Theory

Introduction to Programming Languages

Anthony A. Aaby

DRAFT Version 0.9. Edited July 15, 2004

Copyright © 1992-2004 by Anthony A. Aaby

Walla Walla College
204 S. College Ave.
College Place, WA 99324
E-mail: aabyan@wwc.edu

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit http://creativecommons.org/licenses/by/2.0/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

This book is distributed in the hope it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

No explicit permission is required from the author for reproduction of this book in any medium, physical or electronic.

The author solicits collaboration with others on the elaboration and extension of the material in this text. Users are invited to suggest and contribute material for inclusion in future versions provided it is offered under compatible copyright provisions. The most current version of this text and LaTeXsource is available at http://www.cs.wwc.edu/~aabyan/Logic/index.html.

To Ogden and Amy Aaby

Preface

It is the purpose of this text to explain the concepts underlying programming languages and to examine the major language paradigms that use these concepts.

Programming languages can be understood in terms of a relatively small number of concepts. In particular, a programming language is syntactic realization of one or more computational models. The relationship between the syntax and the computational model is provided by a semantic description. Semantics provide meaning to programs. The computational model provides much of the intuition behind the construction of programs. When a programming language is faithful to the computational model, programs can be more easily written and understood.

The fundamental concepts are supported bindings, abstraction and generalization. Concepts so fundamental that they are included in virtually every programming language. These concepts support the human facility for simile and metaphor which are so necessary in problem solving and in managing complexity.

Programming languages are also shaped by pragmatic considerations. Formost among these considerations are safety, efficiency and applicability. In some languages these external forces have played a more important role in shaping the language than the computational model to the point of distorting the language and actually limiting the applicability of the language. There are several distinct computational models — imperative, functional, and logic. While these models are equivalent (all computable functions may be defined in each model), there are pragmatic reasons for prefering one model over the another.

This text is designed to formalize and consolidate the knowledge of programming languages gained in the introductory courses a computer science curriculum and to provide a base for further studies in the semantics and translation of programming languages. It aims at covering the bulk of the subject area *PL: Programming Languages* as described in the "ACM/IEEE Computing Curricula 1991."

Special Features of the Text

The following are significant features of the text as compared to the standard texts.

- Syntax: an introduction to regular expressions, scanning, context-free grammars, parsing, attribute grammars and abstract grammars.
- Semantics: introductory treatment of algebraic, axiomatic, denotational and operational semantics.
- Programming Paradigms: the major programming paradigms are prominently featured.
  - Functional: includes an introduction to the lambda calculus and uses the programming languages Scheme and Haskell for examples
  - Logic: includes an emphasis on the formal semantics of Prolog
  - Concurrent: introduces both low- and high-level notations for concurency, stresses the importance of the logic and functional paradigms in the debate on concurrency, and uses the programming language SR for examples.
  - Object-oriented: uses the programming language Modula-3 for examples
- Language design principles: Twenty some programming language design principles are given prominence. In particular, the importance of abstraction and generalization is stressed.

Readership

This book is intended as an undergraduate text in the theory of programming languages. To gain maximum benefit from the text, the reader should have experience in a high-level programming language such as Pascal, Modula-2, C++, ML or Common Lisp, machine organization and programming, and discrete mathematics.

Programming is not a spectator sport. To gain maximum benefit from the text, the reader should construct programs in each of the paradigms, write semantic specifications; and implement a small programming language.

Organization

Since the subject area *PL: Programming Languages* as described in the "ACM/IEEE Computing Curricula 1991" consists of a minimum of 47 hours of lecture, the text contains considerably more material than can be covered in a single course.

The first part of the text consists of chapters 1–3. Chapter 1 is an overview of the text, an introduction to the areas of discussion. It introduces the key concepts: the models of computation, syntax, semantics, abstraction, generalization and pragmatics which are elaborated in the rest of the text. Chapter 2 introduces context-free grammars, regular expressions, and attribute grammars. Context-free grammars are utilized throughout the text but the other sections are optional. Chapter 3 introduces semantics: algebraic, axiomatic, denotational and operational. While the chapter is optional, I introduce algebraic semantics in conjunction with abstract types and axiomatic semantics with imperative programming.

Chapter 4 is a formal treatment of abstraction and generalization as used in programming languages.

Chapter 5 deals with values, types, type constructors and type systems. Chapter 6 deals with environments, block structure and scope rules. Chapter 7 deals with the functional model of computation. It introduces the lambda calculus and examines Scheme and Haskell. Chapter 8 deals with the logic model of computation. It introduces Horn clause logic, resolution and unification and examines Prolog. Chapter 9 deals with the imperative model of computation. Features of several imperative programming languages are examined. Various parameter passing mechanisms should be discussed in conjunction with this chapter. Chapter 10 deals with the concurrent model of programming. Its primary emphasis is from the imperative point of view. Chapter 11 is a further elaboration of the concepts of abstraction and generalization in the module concept. It is preparatory for Chapter 12. Chapter 12 deals with the object-oriented model of programming. Its primary emphasis is from the imperative point of view. Features of Smalltalk, C++ and Modula-3 provide examples.

Chapter 13 deals with pragmatic issues and implementation details. It may be read in conjunction with earlier chapters. Chapter 14 deals with programming environments, Chapter 15 deals with the evaluation of programming languages and a review of programming language design principles. Chapter 16 contains a short history of programming languages.

Pedagogy

The text provides pedagogical support through various exercises and laboratory projects. Some of the projects are suitable for small group assignments. The exercises include programming exercises in various programming languages. Some are designed to give the student familiarity with a programming concept such as modules, others require the student to construct an implementation of a programming language concept. For the student to gain maximum benefit from the text, the student should have access to a logic programming language (such as Prolog), a modern functional language (such as Scheme, ML or Haskell), a concurrent programming language (Ada, SR, or Occam), an object-oriented programming language (C++, Small-Talk, Eiffel, or Modula-3), and a modern

programming environment and programming tools. Free versions of Prolog, ML, Haskell, SR, and Modula-3 are available from one or more ftp sites and are recommended.

The instructor's manual contains lecture outlines and illustrations from the text which may be transferred to transparencies. There is also a laboratory manual which provides short introductions to Lex, Yacc, Prolog, Haskell, Modula-3, and SR.

The text has been used as a semester course with a weekly two hour lab. Its approach reflects the core area of programming languages as described in the report **Computing as a Discipline** in CACM January 1989 Volume 32 Number 1.

Knowledge Unit Mapping

To assist in curriculum development, the follow mapping of the ACM knowledge units to the text is provided.

Knowledge Unit	**Chapter(s)**
PL1: History	6,7,8,9,11
PL2: Virtual Machines	2,6,7,8,13
PL3: Data Types	5,13
PL4: Sequence Control	9,10
PL5: Data Control	5,11,12
PL6: Run-time	2,13
PL7: Regular Expressions	2
PL8: Context-free grammars	2
PL9: Translation	2
PL10: Semantics	3
PL11: Programming Paradigms	1,7,8,9,10,12
PL12: Parallel Constructs	10
SE3: Specifications	3

Acknowledgements

There are several programming texts that have influenced this work in particular, texts by Hehner, Tennent, Pratt, and Sethi. I am grateful to my CS208 classes at Bucknell for their comments on preliminary versions of this material and to Bucknell University for providing the excellent environment in and with which to develop this text.

AA 1992

Contents

Preface **v**

1 Introduction **1**

1.1 Models of Computation 3
1.2 Syntax and Semantics 7
1.3 Pragmatics 7
1.4 Language Design Principles 8
1.5 Further Reading 10
1.6 Exercises 11

2 Syntax **13**

2.1 Context-Free Grammars 14
2.2 Regular Expressions 21
2.3 Attribute Grammars and Static Semantics 23
2.4 Further Reading 24
2.5 Exercises 25

3 Semantics **27**

3.1 Algebraic Semantics 28
3.2 Axiomatic Semantics 29
3.3 Denotational Semantics 36
3.4 Operational Semantics 37
3.5 Further Reading 39

4 Abstraction and Generalization I **41**

4.1 Abstraction 43

4.2 Generalization 44
4.3 Substitution 46
4.4 Abstraction and Generalization 46
4.5 Exercises 47

5 Domains and Types **49**
5.1 Primitive Domains 52
5.2 Compound Domains 52
5.3 Abstract Types 60
5.4 Generic Types 63
5.5 Type Systems 64
5.6 Overloading and Polymorphism 67
5.7 Type Completeness 69
5.8 Exercises 69

6 Environment **71**
6.1 Block structure 72
6.2 Declarations 74
6.3 Constants 74
6.4 User Defined Types 74
6.5 Variables 78
6.6 Functions and Procedures 78
6.7 Persistant Types 78
6.8 Exercises 79

7 Functional Programming **81**
7.1 The Lambda Calculus 83
7.2 Recursive Functions 88
7.3 Lexical Scope Rules 90
7.4 Functional Forms 91
7.5 Evaluation Order 92
7.6 Values and Types 93
7.7 Type Systems and Polymorphism 93
7.8 Program Transformation 93
7.9 Pattern matching 94

7.10 Combinatorial Logic 94
7.11 Scheme 96
7.12 Haskell 98
7.13 Discussion and Further Reading 100
7.14 Exercises 101

8 Logic Programming 103
8.1 Inference Engine 105
8.2 Syntax 105
8.3 Semantics 106
8.4 The Logical Variable 114
8.5 Iteration vs Recursion 117
8.6 Backtracking 118
8.7 Exceptions 118
8.8 Prolog ≠ Logic Programming 118
8.9 Database query languages 124
8.10 Logic Programming vs Functional Programming 125
8.11 Further Reading 125
8.12 Exercises 125

9 Imperative Programming 127
9.1 Variables and Assignment 128
9.2 Control Structures 130
9.3 Sequencers 135
9.4 Jumps 136
9.5 Escape 137
9.6 Exceptions 138
9.7 Coroutines 140
9.8 Processes 140
9.9 Side effects 140
9.10 Aliasing 141
9.11 Reasoning about Imperative Programs 143
9.12 Expressions with side effects 143
9.13 Sequential Expressions 143
9.14 Structured Programming 144

9.15 Expression-oriented languages 145
9.16 Further Reading 145

10 Concurrent Programming **147**
10.1 Concurrency 148
10.2 Issues in Concurrent Programming 150
10.3 Syntax 153
10.4 Interfering Processes 153
10.5 Non-interfering Processes 154
10.6 Cooperating Processes 154
10.7 Synchronizing Processes 155
10.8 Communicating Processes 157
10.9 Occam 158
10.10Semantics 158
10.11Related issues 159
10.12Examples 159
10.13Further Reading 159

11 PCN **161**
11.1 Tutorial 161
11.2 The PCN Language 161
11.3 Examples 162

12 Abstraction and Generalization II **163**
12.1 Encapsulation 164
12.2 ADTs 165
12.3 Partitions 165
12.4 Scope Rules 166
12.5 Modules 167

13 Object-Oriented Programming **169**
13.1 History 172
13.2 Subtypes (subranges) 172
13.3 Objects 172
13.4 Classes 173
13.5 Inheritance 174

13.6 Types and Classes 175
13.7 Examples 176
13.8 Further Reading 177
13.9 Exercises 177

14 Pragmatics 179

14.1 Syntax 179
14.2 Semantics 179
14.3 Bindings and Binding Times 180
14.4 Values and Types 181
14.5 Computational Models 182
14.6 Procedures and Functions 182
14.7 Scope and Blocks 183
14.8 Parameters and Arguments 187
14.9 Safety 189
14.10Further Reading 190
14.11Exercises 190

15 Translation 191

15.1 Parsing 193
15.2 Scanning 193
15.3 The Symbol Table 193
15.4 Virtual Computers 193
15.5 Optimization 193
15.6 Code Generation 195
15.7 Peephole Optimization 199
15.8 Further Reading 199

16 Evaluation of Programming Languages 201

16.1 Models of Computation 201
16.2 Syntax 201
16.3 Semantics 202
16.4 Pragmatics 202
16.5 Trends in Programming Language Design 205

17 History 207

17.1 Functional Programming 207
17.2 Logic Programming 208
17.3 Imperative Programming 208
17.4 Concurrent Programming 209
17.5 Object-Oriented Programming 209

A Logic **211**

A.1 Sentential Logic 211
A.1.1 Syntax 211
A.1.2 Semantics 212
A.2 Predicate Logic 214
A.2.1 Syntax 214
A.2.2 Semantics 215

Chapter 1

Introduction

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

Keywords and phrases: Computational model, computation, program, programming language, syntax, semantics, pragmatics, binding, scope.

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

$$3.14 \times 5$$

the result of which is the value:

$$15.7$$

The value 3.14 is readily recognized as an approximation for π. The actual numeric value may be less important than knowing that an approximation to π is intended so we can replace 3.14 with π. *abstracting* the expression 3.14×5 to:

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

We say that π is *bound* to 3.14 and is a *constant*. The "where" introduces a *local environment* or *block* where additional definitions may occur.

If the 5 is intended to be the value of a diameter and the computation is intended to derive the value of the circumference, then the expression can be *generalized*

by introducing a variable for the diameter:

$$\pi \times diameter \text{ where } \pi = 3.14$$

The expression may be further abstracted by assigning a name to the expression as is done in this equation:

$$Circumference = \pi \times diameter \text{ where } \pi = 3.14$$

This definitions binds the name *Circumference* to the expression $\pi \times diameter$. The variable *diameter* is said to be *free* in the right hand side of the equation. It is a variable since its value is not determined. π is not a variable, it is a *constant*, the name of a particular value. Any context (*scope*) in which these definitions appear and in which the variable *diameter* appears and is assigned to a value determines a value for *Circumference*. A further generalization is possible by parameterizing *Circumference* with the variable *diameter*

$$\mathit{Circumference}(diameter) = \pi \times diameter \text{ where } \pi = 3.14$$

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

$$Circumference(5) = 15.7$$

The parameter *diameter* is *bound* to the value 5 and, as a result, *Circumference(5)* is bound to 15.7.

In this form, the definition is a recipe or program for computing the circumference of a circle from the diameter 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 it is difficult to distinguish between the notation and the computational model. This example serves to illustrate several key ideas in the study of programming languages which are summarized in the following definitions:

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 the syntactical elements 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.*

1.1 Models of Computation

Computational models begin with a set of values. The values can be separated into two groups, primitive and composite. The primitive values (or types) are usually numbers, boolean values, and characters. The composite values (or types) 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 type are operations to construct the values of that type and operations to access component elements of the type.

In addition to the set of values and associated operations, each computational model has a set of operations which are used to define computation. There are three basic computational models—functional, logic, and imperative. In addition, there are two programming techniques or programming paradigms (concurrent programming and object-oriented programming); while they are not models of computation, they are so influential that whey rank in importance with computational models.

The Functional Model

The *functional model* of computation consists of a set of values, functions, and the operation of function application. Functions may be named and may be composed with other functions. Functions can take other functions as arguments and return functions as results. Programs consist of definitions of functions and computations are application of functions to values. For example, a linear function y = 2x + 3 can be defined as follows:

```
f x = 2*x + 3
```

A more interesting example is a program to compute the standard deviation of a list of scores. The formula for standard deviation is:

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

where x_i is an individual score and N is the number of scores. An implementation in a functional programming language might look like this:

```
sd xs = sqrt( (sumsqrs( xs ) -
             (sum( xs )^2 / length( xs ) )) / length( xs ))
```

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 Logic Model

The *logic model* of computation is based on relations and logical inference. Programs consist of definitions of relations and computations are inferences. For example the linear function y = 2x + 3 can be represented as:

```
f(X,Y) if Y is 2*X + 3.
```

The function is represented as a relation between X and Y. A more typical application for logic programming is illustrated by a program to determine the mortality of Socrates. Suppose we have the following set of sentences.

1. man(Socrates)
2. mortal(X) if man(X)

The first line is a translation of the statement *Socrates is a man.* The second line is a translation of the phrase *all men are mortal* into the equivalent *for all X, if X is a man then X is mortal.* To determine the mortality of Socrates. The following sentence must be added to the set.

$$\neg \text{ mortal(Y)}$$

This sentence is a translation of the phrase *There are no mortals* rather than the usual phrase *Socrates is not mortal.* It can be viewed as the question, "Is there a mortal?" The first step in the computation is illustrated here

1. man(Socrates)
2. mortal(X) if man(X)
3. ¬ mortal(Y)
4. ¬ man(Y)

The deduction of line 4 from lines 2 and 3 is to be understood from the fact that if the conclusion of a rule is known to be false, then so is the hypothesis (modus tollens). Using this new result, we get a contradiction with the first sentence.

1. man(Socrates)
2. mortal(X) if man(X)
3. ¬ mortal(Y)
4. ¬ man(Y)
5. Y = Socrates

From the resolvent and the first sentence, resolution and unification produce Y=Socrates. That is, there is a mortal and one such mortal is Socrates. *Resolution* is the process of looking 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 Imperative Model

The *imperative model* of computation consists of a state and the operation of assignment which is used to modify the state. Programs consist of sequences of commands and computations are changes of the state. For example, the linear function y = 2x + 3 written as:

```
Y := 2*X + 3
```

requires the implementation to determine the value of `X` in the state and then create a new state which differs from the old state in that the value of `Y` in the new state is the value that 2∗X + 3 had in the old state.

```
Old State:  X = 3, Y = -2, ...
    Y := 2*X+3
New State:  X = 3, Y = 9, ...
```

The imperative model is important because it models change and changes are part and parcel of our environment. In addition, it is the closest to modeling

the hardware on which programs are executed. This tends to make it the most efficient model in terms of execution speed.

Other Models

Programs in the *concurrent programming* model consist of multiple processes or tasks which may exchange information. The computations may occur concurrently or in any order. Concurrent programming is primarily concerned with methods for synchronization and communication between processes. The concurrent programming model may be implemented within any of the other computational models. Concurrency in the imperative model can be viewed as a generalization of control. Concurrency within the functional and logic model is particularly attractive since, subexpression evaluation and inferences may be performed concurrently. For example, $3x$ and $4y$ may be simultaneously evaluated in the expression $3x + 4y$.

Programs in the *object-oriented programming* model consist of a set 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. The object-oriented model may be implemented within any of the other computational models.

Computability

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 predominates. Lisp, Scheme, and ML are based on the functional model of computation but provide imperative constructs as well while, Miranda and Haskell provide a nearly pure implementation of the functional model of computation. Prolog attempts to provide an implementation of the logic computational model but, for reasons of efficiency and practicality, fails in several areas and contains imperative constructs. Imperative programming languages provide a severely limited implementation of the functional and logic model of computation.

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.

1.2 Syntax and Semantics

The notation used in the functional and logic models tends to reflect common mathematical practice and thus, it tends toward simplicity and regularity. On the other hand, the notation used for the imperative model tends to be irregular and of greater complexity. The problem is that in the imperative model the programmer must both manage storage and determine the appropriate computations. This tends to permit programs that are more efficient in their use of time and space than equivalent functional and logic programs. The addition of concurrency to imperative programs results in additional syntactic structures while concurrency in functional and logic programs is more of an implementation issue.

The relationship between the syntax and the computational model is provided by semantic descriptions. The semantics of imperative programming languages tends to receive more attention because changes to state need not be restricted to local values. In fact, the bulk of the work done in the area of programming language semantics deals with imperative programming languages.

Since semantics ties together the syntax and the computational model, there are several programming language design principles which are deal with the interaction between these three areas. Since syntax is the means by which computation is specified, the following programming language design principle deals with the relationship which must exist between syntax and the compuational model.

Principle of Clarity: The mechanisms used by the language should be well defined, and the outcome of a particular section of code easily predicted.

1.3 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. For a language to have wide applicability it must make provision for abstraction, generalization and modularity. Abstraction 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 permits the application of constructs to wider 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.

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.

