CSC3403

Comparative Programming Languages

Faculty of Sciences

Study Book

Published by

The University of Southern Queensland
Toowoomba Qld 4350
Australia

http://www.usq.edu.au/

This book was set in Computer Modern Roman and Helvetica, using the LaTeX typesetting system and a set of special macro definitions. It was produced on a IBM compatible PC running under RedHat Linux.

Preface

This book describes how to study the course CSC3403 Comparative Programming Languages. This course uses much of the material from the text, *Concepts of Programming Languages*, seventh edition, by Robert Sebesta. Each module in this book describes essential topics of study and directs reading from Sebesta.

You should study each module as directed using the study timetable in the accompanying Introductory Book. Work through each module alternating reading from the Study Book with reading from Sebesta where indicated. Complete all the exercises suggested and check your solutions where a worked solution is provided.

Note also that extra material on functional programming, and particularly the Haskell language, are available in the Laboratory Manual for this course.

During this course you will be expected to cover most of the text by Sebesta. You do *not* need to study the following:

- sections 3.4, 3.5
- sections 12.4–12.10
- chapter 13
- sections 15.4–15.6
- chapter 16

This edition of the Study Book has been updated to use the seventh edition of the *Concepts of Programming Languages* by Robert Sebesta. *All references to material from Sebesta refer to the seventh edition.*

Contents

1 Language Evaluation **1**

1.1 Why Study Concepts of Programming Languages 1

1.2 Language Evaluation Criteria 2

1.3 Language Design and Implementation 3

1.4 Evaluating language 3

1.5 Summary 3

2 Functional Programming Languages **5**

2.1 Functional Programming 5

2.2 Types and Type Checking 7

2.2.1 Type Inference 7

2.2.2 Lazy Evaluation 8

2.3 Functional vs Imperative 9

2.4 Summary 9

3 Formal Description of Languages **11**

3.1 Language Elements 11

3.2 Describing Language 13

3.3 Formal Grammars 14

3.4 Kinds of grammars 15

3.4.1 Regular grammar 16

3.4.2 Context-free grammar 16

3.4.3 Context-sensitive grammar 17

3.4.4 The Chomsky Hierarchy 17

3.5 Backus-Naur Form 18

3.6 Parse Trees and Abstract Syntax Trees 19

3.7 Summary 22

4 More Formal Methods **23**

4.1 Ambiguity . . . 23
4.1.1 Precedence and Associativity . . . 24
4.1.2 The "dangling-else" and other problems . . . 25
4.2 Alternative Notations . . . 25
4.2.1 Extended Backus-Naur Form . . . 25
4.2.2 Syntax Graphs . . . 26
4.3 Summary . . . 30

5 Lexical analysis and Parsing **31**
5.1 Lexical Analysis . . . 31
5.2 Parsers and grammars . . . 32
5.3 Top down parsing . . . 32
5.3.1 LL(1) grammars . . . 33
5.4 Bottom up parsing . . . 34
5.5 Summary . . . 35

6 Introduction to Data Types **37**
6.1 Variables . . . 37
6.1.1 Binding, Scope and Lifetime . . . 37
6.1.2 Scope and Lifetime . . . 38
6.1.3 Variable Initialisation . . . 38
6.2 Type Checking . . . 38
6.3 Primitive Data Types . . . 39
6.4 Summary . . . 40

7 Structured Data Types **43**
7.1 Ordinal Types . . . 43
7.2 Compound Types . . . 43
7.2.1 Arrays . . . 44
7.2.2 Record Types . . . 44
7.2.3 Unions . . . 45
7.3 Pointer Types . . . 45
7.3.1 Dangling Pointers and Lost Objects . . . 45
7.3.2 Implementation . . . 46
7.3.3 Garbage collection . . . 46
7.4 Summary . . . 46

8 Expressions, Statements and Flow of Control **49**

8.1 Expressions 49
8.1.1 Function Evaluation 49
8.1.2 Operator Expressions 50
8.1.3 Side Effects 50
8.1.4 Coercion 50
8.2 Assignment Statements 51
8.3 Sequence Control 51
8.3.1 Selection Statements 52
8.3.2 Iterative Statements 52
8.3.3 Explicit Sequence Control 52
8.3.4 Limited GOTOs 53

9 Subprograms and Parameter Passing 57
9.1 Subprograms 57
9.2 Parameter Passing Methods 58
9.3 Program Composition 59
9.3.1 Overloaded Operators/Subprograms 60
9.3.2 Modules and Packages 60
9.3.3 Generic Subprograms 61
9.3.4 Separate Compilation 61
9.4 Implementing Subprograms 62
9.4.1 Procedure Call and Return 62
9.5 Dynamic Scoping 63
9.6 Summary 63

10 Abstract Data Types 67
10.1 Data Abstraction 67

11 Object Oriented Languages 69
11.1 ADTs & OOP 69
11.1.1 Classes and Typing 70
11.2 Inheritance 70
11.2.1 Single vs multiple inheritance 71
11.2.2 Dynamic Binding 71
11.3 Garbage Collection 72
11.4 Summary 72

12 Exception Processing and Event Handling 75

12.1 Error Conditions 75
12.1.1 Signals 76
12.2 Exception Processing 76
12.3 Exception Handlers 78
12.4 Event Handling 78
12.5 Summary 78

List of Figures

3.1 Parse tree for an English sentence 15
3.2 Parse tree for the expression `A := B * ( A + C )` 19
3.3 Abstract syntax tree for the expression `A := B * ( A + C )` 20
3.4 Changes in the robot's position 21

11.1 Dynamic binding in C++ 71

Module 1

Language Evaluation

Overview

As the title of this course suggests, Comparative Programming Languages is all about providing you with the knowledge and skills to be able to compare different programming languages. By the end of the course, you should be able to both evaluate a language's suitability for a particular purpose, and compare the suitability of different languages for a given task. Your evaluation will consider both the design of the language and its implementation.

In this module, you will look at how you benefit from studying programming language concepts, an overview of criteria to use in evaluating languages, and some of the issues influencing language design and implementation.

1.1 Why Study Concepts of Programming Languages

Over the last 30 years or so, hundreds of different programming languages have been designed and implemented. Most computer programmers however, rarely venture beyond a handful of languages, and many program only in one or two. So you may wonder why you should study concepts of different language?

Reading Sebesta Sections 1.1 and 1.2

The text enumerates the following benefits of studying these topics.

1. Increased capacity to express programming concepts.

   The text discusses the possibility of simulating useful features of other languages if the language of choice lacks these capabilities. This is only possible if you are aware of these features. Unfortunately too many programmers limit their search for useful data or program structures to the languages with which they are most familiar. Not only should you be able to recognise useful constructs in other languages, you also need to know about their implementation, as you must provide your own implementation of the structure if it is not directly supported by the language you are using.

2. Improved background for choosing appropriate languages.

In the third part of this course, we will be looking at the logic and functional languages. These are far better suited to certain application areas than the imperative languages such as C or Pascal.

3. Increased ability to learn new languages.

4. Understanding the significance of implementation.

   Remember "appearances can be deceiving". Two apparently similar features in two different languages may be implemented in two completely different ways and thus the cost of using the feature may also differ greatly. e.g. Nearly all languages provide an addition operation, but the cost of performing addition varies enormously from C to COBOL to Smalltalk. Does this mean we should never use Smalltalk or COBOL to perform addition? Obviously not! It does mean that we know the cost of programming, say, repeated addition rather than a single multiplication may be higher in Smalltalk than in C and therefore we may need to adjust an algorithm being borrowed from a C implementation.

5. Increased ability to design new languages.

6. Overall advancement of computing.

Sebesta section 1.2 describes a broad classification of the application areas in which programming languages are used. This should reinforce the notion that certain languages are far better suited to certain situations than others, and that, given a particular problem, the appropriate choice of language is often a major step in providing the solution.

1.2 Language Evaluation Criteria

One of the benefits given for studying this course is that you will be better able to choose the most appropriate language for a given task. In order to do so, you need some set of rules to apply when evaluating alternative languages.

Reading Sebesta Section 1.3

Sebesta categorises evaluation criteria as those affecting a language's readability, writeability and reliability. Be sure you understand how the various factors affect these three broad categories.

As Sebesta points out, while there is little argument that readability, writeability and reliability and the various factors influencing these characteristics are important, there is no consensus as to their relative importance. Is a language that is more readable than another, but less writeable better, or is the more writeable, less readable language better? Is an extremely reliable language that is both difficult to read and write a better language than one which is very easy to read and write, but may be less reliable?

The answer, of course, depends on the situation. Rarely will one language be the perfect choice, and some trade-offs will be made.

1.3 Language Design and Implementation

In Sections 1.4 and 1.6, Sebesta discusses how both computer architecture and programming methodologies have influenced the design of computer languages.

The imperative languages were designed to make the most efficient use of the von Neuman architecture used in the majority of computer systems. You are probably familiar with these ideas, that a program consists of a series of statements to be executed in sequence, changing the value of variables stored in memory. In fact, if you have not seen a functional or logic language before, you may wonder how a language without variables and assignment statements to store data, which is not written as a series of steps to be followed and has no concept of sequence, let alone any means of sequence control, can actually work. We will be looking at this later in the course.

Implementation methods and programming environments are discussed in Sections 1.7 and 1.8. The compilation and interpretation methods of language implementation are compared as well as hybrid approaches. Whether a language is compiled or interpreted, or comes with an integrated programming environment are other factors to be considered when assessing a language's suitability in a given situation.

Reading Sebesta Sections 1.4 – 1.8

1.4 Evaluating language

In Chapter 2, Sebesta presents a history of the more common programming languages, and evaluates these languages in the context of the times and applications for which they were designed. Although you will not be examined on any of the historical detail, this chapter does provide a brief introduction to many issues still current in language design. You should pay particular attention to the design processes and language evaluations. Later in this course you will be asked to perform similar evaluations.

Reading Sebesta Chapter 2

1.5 Summary

Practising programmers can benefit in a number of ways from the study of programming language concepts. One of these benefits is having the skill to choose the most appropriate language for a given task. Because computers are now used to solve problems in a wide variety of application areas, it is not possible to use just one language in all situations. You have studied a number of criteria to be considered when evaluating a language. These criteria are usually categorised as affecting a programming language's readability, writeability and reliability, while cost can also be a deciding factor. You have also seen examples of evaluating languages in terms of these criteria.

