_list(R,T).

match_list([],[]).
match_list([X|R],[H|T]) :- unify(X,H), match_list(R,T).

occurs(A,B) :- A == B.
occurs(A,B) :- nonvar(B), functor(B,F,N), occurs(A,B,N).

occurs(A,B,N) :- N > 0, arg(N,B,AN), occurs(A,AN),!. % RED
occurs(A,B,M) :- M > 0, N is M - 1, occurs(A,B,N).

A Simple Interpreter for Pure Prolog

An interpreter for pure Prolog can be written in Prolog.

Figure M.N: A simple interpreter for pure Prolog

is_true( Goals ) <-- resolved( Goals ).

is_true( Goals ) <-- write( no ), nl.

resolved([]).
resolved(Goals) <-- select(Goal,Goals,RestofGoals),
 % Goal unifies with head of some
rule
    clause(Head,Body), unify( Goal, Head ),
add(Body,RestofGoals,NewGoals),
resolved(NewGoals).

prove(true).
prove((A,B)) <-- prove(A), prove(B). % select first goal
prove(A) <-- clause(A,B), prove(B). % select only goal and
find a rule
Logic Programming is the Prolog code for an interpreter.

The interpreter can be used as the starting point for the construction of a debugger for Prolog programs and a starting point for the construction of an inference engine for an expert system.

The operational semantics for Prolog are given in Figure~\ref{lp:opsem}

---

Logic Programming (Horn Clause Logic) -- Operational Semantics

Abstract Syntax:

- \( P \) in Programs
- \( C \) in Clauses
- \( Q \) in Queries
- \( T \) in Terms
- \( A \) in Atoms
- \( X \) in Variables

\[
P ::= (C | Q)...
C ::= G [ <-- G_{1}[ \wedge G_{2}]... ] .
G ::= A [ ( T [,T]... ) ]
T ::= X | A [ ( T [,T]... ) ]
Q ::= G [,G]... ?
\]

Semantic Domains:

- \( \beta \) in \( \{ \textbf{B} \} = \) Bindings
- \( \epsilon \) in \( \{ \textbf{E} \} = \) Environment

Semantic Functions:

\[
R \text{ in } Q} --> B} --> B } + (B} \times \{ \text{yes } \}) + \{ \text{no } \})
U \text{ in } C } \times C } --> B } --> B }
\]

Semantic Equations:

\[
R[ ?} ] \beta , \epsilon \text{psilon } \&=\& (\beta , \text{yes})
R[ G ] \beta , \epsilon \text{psilon } \&=\& \beta'
\]
Declarative Semantics

The declarative semantics of logic programs is based on the standard model-theoretic semantics of first-order logic.

Definition M.N:

Let P be a logic program. The Herbrand universe of P, denoted by U(P) is the set of ground terms that can be formed from the constants and function symbols appearing in P.

Definition M.N:

The Herbrand base, denoted by \( \mathcal{B}(P) \), is the set of all ground goals that can be formed from the predicates in P and the terms in the Herbrand universe.

The Herbrand base is infinite if the Herbrand universe is.
Definition M.N:

An interpretation for a logic program is a subset of the Herbrand base.

An interpretation assigns truth and falsity to the elements of the Herbrand base. A goal in the Herbrand base is true with respect to an interpretation if it is a member of it, false otherwise.

Definition M.N:

An interpretation $I$ is a model for a logic program if for each ground instance of a clause in the program $A \leftarrow B_1, \ldots, B_n$ $A$ is in $I$ if $B_1, \ldots, B_n$ are in $I$.

This approach to the semantics is often called the term model.

Denotational Semantics

Denotational semantics assignes meanings to programs based on associating with the program a function over the domain computed by the program. The meaning of the program is defined as the least fixed point of the function, if it exists.

Pragmatics

Logic Programming and Software Engineering

Programs are theories and computation is deduction from the theory. Thus the process of software engineering becomes:

- obtain a problem description
- define the intended model of interpretation (domains, symbols etc)
- devise a suitable theory (the logic component) suitably restricted so as to have an efficient proof procedure.
- describe the control component of the program
- use declarative debugging to isolate errors in definitions

Pros and Cons
• Pro
  o Closer to problem domain thus higher programmer productivity
  o Separation of logic and control (focuses on the logical structure of the problem rather than
    control of execution)
  o Simple declarative semantics and referential transparency
  o Suitable for prototyping and exploratory programming
  o Strong support for meta-programming
  o Transparent support for parallel execution

• Con
  o Operational implementation is not faithful to the declarative semantics
  o Unsuitable for state based programming
  o Often inefficient

The Logical Variable

The logical variable, terms and lists are the basic data structures in logic programming.

Here is a definition of the relation between the prefix and suffixes of a list. The relation is named concat
because it may be viewed as defining the result of appending two lists to get the third list.

\{ l \}
concat([ ],[ ]) concat([H|T],L,[H|TL]) <-- concat(T,L,TL)

Logical variables operate in a way much different than variables in traditional programming languages.
By way of illustration, consider the following instances of the concat relation.

1. ?- concat([a,b,c],[d,e],L). L = [a, b, c, d, e] the expected use of the concat operation.
2. ?- concat([a,b,c],S,[a,b,c,d,e]). S = [d, e] the suffix of L.
3. ?- concat(P,[d,e],[a,b,c,d,e]). P = [a, b, c] the prefix of L.
4. ?- concat(P,S,[a,b,c,d,e]). P = [ ], S = [a,b,c,d,e] P = [a], S = [b,c,d,e] P = [a,b], S = [c,d,e] P = [a,
b,c], S = [d,e] P = [a,b,c,d], S = [e] P = [a,b,c,d,e], S = [ ] the prefixes and suffixes of L.
5. ?- concat(_,[c|_],[a,b,c,d,e]). answers Yes since c is the first element of some suffix of L.

Thus concat gives us 5 predicates for the price of one.

    concat(L1,L2,L)
    prefix(Pre,L) <-- concat(Pre,_,L).
    sufix(Suf,L) <-- concat(_,Suf,L).
    split(L,Pre,Suf) <-- concat(Pre,Suf,L).
    member(X,L) <-- concat(_,[X|_],L).

The underscore _ designates an anonymous variable, it matches anything.
There are two simple types of constants, string and numeric. Arrays may be represented as a relation. For example, the two-dimensional matrix

\[
\text{data} = \begin{array}{lr}
\text{mary} & 18.47 \\
\text{john} & 34.6 \\
\text{jane} & 64.4 \\
\end{array}
\]

may be written as

\[
\text{data(1,1,mary) & data(1,2,18.47) data(2,1,john) & data(2,2,34.6) data(3,1,jane) & data(3,2,64.4)}
\]

Records may be represented as terms and the fields accessed through pattern matching.

\[
\text{book(author(last(aaby), first(anthony)), mi(a)), title('programming language concepts), pub(wadsworth), date(1991))}
\]

\[
\text{book(A,T,pub(W),D)}
\]

Lists are written between brackets [ and ], so [ ] is the empty list and [b, c] is the list of two symbols b and c. If H is a symbol and T is a list then [H|T] is a list with head H and tail T. Stacks may then be represented as a list. Trees may be represented as lists of lists or as terms.

Lists may be used to simulate stacks, queues and trees. In addition, the logical variable may be used to implement incomplete data structures.

**Incomplete Data Structures**

The following code implements a binary search tree as an incomplete data structure. It may be used both to construct the tree by inserting items into the tree and to search the tree for a particular key and associated data.

\[
\begin{align*}
\text{lookup(Key,Data,bt(Key,Data,LT,RT))} \\
\text{lookup(Key,Data,bt(Key0,Data0,LT,RT)) } & \leftarrow \text{ Key } @< \text{ Key0}, \\
\text{lookup(Key,Data,LT)} \\
\text{lookup(Key,Data,bt(Key0,Data0,LT,RT)) } & \leftarrow \text{ Key } @> \text{ Key0}, \\
\text{lookup(Key,Data,RT)}
\end{align*}
\]

This is a sequence of calls. Note that the initial call is with the variable BT.

\[
\text{lookup(john,46,BT), lookup(jane,35,BT), lookup(ellen,49, BT), lookup(jane,Age,BT)}.
\]
The first three calls initialize the dictionary to contain those entries while the last call extracts janed's age from the dictionary.

The logical and the incomplete data structure can be used to append lists in constant time. The programming technique is known as difference lists. The empty difference list is \( X/X \). The concat relation for difference lists is defined as follows:

\[
\text{concat}_dl(Xs/Ys, Ys/Zs, Xs/Zs)
\]

Here is an example of a use of the definition.

\[
?- \text{concat}_dl([1,2,3|X]/X,[4,5,6|Y]/Y,Z).
\]

\[_X = [4,5,6 \mid 11]\
_Y = 11\
_Z = [1,2,3,4,5,6 \mid 11] / 11\]

\Yes

The relation between ordinary lists and difference lists is defined as follows:

\[
\text{ol}_dl([ \ ],X/X) \leftarrow \text{var}(X)\
\text{ol}_dl([\mathbf{F}|R],[\mathbf{F}|DL]/Y) \leftarrow \text{ol}_dl(R,DL/Y)
\]

**Arithmetic**

Terms are simply patterns they may not have a value in and of themselves. For example, here is a definition of the relation between two numbers and their product.

\[
\text{times}(X,Y,X\times Y)
\]

However, the product is a pattern rather than a value. In order to force the evaluation of an expression, a Prolog definition of the same relation would be written

\[
\text{times}(X,Y,Z) \leftarrow Z \text{ is } X\times Y
\]

**Iteration vs Recursion**

Not all recursive definitions require the runtime support usually associated with recursive subprogram calls. Consider the following elegant mathematical definition of the factorial function.
Here is a direct restatement of the definition in a relational form.

\[
\begin{align*}
\text{factorial}(0,1) \\
\text{factorial}(N,N \times F) & \leftarrow \text{factorial}(N-1,F)
\end{align*}
\]

In Prolog this definition does not evaluate either of the expressions \(N-1\) or \(N \times F\) thus the value 0 will not occur. To force evaluation of the expressions we rewrite the definition as follows.

\[
\begin{align*}
\text{factorial}(0,1) \\
\text{factorial}(N,F) & \leftarrow M \text{ is } N-1, \text{ factorial}(M,Fm), F \text{ is } N \times Fm
\end{align*}
\]

Note that in this last version, the call to the factorial predicate is not the last call on the right-hand side of the definition. When the last call on the right-hand side is a recursive call (\{\it tail recursion\}) then the definition is said to be an iterative definition. An iterative version of the factorial relation may be defined using an accumulator and tail recursion.

\[
\begin{align*}
\text{fac}(N,F) & \leftarrow \text{fac}(N,1,F) \\
\text{fac}(0,F,F) \\
\text{fac}(N,P,F) & \leftarrow NP \text{ is } N \times P, M \text{ is } N-1, \text{ fac}(M,NP,F)
\end{align*}
\]

In this definition, there are two different fac relations, the first is a 2-ary relation, and the second is a 3-ary relation.

As a further example of the relation between recursive and iterative definitions, here is a recursive version of the relation between a list and its reverse.

\[
\begin{align*}
\text{reverse}([\ ],[\ ]) \\
\text{reverse}([H|T],R) & \leftarrow \text{reverse}(T,Tr), \text{concat}(Tr,[H],R)
\end{align*}
\]

and here is an iterative version.

\[
\begin{align*}
\text{rev}(L,R) & \leftarrow \text{rev}(L,[\ ],R) \\
\text{rev}([\ ],R,R) \\
\text{rev}([H|T],L,R) & \leftarrow \text{rev}(T,[H|L],R)
\end{align*}
\]

Efficient implementation of recursion is possible when the recursion is tail recursion. Tail recursion is implementable as iteration provided no backtracking may be required (the only other predicate in the body are builtin predicates).
Backtracking

When there are multiple clauses defining a relation it is possible that either some of the clauses defining the relation are not applicable in a particular instance or that there are multiple solutions. The selection of alternate paths during the construction of a proof tree is called backtracking.

Exceptions

Logic programming provides an unusually simple method for handling exception conditions. Exceptions are handled by backtracking.

Logic Programming vs Functional Programming

Functional composition vs composition of relations, backtracking, type checking

Prolog and Logic

The Logic of Prolog

Horn Clauses

Translation of first-order predicate logic to horn clause logic:

Replace

- A <-> B with A --> B and B --> A
- A --> B with not A \lor B
- Move negations inward (from the outside inward). Replace
  - not (A and B) with not A or not B
  - not (A or B) with not A and not B
  - not Exists x. P with Forall x. not P
  - not Forall x. P with Exists x. not P
- Skolemize (replace existential variables with skolem constants or skolem funcions of universal variables (from the outside inward). Replace
  - Exists x. P(x) with P(c) where c is new
  - Forall x. ... Exists y. P(y) with Forall x. ... P(f_c(c_k)) where f_c and c_k are new
- Move universal quantifiers outward. Replace

  ... Forall x. P(x) with Forall x. ... P(x)
  (we can just drop the quantifiers)
- Put quantifier free portion into conjunctive normal form (conjunction of disjunctions). Replace
Logic Programming

- (A and B) or C with (A or C) and (B or C)
- (A and C) or (B and C) with (A or B) and C
  (move conjunctions out and disjunctions in)

Each disjunction is of the form: not A_1\lor\ldots\lor not A_m\lor B_1\lor\ldots\lor B_n
which is equivalent to: A_1\land\ldots\land A_m \rightarrow B_1\lor\ldots\lor B_n

- If m=0 and n=1 then we have a Prolog fact.
- If m>0 and n=1 then we have a Prolog rule.
- If m>0 and n=0 then we have a Prolog query.

If n always is 1 then the logic is called Horn Clause Logic which is equivalent in computational power to the Universal Turing Machine.

**Resolution and unification, forward and backward chaining**

The resolution rule combines clauses when a negated and a non-negated literal match.

If Aj and B_y `match' then by resolution: ...not Ai \lor not Aj \lor B_k...
...not A_x \lor B_y \lor B_z...
----------------------------
...not Ai \lor\ldots\lor A_x \lor B_z\ldots\lor B_k
Matching is called unification.

Direction of Proof

- Forward chaining: proofs proceed from facts through rules to conclusions (goals). Also called bottom-up.
- Backward chaining: proofs proceed from goals back through rules toward facts. Also called top-down and goal-directed.

**The Illogic of Prolog**

Prolog (for efficiency reasons) departs from the logic programming model in several ways. Prolog does not perform the ``occurs check''. Prolog is implemented as a sequential programming language by processing goals from left to right and selecting rules in textual order (depth-first search).

Prolog is not logic programming. The execution model for Prolog omits the occurs check, searches the rules sequentially, and selects goals sequentially. Backtracking, infinite search trees ...

As in functional programming, lists are an important data structure logic programming. The empty list is
represented by [], a list of n elements by [X1, ..., Xn] and the first i elements of a list and the rest of the list by [X1, ..., Xi|R]. In addition, data structures of arbitrary complexity may be constructed from the terms.

- Depth-first, left-right search instead of breadth-first parallel search means that rule and clause order can matter. Instead of combinatorial explosion in the size of the search tree, we may have infinite recursion.
- There is no `occurs check' when performing unification. This means that X unifies with f(X) -- infinite terms may be constructed during unification. Since this is an infrequent occurrence, we are trading correctness for reduction in running time.
- Negation by failure, `not'. Closed world assumption. Horn clause logic does not include the `not' operator, however its use simplifies programs.
- The `cut', prunes unnecessary branches. Encourages a `goto' style programming.

**Incompleteness**

Incompleteness occurs when there is a solution but it cannot be found. The depth first search of Prolog will never answer the query in the following logic program.

```prolog
p( a, b ).
p( c, b ).
p( X, Z ) <-- p( X, Y ), p( Y, Z).
p( X, Y ) <-- p( Y, X ).
?- p( a, c ).
```

The result is an infinite loop. The first and fourth clauses imply p( b, c ). The first and third clauses with the p( b, c) imply the query. Prolog gets lost in an infinite branch no matter how the clauses are ordered, how the literals in the bodies are ordered or what search rule with a fixed order for trying the clauses is used. Thus logical completeness requires a breadth-first search which is too inefficient to be practical.

**Unfairness**

Unfairness occurs when a permissible value cannot be found.

```prolog
concat( [ ], L, L ).
concat( [H|L1], L2, [X|L] ) <-- concat( L1, L2, L ).
concat3( L1, L2, L3, L ) <-- concat( L1, L2, L12 ),
    concat( L12, L3 L ).
?- concat3( X, Y, [2], L).
```

Result is that X is always [ ]. Prolog's depth-first search prevents it from finding other values.


**Unsoundness**

Unsoundness occurs when there is a successful computation of a goal which is not a logical consequence of the logic program.

```prolog
test <- p(X, X).
p(Y, f(Y)).
?- test.
```

Lacking the occur check Prolog will succeed but `test` is not a logical consequence of the logic program.

The execution of this logic program results in the construction of an infinite data structure.

```prolog
concat([], L, L).
concat([H|L1], L2, [X|L]) <- concat(L1, L2, L).
?- concat([], L, [1|L]).
```

In this instance Prolog will succeed (with some trouble printing the answer). There are two solutions, the first is to change the logic and permit infinite terms, the second is to introduce the occur check with the resulting loss of efficiency.

**Negation**

Negative information cannot be expressed in Horn clause logic. However, Prolog provides the negation operator `not` and defines negation as failure to find a proof.

```prolog
p(a).
r(b) <- not p(Y).
?- not p(b).
```

The goal succeeds but is not a logical consequence of the logic program.

```prolog
q(a) <- r(a).
q(a) <- not r(a).
r(X) <- r(f(X)).
?- q(a).
```

The query is a logical consequence of the first two clauses but Prolog cannot determine that fact and enters an infinite derivation tree. However the closed world assumption is useful from a pragmatic point of view.
Control Information

Cut (!): prunes the proof tree.

```prolog
a(1).
a(2).
a(3).
p <-- a(I),!,print(I),nl,fail.
?- p.
1
No
```

Extralogical Features

Input-output primitives cannot be fully described in first-order logic. These primitives produce input-output by side-effects.

Some other extralogical primitives include bagof, setof, assert, retract, univ. These are outside the scope of first-order logic.