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 and the formalized symbolic notations of mathematics and logic and various 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. Benjamin Whorf[32] has postulated that one's language has considerable effect on the way that one thinks; indeed on what one can think. This suggests that programming languages should cater to the natural problem solving approaches used by humans. Miller[21] observes that people can keep track of about seven things. This suggests that a programming language should provide mechanisms which support abstraction and generalization. Programming languages should approach the level at which humans reason and should reflect the notational approaches that humans use in problem solving and further must include ways of structuring programs to ease the tasks of program understanding, debugging and maintenance.

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 linear function y := 2*x + 3 example might be written in assembly language as:

```
Load X R1
Mult R1 2 R1
Add R1 3 R1
Store R1 Y
```

This example indicates that machine languages tend to be difficult for humans to read and write.

1.4 Language Design Principles

Programming languages are largely determined by the importance the language designers attach to the areas of readability, writeability 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. As hardware and compiler technology evolves, there is a corresponding evolution toward more efficient implementation and execution. As larger programs are written and new applications are developed, it is the area of readability and writability that must receive the most emphasis. It is this concern for readability and writability that is driving the development of programming languages.

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*.

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 *writability* for its human users and efficient execution on the available hardware.

Readability and writeability are facilitated by the following principles.

Principle of Simplicity: The language should be based upon as few "basic concepts" as possible.

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.

1.5 Further Reading

For a programming languages text which presents programming languages from the virtual machine point of view see Pratt[24]. For a programming languages text which presents programming languages from the point of view of denotational semantics see Tennent[30]. For a programming languages text which presents programming languages from a programming methodology point of view see Hehner[11].

1.6 Exercises

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

2. 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.

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

4. 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.

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

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

7. 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.

8. Compare two programming languages from the same compuational paradigm with respect to the programming language design principles.

Chapter 2

Syntax

The syntax of a programming language describes the structure of programs.

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 appearance and structure of programs. The syntactic elements of a programming language are largely 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. The 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.

This chapter provides an introduction to grammars and shows how they may be used for the description of the syntax of programming languages. 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. Both concrete and abstract grammars are presented.

Preliminary definitions

The following definitions are basic to the definition of regular expressions and context-free grammars.

Definition 2.1 *An* alphabet *is a nonempty, finite set of symbols.*

The alphabet for the lexical tokens of programming language is the character set. The alphabet for the context-free structure of a programming language is the set of keywords, identifiers, constants and delimiters; the lexical tokens.

Definition 2.2 *A* language L *over an alphabet* Σ *is a collection of finite strings of elements of* Σ

Definition 2.3 *Let* L_0 *and* L_1 *be languages.* 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.

Definition 2.4 *Let* Σ *be an alphabet. The set of all possible finite strings of elements of* Σ *is denoted by* Σ^*. *The set of all possible nonempty strings of* Σ *is denoted by* Σ^+.

2.1 Context-Free Grammars

The structure of programming languages is described using context-free grammars. Context-free grammars describe how lexical units (tokens) are grouped into meaningful structures. The alphabet (the set of lexical units) consists of the keywords, punctuation symbols, and various operators. Context-free grammars are sufficient to describe most programming language constructs but they cannot describe the context sensitive aspects of a programming language. For example, context-free languages cannot be used to specify that a name must be declared before reference and that the order and number of actual parameters in a procedure call must match the order and number of formal arguments in a procedure declaration.

Concrete Syntax

Definition 2.5 *A* context-free grammar *is a quadruple (N,T,P,S) where N is an alphabet of nonterminals, T is an alphabet of terminals disjoint from N, and P is a finite set of rewriting rules (productions) of the form*

$N = \{E\}$
$T = \{id, +, *, (,)\}$
$P = \{E \rightarrow id, E \rightarrow (E), E \rightarrow E + E, E \rightarrow E * E\}$
$S = E$

Figure 2.1: An expression grammar

E	$[id + id * id]$
$E + E$	$[id] + [id * id]$
$id + E$	$id + [id * id]$
$id + E * E$	$id + [id] * [id]$
$id + id * E$	$id + id * [id]$
$id + id * id$	$id + id * id$

Figure 2.2: Derivation and recognition of $id + id * id$

$$A \rightarrow w \text{ where } A \in N,\ w \in (N \cup T)^*, \text{ and } S \in N.$$

Figure 2.1 contains an example of a context-free grammar for arithmetic expressions. Notice that expressions are defined recursively.

Grammars may be used both for recognition and generation of strings. Recognition and generation requires finding a rewriting sequence consisting of applications of the rewriting rules which begins with the grammar's start symbol and ends with the string in question. For example, a rewriting sequence for the string

$$id + id * id$$

is given in Figure 2.2. The first column is the rewriting sequence which generates the string. The same rewriting sequence is used to recognize the string in the second column. The bracketed portions of the string indicate the unrecognized portions of the string and correspond to nonterminals in the first column.

The recognition of a program in terms of the grammar is called **parsing**. An algorithm which recognizes programs is called a **parser**.

Ambiguity

Figure 2.3 presents an alternative sequence of applications of the rewriting

E	$[id + id * id]$
$E * E$	$[id + id] * [id]$
$E + E * E$	$[id] + [id] * [id]$
$id + E * E$	$id + [id] * [id]$
$id + id * E$	$id + id * [id]$
$id + id * id$	$id + id * id$

Figure 2.3: A second derivation and recognition of $id + id * id$

rules for the derivation of the string $id + id * id$. Note that the derivation in Figure 2.3 suggests that addition is performed before multiplication in violation of standard arithmetic practice. In these examples we have chosen to rewrite the left-most non-terminal first. When there are two or more left-most derivations of a string in a given grammar, the grammar is said to be **ambiguous**. In some instances ambiguity may be eliminated by the selection of another grammar for the language or by establishing precedences among the operators.

Parsers

Context-free grammars are equivalent to *push-down automata*. A push-down automaton is a machine with a stack, an input string, and a finite control. The input string is consumed as it is read so that the machine cannot reread an earlier portion of the input. At each step of the machine, the action is determined by the next symbol in the input, the top symbol on the stack and the state of the finite control. An action of the machine consists of one or more of the following: consume the next symbol of input, push a symbol onto the stack, pop the top of the stack, change the state of its finite control.

Parsers for programming languages are examples of push-down automata. Parsers are usually built in one of two ways. Either, the parser begins with the start symbol of the grammar and uses the productions to generate a string matching the input string or the parser tries to match the input with the right hand side of a production and when a match is found, it replaces the portion of the matched portion of the input with the left hand side of the production. The first kind of parser is called a *top down parser* while the second is a called a *bottom up parser*. To illustrate, consider parsing the string

$$id + id * id$$

using a top down parser and a bottom up parser for the expression grammar of Figure 2.1. Figure 2.4 shows the state of the stack and input string during the top down parse. Parsing begins with the start symbol on top of the stack.

Stack	Input
E]	id+id*id]
E+E]	id+id*id]
id+E]	id+id*id]
+E]	+id*id]
E]	id*id]
E*E]	id*id]
id*E]	id*id]
E]	*id]
E]	id]
id]	id]
]	]

Figure 2.4: Top down parsing

Since the top of the stack does not match the input, it is popped and the right hand side of the production $E \rightarrow E + E$ is pushed onto the stack. Still a match is not found so the parser pops the top of the stack and pushes the right hand side of the production $E \rightarrow id$ onto the stack. Now the top of the stack matches the input symbol so the top of the stack is popped and the input symbol is consumed. The parser continues until both the stack and the input are empty in which case the input is *accepted* or an error state occurs. The finite control 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 non-terminal 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.

Figure 2.5 shows the state of the stack and input string of a bottom up parser for the expression grammar of Figure 2.1. Parsing begins with an empty stack. Since there are no symbols in the stack, the first symbol of the input is *shifted* to the stack. The top of the stack then corresponds to the right hand side of the production $E \rightarrow id$ so the stack is *reduced* to the left hand side of the production by popping the stack and pushing the symbol E onto the stack. The

Stack	Input
]	id+id*id]
id]	+id*id]
E]	+id*id]
+E]	id*id]
id+E]	*id]
E+E]	*id]
E+E]	id]
id*E+E]	]
E*E+E]	]
E+E	]
E]	]

Figure 2.5: Bottom-up parsing

parse continues by a series of shifts and reductions until the start symbol is the only symbol left on the stack and the input is empty in which case the input is accepted or an error state occurs. The finite control 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.

In both these examples the choice of the which production to use appears to be magical. In the case of a top down parser the grammar should be rewritten to remove the ambiguity. Exercise 8 contains an appropriate grammar for a top down parser for expressions. 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.

```
<Expression> ::= <Identifier> | <Number> |
                 <Expression> <Op> <Expression> |
                 ( <Expression> )
<Op> ::= + | − | ∗ | /
<Identifier> ::= <Letter>
<Identifier> ::= <Identifier> <Letter>
<Number> ::= <Digit>
<Number> ::= <Number> <Digit>
<Letter> ::= A | ...  | Z
<Digit> ::= 0 | ...  | 9
```

Figure 2.6: BNF grammar for arithmetic expressions

Backus-Naur Form

The BNF is a notation for describing the productions of a context-free grammar. The BNF uses the following symbols $<$, $>$, ::=, |. Nonterminals are enclosed between $<$ and $>$. The symbol $\rightarrow$ 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. Figure 2.6 gives a BNF description of arithmetic expressions. The readability of a context-free grammar may be improved by the following extensions to the BNF.

- *Typeface:* The names of BNF categories are written in italics and without $<$ and $>$.
- *Zero or More:* Zero or more repetitions of the previous syntactical unit are indicated with ellipsis (...).
- *Optional:* Optional elements are enclosed between the brackets [and].

The previous grammar may be rewritten and the definition of identifiers improved using these new conventions. The result is in Figure 2.7. As another example, the context-free grammar for a simple imperative programming language called **Simp** is found in Figure 2.8.

Abstract Syntax

The previous syntactic definitions are examples of *concrete syntax.* Concrete syntactical descriptions are appropriate for the automatic generation of parsers for the language. A more abstract description is appropriate for formal language descriptions.

```
Expression ::= Identifier | Number |
               Expression Op Expression |
               (Expression)
Op ::= + | − | ∗ | /
Identifier ::= Letter [ Letter | Digit ]...
Number ::= Digit...
```

Figure 2.7: An EBNF grammar for expressions

```
program ::= command_sequence

command_sequence ::= ϵ | command_sequence command ;

command := SKIP
         | IDENT := exp
         | IF boo_exp THEN command_sequence
                      ELSE command_sequence FI
         | WHILE bool_exp DO command_sequence END

exp ::= exp + term | exp - term | term
term ::  term * factor | term / factor | factor
factor ::= factor↑primary | primary
primary ::= INT | IDENT | ( exp )
bool_exp ::= exp = exp | exp < exp | exp > exp
```

Figure 2.8: An EBNF grammar for Simp

```
program ::= command...

command ::= SKIP | assignment | conditional | while

assignment ::= IDENT exp
conditional ::= exp then_branch else_branch
while ::= exp body

then_branch ::= command...
else_branch ::= command...
body ::= command...

exp ::= INT | IDENT | exp OP exp
```

Figure 2.9: An abstract grammar for Simp

Definition 2.6 *An* abstract syntax *for a language consists of a set of* syntactic domains *and a set of BNF rules describing the abstract structure.*

The idea is to use only as much concrete notation as is necessary to convey the structure of the objects under description. An abstract syntax for Simp is given in Figure 2.9. A fully abstract syntax simply gives the components of each language construct, leaving out the representation details. In the remainder of the text we will use the term *abstract syntax* whenever some syntactic details are left out.

2.2 Regular Expressions

The lexical units (tokens) of programming languages are defined using regular expressions. Regular expression describe how characters are grouped to form tokens. The alphabet consists of the character set chosen for the language.

Definition 2.7 *Let Σ be an alphabet. The regular expressions over Σ and the languages (sets of strings) that they denote are defined as follows:*

1. *$\emptyset$ is a regular expression and denotes the empty set. This language contains no strings.*
2. *ϵ is a regular expression and denotes the set $\{\epsilon\}$. This language contains one string, the empty string.*

	$\mathcal{A}$	$\mathcal{D}$
1	2	
2	2	2

Figure 2.10: Finite state machine transitions for identifiers

3. *For each a in* Σ **a** *is a regular expression and denotes the set* $\{a\}$. *This language contains one string, the expression.*

4. *If r and s are regular expressions denoting the languages R and S, respectively, then* $(r + s)$, (rs), *and* (r^*) *are regular expressions that denote the sets* $R \cup S$, RS, *and* R^*, *respectively.*

Identifiers and numbers are usually defined using regular expressions. If $\mathcal{A}$ represents any letter and $\mathcal{D}$ represents any digit, then identifiers and real numbers may be defined using regular expressions as follows:

identifier = $\mathcal{A}\ (\mathcal{A} + \mathcal{D}\ )^*$

real = $(\ \text{'+'} + \text{'-'} + \epsilon\ )\ \mathcal{D}^+\ (\ \epsilon + (.\ \mathcal{D}^+\ ))\ (\epsilon + \text{E}\ (\ \text{'+'} + \text{'-'} + \epsilon\ )\ \mathcal{D}^+)$

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.

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 transition. The transitions for a finite state machine which recognizes identifiers given in Figure 2.10. The start state is 1 and the accepting state is 2. In the start state if the input is an alphabetic character, then it is consumed and the transition to state 2 occurs. The machine remains in state 2 as long as the input consists of either alphabetic characters or digits.

2.3 Attribute Grammars and Static Semantics

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...
B ::= C ...
C ::= V := E | ...
```

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 2.3 is an attribute grammar for declarations. 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.

P ::= D(Env$\uparrow$) B(Env$\downarrow$)
D(Env$\uparrow$) ::= ...V_i(Env_{i-1} $\downarrow$,Env_i $\uparrow$)...
where $Env_0 = \emptyset$, $Env = Env_n$ and
$Env_i = Env_{i-1} \cup \{V_i\}$
B(Env$\downarrow$) ::= C(Env$\downarrow$)...
C(Env$\downarrow$) ::= V := E(Env$\downarrow$) | ...
where $V \in Env$

Figure 2.11: An attribute grammar for declarations

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

2.4 Further Reading

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[14]. The original paper on attribute grammars was by Knuth[15]. For a more recent source and their use in compiler construction and compiler generators see [8, 23]

2.5 Exercises

1. Construct a scanner for arithmetic expressions.
2. Lex: scanner
3. Ambiguity: if then else,
4. Ambiguity: arithmetic expressions
5. CFG for ??
6. Abstract Grammar for Pascal excluding abbreviations such as multidimensional arrays, `label` and `forward` declarations and `packed` types.
7. Construct a recursive descent parser for the simple imperative programming language of this chapter.
8. Given a context-free grammar, a parser (or recognizer) for the corresponding language may be written by providing a set of procedures; one procedure for each non-terminal in the grammar. Such a set of procedures constitutes a *recursive descent parser*. The grammar for the simple imperative programming language is not suitable for a recursive descent parser. Why? Construct a calculator (using recursive descent) using the following grammar for expressions.

   ```
   exp ::= term exp'
   exp' ::= + term exp' | - term exp' | ϵ
   term ::= factor term'
   term' ::= * factor term' | / factor term' | ϵ
   factor ::= primary factor'
   factor' ::= ^ primary factor' | ϵ
   primary ::= INT | IDENT | ( exp )
   ```

9. BNF
10. Yacc: parser
11. Context sensitivity
12. Attribute grammar: calculator
13. Construct and interpreter for BASIC.

Chapter 3

Semantics

The semantics of a programming language describe the relationship between the syntactical elements 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.

$$semantics : syntax \rightarrow computational\ model$$

This approach is called *syntax-directed semantics.*

There are four widely used techniques (algebraic, axiomatic, denotational, and operational) for the description of the semantics of programming languages. Algebraic semantics describe the meaning of a program by defining an algebra which defines algebraic relationships that exist among the language's syntactic elements. The relationships are described by axioms. Axiomatic semantics method does not give the meaning of the program explicitly. Instead, *proper-*

ties about language constructs are defined. The properties are expressed with axioms and inference rules. A property about a program is deduced by using the axioms and inference rules. 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. 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.

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.

3.1 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.

$$Algebra = < set\ of\ values;\ operations >$$

Figure 3.1 is an example of an algebraic definition. It is an algebraic definition of a fragment of Peano arithmetic. The semantic equations define equivalences between syntactic elements. The equations 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 semantic function sections constitute the *signature* of the algebra. Functions with zero, one, and two operands are referred to as nullary, unary, and binary operations. Often abstract data types require values from several different sorts. Such a type is modeled using a *many-sorted* algebra. The signature of such an algebra is a set of sorts and a set of functions taking arguments and returning values of different sorts. For example, a stack may be modeled as a many-sorted algebra with three sorts and four operations. An algebraic definition of a stack is found in figure 3.2.

The stack example is more abstract than the previous one because the results of the operations are not described. This is necessary because the syntactic structure of the natural numbers and lists are not specified. To be more specific

Domain:

N ∈ Nat (the natural numbers
N ::= 0 | S(N)

Semantic functions:

$+ : \text{Nat} \to \text{Nat} \to \text{Nat}$
$\times : \text{Nat} \to \text{Nat} \to \text{Nat}$

Semantic equations:

$$\begin{array}{lcl} (n+0) & = & n \\ (m+\mathrm{S}(n)) & = & \mathrm{S}(m+n) \\ (n\times 0) & = & 0 \\ (m\times \mathrm{S}(n)) & = & ((m\times n)+m) \end{array}$$

where $m, n \in \text{Nat}$

Figure 3.1: Peano Arithmetic

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. Thus, the operations are defined using semantic axioms instead of semantic equations. The axioms describe the properties of the operations. The axioms impose constraints on the stack operations that are *sound* in the sense that they are consistent with the actual behavior of stacks reguardless of the implementation. Finding axioms that are *complete* in the sense that they completely specify stack behavior 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.

3.2 Axiomatic Semantics

The *axiomatic semantics* of a programming language define properties about language constructs. The axiomatic semantics of a programming language define a *mathematical theory* of programs written in the language.

Domains:

Nat (the natural numbers
Stack (of natural numbers)
Bool = {true, false} (boolean values)

Semantic functions:

$create : () \to \text{Stack}$
$push : \text{Nat} \to \text{Stack} \to \text{Stack}$
$pop : \text{Stack} \to \text{Stack}$
$top : \text{Stack} \to \text{Nat}$
$empty : \text{Stack} \to \text{Bool}$

Semantic axioms:

$$\begin{array}{lcl} pop(push(N,S)) & = & S \\ top(push(N,S)) & = & N \\ empty(push(N,S)) & = & false \\ empty(create()) & = & true \end{array}$$

where $N \in \text{Nat}$ and $S \in \text{Stack}$.

Figure 3.2: A Stack

A mathematical theory has three components.

- **Syntactic rules:** These determine the structure of formulas which are the statements of interest.
- **Axioms:** These are basic theorems which are accepted without proof.
- **Inference rules:** These are the mechanisms for deducing new theorems from axioms and previously proved theorems.

Formulas are triples of the form:

$$\{P\}\ \mathbf{c}\ \{Q\}$$

where **c** is a command in the programming language, P and Q are *assertions* or statements concerning the properties of program objects, 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 are used to summarize the progress of the computation.

> The meaning of
>
> $$\{P\}\ \mathbf{c}\ \{Q\}$$
>
> is that if **c** is executed in a state in which assertion P is satisfied and **c** terminates, then it terminates in a state in which assertion Q is satisfied.

Figure 3.3 is an example of the use of axiomatic semantics in the verification of programs. 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. Lines 1 and 11 are the pre- and post-conditions respectively for the program. The pre-condition asserts that the number of elements in the array is greater than 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+1st 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 on lines 2 and 7

1.		$\{\ 0 = \sum_{i=1}^{0} A[i],\ 0 < n = \lvert A \rvert\ \}$
2.	`S,I := 0,0`	
3.		$\{\ \mathtt{S} = \sum_{i=1}^{I} A[i],\ I \leq n\ \}$
4.	`while I < n do`	
5.		$\{\ \mathtt{S} = \sum_{i=1}^{I} A[i],\ I < n\ \}$
6.		$\{\ \mathtt{S}{+}A[I{+}1] = \sum_{i=1}^{I+1} A[i],\ I{+}1 \leq n\ \}$
7.	`    S,I := S+A[I+1],I+1`	
8.		$\{\ S = \sum_{i=1}^{I} A[i],\ I \leq n\ \}$
9.	`end`	
10.		$\{\ \mathtt{S} = \sum_{i=1}^{I} A[i],\ I \leq n,\ I \geq n\ \}$
11.		$\{\ \mathtt{S} = \sum_{i=1}^{n} A[i]\ \}$

Figure 3.3: Verification of summation

we must use the following

Assignment Axiom:

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

This axiom is read as follows:

> 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 satisfied 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.).

Looking at lines 1, 2 and 3 and also at lines 6, 7 and 8, we can see that this axiom is satisfied in both cases.

To verify the while command of lines 3 through 10, we must use the following

Loop Axiom:

$$\frac{\{I \wedge B \wedge V > 0\}\ C\ \{I \wedge V > V' \geq 0\}}{\{I\}\ \text{while}\ B\ \text{do}\ C\ \text{end}\ \{I \wedge \neg B\}}$$

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 is read as follows:

> 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 3, 5 and 8 satisfy the condition for the application of the axiom. While lines 3 and 10 satisfy the 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 6 and 7 and lines 10 and 11 are justified by the following

Rule of Consequence:

$$\frac{P \rightarrow Q,\ \{Q\}\ C\ \{R\},\ R \rightarrow S}{\{P\}\ C\ \{S\}}$$

The justification for the composition the assignment command in line 2 and the while command in line 4 requires the following axiom.

Sequential Composition Axiom:

$$\frac{\{P\}\ C_0\ \{Q\},\ \{Q\}\ C_1\ \{R\}}{\{P\}\ C_0; C_1\ \{R\}}$$

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 provide a logically complete deductive system.

Selection Axiom:

$$\frac{\{P \wedge B\}\ C_0\ \{Q\},\ \{P \wedge \neg B\}\ C_1\ \{Q\}}{\{P\}\ \text{if } B \text{ then } C_0 \text{ else } C_1 \text{ fi}\ \{Q\}}$$

Conjunction Axiom:

$$\frac{\{P\}\ C\ \{Q\},\ \{P'\}\ C\ \{Q'\}}{\{P \wedge P'\}\ C\ \{Q \wedge Q'\}}$$

Disjunction Axiom:

$$\frac{\{P\}\ C\ \{Q\},\ \{P'\}\ C\ \{Q'\}}{\{P \vee P'\}\ C\ \{Q \vee Q'\}}$$

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 the favored method by software engineers 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