Exercises

1. Complete questions 1, 2, 7, 8, 9, 11, 14, 15, 18, 22, 23, 25 and 30 from the Review Questions in Chapter 1 of Sebesta. Check your answers by referring to the text.

2. Consider questions 8, 12, 17 from the Problem Set in Chapter 1 and questions 7 & 9 from the Problem Set in Chapter 2 if Sebesta.

Solutions to selected exercises

Exercise 2

Q 8 Some things you might consider include:

The English language has many examples where the same word has several different meanings or can be used as a different part of speech e.g. the word name can be used to refer to the identifier by which you are known, and also as a verb describing the action of assigning a name. If you were to write a program which required you to model this situation, it would be useful to use, say, *Name* for the variable referring to the actual name, and, say, *name*, for the procedure or function which set that value of the variable.

However, say you used*Name, nAme, naMe*, and *namE* for 4 different variables, you would then need to be very careful you use the correct variation throughout the program. This affects writeability and reliability. Using the same word, with varying capitalisations, as different identifiers may detract from readability, although we use context to cope with similar variations in written English.

Q 12 There is no "correct" answer. You should justify your choice of features and consider how your language will be used.

Q 17 The difference is subtle. In Java, because every statement must end with a semi-colon, the programmer will always include one quite naturally. There is no room for error. In Ada, however, semicolons are are only required to separate statements, and therefore are not needed when there is another means of determing the separation. e.g. There is no need in Ada to include a semicolon at the end of the last statement before an *end*, as the *end* marks the separation. In addition, Ada treats an*if-then-else* construct as a single statement; placing a semi colon before the *then* will terminate the *if* statement.

Q 7 The machines on which LISP programs were to run were based on the von Neuman architecture. The imperative languages exploit this achitecture far more efficiently than pure functional languages, thus LISP acquired many imperative features in a bid to improve efficiency.

Q 9 Again, no "correct" answer, only an opinion, however you should have some argument to support your opinion. Try thinking about the different purposes for which these languages were intended.

Module 2

Functional Programming Languages

Overview

Most, if not all, of the programs that you have written have used imperative programming languages i.e. languages which fully specify and control the manipulation of named data in a series of steps. Because this means that imperative programs follow a clear "do this, then do that" recipe-like approach, most programmers are quite comfortable with this structure. A disadvantage of the imperative approach is that the programmer may become burdened by the large amounts of detail necessary to specify both the data and its manipulations.

The imperative paradigm, however, is not the only programming paradigm. In this module we will look at those languages which offer one alternative: the functional languages.

This module focuses on general issues about functional languages, especially how they differ from imperative languages. There are a range of functional programming languages. We use the Haskell language for the practical work in this course. The CSC3403 Laboratory Manual provides a wealth of resources for learning about Haskell. While other functional languages have interesting features, we do not expect you to have a deep knowledge of them.

2.1 Functional Programming

The programmer using an imperative language is responsible for not only describing the computation to be performed but must also specify the sequence of steps that must be carried out to perform the computation and, because the program uses variables which refer to storage, must also be concerned with memory management. The latter two tasks are purely administrative, and the major contributors to the multitude of detail which hinders the programmer's function.

If we were to remove, or at least alleviate, much of the responsibility for the purely administrative detail, and allow the programmer to concentrate on describing the computation itself, how much easier would the programming task become?

With functional languages, the computation to be carried out is described using

functions, in the mathematical sense that a function is a mapping from a particular domain onto some particular range of values. When programming in a functional language, the programmer builds function definitions and expressions using those definitions, and then the computer evaluates the expressions. An expression denotes a value.

e.g. We may define a function square as:

```
square x = x*x
```

Then write expressions using that function:

```
square(7)
```

The computer will evaluate this expression, and return the answer, 49.

Consider also:

```
square(3 + 4)
square(35/5)
```

Each of these expressions denotes the same value as the first, 49.

Note that when writing the expression `square (35/5)` we did not have to specify the steps the computer should use to evaluate the expression. The expression could have been reduced a number of different ways e.g.

```
square (35/5)
⇒ (35/5) * (35/5)
⇒ 7 * (35/5)
⇒ 7 * 7
⇒ 49
```

```
square (35/5)
⇒ square 7
⇒ 7 * 7
⇒ 49
```

We also did not have to declare a variable in which to store the intermediate results for the reduction steps. But what if I want to use the result of this evaluation, you may ask. Because the expression `square (35/5)` denotes a value, we simply use that expression anywhere the value is to be used. Assuming the function `max` is appropriately defined, we can write the expression

```
max(48 square(35/5))
```

which will be evaluated, correctly, to 49.

What if we want to use the result of `square (35/5)` more than once? Does the function need to evaluated as many times as the value is to be used? If the language uses lazy evaluation rules to reduce the expression, no. We look at lazy evaluation further in Section 2.2.2.

These simple examples illustrate the idea that a functional programmer concentrates on the computation itself, and not on the steps taken to perform the computation, nor the organisation of named storage locations. The functional languages are designed to not only relieve the programmer of this administrative burden, but, because the computation is described using a notation founded in mathematics, also facilitate the task of proving program correctness.

Reading Sebesta Chapter 15 to the end of Section 15.3.

2.2 Types and Type Checking

Modules 6 and 7 describe how type checking can be one of the most important ways in which a programming language can ensure safety. In the imperative languages this involves declaring a variable to be of a particular type, then part of the language's compilation process checks that all operations the programmer is attempting to perform on that variable are permitted.

The functional languages are strongly typed too. But, how does this work when there are no variables and therefore no type declarations. Instead of variables for storing values, we now have expressions that represent values. So now it is the expressions denoting the values that are typed, rather than the variables naming the storage location for the value.

The functional languages usually have built in types for numbers, characters and booleans. For the purposes of the following discussion we will assume these are named Int, String and Bool as in the functional language Haskell.

The simplest expressions are just values e.g.

```
41      'z'      True      103      'A'      'H'      False
```

which have the following types:

Int *String* *Bool* *Int* *String* *String* *Bool*

The types of more complex expressions are built up by combining these primitive types to form derived types.

2.2.1 Type Inference

Consider the function `square` defined earlier. This function takes an argument of type Int and returns a value of type Int. The function square therefore has the type:

$Int \rightarrow Int$

This type is easily deduced from the function's definition. The symbol * is reserved for the multiplication operation on arguments of type Int, then the type of the argument, x, must be Int, and the type of the result will also be Int. The use of the symbol *, infers that the type of x must be Int.

As a further example, let's define a new function `even`[1], which returns `True` if its argument is an even number.

```
even x | x mod 2 == 0 = True
       | otherwise    = False
```

This function has type $Int \rightarrow Bool$. The argument, `x`, has type Int, inferred from the use of the `mod` function, and returns a Bool, inferred from the type of the expressions `True` and `False`.

Consider the function `avepair` which returns the average of a pair of numbers.

```
avepair (x, y) = (x + y)/2
```

The type of this function is $(Int, Int) \rightarrow Int$. The function has a single argument,

[1] Note that the usual Haskell definition would be simply

```
even x = x mod 2 == 0
```

however, that argument is a tuple. By pairing two arguments of type Int, we have created a (Int, Int) type. We can create a $(Bool, Char)$ type by pairing a $Bool$ with a $Char$. This type is distinct from the type $(Char, Bool)$, which also pairs $Bool$s and $Char$s, but in the reverse order.

As a function's type is deduced using type inference in the manner described above, most functional languages do not require a type declaration for function definitions. Having no type declarations does not mean the language is not typed. In fact the functional languages are strongly typed.

Reading Sebesta Sections 15.7 to 15.8

In Section 15.7 on ML, Sebesta points at that for function square, very similar to the function we looked at above, the type of the arguments cannot be deduced by the compiler. Haskell, on the other hand, has a type class system which allows definitions like the one we have described, as long as the argument has a type which is numeric.

2.2.2 Lazy Evaluation

Using lazy evaluation, the arguments to a function are not evaluated until needed. This is also discussed in Module 8, Section 8.1.1. However, as we foreshadowed in Section 2.1, simply evaluating as needed is inefficient as a result needed more than once will be evaluated more than once, e.g. assume double is defined as

```
double x = x + x
```

then in `double (square(3)), square (3)` must be evaluated twice.

```
double (square 3) ⇒ (square 3) + (square 3)
```

We could define a new function

```
n = square(3)
```

which then simplifies writing the expression to

```
double n
```

However, this will still be expanded to `n + n`, and each time `n` is encountered when evaluating the expression, it is replaced by the definition of `n` and so `square(3)` is still evaluated twice.

This is call by need rather than true lazy evaluation. We replace each argument by a copy of the expression and don't evaluate the expression till needed.

With true lazy evaluation, instead of replacing the argument `n` with a copy of the expression `square (3)`, the argument is replaced by a pointer to the expression. In this way the result of any evaluation can be shared by each reference to the expression.

This evaluation technique is not possible with languages where side-effects may change the outcome of an evaluation, e.g. Suppose a function F has a side effect that prints some message prior to returning an integer result, and we have defined a function multiply as:

```
multiply x y = x * y
```

Then `square F` and `multiply F F` will produce different side effects; in the first instance, F is evaluated only once, and therefore the message will be printed only once.

2.3 Functional vs Imperative

At the beginning of this module we saw that the main difference between programming in a functional language and programming in an imperative language is that the programmer need no longer be concerned with memory management or sequencing the individual steps to perform some computation. However, this is not the only way in which the two paradigms differ. There a a number of other advantages, some disadvantages, and the areas of application also differ.

Reading Sebesta Sections 15.9 to 15.10

2.4 Summary

Functional programming and functional programming languages are based on a computational model that is quite different from that of the imperative languages.

True functional programming languages have no destructive assignment. Any "variables" do not refer to storage locations whose contents may change, but are names for defined values which will not change throughout execution. All data is passed as parameters to, or results from, function calls.

Functional languages are strongly typed, however the types of functions can usually be deduced by the compiler/translator using type inferencing.

The form of lazy evaluation found in a number of functional languages is extremely powerful and permits programmers to program with infinite lists.

Exercises

1. Complete questions 1 and 14 from the Review Questions in Chapter 15 of Sebesta. Check your answers by referring to the text.

2. Give a definition of a function `sign:: Int -> Int` which returns 1 if its argument is positive, -1 if its argument is negative, 0 otherwise.