Input and output introduce side effects.

The extralogical primitives `\verb+bagof+, \verb+setof+, \verb+assert+, and \verb+retract+ are outside the scope of first-order logic but are useful from the pragmatic point of view.

In Prolog there are builtin predicates to test for the various syntactic types, lists, numbers, atoms, clauses. Some predicates which are commonly available are the following.

{ll} var(X)&X is a variable atomic(A)&A is an atom or a numeric constant functor(P,F,N)&P is an N-ary predicate with functor F clause(Head,Body)&Head <-- Body is a formula. L =..List, call(C), assert (C), retract(C), bagof(X,P,B), setof(X,P,B)

Figure M.N:

```prolog
trace(Q) <-- trace1([Q])
trace1([])
trace1([true|R]) <-- !, trace1(R).
trace1([fail|R]) <-- !, print('</'), print(fail), nl, fail.
```
contains an example of meta programming. The code implements a facility for tracing the execution of a Prolog program. To trace a Prolog program, instead of entering `?- P.` enter `?- trace(P).`

**Multidirectionality**

Computation of the inverse function must be restricted for efficiency and undecidability reasons. For example consider the query `{!} ?- factorial(N,5678).` An implementation must either generate and test possible values for N (which is much too inefficient) or if there is no such N the undecidability of first-order logic implies that termination may not occur.

**Rule Order**

Rule order affects the order of search and thus the shape of the proof tree. In the following program

```
concat([],L,L).
concat([H|T],L,[H|R]) <-- concat(T,L,R).
?- concat(L1,[2],L).
```

the query results in the sequence of answers.
Logic Programming

L₁ = [ ], L = [2]
L₁ = [V₁], L = [V₁, 2]
L₁ = [V₁, V₂], L = [V₁, V₂, 2]

... However, if the order of the rules defining $append$ are interchanged,

append([H|T],L,[H|R]) :- append(T,L,R).
append([ ],L,L).
?- append(L₁,[2],L).

then the execution fails to terminate, entering an infinite loop since the first rule is always applicable.

Historical Perspectives and Further Reading

- Intuitionistic mathematics and proof theory.
- Literal normal form, conjunctive normal form and Horn Clause Logic
- Robinson's unification algorithm and the resolution principle. Two terms are said to be unifiable iff there is are substitutions which applied to each makes them the same.
- Kowalski normal form -- Kowalski
- Definite clause grammars -- Colmerauer
- Relational Data Bases -- Codd

Relations and the Relational Algebra
DataLog
- Prolog -- Colmerauer, Warren (David)

Logic programming languages are abstractions and generalization of tuples (relations). History

- Aristotle: (384-322 BCE) -- Theory of syllogistic
- Liebniz (1646-1716): De Arte Combinatoriua 1666 -- calculus of reasoning
- Boole: 1854 -- Boolean logic
- Frege: 1879 Begriffsschrift -- separation of logic from mathematics
- Russell, B, & Whitehead, A. N.: 1910-13 -- Logicism (reduction of mathematics to logic)
- Hilbert, David: 1900 -- Formalism (finitary proofs of consistency)
- Brouwer, L.E.J.: (1881-1966) -- Intuitionism (mathematical certitude is in intuition & explicit construction)
- Gödel, Kurt: 1933 -- incompleteness theorem
- Tarski, Alfred: 1936 -- separation of logic and models
- Church, Alonso: 1936 -- non-termination of proof algorithm for non-theorems
- Robinson, J. Alan: 1965 -- resolution principle
Logic Programming

- Kowalski, Robert: 1974 -- predicate logic as a programming language
- Tärnlund, S-A: 1977 -- Horn clause computability
- Pereira, Fernando: -- implementation of Prolog
- Warren, David: -- implementation of Prolog, Warren abstract machine (WAM)
- Classical logic (propositional, predicate/first-order )
- Other logics: Fuzzy, non-monotonic

Future

- Improved implementations: the Gödel programming language
- Combination of logic and functional paradigms: the Escher programming language

Integration of Database management systems and logic programming and parallel programming languages based on the logic paradigm. References


1969 J Robinson and Resolution 1972 Alain Colmerauer History Kowalski’s paper\cite{Kowalski79}
Logic programming techniques Implementation of Prolog SQL DCG

Exercises

1. Modify concat to include an explicit occurs check.
2. Construct a Prolog based family database. Include the following relations: parentof, grandparentof, ancestorof, uncleof, auntof, and any others of your choice.
3. The relational algebra is ... query languages of relational database management systems is another approach to the logic model.

The fundamental entity in a relational database is a relation which is viewed as a table of rows and columns, where each row, called a tuple, is an object and each column is an attribute or property of the object.

A database consists of one or more relations. The data stored in the relations is manipulated using commands written in a query language. The operations provided the query language include union, set difference, cartesian product, projection, and selection. The first-order predicate logic can be used to represent knowledge and as a language for expressing operations on relations. -- Ullman (Principles of Database and Knowledge-base Systems) CSP 1988.
The tables of a relational database are represented as Prolog facts.

The Relational algebra implemented via Prolog rules and queries.

- **Selection:**

  
  
  select( variables ) :- conditions on the constants.
  
  Constants select rows in the relation.

- **Intersection:**

  
  
  r_1 \cap r_2( Vars ) :- r_1( Vars ), r_2( Vars ).
  
  selects the entities that are in both r_1 and r_2 (use the same variables).

- **Difference:**

  
  
  diff_r_1_r2( Vars ) :- r_1( Vars ), not r_2( Vars ).
  
  selects the entities in r_1 that are not in r_2.

- **Projection:**

  
  pr( variables ) :- r( variables and don't cares ).
  
  Don't cares represent columns to be deleted.

- **Cartesian product:**

  
  prod( variables ) :- r_1( vars ), r_2( vars ).
  
  prod variables is the list of variables both in r_1 and r_2.

- **Union:** (two rules are required to perform union.)

  
  union( variables ) :- first_relation( variables ).
  
  union( variables ) :- second_relation( variables ).

- **Natural Join:** In the rule,

  
  
  nat_join( variables shared variables ) :-
  
  r_1( variables, shared variables ),
  
  r_2( variables, shared variables ).

  
  
  the shared variables restrict search to common elements, reduced number of
  variables in the join eliminate multiple columns.

4. Construct a family data base f_db(f,m,c,sex) and define the following relations, f_of, m_of, son_of, dau_of, gf, gm, aunt, uncle, ancestor, half_sis, half_bro.

5. Business Data base

6. Blocks World
7. CS Degree requirements; course(dept,name,prereq). don't forget w1 and w2 requirements.
8. Circuit analysis
9. Tail recursion
10. Compiler
11. Interpreter
12. Tic-Tac-Toe
13. DCG
15. Airline reservation system

© 1996 by A. Aaby

Objective:

- Atoms and Terms
- Relations, predicates and facts
- Queries
- Terms, logical variable, substitutions, instances
- Unification and the MGU
- Variables and Quantification
- Rules
- Inference
- Abstract Interpreter for Logic Programs
- The Meaning of Logic Programs

The logical variable, substitutions and instances

Rules

A :- B1, B2, ..., Bn.

A is the head. B1, B2, ..., Bn. is the body. Facts, rules, and queries are also called Horn clauses or just clauses. Facts are also called unit clauses.

ancestor(X,Y) :- father(X,Y).
ancestor(X,Z) :- father(X,Y), ancestor(Y,Z).

How to read rules. For every X and Y, if X is the father of Y and Y is an ancestor of Z, X is the ancestor
of Z. For every X and Y, X is the ancestor of Z, if X is the father of Y and Y is an ancestor of Z.

\[
\text{father}(\text{bill}, \text{jim}). \\
\text{father}(\text{jim}, \text{jane}). \\
?- \text{father}(\text{bill}, Y), \text{father}(Y, \text{jane}).
\]

Operationally, to solve a conjunctive query, a single substitution must be found applicable to each conjunct.

For A1,...,An and theta
A1theta,...,Antheta each is deducible.

**Quantifiers**

Variables in queries are existentially quantified. Operationally, to answer a query, using a program, is to perform a computation whose output is the substitution that unifies the query with an instance of the query which is deducible from the program. Note that it may be possible to compute more than one substitution. Variables occurring in facts are universally quantified.

\[
\text{father}(\text{adam}, X).
\]

Variables in the body of a rule are read as universally quantified outside the rule and read as existentially quantified inside the rule. For every X and Y, if X is the father of Y and Y is an ancestor of Z, X is the ancestor of Z. if there exist X and Y such that X is the father of Y and Y is an ancestor of Z, then X is the ancestor of Z.

\[
\text{father}(\text{bill}, \text{jim}). \\
\text{father}(\text{jim}, \text{jane}). \\
?- \text{father}(\text{bill}, Y), \text{father}(Y, \text{jane}).
\]

Operationally, to solve a conjunctive query, a single substitution must be found applicable to each conjunct.

For A1,...,An and theta
A1theta,...,Antheta each is deducible.
A functional program consists of an expression E (representing both the algorithm and the input). This expression E is subject to some rewrite rules. Reduction consists of replacing some part P of E by another expression P' according to the given rewrite rules. ... This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. The expression E* thus obtained is called the normal form of E and constitutes the output of the functional program -H. P. Barendregt

Functional programming is characterized by the programming with values, functions and functional forms.

Keywords and phrases: Lambda calculus, free and bound variables, scope, environment, functional programming, combinatorial logic, recursive functions, functional, curried function.

Functional programming languages are the result of both abstracting and generalizing the data type of maps. Recall, the mapping m from each element x of S (called the domain) to the corresponding element m(x) of T (called the range) is written as:

\[ m : S \rightarrow T \]

For example, the squaring function is a function of type:

\[ \text{sqr} : \text{Num} \rightarrow \text{Num} \]

and may be defined as:

\[ \text{sqr \ where \ } x \mid\mapsto x^2 \]

A linear function f of type

\[ f : \text{Num} \rightarrow \text{Num} \]

may be defined as:

\[ f \ where \ x \mid\mapsto 3x + 4 \]
The function:

\[ g \text{ where } x \rightarrow 3x^2 + 4 \]

may be written as the composition of the functions \( f \) and \( \text{sqr} \) as:

\[ f \circ \text{sqr} \]

where

\[ f \circ \text{sqr} (x) = f(\text{sqr}(x)) = f(x^2) = 3 \cdot x^2 + 4 \]

The compositional operator is an example of a \textit{functional form}. Functional programming is based on the mathematical concept of a function and functional programming languages include the following:

- A set of primitive functions.
- A set of functional forms.
- The \textit{application} operation.
- A set of data objects and associated functions.
- A mechanism for binding a name to a function.

LISP, FP, Scheme, ML, Miranda and Haskell are just some of the languages to implement this elegant computational paradigm.

The basic concepts of functional programming originated with LISP. Functional programming languages are important for the following reasons.

- Functional programming dispenses with the assignment command freeing the programmer from the rigidly sequential mode of thought required with the assignment command.
- Functional programming encourages thinking at \textit{higher levels of abstraction} by providing higher-order functions -- functions that modify and combine existing programs.
- Functional programming has natural implementation in concurrent programming.
- Functional programming has important application areas. Artificial intelligence programming is done in functional programming languages and the AI techniques migrate to real-world applications.
- Functional programming is useful for developing \textit{executable specifications} and \textit{prototype implementations}.
- Functional programming has a close relationship to computer science theory. Functional programming is based on the lambda-calculus which in turn provides a framework for studying decidability questions of programming. The essence of denotational semantics is the translation of conventional programs into equivalent functional programs.
**Terminology.** Functional programming languages are called *applicative* since the functions are applied to their arguments, *declarative* and *non-procedural* since the definitions specify what is computed and not how it is computed.

## 1 The Lambda Calculus

Functional programming languages are based on the lambda-calculus. The lambda-calculus grew out of an attempt by Alonzo Church and Stephen Kleene in the early 1930s to formalize the notion of computability (also known as *constructibility* and *effective calculability*). It is a formalization of the notion of functions as rules (as opposed to functions as tuples). As with mathematical expressions, it is characterized by the principle that the *value of an expression depends only on the values of its subexpressions*. The lambda-calculus is a simple language with few constructs and a simple semantics. But, it is expressive; it is sufficiently powerful to express all computable functions.

As an informal example of the lambda-calculus, consider the function defined by the polynomial expression

\[ x^2 + 3x - 5. \]

The variable \( x \) is a parameter. In the lambda-calculus, the notation \( \lambda x.M \) is used to denote a function with parameter \( x \) and body \( M \). That is, \( x \) is mapped to \( M \). We rewrite our function in this format

\[ \lambda x.(x^2 + 3x - 5) \]

and read it as "the function of \( x \) whose value is defined by \( x^2 + 3x - 5 \)." The lambda-calculus uses prefix form and so we rewrite the body in prefix form,

\[ \lambda x.(- (+ (\times x x) (\times 3 x)) 5). \]

The lambda-calculus *curries* its functions of more than one variable i.e. \( (+ x y) \) is written as \( ((+ x) y) \), the function \( (+ x) \) is the function which adds something to \( x \). Rewriting our example in this form we get:

\[ \lambda x.((- (((+ (\times x x)) ((\times 3) x))) 5) \]

To denote the application of a function \( f \) to an argument \( a \) we write

\[ f a \]

To apply our example to the value 1 we write

\[ \lambda x.((-(+((\times x x))((\times 3) x))) 5) 1. \]
To evaluate the function application, we remove the \( \lambda x. \) and replace each remaining occurrence of \( x \) with 1 to get

\[
((- ((+ ((\times 1) 1)) ((\times 3) 1))) 5)
\]

then evaluate the two multiplication expressions

\[
((- ((+ 1) 3)) 5)
\]

then the addition

\[
((- 4) 5)
\]

and finally the subtraction

\[-1.\]

We say that the variable \( x \) is *bound* in the lambda-expression \( \lambda x. B \). A variable occurring in the lambda-expression which is not bound is said to be *free*. The variable \( x \) is free in the lambda-expression \( \lambda y.((+ x) y) \). The *scope* of the variable introduced (or bound) by lambda is the entire body of the lambda-abstraction.

The lambda-notation extends readily to functions of several arguments. Functions of more than one argument can be *curried* to produce functions of single arguments. For example, the polynomial expression \( xy \) can be written as

\[
\lambda x. \lambda y. xy
\]

When the lambda-abstraction \( \lambda x. \lambda y. xy \) is applied to a single argument as in \( \lambda x. \lambda y. xy 5 \) the result is \( \lambda y. 5y \), a function which multiplies its argument by 5. A function of more than one argument is regarded as a *functional* of one variable whose value is a function of the remaining variables, in this case, ``multiply by a constant function.''

The special character of the lambda-calculus is illustrated when it is recognized that functions may be applied to other functions and even permit self application. For example let \( C = \lambda f. \lambda x . (f(fx)) \)

The pure lambda-calculus does not have any built-in functions or constants. Therefore, it is appropriate to speak of the lambda-calculi as a family of languages for computation with functions. Different languages are obtained for different choices of functions and constants.

We will extend the lambda-calculus with common mathematical operations and constants so that \( \lambda x.((+ \)
3) x) defines a function that maps x to x+3. We will drop some of the parentheses to improve the readability of the lambda expressions.

A lambda-expression is executed by evaluating it. Evaluation proceeds by repeatedly selecting a reducible expression (or redex) and reducing it. For example, the expression (+ (* 5 6) (* 8 3)) reduces to 54 in the following sequence of reductions.

\[
(+ (* 5 6) (* 8 3)) \quad \rightarrow \quad (+ 30 (* 8 3)) \\
\rightarrow \quad (+ 30 24) \\
\rightarrow \quad 54
\]

When the expression is the application of a lambda-abstraction to a term, the term is substituted for the bound variable. This substitution is called \beta-reduction. In the following sequence of reductions, the first step an example of \beta-reduction. The second step is the reduction required by the addition operator.

\[
(\lambda x.(+ 3) x) 4 \\
( (+ 3) 4 )
\]

7

The pure lambda-calculus has just three constructs: primitive symbols, function application, and function creation. Figure N.1 gives the syntax of the lambda-calculus.

\[\text{Figure N.1: The Lambda Calculus}\]

\[\text{Syntax:}\]

L in Lambda Expressions
x in Symbols

\[L ::= x | (L L) | (\lambda x.L)\]

(L L) is function application, and
We say that the variable \( x \) is *bound* in the lambda-expression \( \lambda x . B \). A variable which occurs in but is not bound in a lambda-expression is said to be *free*. The *scope* of \( \lambda x . \) is \( B \). In the lambda-expression \( \lambda y . x + y \), \( x \) is free and \( y \) is bound.

We adopt the following notational conventions:

- We extend the lambda-calculus with the usual constants and functions so we allow

  \( (\lambda x . ((+ x) 3)) \) to represent the function \( x + 3 \)

- We usually drop the outermost parentheses so we may write

  \( \lambda x . ((+ x) 3) \) instead of \( (\lambda x . ((+ x) 3)) \) and
  \( \lambda x . ((+ x) 3) 4 \) instead of \( (\lambda x . ((+ x) 3) 4) \)

- Function application associates to the left so we may write

  \( (+ x 3) \) instead of \( ((+ x) 3) \) that is, we may write
  \( \lambda x . + x 3 \) instead of \( \lambda x . ((+ x) 3) \)

- The body of a lambda-abstraction extends as far right as possible so we must write

  \( (\lambda x . + x 3) 4 \) instead of \( \lambda x . + x 3 4 \)

- Replace the body of a lambda-abstraction with conventional infix notation so we may write

  \( (\lambda x . x + 3) 4 \) instead of \( (\lambda x . + x 3) 4 \)

- Multiple parameters are written together so we may write

  \( \lambda y . x + y \) instead of \( \lambda y . x + y \)

**Operational Semantics**

Calculation in the lambda-calculus is by rewriting (reducing) a lambda-expression to a normal form. For the pure lambda-calculus, lambda-expressions are reduced by substitution. That is, occurrences of the parameter in the body are replaced with (copies of) the argument. In our extended lambda-calculus we
also apply the usual reduction rules. For example,

1. \( \lambda x (x^2 - 5) \) \( 3 \) \( f(3) \) where \( f(x) = x^2 - 5 \)
2. \( 3^2 - 5 \) by substitution
3. \( 9 - 5 \) power
4. \( 4 \) subtraction

The normal form is formally defined in the following definition.