```
1.  S,I := 0,0
2.  loop:  if I < n then S,I := S+A[I+1],I+1; loop
3.                  else skip
```

Figure 3.4: Recursive version of summation

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 requires a loop variant and a loop invariant. In the summation example, a loop variant is readily appearent 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 3.4. 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 $\{\ 0 \leq r < d,\ (a = q \times d + r)\ \}$ by *deleting a conjunct.* In this case the invariant is $\{\ 0 \leq r,\ (a = q \times d + r)\ \}$ 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:

$$\{\forall i.(0 < i < n \rightarrow A[i] \leq A[i+1]), s = perm(A)\}$$

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

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

To develop an invariant in these cases requires that the assertion be strengthened

Abstract Syntax:

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

Semantic Algebra:

N = **Nat** (the natural numbers (*0, 1, ...*)
$+ \in \mathbf{N} \to \mathbf{N} \to \mathbf{N}$

Valuation Function:

$\mathcal{D} \in \text{Nat} \to \mathbf{Nat}$

$$
\begin{array}{lcl}
\mathcal{D}[\![(n+0)]\!] & = & \mathcal{D}[\![n]\!] \\
\mathcal{D}[\![(m+S(n))]\!] & = & \mathcal{D}[\![(m+n)]\!] + \mathit{1} \\
\mathcal{D}[\![(n \times 0)]\!] & = & \mathit{0} \\
\mathcal{D}[\![(m \times S(n))]\!] & = & \mathcal{D}[\![((m \times n)+m)]\!]
\end{array}
$$

where m,n ∈ Nat

Figure 3.5: Peano Arithmetic

by adding additional constraints. The additional constraints make assertions about different parts of the array.

3.3 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.

Figure 3.5 is an example of a denotational definition. 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 seem 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 3.6 is denotational definition of the small imperative programming language Simp encountered in the previous chapter.

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.

3.4 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 3.7 an example of an operational definition. It is 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 3.8. The operational semantics are defined by using two semantic functions, $\mathcal{I}$ which interprets commands and ν 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** com-

Abstract Syntax:

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

C ::= V := E | if E then C_1 else C_2 end | while E do C_3 end | $C_1; C_2$ | skip
E ::= V | N | E_1 O E_2 | (E)
O ::= $+ | - | * | / | = | < | > | <>$

Semantic Algebra:

	Domains:
$\tau \in \mathbf{T} = \{\textit{true, false}\}$	boolean values
$\zeta \in \mathbf{Z} = \{\textit{...-1,0,1,...}\}$	integers
$\sigma \in \mathbf{S} =$ Variable → Numeral	state

Valuation Functions:

$\mathcal{C} \in \mathbf{C} \to (\mathbf{S} \to \mathbf{S})$
$\mathcal{E} \in \mathbf{E} \to \mathrm{E} \to (\mathbf{N} \cup \mathbf{T})$

$\mathcal{C}[\![\text{skip}]\!]\sigma = \sigma$
$\mathcal{C}[\![V := E]\!]\sigma = \sigma[V : \mathcal{E}[\![E]\!]\sigma]$
$\mathcal{C}[\![C_1; C_2]\!]\sigma = \mathcal{C}[\![C_2]\!]\mathcal{C}[\![C_1]\!]\sigma$
$\mathcal{C}[\![\text{if } E \text{ then } \mathrm{C}_1 \text{ else } \mathrm{C}_2 \text{ end}]\!]\sigma = \begin{cases} \mathcal{C}[\![C_1]\!]\sigma & \text{if } \mathcal{E}[\![E]\!]\sigma = \text{true} \\ \mathcal{C}[\![C_2]\!]\sigma & \text{if } \mathcal{E}[\![E]\!]\sigma = \text{false} \end{cases}$
$\mathcal{C}[\![\text{while } E \text{ do } C \text{ end}]\!]\sigma = \lim_{n\to\infty} \mathcal{C}[\![(\text{if E then C else skip end})^n]\!]\sigma$
$\mathcal{E}[\![V]\!]\sigma = \sigma(V)$
$\mathcal{E}[\![N]\!] = N$
$\mathcal{E}[\![E_1 + E_2]\!] = \mathcal{E}[\![E]\!]\sigma + \mathcal{E}[\![E]\!]\sigma$
...
$\mathcal{E}[\![E_1 = E_2]\!]\sigma = \mathcal{E}[\![E]\!]\sigma = \mathcal{E}[\![E]\!]\sigma$
...

Figure 3.6: Denotational semantics for Simp

Abstract Syntax:

N $\in$ Nat (the natural numbers)

N ::= 0 | S(N) | (N + N) | (N $\times$ N)

Interpreter:

$\mathcal{I}$: N $\rightarrow$ N

$$
\begin{array}{lcl}
\mathcal{I}[\![(n+0)]\!] & \Rightarrow & n \\
\mathcal{I}[\![(m+\mathrm{S}(n))]\!] & \Rightarrow & \mathrm{S}(\mathcal{I}[\![(m+n)]\!]) \\
\mathcal{I}[\![(n\times 0)]\!] & \Rightarrow & 0 \\
\mathcal{I}[\![(m\times \mathrm{S}(n))]\!] & \Rightarrow & \mathcal{I}[\![((m\times n)+m)]\!]
\end{array}
$$

where $m, n \in$ Nat

Figure 3.7: Operational semantics for Peano arithmetic

mand 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.

3.5 Further Reading

Algebraic semantics

Gries [10] and Hehner [11] are excellent introductions to axiomatic semantics as applied to program construction. For a description of denotational semantics see Schmidt [26] or Scott [28]

Operational semantics??

Interpreter:

$\mathcal{I} : \mathbf{C} \times \Sigma \rightarrow \Sigma$
$\nu \in \mathbf{E} \times \Sigma \rightarrow \mathbf{T} \cup \mathbf{Z}$

Semantic Equations:

$\mathcal{I}(\text{skip},\sigma) = \sigma$
$\mathcal{I}(\text{V} := \text{E},\sigma) = \sigma[\text{V}:\nu(\text{E},\sigma)]$
$\mathcal{I}(\text{C}_1\ ;\text{C}_2,\sigma) = \mathcal{E}(\text{C}_2,\mathcal{E}(\text{C}_1,\sigma))$
$\mathcal{I}(\text{if E then C}_1 \text{ else C}_2 \text{ end},\sigma) = \begin{cases} \mathcal{I}(\text{C}_1,\sigma) & \text{if } \nu(\text{E},\sigma) = \text{true} \\ \mathcal{I}(\text{C}_2,\sigma) & \text{if } \nu(\text{E},\sigma) = \text{false} \end{cases}$
while E do C end = if E then C;while E do C end else skip
$\nu(\text{V},\sigma) = \sigma(\text{V})$
$\nu(\text{N},\sigma) = \text{N}$
$\nu(\text{E}_1\text{+E}_2,\sigma) = \nu(\text{E}_1,\sigma) + \nu(\text{E}_2,\sigma)$
...
$\nu(\text{E}_1\text{=E}_2,\sigma) = \begin{cases} \text{true} & \text{if } \nu(\text{E},\sigma) = \nu(\text{E},\sigma) \\ \text{false} & \text{if } \nu(\text{E},\sigma) \neq \nu(\text{E},\sigma) \\ \text{otherwise} & \end{cases}$
...

Figure 3.8: Operational semantics for Simp

Chapter 4

Abstraction and Generalization I

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.

Keywords and phrases: Name, binding, abstract, definition, declaration, variables, parameters, arguments, formals, actuals.

The ability to abstract and to generalize is an essential part of any intellectual activity. Abstraction and generalization are the basis for 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

Abstraction	$name : abstract$
Invocation	$name$
Substitution	$E[x : a]$
Generalization	$\lambda x.E$
Specialization	$(\lambda x.E\ a) = E[x : a]$
Abstraction and generalization	$name(x) : E$
Invocation and specialization	$name(a)$

Figure 4.1: Abstraction and Generalization

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, parameterization, generics and polymorphism. Generalization is essential in the construction of programs. It places the emphasis on the similarities between objects. Thus, it helps to manage complexity by collecting individuals into groups and providing a representative which can be used to specify any individual of the group.

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 4.1 summarizes the notation which will be used for abstraction and generalization.

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.

4.1 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

$$name \ : \ abstract$$

to denote the *binding* of a name to an abstract.

Abstraction permits the recognition and elimination of common subexpressions as in the following example:

```
(x+y-z)*(x+y-z) ⟹ a*a where a = x+y-z
```

This example motivates the use of functions and procedures in programming languages.

In addition to naming and substitution there is a third 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. □

Binding

The concept of binding is common to all programming languages. A binding is an association of two entities. The objects which may be bound to identifiers 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.

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.

Terminology: We could equally well say *identifier* instead of name as in some other texts. Among the various terms for abstracts found in other texts are *module*, *package*, *library*, *unit*, *subprogram*, *subroutine*, *routine*, *function*, *procedure*, *abstract type*, *object*. □

4.2 Generalization

Principle of Generalization: A *generic* is an entity which may be elaborated upon invocation.

Generalization permits the use of a single pattern to represent each member of a group. We use the notation:

$$\lambda x.B'$$

to denote the generalization of the object B where x is called a *parameter* and B' is the object B with x replacing any number of occurences of some part of B by x. The parameter x is said to be *bound* in the expression but *free* in B' and the *scope* of x is said to be B'.

The symbol λ (lambda) is a *quantifier*. Quantifiers introduce variables into objects.

Aside: The symbol λ was introduced by Church for variable introduction in the lambda calculus. It roughly corrresponds to the symbol $\forall$, the universal quantifier, of first-order logic. The appendix contains a brief introduction to first-order logic. □

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

$$p(x) : B$$

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

$$(p\ a) \text{ or } p(a)$$

where p is the name and a is called the *argument* whose value (called the *argument*) is substituted for the parameter. Upon invocation of the abstract, the argument is bound to the parameter.

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:

```
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 *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 with reference to both the definition of `c` and to the environment (`pi` is viewed as a constant). Therefore, this program is not "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 restrctions on parameters or the parts of an entity that may be parameterized. Neither should programming lanauges. 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* (*actuals*) are sometimes used instead of the terms parameters and arguments respectively. □

4.3 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

$$E[x : a]$$

to denote the substitution of a for x in E. The notation is read as "$E[x : a]$ is the expression obtained from E by replacing all occurrences of x with a."

Terminology: The notation for substitution was chosen to emphasize the relationship between abstraction and substitution. Other texts use the notation $E[a/x]$ for substitution. That notation is motivated by the cancelation that occurs when a number is multiplied by its inverse ($x(a/x) = a$). □

4.4 Abstraction and Generalization

Together, abstraction and generalization provide a powerful mechanism 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.

$$
\begin{aligned}
\texttt{(a+b-c)*(x+y-z)} &\Longrightarrow (\lambda \text{ i j k. i+j-k) a b c * } (\lambda \text{ i j k. i+j-k) x y z} \\
&\Longrightarrow \text{f a b c * f x y z where f i j k = i+j-k}
\end{aligned}
$$

4.5 Exercises

1. Extend the compiler to handle constant, type, variable, function and procedure definitions and references to the same.
2. What is the effect of providing lazy evaluation in an imperative programming language?
3. Extend the compiler to handle parameterization of functions and procedures.
4. 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?

Chapter 5

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: *No. A type is a way to classify values by their properties and behavior.*

Advanced Programmer: *No. A type is a set of values and operations.*

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.

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.

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 with in bounds of time and space. □

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. □

5.1 Primitive Domains

Among the primitive types provided by programming languages are

> Truth-value = { *false, true*}
> Integer = {..., -2, -1, 0, +1, +2, ...}
> Real = {..., -1.0, ..., 0.0, ..., +1.0, ...}
> Character = {..., a, b, ..., z, ...}

The values are represented as a particular pattern of bits in the storage of the computer.

Aside: Programming language definitions do not place restrictions on the primitive types. However hardware limitations and variation have considerable influence on actual programs so that, Integer is an implementation defined range of whole numbers, Real is an implementation defined subset of the rational numbers and Character is an implementation defined set of characters. □

Several languages permit the user to define additional primitive types. These primitive types are called *enumeration* types.

5.2 Compound Domains

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

Assembly operation: $(a_0,...,a_n) \in D_0 \times ... \times D_n$ where $a_i \in D_i$ and
$D_0 \times ... \times D_n = \{(a_0, ..., a_n) \mid a_i \in D_i\}$

Disassembly operation: $(a_0,...,a_n)\downarrow i = a_i$ for $0 \leq i \leq n$

Figure 5.1: Product Domain: $D_0 \times ... \times D_n$

- 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.

$$\mathit{fst}(a, b) = a$$
$$\mathit{snd}(a, b) = b$$

The product domain is easily generalized (see Figure 5.1 to permit the product of an arbitrary number of domains.

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. □

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 the 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 $(\mathtt{A}, a)$ and $(\mathtt{B}, b)$ are elements of $A + B$. That is,
>
> $$A + B = \{(\mathtt{A}, a) \mid a \in A\} \cup \{(\mathtt{B}, b) \mid b \in B\}$$
>
> 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.

> $\mathtt{A}(\mathtt{A}, a) = a$

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 5.2) to sums of an arbitrary number of domains.

Terminology: The *sum* domain is also called the *disjoint union* or *co-product* domains. □

Assembly operations: $(\mathcal{D}_i, \mathrm{d}_i) \in \mathrm{D}_0 + ...+\mathrm{D}_n$ and
$\mathrm{D}_0 + ...+\mathrm{D}_n = \cup_{i=0}^{n}\{(\mathcal{D}_i, d) \mid d \in \mathrm{D}_i\}$

Disassembly operations: $\mathcal{D}_i(\mathcal{D}_i, \mathrm{d}_i) = \mathrm{d}_i$

Figure 5.2: Sum Domain: $\mathrm{D}_0 + ...+\mathrm{D}_n$

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

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

Figure 5.3: Function Domain: $\mathrm{A} \rightarrow \mathrm{B}$

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 differs from 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:

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

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 5.3.

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:

$$m\ :\ S \to T$$

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.

$$+ : number \times number \to number$$

The functional programming paradigm is based on programming with maps.

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

Power Domain

Set theory provides an elegant notation for the description of computation. However, it is difficult to provide efficient implementation of the 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.

operations??

The set of all subsets of a set is the power set and is defined:

$$\mathcal{P}^S = \{\ s \mid s \subseteq S\ \}$$

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 $Nat \times Nat$.

$$\text{sqr}\ = \{(0,0),(1,1),(2,4),(3,9),...\}$$

Assembly operations: $\emptyset \in \mathcal{P}^{\mathrm{D}}$, $\{a\} \in \mathcal{P}^{\mathrm{D}}$ for a in D, and
$S_i \cup S_j \in \mathcal{P}^{\mathrm{D}}$ for S_i, $S_j \in \mathcal{P}^{\mathrm{D}}$

Figure 5.4: Power Domain: $\mathcal{P}^{\mathrm{D}}$

Generalization helps to simplify this infinite list to:

$$\text{sqr} \ = \{(x, x * x) \mid x \in \ \text{Nat}\}$$

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 subranges are another since, enumeration is tedious for large subdomains 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 $\mathcal{P}()$ creates the domain $\mathcal{P}(\mathbf{A})$, a collection whose members are subsets of **A**.

Recursively Defined Domain

Recursively defined domains are domains whose definition is of the form:

$$D : ...D...$$

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.

$$\begin{array}{lll} D_0 & = & \text{null} \\ D_{i+1} & = & e[D:D_i] \;\; \text{for } i = 0... \\ D & = & \lim_{i\to\infty} D_i \end{array}$$

Figure 5.5: Limit construction

The least solution of a recursively defined domain is obtained through a sequence of approximations (D_0, D_1,...) 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 of Figure 5.5 with three examples.

The Natural Numbers

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

N ::= 0 | S(N)

The defining sequence for the natural numbers is:

$$\begin{array}{ll} N_0 & = Null \\ N_{i+1} & = 0 \mid S(N_i) \;\; \text{for } i = 0... \end{array}$$

The definition results in the following:

$$\begin{array}{ll} N_0 & = Null \\ N_1 & = 0 \\ N_2 & = 0 \mid S(0) \\ N_3 & = 0 \mid S(0) \mid S(S(0)) \\ N_4 & = 0 \mid S(0) | S(S(0)) \mid S(S(S(0))) \\ ... & \end{array}$$

The factorial function

For functions, Null can be replaced with $\perp$ 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

$$\text{fac}_0(n) = \bot$$

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.

$$\begin{array}{ll} \text{fac}_0 & = \{\} \\ \text{fac}_1 & = \{(0,1)\} \\ \text{fac}_2 & = \{(0,1),(1,1)\} \\ \text{fac}_3 & = \{(0,1),(1,1),(2,2)\} \\ \text{fac}_4 & = \{(0,1),(1,1),(2,2),(3,6)\} \\ \ldots & \end{array}$$

Note that each function in the sequence includes the previously defined function and the sequence suggests that

$$\text{fac} = \lim_{i \to \infty} \text{fac}_i$$

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 **false**. A recursive definition of the ancestor relation is:

$$\text{ancestor}(A,D),\ \text{if} \begin{cases} \text{parent}(A,D) & \text{or} \\ \text{parent}(A,I) & \&\quad \text{ancestor}(I,D) \end{cases}$$

The ancestor relation is approximated by a sequence of relations:

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

And the relation ancestor_i is defined as

$$\text{ancestor}_{i+1}(A,D),\ \text{if} \begin{cases} \text{parent}(A,D) & \text{or} \\ \text{parent}(A,I) & \&\quad \text{ancestor}_i(I,D) \end{cases}$$

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

parent(John, Mary)
parent(Mary, James)
parent(James, Alice)

then we have:

$$\begin{aligned}
\text{ancestor}_0 &= \{\} \\
\text{ancestor}_1 &= \{(\text{John, Mary}), (\text{Mary, James}), (\text{James, Alice})\} \\
\text{ancestor}_2 &= \text{ancestor}_1 \cup \{(\text{John, James}), (\text{Mary, Alice})\} \\
\text{ancestor}_3 &= \text{ancestor}_2 \cup \{(\text{John,Alice})\}
\end{aligned}$$

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

$$\text{ancestor} = \lim_{i \to \infty} \text{ancestor}_i$$

Linear Search

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

```
Loop : if i < n → if a[i] ≠ target → i := i + 1; Loop
                   fi
       fi
```

Loop_0 is defined as:

$$\mathsf{Loop}_0 = \perp$$

and Loop_{i+1} is defined as:

```
Loop_{i+1} : if i < n → if a[i] ≠ target → i := i + 1; Loop_i
                         fi
             fi
```

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

Since recursively defined domains like lists, stacks and trees are unbounded (in general may be infinite objects) they are implemented using pointers. 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.