3. Using the primitive types of Haskell, together with type variables a, b, and c, suggest possible types for the following functions:

```
one x = 1
apply f x = f x
compose f g x = f (g x)
```

Solutions to Selected Exercises

Exercise 2

```
sign x | x > 0      = 1
       | x < 0      = -1
       | otherwise  = 0
```

Exercise 3

```
one :: a → Int
apply :: (a → b) → a → b
compose :: (a → b) → (c → a) → c → b
```

Module 3

Formal Description of Languages

Overview

Programming languages today need to be formally specified to ensure that all compilers conforming to the specification will translate source code to produce a program that executes as expected. Any of you who have experienced non-ANSI C compilers, or have worked with standard Pascal and Turbo Pascal will be aware of how frustrating it can be when a compiler does not behave as expected.

Before we look at formal methods for specifying programming language, we will draw some analogies between programming languages and natural languages, particularly English.

3.1 Language Elements

An English sentence is made up of words which may be classified as nouns, verbs, adjectives, pronouns, prepositions or conjunctions.

Nouns

A noun allows us to refer to some object.

> The cat, Tom, chased the mouse, Jerry.

The names Tom and Jerry, provide information about who was doing the chasing and who was being chased. This corresponds to using a name to allow us to refer to some program object. The nouns cat and mouse tell us more about the kind of animals involved in the chase. You should already be familiar with the notion of declaring a variable identifier which allows you to refer to some storage location, as well as naming constants, functions and types.

```
typedef {..}: mouse;
typedef {..}: cat;
void chase(cat c; mouse m) {
....
}
```

```
cat Tom;
mouse Jerry;

main(){
chased(Tom, Jerry);
....
```

Originally the only program objects were named storage locations i.e. variables. Gradually languages evolved which allowed aggregate objects such as arrays and records to be created, initialised, compared, assigned to, passed as arguments and returned as results. More recently languages have been developed which also treat functions and types as first class objects i.e. allow them to be manipulated and processed a unit.

Pronouns

A pronoun provides a way of referring to an object without using the object's name. The same pronoun can be used to refer to different objects at different times. e.g. We know that Tom is a male cat so we could write

He chased Jerry.

and providing we know that he refers to Tom, the sentence is meaningful. However we could also use the pronoun he to refer to other cats, say Garfield. Then the same sentence tells us that Garfield chased Jerry.

Pronouns in English correspond to pointers in programming languages. Pointers may be used to refer to objects which are not named explicitly.

```
cat *t;
t = &Tom;
chased(t, Jerry);
```

Adjectives

An adjective describes attributes of an object e.g. its size, shape or colour. In programming languages we describe an object's characteristics by declaring it to be of a certain type. So a data type corresponds to an adjective.

Verbs

In English verbs are described as "doing" words as they describe actions, occurrences or states of being. Many programming languages also use action words; `return, break, GOTO, STOP`. Procedures and function calls, arithmetic operators, assignment statements are all commands to perform some action, so are just like action verbs. Similarly relational operators are like verbs that refer to states of being.

Prepositions and Conjunctions

Prepositions are used with nouns to denote time, place, position, means e.g. at home, by train. Conjunctions join words, phrases or sentences e.g. but, and. Programming languages also contain similar sets of words e.g. `WHILE, ELSE, CASE, BY, WITH.`

Of course the actual set of words differs from language to language.

3.2 Describing Language

In English we can refer to the various units in the language by name. We have *words* which can be combined into *phrases* and *sentences*. Sentences can be combined into *paragraphs* and *paragraphs* combined to form *essays*. We know the rules governing these combinations: a sentence is a group of words containing a verb (expressed or implied), the first letter of the first word must be uppercase, and the sentence ends with a full stop (period), exclamation mark or question mark.

We call those parts of a language that we can use to talk about the language the metalanguage. There is also a *metalanguage* used to describe programming languages and in this section we will look at this metalanguage and compare it with the metalanguage we use to describe English and other natural languages.

Reading Sebesta Chapter 3, to the end of Section 3.2.

The distinction between syntax and semantics is important. e.g. Consider this grammatically correct English sentence.

> The ice sang across Mary's ravaged roses.

Does this sentence make sense? When you read something written in a natural language, you are actually translating the symbols written on the page into some message that you understand. The first step in reading an English sentence is to recognise the words that make up the sentence. We can do this as we know that each word is a string of letters delimited by spaces. If the words are combined in a meaningful way and follow the conventions you are familiar with to form phrases and sentences, you can easily identify those phrases and sentences and so make sense of what you are reading. If you do not recognise any of the words, or the words are combined in an unusual way, you may not be able to understand the writer's message.

A similar process occurs in translating a source program into executable code. In Chapter 1 of Sebesta you studied the various stages of this process. Lexical analysis is the process of grouping characters into *lexemes*, and identifying to which category of *token* the lexeme belongs. The second stage of translation is *parsing* where the combination of tokens are recognised as forming legal structures such as statements, expressions, sub-routines, according to the grammar rules specified for the particular language. The final translation stages of semantic analysis and code generation correspond to understanding the written message.

We can draw parallels between the metalanguage we use to describe English and the metalanguage used to describe programming languages in much the same way we compared elements of language in the previous section.

Both written English and source code are streams of characters. Words correspond to lexemes; the smallest unit of the language. Words can be characterised as various parts of speech, verbs, nouns etc., whereas lexemes are characterised as a particular category of token.

An English sentence is a legal combination of words which conveys some meaning.

A program *statement* is a syntactically correct combination of tokens which, usually, specifies some action and the object on which the action is to be performed. In English, sentences are combined to form paragraphs. In programming languages statements can be combined to form scopes. In the same way there are rules to delimit paragraphs, programming languages often delimit *scopes* using special symbols such as `THEN`, `ELSE`, `WHILE`, `DO` or begin-scope, end-scope symbols such as matched braces in C.

An English essay will often contain footnotes or references which convey additional or explanatory information about the text. This is analogous to comments in source code.

Of course we cannot just take any set of written characters and claim they form an English sentence or a computer program. Only certain combinations of words or tokens form syntactically or grammatically correct English sentences or program statements, and not all these are semantically sensible.

The set of rules used to recognise tokens and determine syntactically correct sequences of tokens form a *grammar* in the same way that the set of rules governing correct English sentences is called a grammar. A formal notation for specifying grammars is *BNF*, or *Backus-Naur form*.

3.3 Formal Grammars

Consider the following sentence.

> This course gave him a headache.

In the traditional study of English grammar, parsing identifies the grammatical structure of a sentence. A parse tree for this sentence might be constructed as in Figure 3.1. This parse tree represents rules for structuring this sentences. These rules can be specified using constructs called *productions*.

$$
\begin{array}{lcl}
\textit{<Sentence>} & \rightarrow & \textit{<Noun\_phrase> <Verb\_phrase>} \\
\textit{<Noun\_phrase>} & \rightarrow & \textit{<Article> <noun>} \\
\textit{<Verb\_Phrase>} & \rightarrow & \textit{<Verb> <Indirect\_object> <Noun\_phrase>} \\
\textit{<Indirect\_object>} & \rightarrow & \textit{<Pronoun>}
\end{array}
$$

When reading these rules the arrow symbol is read as "is defined as being". The following rules are implied. (The | symbol is read as "or".)

$$
\begin{array}{lcl}
< Noun > & \rightarrow & \text{headache} \mid \text{course} \\
< Verb > & \rightarrow & \text{gave} \\
< Article > & \rightarrow & \text{The} \mid \text{a} \\
< Pronoun > & \rightarrow & \text{him}
\end{array}
$$

Strings such as Noun, Verb_phrase, Article which appear on the left side of a production are called non-terminal symbols. These do not appear in the sentence being generated. Strings such as "The", "headache", which are found only on the right side of productions and do appear in the final sentence, are called terminal symbols. In the rules above non-terminal symbols are enclosed in angle brackets, terminal symbols are simple strings.

The production rules can be used to "generate" the sentence. Starting from the rule describing a sentence, production rules are used to replace one symbol at each step.

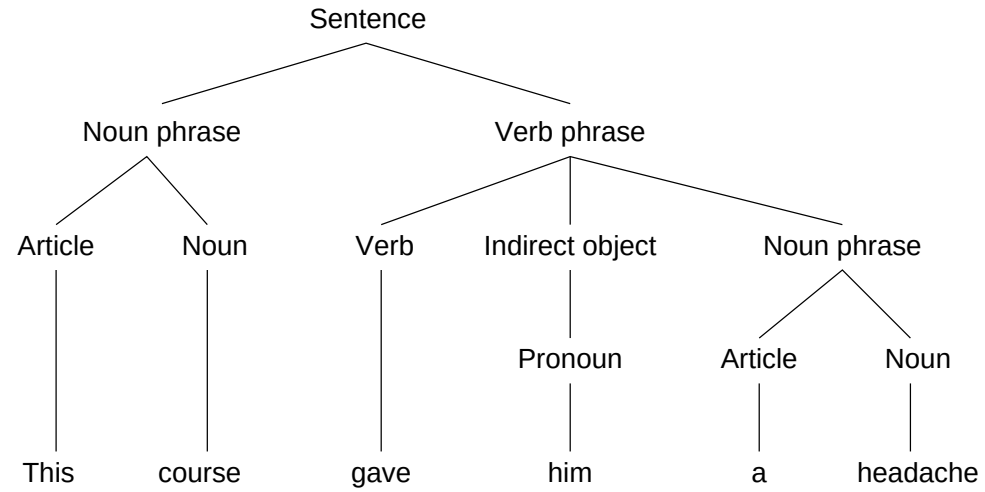


Figure 3.1: Parse tree for an English sentence

$$
\begin{aligned}
< Sentence > &\rightarrow < Noun\_phrase >< Verb\_phrase > \\
&\rightarrow < Article >< Noun >< Verb\_phrase > \\
&\rightarrow < Article >< Noun >< Verb >< Indirect\_object >< Noun\_phrase > \\
&\rightarrow < Article >< Noun >< Verb >< Pronoun >< Noun\_phrase > \\
&\rightarrow < Article >< Noun >< Verb >< Pronoun >< Article >< Noun > \\
&\rightarrow This\ < Noun >< Verb >< Pronoun >< Article >< Noun > \\
&\rightarrow This\ course\ < Verb >< Pronoun >< Article >< Noun > \\
&\rightarrow This\ course\ gave\ < Pronoun >< Article >< Noun > \\
&\rightarrow This\ course\ gave\ him\ < Article >< Noun > \\
&\rightarrow This\ course\ gave\ him\ a\ < Noun > \\
&\rightarrow This\ course\ gave\ him\ a\ headache
\end{aligned}
$$

These same rules can also be used to generate another sentence:

This headache gave him a course

This clearly illustrates that a grammatically correct sentence is not necessarily meaningful.