**Definition:** A lambda-expression is said to be in **normal form** if no **beta-redex**, a subexpression of the form \( (\lambda x.P \ Q) \), occurs in it.

Non-terminating computations are examples of expressions that do not have normal forms. The lambda-expression

\[
(\lambda x \ x \ x) \ (\lambda x \ x \ x)
\]

does not have a normal form as we shall soon see.

We define substitution, \( B[x:M] \), to be the replacement of all free occurrences of \( x \) in \( B \) with \( M \). Figure N.2 contains a formal definition of substitution.

---

**Figure N.2: Substitution**

\[
\begin{align*}
  s[x:M] & = \text{if } (s=x) \text{ then } M \text{ else } s \\
  (A \ B)[x:M] & = (A[x:M] \ B[x:M]) \\
  (\lambda x.B)[x:M] & = (\lambda x.B) \\
  (\lambda y.B)[x:M] & = \text{if } (z \text{ is a symbol not free in } B \text{ or } M) \text{ then } \lambda z.(B[y:z][x:M])
\end{align*}
\]

where \( s \) is a symbol, \( M, A \) and \( B \) are lambda-expressions.

---

Lambda expressions are simplified using beta-reduction. Beta-reduction applies a lambda-abstraction to an argument producing an instance of the body of the lambda-abstraction in which (free) occurrences of the formal parameter in the body are replaced with (copies of) the argument. With the definition of substitution in Figure N.2 and the formal definition of beta-reduction in Figure N.3, we have the tools needed to reduce lambda-expressions to normal forms.
Figure N.3: **Beta-reduction**

\[(\lambda x. B) e \Rightarrow B[x:=e]\]

It is easy to see that the lambda-expression

\[(\lambda x. x) (\lambda x. x)\]

does not have a normal form because when the second expression is substituted into the first, the resulting expression is identical to the given lambda-expression.

Figure 2 defines the operational semantics of the lambda-calculus in terms of beta-reduction.

---

Figure N.4: **Operational semantics for the lambda-calculus**

Interpreter: reduce expression E to normal form.

**Reduce** in L \(\Rightarrow\) L

- \(\text{Reduce}[s] = s\)
- \(\text{Reduce}[\lambda x. B M] = \text{Reduce}[B[x:=M]]\)
- \(\text{Reduce}[L_1 L_2] = (\text{Reduce}[L_1] \ \text{Reduce}[L_2])\)

where

- s is a symbol and B, L_1, L_2, and M are lambda-expressions

The operational semantics of Figure N.4 describe a syntactic transformation of the lambda-expressions.

**Reduction Order**

Given a lambda-expression, the substitution and beta-reduction rules provide the tools required to reduce
a lambda-expression to normal form but do not tell us what order to apply the reductions when more
than one redex is available. The following theorem, due to Curry, states that if an expression has a
normal form, then that normal form can be found by leftmost reduction.

**Theorem:** If E has a normal form N then there is a leftmost reduction of E to N.

The leftmost outermost reduction (*normal order reduction*) strategy is called *lazy reduction* because it
does not first evaluate the arguments but substitutes the arguments directly into the expression. *Eager
reduction* is when the arguments are reduced before substitution.

A function is *strict* if it is sure to need its argument. If a function is non-strict, we say that it is *lazy*.

parameter passing: by value, by name, and lazy evaluation

Infinite Data Structures

call by need

streams and perpetual processes

A function f is *strict* if and only if \( f(\bot) = \bot \)

Scheme evaluates its parameters before passing (eliminates need for renaming) a space and time
efficiency consideration.

**Denotational Semantics**

In the previous section we looked at the *operational* semantics of the lambda-calculus. It is called
operational because it is `dynamic', it sees a function as a sequence of operations. A lambda-expression
was evaluated by purely *syntactic* transformations without reference to what the expressions `mean'. The
purpose of the *denotational semantics* of a language is to assign a value to every expression in the
language.

We can express the semantics of the lambda-calculus as a mathematical function, \( \text{Eval} \), from
expressions to values. For example,

\[
\text{Eval}[+ 3 4] = 7
\]
defines the value of the expression \(+ 3 4\) to be 7. Actually something more is required, in the case of
variables and function names, the function \( \text{Eval} \) requires a second parameter containing the environment
\( \rho \) which contains the associations between variables and their values. Some programs go into infinite
loops, some abort with a runtime error. To handle these situations we introduce the symbol \( \bot \)
pronounced `bottom'.

Figure N.5 gives a denotational semantics for the lambda-calculus.

---

Figure N.5: **Denotational semantics for the lambda-calculus**

Semantic Domains:

\[ s \text{ in } D \]

Semantic Function:

\[ Eval \text{ in } L \rightarrow D \]

Semantic Equations:

\[
\begin{align*}
Eval \ [ s ] &= s \\
Eval \ [ (\lambda x.B \ M) ] &= Eval \ [ B[x:M] ] \\
Eval \ [ (L_1 \ L_2) ] &= (Eval \ [ L_1 ] \ Eval \ [ L_2 ]) \\
Eval \ [ E ] &= \bot
\end{align*}
\]

where \( s \) is a symbol, \( B, L_1, L_2, \) and \( M \) are expressions, \( B[x:M] \) is substitution as in Figure N.2, \( E \) is an expression which does not have a normal form, and \( \bot \) is pronounced bottom.

---

The denotational semantics of Figure N.5 describe a mapping of lambda expressions to values in some semantic domain.

**Recursive Functions**

We extend the syntax of the lambda-calculus to include named expressions as follows:

**Lambda Expressions**

\[ L ::= \cdots \mid x : L \mid \cdots \]
where x is the name of the lambda-expression L.

With the introduction of named expressions we have the potential for recursive definitions since the extended syntax permits us to name lambda-abstractions and then refer to them within a lambda-expression. Consider the following recursive definition of the factorial function.

\[ \text{FAC} : \lambda n. (\text{if } (n = 0) 1 (* n (\text{FAC} (- n 1)))) \]

which with syntactic sugaring is

\[ \text{FAC} : \lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } (n * \text{FAC} (n - 1)) \]

We can treat the recursive call as a free variable and replace the previous definition with the following.

\[ \text{FAC} : (\text{\text{\text{\lambda}}} \text{fac.} \lambda n. (\text{if } (n = 0) (* n (\text{fac} (- n 1)))))) \text{ FAC} \]

Let

\[ \text{H} : \lambda \text{fac.} \lambda n. (\text{if } (n = 0) 1 (* n (\text{fac} (- n 1)))) \]

Note that H is not recursively defined. Now we can redefine FAC as

\[ \text{FAC} : (\text{H} \text{ FAC}) \]

This definition is like a mathematical equation. It states that when the function H is applied to FAC, the result is FAC. We say that FAC is a fixed point or fixpoint of H. In general functions may have more than one fixed point. In this case the desired fixed point is the mathematical function factorial. In general, the `right' fixed point turns out to be the unique least fixed point.

It is desirable that there be a function which applied to a lambda-abstraction returns the least fixed point of that abstraction. Suppose there is such a function Y where,

\[ \text{FAC} : Y \text{ H} \]

Y is called a fixed point combinator. With the function Y, this definition of FAC does not use of recursion. From the previous two definitions, the function Y has the property that

\[ Y \text{ H} = H (Y \text{ H}) \]

As an example, here is the computation of FAC 1 using the Y combinator.
The function $Y$ can be defined in the lambda-calculus.

$$Y : \lambda h. (\lambda x. (h (x \, x)) \, \lambda x. (h (x \, x)))$$

It is especially interesting because it is defined as a lambda-abstraction without using recursion. To show that this lambda-expression properly defines the $Y$ combinator, here it is applied to $H$.

$$(Y \, H) = (\lambda h. (\lambda x. (h (x \, x)) \, \lambda x. (h (x \, x)))) \, H$$

$$= (\lambda x. (H (x \, x)) \, \lambda x. (H (x \, x)))$$

$$= H \, (\lambda x. (H (x \, x)) \, \lambda x. (H (x \, x)))$$

$$= H \, (Y \, H)$$

**Lexical Scope Rules**

Blocks with local definitions may be defined in the lambda-calculus. We introduce two kinds of blocks, let and letrec expressions. Nonrecursive definitions are introduced with let expressions:

$$\text{let } n : E \text{ in } B \text{ is an abbreviation for } (\lambda n. B) \, E$$

Here is an example using the let-extension.

$$\text{let } x : 3 \text{ in } (\star \, x \, x)$$

Lets may be used where ever a lambda-expression is permitted. For example,

$$\lambda y. \text{let } x : 3 \text{ in } (\star \, y \, x)$$

is equivalent to
Simple recursive definitions are introduced with letrec expressions which are defined in terms of let expressions and the Y combinator:

\[ \text{letrec } n : E \text{ in } B \] is an abbreviation for \[ \text{let } n : Y (\!:\!n. E) \text{ in } B \]

Let and letrec expressions may be nested. The definitions of the let and letrec expressions are restated in Figure N.6.

---

**Figure M.6: Lexical Scope Rules**

\[
\begin{align*}
\text{let } n : E \text{ in } B &= (\!:\!n. B) E \\
\text{letrec } n : E \text{ in } B &= \text{let } n : Y (\!:\!n. E) \text{ in } B
\end{align*}
\]

---

Mutual recursion may also be defined but is beyond the scope of this text.

**Translation Semantics and Combinators**

The beta-reduction rule is expensive to implement. It requires the textual substitution of the argument for each occurrence of the parameter and further requires that no free variable in the argument should become bound. This has lead to the study of ways in which variables can be eliminated.

Curry, Feys, and Craig define a number of *combinators* among them the following:

\[
\begin{align*}
S &= \!:\!f.(\!:\!g.( \!:\!x. f x ( g x ) ) ) \\
K &= \!:\!x. \!:\!y. x \\
I &= \!:\!x. x \\
Y &= \!:\!f. \!:\!x.( f(x x) ) \!:\!x.(f (x x))
\end{align*}
\]

These definitions lead to transformation rules for sequences of combinators. The reduction rules for the SKI calculus are given in Figure N.7.
Functional Programming

Figure N.7: Reduction rules for SKI calculus

\[ S f g x \rightarrow f (g x) \]
\[ K c x \rightarrow c \]
\[ I x \rightarrow x \]
\[ Y e \rightarrow e (Y e) \]
\[ (A B) \rightarrow A B \]
\[ (A B C) \rightarrow A B C \]

The reduction rules require that reductions be performed left to right. If no S, K, I, or Y reduction applies, then brackets are removed and reductions continue.

The SKI calculus is computationally complete; that is, these three operations are sufficient to implement any operation. This is demonstrated by the rules in Figure N.8.

Figure N.8: Translation Semantics for the Lambda calculus

\[
\begin{align*}
\text{Compile } [ s ] & \rightarrow s \\
\text{Compile } [ (E_1 E_2) ] & \rightarrow (\text{Compile } [ E_1 ] \text{ Compile } [ E_2 ]) \\
\text{Compile } [ \lambda x. E ] & \rightarrow \text{Abstract } [ (x, \text{ Compile } [ E ]) ] \\
\text{Abstract } [ (x, s) ] & \rightarrow \text{if (s=x) then I else (K s) } \\
\text{Abstract } [ (x, (E_1 E_2))] & \rightarrow ((S \text{ Abstract } [ (x, E_1)] ) \text{ Abstract } [ (x, E_2) ] )
\end{align*}
\]

where \( s \) is a symbol.

which translate lambda-expressions to formulas in the SKI calculus.

Any functional programming language can be implemented by a machine that implements the SKI combinators since, functional languages can be transformed into lambda-expressions and thus to SKI formulas.

Function application is relatively expensive on conventional computers. The principle reason is the
complexity of maintaining the data structures that support access to the bound identifiers. The problems are especially severe when higher-order functions are permitted. Because a formula of the SKI calculus contains no bound identifiers, its reduction rules can be implemented as simple data structure manipulations. Further, the reduction rules can be applied in any order, or in parallel. Thus it is possible to design massively parallel computers (graph reduction machines) that execute functional languages efficiently.

Recursive functions may be defined with the Y operator.

**Optimizations**

Notice that the size of the SKI code grows quadratically in the number of bound variables. Figure N.9.

\[ B = \lambda x . (\lambda y . (\lambda z. ((x y) z))) \]
\[ C = \lambda x . (\lambda y. (\lambda z((x z) y))) \]

with the corresponding reduction rules.

\[ B a b c \rightarrow ((a b) c) \]
\[ C a b c \rightarrow ((a c) b) \]

Having these combinators we can simplify the expressions obtained by applying the rules in Figure N.9.

---

**Figure N.9: Optimizations for SKI code**

\[
\begin{align*}
S (K e) (K f) & \rightarrow K (e f) \\
S (K e) I & \rightarrow e \\
S (K e) f & \rightarrow (B e) f \\
S e (K f) & \rightarrow (C e) f
\end{align*}
\]

The optimizations must be applied in the order given.

---

Just as machine language (assembler) can be used for programming, combinatorial logic can be used as a programming language. The programming language FP is a programming language based on the idea of combinatorial logic.
2 Scheme

Scheme, a descendant of LISP, is based on the lambda-calculus. Although it has imperative features, in this section we ignore those features and concentrate on the lambda-calculus like features of Scheme. Scheme has two kinds of objects, atoms and lists. Atoms are represented by strings of non-blank characters. A list is represented by a sequence of atoms or lists separated by blanks and enclosed in parentheses. Functions in Scheme are also represented by lists. This facilitates the creation of functions which create other functions. A function can be created by another function and then the function applied to a list of arguments. This is an important feature of languages for AI applications.

Syntax

The syntax of Scheme is similar to that of the lambda calculus.

Scheme Syntax

\[
E \in \text{Expressions} \\
A \in \text{Atoms ( variables and constants )} \\
\ldots \\
E ::= A \mid (E...) \mid (\text{lambda (A...) E}) \mid \ldots
\]

Expressions are atoms which are variables or constants, lists of arbitrary length (which are also function applications), lambda-abstractions of one or more parameters, and other built-in functions.

Scheme provides a number of built in functions among which are +, -, *, /, <, <=, =, >=, >, and not. Scheme provides for conditional expressions of the form (if $E_0$ $E_1$ $E_2$) and (if $E_0$ $E_1$). Among the constants provided in Scheme are numbers, #f and the empty list () both of which count as false, and #t and any thing other than #f and () which count as true. nil is also used to represent the empty list.

Definitions

Scheme implements definitions with the following syntax

\[
E ::= \ldots | (\text{define I E}) \mid \ldots
\]

Lists with nil, cons, car and cdr

The list is the basic data structure with nil representing the empty list. Among the built in functions for list manipulation provided in Scheme are cons for attaching an element to the head of a list, car for extracting the first element of a list, and cdr which returns a list minus its first element.
Figure N.10: **Stack operations in Scheme**

( define empty_stack
  ( lambda ( stack ) ( if ( null? stack ) \#t \#f )))

( define push
  ( lambda ( element stack ) ( cons element stack )))

(define pop
  ( lambda ( element stack ) ( cdr stack )))

(define top
  ( lambda ( stack ) ( car stack )))

Figure N.10 contains an example of stack operations written in Scheme. The figure illustrates definitions, the conditional expression, the list predicate null? for testing whether a list is empty, and the list manipulation functions cons, car, and cdr.

**Local Definitions**

Scheme provides for local definitions with the following syntax

**Scheme Syntax**

...  
B in Bindings  
...  

E ::= ... | (let B E0) | (let* B1 E1) | (letrec B2 E2) ...  
B ::= ((I E)...)

The let definitions are done independently of each other (collateral bindings), the let* values and bindings are computed sequentially and the letrec bindings are in effect while values are being computed to permit mutually recursive definitions.

3 ML
4 Haskell

In contrast with LISP and Scheme, Haskell is a modern functional programming language.

---

**Figure N.11: A sample program in Haskell**

```haskell
module AStack( Stack, push, pop, top, size ) where
data Stack a = Empty
    | MkStack a (Stack a)
push :: a -> Stack a -> Stack a
push x s = MkStack x s

size :: Stack a -> Integer
size s = length (stkToLst s) where
    stkToLst Empty         = []
    stkToLst (MkStack x s) = x:xs where xs =

pop :: Stack a -> (a, Stack a)
pop (MkStack x s) = (x, case s of r -> i r where i x = x)

top :: Stack a -> a
top (MkStack x s) = x

---

module Qs where

qs :: [Int] -> [Int]
qs [] = []
qs (a:as) = qs [x | x <- as, x <=a] ++ [a] ++ qs [x | x <- as, x> a]

module Primes where

primes :: [Int]
primes = map head (iterate sieve [2 ..])

sieve :: [Int] -> [Int]
sieve (p:ps) = [x | x <- ps, (x `mod` p)=0]
```

http://cs.wwc.edu/~aabyan/PLBook/HTML/Functions.html (18 of 22)8/10/2006 8:27:52 PM
module Fact where

fact :: Integer -> Integer
fact 0 = 1
fact (n+1) = (n+1)*fact n -- * "Foo"
fact _ = error "Negative argument to factorial"

module Pascal where

pascal :: [[Int]]
pascal = [1] : [[x+y | (x,y) <- zip ([0]++r) (r++[0])] | r <- pascal]
tab :: Int -> ShowS
tab 0 = \x -> x
tab (n+1) = showChar ' ' . tab n

showRow :: [Int] -> ShowS
showRow [] = showChar '\n'
showRow (n:ns) = shows n . showChar ' ' . showRow ns

showTriangle 1 (t:_:ts) = showRow t
showTriangle (n+1) (t:ts) = tab n . showRow t . showTriangle n ts

module Merge where

merge :: [Int] -> [Int] -> [Int]
merge [] x = x
merge x [] = x
merge l1@(a:b) l2@(c:d) = if a < c then a:(merge b l2)
else c:(merge l1 d)

half [] = []
half [x] = [x]
half (x:y:z) = x:r where r = half z

sort [] = []
sort [x] = [x]
sort l = merge (sort odds) (sort evens) where
  odds = half l
evens = half (tail l)

5 Historical Perspectives and Further Reading
In the 1930s Alonso Church developed the lambda-calculus as an alternative to set theory for the foundations of mathematics and Haskell B. Curry developed combinatory logic for the same reason. While their goal was not realized, the lambda-calculus and combinators capture the most general formal properties of the notion of a mathematical function.