5.3 Abstract Types

An abstract type is a type which is defined by its operations rather than its values.

The data types provided in programming languages are abstract types. For example, the representation of the integer type is hidden from the programmer.

The programmer is provided with a set of operations and a high-level representation of integers. The programmer only becomes aware of the lower level when an arithmetic overflow occurs.

An *Abstract data type* consists of a type name and operations for creating and manipulating objects of the type. A key idea is that of the separation of the implementation from the type definition. The actual format of the data is hidden (information hiding) from the user and the user gains access to the data only through the type operations.

There are two advantages to defining an abstract type as a set of operations. First, the separation of operations from the representation results in data independence. Second, the operations can be defined in a rigorous mathematical manner. As indicated in Chapter 3, algebraic definitions provide appropriate method for defining an abstract type. The formal specification of an abstract type can be separated into two parts. A syntactic specification with gives the signature of the operations and a semantic part in which axioms describe the properties of the operations.

In order to be fully abstract, the user of the abstract type must not be permitted access to the representation of values of the type. This is the case with the primitive types. For example, integers might be represented in two's complement binary numbers but there is no way of finding out the representation without going outside the language. Thus, a key concept of abstract types is the hiding of the representation of the values of the type. This means that the representation information must be local to the type definition.

Modula-3's approach is typical. An abstract type is defined using Modules – a definition module (called an interface), and an implementation module (called a module).

Since the representation of the values of the type is hidden, abstract types must be provided with *constructor* and *destructor* operations. A constructor operation composes a value of the type from values from some other type or types while a destructor operation extracts a constituent value from an abstract type. For example, an abstract type for rational numbers might represent rational numbers as pairs of integers. This means that the definition of the abstract type would include an operation which given a pair of integers returns a rational number (whose representation as an ordered pair is hidden) which corresponds to the the quotient of the two numbers. The rational additive and multiplicative identities corresponding to zero and one would be provided also.

Figure 5.6 is a definition definition module of an abstract type for complex numbers using Modula-2 and 5.7 is the corresponding implementation module.

Terminology: The terms *abstract data type* and *ADT* are also used to denote what we call an *abstract type*. □

```
DEFINITION MODULE ComplexNumbers;

TYPE Complex;

PROCEDURE MakeComplex ( firstNum, secondNum : Real ) : Complex;

PROCEDURE AddComplex ( firstNum, secondNum : Complex ) : Complex;

PROCEDURE MultiplyComplex ( firstNum, secondNum : Complex ) : Complex;
...
END ComplexNumbers.
```

Figure 5.6: Complex numbers abstract type

```
IMPLEMENTATION MODULE ComplexNumbers;

TYPE
    Complex = POINTER TO ComplexData
    ComplexData = RECORD
                        RealPart, ImPart : REAL;
                     END;

PROCEDURE MakeComplex ( firstNum, secondNum : Real ) : Complex;
    VAR result : Complex;

BEGIN
    new( result );
    result↑.RealPart := firstNum;
    result↑.ImPart := secondNum
    return result
END NewComplex;

PROCEDURE AddComplex ( firstNum, secondNum : Complex ) : Complex;
    VAR result : Complex;

BEGIN
    new( result );
    result↑.RealPart := firstNum↑.RealPart + secondNum↑.RealPart;
    result↑.ImPart := firstNum↑.ImPart + secondNum↑.ImPart
    return result
END AddComplex;

...

BEGIN
...
END ComplexNumbers.
```

Figure 5.7: Complex numbers implementation

```
DEFINITION MODULE GenericStack;

TYPE stack(ElementType);

PROCEDURE Push ( Element:ElementType; Var Stack : stack(ElementType) );
...
END GenericStack
```

Figure 5.8: A Generic Stack

5.4 Generic Types

Given an abstract type stack, the stack items would be restricted to be a specific type. This means that an abstract type definition is required for stacks which differ only in the element type contained in the stack. Since the code required by the stack operations is virtually identical, a programmer should be able to write the code just once and share the code among the different types of stacks. Generic types or *generics* are a mechanism to provide for sharing the code. The sharing provided by generics is through permitting the parameterization of type definitions. Figure 5.8 contains a Modula-2 definition module for a generic stack. The definition differs from that of an abstract type in that the type name is parameterized with the element type. At compile time, code appropriate to the parameter is generated.

Type Checking type free languages, data type parameterization (polymorphism)

The problem of writing generic sorting routines points out some difficulties with traditional programming languages. A sorting procedure must be able to detect the boundaries between the items it is sorting and it must be able to compare the items to determine the proper ordering among the items. The first problem is solved by parameterizing the sort routine with the type of the items and the second is solved by either parameterizing the sort routine with a compare function or by overloading the relational operators to permit more general comparisons.

Generic packages in Ada is a cheap way to obtain polymorphism. Generic packages are not compiled at compile time, rather they are compiled whenever they are parameterized with a type. So that if a programmer desires to sort a array of integers and an array of reals, the compiler will generate two different sort routines and the appropriate routine is selected at run-time.

5.5 Type Systems

Definition 5.1 *A* type system *is a set of rules for associating a type with expression in the language. A type system* rejects *an expression if it does not associate a type with the expression.*

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 reuseable 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.

A large percentage of errors in programs is due to the application of operations to objects of incompatible types. To assist the programmer in the detection of these errors, several schemes have been developed. 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 and Ada. 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 but they also permit programmer to provide type specifications.

If the error detection is to be delayed until execution time, then **dynamic** type checking is required. The programming languages Scheme and Prolog do not require the programmer to provide any information concerning an object's type and type checking is necessarily delayed until run time.

Neither Prolog, Scheme or Miranda require type declarations. Types may be declared in Miranda. For example the types for the arithmetic + operation are declared as follows:

```
+ ::  num -> num -> num
```

Modula-2, Ada, C++, Prolog, Scheme, Miranda – list primitive 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 iff they have the same name. Name equivalence was chosen for Modula-2, Ada, C (for records), and Miranda and requires type definitions to be global. 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. Formally,

Definition 5.2 *Two types* T, T′ *are* structurally equivalent *iff* T, 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 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 and type definitions are not required to be global.

Strong vs. Weak Type Checking

Definition 5.3 *A language is said to be* strongly typed *if it enforces type abstractions. That is, operations may be applied only to objects of the appropriate type.*

The language is said to be weakly typed.

For example, if the boolean constants are represented by 0 and 1 and the language permits a boolean to occur in arithmetic expressions, then the language is not enforcing the boolean type abstraction.

Most languages are strongly typed with respect to the primitive types supported by the language.

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.

SORT PROGRAMS

Weak type checking has the advantage of providing representation independence.

SETS VS LISTS

Type Inference

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 language Miranda provides a powerful type inference system so that a programmer need not declare any types. However, Miranda is strongly typed.

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 5.1 *given that: f is of type $A \rightarrow B$ and x is of type A*
infer that: f(x) is type correct and has type B

Static vs Dynamic type checking

If a language requires that the type of all variables be known compile time, then the a 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.

If a language does not require that the type of a variable be known at compile time, then a 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 combine the safety of static type checking with the flexibility of dynamic type checking through polymorphic types.

5.6 Overloading and Polymorphism

Overloading

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 5.4 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.*

The type of the plus operation defined for integer addition is

$$+ : int \times int \rightarrow int$$

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 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.*

Polymorphism

A type system is *polymorphic* if abstractions operate uniformly on arguments of a family of related types.

Definition 5.5 *A* polymorphic *operation is one that can be applied to different but related types of arguments.*

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).

5.7 Type Completeness

5.8 Exercises

1. Define algebraic semantics for the following data types.
   (a) Boolean

   `ADTBoolean`

   Operations

   ```
   and(boolean,boolean) → boolean
   or(boolean,boolean) → boolean
   not(boolean) → boolean
   ```

   Semantic Equations
   ```
   and(true,true) = true
   or(true,true) = true
   not(true) = false
   not(false) = true
   ```

   Restrictions

   (b) Integer
   (c) Real
   (d) Character
   (e) String
2. Name or Structure equivalence (type checking)
3. Algebraic Semantics: stack, tree, queue, grade book etc
4. Abstraction
5. Generalization

6. Name or Structure equivalence (type checking)

7. 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.

Chapter 6

Environment

Keywords and phrases: 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.

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 is the portion of the program text over which a definition is effective.

It is concerned with name control.

Binding Sequence

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.

Collateral

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 vary common but occur in Scheme and ML.

Sequential

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

Recursive

A *recursive* binding is one that uses the very bindings that it produces itself. In Pascal type definitions are recursive to permit the construction of recursive types. However, there is a constraint to insure that recursive types are restricted to pointer types.

6.1 Block structure

A *block* is a construct that delimits the scope of any declarations 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.

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.

A program has a *flat* block structure if it is partitioned into distinct blocks, an outer all inclosing block one or more inner 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 accessable by all subprograms even though it is used in only a couple of subprograms.

A program has *nested* block structure if blocks may be nested inside other blocks. This is typical of the block structure of the Algol-like languages. A block can be located close to the point of use.

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

A *global variable* is a variable that is referenced within a block but 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*. These are programs which are comprehensible in their entirety by an individual programmer. They are not general enough for very large programs. Programs which are written by many individual and which must consist of module which can be developed and tested independently of other modules. Such programming is called *programming in the large*. The subject of modules will be examined in a later chapter.

Activation Records

Each block

Storage for local variables.

Scope Rules

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

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

6.2 Declarations

6.3 Constants

literals

6.4 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}; { Modula-2}
enum T {I_1, ..., I_n}; // C++
T ::= I_1 | ...  | I_n || 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:

```
record
   I1 :  T1;
   ...
   In :  Tn;
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 elment 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:

```
struc name {
   T1 I1;
   ...
   Tn :  In;
};
```

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(I1, ...In)
```

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.

```
(I1, ...In)
```

The $I_i s$ are the entries in the tuple.

Here is an example of a variant record in 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.

```
FourSidedObject.WhatShape := Rectangle;
FourSidedObject.Length := 4.3;
FourSidedObject.Width := 7.5;
```

The corresponding definition in Miranda is

```
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 [exp | qualifier]

```
sqs = [ n*n | n <-[1..]  ]
factors n = [ r | r <-[1..n div 2]; n mod r = 0 ]
knights_moves [i,j] = [ [i+a,j+b] | a,b <-[-2..2]; a^2+^2=5 ]
```

Modula-2, Ada, C++, Prolog, Scheme, Miranda – power set

Prolog

```
[ ]
```

The Miranda syntax for lists is similar to that of Prolog however, elements of lists must be all of the same type.

```
[*]
```

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 = ^ListType
     ListType = record
                   item :  Itemtype;
                   next :  NextItem
                end;

// C++:  the list type
struc 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.

6.5 Variables

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.

6.6 Functions and Procedures

6.7 Persistant 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.

6.8 Exercises

1. Extend the compiler to handle constant, type, variable, function and procedure definitions and references to the same.
2. Static and dynamic scope

Chapter 7

Functional Programming

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:

$$sqr \ : \ Num \rightarrow Num$$

and may be defined as:

$$x \stackrel{sqr}{\longmapsto} x * x$$

A linear function f of type

$$f \ : \ Num \rightarrow Num$$

may be defined as:

$$x \stackrel{f}{\longmapsto} 3 * x + 4$$

The function:

$$x \overset{g}{\longmapsto} 3 * x^2 + 4$$

may be written as the composition of the functions f and sqr as:

$$f \circ sqr$$

where

$$f \circ sqr(x) = f(sqr(x)) = f(x * x) = 3 * x^2 + 4$$

The compositional operator is an example of a 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 *application* operation.
- A set of data objects.
- 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 *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 *executable specifications* and *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 and further, 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. □

7.1 The Lambda Calculus

Functional programming languages are based on the lambda calculus. The lambda calculus grew out of an attempt by Church and 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 polynomial expression $x^2 + 3x - 5$. The lambda *abstraction*

$$\lambda x.x^2 + 3x - 5$$

is used to denote the polynomial function whose values are given by the polynomial expression. Thus, the lambda abstraction is can be read as "the function of x whose value is $x^2 + 3x - 5$."

Instead of writing $p(x) = x^2 + 3x - 5$, we write

$$p = \lambda x.x^2 + 3x - 5$$

Instead of writing $p(1)$ to designate the value of p at 1, we write (p *1*) or

$$(\lambda x.x^2 + 3x - 5\ 1)$$

The value is obtained by *applying* the lambda abstraction to the 1. The application of a lambda abstraction to a value entails the substitution of 1 for x to obtain $1^2 + 3 \cdot 1 - 5$ which evaluates to -1.

We say that the variable x is *bound* in the body B of the lambda expression $\lambda x.B$. A variable which is not bound in a lambda expression is said to be *free*.

Abstract Syntax:

L ∈ Lambda Expressions
x ∈ Variables

L ::= x | (L_1 L_2) | (λx.L_3)

where (L_1 L_2) is function application, and (λx.L_3) is a lambda abstraction which defines a function with argument x and body L_3.

Figure 7.1: The Lambda Calculus

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 reguarded 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 $C = \lambda f.\lambda x.(f\ (f\ x))$

The pure lambda calculus has just three constructs: variables, function application, and function creation. Figure 7.1 gives the syntax of the lambda calculus. The notation $\lambda x.M$ is used to denote a function with parameter x and body M. For example, $\lambda x.(+\,x\,x)$ denotes the function that maps 3 to 3 + 3. Notice that functions are written in prefix form so in this instance the application of the function to 3 would be written: $\lambda x.(+\,x\,x)\,3$.

Lambda expressions are simplified using the β-rule:

$$((\lambda x.B)\ y) \Rightarrow B[x : E]$$

which says that an occurrence of x in B can be replaced with E. All bound

identifiers in B are renamed so as not to clash with the free identifiers in E.

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 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.

Operational Semantics

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.

$$\begin{array}{rcl}(+\ (*\ 5\ 6)\ (*\ 8\ 3)) & \rightarrow & (+\ 30\ (*\ 8\ 3)) \\ & \rightarrow & (+\ 30\ 24) \\ & \rightarrow & 54\end{array}$$

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 β-*reduction*. In the following sequence of reductions, the first step an example of β-*reduction*. The second step is the reduction required by the addition operator.

$$\begin{array}{r}(\lambda x.((+\ 3)\ x))\ 4 \\ ((+\ 3)\ 4) \\ 7\end{array}$$

The operational semantics of the lambda calculus define various operations on lambda expressions which enable lambda expressions to be *reduced* (evaluated) to a *normal form* (a form in which no further reductions are possible). Thus, the operational semantics of the lambda calculus are based on the concept of substitution.

A lambda abstraction denotes a function, to apply a lambda abstraction to an argument we use what is called β-*reduction*.

> The result of applying a lambda abstraction to an argument is 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.

The following formula is a formalization of *β-reduction.*

$$(\lambda x.B\ M) \leftrightarrow B[x : M]$$

The notation $B[x : M]$ means the replacement of free occurrences of x in B with M.

One problem which arises with *β-reduction* is the following. Suppose we apply β-reduction as follows.

$$(\lambda x.B\ M) \leftrightarrow B[x : M]$$

where y is a free variable in M but y occurs bound in B. Then upon the substitution of M for x, y becomes bound. To prevent this from occurring, we need to do the following.

> To prevent free variables from becoming bound requires the replacement of free variables with new free variable name, a name which does not occur in B.

This type of replacement is called *α-reduction.* The following formula is a formalization of *α-reduction*,

$$\lambda x.B \leftrightarrow \lambda y.B[x : y]$$

where y is not free in B.

Now we define *replacement* (substitution) as follows:

$x[x : M]$	$=$	M
$c[x : M]$	$=$	c where c is a variable or constant other than x.
$(A\ B)[x : M]$	$=$	$(A[x : M]\ B[x : M])$
$(\lambda x.B)[x : M]$	$=$	$(\lambda x.B)$
$(\lambda y.B)[x : M]$	$=$	$\lambda z.(B[y : z][x : M])$ where z is a new variable name which does not occur free in B or M.

Figure 7.2 defines the operational semantics of the lambda calculus in terms of β-reduction.

Reduction Order

Given a lambda expression, the previous section provides the tools required to evaluate the expression but, the previous section did not tell us what order we should use in reducing the lambda expressions.

Church-Rosser Theorem I If $E_1 \leftrightarrow E_2$ then there is an expression E, such that $E_1 \rightarrow E$ and $E_2 \rightarrow E$.

Interpreter:

$\mathcal{I} \in \mathbf{L} \rightarrow \mathbf{L}$

$$
\begin{array}{lcl}
\mathcal{I}[\![c]\!] & = & c \\
\mathcal{I}[\![x]\!] & = & x \\
\mathcal{I}[\![(\lambda x.B\ M)]\!] & = & \mathcal{I}[\![B[x:M]]\!] \\
\mathcal{I}[\![(L_1\ L_2)]\!] & = & \sigma(\phi) \\
 & & \quad \text{where } \mathcal{I}[\![L_1]\!] = \sigma,\ \mathcal{I}[\![L_2]\!] = \phi
\end{array}
$$

where

c is a constant
x is a variable
B, L_1, L_2, M are expressions

Figure 7.2: Operational semantics for the lambda calculus

An expression E is in *normal form* if it contains no redex.

Normal order reduction specifies that the *leftmost outermost redex* should be reduced first.

Church-Rosser Theorem II If $E_1 \rightarrow E_2$, and E_2 is in normal form, then there exists a *normal order* reduction sequence from E_1 to E.

Denotational Semantics

In the previous sections 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'.

In *denotational semantics* a function is viewed as a fixed set of associations between arguments and corresponding values.

We can express the semantics of the lambda calculus as a mathematical function, **Eval**, from expressions to values. For example,

$$\mathbf{Eval}[\![+\ 3\ 4]\!] = 7$$

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 **Eval** requires a second parameter containing the environment ρ 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 $\perp$ pronounced 'bottom'.

$$
\begin{array}{lcl}
\mathbf{Eval}[\![c]\!] & = & c \\
\mathbf{Eval}[\![x]\!]\ \rho & = & \rho\ x \\
\mathbf{Eval}[\![L_1\ L_2]\!]\ \rho & = & (\mathbf{Eval}[\![L_1]\!]\ \rho)\ (\mathbf{Eval}[\![L_2]\!]\ \rho) \\
\mathbf{Eval}[\![\lambda x.B]\!]\ \rho\ a & = & \mathbf{Eval}[\![\lambda x.B]\!]\ \rho[x : a] \\
\mathbf{Eval}[\![E]\!] & = & \perp
\end{array}
$$

where c is a constant or built-in function, x is a variable, B, L_1, L_2 are expressions and E is an expression which does not have a normal form.

Figure 7.3 gives a denotational semantics for the lambda calculus.

7.2 Recursive Functions

We extend the syntax of the lambda calculus to include named functions as follows:

Lambda Expressions

L ::= ...| x = L′ | ...

Where x is the name of the lambda expression L. With the introduction of named functions we obtain the potential for recursive definitions. 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.

$$\text{FAC} = \lambda n.(\text{if } (=\ n\ 0)\ 1\ (*\ n\ (\text{FAC } (-\ n\ 1))))$$

We can treat the recursive call as a free variable and replace the previous definition with the following.

$$\text{FAC} = (\lambda\text{fac}.(\lambda n.(\text{if } (=\ n\ 0)\ 1\ (*\ n\ (\text{fac } (-\ n\ 1)))))\ \text{FAC})$$

Let

$$H = \lambda\text{fac}.(\lambda n.(\text{if } (=\ n\ 0)\ 1\ (*\ n\ (\text{fac } (-\ n\ 1)))))$$

Note that H is not recursively defined. Now we can redefine FAC as

$$\text{FAC} = (H\ \text{FAC})$$

Semantic Domains:

EV = Expressible values

Semantic Function:

$\mathcal{E} \in \mathbf{L} \to \mathbf{L}$
$\mathcal{B} \in \mathbf{Constants} \to \mathbf{EV}$

Semantic Equations:

$$
\begin{array}{lcl}
\mathcal{E}[\![c]\!] & = & \mathcal{B}[\![c]\!] \\
\mathcal{E}[\![x]\!] & = & x \\
\mathcal{E}[\![(\lambda x.B\ M)]\!] & = & \mathcal{E}[\![B[x:M]]\!] \\
\mathcal{E}[\![(L_1\ L_2)]\!] & = & \sigma(\phi) \\
 & & \text{ where } \mathcal{E}[\![L_1]\!] = \sigma,\ \mathcal{E}[\![L_2]\!] = \phi \\
\mathcal{E}[\![E]\!] & = & \bot
\end{array}
$$

where

c is a constant
B, L_1, L_2, M are expressions
E is an expression which does not have a normal form

and

$$
\begin{array}{lcl}
x[x:M] & = & M \\
vc[x:M] & = & vc \text{ where } vc \text{ is a variable or constant other than } x. \\
(A\ B)[x:M] & = & (A[x:M]\ B[x:M]) \\
(\lambda x.B)[x:M] & = & (\lambda x.B) \\
(\lambda y.B)[x:M] & = & \lambda z.(B[y:z][x:M]) \text{ where } z \text{ is a new variable name} \\
 & & \text{which does not occur free in } B \text{ or } M.
\end{array}
$$

Figure 7.3: Denotational semantics for the lambda calculus

This equation 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,

$$\text{FAC} = Y\ H$$

Y is called a *fixed point combinator*. With the function Y this equation defines FAC without the use of recursion. From the previous two equations, the function Y has the property that

$$Y\ H = H\ (Y\ H)$$

As an example, here is the computation of FAC 1 using the Y combinator.

$$\begin{array}{lcl}
\text{FAC } 1 & = & (Y\ H)\ 1 \\
 & = & H\ (Y\ H) \\
 & = & \lambda\text{fac}.(\lambda n.(\text{if } (=\ n\ 0)\ 1\ (*\ n\ (\text{fac } (-\ n\ 1)))))\ (Y\ H)\ 1 \\
 & = & \lambda n.(\text{if } (=\ n\ 0)\ 1\ (*\ n\ ((Y\ H)\ (-\ n\ 1))))\ 1 \\
 & = & \text{if } (=\ 1\ 0)\ 1\ (*\ 1\ ((Y\ H)\ (-\ 1\ 1))) \\
 & = & (*\ 1\ ((Y\ H)\ (-\ 1\ 1))) \\
 & = & (*\ 1\ ((Y\ H)\ 0)) \\
 & = & (*\ 1\ (H\ (Y\ H)\ 0)) \\
\ldots & & \\
 & = & (*\ 1\ 1) \\
 & = & 1
\end{array}$$

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.

$$\begin{array}{lcl}
(Y\ H) & = & (\lambda h.(\lambda x.(h\ (x\ x))\ \lambda x.(h\ (x\ x)))\ H) \\
 & = & (\lambda x.(H\ (x\ x))\ \lambda x.(H\ (x\ x))) \\
 & = & H\ (\lambda x.(H\ (x\ x))\ \lambda x.(H\ (x\ x))) \\
 & = & H\ (Y\ H)
\end{array}$$

7.3 Lexical Scope Rules

Lexical scope rules vars refer to nearest enclosing environment, parameters are passed, after renaming, by textual substitution

An aspect of abstraction is the ability to have local definitions. This is achieved in the lambda calculus by the following syntactic extensions.

Lambda Expressions

L ∈ Lambda Expressions
x ∈ Variables
...

L ::= ...| let x_0 = L_0 in L'_0 | letrec x_1 = L_1... x_n = L_n in L′ |...

The `let` extension is for non-recursive definitions and the *letrec* extension is for recursive definitions Here is a simple `let` example.

$$let\ x = 3\ in\ (*\ x\ x)$$

Lets may be used where ever a lambda expression is permitted. For example,

$$\lambda y.let\ x = 3\ in\ (*\ y\ x)$$

is equivalent to

$$\lambda y.(*\ y\ 3)$$

`Let` expressions may be nested. Actually `lets` do not extend the lambda calculus they are just an abstraction mechanism as this equivalence shows.

$$(let\ v = B\ in\ E) \equiv ((\lambda v.E)\ B)$$

`Letrecs` do not extend the lambda calculus they are also just an abstraction mechanism as this equivalence shows.

$$(letrec\ v = B\ in\ E) \equiv (let\ v = Y\ (\lambda v.B)\ in\ E)$$

7.4 Functional Forms

Functional languages treat functions as first-class values: they can be passed as parameters, returned as results, built into composite values, and so on. Functions whos parameters are all non-functional are called *first-order*. A function that has a functional parameter is called a *higer-order* function or functional. Functional composition is an example of a functional.