3.4 Kinds of grammars

Formally, a grammar consists of four elements:

1. A set of *nonterminal symbols*. Non-terminal symbols typically denote phrases or sub-components of the language (for example: *while-statement*). In examples, these are usually written with upper case characters like A, B, C etc.

2. A set of *terminal symbols*. A legal sentence will contain only terminal symbols (for example: *identifier*, `else`). In examples, these are usually written with lower case characters like a, b, c etc.

3. A set of *productions*. A production is a *rule* which describes how to *replace* a string of symbols with another string. A production rule has a left and a

right hand side separated by an arrow. The rule states "in deriving a sentence of the language defined by the grammar, any occurrence of the left hand side symbols may be replaced by the right hand side symbols".

4. A distinguished nonterminal or *start* symbol.

We commonly use the | symbol to abbreviate grammar descriptions when there is more than one production with the same LHS. For example, $A \rightarrow a \mid b$ defines two production rules: $A \rightarrow a$ and $A \rightarrow b$.

A sentence of the language is derived by starting with the start symbol, then repeatedly applying *any* rule whose LHS appears in the current string. Eventually, only terminal symbols will remain in the string, which is a valid sentence of the language. The intermediate form, which will contain non-terminal symbols, is called a *sentential form*.

The distinguished linguist Noam Chomsky has classified grammars into four kinds. We will look briefly at them.

3.4.1 Regular grammar

Consider the simple grammar

$S \rightarrow A$
$A \rightarrow aA \mid aB$
$B \rightarrow bB \mid b$

Here is an example derivation of a sentence from this grammar.

$$\begin{aligned} S &\Rightarrow A \\ &\Rightarrow aA \\ &\Rightarrow aaB \\ &\Rightarrow aabB \\ &\Rightarrow aabbB \\ &\Rightarrow aabbb \end{aligned}$$

Some reflection will confirm that all sentences have the same form, $a^m b^n$ where $m \geq 1$, $n \geq 1$, and a^x means that a is repeated x times. Note that there is usually a different number of a and b symbols.

This example is a *regular grammar*. The productions of a regular grammar have a restricted form. A production can be either $A \rightarrow xB$ or $A \rightarrow x$ where A and B are non-terminals and x is either a terminal or the empty symbol.

Regular grammars are sufficient for describing lexical characteristics of languages (e.g. the format of identifiers) but not for describing the syntax of a programming language. In particular, a regular grammar cannot be used to generate sentences with a form like $a^m b^n$, where $m = n$. This is absolutely necessary for a programing language where for instance parentheses must always match—there must be the same number of opening parentheses as closing ones.

3.4.2 Context-free grammar

Consider an example of a context free grammar.

$S \rightarrow aSb \mid ab$

Here is an example derivation of a sentence from this grammar.

$$
\begin{aligned}
S &\Rightarrow aSb \\
&\Rightarrow aaSbb \\
&\Rightarrow aaabbb
\end{aligned}
$$

This grammar is more powerful than the regular grammar, as it is possible to produce sentences with matching counts of symbols. Sentences of the language all have the form $a^n b^n$ where $n \geq 1$

Context-free grammar rules have the form $A \rightarrow x_1 \ldots x_n$, where $n \geq 0$, A is a nonterminal, and x_i can be a nonterminal or terminal.

The main restriction on a context free grammar (CFG) is that the LHS of a production must be a single non-terminal (compare this with the context sensitive grammar or the following section).

CFGs are powerful enough to describe the syntax of standard programming languages.

3.4.3 Context-sensitive grammar

As a final example, consider our most complex example, a context sensitive grammar.

$$
\begin{aligned}
S &\rightarrow aSBC \mid abC \\
CB &\rightarrow BC \\
bB &\rightarrow bb \\
bC &\rightarrow bc \\
cC &\rightarrow cc
\end{aligned}
$$

Here is an example derivation of a sentence from this grammar.

$$
\begin{aligned}
S &\Rightarrow aSBC \\
&\Rightarrow aabCBC \\
&\Rightarrow aabBCC \\
&\Rightarrow aabbCC \\
&\Rightarrow aabbcC \\
&\Rightarrow aabbcc
\end{aligned}
$$

This grammar generates sentences of the form: $a^n b^n c^n$ where $n \geq 1$. (A CFG can only generate pairs of symbols of the same quantity.)

A CSG can have more than one symbols on LHS of the production, and these can include terminal symbols. Note that expansion of C above depends on its context (what surrounds it in the sentence). Formally, rule of a CSG have the form $x_1 \ldots x_n \rightarrow y_1 \ldots y_m$, where $n \geq 1$, $n \leq m$ and x_i and y_i can be either terminal or non-terminal.

3.4.4 The Chomsky Hierarchy

Chomsky recognised that these kinds of grammars form a hierarchy. A grammar lower down the hierarchy is less powerful than one higher up. The hierarchy is as follows

- Unrestricted
- Context-sensitive
- Context-free
- Regular

Only the lower two are of interest to us in describing programming language syntax. The hierarchy demonstrates that programming languages are in fact linguistically simple artifacts, in that they can be described using relatively simple grammar notations.

3.5 Backus-Naur Form

The notation used above to describe the production rules we used to parse an English sentence is BNF. BNF is the metalanguage commonly used to describe syntax of programming languages.

Reading Sebesta Section 3.3 up to Section 3.3.1.8

Note that Sebesta calls the process of using production rules to generate a sentence a "derivation". The English sentence generated in the previous section is a "leftmost" derivation.

Examples

Question

Define the signed integers using BNF notation.

Solution

A signed integer is a sign, followed by a variable length list of digits. We can write this as:

$< signed\_integer > \rightarrow < sign > < digit\_list >$

A sign can be either a "+" or a "-" sign.

$<sign> \rightarrow + \mid -$

Writing the rule for a single digit should also be straight-forward.

$<digit> \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

The only slightly tricky part is defining an integer list. Consider again how Sebesta defines a variable identifier list on page 111. The rule is reproduced below.

$<ident\_list> \rightarrow <identifier> \mid <identifier> <ident\_list>$

The rule we need will be very similar. Simply replace the symbols <ident_list> with <digit_list> and <identifier> with <digit>. In fact the rule you will use to define any variable length list of symbols should nearly always follow this format.

The complete set of rules defining a signed integer is:

$$
\begin{array}{lcl}
< signed\_integer > & \rightarrow & < sign > < digit\_list > \\
< sign > & \rightarrow & + \mid - \\
< digit > & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
< digit\_list > & \rightarrow & < digit > \mid < digit > < digit\_list >
\end{array}
$$

Question

Would it be correct to write the following production i.e. reverse the position of <digit> and <digit_list> in the recursive definition?

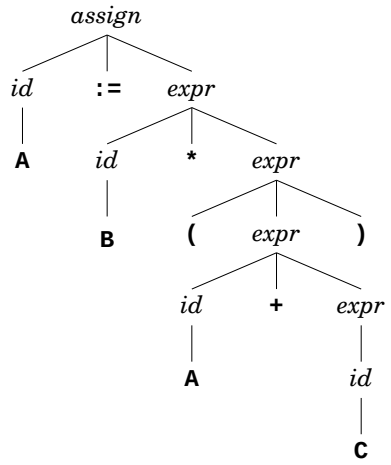


Figure 3.2: Parse tree for the expression `A := B * ( A + C )`

<*digit_list*> → <*digit*> | <*digit_list*><*digit*>

Solution

If you are unsure, try generating an integer. Using the left option obviously works for a single digit integer, so let's just consider longer cases. First try using the right (recursive) definition just once.

$$\begin{aligned} < digit\_list > &\rightarrow < digit\_list > < digit > \\ &\rightarrow < digit > < digit > \end{aligned}$$

So it works for double digit integers! Now try replacing <digit_list> on the RHS by the recursive definition.

$$\begin{aligned} < digit\_list > &\rightarrow < digit\_list >< digit > \\ &\rightarrow < digit\_list > < digit > < digit > \\ &\rightarrow < digit > < digit > < digit > \end{aligned}$$

This will generate an integer with three digits. So it would appear, in this case, that the position of the recursive symbol is irrelevant.

Beware! This may not always be true. In the next module we will look at other examples.

3.6 Parse Trees and Abstract Syntax Trees

A *parse tree* is just a graphical representation of the derivation of a sentence. In section 3.3.1.6, Sebesta shows a grammar and a parse tree for the expression `A := B * ( A + C )`. The tree is reproduced in figure 3.2.

Each mode represents a derivation step, so all internal nodes are labeled with non-terminal symbols. If these non-terminals, together with punctuation, are stripped away, then we have an *abstract syntax tree* (AST). The AST alone is sufficient for a compiler to use in generating machine code. For instance, parentheses are useful only to force the order of a derivation, and hence the *shape* of the tree. It is the shape of an expression tree that defines expression evaluation order.

Figure 3.3 shows the corresponding AST. Note that it is clear that the addition must

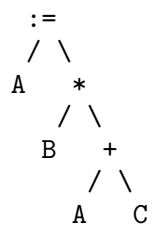


Figure 3.3: Abstract syntax tree for the expression `A := B * ( A + C )`

take place before the multiplication, even in the absence of parentheses.

Exercises

1. Complete questions 6, 10 and 11 from the Problem Set in Chapter 3 of Sebesta.

2. Write a BNF grammar to describe arithmetic expressions consisting of single digits and plus and minus signs e.g. $9 - 5 + 6$, $4 - 3$, 2 etc.

   Hint: think of each expression as a list of digits separated by a plus or minus sign.

3. Write a BNF grammar to specify the set of all strings of a's and b's where all the a's precede the b's e.g. ab, aab, aaab, aabb. A legal string in this language must contain at least one a and one b.

4. A robot can move one step east, north, south, or west from its current position. The robot can understand a sequence of instructions which starts with the word *begin*, and includes the directions as commands. Any sequence must include at least one command.

   e.g. The robot can follow the following sequence of instructions.

   begin west south east east east north south

   If we assume the robot's position is given by an (x, y) pair, its initial position is (0,0), and each step is one unit, then the changes in the robot's position after following these commands are shown in Figure 3.4. In the figure each arrow represents one step.

   (a) Construct a BNF grammar for the language understood by the robot.

   (b) Construct a parse tree for the sequence of instructions shown above.