The lambda-calculus and combinatory logic are abstract models of computation equivalent to the Turing machine, recursive functions, and Markov chains. Unlike the Turning machine which is sequential in nature, they retain the implicit parallelism that is present in mathematical expressions.

The lambda-calculus is a direct influence on the programming language LISP, the call by name parameter passing mechanism of Algol-60, and textual substitution performed by macro generators.

Explicit and systematic use of the lambda-calculus in computer science was initiated in the early 1960s by Peter Landin, Christopher Strachy and others who started a formal theory of semantics for programming languages called denotational semantics. Dana Scott (1969) discovered the first mathematical model for the type-free lambda-calculus.

New hardware designs are appearing to support the direct execution of the lambda-calculus or combinators which support parallel execution of functional programs, removing the burden (side-effects, synchronization, communication) of controlling parallelism from the programmer.

LISP (LISt Processing) was designed by John McCarthy in 1958. LISP grew out of interest in symbolic computation. In particular, interest in areas such as mechanizing theorem proving, modeling human intelligence, and natural language processing. In each of these areas, list processing was seen as a fundamental requirement. LISP was developed as a system for list processing based on recursive functions. It provided for recursion, first-class functions, and garbage collection. All new concepts at the time. LISP was inadvertently implemented with dynamic rather than static scope rules. Scheme is a modern incarnation of LISP. It is a relatively small language with static rather than dynamic scope rules. LISP was adopted as the language of choice for artificial intelligence applications and continues to be in wide use in the artificial intelligence community.

ML

Miranda

Haskell is a modern language named after the logician Haskell B. Curry, and designed by a 15-member international committee. The design goals for Haskell are have a functional language which incorporates all recent "good ideas" in functional language research and which is suitable for for teaching, research and application. Haskell contains an overloading facility which is incorporated with the polymorphic type system, purely functional i/o, arrays, data abstraction, and information hiding.

Functional programming languages have been presented in terms of a sequence of virtual machines.
Functional programming languages can be translated into the lambda-calculus, the lambda-calculus into combinatory logic and combinatory logic into the code for a graph reduction machine. All of these are virtual machines.

Models of the lambda-calculus.

History \cite{McCarthy60} For an easily accessible introduction to functional programming, the lambda-calculus, combinators and a graph machine implementation see Revesz (1988). For Backus' Turing Award paper on functional programming see \cite{Backus78}. The complete reference for the lambda-calculus is \cite{Bare84}. For all you ever wanted to know about combinatory logic see \cite{CF68,CHS72,HS86}. For an introduction to functional programming see Henderson (1980), BirdWad88, MLennan90. For an introduction to LISP see \cite{McCarthy65} and for common LISP see \cite{Steele84}. For an introduction to Scheme see \cite{AbSus85}. Haskell On the relationship of the lambda-calculus to programming languages see \cite{Landin66}. For the implementation of functional programming languages see Henderson (1980) and Peyton-Jones (1987).

Henderson, Peter (1980)


Peyton-Jones, Simon L (1987)

*The Implementation of Functional Programming Languages* Prentice-Hall International.


6 Exercises

1. [Time/Difficulty](section)
2. Simplify the following expressions to a final (normal) form, if one exists. If one does not exist, explain why.
   1. \[((x. (xy))(z.z))
   2. \[((x. ((y.(xy))x))(z.w))
   3. \(((f.(g.((fx)(gx)))))(m.(n.(nm))))(n.z)p
   4. \(((x. (xx))(x.(xx)))
   5. \(((f.((g.((ff)g))(h.(kh)))))(y.y))
   6. \(((g.((f.((x.(ffx)))((x.f(xx))))))(x.((f(xx))))g))
   7. \(((x.((y.((y)x)))45
   8. \(((f.((3)(x.((1)x)))))
3. Find a lambda-expression that not only does not have a normal form but grows in length as well.
4. In addition to the \beta-rule, the lambda-calculus includes the following two rules:

   \begin{align*}
   \alpha\text{-rule: } (x. E) & \Rightarrow (y. E[x:y]) \\
   \eta\text{-rule: } (x. E x) & \Rightarrow E \text{ where } x \text{ does not occur free in } E
   \end{align*}
Redo the previous exercise making use of the \eta-rule whenever possible. What value is there in the \alpha-rule?

5. The lambda-calculus can be used to simulate computation on truth values and numbers.
   1. Let true be the name of the lambda-expression \( \lambda x. \lambda y. x \) and false be the name of the lambda-expression \( \lambda x. \lambda y. y \). Show that \((\lambda \Box \{\text{true}\} E_1)E_2) \Rightarrow E_1\) and \((\lambda \Box \{\text{false}\} E_1)E_2) \Rightarrow E_2\). Define lambda-expressions not, and, and or that behave like their Boolean operation counterparts.

   2. Let 0 be the name of the lambda-expression \( \lambda x. \lambda y. x \), 1 be the name of the lambda-expression \( \lambda x. \lambda y. (xy) \), 2 be the name of the lambda-expression \( \lambda x. \lambda y. (x(xy)) \), 3 be the name of the lambda-expression succ defined as \( \lambda z. \lambda x. \lambda y. (x ((zx)y)) \), and so on. Prove that the lambda-expression succ rewrites a number to its successor.

6. Recursively defined functions can also be simulated in the lambda-calculus. Let Y be the name of the expression \( \lambda f. \lambda x. (f(xx)) \).
   1. Show that for any expression E, there exists an expression W such that \((Y E) \Rightarrow (WW)\), and that \((WW) \Rightarrow (E(WW))\). Hence, \((Y E) \Rightarrow E(E(...E(WW)...)))\)
   2. Using the lambda-expressions that you defined in the previous parts of this exercise, define a recursive lambda-expression add that performs addition on the numbers defined earlier, that is, \((\text{add} m n) \Rightarrow m+n\).

7. Let T = AA where A = \( \lambda xy. y(xxy) \). Show T F = F (T F). T is Turing's fixed point combinator.

8. Data constructors can be modeled in the lambda-calculus. Let cons = \( \lambda a. \lambda b. \lambda f. f a b \), head = \( \lambda c. c (\lambda a. \lambda b. a) \) and tail = \( \lambda c. c (\lambda a. \lambda b. b) \). Show that
   1. head (cons a b) = a
   2. tail (cons a b) = b

9. Show that \(((S(KK))I)S\) is (KS).

10. What is \(((SI)I)X\) for any formula X?

11. Compile \( \lambda x. +xx \) to SKI code.

12. Compile \( \lambda x. (F (xx)) \) to SKI code.

13. Compile \( \lambda x. \lambda y. xy \) to SKI code. Check your answer by reducing both \( ((\lambda x. \lambda y. xy) a b) \) and the SKI code applied to a b.

14. Apply the optimizations to the SKI code for \( \lambda x. \lambda y. xy \) and compare the result with the unoptimized code.

15. Apply the optimizations to the SKI code for \( \lambda x. (F (xy)) \) and \( \lambda y. (F (xy)) \).

16. Association lists etc

17. HOF

18. Construct an interpreter for the lambda calculus.

19. Construct an interpreter for combinatorial logic.

20. Construct a compiler to compile lambda expressions to combinators.

© 1996 by A. Aaby
The Imperative Programming Paradigm

**Imperative programming is characterized by programming with a state and commands which modify the state.**

**Imperative:**
- a command or order

**Procedure:**
- the act, method or manner of proceeding in some process or course of action
- a particular course of action or way of doing something.

When imperative programming is combined with subprograms it is called procedural programming. In either case the implication is clear. Programs are directions or orders for performing an action.

**Keywords and phrases:** Assignment, goto, structured programming, command, statement, procedure, control-flow, imperative language, assertions, axiomatic semantics. state, variables, instructions, control structures.

The imperative programming paradigm is an abstraction of real computers which in turn are based on the Turing machine and the Von Neumann machine with its registers and store (memory). At the heart of these machines is the concept of a modifiable store. Variables and assignments are the programming language analog of the modifiable store. The store is the object that is manipulated by the program. Imperative programming languages provide a variety of commands to provide structure to code and to manipulate the store. Each imperative programming language defines a particular view of hardware. These views are so distinct that it is common to speak of a Pascal machine, C machine or a Java machine. A compiler implements the virtual machine defined by the programming language in the language supported by the actual hardware and operating system.

In imperative programming, a name may be assigned to a value and later reassigned to another value. The collection of names and the associated values and the location of control in the program constitute the state. The state is a logical model of storage which is an association between memory locations and values. A program in execution generates a sequence of states (See Figure N.1). The transition from one state to the next is determined by assignment operations and sequencing commands.

**Figure N.1:** State Sequence

\[ S_0 \rightarrow O_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{n-1} \rightarrow O_{n-1} \rightarrow S_n \]
Unless carefully written, an imperative program can only be understood in terms of its execution behavior. The reason is that during the execution of the code, any variable may be referenced, control may be transferred to any arbitrary point, and any variable binding may be changed. Thus, the whole program may need to be examined in order to understand even a small portion of code.

Since the syntax of C, C++ and Java are similar, in what follows, comments made about C apply also to C++ and Java.

Variables and Assignment

Imperative programs are characterized by sequences of bindings (state changes) in which a name may be bound to a value at one point in the program and later bound to a different value. Since the order of the bindings affects the value of expressions, an important issue is the proper sequencing of bindings.

Terminology. When speaking of hardware we use terms like *bit pattern*, *storage cell*, and *storage address*. Somewhat analogous terms in programming languages are *value*, *variable*, and *name*. Since variables are usually bound to a name and to a value, the word variable is often used to mean the name of a value.

Aside. Most descriptions of imperative programming languages are tied to hardware and implementation considerations where a name is bound to an address, a variable to a storage cell, and a value to a bit pattern. Thus, a name is tied to two bindings, a binding to a location and to a value. The location is called the l-value and the value is called the r-value. The necessity for this distinction follows from the implementation of the assignment. For example,

\[ X := X + 2 \]

the X on the left of the assignment denotes a location while the X on the right hand side denotes the value. Assignment changes the value at a location.

A variable may be bound to a hardware location at various times. It may be bound at compile time (rarely), at load time (for languages with static allocation) or at run time (for languages with dynamic allocation). From the implementation point of view, variable declarations are used to determine the amount of storage required by the program.

The following examples illustrate the general form for variable declarations in imperative programming languages.

**Pascal style declaration:** var *name* : *Type*;

**C style declaration:** *Type* *name*;

A variable and a value are **bound** by an assignment. A variety of notations is used to indicate the
The imperative programming paradigm

binding of a variable $V$ and the value of an expression $E$.

Pascal  $V := E$
C       $V = E$
APL     $V <-- E$
Scheme  (setq V E)

Aside. The use of the assignment symbol, =, in C confuses the distinction between
definition, equality and assignment. The equal symbol, =, is used in mathematics in two
distinct ways. It is used to define and to assert the equality between two values. In C it
neither means define nor equality but assign. In C the double equality symbol, ==, is used
for equality, while the form: \textit{type variable; } is used for definitions.

The \textbf{assignment} command is what distinguishes imperative programming languages from other
programming languages. The assignment typically has the form:

\[ V := E. \]

The command is read "assign the name $V$ to the value of the expression $E$ until the name $V$ is reassigned
to another value". The assignment binds a name and a value.

Aside. The word "assign" is used in accordance with its English meaning; a name is
assigned to an object, not the reverse. The name then stands for the object. The name is
the assignee. This is in contrast to wide spread programming usage in which a value
assigned to a variable.

The assignment is not the same as a constant definition because it permits redefinition. For example, the
two assignments:

\[
\begin{align*}
X & := 3; \\
X & := X + 1
\end{align*}
\]

are understood as follows: assign $X$ to three and then reassign $X$ to the value of the expression $X+1$
which is four. Thus, after the sequence of assignments, the value of $X$ is four.

Several kinds of assignments are possible. Because of the frequent occurrence of assignments of the
form: $X := X \; op \; E$, C provides an alternative notation of the form: $X \; op= \; E$. A \textit{multiple
assignment} of the form:

\[ V_0 := V_1 := \ldots := V_n := E \]
causes several names to be assigned to the same value. This form of the assignment is found in C. A *simultaneous assignment* of the form:

\[ V_0, \ldots, V_n := E_0, \ldots, E_n \]

causes several assignments of names to values to occur simultaneously. The simultaneous assignment permits the swapping of values without the explicit use of an auxiliary variable.

From the point of view of axiomatic semantics, the assignment is a predicate transformer. It is a function from predicates to predicates. From the point of view of denotational semantics, the assignment is a function from states to states. From the point of view of operational semantics, the assignment changes the state of an abstract machine.

### Unstructured Commands

Given the importance of sequence control, it is not surprising that considerable effort has been given to finding appropriate control structures. Figure N.M gives a minimal set of basic control structures.

---

**Figure N.M:** A set of unstructured commands

\[
\text{command ::= identifier ::= expression} \\
    | \text{command; command} \\
    | \text{label : command} \\
    | \text{GOTO label} \\
    | \text{IF boo_exp THEN GOTO label}
\]

The unstructured commands contain the assignment command, sequential composition of commands, a provision to identify a command with a label, and unconditional and conditional GOTO commands. The unstructured commands have the advantage, they have direct hardware support and are completely general purpose. However, the programs are flat without hierarchical structure with the result that the code may be difficult to read and understand. The set of unstructured commands contains one of the most powerful commands, the GOTO. It is also the most criticized. The GOTO can make it difficult to understand a program by producing `spaghetti' like code. So named because the control seems to wander around in the code like strands of spaghetti.
The GOTO commands are explicit transfer of control from one point in a program to another program point. These *jump* commands come in unconditional and conditional forms:

\[
\text{goto label} \\
\text{if conditional expression goto label}
\]

At the machine level alternation and iteration may be implemented using *labels* and *goto* commands. Goto commands often take two forms:

1. **Unconditional goto.** The unconditional goto command has the form:

   \[
   \text{goto LABEL_i}
   \]

   The sequence of instructions next executed begin with the command labeled with \(\text{LABEL_i}\).

2. **Conditional goto.** The conditional goto command has the form:

   \[
   \text{if conditional expression then goto LABEL_i}
   \]

   If the conditional expression is true then execution transfers to the sequence of commands headed by the command labeled with \(\text{LABEL_i}\) otherwise it continues with the command following the conditional goto.

**Structured Programming**

The term *structured programming* was coined to describe a style of programming that emphasizes hierarchical program structures in which each command has one entry point and one exit point. The goal of structured programming is to provide control structures that make it easier to reason about imperative programs. Figure M.N gives a minimal set of structured commands.

---

**Figure N.M:** A set of structured commands

\[
\text{command}::=\text{SKIP} \\
| \text{identifier}:=\text{expression} \\
| \text{IF guarded_command} [ [ ] guarded_command ]^+ \text{FI} \\
| \text{DO guarded_command} [ [ ] guarded_command ]^+ \text{OD} \\
| \text{command} ; \text{command} \\
\text{guarded_command}::=\text{guard} --\rightarrow \text{command} \\
\text{guard}::=\text{boolean expression}
\]
The IF and DO commands which are defined in terms of guarded commands require some explanation. The IF command allows for a choice between alternatives while the DO command provides for iteration. In their simplest forms, an IF statement corresponds to an If condition then command and a DO statement corresponds to a While condition Do command.

```
IF guard --> command FI = if guard then command
DO guard --> command OD = while guard do command
```

A command preceded by a guard can only be executed if the guard is true. In the general case, the semantics of the IF - FI and DO - OD commands requires that only one command corresponding to a guard that is true be selected for execution. The selection is nondeterministic.

Control structures are syntactic structures that define the order in which assignments are performed. Imperative programming languages provide a rich assortment of sequence control mechanisms. Three control structures are found in traditional imperative languages: sequential composition, alternation, and iteration.

Aside. Imperative programming languages often call assignments and control structures commands, statements or instructions. In ordinary English, a statement is an expression of some fact or idea and thus is an inappropriate designation. Commands and instructions refer to an action to be performed by a computer. Lacking a more neutral term we will use command to refer to assignment, skip, and control structures.

Sequential Composition. Sequential composition specifies a linear ordering for the execution of commands. It is usually indicated by placing commands in textual sequence and either line separation or a special symbol (such as the semicolon) is used to indicate termination point of a command. In C the semicolon is used as a terminator, in Pascal it is a command separator. At a more abstract level, composition of commands is indicated by using a composition operator such as the semicolon (C₀;C₁).

Selection: Selection permits the specification of a sequence of commands by cases. The selection of a particular sequence is based on the value of an expression. The if and case commands are the most common representatives of alternation.

Iteration: Iteration specifies that a sequence of commands may be executed zero or more times. At run time the sequence is repeatedly composed with itself. There is an expression whose value at run time determines the number of compositions. The while, repeat and for commands are the most common representatives of iteration.

Abstraction: A sequence of commands may be named and the name used to invoke the sequence of
commands. Subprograms, procedures, and functions are the most common representatives of abstraction.

**Skips**

The simplest kind of command is the `skip` command. It has no effect.

**Composition**

The most common sequence is the **sequential** composition of two or (more) commands (often written `S_0;S_1`). Sequential composition is available in every imperative programming language.

**Alternation**

An **alternative command** may contain a number of alternative sequences of commands, from which exactly one is chosen to be executed. The nondeterministic IF-FI command is unusual. Traditional programming languages usually have one or more if commands and a case command.

```ada
-- Ada
if condition then
  commands
{ elsif condition then
  commands }
[ else
  commands ]
endif

case expression is
when choice | choice => commands
when choice | choice => commands
[when others => commands]
end case;
```

**Iteration**

An **iterative command** has a body which is to be executed repeatedly and has an expression which determines when the execution will cease. The three common forms are the while-do, the repeat-until, and the for-do.
The Imperative Programming Paradigm

The while-do command semantics require the testing of the condition before the body is executed. The semantics of the repeat-until command require the testing of the condition after the body is executed. The for-do command semantics require testing of the condition before the body is executed.

The iterative commands are often used to traverse the elements of a data structure - search for an item etc. This insight leads to the concept of generators and iterators.