```
double (x) = x * x
quad = double ∘ double
twice (f) = f ∘ f
odd = not ∘ even
```

Functionals are often used to construct reusable code and to obtain much of the power of imperative programming within the functional paradigm.

An important functional results from *partial application* For example, suppose there is the function `power` defined as follows:

```
power ( n, b) = if n = 0 then 1 else b*power(n-1, b)
```

As written `power` raises a base `b` to a positive integer power `n`. The expression `power(2)` is a function of one argument which when applied to a value returns the square of the value.

composition f ∘ g (x) = f(g(x))

dispatching f & g (x) = (f(x), g(x))

parallel

currying

apply apply(f,a) = f(a)

iterate iterate(f,n) (a) = f(f(...(f(a))...))

$(\lambda$ x.λ y.((* x) y) 3 = λy.((* 3) y)

7.5 Evaluation Order

A function is *strict* if it is sure to need its argument. If a function is non-strict, we say that it is *lazy*. Why???

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.

Concurrent evaluation

7.6 Values and Types

Pre-defined Data Types

Integer, Real, Character, Tuples, Lists Enumerations, algebraic types (unions)

7.7 Type Systems and Polymorphism

Pattern matching, Product type, sequences, functions, ML, Miranda

7.8 Program Transformation

Since functional programs consist of function definitions and expression evaluations they are suitable for program transformation and formal proof just like any other mathematical system. It is the principle of referential transparency that makes this possible. The basic proof rule is that: *identifiers may be replaced by their values.*

For example,

```
f 0 = 1
f n+1 = (n+1)*(f n)


fp 0 fn = fn
fp n+1 in = fp n (n+1)*in


f n = fp n 1

f 0 = fp 0 1 by definition of f and fp

assume f n = fp n 1
```

```
show f n+1 = fp n+1 1
f n+1 = (n+1)*f n
      = (n+1)*fp n 1
and
k*fp m n = fp m k*n since
1*fp m n = fp m 1*n
and
(k+1)*fp m n = k*fp m n + fp m n = fp m k*n + fp m n
```

fold

unfold

7.9 Pattern matching

```
f 0 = 1
f (n+1) = (n+1)*f(n)
```

```
insert (item Empty_Tree) = BST item Empty_Tree Empty_Tree
insert (item BST x LST RST) = BST x insert (item LST) RST if item < x
                              BST x LST insert( item RST ) if item > x
```

7.10 Combinatorial Logic

The β 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:

$$
\begin{array}{lcl}
\mathrm{S} & = & \lambda f.(\lambda g.(\lambda x.fx(gx))) \\
\mathrm{K} & = & \lambda x.\lambda y.x \\
\mathrm{I} & = & \lambda x.x \\
\mathrm{Y} & = & \lambda f.\lambda x.(f\ (x\ x))\lambda x.(f\ (x\ x))
\end{array}
$$

These definitions lead to transformation rules for sequences of combinators. The reduction rules for the SKI calculus are given in Figure 7.4. 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.

- S f g x → f x (g x)
- K c x → c
- I x → x
- Y e → e (Y e)
- (A B) → A B
- (A B C)→ A B C

Figure 7.4: Reduction rules for SKI calculus

$$
\begin{array}{lcl}
\mathcal{C}[\![CV]\!] & \rightarrow & CV \\
\mathcal{C}[\![(E_1\ E_2)]\!] & \rightarrow & (\mathcal{C}[\![E_1]\!]\ \mathcal{C}[\![E_2]\!]) \\
\mathcal{C}[\![\lambda x.E]\!] & \rightarrow & \mathcal{A}[\![(x, \mathcal{C}[\![E]\!])]\!] \\
\mathcal{A}[\![(x, x)]\!] & \rightarrow & I \\
\mathcal{A}[\![(x, c)]\!] & \rightarrow & (Kc) \\
\mathcal{A}[\![(x, (E_1\ E_2))]\!] & \rightarrow & ((S\ \mathcal{A}[\![(x, E_1)]\!])\ \mathcal{A}[\![(x, E_2)]\!])
\end{array}
$$

Where CV is a constant or a variable.

Figure 7.5: Translation rules for the lambda calculus to SKI code

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 7.5 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.

- S (K e) (K f) → K (e f)
- S (K e) I → e
- S (K e) f → (B e) f
- S e (K f) → (C e) f

The optimizations must be applied in the order given.

Figure 7.6: Optimizations for SKI code

Optimizations

Notice that the size of the SKI code grows quadratically in the number of bound variables. Figure 7.6 contains a number of optimizations for SKI code. Among the optimizations are two additional combinators, the `B` and `C` combinators.

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.

7.11 Scheme

Scheme, a descendent 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 ∈ Expressions
A ∈ Atoms (variables and constants)
...
E ::= A | (E...) | (lambda (A...) E) | ...

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 permits lambda abstractions of more than one parameter.

Scheme provides a number of built in functions among which are $+$, $-$, $*$, $/$, $<$, $\leq$, $=$, $\geq$, $>$, 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 ::= ...| (define I E) | ...

Cons, Car and Cdr

The list is the basic data structure. 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 7.7 contains an example of stack operations writtem 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`.

```
( 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 7.7: Stack operations defined in Scheme

Local Definitions

Scheme provides for local definitions with the following syntax

Scheme Syntax
...
B ∈ Bindings
...

E ::= ...| (let B_0 E_0) | (let* B_1 E_1) | (letrec B_2 E_2) |...
B ::= ((I E)...)

The let values are computed and bindings are done in parallel, 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.

7.12 Haskell

In contrast with LISP and Scheme, Haskell is a modern functional programming language.

Figure 7.8

```
module Qs where
```

```
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 = stkToLst s

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
```

Figure 7.8: A sample program in Haskell

```
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]


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:_) = 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)
```

```
this
```

7.13 Discussion and Further Reading

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 combinatorial logic and combinatorial logic into the code for a graph reduction machine. All of these are virtual machines.

Models of the lambda calculus.

History [18] For an easily accessable introduction to functional programming, the lambda calculus, combinators and a graph machine implementation see [25]. For Backus' Turing Award paper on functional programming see [2]. The complete reference for the lambda calculus is [3]. For all you ever wanted to know about combinatory logic see [6, 7, 13]. For an introduction to functional programming see [12, 4, 20]. For an intoduction to LISP see [19] and for common LISP see [29]. For a through introduction to Scheme see [1]. Haskell On the relationship of the lambda calculus to programming languages see [17]. For the implementation of functional programming languages see [12, 22].

7.14 Exercises

1. Simplify the following expressions to a final (*normal*) form, if one exists. If one does not exist, explain why.

   (a) $((\lambda x.(x\ y))(\lambda z.z))$

   (b) $((\lambda x.((\lambda y.(x\ y))\ x))(\lambda z.w))$

   (c) $((((\lambda f.(\lambda g.(\lambda x.((f\ x)(g\ x)))))(\lambda m.(\lambda n.(n\ m))))(\lambda n.z))p)$

   (d) $((\lambda x.(x\ x))(\lambda x.(x\ x)))$

   (e) $((\lambda f.((\lambda g.((f\ f)\ g))(\lambda h.(k\ h))))(\lambda y.y)))$

   (f) $(\lambda g.((\lambda f.((\lambda x.(f\ (x\ x)))(\lambda x.(f\ (x\ x))))))\ g))$

   (g) $(\lambda x.(\lambda y.((-\ y)\ x)))\ 4\ 5$

   (h) $((\lambda f.(f\ 3))(\lambda x.((+\ 1)\ x)))$

2. In addition to the β-rule, the lambda calculus includes the following two rules:

   α-rule: $(\lambda x.E) \Rightarrow (\lambda y.E[x:y])$
   η-rule: $(\lambda x.E\ x) \Rightarrow E$ where x does not occur free in E

   Redo the previous exercise making use of the η-rule whenever possible. What value is there in the α-rule?

3. The lambda calculus can be used to simulate computation on truth values and numbers.

   (a) 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 $((\text{true}\ E_1)\ E_2) \Rightarrow E_1$ and $((\text{false}\ E_1)\ E_2) \Rightarrow E_2$. Define lambda expressions **not, and,** and **or** that behave like their Boolean operation counterparts.

   (b) Let **0** be the name of the lambda expression $\lambda x.\lambda y.y$, **1** be the name of the lambda expression $\lambda x.\lambda y.(x\ y)$, **2** be the name of the lambda expression $\lambda x.\lambda y.(x\ (x\ y))$, **3** be the name of the lambda expression $\lambda x.\lambda y.(x; (x\ (x\ y)))$, and so on. Prove that the lambda expression **succ** defined as $\lambda z.\lambda x.\lambda y.(x\ ((z\ x)\ y))$ rewrites a number to its successor.

4. Recursively defined functions can also be simulated in the lambda calculus. Let **Y** be the name of the expression $\lambda f.\lambda x.(f\ (x\ x))\lambda x.(f\ (x\ x))$

   (a) Show that for any expression E, there exists an expression W such that $(\mathbf{Y}\ E) \Rightarrow (W\ W)$, and that $(W\ W) \Rightarrow (E\ (W\ W))$. Hence, $(\mathbf{Y}\ E) \Rightarrow E(E(E(...E(W\ W)...)))$

(b) 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, $((\textbf{add}\ m)\ n) \Rightarrow m + n$.

5. Data constructors can be modeled in the lambda calculus. Let **cons** $= (\lambda a.\lambda b.\lambda f.fab)$, **head** $= (\lambda c.c(\lambda a.\lambda b.a))$ and **tail** $= (\lambda c.c(\lambda a.\lambda b.b))$. Show that

   (a) **head** (**cons** a b) = a

   (b) **tail** (**cons** a b) = b

6. Show that (((S(KK))I)S) is (KS).

7. What is (((SI)I)X) for any formula X?

8. Compile $(\lambda x.\ +\ x\ x)$ to SKI code.

9. Compile $\lambda x.(F(x\ x))$ to SKI code.

10. 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.

11. Apply the optimizations to the SKI code for $\lambda x.\lambda y.xy$ and compare the result with the unoptimized code.

12. Apply the optimizations to the SKI code for $\lambda x.(F(x\ y))$ and $\lambda y.(F(x\ y))$.

13. Association lists etc

14. HOF

Chapter 8

Logic Programming

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.

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

$$S_0 \times S_1 \times ... \times S_n$$

The squaring function for natural numbers may be written as a set of tuples as follows:

$$\{(0,0),(1,1),(2,4)...\}$$

Such a set of tuples is called a relation and in this case the tuples define the squaring relation.

$$sqr = \{(0,0),(1,1),(2,4)...\}$$

Abstracting to the name sqr and generalizing an individual tuple we can define the squaring relation as:

$$sqr = (x, x^2)$$

Parameterizing the name gives:

$$sqr(X,Y) \leftarrow Y \text{ is } X * X$$

In the logic programming language Prolog this would be written as:

```
sqr(X,Y) ← Y  is  X*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 is X*X forces the evaluation of X*X and unifies the answer with Y. The Prolog code

```
P ← Q.
```

may be read in a number of ways; it could be read P where Q or P if 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*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, male(X), siblingof(X,Y), and parentof(Y,Z) which define the obvious relations, then we can define the predicate uncleof(X,Z) which implements the obvious relation as follows:

```
uncleof(X,Z) ← male(X), siblingof(X,Y), 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."

The difference between logic programming and functional programming may be illustrated as follows. The logic program

```
f(X,Y) ← Y = X*3+4
```

is an abreviation 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.

8.1 Inference Engine

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

$\texttt{f if a}_0\texttt{, ..., a}_m\texttt{.}$

$\texttt{g if f, b}_0\texttt{, ..., b}_n\texttt{.}$

the fact

$\texttt{g if a}_0\texttt{, ..., a}_m\texttt{, b}_0\texttt{, ..., b}_n\texttt{.}$

can be derived.

Unification is the binding of variables. For example

8.2 Syntax

There are just four constructs: constants, variables, function symbols, predicate symbols, and two logical connectives, the comma (and) and the implication

symbol.

Horn Clause Logic

P ∈ Programs
C ∈ Clauses
Q ∈ Queries
A ∈ Atoms
T ∈ Terms
X ∈ Variables

P ::= C... Q...
C ::= G [$\leftarrow$ G_1 [$\wedge$ G_2]...] .
G ::= A [(T [,T]...)]
T ::= X | A [(T [,T]...)]
Q ::= G [,G]... ?

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.

8.3 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.

```
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.

$\{A[N] = \sum_{i=N}^{N} A[i]\}$

$\mathrm{sum}([A[N]], A[N]).$

$\{\sum_{i=I}^{N} A[i] = A[I] + S \text{ if } 0 < I, \sum_{i=I+1}^{N} A[i] = S\}$

$\mathrm{sum}([A[I], ..., A[N]], A[I] + S) \leftarrow \mathrm{sum}([A[I+1], ..., A[N]], S).$

Operational Semantics

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 ← a.
b?
```

We can conclude b by *modus ponens* given that b ← a and a. Alternatively, if b is assume to be false then from b ← a and *modus tollens* we infer ¬ a but since a is given we have a contradiction and b must hold. The following program illustrates unification.

```
parent_of(a,b).
parent_of(b,c).
ancestor_of(Anc,Desc) ← parent_of(Anc,Desc).
ancestor_of(Anc,Desc) ← parent_of(Anc,Interm) ∧
                        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 8.1 *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 $\Box$ for contradiction. Resolution is based on the inference rule *modus tollens* and *unification*. This is the *modus tollens* inference rule.

From	$\neg B$
and	$B \leftarrow A_0, ..., A_n$
infer	$\neg A_0$ 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 hypothesises A_0 and $B \leftarrow A_0, A_1$.

1	From	$\neg B$
2	and	A_0
3	and	$B \leftarrow A_0, A_1$
4	infer	$\neg A_0$ or $\neg A_1$
5	choose	$\neg A_0$
6	contradiction	$\Box$
7	choose	$\neg A_1$
8	no further possibilities	open

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 $\Box$.

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) $\leftarrow$ parent_of(A, D)
5. ancestor_of(A, D) $\leftarrow$ 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.

$\neg$ancestor($ogden, mikko$)	the assumption
$\neg$parent($ogden, X$) or $\neg$ancestor($X, mikko$)	resolution
$\neg$parent($ogden, X$)	first choice
$\neg$parent($ogden, anthony$)	unification with first entry
$\square$	produces a contradiction
$\neg$ancestor($anthony, mikko$)	second choice
$\neg$parent($anthony, mikko$)	resolution
$\square$	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 *anthonoy* 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 8.1 contains a formal definition of unification in Prolog Unification subsumes

- single assignment
- parameter passing
- record allocation
- read/write-once field-access in records

To illustrate the inference rules, consider the following program consisting of a rule, two facts and a query:

a $\leftarrow$ b $\wedge$ c .