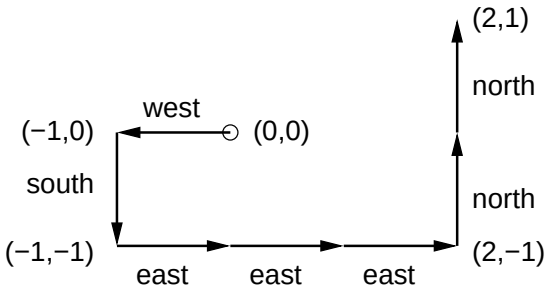
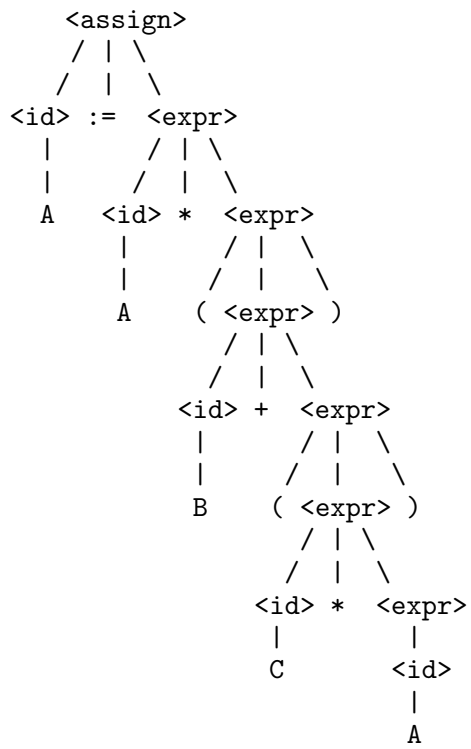


Figure 3.4: Changes in the robot's position

Solutions to Selected Exercises

Exercise 1

6(a)



10 One or more a's, followed by one or more b's, followed by one or more c's.

11 a, d

You should have seen straightaway that (c) was not in the language as every sentence must end in b. All you had to do then was check the others could be generated from the rules.

Exercise 2

$< expr > \rightarrow < expr >$ + $< digit >$
$< expr > \rightarrow < expr >$ - $< digit >$
$< expr > \rightarrow < digit >$
$< digit > \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0$

Exercise 3 This is almost exactly the same as the grammar in problem 10 of the text's Problem Set; only the c's have been omitted.

$$
\begin{array}{lcl}
< S > & \rightarrow & < A >< B > \\
< A > & \rightarrow & \text{a}< A > \mid \text{a} \\
< B > & \rightarrow & \text{b}< B > \mid \text{b}
\end{array}
$$

Exercise 4

$$
\begin{array}{lcl}
< instructions > & \rightarrow & \text{begin} < command\_list > \\
< command\_list > & \rightarrow & < command > < command\_list > \mid < command > \\
< command > & \rightarrow & \text{north} \mid \text{south} \mid \text{east} \mid \text{west}
\end{array}
$$

3.7 Summary

In this module we saw that the syntax of languages can be formally specified using the notation Backus-Naur Form. Production rules specified in BNF can be used for both generating and recognising legal sentences in the language.

The simple examples we have studied here will be extended next module and we will look at issues such as resolving ambiguities and associated issues.

Module 4

More Formal Methods

Overview

The previous module introduced a formal notation for describing the syntax of languages. The examples we studied involved very simple languages, or subsets of a language, and were quite straight forward. However, if we wish to describe the syntax of a more complex language there are a number of other issues to consider that were not illustrated in our simple examples.

In this module we discuss the more important of these issues, look at some alternatives to BNF and finally present an overview of some formal methods for describing semantics.

4.1 Ambiguity

An ambiguous sentence is one that can have several meanings. e.g. The sentence

> He saw a pretty bird

could mean "He saw an attractive feathered animal" or "He saw an attractive female". The different meanings of the sentence arise due to different meanings we may attach to the word "bird". In both cases "bird" is a noun, and the sentence has just one parse tree. In the context of programming languages we can avoid this sort of semantic ambiguity by preventing words or tokens from having more than one meaning.

However, consider the sentence

> Time flies like an arrow.

This sentence also has a number of different possible meanings.

> "Time moves in the same manner that an arrow moves."
>
> "Measure the speed of flies in the same way you measure the speed of an arrow."
>
> "Measure the speed of flies that resemble arrows."
>
> "The variety of flies called 'time flies' are fond of an arrow."

All these different meanings arise depending on the way the sentence is parsed. If we parse *time* as a noun, *flies* as a verb and *like* as a preposition we get the first meaning. If we parse *time* as a verb, *flies* as a noun and *like* as a preposition we get

the second meaning. By changing our parse of *like* to being an adjective, the third meaning results. Finally, if our parse recognises *time* as being an adjective, *flies* as a noun and *like* as a verb, the sentence conveys the fourth meaning.

This is syntactic ambiguity where the ambiguity is a result of there being more than one way of parsing the sentence. It is this kind of ambiguity that we need to be concerned about when specifying programming languages as most compilers use the syntactic form of a language to determine the semantics or meaning of program statements.

Reading Sebesta Sections 3.3.1.8 to 3.3.1.10

4.1.1 Precedence and Associativity

In Example 3.4, Sebesta presents an unambiguous grammar for arithmetic expressions involving the +, -, * and / operators. We will now look at the way in which such a grammar might be produced.

We start with a precedence table, showing the operators in order of increasing precedence.

Associativity	**Operators**
left	+ -
left	/ *

Create two non-terminals, <*expr*> and <*term*> for the two levels of precedence. For each additional level of precedence you would need to introduce another non-terminal.

We also need a non-terminal, <*factor*>, to generate basic units (the operands) in each expression. In arithmetic expressions our operands would normally be any actual number (e.g. 5, 8, 9.3, 1004.1333) or another arithmetic expression. Actual numbers would be terminals, however, in this example our terminals are not numbers, but *A*, *B* or *C*. Basic units can either be a parenthesised expression, or an <*id*>. So the production for our basic unit will be

<*factor*> → <*id*> | *(*<*expr*>*)*

Now consider the operators with the highest precedence i.e. * and /. These associate to the left, so the productions will be left recursive i.e. the LHS of the rule appears at the beginning of the RHS.

<*term*> → <*term*> * <*factor*> | <*term*> / <*factor*> | <*factor*>

We define the rule for <*expr*> similarly.

<*expr*> → <*expr*> + <*term*> | <*expr*> - <*term*> | , <*term*>

These three definitions, together with the rule for <*id*>, make up the grammar shown in Sebesta. This grammar treats each expression as a list of terms separated by either a + or a - sign. The grammar also permits expressions which have arbitrarily deep nesting.

The process just described can be extended to add further operators and other precedence levels.

4.1.2 The "dangling-else" and other problems

The ambiguous grammar for if-then-else statements described Sebesta section 3.3.1.10 produces the "dangling-else" problem. i.e. The grammar does not unambiguously determine to which *then* clause an *else* clause should be matched, and allows this statement to be derived.

if <*logic_expr*>*then if* <*logic_expr*>*then* <*stmt*> *else* <*stmt*>

This same problem arose during early attempts to define ALGOL 60. Suppose we change the rules as follows.

<*stmt*>	→	*if* <*logic_expr*> *then* <*unmatched*>
	|	*if* <*logic_expr*> *then* <*unmatched*> *else* <*stmt*>
	|	*other*
<*unmatched*>	→	*for* <*cond*> *do* <*stmt*>
	|	*other*

These definitions were also part of the early efforts the ALGOL 60 designer. The "dangling-else" problem has been removed however a further ambiguity has been introduced as the statement below has two parse trees.

if <*logic_expr*> *then for* <*cond*> *do if* <*logic_expr*> *then other else other*

Exercises

1. Construct parse trees for the two ambiguous strings discussed above.
2. Complete question 8 of the Problem Set in Chapter 3 of Sebesta.
3. It is claimed that the following grammar solves the "dangling else" problem.

   <*stmt*> → *if* <*expr*> *then* <*stmt*> | <*matched_statement*>

   <*matched_stmt*> → *if* <*expr*> *then* <*matched_stmt*> *else* <*stmt*> | <*other*>

   Show that this grammar is still ambiguous.

4.2 Alternative Notations

BNF is not the only notation used for formal descriptions of syntax. Various extensions to BNF, collectively called Extended Backus-Naur Form (EBNF), and Syntax Graphs are other common methods.

Reading Sebesta Sections 3.3.2 and 3.3.3.

4.2.1 Extended Backus-Naur Form

The original BNF has been extended a number of ways, producing different versions of EBNF. The extensions we will use are defined below.

Optional elements appear in square brackets. e.g. In the following rule specifying a Pascal program heading, the program parameters and the parentheses in which they appear, are optional.

<*program-heading*> → '`program`' <*identifier*> ['`(`' <*program-parameters*> '`)`']

Therefore both the following program headings are legal Pascal[1].

`program other`	`program another (input,output)`

Zero or more repetitions of an element are enclosed in curly braces. e.g. a sequence of statements in a Pascal program might be defined as a single statement followed by zero or more repetitions of a semi-colon, statement combination.

<*statement_sequence*> → <*statement*> { ';' <*statement*> }

Parentheses may be used to indicate grouping. e.g. A Pascal statement can be either a simple statement or a structured statement, optionally preceded by a label.

<*statement*> → [<*label*>] (<*simple-statement*> | *(structured-statement*>)

We will not use other extensions such as numeric superscripts or ellipses.

4.2.2 Syntax Graphs

Syntax graphs were developed by Nicklaus Wirth to describe the syntax of Pascal. Most text books describe Pascal syntax in this form. Syntax graphs are often called "railroad diagrams" because of their resemblance to a map of interconnecting railroads showing stations.

Syntax graphs are just as expressive as BNF or EBNF, however the graphical interpretation of relationships between language elements is usually easier for people to follow. EBNF rules are used to write grammars intended as input to parser generators. Even Pascal, which, uses railroad diagrams to describe syntax to people, has its grammar formally defined in EBNF.

It is usually possible to translate directly from EBNF to railroad diagrams or vice-versa. Translating to or from BNF is a little trickier, as BNF usually requires more rules than if the same language is defined in EBNF.

Examples

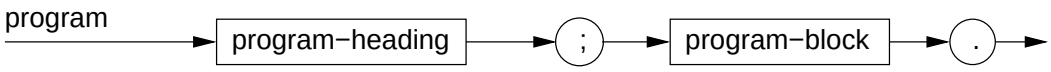
Question

Translate the following productions, which describe part of Pascal, into railroad diagrams.