**Definition:** A *generator* is an expression which generates a sequence of values contained in a data structure.

The generator concept appears in functional programming languages as *functionals*.

**Definition:** An *iterator* is a generalized looping structure whose iterations are determined by a generator.

An iterator is used with the an extended form of the for loop where the iterator replaces the initial and final values of the loop index. For example, given a binary search tree and a generator which performs inorder tree traversal, an iterator would iterate for each item in the tree following the inorder tree traversal.

FOR Item in Tree DO S;

**Sequential Expressions**

Imperative programming languages with their emphasis on the sequential evaluation of commands often fail to provide a similar sequentiality to the evaluation of expressions. The following code illustrates a common programming situation where there are two or more conditions which must remain true for iteration to occur.
The code implements a sequential search for a value in a table and terminates when either the entire table has been searched or the value is found. Assuming that the subscript range for list is 0 to length it seems reasonable that the termination of the loop should occur either when the index is out of bounds or when the value is found. That is, the arguments to the and should be evaluated sequentially and if the first argument is false the remaining argument need not be evaluated since the value of the expression cannot be true. Such an evaluation scheme is call short-circuit evaluation. In languages without short-circuit evaluation, if the value is not in the list, the program aborts with a subscript out of range error.

The Ada language provides the special operators and then and or else so that the programmer can specify short-circuit evaluation.

Subprograms, procedures, and functions

A procedure is an abstraction of a sequence of commands. The procedure call is a reference to the abstraction. The syntax of procedure definition and invocation (call) is simple.

Procedure definition: \texttt{name}( \texttt{parameter list}) \{ \texttt{body} \}

Procedure invocation: \texttt{name}( \texttt{argument list} )

The semantics of the procedure call is determined by the semantics of the procedure body. For many languages with non-recursive procedures, the semantics may be viewed as simple textual substitution.

Terminology: Parameters are often called formal parameters and arguments are often called actual parameters.

Parameters and arguments have a simple syntax

Parameter list: \texttt{t_0 name_1, ..., t_{n-1} name_{n-1}}

Argument list: \texttt{expression_1, ..., expression_{n-1}}

An in parameter designates that the body of the procedure may not modify the value of the argument (often implemented as a copy of the argument). An out parameter designates that value of the argument is undefined on entry to the procedure and when the procedure terminates, the argument is assigned to a
value (often copied to the argument). An **in-out** parameter designates that the value of the parameter may be defined on entry to the procedure and may be modified when the procedure terminates.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal</td>
<td></td>
</tr>
<tr>
<td><strong>in</strong></td>
<td><em>name : type</em></td>
</tr>
<tr>
<td><strong>in-out</strong></td>
<td><em>var name : type</em></td>
</tr>
<tr>
<td>Ada</td>
<td></td>
</tr>
<tr>
<td><strong>in</strong></td>
<td><em>name : in type</em></td>
</tr>
<tr>
<td><strong>out</strong></td>
<td><em>name : out type</em></td>
</tr>
<tr>
<td><strong>in-out</strong></td>
<td><em>name : in out type</em></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td><strong>in</strong></td>
<td><em>type name</em></td>
</tr>
<tr>
<td><strong>in-out</strong></td>
<td>*type <em>name</em></td>
</tr>
<tr>
<td></td>
<td>(internal reference to the in-out parameter must be <em>name</em>)</td>
</tr>
</tbody>
</table>

### Other Control Structures

Other control structures are possible. In simulations, it is often easier to structure a program as a set of cooperating tasks. When the task activities are interdependent, they can be structured as a collection of **coroutines**. Unlike subroutines where control is passed back to the calling routine from the subroutine when the subroutine is finished, control is passed back and forth between the subroutines with control resuming where it left off (right after the resume commands in the following).

```
Coroutine C1 ...
  \text{resume C2} ...

Coroutine C2 ...
  \text{resume C3} ...

Coroutine C3 ...
  \text{resume C1} ...
```

There is a single thread of control that moves from coroutine to coroutine. The multiple calls to a coroutine do not necessarily require multiple activation records.

In addition to coroutines there are concurrent or parallel processes

```
[|| Process_0, \ldots, Process_{n-1} ]
```

with multiple threads of control which communicate either through shared variables or message passing. Concurrency and parallel programming languages are considered in a later chapter.

### Reasoning about Imperative Programs

Imperative constructs jeopardize many of the fundamental techniques for reasoning about mathematical
objects. For example, the assignment axiom of axiomatic semantics is valid only for languages without aliasing and side effects. Much of the work on the theory of programming languages is an attempt to explain the "referentially opaque" features of programming languages in terms of well-defined mathematical constructs. By providing descriptions of programming language features in terms of standard mathematical concepts, programming language theory makes it possible to manipulate programs and reason about them using precise and rigorous techniques. Unfortunately, the resulting descriptions are complex and the notational machinery is difficult to use in all but small examples. It is this complexity that provides a strong motivation to provide functional and logic programming as alternatives to the imperative programming paradigm.

Sequencers

There are several common features of imperative programming languages that tend to make reasoning about the program difficult. The goto command \cite{Dijk68} breaks the sequential continuity of the program. When the use of the goto command is undisciplined, the breaks involve abrupt shifts of context.

In Ada, the exit sequencer terminates an enclosing loop. All enclosing loops up to and including the named loop are exited and execution follows with the command following the named loop.

Ada uses the return sequencer to terminate the execution of the body of a procedure or function and in the case of a function, to return the result of the computation.

Exception handlers are sequencers that take control when an exception is raised.

A sequencer is a construct that allows more general control flows to be programmed.

- Jumps
- Exits
- Exceptions -- propagation, raising, resumption, handler (implicit invocation)
- Coroutines

The machine language of a typical computer includes instructions which allow any instruction to be selected as the next instruction. A sequencer is a construct that is provided to give high-level programming languages some of this flexibility. We consider three sequencers, jumps, escapes, and exceptions. The most powerful sequencer (the goto) is also the most criticized. Sequencers can make it difficult to understand a program by producing `spaghetti' like code. So named because the control seems to wander around in the code like the strands of spaghetti.

Escape

An escape is a command which terminates the execution of a textually enclosing construct. An escape of
The form:

\[
\text{return expr}
\]

is used in C to exit a function call and return the value computed by the function.

An escape of the form:

\[
\text{exit(n)}
\]

is used to exit \( n \) enclosing constructs. The exit command can be used in conjunction with a general loop command to produce \text{while} and \text{repeat} as well as more general looping constructs.

In C a \text{break} command sends control out of the enclosing loop to the command following the loop while the \text{continue} command transfers control to the beginning of the enclosing loop.

**Exceptions**

There are many "exception" conditions that can arise in program execution. Some exception conditions are normal for example, the end of an input file marks the end of the input phase of a program. Other exception conditions are genuine errors for example, division by zero. Exception handlers of various forms can be found in PL/1, ML, CLU, Ada, Scheme and other languages.

There are two basic types of exceptions which arise during program execution. They are domain failure, and range failure.

**Domain failure**

occurs when the input parameters of an operation do not satisfy the requirements of the operation. For example, end of file on a read instruction, division by zero.

**Range failure**

occurs when an operation is unable to produce a result for values which are in the range. For example, division by numbers within an epsilon of zero.

**Definition:** An exception condition is a condition that prevents the completion of an operation. The recognition of the exception is called raising the exception.

Once an exception is raised it must be handled. Handling exceptions is important for the construction of robust programs. A program is said to be robust if it recovers from exceptional conditions.

**Definition:** The action to resolve the exception is called handling the exception. The propagation of an exception is the passing of the exception to the context where it can be handled.
The simplest method of handling exceptions is to ignore it and continue execution with the next instruction. This prevents programmer from learning about the exception and may lead to erroneous results.

The most common method of handling exceptions is to abort execution. This is not exceptable for file I/O but may be acceptable for an array index being out of bounds or for division by zero.

The next level of error handling is to return a value outside the range of the operation. This could be a global variable, a result parameter or a function result. This approach requires explicit checking by the programmer for the error values. For example, the `eof` boolean is set to true when the program has read the last item in a file. The `eof` condition can then be checked before attempting to read from a file. The disadvantage of this approach is that a program tends to get cluttered with code to test the results. A more serious consequence is that a programmer may forget to include a test with the result that the exception is ignored.

**Responses to an Exception**

Return a label and execute a goto -- Fortran

**Issues**

**Resumption of Program Execution**

Once an exception has been detected, control is passed to the handler that defines the action to be taken when the exception is raised. The question remains, what happens after handling the exception?

One approach is to treat exception handlers as subroutines to which control is passed and after the execution of the handler control returns to the point following the call to the handler. This is the approach taken in PL/1. It implies that the handler ``fixed" the state that raised the condition.

Another approach is that the exception handler's function is to provide a clean-up operation prior to termination. This is the approach taken in Ada. The unit in which the exception occurred terminates and control passes to the calling unit. Exceptions are propagated until an exception handler is found.

**Suppression of the Exception**

Some exceptions are inefficient to implement (for example, run time range checks on array bounds). The such exceptions are usually implemented in software and may require considerable implementation overhead. Some languages give the programmer control over whether such checks and the raising of the corresponding exception will be performed. This permits the checks to be turned on during program development and testing and then turned off for normal execution.
1. Handler Specification
2. Default Handlers

Propagation of Exception

Side effects

Side effects are a feature of imperative programming languages that make reasoning about the program difficult. Side effects are used to provide communication among program units. When undisciplined access to global variables is permitted, the program becomes difficult to understand. The entire program must be scanned to determine which program units access and modify the global variables since the call command does not reveal what variables may be affected by the call.

At the root of differences between mathematical notations and imperative programs is the notion of referential transparency (substitutivity of equals for equals). Manipulation of formulas in algebra, arithmetic, and logic rely on the principle of referential transparency. Imperative programming languages violate the principle. For example:

```plaintext
integer f(x:integer)
{
    y := y+1;
    f := y + x
}
```

This ```function`` in addition to computing a value also changes the value of the global variable $y$. This change to a global variable is called a side effect. In addition to modifying a global variable, it is difficult to reason with the function itself. For example, if at some point in the program it is known that $y = z = 0$ then $f(z) = 1$ in the sense that after the call $f(z)$ will return 1. But, should the following expression occur at that point in the program, it will be false.

$$1 + f(z) = f(z) + f(z)$$

I/O functions of necessity involve side effects. The following expressions involving the C function `getint` may return different values even though algebraically they appear to have the same value.

```plaintext
2 * getint()
gGetInt() + getInt()
```

The first multiplies the next integer read from the input file by two while the second expression denotes the sum of the next two successive integers read from the input file.
Aliasing and dangling references

Two names are *aliases* if they denote (share) the same data object during a unit activation. Aliasing is another feature of imperative programming languages that makes programs harder to understand and harder to reason about.

One way aliases occur is when two or more arguments to a subprogram are the same. When a data object is passed by "reference" it is referenced both by its name in the calling environment and its parameter name in the called environment. In the following subprogram, the parameters are *in-out* parameters.

```plaintext
aliasingExample (m, n : in out integer);
{
    n := 1;
    n := m + n
}
```

The two parameters are used as different names for the same object in the call `aliasingExample (i, i)`. In this example, the result is that `i` is set to 2. In the call `aliasingExample (a[i], a[j])` the result depends on the values of `i` and `j` with aliasing occurring when they are equal. This second call illustrates that aliasing can occur at run time so the detection of aliasing may be delayed until run time and so compilers cannot be relied on to detect aliasing.

Aliasing interferes with optimizing phase of a compiler. Optimization sometimes requires the reordering of steps or the deletion of unnecessary steps. The following assignments which appear to be independent of each other illustrate an order dependency.

```plaintext
x := a + b
y := c + d
```

If `x` and `c` are aliases for the same object, the assignments are interdependent and the order of evaluation is important.

The purpose of the equivalence command in FORTRAN is the creation of aliases. It permits the efficient use of memory (historically a scarce commodity) and can be used as a crude form of a variant record. Another way in which aliasing can occur is when a data object may be a component of several data objects (referenced through pointer linkages).

- Formal and actual parameters share the same data object.
- Procedure calls have overlapping actual parameters.
● A formal parameter and a global variable denote the same data object.

Pointers are intrinsically generators of aliasing.

```pascal
var p, q : ^T;
...
new(p);
q := p
```

When a programming language requires programmers to manage memory for dynamically allocated objects and the language permits aliasing, an object returned to memory may still be accessible though an alias and the value may be changed if the memory manager allocates the same storage area to another object. In the following code,

```pascal
type pointer = ^Integer
var p : Pointer;

procedure Dangling;
var q : Pointer;
begin;
  new(q);  q^ := 23;  p := q;  dispose(q)
end;

begin
  new(p);  Dangling(p)
end;
```

the pointer p is left pointing to a non-existent value.

The problem of aliasing arises as soon as a language supports variables and assignment. If more than one assignment is permitted on the same variable x, the fact that x=a cannot be used at any other point in the program to infer a property of x from a property of a. Aliasing and global variables only magnify the issue.

**Historical Perspectives and Further Reading**

Imperative languages have a rich and varied history. The first imperative programming languages were machine languages. Machine instructions were soon replaced with assembly languages which are essentially transliterations of machine code.

**Early Imperative Languages**
FORTRAN (FORmula TRANslation) was the first high level language to gain wide acceptance. It was designed for scientific applications and featured an algebraic notation, types, subprograms, and formatted input/output. It was implemented in 1956 by John Backus at IBM specifically for the IBM 704 machine. Efficient execution was a major concern consequently, its structure and commands have much in common with assembly languages. FORTRAN won wide acceptance and continues to be in wide use in the scientific computing community.

COBOL (COmmon Business Oriented Language) was designed (by a committee of representatives of computer manufactures and the Department of Defense) at the initiative of the U. S. Department of Defense in 1959 and implemented in 1960 to meet the need for business data processing applications. COBOL featured records, files and fixed decimal data. It also provided a "natural language" like syntax so that programs would be able to be read and understood by non-programmers. COBOL won wide acceptance in the business data processing community and continues to be in wide use.

ALGOL 60 (ALGorithmic Oriented Language) was designed in 1960 by an international committee for use in scientific problem solving. Unlike FORTRAN it was designed independently of an implementation, a choice which lead to an elegant language. The description of ALGOL 60 introduced the BNF notation for the definition of syntax and is a model of clarity and completeness. Although ALGOL 60 failed to win wide acceptance, it introduced block structure, structured control statements and recursive procedures into the imperative programming paradigm.

**Evolutionary Developments**

PL/I (Programming Language I) was developed at IBM in the mid 1960s. It was designed as a general purpose language to replace the specific purpose languages like FORTRAN, ALGOL 60, COBOL, LISP, and APL (APL and LISP were considered in the chapter on functional programming. PL/I incorporated block structure, structured control statements, and recursion from ALGOL 60, subprograms and formatted input/output from FORTRAN, file manipulation and the record structure from COBOL, dynamic storage allocation and linked structures from LISP, and some array operations from APL. PL/I introduced exception handling and multitasking for concurrent programming. PL/I was complex, difficult to learn, and difficult to implement. For these and other reasons PL/I failed to win wide acceptance.

ALGOL 68 was designed to be a general purpose language which remedied PL/I's defects by using a small number of constructs and rules for combining any of the constructs with predictable results--orthogonality. The description of ALGOL 68 issued in 1969 was difficult to understand since it introduced a notation and terminology that was foreign to the computing community. ALGOL 68 introduced orthogonality and data extensibility as a way to produce a compact but powerful language. The ALGOL 68 Report" considered to be one of the most unreadable documents ever printed and implementation difficulties prevented ALGOL 68's acceptance.

Pascal was developed by Nicklaus Wirth partly as a reaction to the problems encountered with ALGOL
68 and as an attempt to provide a small and efficient implementation of a language suitable for teaching good programming style. C, which was developed about the same time, was an attempt to provide an efficient language for systems programming.

Modula-2 and Ada extended Pascal with features to support module based program development and abstract data types. Ada was developed as the result of a Department of Defense initiative while Modula-2 was developed by Nicklaus Wirth. Like PL/1 and Algol-68, Ada represents an attempt to produce a complete language representing the full range of programming tasks.

Simula 67 added coroutines and classes to ALGOL 60 to provide a language more suited to solving simulation problems. The concept of classes in object-oriented programming can be traced back to Simula's classes. Small-talk combined classes, inheritance, and ease of use to provide an integrated object-oriented development environment. C++ is an object-oriented programming language derived from C. Java, a simplified C++, is an object-oriented languages designed to dynamically load modules at runtime and to reduce programming errors.

### Expression-oriented languages

Expression-oriented languages achieve simplicity and regularity by eliminating the distinction between expressions and commands. This permits a simplification in the syntax in that the language does not need both procedure and function abstractions nor conditional expressions and commands since the evaluation of an expression may both yield a value and have a side effect of updating variables. Since the assignment expression \( V := E \) can be defined to both yield the value of the expression \( E \) and assign \( V \) to the value of \( E \), expressions of the form \( V_0 := \ldots := V_n := E \) are possible giving multiple assignment for free. Algol-68, C, Scheme, and ML are examples of expression oriented languages.

### Exercises

1. Give all possible forms of assignment found in the programming language C.
2. Give axiomatic, denotational and operational semantics for the simultaneous assignment command.
3. Discuss the relationship between the assignment command and input and output commands.
4. Give axiomatic, denotational and operational semantics for the \texttt{goto} command.
5. Find an algorithm which transforms a program containing \texttt{gos} into an equivalent program without \texttt{gos}.
6. Give axiomatic, denotational and operational semantics for the \texttt{skip} command.
7. What is used to indicate sequential composition in
   a. the Pascal family of languages?
   b. the C family of languages?
8. Show how to implement the if-then-else command using unstructured commands.
9. Show how to implement a structured while-do and if-then-else commands using unstructured
commands.
10. Show that case and if commands are equivalent by implementing a case command using if commands and an if command using a case command.
11. Compare and contrast the if and case/switch commands of Ada and C.
12. Compare and contrast the looping commands of Ada and C.
13. Show how to implement the repeat-until and for-do commands in terms of a while-do command.
14. Show that while and repeat until control structures are equivalent.
15. Design a generalized looping command and give its axiomatic semantics.
16. Give axiomatic semantics for the IF-FI and DO-OD commands.
17. Give axiomatic semantics for the C/C++/Java for command.
18. Provide recursive definitions of the iterative control structures while, repeat, and for.
19. Alternative control structures
20. What is the effect on the semantic descriptions if expressions are permitted to have side effects?
21. Axiomatic semantics
22. Denotational semantics
23. Operational semantics
24. Classify the following common error/exception conditions as either domain or range errors.
   a. overflow -- value exceeds representational capabilities
   b. undefined value -- variable value is undefined
   c. subscript error -- array subscript out of range
   d. end of input -- attempt to read past end of input file
   e. data error -- data of the wrong type

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of Anthony A. Aaby. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee.
© 1998 Anthony A. Aaby. Last Modified - . Send comments to aabyan@wwc.edu
The Concurrent Programming

The root of all successful human organization is co-operation not competition. Concurrent programming is characterized by programming with more than one process.