```
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).
```

Figure 8.1: Unification

$$\frac{\text{A}_1 \leftarrow \text{B.},\ \text{?- A}_1,\ \text{A}_2,\ldots,\text{A}_n.}{\text{?- B, A}_2,\ldots,\text{A}_n.}$$

$$\frac{\text{?- true, A}_1,\ \text{A}_2,\ldots,\text{A}_n.}{\text{?- A}_1,\ \text{A}_2,\ldots,\text{A}_n.}$$

Figure 8.2: Inference Rules

```
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
```

Figure 8.3: A simple interpreter for pure Prolog

b ← d .
b ← e .
?- a .

By applying the inference rules to the program we derive the following additional queries:

?- b ∧ c .
?- d ∧ c .
?- e ∧ c.
?- c.
?-

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.

A simple interpreter for Pure Prolog

An interpreter for pure Prolog can be written in Prolog. Figure 8.3 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 8.4

Declarative Semantics

The *declarative* semantics of logic programs is based on the standard model-theoretic semantics of first-order logic.

Definition 8.2 *Let P be a logic program. The* Herbrand universe *of P, denoted by $\mathcal{U}(P)$ is the set of ground terms that can be formed from the constants and function symbols appearing in P*

Definition 8.3 *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 8.4 *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 8.5 *An interpretation I is a* model *for a logic program if for each ground instance of a clause in the program $A \leftarrow B_1, ..., B_n$ A is in I if $B_1, ..., B_n$ 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 0f the function, if it exists.

Logic Programming (Horn Clause Logic) – Operational Semantics
Abstract Syntax:

P ∈ Programs
C ∈ Clauses
Q ∈ Queries
T ∈ Terms
A ∈ Atoms
X ∈ Variables

P ::= (C | Q)...
C ::= G [$\leftarrow$ G_1 [$\wedge$ G_2]...] .
G ::= A [(T [,T]...)]
T ::= X | A [(T [,T]...)]
Q ::= G [,G]... ?

Semantic Domains:

$\beta \in \mathbf{B}$ = Bindings
$\epsilon \in \mathbf{E}$ = Environment

Semantic Functions:

$\mathcal{R} \in \mathrm{Q} \to \mathrm{B} \to \mathrm{B} + (\mathrm{B} \times \{\text{ yes }) + \{\text{ no }\}$
$\mathcal{U} \in \mathrm{C} \times \mathrm{C} \to \mathrm{B} \to \mathrm{B}$

Semantic Equations:

$$\mathcal{R}[\![?]\!]\beta, \epsilon = (\beta,\ \text{yes})$$

$$\mathcal{R}[\![G]\!]\beta, \epsilon = \beta' \quad \text{where} \quad G' \in \epsilon,\ \mathcal{U}[\![G, G']\!]\beta = \beta'$$

$$\mathcal{R}[\![G]\!]\beta, \epsilon = \mathcal{R}[\![B]\!]\beta', \epsilon \quad \text{where} \quad (G' \leftarrow B) \in \epsilon,\ \mathcal{U}[\![G, G']\!]\beta = \beta'$$

$$\mathcal{R}[\![G_1, G_2]\!]\beta, \epsilon = \mathcal{R}[\![B, G_2]\!](\mathcal{R}[\![G_1]\!]\beta, \epsilon), \epsilon$$

$$\mathcal{R}[\![\ \mathrm{G}\ ]\!]\beta, \epsilon = \text{no} \quad \text{where no other rule applies}$$

Figure 8.4: Operational semantics

8.4 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.

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 sufixes 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 two simple types of constants, string and numeric. Arrays may be represented as a relation. For example, the two dimensional matrix

$$\text{data} = \begin{pmatrix} \text{mary} & 18.47 \\ \text{john} & 34.6 \\ \text{jane} & 64.4 \end{pmatrix}$$

may be written as

```
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.

```
book(author( last(aaby), first(anthony), mi(a)),
     title('programming language concepts),
     pub(wadsworth),
     date(1991))
```

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.

lookup(Key,Data,bt(Key,Data,LT,RT))
lookup(Key,Data,bt(Key0,Data0,LT,RT)) ← Key @< Key0,
    lookup(Key,Data,LT)
lookup(Key,Data,bt(Key0,Data0,LT,RT)) ← Key @> Key0,
    lookup(Key,Data,RT)

This is a sequence of calls. Note that the initial call is with the variable BT.

lookup(john,46,BT), lookup(jane,35,BT), lookup(allen,49,BT), lookup(jane,Age,BT).

The first three calls initialize the dictionary to contain those entries while the last call extracts janes'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:

concat_dl(Xs/Ys, Ys/Zs, Xs/Zs)

Here is an example of a use of the definition.

```
?- concat_dl([1,2,3|X]/X,[4,5,6|Y]/Y,Z).

   _X = [4,5,6 | _11]
   _Y = _11
   _Z = [1,2,3,4,5,6 | _11] / _11
Yes
```

The relation between ordinary lists and difference lists is defined as follows:

ol_dl([ ],X/X) ← var(X)
ol_dl([F|R],[F|DL]/Y) ← 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.

times(X,Y,X×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

$$\text{times(X,Y,Z)} \leftarrow Z \text{ is X}\times\text{Y}$$

8.5 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.

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

Here is a direct restatement of the definition in a relational form.

```
factorial(0,1)
factorial(N,N×F) ← factorial(N-1,F)
```

In Prolog this definition does not evaluate either of the expressions N-1 or N×F thus the value 0 will not occur. To force evaluation of the expressions we rewrite the definition as follows.

```
factorial(0,1)
factorial(N,F)← M is N-1, factorial(M,Fm), F is N×Fm
```

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 (*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.

```
fac(N,F) ← fac(N,1,F)
fac(0,F,F)
fac(N,P,F) ← NP is N×P, M is N-1, fac(M,NP,F)
```

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.

$$\text{reverse}([\,],[\,])$$
$$\text{reverse}([H|T], R) \leftarrow \text{reverse}(T, Tr), \text{concat}(Tr, [H], R)$$

and here is an iterative version.

$$\begin{array}{l}\text{rev}(L, R) \leftarrow \text{rev}(L, [\,], R)\\ \text{rev}([\,], R, R)\\ \text{rev}([H|T], L, R) \leftarrow \text{rev}(T, [H|L], R)\end{array}$$

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).

8.6 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*.

8.7 Exceptions

Logic programming provides an unusually simple method for handling exception conditions. Exceptions are handled by backtracking.

8.8 Prolog $\neq$ Logic Programming

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 ...

While there is no standard syntax for Prolog, most implementations recognize the syntax given in Figure 8.5.

As in functional programming, lists are an important data structure logic programming. The empty list is represented by [ ], a list of n elements by $[X_1, ..., X_n]$ and the first i elements of a list and the rest of the list by $[X_1, ..., X_i|R]$. In addition, data structures of arbitrary complexity may be constructed from the terms.

Prolog

P ∈ Programs
C ∈ Clauses
Q ∈ Query
H ∈ Head
B ∈ Body
A ∈ Atoms
T ∈ Terms
X ∈ Variable

P ::= C... Q...
C ::= H [← B] .
H ::= A [(T [,T]...)]
B ::= G [, G]...
G ::= A [([X | T]...)]
T ::= X | A [(T...)]
Q ::= ?- B .

Figure 8.5: Abstract Syntax

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.

```
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.

```
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 [ ]. Prologs 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.

```
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.

```
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.

```
p( a ).
r( b ) ← not p( Y ).
?- not p(b).
```

The goal succeeds but is not a logical consequence of the logic program.

```
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.

```
a(1).
a(2).
a(3).
p ← a(I),!,print(I),nl,fail.
?- p.
1

No
```

Extralogical Features

Input and output introduce side effects.

The extralogical primitives `bagof`, `setof`, `assert`, and `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.

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 8.6 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).`

```
trace(Q) ← trace1([Q])
trace1([])
trace1([true|R]) ← !, trace1(R).
trace1([fail|R]) ← !, print('< '), print(fail), nl, fail.
trace1([B|R]) ← B =..[','|BL], !, concat(BL,R,NR), trace1(NR).
trace1([F|R]) ← builtin(F),
                print('> '), print([F|R]), nl,
                F,
                trace1(R),
                print('<'), print(F), nl
trace1([F|R]) ← clause(F,B),
                print('>'), print([F|R]),nl,
                trace1([B|R]),
                print('< '), print(F), nl
trace1([F|R]) ← \+ builtin(F), \+ clause(F,B),
                print('> '), print([F|R]),nl,
                print('< '), print(F), print(' '), print(fail), nl, fail
```

Figure 8.6: Program tracer for Prolog

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.

```
L1 = [ ], L = [2]
L1 = [V1], L = [V1,2]
L1 = [V1,V2], L = [V1,V2,2]
...
```

However, if the order of the rules defining *concat* are interchanged,

```
concat([H|T],L,[H|R]) ← concat(T,L,R).
concat([ ],L,L).
?- concat(L1,[2],L).
```

then the execution fails to terminate, entering an infinite loop since the first rule is always applicable.

8.9 Database query languages

The 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 propery 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.

8.10 Logic Programming vs Functional Programming

Functional composition vs composition of relations, backtracking, type checking

8.11 Further Reading

History
Kowalski's paper[16]
Logic programming techniques
Implementation of Prolog
SQL
DCG

8.12 Exercises

1. Modify concat to include an explicit occurs check.
2. 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.
3. Business Data base
4. Blocks World
5. CS Degree requirements; course(dept,name,prereq). don't forget w1 and w2 requirements.
6. Circuit analysis
7. Tail recursion
8. Compiler
9. Interpreter
10. Tic-Tac-Toe
11. DCG
12. Construct Prolog analogues of the relational operators for union, set difference, cartesian product, projection and selection.
13. Airline reservation system

Chapter 9

Imperative Programming

Imperative programming is characterized by programming with a state and commands which modify the state.

Imperative: a command or order

Procedure: a) the act, method or manner of proceeding in some process or course of action b) a particular course of action or way of doing something.

Keywords and phrases: Assignment, goto, structured programming, command, statement, procedure, control-flow, imperative language, assertions, axiomatic semantics. state, variables, instructions, control structures.

Imperative programming languages are characterized by sequences of bindings. Programs are sequences of bindings (state changes) in which a name may be bound an object at one point in the program and later bound to a different object. Since the order of the bindings affects the value of expressions, one of the prime issue in the imperative paradigm is control over the sequence of bindings. These bindings are commands which are issued to a machine. The machine may not be an actual machine but a virtual machine. Thus, it is common to refer to the Pascal machine or the Ada machine. The virtual machine provides access to the hardware through the compiler and the operating system.

In the context of imperative programming languages, a state is more than just the association between variables and values. It also includes the location of

control in the program. A portion of code cannot be understood by examining its constituent parts. The reason is that during the execution of the code, control may be transferred to an arbitrary point outside of the code. Thus, the whole program may need to be examined in order to understand a small portion of code.

The imperative programming paradigm is an abstraction of real computers which in turn are based on Turing machines and the Von Neumann Machine with its registers and memory. At the heart of each is the concept of a modifiable store. Variables and assignment are the programming language analog of the modifiable store. Thus memory is the object that is manipulated and imperative programming languages provide a variety of commands to manipulate memory.

9.1 Variables and Assignment

assignables–arrays records etc

In imperative programming, a **variable** is a name that may be assigned to a value and later reassigned to a value. The collection of variables and the associated values constitute the *state*. The state is a logical model of storage which is an association between memory locations and values. **Aside:** A variable is not a storage cell, nor is it a name of a storage cell; it is the name of a value. Most descriptions of imperative programming languages are tied to hardware and implementation considerations where (name, variable, value) = (address, storage cell, 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 where in the assignment, `X := X+2`, the `X` on the **l**eft of the assignment denotes a location while the `X` on the **r**ight hand side denotes the value.

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.

```
Modula-2:  variable V of type T
var V : T;

// C: variable V of type T
T V;
```

A variable is **bound** to a value by an assignment. Binding of a variable V to the value of an expression E.

```
−−Ada
V := E;

Modula-2
V := E;

//C
V = E;

APL
V ←

; Scheme (setq V E)
```

Aside: The use of the symbol (=) in C confuses the distinction between definition, equality and assignment. The 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. In C the symbol (==) is used for equality, while the form: `Type Variable` is used for definitions. □

The **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 to a value. **Aside:** The word "assign" is used in accordance with its English meaning; a name is assigned to an object, not the reverse. This is in contrast to the usual 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:

```
X := 3;
X := X + 1
```

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 occurence of assignments of the form: `X := X <op> E` Algol-68 and C provide an alternative notation of the form: `X <op> := E`. A *multiple assignment* of the form: V_0 := V_1 := ... := V_n := E causes several names to be assigned to the same value. This form of the assignment is found in Algol-60 and in C. A *simultaneous assignment* of the form: V_0,...,V_n := E_0,...,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 the 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.

9.2 Control Structures

Since the assignment can assign names to new values, the order in which assignments are executed is important. *Control structures* are syntactic structures that allow assignments to be combined in useful ways.

Terminology: 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 assignments, skip, and control structures. □

Terminology: *Imperative* programming is also called *procedural* programming. In either case the implication is clear. Programs are directions or orders for performing an action. □

In addition to the simple assignment, imperative programming languages provide a rich assortment of sequence control mechanisms. Three types of commands are required for sequence control: *composition*, *alternation* and *iteration.*

Composition. Composition is usually indicated by placing commands in textual sequence and when it is necessary to determine the syntactic termination point of a command either line separation or a special symbol (such as the semicolon) is used. Operationally the ordering describes the order or sequence in which the elements are to be executed. At a more abstract level, composition of commands is indicated by using a composition operator such as the semicolon (C_0;C_1). In PL/1 and C the semicolon is used as a terminator rather than as a composition operator.

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: 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 which whose value at run time determines the number of compositions. The **while**, **repeat** and **for** commands are the most common representative of iteration.

Abstraction: A sequence of commmands may be named and the name used to invoke the sequence of commands.

Skips

The simplest kind of command is the **skip** command. It has no effect.

Procedure Calls/Coroutines

A procedure is an abstraction of a sequence of commands. The **procedure call** is a reference to the abstraction. The semantics of the procedure call is determined by the semantics of the procedure body.

Parameters and arguments

Parameter passing

in, out, in/out, name

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.

Conditional execution discussion: if-then-else, case/switch

```
-- 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;
```

Often the case command is extended to include an alternative for unspecified cases.

```
case <selector expression> of
...
   <value> :  <command>;
...
   otherwise <command>
end
```

Iteration

An **iterative command** has a body which is to be executed repeatedly and has an expression which determine when the execution will cease.

Loop discussion: while, repeat, for, escapes

```
[ loop_name:  ]
[ iteration_scheme ]
loop
    commands
end loop [ loop_name ]

[ loop_name:  ]
while condition loop
      commands
end loop [ loop_name ]

[ loop_name:  ]
for identifier in [ reverse ] descrete_range loop
```

```
    commands
end loop [ loop_name ]

[ loop_name:  ]
for identifier in [ reverse ] descrete_range loop
    ...
    exit [ loop_name ] [ when condition];
    ...
end loop [ loop_name ]
```

Iterators and Generators

Definition 9.1 *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 9.2 *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 [I..J] DO S;
FOR Item in List DO S;
FOR Item in Set DO S;
FOR Item in Tree DO S;
...
```

etc

Productive Use of Failure

1. Iterators and Generators in Icon (ML, Prolog...)
2. Backtracking in Prolog

In Prolog ... the form ... generator(P) ... fail . Backtracking produces the successive elements of the generator.

```
% Generators

% Natural Numbers

nat(0).
nat(N) :- nat(M), N is M + 1.

% Infinite sequence from I

inf(I,I).
inf(I,N) :- I1 is I+1, inf(I1,N).

% An Alternate definition of natural numbers (more efficient)
alt_nat(N) :- inf(0,N).

% Sequence of Squares

square(N) :- alt_nat(X), N is X*X.

% Infinite Arithmetic Series

inf(I,J,I) :- I =< J.
inf(I,J,N) :- I < J, I1 is J + (J-I), inf(J,I1,N).

inf(I,J,I) :- I > J.
inf(I,J,N) :- I > J, I1 is J + (J-I), inf(J,I1,N).

% Finite Arithmetic Sequences

% Numbers between I and J increment by 1

between(I,J,I) :- I =< J.
between(I,J,N) :- I < J, I1 is I+1, between(I1,J,N).

between(I,J,I) :- I > J.
between(I,J,N) :- I > J, I1 is I-1, between(I1,J,N).

% Numbers between I and K increment by (J-I)

between(I,J,K,I) :- I =< K.
between(I,J,K,N) :- I < K, J1 is J + (J-I), between(J,J1,K,N).

between(I,J,K,I) :- I > K.
between(I,J,K,N) :- I > K, J1 is J + (J-I), between(J,J1,K,N).

% Infinite List -- Arithmetic Series the Prefixes
```

```
inflist(N,[N]).
inflist(N,[N|L]) :- N1 is N+1, inflist(N1,L).

% Primes -- using the sieve

prime(N) :- primes(PL), last(PL,N).

% List of Primes

primes(PL) :- inflist(2,L2), sieve(L2,PL).
sieve([],[]).
sieve([P|L],[P|IDL]) :- sieveP(P,L,PL), sieve(PL,IDL).

sieveP(P,[],[]).
sieveP(P,[N|L],[N|IDL]) :- N mod P  >  0, sieveP(P,L,IDL).
sieveP(P,[N|L],   IDL)  :- N mod P =:= 0, L \= [], sieveP(P,L,IDL).

last([N],N).
last([H|T],N) :- last(T,N).

% Primes -- using the sieve (no list)

sprime(N) :- inflist(2,L2), ssieve(L2,N).

ssieve([P],P).
ssieve([P|L],NP) :- L \= [], sieveP(P,L,PL), ssieve(PL,NP).

% B-Tree Generator -- Inorder traversal (Order important)
traverse(btree(Item,LB,RB),I) :- traverse(LB,I).
traverse(btree(Item,LB,RB),Item).
traverse(btree(Item,LB,RB),I) :- traverse(RB,I).
```

9.3 Sequencers

There are several common features of imperative programming languages that tend to make reasoning about the program difficult. The **goto** command [9] breaks the sequential continuity of the program. When 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 upto 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 handers 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 languge 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 flexibilty. 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.

9.4 Jumps

A *jump* is an explicit transfer of control from one point in a progam to another program point. Jumps come in unconditional and conditional forms:

```
goto L
if conditional expression goto L
```

At the machine level alternation and iteration may be implmented using **labels** and **goto** commands. Goto commands often take two forms:

1. *Unconditional* **goto**. The unconditional goto command has the form:

   goto $\texttt{LABEL}_i$

   The sequence of instructions next executed begin with the command labeled with $\texttt{LABEL}_i$.

2. *Conditional* **goto**. The conditional goto command has the form:

   if *conditional expression* then goto $\texttt{LABEL}_i$

   If the conditional expression is true then execution transfers to the sequence of commands headed by the command labeled with $\texttt{LABEL}_i$ otherwise it continues with the command following the conditional goto.

The goto commands have a number of advantages, they have direct hardware support and are completely general purpose. There are also a number of disadvantages, programs are flat without hierarchical structure and the code may be difficult to read and understand.

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.

9.5 Escape

An *escape* is a sequence which terminates the execution of a textually enclosing construct.

An escape of the form:

```
return expr
```

is often used to exit a function call and return the value computed by the function.

An escape of the form:

```
exit(n)
```

is used to exit `n` enclosing constructs. The exit command can be used in conjunction with a general loop command to produce `while` and `repeat` as well as more general looping constructs.