<*program*> → <*program-heading*> ';' <*program-block*> '.'
<*program-heading*> → '`program`' <*identifier*> ['(' <*program-parameters*> ')']
<*program-parameters*> → <*identifier-list*>
<*identifier-list*> → <*identifier*> { ',' <*identifier*> }

Solution

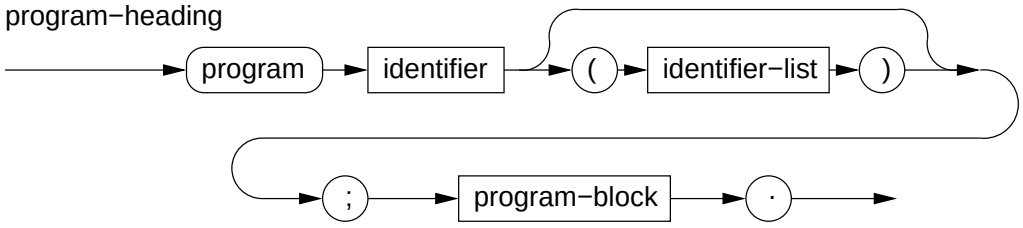
The start symbol for this grammar is <program>. This rule is a simple sequence of elements, so the translation is straight-forward.



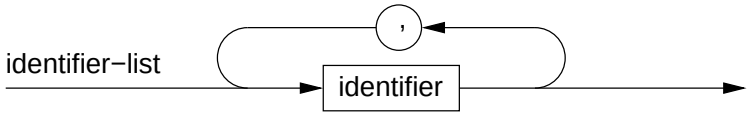
Now consider <program-heading>. This is a sequence that starts with the terminal

[1]Note that there is no semi-colon at the end of the program heading. This requirement is contained in the rule for <program>, i.e.
<program> → <program-heading> ';' <program-block. '.'

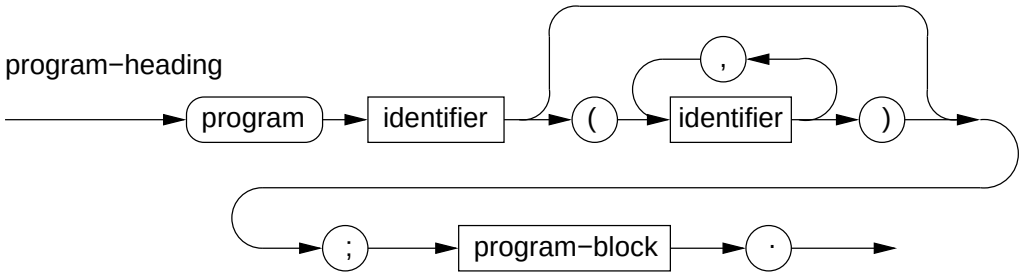
'program', which is followed by an identifier, then, optionally, program parameters in parentheses. Note that the production for <program-parameters> just replaces this non-terminal with another non-terminal, <identifier-list>. We can immediately make that replacement in the railroad diagram.



All that's left to do, is to incorporate the production for <identifier_list>.



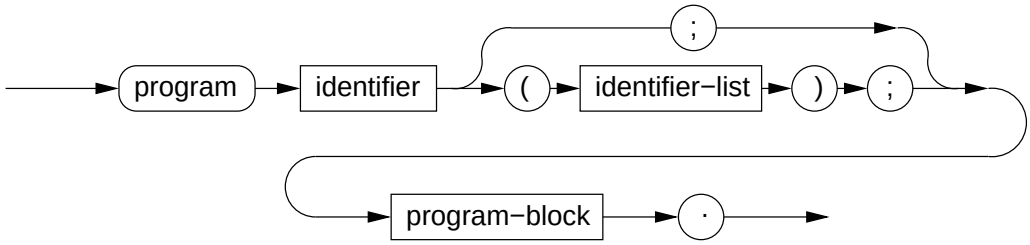
The final railroad diagram is shown below.



Note

In this example we were able to produce a single railroad diagram from four (4) EBNF productions. In many cases it will not be possible to combine all the diagrams in this way. However, if you can combine them, you should do so.

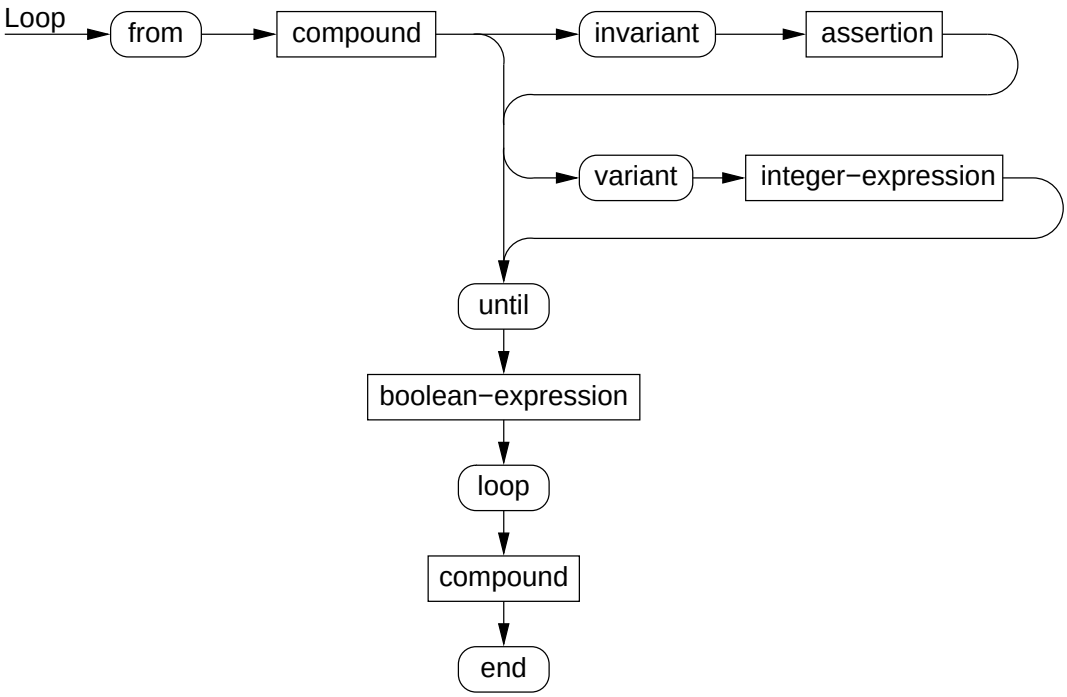
You should also make sure any element appears only as many times as necessary. e.g. the following is not the most concise diagram as the semicolon terminal appears twice.

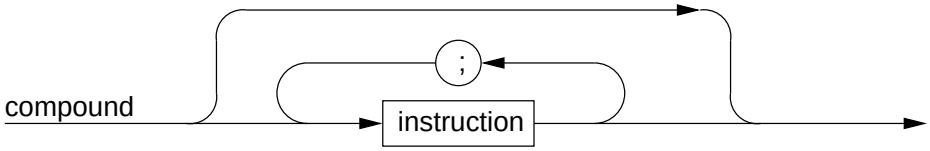


Question

Rewrite the following syntax diagrams as a BNF grammar. Do not attempt to supply productions for those non-terminals which are not defined in these syntax

diagrams.





Solution

Let's first look at rewriting these as an EBNF grammar. This is a much simpler task that trying to convert to BNF immediately.

We'll start with the <loop> productions. Although this is the longer syntax diagram it is actually easier to translate, as the rule is a just straight sequence with two optional parts in the sequence. There is no nesting or iteration in the rules.

In EBNF we denote optional elements by enclosing them in square brackets. If we treat the two options, <invariant> and <variant>, as non-terminals, the EBNF rule is:

<Loop> →
  '`from`*'* *<Compound>* [*<invariant>*] [*<variant>*] *'*`until`*'* *<Boolean-expression>*
  '`loop`*'* *<Compound>* *'*`end`*'*

The rules for invariant and variant are straight-forward.

<invariant> → *'invariant'* *<assertion>*

<variant> → *'variant'* *<integer-expression>*

Now we'll look at the <compound> rule. A <compound> consists of zero or more <instruction>s. Another way of saying this is that anything that appears on the

RHS of the production is optional. So to cover the zero case we simply include the entire RHS of the rule in square brackets.

Every instruction, except the last, is followed by a semi-colon. Or, alternatively, a <compound> consists of an instruction, followed by zero or more semi-colon <instruction> semi-colon sequences. The rule has the same form as that used for a Pascal <statement-sequence>, which you saw on page 26 of this book.

<compound> → [*<instruction>*{ *';' <instruction>*}]

We now have an EBNF grammar which describes the same language as the syntax diagrams. However, the question asked us to translate into BNF. So we must now remove all the extensions to BNF we have used in our solution so far.

The rule for <Loop> includes square brackets to indicate optional elements. To remove these we could simply introduce three choices.

<Loop> → *'*`from`*' <Compound> <invariant> <variant> '*`until`*'*
<Boolean-expression>'`loop`*' <Compound> '*`end`*'*
| *'*`from`*' <Compound> <invariant> '*`until`*'*
<Boolean-expression> '`loop`*' <Compound> '*`end`*'*
| *'*`from`*' <Compound> <variant> '*`until`*'*
<Boolean-expression>'`loop`*' <Compound> '*`end`*'*
| *'*`from`*' <Compound> '*`until`*'*
<Boolean-expression>'`loop`*' <Compound> '*`end`*'*

Notice how much repetition there is in each rule. Perhaps it would be better to introduce some other rules to define the repeated parts and make this easier to follow. A <Loop> seems to consist of a loop-header, followed by the optional parts, followed by a loop tail. So let's try that. First define the <Loop-header> and the <Loop-tail>.

<Loop-header> → *'*`from`*' <Compound>*
<Loop-tail> → *'*`until`*' <Boolean-expression>'loop' <Compound> '*`end`*'*

Now we use these new rules in the definition of <Loop>

<Loop> → *<Loop-header> <invariant> <variant> <Loop-tail>*
| *<Loop-header> <invariant> <Loop-tail>*
| *<Loop-header> <variant> <Loop-tail>*
| *<Loop-header> <Loop-tail>*

All that's left is to remove the EBNF extensions in the rule for <Compound>. A <Compound> may contain infinite list of semi-colon <instruction> sequences. You have already seen how to recursively define such infinite lists in Module 3.