Keywords and phrases Pipelines, parallel processes, message passing, monitors, concurrent programming, safety, liveness, deadlock, live-lock, fairness, communication, synchronization producer-consumer, dining philosophers.

There are several reasons for a programmer to be interested in concurrency:

1. To better understand computer architecture (it has a great deal of concurrency with pipelining (multiple steps) and super-scalar (multiple instructions)) and
2. compiler design,
3. some problems are most naturally solved by using a set of co-operating processes,
4. A sequential solution constitutes over specification, and
5. to reduce the execution time.

At the machine level, operations are sequential, if they occur one after the other, ordered in time. Operations are concurrent, if they overlap in time. In Figure 1, sequential operations are connected by a single thread of control while concurrent operations have multiple threads of control.

Figure 1: Sequential and Concurrent Operations

Sequential operations: \[ --\circ--\circ--\circ--\circ \] O: operation

Concurrent operations: \[ --|\ldots|--\rightarrow \] :- thread

Operations in the source text of a program are concurrent if they could be, but need not be, executed in
parallel. Thus concurrency occurs in a programming language when two or more operations could be but need not be executed in parallel. In Figure 2a the second assignment depends on the outcome of the first assignment while in Figure 2b neither assignment depends on the other and may be executed concurrently.


Concurrent programming involves the notations for expressing potential parallelism so that operations may be executed in parallel and the techniques for solving the resulting synchronization and communication problems. Notations for explicit concurrency are a program structuring technique while parallelism is mode of execution provided by the underlying hardware. Thus we can have parallel execution without explicit concurrency in the language. We can have concurrency in a language without parallel execution. This is the case when a program (with or without explicit concurrent sections) is executed on a single processor. In this case, the program is executed by interleaving executions of the concurrent operations in the source text.

**Aside.** The terms, concurrent, distributed and parallel have been used at various times to describe various types of concurrent programming. Multiple processors and disjoint or shared store are implementation concepts and are not important from the programming language point of view. What matters is the notation used to indicate concurrent execution, communication and synchronization.

Functional and logic programming languages do not necessarily need explicit specification of concurrency and, with a parallelizing compiler, may be executed on parallel hardware. It is important to note that the notion of processes is orthogonal to that of inference, functions and assignments.

The two fundamental concepts in concurrent programming are processes and resources. A process corresponds to a sequential computation with its own thread of control. Concurrent programs are distinguished from sequential programs in that, unlike sequential programs, concurrent programs permit multiple processes. Processes may share resources. Shared resources include program resources -- data
structures and hardware resources -- CPU, memory, & I/O devices.

Aside. Processes which share an address space are called threads or light-weight processes. For some programming languages (C, C++) there are threads packages to permit concurrent programming. In other cases, the operating system (Microsoft Windows NT, Sun Solaris) provides system calls for threads. Processes which do not share an address space are called heavy-weight processes. The Unix family of operating systems provide a system call to allow programmers to create heavy-weight processes.

The Concurrent Nature of Systems

Co-operation

The Bakery. busy waiting, fairness, liveness

The Secretary. scheduling, priority, co-operative multitasking, interrupts, competitive multitasking, pre-emptive multitasking

The Secretarial Pool parallel tasking

Geoffrey Fox's wall. the construction of a brick wall by a number of workers.

Questions:

● How do we break down the task to extract maximum parallelism?
● Wow do we get the task done in the shortest possible time with a given number of workers.
● What is the minimum amount of supervision needed?
● Can all workers be kept equally busy?
● Does the task demand specialized workers?
● Can we maintain efficiency as either the size of the problem or the number of workers grows?

The Nature of Concurrent Systems

Abstraction

Performance

Communication

In the previous solution, it was assumed that the processes shared the address space and that
synchronization was achieved by the use of monitor and condition queues. If the address spaces are disjoint, then both communication and synchronization must be achieved through message passing. There are two choices, message passing can be synchronous or asynchronous. When message passing is asynchronous, synchronization can be obtained by requiring a reply to a synchronizing message. In the examples that follow, synchronized message passing is assumed.

**Behavior**

**Synchronization and Communication**

Two processes are said to communicate if an action of one process must entirely precede an action of a second process. Synchronization is related to communication.

Live-lock may result if there are more than one waiting process and when the signal is received access is not granted fairly.

Starvation: (live-lock) multiple processes waiting for access but access is not provided in a fair manner

Coroutines.

Real-time Programming language issues

When message passing is asynchronous, synchronization can be obtained by requiring a reply to a synchronizing message. In the examples that follow, synchronized message passing is assumed.

Communication commands in the guards. Most communication based programming languages permit input commands in the guards but not output commands. The asymmetry is due to the resulting complexity required to implement output commands in the guards.

```pascal
process Q;
const qsize = 10;
var head, tail : integer;
    queue : array[0..qsize-1] of integer;
begin
    head, tail := 0, 0;
    *[ head != tail, C?X --> C!queue[head]; head := (head + 1) mod qsize 
       [] head != (tail + 1) mod qsize, P?X --> queue[tail],
       tail := X, (tail + 1) mod qsize]
end;
```
process P;
begin
    *[ true --> produce(X); Q!X]
end;

process C;
begin
    *[ true --> Q!X, Q?X; consume(X)]
end;

begin
    [ P || C || Q ]
end.

**Nondeterminism**

A program is *deterministic* if its evaluations on the same input it always produce the same output. The evaluation strategy might not always be unique.

A program is *nondeterministic* if it has more than one allowable evaluation strategy and different evaluation strategies lead to different results.

A concept related to nondeterminism is *parallel evaluation*. Parallel evaluation that does not involve interaction on the part of its subparts is called *noninterfering parallelism*. Processes which have disjoint address spaces cannot interfere with each other and thus can operate without fear of corrupting each other. For example, the two processes in

\[
|| i:=1, j:=2
\]

do not share an address space therefore, the assignments may take place in parallel.

Another example of non-interfering processes is found in matrix multiplication. When two matrices are multiplied, each entry in the product matrix is the result of multiplying a row times a column and summing the products. This is called an inner product. Each inner produce can be computed independently of the others. Figure~\ref{cp:mm}
is an example of a matrix multiplication routine written in the SR programming language. This particular example also illustrated dynamic process creation in that \(n^2\) processes are created to perform the multiplication.

In interfering parallelism, there is interaction and the relative speeds of the subparts can affect the final result.

Processes that access a common address space may interfere with each other. In this program,

\[[i:=1 \parallel i:=2]\]

the resulting value of \(i\) could be either 1 or 2 depending on which process executed last and in this program,

\[[i:=0;i:=i+1 \parallel i:=2]\]

the resulting value of \(i\) could be either 1, 2 or 3.

A language is concurrent if it uses interfering parallelism.

Sequential programs are nearly always deterministic. A deterministic program follows a sequence of step that can be predicted in advance. Its behavior is reproducible and thus, deterministic programs are testable. Concurrent programs are likely to be nondeterministic because the order and speed of execution of the processes is unpredictable. This makes testing of concurrent programs a difficult task.

The requirement for disjoint address space may be too severe a requirement. What is required is that shared resources may need to be protected so that only one process is permitted access to the resource at a time. This permits processes to cooperate, sharing the resource but maintaining the integrity of the resource.

**Mutual Exclusion**
Often a process must have exclusive access to a resource. For example, when a process is updating a data structure, no other process should have access to the same data structure otherwise the accuracy of the data may be in doubt. The necessity to restrict access is termed *mutual exclusion* and involves the following:

- At most one process has access
- If there are multiple requests for a resource, it must be granted to one of the processes in finite time.
- When a process has exclusive access to a shared resource it release it in finite time.
- When a process requests a resource it must obtain the resource in finite time.
- A process should not consume processing time while waiting for a resource.

There are several solutions to the mutual exclusion problem. Among the solutions are semaphores, critical regions and monitors.

**Deadlock**

*Deadlock* is a liveness problem; it is a situation in which a set of processes are prevented from making any further progress by their mutually incompatible demands for additional resources. For example, in the dining philosophers problem, deadlock occurs if each philosopher picks up his/her left fork. No philosopher can make further progress.

Deadlock can occur in a system of processes and resources if, and only if, the following conditions all hold together.

- *Mutual exclusion*: processes have exclusive access to the resources.
- *Wait and hold*: processes continue to hold a resource while waiting for a new resource request to be granted.
- *No preemption*: resources cannot be removed from a process.
- *Circular wait*: there is a cycle of processes, each is awaiting a resource held by the next process in the cycle.

There are several approaches to the problem of deadlock. A common approach is to *ignore* deadlock and hope that it will not happen. If deadlock occurs, (much as when a program enters an infinite loop) the system's operators abort the program. This is not an adequate solution in highly concurrent systems where reliability is required.

A second approach is to allow deadlocks to occur but *detect* and *recover* automatically. Once deadlock is detected, processes are selectively aborted or one or more processes are *rolled back* to an earlier state and temporarily suspended until the danger point is passed. This might not an acceptable solution in real-time systems.
A third approach is to prevent deadlock by weakening one or more of the conditions. The wait-and-hold condition may be modified to require a process to request all needed resources at one time. The circular-wait condition may be modified by imposing a total ordering on resources and insisting that they be requested in that order.

Another example of a liveness problem is live-lock (or lockout or starvation). Live-lock occurs when a process is prevented from making progress (other processes are running). This is an issue of fairness.

**Scheduling**

When there are active requests for a resource there must be a mechanism for granting the requests. Often a solution is to grant access on a first-come, first-served basis. This may not always be desirable since there may be processes whose progress is more important. Such processes may be given a higher priority and their requests are processed first. When processes are prioritized, some processes may be prevented from making progress (such a process is live-locked). A fair scheduler insures that all processes eventually make progress thus preventing live-lock.

**Semantics**

Parallel processes must be... 
\begin{enumerate}
  \item Synchronization-coordination of tasks which are not completely independent.
  \item Communication-exchange of information
  \item Scheduling-priority,
  \item Nondeterminism-arbitrary selection of execution path
\end{enumerate} 

Explicit Parallelism (message passing, semaphores, monitors) Languages which have been designed for concurrent execution include Concurrent Pascal, Ada and Occam. Application areas are typically operating systems and distributed processing. Ensemble activity

**Concurrency in Programming Languages**

Threads/Communication/Metaphor

From the programmer's point of view, concurrent programming notations allow programs to be structured as a set of possibly interactive processes. Such an organization is particularly useful for operating systems, real-time control systems, simulation studies, and combinatorial search applications.

To permit the effective use of multiple processes, concurrent programming languages must provide notations for:

1. **Concurrent execution**: A notation that denotes operations that could be, but need not be, executed in parallel.
2. **Communication**: A notation that permits processes to exchange information either through shared variables (visible to each process) or a message passing mechanism.

**Shared Memory**

Assignment: \( X := E \)

**Message Passing**

*Synchronous* \( P_i!E, P_j?X \)

*Asynchronous* \( P_i!E, P_j?X \)

Remote procedure call

3. **Synchronization**: A notation to require a process to *wait* for a *signal* from another process. In general processes are not independent. Often a process depends on data produced by another process. If the data is not available the process must wait until the data is available.

\[ \text{wait}(P_i), \text{signal}(P_j) \]

A process can change its state to Blocked (waiting for some condition to change) and can signal Blocked processes so that they can continue.

In this case, the OS must provide the system calls BLOCK and WAKEUP. cking version of a semaphore

\[
\text{type semaphore} = \text{record}
\begin{align*}
\text{value} : \text{integer}; \\
\text{L} : \text{list of processes}; & // \text{or queue blocked waiting for} \\
\text{end}; & // \text{the signal}
\end{align*}
\]

\[
\text{down}(S): S.\text{value} := S.\text{value} - 1; \ // \text{wait} \\
\text{if} S.\text{value} < 0 \text{ then} \\
\text{add this process to } S.\text{L}; \\
\text{block}; \\
\text{end}; \\
\]

\[
\text{up}(S): S.\text{value} := S.\text{value} + 1; \ // \text{signal}
\]
if S.value <= 0 then
  remove a process P from S.L;
  wakeup(P);
end;

Implementation

- **Single processor:** The normal way is to implement the semaphore operations (up and down) as system calls with the OS disabling the interrupts while executing the code.
- **Multiprocessor:** Each semaphore should be protected by a lock variable, with the TSL instruction used to be sure that only one CPU at a time examines the semaphore. Using the TSL instruction to prevent several CPUs from accessing the semaphore at the same time is different from busy waiting.

In many applications it is necessary to order the actions of a set of processes as well as interleave their access to shared resources: common address space, critical section protected by a monitor, synchronization provided through wait and signal.

Some alternative synchronization primitives are

- Semaphores
- Critical Regions
- Monitors
- Synchronized Message Passing

4. **Mutual exclusion:** A notation to synchronize access to shared resources.

**Semaphores**

Monitors: One approach is to protect the critical section by a monitor. The monitor approach requires that only one process at a time may execute in the monitor.

```plaintext
monitor Queue_AD
const qsize = 10;
var head, tail : integer;
  queue : array[0..qsize-1] of integer;
  notempty, notfull : condition;
procedure enqueue (x : integer);
  begin
    [ head=(tail+1) mod qsize --> wait(notfull)
    [] head!=(tail+1) mod qsize --> skip];
    queue[tail], tail := x, (tail + 1) mod qsize
    signal(notempty)
  end;
procedure dequeue (var x : integer);
  begin
    [ head=tail --> wait(notempty)
    [] head!=tail --> skip];
```
The Concurrent Programming

x, head := queue[head], (head + 1) mod qsize;
signal(notfull)
end;
begin
  head, tail := 0, 0;
end;
begin
  [ produce(x); enqueue(x) || dequeue(y); consume(y) ||
    dequeue(y); consume(y) ]
end.

- Correctness (safety and liveness)
- Performance
- Architecture
- Implementation

Aside.

concurrency: Fork (P) & Join (P)

combined notation for communication and synchronization C, Scheme, Ada, PVM, PCN, SR, Java and Occam are just some of the programming languages that provide for processes.

Producer-Consumer

In the following program there is a producer and a consumer process. The producer process adds items to the queue and the consumer process removes items from the queue. The safety condition that must be satisfied is that the head and tail of the queue must not over run each other. The liveness condition that must be satisfied is that when the queue contains an item, the consumer process must be able to access the queue and when the queue contains space for another item, the producer process must be able to access the queue.

const qsize = 10;
var count: integer;
queue : array[0..qsize-1] of integer;
procedure enqueue (x : integer);
begin
  *[ head=(tail+1) mod qsize --> skip];
  queue[tail], tail := x, (tail + 1) mod qsize
end;
procedure dequeue (var x : integer);
begin
  *[ head=tail --> skip];
\[
x, \text{head} := \text{queue[head]}, (\text{head} + 1) \mod qsize \\
\text{end;}
\]

\[
\begin{align*}
\text{begin} \\
\text{head}, \text{tail} := 0, 0; \\
[ \ast \text{produce}(x); \text{enqueue}(x) ] \mid [ \ast \text{dequeue}(y); \text{consume}(y) ]
\end{align*}
\]

end.

Since the processes access different portions of the queue and test for the presence or absence of items in the queue before accessing the queue, the desired safety properties are satisfied. Note however, that busy waiting is involved.

**Shared Memory Model**

**Process Creation**

- Static
- Dynamic

**Process Identification**

- Named
- Anonymous

**Synchronization**

- Semaphore
- Monitor

In many applications it is necessary to order the actions of a set of processes as well as interleave their access to shared resources. common address space, critical section protected by a monitor, synchronization provided through wait and signal.

Some alternative synchronization primitives are

- Semaphores
- Critical Regions
- Monitors
- Synchronized Message Passing

If in the previous example another process where to be added, either a producer or a consumer process, an unsafe condition could result. Two processes could compete for access to the same item in the queue. The solution is to permit only one process at a time to access the enqueue or dequeue routines. One
approach is to protect the critical section by a monitor. The monitor approach requires that only one process at a time may execute in the monitor. The following monitor solution is incorrect.

```
monitor Queue_ADT
  const qsize = 10;
  var count:integer;
  queue : array[0..qsize-1] of integer;
procedure enqueue (x : integer);
  begin
    *[ head=(tail+1) mod qsize -> skip];
    queue[tail],tail := x, (tail + 1) mod qsize
  end;
procedure dequeue (var x : integer);
  begin
    *[ head=tail -> skip];
    x,head := queue[head],(head + 1) mod qsize
  end;
begin
  head,tail := 0,0;
end;
begin
  [ produce(x); enqueue(x) $\parallel$ dequeue(y); consume(y) $\parallel$ dequeue(y); consume(y) ]
end.
```

Note that busy waiting is still involved and further once a process is in the monitor and is waiting, no other process can get in and the program is {\it deadlocked}.

**Message Passing Model**

**Process Creation**

- Static
- Dynamic

**Process Identification**

- Named
- Anonymous

**Message Passing**
The Concurrent Programming

- Synchronous
- Asynchronous

Data Flow

- Unidirectional
- Bidirectional

**MPI**

**Hardware**

**Processes**

- Process: single flow of control through a set of instructions
- Processor: hardware device for executing
- Parallel computer: two or more processors connected through an interconnection network.

**Flynn's Taxonomy**

System classification by number of instruction and data streams.