In C a `break` command sends control out of the enclosing loop to the command following the loop while the `continue` command transfers control to the beginning of the enclosing loop.

9.6 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 ϵ of zero.

Definition 9.3 *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 constuction of *robust* programs. A program is said to be *robust* if it recovers from exceptional conditions.

Definition 9.4 *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 occured 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
3. Propagation of Exception

Productive Use of Failure

Prolog, Icon

9.7 Coroutines

Multiple threads of control but control is passed from thread to thread under the active thread's control.

9.8 Processes

Multiple threads of control where each thread may be active concurrently.

9.9 Side effects

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:

```
function f(x:integer) :  integer;
   begin y := y+1; f := y + x end
```

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. 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 in the program,

```
1 + f(z) = f(z) + f(z)
```

it will be false.

As another example of side effects, consider the C function *getint* as used in the following two expressions.

```
2 * getint ()
getint () + getint ()
```

The two expressions are different. 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. And yet as mathematical expressions they should denote the same value.

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.

9.10 Aliasing

Two variables 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.

The following assignments appear to be independent of each other.

```
x := a + b
y := c + d
```

But suppose `x` and `c` are aliases for the same object. In this case, the assignments are interdependent and the order of evaluation is important. Often in the optimization of code it is desireable to reorder steps or to delete unecessary steps. This cannot be done when aliasing is possible.

Aliasing can occur in several ways.

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.

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. Another way in which aliasing can occur is when a data object may be a component of several data objects (referenced through pointer linkages).

Aliasing can arise from variable definitions and from variable parameters. Consider calls `confuse( i, i)` and `confuse( a[i], a[j] )` given the following

Pascal procedure

```
procedure confuse (var m, n :  integer);
   begin n := 1; n := m + n end
```

in the first call `i` is set to 2. The result of the second call depends on the values of `i` and `j`. The second call shows that the detection of aliasing may be delayed until run time. No compiler can detect aliasing in general.

- 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.

```
var p, q :  ↑ T;
...
new(p);
q := p
```

Dangling References

```
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.

Imperative constructs jeopardize many of the fundamental techniques for reasoning about mathematical objects. 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.

9.11 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 motivativation to provide functional and logic programming as alternatives to the imperative programming paradigm.

9.12 Expressions with side effects

Most imperative programming languges permit expressions to have side effects.

9.13 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.

```
i := 1;
while (i <= length) and (list[i] <> value) do i := i+1
```

```
command := |IDENT := exp
           |label :  command
           | GOTO label
           | IF boo_exp THEN GOTO label
           | command; command
```

Figure 9.1: A set of structured commands

```
command := SKIP
         | IDENT := exp
         | IF guarded_command alternative ...  FI
         | DO guarded_command alternative ...  OD
         | command; command
guarded_command  := guard  → command
alternative  := [] guarded_command
```

Figure 9.2: A set of structured commands

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 1 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 most implementations, if the value is 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.

9.14 Structured Programming

Given the imporatance of sequence control, it is not suprizing that considerable effort has been given to finding appropriate control structures. Figure 9.1 gives a set of basic control structures. Figure 9.2 gives a set of structured commands.

9.15 Expression-oriented languages

An expression-oriented language is an imperative languge in which distinctions between expressions and commands are eliminated. This permits a simplification in the syntax in that the language does not need both procedure and function abstraction nor conditional expressions and commands since he evaluation of an expression may both yield a value and have a side effect of updating variables. The assignment `V := E` can be defined to yield the value of the expression `E` and assign `V` to the value of `E`. Since the assignment is an expression, expressions of the form $\texttt{V}_0$ `:= ... := (`$\texttt{V}_n$ `:= E)` are possible giving multiple assigment for free. The assignment returns the 0-tuple () in ML.

The value of $\texttt{E}_0\texttt{;E}_1$ the value of $\texttt{E}_1$. $\texttt{E}_0$ is evaluated for its side-effect and the value is discarded. There is no obvious result for iterative control structures. Typically they are defined to return zero or a null value.

Expression-oriented languages achieve simplicity and regularity by eliminating the distinction between expressions and commands. Often this can result in an opaque programming style. Algol-68, Scheme, ML and C are examples of expression oriented languages.

9.16 Further Reading

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. What is the effect on the semantic descriptions if expressions are permitted to have side effects?
3. Ambiguity of Pascal's if command.
4. Compare the case command of Ada to the switch command of C++.
5. Compare the looping constructs of Ada and C++
6. Alternative control structures
7. Goto elimination
8. Axiomatic semantics
9. Denotational semantics

10. Operational semantics
11. Provide implementations of the alternative and iterative control structures in terms of labels and gotos.
12. 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

Chapter 10

Concurrent Programming

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, livelock, fairness, communication, synchronization producer-consumer, dining philosophers.

Operations are *sequential*, if they occur one after the other, ordered in time. Operations are *concurrent*, if they overlap in time. Operations in the source text of a program are *concurrent* if they could be, but need not be, executed in parallel. *Concurrent programming* involves the notations for expressing potential parallelism 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. This is the case when functional and logic programming languages are executed on appropriate hardware or a program is compiled by a parallelizing compiler. 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 concurrent operations in the source text.

PL/1

ALGOL 68

Ada

SR

10.1 Concurrency

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 singnals called *interrupts* provided the synchonization between the CPU and the I/O devices.

Interrupts together with a hardware clock made it possible to implemenent *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.

Other advances in hardware have lead to the the development of alternative ar-

chitectures. *Pipeline processors* which fetch the next instruction while the first instruction is being decoded. *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.

From the programmer's point of view, concurrent programming facilities 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.

Concurrency occurs in a programming language when two or more operations could (but need not) be executed in parallel. 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 such as data structures and hardware resources like printers. To permit the effective use of multiple processes concurrent programming languages must provide three constructs.

1. *Concurrent execution*: Constructs that denote operations that could be, but need not be, executed in parallel.

2. *Communication:* Constructs that permit processes to exchange information usually either through shared variables (visible to each process) or a message passing mechanism.

3. *Synchronization*: 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.

Aside: The terms, concurrent, distributed and parallel have a 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. □

Processes which communicate and synchronize characterize the computational paradigm based on processes. C, Scheme, Ada and Occam are just some of the programming languages that provide for processes. It is important to note that the notion of processes is orthogonal to that of inference, functions and assignments.

10.2 Issues in Concurrent Programming

The basic correctness issues in the design of concurrent programs are *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.

To illustrate the issues involved in concurrent programming we consider the dining philosophers problem.

> **The Dinning Philosophers**: Five philosophers sit at a circular table, alternating between eating spaghetti and thinking. In order to eat, a philosopher must have two forks. There is a single fork between each philosopher, so if one philosopher is eating, a neighboring philosopher cannot eat. A philosopher puts down the forks before thinking.

The problem is to construct an algorithm that permits the philosophers to 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 desireable liveness property is that whenever a philosopher wants to eat, eventually the philosopher will get to eat.

Nondeterminism

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 unpredicatable. This makes testing of concurrent programs a difficult task.

Communication

Two processes are said to communicate if an action of one process must entirely precede an action of a second process.

Synchronization

Synchronization is related to communication.

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 semiphores, critical regions and monitors.

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 desireable 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*.

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 philosoper 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 reliablility 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 weaking 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 condtion may be modified by imposing a total ordering on resources and insisting that they be requirested in that order.

Another example of a liveness problem is *livelock* (or lockout or starvation). Livelock occurs when a process is prevented from making progress (other processes are running). This is an issue of *fairness*.

10.3 Syntax

In this section we develop a notation for the specification of concurrency, communication and synchronization.

notation for explicit concurrency

$$[P_1 \parallel P_2 \parallel ... \parallel P_n]$$

notation for communication

$$P_i!E, P_j?X$$

notation for synchronization

$$wait(P_i), signal(P_j)$$

combined notation for communication and synchronization

10.4 Interfering Processes

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.

```
# multiply n by n matrices a and b in parallel

# place result in matrix c
# all matrices are global to multiply
process multiply(i := 1 to n, j := 1 to n)
  var inner_prod := 0
  fa k := 1 to n →
        inner_prod := inner_prod + a[i,k] * b[k,j]
  af
  c[i,j] := inner_prod
end
```

Figure 10.1: Matrix multiplication

10.5 Non-interfering Processes

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

$$[i := 1 \parallel j := 2]$$

do not share an address space therefore, the assignments may take place in parallel.

Another example of non-interfering processes is found in matric 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 10.1 is an example of a matrix multiplication routine written in the SR programming language. This particular example also illustrated dynamic process creation in that $\mathtt{n}^2$ processes are created to perform the multiplication.

10.6 Cooperating Processes

vs. message passing later.

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 resourse at a time. This permits processes to cooperate, sharing the resource but maintaining the integrity of the resource.

semiphore

critical section

monitor

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,head := queue[head],(head + 1) mod qsize
        end;
begin
    head,tail := 0,0;
    [ *[produce(x); enqueue(x)] ‖ *[dequeue(y); consume(y)]]
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.

10.7 Synchronizing Processes

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) ‖ dequeue(y); consume(y) ‖ 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 *deadlocked*.

The solution is to put the waiting processes on a queue.

```
monitor Queue_ADT
```

```
    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];
            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.
```

Busy waiting is no longer involved.

Livelock may result if there are more than one waiting process and when the signal is received access is not granted fairly.

Starvation: (livelock) multiple processes waiting for access but access is not provided in a fair manner

Coroutines.

Real-time Programming language issues

10.8 Communicating Processes

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.

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.

```
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.
```

10.9 Occam

10.10 Semantics

Parallel processes must be...

1. Synchronization-coordination of tasks which are not completely indepen-

dent.

2. Communication-exchange of information
3. Scheduling-priority,
4. Nondeterminism-arbitrary selection of execution path

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

10.11 Related issues

Lazy evaluation vs Parallel execution

Backtracking vs Parallel execution

10.12 Examples

Pipelines

unix, pipeline sort, sieve

Systolic arrays

Dining Philosophers

10.13 Further Reading

Exercises

1. independent
2. pipe line

3. synchronized

Chapter 11

PCN

11.1 Tutorial

Program Composition

Concurrent Programming Concepts

Getting Started

Example Program - compiling, linking, running

11.2 The PCN Language

PCN Syntax

Sequential Composition and Mutable Variables

Parallel Composition and Definitional Variables

Choice Composition

Definitional Variables as Communication Channels

Lists and Tuples

Stream Communication

Advanced Stream Handling

11.3 Examples

Exercises

Chapter 12

Abstraction and Generalization II

Encapsulate—To completely enclose.

Keywords and phrases: Modularity, encapsulation, function, procedure, abstract type, generic, library, object, class, inheritance, partition, package, unit, separate compilation, linking, import, export, instance, scope.

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

12.1 Encapsulation

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

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 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.

In this section we consider various language design considerations which can be used as weapons in the war against complexity, the pragmatics of programming. In particular, we discuss scope, modularity.

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

12.2 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.

```
name :  adt
operation signatures
...

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

12.3 Partitions

Definition 12.1 *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
- 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.

12.4 Scope Rules

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

The scope rules for modules define relationships among the names within the modules. 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 module name.

12.5 Modules

A *module* is a named program unit which is an (more or less) independent entity. 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 reuseable (able to be incorporated into may programs) and modifiable with out forcing changes in other modules.

Modules must provide answers to two questions:

- *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.

Exercises

1. Algebraic Semantics: stack, tree, queue, grade book etc

2. Lexical Scope Rules
3. Dynamic Scope Rules
4. Parameter Passing
5. Run-time Stack

Chapter 13

Object-Oriented Programming

Object-oriented programming is characterized by programming with objects, messages, and hierarchies of objects.

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

- History
  - Simula
  - ADT
  - Small-Talk
  - Modula-2, C++, Eiffel
- Subtypes (subranges)
- Generic types
- Inheritance – Scope generalization

- OOP
  - Objects–state + operations
  - Object Classes– Class, Subclass

- Objects–state + operations
- Object Classes– Class, Subclass
- Inheritance mechanism

Object-oriented programming shifts the emphasis from data as passive elements acted on by procedures to data as active elements interacting with their environment. From control flow to interacting objects.

Object-oriented programming developed out of simulation programs. The conceptual model used is that the structure of the simulation should reflect the environement 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.

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.

Aternatively, 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.

Definition 13.1

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

13.1 History

- Simula
- Small-Talk
- Modula-2, C++, Eiffel
- Flavors, CLOS

13.2 Subtypes (subranges)

The subtype principle states that a subtype may appear wherever an element of the super type is expected.

13.3 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:

```
name : 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:

```
name : object
variables
...
methods
...
```

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.

13.4 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.

13.5 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.

13.6 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 13.1 *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 13.2 *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 13.3 *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 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 distingushes 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 necessaryly 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.

13.7 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,θ)

13.8 Further Reading

Much of this section follows Peter Wegner[31].

13.9 Exercises

- Stack
- Queue
- Tree
- Construct a "turtle graphics"
- Construct a table handler
- Grammar
- Prime number sieve
- Account, Checking, Savings
- Point, circle

Chapter 14

Pragmatics

14.1 Syntax

In view of abstract syntax it may seem that concrete syntax does not matter. In fact, details such as placement of keywords, semicolons and case do matter.

In Pascal the semicolon is a statement separator. In C the semicolon is a statement terminator. Pascal permits the empty statement. This may lead to unintended results. A misplaced semicolon can change the meaning of a program.

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

`if` $< \mathit{conditionalexpression} >$ `then` $< \mathit{command} >$ `fi`

The assignment operator varies among imperative programming languages. In Pascal and Ada it is `:=` while in FORTRAN and C it is `=` and in APL it is $\leftarrow$. The choice in FORTRAN and C is unfortunate since assignment is different from equality. This leads to the use of `==` for equality in C and `.EQ.` in FORTRAN.

14.2 Semantics

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.

For effective use axiomatic semantics require support for program varification.

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 –

14.3 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. For example, the *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. *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. *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 while late binding permits more flexibility. The implementation of recursion may require allocation of memory at run-time in contrast a one time to allocation of memory at compile-time.

14.4 Values and Types

The primitive types are implemented using both the underlying hardware and special purpose software. So that only appropriate operations are applied to values, the value's type must be known. The type information is contained in a descriptor. When the type of a value is known at compile time the type descriptor is a part of the symbol table and is not needed at run-time and therefore, the descriptor is discarded after compilation. When the type 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.

Integer values are most often implemented using a hardware defined integer storage representation. 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 roundoff 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. 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.

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.

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 manipuation is restricted to the implementation module and the pointer is notationally hidden.

14.5 Computational Models

14.6 Procedures and Functions

In the discussion which follows, the term *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 *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 destroyed when the subprogram terminates.

The run time 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*.

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.

14.7 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.

Dynamic 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.

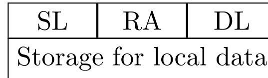
A reference to a variable involves the inspection or updating of the top element of the appropriate stack. This provides access to the variable in closest block with respect to the dynamic calling sequence.

When a subprogram terminates, the stacks corresponding to the variables of the block are popped, restoring the calling environment.

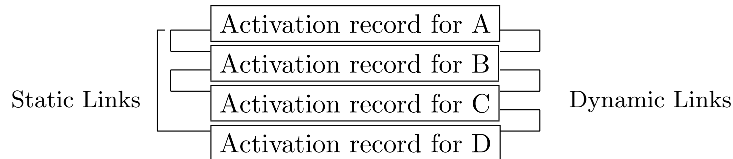
Static Scope Rules

The static scope rules are 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 it 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.

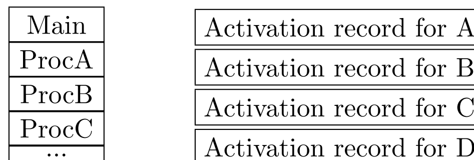


The runtime stack of activation records (local data, static and dynamic links).



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.



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 mannner.

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 accessable.

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 inaccessable 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. 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: Ub the collecting phase the heap is scanned and each element which is not active is returned to the free-space list. During this phase the marked bits are reset to prepare for a later garbage collection.

This unuseable space may be reclamed 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 it own activation record. The address of the *CI* for the resumed coroutine is obtained from the activation record of the resumed coroutine.

Concurrency

14.8 Parameters and Arguments

In the previous sections, the arguments to an invocation are textually substituted for the parameters. That is, the body of the abstract is rewritten with the arguments substituted for the parameters.

Eager vs Lazy evaluation

An abstraction 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 it is not sure to require the value of the parameter. The arithmetic operators are strict but the conditional expression 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 first argument.

The programming language Scheme assumes that all functions are strict in their parameters, therefore, the parameters are evaluated when the function is called. This evaluation scheme is called *eager evaluation.* This is not always desirable and so Scheme provides for the quote operator to inform a function not to evaluate its parameters. Miranda evaluates the arguments only when the value is required. This evaluation scheme is called *normal-order evaluation* or *lazy evaluation.*

Argument Passing Mechanisms

With respect to functional and logic programming languages, the question of how to pass parameters is relatively simple. The imperative programming paradigm is another issue.

Copy Mechanisms

The **copy** mechanism requires values to be copied into an abstraction when it is entered and copied out of the abstraction when the abstraction is exited. The formal parameters are local variables. Therefore, the actual parameter is copied into the local variable on entry to the abstraction and copied out of the local variable to the actual parameter on exit from the abstraction.

The parameters of C++, the *value parameter* of Modula-2 and the *in parameter* of Ada are examples of parameters which may be passed by using the copy mechanism. The value of the actual parameter is copied into the

formal 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. This form of parameter passing is often referred to as **passing by value**

Ada's *out parameter*, and function results in general are examples of parameters which may be passed by using the copy mechanism. The value of the actual parameter is not copied into the formal parameter on entry but the value of the formal parameter is copied into the actual parameter upon exit. In Pascal the function name is used as the formal 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 (eg file types in Pascal are not assignable).

Definitional Mechanisms

The effect is as if the abstraction body were surrounded by a block, in which there is a definition that binds the formal parameter to the actual parameter.

The *reference parameter* of Modula-2, the array and structure parameters of C++ are passed using this mechanism.

Name

Algol-60 provides a parameter passing mechanism which is based on that of the functional model however it does not provide the generality that is required in the imperative model as the following example shows.

Algol-60, Jensen's device

```
procedure swap(x,y:sometype);
var x:sometype
begin
```

```
t := x; x := y; y := t
end;
...
I := 1
a[I] := 3
swap(I,a[I])
```

Parameter passing by name with assignment and when there are two parameters one of which references the other.

aliasing

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

Unification

14.9 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 mispell 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 privious 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.

type checking–static, dynamic

import/export

Declarations and strong type checking facilitate safety by providing redundancy. When the programmar 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.

14.10 Further Reading

14.11 Exercises

Chapter 15

Translation

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. This chapter shows how context-free grammars are used in the construction of language translators. Since the translation is base 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 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 *semantic analysis phase* analyzes 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.

- 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.

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.

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.

15.1 Parsing

15.2 Scanning

15.3 The Symbol Table

15.4 Virtual Computers

tack machine

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 Pascal computer or a Scheme computer. For the programmer, the programming language is the computer; the programming language defines a virtual computer. The virtual machine for Simp consists of a data area which contains the association between variables and values and the program which manipulates the data area.

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 Simp 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.

15.5 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;  --> I := J - 1;
or
 I := 3; J := I + 2;  --> I := 3; 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
          A[I] := A[I] + E
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
          A[I] := A[I] + E
```

Common subexpression elimination:

```
From:
      A := 6 * (B+C);
      D := 3 + 7 * (B+C);
      E := A * (B+C);
to:
      TEMP := B + C;
      A    := 6 * TEMP;
      D    := 3 * 7 * TEMP;
      E    := A * TEMP;
```

Strength reduction:

```
2*x  --> x + x
2*x  --> shift left x
```

Mathematical identities:

```
a*b + a*c --> a*(b+c)
a - b --> a + ( - b )
```

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

15.6 Code Generation

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”.