<Instruction-list>' <Instruction> ';' | *<Instruction> ';'<Instruction-list>*

That takes care of the sequence of instructions. Now, a <Compound> is either an <Instruction-list> followed by a final <instruction> which has no semi-colon, or a single <instruction> with no semi-colon. We just need to specify as alternatives.

Compound → *<Instruction>* | *<Instruction-list> <instruction>*

Have we finished? Well, not quite. Our BNF grammar does not allow a null string as one production of a <Compound>. Some texts use a special symbol, ϵ, to stand for the empty string of symbols. We shall use this symbol also. The alternative is that we must redefine the rules for <Loop> so that <Compound> becomes optional.

A complete BNF grammar that describes the same language as the given syntax diagrams appears below.

$$
\begin{array}{rcl}
\langle \textit{Loop} \rangle & \rightarrow & \langle \textit{Loop-header} \rangle\ \langle \textit{invariant} \rangle\ \langle \textit{variant} \rangle\ \langle \textit{Loop-tail} \rangle \\
& | & \langle \textit{Loop-header} \rangle\ \langle \textit{invariant} \rangle\ \langle \textit{Loop-tail} \rangle \\
& | & \langle \textit{Loop-header} \rangle\ \langle \textit{variant} \rangle\ \langle \textit{Loop-tail} \rangle \\
& | & \langle \textit{Loop-header} \rangle\ \langle \textit{Loop-tail} \rangle \\
\langle \textit{invariant} \rangle & \rightarrow & \text{`}\texttt{invariant}\text{'}\ \langle \textit{assertion} \rangle \\
\langle \textit{variant} \rangle & \rightarrow & \text{`}\texttt{variant}\text{'}\ \langle \textit{integer-expression} \rangle \\
\langle \textit{Loop-header} \rangle & \rightarrow & \text{`}\texttt{from}\text{'}\ \langle \textit{Compound} \rangle \\
\langle \textit{Loop-tail} \rangle & \rightarrow & \text{`}\texttt{until}\text{'}\ \langle \textit{Boolean-expression} \rangle\ \text{`}\texttt{loop}\text{'}\ \langle \textit{Compound} \rangle\ \text{`}\texttt{end}\text{'} \\
\langle \textit{Instruction-list} \rangle & \rightarrow & \langle \textit{Instruction} \rangle\ \text{`;'} \mid \langle \textit{Instruction} \rangle\ \text{`;'} \langle \textit{Instruction-list} \rangle \\
\langle \textit{compound} \rangle & \rightarrow & \langle \textit{Instruction} \rangle \mid \langle \textit{Instruction-list} \rangle\ \langle \textit{instruction} \rangle \mid \epsilon
\end{array}
$$

Additional comments

1. It is not necessary to translate to EBNF first. I have done so for two reasons.
   (a) to demonstrate how much more direct this process is than translating to BNF
   (b) some of you may find it helpful to do the translation in two steps, rather than directly.
2. It is quite common to have to introduce new non-terminals when translating from railroad diagrams. Recall that in the previous example, when translating from EBNF to syntax diagrams we combined several rules into one diagram.
3. There is not necessarily a unique answer e.g. when introducing new non-terminals your choice may be different from mine.

4.3 Summary

In this module we began by looking at some problems that may arise when formally describing a language. In particular we saw examples of ambiguities, and the related issues of describing operator precedence and associativity.

We also looked at some extensions to BNF, known as EBNF and saw that BNF grammars can be represented graphically using syntax diagrams.

Parse trees for sentences generated using the BNF rules of a language, graphically describe the syntactic structure underlying the generated sentence. A BNF grammar and parse trees can also assist in recognising sentences written in the language, and allow lexical analysers and compilers to constructed relatively easily. This topic is addressed in the following module.

Module 5

Lexical analysis and Parsing

Overview

Recognising and translating strings belonging to a language is a fundamental computing application. It is not restricted to programming language compilers, as many applications in all computing application domains exhibit an interface language which must be parsed and analysed.

This module presents an overview of this parsing process. While not designed to make you instant experts in the field, enough information is presented to enable you to write a recursive descent parser, which is often a quite acceptable parsing technique.

5.1 Lexical Analysis

A program, or sentence of a language, is just a character string, though it may be hundreds of thousands of characters long. Lexical analysis is the process of taking this string as input and returning as output a list of *tokens*. A token is a computer representation of a *lexeme*. Lexemes are just the terminal symbols of the grammar. They include identifiers, punctuation symbols like `;` `.` `(` }, operator symbols (e.g. `+` `*`), and reserved words (e.g. `begin` `end`).

Reading Sebesta sections 4.1.and 4.2

Although lexical analysis can usually be performed with standard language parsing tools and included as an integral part of the language parser, Sebesta lists some very good reasons for writing separate lexers. One reason that he does not mention is of particular relevance to functional programmers: namely that the layout rule employed by Haskell and Miranda cannot be encoded in a context free grammar and so cannot be analysed by standard parsing tools.

Writing lexers is relatively straightforward and mechanical. The difficulty is in making sure that the lexer actually reflects the lexical rules of the language. Drawing a state transition diagram to describe these rules, and then writing code to implement it, is probably the best way of ensuring that the lexical rules have been captured in the lexer.

5.2 Parsers and grammars

Conceptually, a parser for a language takes a sentence represented by a sequence of tokens and produces either a parse tree corresponding to that sequence, or a message indicating that the sentence is not in the language.

A large number of algorithms for constructing a parser given a CFG for a language have been proposed. The most general parsing algorithms will accept any legal CFG, but they perform poorly compared to more specialised parsing algorithms which are designed to work with restricted forms of CFG.

The most commonly use parsers are the *top-down* and *bottom-up* parsers.

A top-down parser takes a sentence and attempts to build the parse tree from the top down and from left to right. That is, the root node and its sub-nodes are recognised before the leaves. Such parsers are often referred to as *LL* parsers. The first L means that the input string is scanned **L**eft to right, and the second L indicates that a **L**eftmost derivation of the parse tree is being produced. The recursive descent parser described in Sebesta is a LL(1) parser: the number indicated that the parser need only look at a single token (the next one) of the input to decide what parsing action to take. Grammars which can be parsed using a LL(1) parser are sometimes called LL(1) grammars. See section 5.3.1 for more on the features of LL(1) grammars.

A bottom-up parser takes a sentence and attempts to build the parse tree from the bottom up and from right to left. That is, leaf nodes and their associated non-terminal parent nodes are created first and are assembled to form the complete parse tree, with the root node being added as the last step. Such parsers are often referred to as *LR* parsers. As before the strings are scanned left to right, but the R indicates a **R**ight to left derivation of the parse tree. The commonly used shift-reduce parsing algorithm is a LR(1) parser — it requires just one lookahead token.

Reading Sebesta section 4.3

5.3 Top down parsing

Consider the simple grammar

<S>	→	a <B> <C>	(1)
<B>	→	b <B>	(2)
	|	d <B>	(3)
	|	e	(4)
<C>	→	c	(5)

The productions are numbered for later reference. Now consider the sentence of the grammar "abdbec". A leftmost derivation would look like:

<S>	⇒	a <B> <C>	prod. 1
	⇒	a b <B> <C>	prod. 2
	⇒	a b d <B> <C>	prod. 3
	⇒	a b d b <B> <C>	prod. 2
	⇒	a b d b e <C>	prod. 4
	⇒	a b d b e c	prod. 5

where each derivation step is labelled with the production that was used to expand

the leftmost non-terminal.

The top-down parse algorithm recreates this leftmost derivation. The algorithm maintains a single local state variable that holds the current sentential form of the derivation sequence. At each iteration the leftmost non-terminal of the current sentential form is replaced by the RHS of a production for that non-terminal. Often there is a choice of productions for a single non-terminal. The next token of the sentence being parsed is used to determine which production to choose.

The following table illustrates the steps a top down parser would take in parsing the string "abdbec". Note that after a token is used to select a production, it is replaced by the next token.

Next token	Current Sentential Form	Production	New Sentential Form
a	<S>	1	a<B><C>
b	a<B><C>	2	ab<B><C>
d	ab<B><C>	3	abd<B><C>
b	abd<B><C>	2	abdb<B><C>
e	abdb<B><C>	4	abdbe<C>
c	abdbe<C>	5	abdbec

This is a simplified view of the process but is completely accurate. Notice that the steps precisely correspond to those of the leftmost derivation.

Reading Sebesta section 4.4.

Sebesta shows how to write a recursive descent parser. A function is written to recognise the tokens in each non-terminal. Each step of the recursive descent parser, where a parsing function is called to parse a phrase derived from a non-terminal, corresponds to a single step in the above algorithm.

5.3.1 LL(1) grammars

Recursive descent parsers are very natural and easy to write. Unfortunately they require that the corresponding grammar be written with some care. A grammar for a LL(1) parser cannot contain

- left recursion or
- common prefixes.

Left recursion is exhibited by rules like

<A>→<A> a | a

which lead to a parser which recurses indefinitely. The common prefix is seen in simple rules like

<A>→a <X> | ab

The problem is that the parser cannot know *with a single lookahead symbol 'a'* which production of <A> to choose. This problem can be avoided by implementing n-token lookahead; parsers which do this are called LL(n) parsers but require more sophistocated algorithms.

Both the above grammar features can be eliminated by careful grammar writing. There are standard algorithms which describe how to rewrite grammars to remove left recursion and common prefixes, but they are beyond the scope of this course. Any text on compiler design will describe these algorithms.

5.4 Bottom up parsing

We introduce the idea of bottom up parsing by giving an example of parsing a sentence from the example grammar of the previous section using the shift-reduce parser algorithm. A shift-reduce parser operates using a stack of symbols. The sequence of symbols on the stack always represents a sentential form from the right derivation of the sentence. At each step either a terminal symbol is shifted onto the top of the stack, or a phrase equivalent to a production's RHS on the stack is replaced by (reduced to) the LHS of that production. This is rightmost derivation in reverse.

Here are the steps for parsing "abdbec". The stack is represented by a list. The top of stack is to the right.

Next token	Stack	Action	New Stack
a	*empty*	shift	a
b	a	shift	ab
d	ab	shift	abd
b	abd	shift	abdb
e	abdb	shift	abdbe
c	abdbe	reduce (4)	abdb<B>
c	abdb<B>	reduce (2)	abd<B>
c	abd<B>	reduce (3)	ab<B>
c	ab<B>	reduce (2)	a<B>
c	a<B>	shift	a<B>c
_	a<B>c	reduce (5)	a<B><C>
_	a<B><C>	reduce (1)	<S>

It is interesting to note that the top-down parse completes when the sentence is fully derived, while the bottom-up parse completes when the root symbol is produced on top of the stack.