- SISD: classical sequential von Neumann machine. Inherently sequential. Parallelism may be simulated by interleaving instructions & multiprogramming.
- Pipelining and vector architectures
- SIMD: synchronous since there is a single instruction stream, each processor has its own data stream. Matrix operations are a good example. Thinking Machines - CM, Maspar Computer Corp -- MP (single sequencing units)
- MISD: does not seem to be useful
- MIMD/SPMD: asynchronous processes but with occasional pauses to synchronize; Intel iPSC, nCUBE, Sequent Symmetry, SGI Onyx, SUN MP system
  - shared-memory (sometimes called multiprocessors) locking and protection mechanism
  - distributed-memory (sometimes called multicomputers) message passing

**Shared-Memory MIMD**

- Bus-based architectures
- Cache coherence -- for bus based systems use the snoopy protocol
- Switch-based architectures, crossbar switch
- NUMA - nonuniform memory access
Distributed-Memory MIMD

- Dynamic interconnection networks
- Static interconnection networks
  - linear array, 2D mesh, 3D mesh
  - ring, torus
  - hypercube
  - bus

Communication and routing

- routing
- store-and-forward
  - cut-through

The Engineering of Concurrent Programs

A parallel programming environment must support the following three phases of system behavior specification.

- **Programming** Behavior of processes and their interconnection
- **Network description** Processors and their interconnection
- **Configuration** Mapping of software onto hardware

Programming

The way to design parallel software is to begin with the most parallel algorithm possible and then gradually render it more sequential ... until it suits the machine on which it is to run.

*East (1995)*

*Chandy and Taylor (1992)* define an elegant parallel programming language PCN (Program Composition Notation) based on:

- Shared definition variables (single assignment) -- \( X = \text{Exp} \),
- Parallel composition -- \( \[\![P_0, \ldots, P_n]\] \),
- Choice composition -- \( [? \text{G}_0 \rightarrow P_0, \ldots, \text{G}_n \rightarrow P_n] \),
- Sequential composition -- \( [; S_0, \ldots, S_n] \), and
- Recursion -- \( \text{name(parameters)} \text{ composition expression} \)
The definition variable eliminates the indeterminacy problems. Communication is through shared variables which may be streams. Synchronization is achieved by requiring a process that references an undefined variable to wait until it is defined by some other process before continuing. Recursion with parallel composition permits dynamic process creation.

If a program that uses only parallel and choice composition and definition variables does not have adequate efficiency, ...

We use the following steps in the introduction of mutables and sequencing into a parallel block.

1. We order the statements in a parallel block so that all variables that appear on the right-hand sides of definition statements reduce to ground values or tuples, and all guards reduce to the ground values true or false, give only the definitions established by statements earlier in the ordering. In other words, we order statements in the direction of data flow; statements that write a variable appear earlier than statements that read that variable. Then we convert the parallel block into a sequential block by replacing "||" by ";" retaining the data-flow order of statements.

2. Next, we introduce mutables, add assignment statements to our program, and show that the mutable \( m \) has the same value as the definition variable \( x \) it is to replace, at every point in the program in which \( x \) is read - i.e., where \( x \) appears on the right-hand side of a definition statement or assignment or guard.

3. Finally, we remove the definition variables that are replaced by mutables, secure in the knowledge that the mutables have the same value as the definition variables in the statements in which they are read. We must, of course, be sure that mutables shared by constituent blocks of a parallel block are not modified within the parallel block.

Chandy and Taylor (1992)

Decomposition

Function decomposition

Break down the task so that each worker performs a distinct function.

Advantages
Disadvantages

- Fewer tasks than workers
- Some tasks are easier than others
**Domain decomposition**

Divide the domain by the number of workers available.

- Horizontal domain decomposition: group is responsible for the entire project.
- Vertical domain decomposition: Assembly line, pipelining

**Communication and Synchronization**

Co-operation requires communication. Communication requires a protocol.

**Alternation and Competition**

Allocate time to multiple tasks.

- Scheduling
- Co-operative multitasking: multi-person game, using the copy machine
- Priority: telephone vs email
- Competitive multitasking: time slice
- Client-server: bakery
- Busy waiting
- Fairness

**Correctness**

Partial correctness, Total correctness, satisfaction of specifications...

*Chandy & Taylor (1992)* require

1. Shared mutable variables remain constant during parallel composition.
2. Mutable variables to copied when used in definitions.
3. When defined, definition variables act as constants in assignment.

*Lewis (1993)* develops a theory of program correctness called *flow-correctness*. Lewis requires for each shared variable:

1. it must be defined before it is referenced,
2. it must be referenced before it is updated, and
3. only one process at a time may (re)define it.

These rules apply only to the dependencies among variables and do not include either total correctness
Correctness issues in the design of concurrent programs fall in one of two categories: safety and liveness.

- **Safety**: nothing bad will happen. For example, access to a shared resource like a printer requires that the user process have exclusive access to the resource. So there must be a mechanism to provide *mutual exclusion*.
- **Liveness**: something good will happen. On the other hand, no process should prevent other processes from eventual access to the printer. Thus any process which wants the printer must eventually have access to the printer.

Safety is related to the concept of a loop invariant. A program should produce the "right" answer. Liveness is related to the concept of a loop variant. A program is expected to make progress. Termination is an example of a liveness property when a program is expected to terminate.

### Network description

### Configuration

### Implementation
Concurrency occurs in hardware when two or more operations overlap in time. Concurrency in hardware dates back to the 1950s when special-purpose processors were developed for controlling input/output devices. This permitted the overlapping of CPU instructions with I/O actions. For example, the execution of an I/O instruction no longer delayed the execution of the next instruction. The programmer was insulated from this concurrency by the operating system. The problems presented to the operating systems by this concurrency and the resulting solutions form the basis for constructs supporting concurrency in programming languages. Hardware signals called interrupts provided the synchronization between the CPU and the I/O devices.

Other advances in hardware have lead to the development of alternative architectures. Pipeline processors which fetch the next instruction while the first instruction is being decoded. Super scalar processors combine multiple pipelines to provide an even greater level of concurrency. Array processors provide a large number of identical processors that operate simultaneously on different parts of the same data structure. Data flow computers aim at extracting maximum concurrency from a computation by performing as much of the computation in parallel as possible. Connectionism based hardware models provide concurrency by modeling computation after the neural networks found in the brain.

Interrupts together with a hardware clock made it possible to implement multiprogramming systems which are designed to maximize the utilization of the the computer systems resources (CPU, store, I/O devices) by running two or more jobs concurrently. When one job was performing I/O another job could be executing using the CPU.
Interrupts and the hardware clock also made possible the development of interactive systems where multiple users have simultaneous access to the system resources. Such a system must provide for a large number of jobs whose combined demands on the system may exceed the system resources. Various techniques of swapping and paging meet this need by moving jobs in and out of the store to the larger capacity of backing store devices. With the increase of jobs comes the need to increase the capacity of the CPU. The solution was to develop multiprocessor systems in which several CPUs are available and simultaneously operate on separate jobs in the shared store.

An alternate solution is to develop distributed systems consisting of several complete computers (each containing both CPU and an associated store) that can operate independently but also communicate efficiently. Such systems of local area networks permit the efficient use of shared resources (such as printers and large backing store via file servers) and increase the computational throughput of the system.

*The SR Programming Language*, Benjamin/Cummings, Redwood City, CA.


Chandy, K. Mani and Taylor, Stephen (1992)  
*An Introduction to Parallel Programming*, Jones and Bartlett, Boston.


Foster, I. (1996)  


Pacheco, Peter S. (1997)  
*Parallel Programming with MPI*, Morgan Kaufmann Publishers Inc., San Francisco, CA.

Watt, David A. (1990)  

**Exercises**

For each of the following problems identify the potential for concurrent execution and the synchronization and communication requirements. Define appropriate safety and liveness invariants. Construct solutions using ...
- **Producer-Consumer/Bounded Buffer** (Models race conditions) Producers create data elements which are placed in a buffer. The consumers remove data elements from the buffer and perform some internal computation. The problem is to keep the producer from overwriting full buffers and the consumer from rereading empty buffers.

- **Readers and Writers** (Models access to a database) A data object is shared among several concurrent processes. Some of which only want to read the content of the shared object, whereas others want to update (read and write) the shared object. The problem is to ensure that only one writer at a time has access to the object. Readers are processes which are not required to exclude one another. Writers are required to exclude every other process, readers and writers alike.

- **The Dining Philosophers.** (Models exclusive access to limited resources) N philosophers spend their lives seated around a circular table thinking and eating. Each philosopher has a plate of spaghetti and, on each side, shares a fork his/her neighbor. To eat, a philosopher must first acquire the forks to its immediate left and right. After eating, a philosopher places the forks back on the table. The problem is to write a program that lets each philosopher eat and think.

  The philosophers correspond to processes and the forks correspond to resources.

  A safety property for this problem is that a fork is held by one and only one philosopher at a time. A desirable liveness property is that whenever a philosopher wants to eat, eventually the philosopher will get to eat.

  1. Solve the dining philosophers problem using a central fork manager (centralized).
  2. Solve the dining philosophers problem where there is a manager for each fork (distributed).
  3. Solve the dining philosophers problem where the philosophers handle their own forks (decentralized).
  4. Solve the dining philosophers problem if the philosophers must acquire all the forks in order to eat (distributed mutual exclusion).

- **Sleeping Barber** The barber shop has one barber, a barber chair, and n chairs for waiting customers. The problem is to construct an appropriate simulation.

- **Searching**
  1. Find the largest element in an unordered list

- **Sorting**
  1. Merge sort: Your program should break the list into two halves and sort each half concurrently. While sorting, the two halves should be concurrently merged.
  2. Parallel merge of sorted lists -- if X[i] should just precede Y[j], then X[i] should appear at Z[i+j-1].
  3. Rank sort: X[i] has rank k if X has exactly k items less than X[i] i.e., X[i] should be placed in position k.
  4. Insertion sort: value is placed into its place in the sorted list.
  5. Exchange/Bubble sort: small values flow left and large values flow right.
  6. Quicksort
  7. Bitonic sort

- **The N-body problem.** The N-body problem is used in astrophysics to calculate the dynamics of
the solar system and galaxies. Each mass in this problem experiences a gravitational attraction by every other mass, in proportion to the inverse square of the distance between the objects.

- **The sieve of Eratosthenes.** The sieve of Eratosthenes is a method of generating prime numbers by deleting composite numbers. This is done by the following beginning with two as the first prime:
  1. Delete all multiples of the prime number other than the prime number.
  2. Iterate with the next remaining number which is prime.

- **Polynomial Multiplication** -- initialize, form cross-product, sort by power, combine like powers

- **The quadrature problem.** The quadrature problem is to approximate the area under a curve, i.e., to approximate the integral of a function. Given a continuous, non-negative function f(x) and two endpoints l and r, the problem is to compute the area of the region bounded by f(x) the x axis, and the vertical lines through l and r. The typical way to solve the problem is to subdivide the regions into a number of smaller ones, using something like a trapezoid to approximate the area of each smaller region, and then sum the areas of the smaller regions.

- **Matrix Operations.**
  1. Multiplication: \( AB = C \) where A is a \( p \times q \) matrix, B a \( q \times r \) matrix, C a \( p \times r \) matrix and \( C[i,j] = \sum_{k1}^m A[i,k]B[k,j] \)
  2. Triangularization: Triangularization is a method for reducing a real matrix to upper-triangular form. It involves iterating across the columns and zeroing out the elements in the column below the diagonal element. This is done by performing the following step for each column.
    1. For each row \( r \) below the diagonal row \( d \), subtract a multiple of row \( d \) from row \( r \). The multiple is \( m[r,d]/m[d,d] \); subtracting this multiple of row \( d \) has the effect of setting \( m[r,d] \) to zero.
  3. Backsubstitution:
  4. Gaussian elimination: Gaussian elimination with partial pivoting is a method for reducing a real matrix to upper-triangular form. It involves iterating across the columns and zeroing out the elements in the column below the diagonal element. This is done by performing the following three steps for each column.
    1. Select a pivot element, which is the element in column \( d \) having the largest absolute value.
    2. Swap row \( d \) and the row containing the pivot element.
    3. For each row \( r \) below the new diagonal row, subtract a multiple of row \( d \) from row \( r \). The multiple is \( m[r,d]/m[d,d] \); subtracting this multiple of row \( d \) has the effect of setting \( m[r,d] \) to zero.

Assume the matrix is non-singular (the divisor is non-zero).

- **Shortest Path** between two vertices of a graph (edges are weighted).

- **Traveling salesman problem.** Find the shortest tour that visits each city exactly once.

- **Dutch national flag.** A collection of colored balls is distributed among N processes. There are at most N different colors of balls. The goal is for the processes to exchange balls so that eventually, for all i, process i holds all balls of color i. The number of balls in the collection is unknown to the processes.

- **Distributed Synchronization**
1. Write a program that polls N processes for yes or no votes and terminates when at least N/2 responses have been received. Assume N is even.

2. Repeat the previous exercise, but terminate when a majority of identical responses have been received. Assume N is even.

3. Random election of a leader amongst n processes. Create n processes. Let each process flip a coin to decide whether the process wants to contest the "elections". Broadcast this to all other processes. Now, each process generates a random number to decide its "vote", and sends the "vote" to the process it is voting for. Each process counts its votes, and broadcasts the results to all other processes. Now everyone knows the leader. (May have to think of starting the process over again in case of a tie, or simply deciding that the process with the larger Id is the leader, or some such thing.) This is a rather silly problem, but it will help your to learn about broadcasts and synchronizing processes, both of which are extremely important for any kind of parallel programming.

- The eight-queens problem. The eight-queens problem is concerned with placing eight queens on a chess board in such a way that none can attack another. One queen can attack another if they are in the same row or column or are on the same diagonal.

- Miscellaneous
  1. Sum a set of numbers
  2. (Conway) Read 80-character records, write 125 character records. Add an extra blank after each input record. Replace every pair of asterisks (**) with an exclamation point (!).
  3. (Manna and Pnueli) Compute \( (n \choose k) = \frac{n(n-1)...(n-k+1)}{k!} \)
  4. (Roussel) Compare the structure of two binary trees
  5. (Dijkstra) Let \( S \) and \( T \) be two disjoint sets of numbers with \( s \) and \( t \) the number of elements respectively. Modify \( S \) and \( T \) so that \( S \) contains the \( s \) smallest members of \( S \) union \( T \) and \( t \) the \( t \) largest members of \( S \) union \( T \).
  6. (Conway) The game of life
  7. (Hoare) Write a disk server that minimizes amount of seek time
  8. Show that Lewis' flow-correctness rules are safety or liveness rules.
  9. PCN is a single assignment language (in a single assignment language, the assignment of a value to a variable may occur just once within a program). In addition, when a program must reference an undefined variable, it waits until the variable becomes defined. Show that PCN programs satisfy Lewis' flow-correctness rules.

© 1996 by A. Aaby
Object-Oriented Programming

*Object-oriented programming is characterized by programming with objects, messages, and hierarchies of objects.*

The surest way to improve programming productivity is so obvious that many programmers miss it. Simply write less code.

-- Samuel P. Harbison

**Keywords and phrases:** Abstract Data Type, object-based, object-oriented, Inheritance, Object, sub-type, super-type, sub-range, sub-class, super-class, polymorphism, overloading, dynamic type checking, Class, Instance, method, message

Object-oriented programming shifts the emphasis from data as passive elements defined by relations or acted on by functions and procedures to active elements interacting with their environment. In the context of imperative programming, the emphasis shifts from describing control flow to describing interacting objects.

Object-oriented programming developed out of simulation programs. The conceptual model used is that the structure of the simulation should reflect the environment that is being simulated. For example, if an industrial process is to be simulated, then there should be an object for each entity involved in the process. The objects interact by sending messages.

Each object is designed around a data invariant.

Object-oriented programming is an abstraction and generalization of imperative programming. Imperative programming involves a state and a set of operations to transform the state. Object-oriented programming involves collections of objects each with a state and a set of operations to transform the state. Thus, object-oriented programming focuses on data rather than on control. As in the real world, objects interact so object-oriented programming uses the metaphor of message passing to capture the interaction of objects.

Functional objects are like values, imperative objects are like variables, active objects are like processes.

Alternatively, OOP, an object is a parameter (function and logic), an object is a mutable self (imperative).
Programming in an imperative programming language requires the programmer to think in terms of data structures and algorithms for manipulating the data structure. That is, data is placed in a data structure and the data structure is manipulated by various procedures.

Programming in an object-oriented language requires the programmer to think in terms of a hierarchy of objects and the properties possessed by the objects. The emphasis is on generality and reusability.

Procedures and functions are the focus of programming in an imperative language. Object-oriented programming focuses on data, the objects and the operations required to satisfy a particular task.

Object-oriented programming, as typified by the Small-talk model, views the programming task as dealing with objects which interact by sending messages to each other. Concurrency is not necessarily implied by this model and destructive assignment is provided. In particular, to the notion of an abstract data type, OOP adds the notion of inheritance, a hierarchy of relationships among types. The idea of data is generalized from simple items in a domain to data type (a domain and associated operations) to an abstract data type (the addition of information hiding) to OOP & inheritance.

Here are some definitions to enable us to speak the language of object-oriented programming.

- **Object**: Collection of private data and public operations.
- **Class**: Description of a set of objects. (encapsulated type: partitioned into private and public)
- **Instance**: An instance of a class is an object of that class.
- **Method**: A procedure body implementing an operation.
- **Message**: A procedure call. Request to execute a method.
- **Inheritance**: Extension of previously defined class. Single inheritance, multiple inheritance
- **Subtype principle**: a subtype can appear wherever an object of a supertype is expected.

I think a classification which helps is to classify languages as object-based and object-oriented. A report we recently prepared on OO technology trends reported that object-based languages support to varying degrees: object-based modularity, data abstraction (ADTs) encapsulation and garbage collection. Object-oriented languages additionally include to varying degrees: grouping objects into classes, relating those classes by inheritance, polymorphism and dynamic binding, and genericity.

Dr. Bertrand Meyer in his book 'Object-oriented Software Construction' (Prentice Hall) gives his 'seven steps to object-based (oriented) happiness'

1. Object-based modular structure
2. Data abstraction
3. Automatic memory management
4. Classes
5. Inheritance
6. Polymorphism and Dynamic Binding
7. Multiple and Repeated Inheritance

Subtypes (subranges)

The subtype principle states that a subtype may appear wherever an element of the super type is expected.