As an example of code generation, we extend our Lex and Yacc files for Simp to generate code for a stack machine. First, we must extend the Yacc and Lex files to pass the values of constants from the scanner to the parser. The definition of the semantic record in the Yacc file is modified that the constant may be returned as part of the semantic record.

```
%union semrec         /* The Semantic Records              */
{
 int     intval;      /* Integer values                    */
 char    *id;         /* Identifiers                       */
...
```

Then the Lex file is extended to place the value of the constant into the semantic record.

```
%{
#include <string.h>     /* for strdup                        */
#include <stdlib.h>     /* for atoi                          */
#include "simple.tab.h" /* for token definitions and yylval */
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
{DIGIT}+ { yylval.intval = atoi( yytext );
           return(INT);      }
...
{ID}     { yylval.id = (char *) strdup(yytext);
           return(IDENT);    }
[ \t\n]+ /* eat up whitespace */
.        { return(yytext[0]);}
%%
```

The symbol table record is extended to contain the offset from the base address of the data segment (the storage area which is to contain the values associated with each variable) and the `putsym` function is extended to place the offset into the record associated with the variable.

```
struct symrec
{
  char *name;                     /* name of symbol      */
  int offset;                     /* data offset         */
  struct symrec *next;            /* link field          */
};
typedef struct symrec symrec;
symrec *sym_table = (symrec *)0; /* Ptr to symbol table */
symrec *st_entry;                 /* Ptr to an entry     */
symrec *putsym ();
symrec *getsym ();
symrec *
putsym (sym_name)
```

```
     char *sym_name;
{
  symrec *ptr;
  ptr = (symrec *) malloc (sizeof(symrec));
  ptr->name = (char *) malloc (strlen(sym_name)+1);
  strcpy (ptr->name,sym_name);
  ptr->offset = data_offset++;
  ptr->next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}
...
```

The parser is extended to generate and assembly code. The code implementing the if and while commands must contain the correct jump addresses. In this example, the jump destinations are labels. Since the destinations are not known until the entire command is processed, *back-patching* of the destination information is required. In this example, the label identifier is generated when it is known that an address is required. The label is placed into the code when its position is known. An alternative solution is to store the code in an array and back-patch actual addresses.

```
%{
#include <stdio.h>    /* For I/O                             */
#include <stdlib.h>   /* for malloc here and in symbol table */
#include <string.h>   /* for strcmp in symbol table          */
int data_offset = 0;  /* for data area address offsets       */
int label_count = 0;  /* for label identifiers               */
#include "ST.h"
#define YYDEBUG 1
%}
%union semrec         /* The Semantic Records                */
{
 int     intval;      /* Integer values                      */
 char    *id;         /* Identifiers                         */
 struct  lbs          /* Labels for if and while             */
 {
  int label0;
  int label1;
  }   *lbls;
}
%token <intval>  INT      /* Simple integer                  */
%token <id>      IDENT    /* Simple identifier               */
%token <lbls>    IF WHILE /* For back-patching labels        */
```

The semantic record is extended to hold two label identifiers since two labels will be required for the if and while commands.

The remainder of the file contains the actions associated with code generation for a stack-machine based architecture.

```
%token SKIP THEN ELSE FI DO END
%left '-' '+'
%left '*' '/'
%right '^'    /* Exponentiation        */
%%
program : command_sequence { /* print code */ }
;
command_sequence : /* empty */
   | command_sequence command ';'
;
command : SKIP
   | IDENT ':' '=' exp { install( $1, IDENT );
             printf("assign %ld\n", st_entry->offset );      }

   | IF exp { $1 = (struct lbs *) malloc(sizeof(struct lbs));
              $1->label0 = label_count++;
              printf("jmp_false label %ld\n",$1->label0);    }
     THEN command_sequence { $1->label1 = label_count++;
              printf("goto label %ld\n",$1->label1);         }
     ELSE        { printf("label %ld\n",$1->label0);         }
          command_sequence
     FI          { printf("label %ld\n",$1->label1);         }

   | WHILE  { $1 = (struct lbs *) malloc(sizeof(struct lbs));
              $1->label0 = label_count++;
              printf("label %ld\n", $1->label0);             }
        exp { $1->label1 = label_count++;
              printf("jmp_false label %ld\n",$1->label1);    }
     DO
        command_sequence
     END    { printf("goto label%ld\n",$1->label0);
              printf("label%ld\n",$1->label1);               }
;
exp : INT          { printf("load_int %ld\n", $1);           }
   | IDENT { if (getsym($1) == 0)
               printf("Undefined variable: %s\n", $1);
             else
               printf("load_var %ld\n", getsym($1)->offset); }
   | exp '<' exp  { printf("less_than%\n");                  }
   | exp '=' exp  { printf("equal%\n");                      }
```

```
n := 1;
if n < 10 then x := 1; else skip; fi;
while n < 10 do x := 5; n := n+1; end;
skip;
```

Figure 15.1: A Simp program

```
   | exp '>' exp  { printf("greater_than%\n");            }

   | exp '+' exp  { printf("add%\n");                     }
   | exp '-' exp  { printf("sub%\n");                     }
   | exp '*' exp  { printf("mult%\n");                    }
   | exp '/' exp  { printf("div%\n");                     }
   | exp '^' exp  { printf("power%\n");                   }
   | '(' exp ')'
;
%%
...
```

To illustrate the code generation capabilities of the compiler, Figure 15.1 is a program in Simp and Figure 15.2.

15.7 Peephole Optimization

Following code generation there are further optimizations that are possible. The code is scanned a few instructions at a time (the peephole) looking for combinations of instructions that may be replaced by more efficient combinations. Typical optimizations performed by a peephole optimizer include copy propagation across register loads and stores, strength reduction in arithmetic operators and memory access, and branch chaining.

We do not illustrate a peephole optimizer for Simp.

15.8 Further Reading

For information on compiler construction using Lex and Yacc see[27]. Pratt [24] emphasizes virtual machines.

```
load_int 1
assign 0
load_var 0
load_int 10
less_than
jmp_false label 0
load_int 1
assign 1
goto label 1
label 0
label 1
label 2
load_var 0
load_int 10
less_than
jmp_false label 3
load_int 5
assign 1
load_var 0
load_int 1
add
assign 0
goto label 2
label 3
```

Figure 15.2: Stack code

Chapter 16

Evaluation of Programming Languages

16.1 Models of Computation

The first requirement for a general purpose programming languge 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 *implementatable* on a computer.

Functional Programming:

Logic Programming:

Imperative Programming:

Object-Oriented Programming:

Concurrent Programming:

16.2 Syntax

1 Principle of Simplicity: *The language should be based upon as few "basic concepts" as possible.*

2 Principle of Orthogonality: *Independent functions should be controlled by independent mechanisms.*

3 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. The basic concepts of the language should be applied consistently and universally.*

4 Principle of 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.*

5 Principle of Parameterization: *A formal parameter to an abstract may be from any syntactic class.*

6 Principle of 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.*

7 Principle of Correspondence: *For each form of definition there exists a corresponding parameter mechanism and* vice versa*.*

16.3 Semantics

8 Principle of Clarity: *The mechanisms used by the language should be well defined, and the outcome of a particular section of code easily predicted.*

9 Principle of 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).*

10 Principle of Sub-types: *A sub-type may appear wherever an element of the super-type is expected.*

16.4 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

11 Principle of Expressivity: *The language should allow us to construct as wide a variety of programs as possible.*

12 Principle of 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

13 Principle of Safety: *Mechanisms should be available to allow errors to be detected.*

Type checking-static and dynamic, range checking

14 Principle of 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.*

15 Principle of Information Hiding: *Each "basic program unit" should only have access to the information that it requires.*

16 Principle of Explicit Interfaces: *Interfaces between basic program units should be stated explicitly.*

17 Principle of Privacy: *The private members of a class are inaccessible from code outside the class.*

Abstraction

18 Principle of 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.*

19 Principle of 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.*

20 Principle of 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

21 Principle of 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

22 Principle of 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.*

23 Principle of 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.

16.5 Trends in Programming Language Design

streams, lazy evaluation, reactive systems, knowledge based systems, concurrency, efficient logic and functional languages, OOP.

Chapter 17

History

17.1 Functional Programming

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 inadvertantly 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 aritficial 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.

17.2 Logic Programming

1969 J Robinson and Resolution 1972 Alain Colmerauer

17.3 Imperative Programming

Imperative languages have a rich and varied history. The first imperative programming languages were machine instructions. Machine instructions were soon replaced with Assembly languages, essentially transliterations of machine code.

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.

PL/I (Programming Languge 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 consdered in chapter 7). 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.

Simula 67

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 the 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 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 is an attempt to provide an efficient language for systems programming.

Modula-2

Ada was developed as the result of a Department of Defense initiative. Like PL/1 and Algol-68, Ada represents an attempt to produce a complete langauge representing the full range of programming tasks.

17.4 Concurrent Programming

17.5 Object-Oriented Programming

Appendix A

Logic

A.1 Sentential Logic

A.1.1 Syntax

The formulas of sentential logic are defined as follows.

1. true, false, P_0, P_1... are (atomic) formulas.

2. If A and B are formulas, than the following are (compound) formulas ¬A, A ∧ B, A ∨ B, A → B, A ↔ B.

The compound formulas of sentential logic (with the exception of the negation of an atomic formula) are classified as of type α with subformulas α_1 and α_2

α	α_1	α_2
$\neg\neg A$	A	A
$A \wedge B$	A	B
$\neg(A \vee B)$	$\neg A$	$\neg B$
$\neg(A \to B)$	A	$\neg B$
$A \leftrightarrow B$	$A \to B$	$B \to A$

or of type β with subformulas β_1 and β_2 as follows.

β	β_1	β_2
$A \vee B$	A	B
$\neg(A \wedge B)$	$\neg A$	$\neg B$
$A \rightarrow B$	$\neg A$	B
$\neg(A \leftrightarrow B)$	$\neg(A \rightarrow B)$	$\neg(B \rightarrow A)$

The formulas of sentential logic are often called: sentential formulas, sentential expressions, propositional formulas, propositional expressions, or simply sentence or proposition.

A.1.2 Semantics

The semantics of sentential logic are the rules for classifying a sentence as true or false. The semantic rules that we give here are analytic rules because the truth of a compound formula is determined by the truth of its subformulas. A type α formula is classified as true iff both of its subformulas are true. A type β formula classified as false iff one of its subformulas is true. Here are some essential definitions.

Definition A.1 *A* **formal system** *is a set of formulas.*

Definition A.2 *An* **axiom** *is a sentence given as true.*

Definition A.3 *A formal system is* **complete** *iff every sentence is either true or false.*

Definition A.4 *A formal system is* **consistent** *iff no sentence is both true and false.*

Definition A.5 *A sentence that is true reguardless of the classification of its subformulas is called a* **tautology***.*

Definition A.6 *A sentence that can be true for some classification of its subformulas is called* **satisfiable***.*

Definition A.7 *A sentence that is false reguardless of the classification of its subformulas is called a* **contradiction***.*

Definition A.8 *A* **theorem** *is a sentence that is true either because it is a tautology or by inference from axioms.*

Figure A.1: **Configuration Reduction Rules for Sentential Logic**

A: $$\frac{\mathcal{C}\ \boxed{\text{S}, \alpha}}{\mathcal{C}\ \boxed{\text{S}, \alpha_1, \alpha_2}}$$

B: $$\frac{\mathcal{C}\ \boxed{\text{S},\beta}}{\mathcal{C}\ \boxed{\text{S},\beta_1}\ ,\ \boxed{\text{S},\beta_2}}$$

where $\mathcal{C}$ is the rest of the configuration, S is the set of the rest of the elements of the block, and α and β indicate the type of the compound formula.

Definition A.9 *A* **model** *is a classification of sentences such that each sentence is either true or false (not both).*

These definitions and the α and β rules form the base for the method of proof using analytic tableaux. The method involves searching for contradictions among the formulas generated by application of the analytic properties.

Definition A.10 *By a* **configuration** $\mathcal{C}$ *we shall mean a finite collection of finite sets of sentences.*

If $\mathcal{C} = \{B_1, ..., B_n\}$ we also write $\mathcal{C}$ in the form

$$\boxed{B_1}\ ,...,\ \boxed{B_n}$$

and we refer to the elements $B_1, ..., B_n$ of $\mathcal{C}$ as the *blocks* of the configuration.

Definition A.11 $\mathcal{C}_2$ *is a* **reduction** *of* $\mathcal{C}_1$ *if* $\mathcal{C}_2$ *can be obtained from* $\mathcal{C}_2$ *by finitely many applications of configuration reduction rules.*

The configuration reduction rules have the form

$$\frac{\mathcal{C}\ \boxed{B_i}}{\mathcal{C}\ \boxed{B'_i}}$$

The rule is read as "replace block B_i in the configuration with block B'_i." The configuration reduction rules are based on the analytic properties and are found in Table A.1. Each reduction rule corresponds to one of the analytic

properties. Given a block with a formula of type α or β the reduction rules specify the replacement of a block with one or more blocks in which the formula is replaced with its subformulas. For example, rule A permits the replacement of a conjunction with the conjuncts and rule B requires the block to be replaced with two blocks each containing one of the disjuncts.

The configuration reduction rules may be used to construct a model under which a given formula is satisfiable. For example, the configuration reduction rules generate the following sequence of configurations given the propositional formula, $\neg[(p \vee q) \rightarrow (p \wedge q)]$.

$$\boxed{\neg[(p \vee q) \rightarrow (p \wedge q)]}$$

$$\boxed{(p \vee q),\ \neg(p \wedge q)}$$

$$\boxed{p,\ \neg(p \wedge q)}\ ,\ \boxed{q,\ \neg(p \wedge q)}$$

$$\boxed{p,\ \neg p}\ ,\ \boxed{p,\ \neg q}\ ,\ \boxed{q,\ \neg p}\ ,\ \boxed{q,\ \neg q}$$

Note that the outer two blocks of the last line are contradictory and the inner two define an interpretation under which the formula is satisfiable.

A.2 Predicate Logic

Predicate Logic (or Predicate Calculus or First-Order Logic) is a generalization of Sentential Logic. Generalization requires the introduction of variables and the variables of Predicate Logic are of two kinds: free and bound.

A.2.1 Syntax

1. A sentence is also a predicate with no free variables.
2. If p is a predicate and x is a variable but not a variable in p and T is a type then the following are also predicates. $\forall x.T : p(x)$ and $\exists x.T : p(x)$ which have the same free variables as p but have in addition the bound variable x. $p(x)$ is formed from p by replacing any number of occurrences of some constant of type T in p with x. x is said to be free in $p(x)$ and is said to be bound in $\forall x.T : p(x)$ and $\exists x.T : p(x)$.

The additional formulas of Predicate Logic are compound formulas and the universally quantified formulas are of type γ with subformula $\gamma(c)$. A type γ formula holds iff its subformula holds for each constant of the appropriate type.

Figure A.2: **Configuration Reduction Rules for Predicate Logic**

C: $\dfrac{\mathcal{C}\ \boxed{\text{S},\gamma}}{\mathcal{C}\ \boxed{\text{S},\gamma(c),\gamma}}$ some constant c

D: $\dfrac{\mathcal{C}\ \boxed{\text{S},\delta}}{\mathcal{C}\ \boxed{\text{S},\delta(c)}}$ a constant c new to $\mathcal{C}\ \boxed{\text{S},\delta}$

where $\mathcal{C}$ is the rest of the configuration, S is the set of the rest of the elements of the block, and γ and δ are the types of compound formulas.

γ	$\gamma(c)$
$\forall x.T : P(x)$	$P(c)$
$\neg\exists x.T : P(x)$	$\neg P(c)$

Existentially quantified formulas are of type δ with subformula $\delta(c)$.

δ	$\delta(c)$
$\exists x.T : P(x)$	$P(c)$
$\neg\forall x.T : P(x)$	$\neg P(c)$

A.2.2 Semantics

A type γ formula holds iff its subformula holds for each constant in the universe of discourse. A type δ formula holds iff its subformula holds for some constant in the universe of discourse. Here are some essential definitions. The configuration reduction rules for these formulas are based on the analytic properties and are found in Table A.2.

Bibliography

[1] Abelson, H., Sussman, G.J., and Sussman, J. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[2] Backus, J. W., "Can Programming Be Liberated from the von Neumann Style?" CACM, vol. 21, no. 8, pp. 613-614.

[3] Barendregt, H. P., *The Lambda Calculus: Its Syntax and Semantics.* 2d ed. North-Holland, 1984.

[4] Bird, R.A. and Wadler, P.L., *Introduction to Functional Programming.* Prentice/Hall International, 1988.

[5] Boehm, C. and Jacopini, G., "Flow Diagrams, Turnign Machines, and Languages with Only Two Foramation Rules." CACM, vol. 9, no. 5, pp. 366-371.

[6] Curry. H. B. and Feys, R., *Combinatory Logic*, Vol. I. North-Holland, 1968.

[7] Curry. H. B., Hindley, J. R., and Seldin, J. P., *Combinatory Logic*, Vol. II. North-Holland, 1972.

[8] Deransart, P., Jourdan, M., and Lorho, B., *Attribute Grammars: Definitions, Systems and Bibliography.* Lecture Notes in Computer Science 323. Springer-Verlag, 1988.

[9] Dijkstra, E. W., "Goto Statement Considered Harmful." Communications of the ACM vol. 11 no. 5 (May 1968): pp. 147-149.

[10] Gries, D., *The Science of Programming* Springer-Verlag, New York, 1981.

[11] Hehner, E. C. R., *The Logic of Programming.* Prentice/Hall International, 1984.

[12] Henderson, Peter, *Functional Programming: Application and Implementation.* Prentice/Hall International, 1980.

[13] Hindley, J. R., and Seldin, J. P., *Introduction to Combinators and λ Calculus*, Cambridge University Press, London, 1986.

[14] Hopcroft, J. E. and Ullman, J. D., *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[15] Knuth, D. E., "Semantics of context-free languages." Mathematical Systems Theory, vol. 2, 1968, pp. 127-145. Correction in Mathematical Systems Theory, vol. 5, 1971, p. 95.

[16] Kowalski, R. A., "Algorithm = Logic + Control". CACM vol. 22 no. 7, pp. 424-436, 1979.

[17] Landin, P. J., The next 700 programming languages, *Communications of the ACM* 9, 157-64 1966.

[18] McCarthy, J., "Recursive functions of symbolic expressions and their computation by machine, Part I." CACM vol. 3 no. 4, pp. 184-195, 10, 1960.

[19] McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., and Levin, M., *LISP 1.5 Programmer's Manual*. 2d ed. MIT Press, Cambridge, MA. 1965.

[20] MacLennan, Bruce J., *Functional programming: practice and theory*. Addison-Wesley Publishing Company, Inc. 1990.

[21] Miller, G. A., *The Psychology of Communication*. Basic Books, New York, 1967.

[22] Peyton Jones, Simon L., *The Implementation of Functional Programming Languages*. Prentice/Hall International, 1987.

[23] Pittman, T. and Peters, J., *The Art of Compiler Design: Theory and Practice*. Prentice-Hall, 1992.

[24] Pratt, T. W., *Programming Languages: Design and Implementation*. Printice-Hall, 1984.

[25] Révész, G, E., *Lambda-Calculus, Combinators, and Functional Programming*. Cambridge University Press, Cambridge, 1988.

[26] Schmidt, D. A., *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown, Dubuque, Iowa, 1988.

[27] Schreiner, A. T. and Freidman, H. G., *Introduction to Compiler Construction with Unix* Prentice-Hall, 1985.

[28] Scott, D. S., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1987.

[29] Steele, G. L., Jr., *Common Lisp.* Digital Press, Burlington, MA. 1984.

[30] Tennent, R. D., *Principles of Programming Languages*, Prentice-Hall International, 1981.

[31] Wegner, Peter, "Concepts and Paradigms of Object-Oriented Programming." OOPS Messenger vol. 1 no. 1 (August 1990): pp. 7-87.

[32] Worf, Benjamin, *Language thought and reality,* MIT Press, Cambridge Mass., 1956.