Objects

Objects are collections of operations that share a state. The operations determine the messages (calls) to which the object can respond, while the shared state is hidden from the outside world and is accessible only to the object's operations. Variables representing the internal state of an object are called instance variables and its operations are called methods. Its collection of methods determines its interface and its behavior.

Objects which are collections of functions but which do not have a state are functional objects. Functional objects are like values, they have the object-like interface but no identity that persists between changes of state. Functional objects arise in logic and functional programming languages.

Syntactically, a functional object can be represented as:

```latex
ame : object
methods
...
```

For example,

Objects which have an updateable state are imperative objects. Imperative objects are like variables. They are the objects of Simula, Smalltalk and C++. They have a name, a collection of methods which are activated by the receipt of messages from other objects, and instance variables which are shared by the methods of the object but inaccessible to other objects.

Syntactically, an imperative object can be represented as:

```latex
name : object
variables
...
```
Objects which may be active when a message arrives are active objects. In contrast, functional and imperative objects are passive unless activated by a message. Active objects have three modes: when there is nothing to do the object is dormant, when the agent is executing it is active, and when an object is waiting for a resource or the completion of subtasks it is waiting. Messages sent to an active object may have to wait in queue until the object finishes a task. Message passing among objects may be synchronous or asynchronous.

**Figure M.N: Object implementation**

<table>
<thead>
<tr>
<th>Instance data</th>
<th>Shared methods</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>methods</td>
<td>method₁</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>field₁</td>
<td>methodₙ</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fieldₘ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Classes**

Classes serve as templates from which objects can be created. Classes have the same instance variables and operations as corresponding objects but their interpretation is different. Instance variables in an object represent actual variables while class instance variables are potential, being instantiated only when an object is created.

We may think of a class as specifying a behavior common to all objects of the class. The instance variables specify a structure (data structure) for realizing the behavior. The public operations of a class determine its behavior while the private instance variables determine its structure.

Private copies of a class can be created by a make-instance operation, which creates a copy of the class
instance variables that may be acted on by the class operations.

Syntactically, a class can be represented as:

```
name : class

  instance variables
  ...

  class variables
  ...

  instance methods
  ...
  class methods
  ...
```

Classes specify the behavior common to all elements of the class. The operations of a class determine the behavior while the instance variables determine the structure.

**Algebraic semantics**

Many sorted algebras may be used to model classes.

**Inheritance**

Inheritance allows us to reuse the behavior of a class in the definition of new classes. Subclasses of a class inherit the operations of their parent class and may add new operations and new instance variables.

Inheritance captures a form of abstraction called *super-abstraction*, that complements data abstraction. Inheritance can express relations among behaviors such as classification, specialization, generalization, approximation, and evolution.

Inheritance classifies classes in much the way classes classify values. The ability to classify classes provides greater classification power and conceptual modeling power. Classification of classes may be referred to as second-order classification. Inheritance provides second-order sharing, management, and manipulation of behavior that complements first-order management of objects by classes.

Syntactically, inheritance may be specified in a class as:

```
name : class
```
super class
...
instance variables
{ as before }

What should be inherited? Should it be behavior or code: specification or implementation? Behavior and code hierarchies are rarely compatible with each other and are often negatively correlated because shared behavior and shared code have different requirements.

Representation, Behavior, Code

DYNAMIC/STATIC/INHERITANCE

Inheritance and OOP

Type hierarchy

Semantics of inheritance in the functional paradigm.

type op params = case op of
  f0 : f0 params
...
  fn : fn params
  otherwise : supertype op params
  where
    f0 params = def0
...
  fn params = defn

inheritance in the logic programming paradigm.

object(structure,methodslist).

isa(type1,type2).

object(rectangle(Length,Width),[area(A is Length*Width)]).
Algebraic semantics

*Order-sorted algebras* are required to capture the ordering relations among sorts that arise in subtypes and inheritance.

**Types and Classes**

The concept of a type and the concept of a class have much in common and depending on the point of view, they may be indistinguishable. The distinction between types and classes may be seen when we examine the compare the inheritance relationship between types and subtypes with the inheritance relationship between classes and subclasses.

**Example:** The natural numbers are a subtype of the integers but while subtraction is defined for all pairs of integers it is not defined for all pairs of natural numbers.

This is an example of subtype inheritance. Subtypes are defined by additional constraints imposed on a type. The set of values satisfying a subtype is a subset of the set of values satisfying the type and subtypes inherit a subset of the behaviors of the type.

**Example:** The integers are a subclass of the natural numbers since, the subtraction operation of the natural numbers can be extended to subtraction for integers.

**Example:** The rational numbers are a subclass of the integers since, they can be defined as pairs of natural numbers and the arithmetic operations on the rational numbers can be defined in terms of the
Object-Oriented Programming

arithmetic operations on natural numbers.

These are examples of subclass inheritance. Subclasses are defined by extending the class behavior. This means that subclasses are more loosely related to their parent than a subtype to a type. Both state and methods may be extended.

Subtyping strongly contrains behavior while subclassing is an unconstrained mechanism. It is the inheritance mechanism of OOP that distinguishes between types and classes.

These examples illustrate that subtype inheritance is different from subclass inheritance. Subclasses may define behavior completely unrelated to the parent class.

Types are used for type checking while classes are used for generating and managing objects with uniform properties and behaviors. Every class is a type. However, not every type is a class, since predicates do not necessarily determine object templates. We will use the term type to refer to structure and values while the term class will be used to refer to behavior.

Examples

Queue -- insert_rear, delete_front
Deque -- insert_front, delete_front, insert_rear, delete_rear
Stack -- push, pop
List -- cons, head, tail
Binary tree -- insert, remove, traverse
Doublely linked list --
Graph -- linkto, path,
Natural numbers -- Ds
Integers -- (=-,Ds)
Rationals
Reals -- (+-,Ds,Ds)
Complex (a,b) or (r,\theta)

Historical Perspectives and Further Reading

- History
  - Simula
  - ADT
  - Small-Talk
  - Modula-2, C++, Eiffel
  - Flavors, CLOS
- Subtypes (subranges)
Object-Oriented Programming

- Generic types
- Inheritance -- Scope generalization
- OOP
  - Objects--state + operations
  - Object Classes-- Class, Subclass

- Objects--state + operations
- Object Classes-- Class, Subclass
- Inheritance mechanism

Much of this section follows Peter Wegner\cite{Wegner90}.

**Exercises**

1. [time/difficulty] (section) Problem statement
2. Stack
3. Queue
4. Tree
5. Construct a `turtle graphics`
6. Construct a table handler
7. Grammar
8. Prime number sieve
9. Account, Checking, Savings
10. Point, circle

© 1996 by A. Aaby
Introduction

Evaluation of Programming Languages

Models of Computation

The first requirement for a general purpose programming language is that its computational model must be universal. That is, every problem that has an algorithmic solution must be solvable in the computational model. This requirement is easily met as the lambda calculus and the imperative model show. The computational model must be implementable on a computer.

Syntax

Principle

**Simplicity:** The language should be based upon as few "basic concepts" as possible.

**Orthogonality:** Independent functions should be controlled by independent mechanisms.

**Regularity:** A set of objects is said to be regular with respect to some condition if, and only if, the condition is applicable to each element of the set. The basic concepts of the language should be applied consistently and universally.

**Type Completeness:** There should be no arbitrary restriction on the use of the types of values. All types have equal status. For example, functions and procedures should be able to have any type as parameter and result. This is also called the principle of regularity.

**Parameterization:** A formal parameter to an abstract may be from any syntactic class.

**Analogy:** An analogy is a conformation in pattern between unrelated objects. Analogies are generalizations which are formed when constants are replaced with variables resulting in similarities in structure. Analogous operations should be performed by the same code parameterized by the type of the objects.

**Correspondence:** For each form of definition there exists a corresponding parameter mechanism and **vice versa**.

Semantics
Principle

**Clarity:** The mechanisms used by the language should be well defined, and the outcome of a particular section of code easily predicted.

**Referential Transparency:** Any part of a syntactic class may be replaced with an equal part without changing the meaning of the syntactic class (substitutivity of equals for equals).

**Sub-types:** A sub-type may appear wherever an element of the super-type is expected.

Pragmatics

- Naturalness for the application (relations, functions, objects, processes)
- Support for abstraction
- Ease of program verification
- Programming environment (editors, debuggers, verifiers, test data generators, pretty printers, version control)
- Operating Environment (batch, interactive, embedded-system)
- Portability
- Cost of use (execution, translation, programming, maintenance)

Applicability

Principle

**Expressivity:** The language should allow us to construct as wide a variety of programs as possible.

**Extensibility:** New objects of each syntactic class may be constructed (defined) from the basic and defined constructs in a systematic way.

Example: user defined data types, functions and procedures. Binding, Scope, Lifetime,

Safety

Principle

**Safety:** Mechanisms should be available to allow errors to be detected.
Type checking-static and dynamic, range checking

Principle

the Data Invariant: A data invariant is a property of an object that holds whenever control is not in the object. Objects should be designed around a data invariant.

Information Hiding: Each "basic program unit" should only have access to the information that it requires.

Explicit Interfaces: Interfaces between basic program units should be stated explicitly.

Privacy: The private members of a class are inaccessible from code outside the class.

Abstraction

Principle

Abstraction: Abstraction is an emphasis on the idea, qualities and properties rather than the particulars (a suppression of detail). An abstract is a named syntactic construct which may be invoked by mentioning the name. Each syntactic class may be referenced as an abstraction. Functions and procedures are abstractions of expressions and commands respectively and there should be abstractions over declarations (generics) and types (parameterized types). Abstractions permit the suppression of detail by encapsulation or naming. Mechanisms should be available to allow recurring patterns in the code to be factored out.

Qualification: A block may be introduced in each syntactic class for the purpose of admitting local declarations. For example, block commands, block expressions, block definitions.

Representation Independence: A program should be designed so that the representation of an object can be changed without affecting the rest of the program.

Generalization

Principle

Generalization: Generalization is a broadening of application to encompass a larger domain of objects of the same or different type. Each syntactic class may be generalized by replacing a constituent element with a variable. The idea is to enlarge of domain of
applicability of a construct. Mechanisms should be available to allow analogous operations to be performed by the same code.

polymorphism, overloading, generics

**Implementation**

**Principle**

**Efficiency:** The language should not preclude the production of efficient code. It should allow the programmer to provide the compiler with any information which may improve the resultant code.

**Modularity:** Objects of each syntactic class may be compiled separately from the rest of the program.

Novice users of a programming language require *language tutorials* which provide examples and intuitive explanations. More sophisticated users require *reference manuals* which catalogue all the features of a programming language. Even more sophisticated students of a programming language require complete and formal descriptions which eliminate all ambiguity from the language description.

**Trends in Programming Language Design**

streams, lazy evaluation, reactive systems, knowledge based systems, concurrency, efficient logic and functional languages, OOP.


© 1996 by A. Aaby
Stack Machine

Objectives
To introduce the machine organization and programming of a stack machine.

Concepts
Lab Techniques
Prerequisites

Background
In a stack machine most instructions obtain their arguments from the stack and place their results on the stack.

Machine Organization
The stack machine consists of a code segment C for the program, a data segment D for data and a stack, a program counter PC to contain the address of the next instruction, a stack top T pointer to point to the top of the expression stack (also part of the Store), an instruction register I to hold the current instruction, an input device Input and an output device Output.

Instruction Set
An instruction consists of an operation code and at most one parameter. The action of the instruction is described using a mixture of English language description and mathematical formalism. The mathematical formalism is used to note changes in values that occur to the registers, the data store, the program counter and the input and output devices.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Semantics</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>lit</td>
<td>X</td>
<td>D[T+1] := X; T := T+1</td>
<td>Push X on the Stack</td>
</tr>
</tbody>
</table>
load src \(D[T+1] := D[src]; T := T+1\) Push from Store to Stack
store dst \(D[dst] := D[T]; T := T-1\) Copy top of stack to Store
st X \(T := X\) Set stack top pointer to X
jmp \(PC := D[T]; T := T-1\) Unconditional jump
jmpz if \(D[T] = 0\) then
\(PC := D[T-1]; T := T-2\) Jump on zero
else \(T := T-2\)
jmpn if \(D[T] < 0\) then
\(PC := D[T-1]; T := T-2\) Jump on negative
else \(T := T-2\)
halt Halt
read \(D[T+1] := Input; T := T+1\) Push input item on the Stack
write \(Output := D[T]; T := T-1\) Put top of Stack to Output

src and dst designate source and destination respectively. Division by zero results in unpredictable results.

**Operation**
Most operations find their arguments on the expression stack. The program counter is initialized to the location of the first instruction. The machine repeatedly fetches the instruction at the address in the PC, increments the PC and executes the instruction and stops when the the halt instruction encountered.

```
PC := 0;
repeat
  I := C[PC]; {Fetch instruction}
  PC := PC+1; {Increment program counter}
  execute(I); {Execute instruction}
until I = halt
```

**Activities**

**Assignment**

1. Write a program to read two numbers from the input, compute and print their sum on the output.
2. Write a program to read two numbers from the input and print the value of the largest to the output.
3. Use a sentinel controlled loop to read non-negative numbers, compute and print their sum.
4. Use a counter controlled loop to read seven numbers, some positive and some negative and compute and print their average.
5. Read a series of numbers and determine and print the largest number. The first number read indicates how many numbers should be processed.
6. Implement the stack machine.

**Hand in**  
**Extra Credit**

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of Anthony A. Aaby. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee.

© 1998 Anthony A. Aaby. Last Modified -. Send comments to aabyan@wwc.edu
The goal of this work is

- to design an abstract grammar for those elements that programming languages have in common in particular, for abstraction, generalization, and modules and
- to integrate the grammar with abstract grammars for a variety of programming paradigms.

This work is supports ideas developing in *Introduction to Programming Languages* where abstraction, generalization and computational models are used as unifying concepts for understanding programming languages. The goal in that document is to provide a top-down description of the language design process - idea, abstract syntax, semantics, concrete syntax, formal semantics, and implementation

- The design [description](#)
- The syntax ([grammar](#))
- The semantics
- The implementation

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of Anthony A. Aaby. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. © 1998 Anthony A. Aaby. Last Modified - . Send comments to [aabyan@wwc.edu](mailto:aabyan@wwc.edu)
AbSus85
Abelson, H., Sussman, G.J., and Sussman, J.

Backus78
Backus, J. W., ``Can Programming Be Liberated from the von Neumann Style?'' CACM, vol. 21, no. 8, pp. 613-614.

Bare84

BirdWad88
Bird, R.A. and Wadler, P.L.,

Boehm66

CF68

CHS72

DJL88

Dijk68

Foster96
Foster, I., Compositional Parallel Programming Languages *TOPLAS* Vol 18 No. 4 (July 1996): pp. 454-476.

Gries81

Hehner84
Hehner, E. C. R.,

Hend80
Henderson, Peter,
HS86

HU79

Knuth68

Kowalski79

Landin66

McCarthy60

McCarthy65

MLennan90

Miller67

PJones87

PittPet92

Pratt84

Revesz88

Schmidt88

SchFre85
Schreiner, A. T. and Freidman, H. G., *Introduction to Compiler Construction with Unix* Prentice-
Hall, 1985.

Scott87

Steele84

Tenn81

Wegner90

Worf56
Worf, Benjamin,
Definitions

abstract type
An abstract data type ---
abstraction
An abstraction is the
actual parameter
aliasing
Aliasing occurs whenever a given object becomes accessible through more than one name.
actual parameter
argument
assembly language
assertion
backtrack
binding
Binding is an association between two objects.
block
class
clause
coertion
composite type
A composite type is a type whose values are compose of simpler values.
computation
A computation is the application of a sequence of operations to a set of values to yield a value.
computational model
A computational model is a collection of values and operations.
concurrent programming
Concurrent programming is characterized by programming with more than one process.
context
context-sensitive
A syntactic element is context-sensitive if its value depends on the context in which it appears.
coroutine
deadlock
domain
environment
exception
formal parameter
functional programming
Functional programming is characterized by programming with values, functions and functional forms.

generator

imperative programming

Imperative programming is characterized by programming sequential modifications to a state.

inheritance

instance

iterator

lexical analyzer

a scanner

live-lock

liveness

logic programming

Logic programming is characterized by programming with relations and deduction.

machine language

1's and 0's.

method

module

object

object-oriented programming

Object-oriented programming is characterized by programming with objects, messages, and hierarchies of objects.

overloading

parameter

partition

polymorphism

pragmatics

The pragmatics of a programming language describe the degree of success with which a programming language meets its goals both in its faithfulness to the underlying model of computation and in its utility for human programmers.

primitive type

A primitive type is a type whose values cannot be decomposed. The values are atomic.

process

program

A program is a specification of a computation.

programming language

A programming language is a notation for writing programs.

recursive type

A recursive type is a type whose values may be composed of other values of the same type.

safety

scanner

A scanner is a program which groups characters in an input stream into a sequence of tokens.

scope
semantics
The *semantics* of a programming language describe the relationship between the syntactical elements and the model of computation.

semantic algebra
A *semantic algebra* is a set of values and operations defined on those values. A semantic algebra is distinguished from a type in that semantic algebras are the objects denoted in denotational semantics while types are the syntactic objects.

semantic domain
A *semantic domain* is a set of values.

side effect
A *side effect* is a modification of a non-local environment.

starvation
state

static semantics
The *static semantics* is the description of the structural constraints (context-sensitive aspects) that cannot be adequately described by context-free grammars.

symbol table

syntax
The *syntax* of a programming language describes the structure of programs.

type
A *type* is a set of values and a set of operations (see semantic algebra).

value
A *value* is any thing that may be evaluated, stored, incorporated in a data structure, passed as an argument to a procedure or function, returned as a function result, and so on.

variable

virtual machine

© 1996 by [A. Aaby](mailto:aabyan@wwc.edu)
Index

A B C L Y Z

A
B
C
D
E
F
G
H
I
J
K
L

lazy evaluation
lazy reduction

M
N
O

R

Reduction
Eager
lazy

Z

© 1996 by A. Aaby