Reading Sebesta section 4.5.

The material in this section of Sebesta is a little more complex than the simple example just presented, but will repay careful study. In particular note that it is not in general a simple matter to identify which phrase should be reduced at the reduce step.

LR parsers are too complex to construct by hand, so we commonly use a program which takes a grammer as input and produces the parser as output. So, even though building a shift-reduce parser by hand would be unreasonably complex, using a so-called "compiler-compiler" like YACC is usually easier and faster than building a recursive descent parser if the grammar is of a significant size (say greater than 20 productions).

5.5 Summary

Parsing is a very large topic. It was one of the earliest research topics in computer science, and by now a number of mature solutions have been developed and extensively used. While the theory behind parsing technology can be quite complex and difficult to understand, fortunately writing parsers for the simple languages you are likely to wish to translate is quite straightforward using either recursive descent parsing or using a parser generator.

Exercises

1. Complete questions 3, 9, 12, and 15 from the Review Questions in Chapter 4 of Sebesta. Check your answers by referring to the text.

2. Complete questions 1 and 5 from the Problem Set in Chapter 4 of Sebesta.

Module 6

Introduction to Data Types

Overview

The concepts of types and type-checking play a major role in the imperative programming languages. You should all be familiar with the idea of variable declarations which, in most languages, involve a name and a type. In this module we study variables and their attributes, then we take a closer look at related issues such as binding, type checking and scope. Finally we look at the primitive data types provided by most languages.

6.1 Variables

Variables are a major feature of the imperative programming languages, providing the means by which such languages can address memory. Each variable has six attributes associated with it; name, address, value, type, lifetime and scope.

Reading Sebesta Chapter 5, to the end of Section 5.3.

You should take particular note of the distinction made between keywords and reserved words. The descriptions of some languages use these two terms interchangeably. In this course we will maintain the difference.

6.1.1 Binding, Scope and Lifetime

In most of the languages you are familiar with, identifiers or symbols may mean different things in different parts of the program. This ability for the same name or symbol to have different semantics is directly related to the concepts of binding, lifetime and scope.

Reading Sebesta Section 5.4.

Binding is just an association e.g. a variable identifier is associated with a particular memory cell, or a variable will be bound to be a particular data type. The terms static and dynamic binding are used to refer to the time at which such bindings occur, either prior to, or during execution. When referring to type bindings we usually drop the word binding and simply talk about static and dynamic typing.

Note that the dynamic binding referred to here is not quite the same as the dynamic binding provided by some object-oriented languages.

Note carefully that the type of a variable may be statically bound and it's storage dynamically bound. This is the case with both stack dynamic and explicit heap dynamic variables. We will look at this process in more detail later in the course.

6.1.2 Scope and Lifetime

Reading Sebesta Sections 5.8, 5.9 and 5.10.

The issue of scoping is of major importance to the block structured languages.These languages are usually statically scoped i.e. a variable's visibility is determined at compile-time, based on where it's declaration appears in the source code. This has the advantage of allowing the programmer to restrict access when required. However, as Sebesta points out in his evaluation of static scoping, there are also some potential problems.

Despite these problems, scope is still determined statically by the vast majority of languages. The disadvantages of the alternative, dynamic scoping, render it far less reliable, so static scoping remains more common.

As Sebesta points out, scope and lifetime are two different, unrelated concepts, although a pseudo-relationship between the two is apparent in some languages. There is, however, a definite relationship between the scope in which a program statement lies and its referencing environment.

6.1.3 Variable Initialisation

So far we have looked at binding a variable to storage and to type. One further binding that can occur is the binding of a variable to a value.The binding of a variable to a value at the same time it is bound to storage is called initialisation. If a "variable" is bound to a value only at the same time it is bound to storage, it is a constant.

Reading Sebesta Section 5.11.

6.2 Type Checking

As we have already seen, one of the attributes of a variable is its declared type. When declaring a variable to be of a particular type, we are actually making a statement about the operations we are permitted to perform with that variable. Type checking is the process of determining whether or not the operations we wish to perform using a particular variable, are permissible with an object of that type.

The kind and extent of type checking affects greatly the reliability of a language. In general, languages with type systems do not permit type errors will remove the opportunity for a programmer to write code containing certain kinds of errors.

A *type error* occurs when a function (an operator is a function, usually of two arguments) is passed parameters whose values are of the wrong type. How can this happen? Consider the C skeleton:

```
union  { int a;
         float *b;
       } s;

float f = 1.23;
int   n;

int main(){
        s.b = &f;
        n = s.a + 1;
}
```

In the final assignment the integer `n` is assigned the value of the integer `s.a` incremented by 1. Unfortunately, the union `s` currently holds a pointer to a floating point number. This is a type error.

The undiscriminated union is the prime reason why many languages are not strongly typed.

A type error does not always result in a runtime error. The effect may be a some error in computation occurring well after the type error, which will be quite hard to detect. If all type errors generated an immediate runtime error message they would not be so dangerous and difficult to detect.

In *strongly typed languages*, all type errors are detected.

Languages that provide *implicit type conversions* (also called a *coercion*), like C, can fail to detect typographic errors. For instance, if we write

```
int a,b;
float c;

a = a+c;
```

instead of the intended `a = a+b` then the compiler will not object. It will convert `a` to a `float`, add two `float`s, then convert the resulting `float` result to an `int`. If we mistakenly typed `c` instead of `a` in a language like Pascal that does not perform automatic conversions, then a compiler error indication incompatible types would be reported.

Reading Sebesta Sections 5.5 to 5.7.

It is not necessary that a language declare all types to be strongly typed. Miranda and ML are both strongly typed, but can use type inference to determine type. However, these are not imperative languages, although ML does has some imperative features.

6.3 Primitive Data Types

The previous discussion of type checking did not consider what data types are. A data type is an abstraction that describes the common properties of a set of

similar data objects. e.g. The C data type *int* describes data objects which can be manipulated using the arithmetic operators +, - and * while *float* objects can also be manipulated using /.

The data a computer manipulates should represent as closely as possible the real world data structures being modelled. Most languages provide data types such as integers, reals and characters, however, much of the world does not consist of only these simple types. As many data objects are actually combinations of these simple types, today's languages usually provide ways of constructing new types which are defined in terms of other types.

Primitive data types are not defined in terms of other types. The primitive data types provided by most languages are integer, floating point, character and boolean. Some languages provide other primitive types e.g. C has several variations of integer and floating point types;*int, short int, long int, real, double* etc.

Character strings are usually stored as arrays of characters, however in FORTRAN and BASIC strings are provided as a primitive type.

Reading Sebesta Sections 6.1, 6.2 and 6.3.

You should make sure you are comfortable with the use of a descriptor to store the attributes of a variable. Being able to produce a descriptor for various types can help you to understand the issues involved in implementing those types.

6.4 Summary

In this module we considered the six attributes usually associated with variables; name, address, value, type, lifetime and scope.

The association of an object with a particular attribute is called binding. Variables may be associated with their attributes either statically or dynamically.

Most languages have reserved words which are not available for naming variables. Some languages also have keywords which have a special meaning. Keywords may be used to name variables, but then the special meaning is lost.

Some languages permit aliases, where the same storage location can be referred to in different ways. There are a number of potential problems with aliasing. We will discuss this further in a later module. We also saw that static scoping is a feature of most imperative languages.

Typing determines which operations may be performed on a variable. Types need not always be explicitly declared. Strong typing means all type errors can be detected. In Module 7 we will see some ways languages circumvent their typing rules.

Finally we discussed the primitive types provided by modern languages.

Exercises

1. Complete questions 2, 8, 9, 10, 11, 12, 13 & 14 from the Problem Set in Chapter 5 of Sebesta.

2. Consider the following skeletal C program.

```
int x, y, z;
fun1 () {
      int i, j, k; {
              int m, n, y;
              ...
              }
}
fun2 () {
      int x, z, l;
      ...
}
fun3 (int j, k){
      int m, n;
      ...
}
main () {
      ...
}
```

(a) Draw a box around each block, and label the blocks A, B, C etc.

(b) Name a block that is nested within another block.

(c) Are the global variables `x`, `y` and `z` accessible in `fun2`? Why or why not?

(d) In which blocks does x refer to a global variable? A local variable?

(e) In which blocks can we use both `n` and `k`?

Solutions to Selected Exercises

Exercise 1 Q 8 (a) (i) sub1 (ii) sub1 (iii) main

Q 8 (b) (i) sub1 (ii) sub1 (iii) sub1

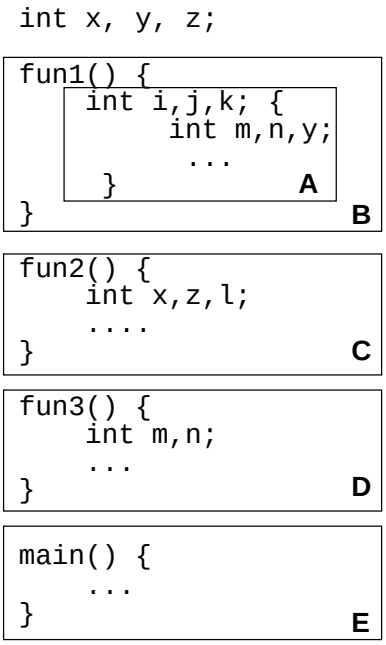
Q 10

sub1: a, y, z in sub1; x in main
sub2: a, b, z in sub2; y in sub1; x in main
sub3: a, x, w in sub3; y, z in main

Q 13 (a) a main, b fun1, c fun2, d, e, f fun3

Q 13 (b) a main, b, c fun1 d, e, f fun3

Exercise 2



(ii) B

(iii) y is accessible, x and z are not as the local declaration hides the global variables of the same name.

(iv) x refers to the global variable in blocks E, A, B, D. The x in C is local.

(v) A, D

Module 7

Structured Data Types

Overview

In Module 6 we studied the primitive types provided by most of today's programming languages, however, it was also mentioned that many of the data objects we will be want to manipulate are actually structured combinations of these types. In this module you will study the most common mechanisms available to the user for constructing structured data types.

7.1 Ordinal Types

Ordinal types define a range of possible values that can be associated with the set of possible integers. The two kinds of ordinal types commonly available are enumeration and sub-range types. An enumeration type defines a set of possible values where each of those values is a symbolic constant. e.g. In C we can define the following enumeration type.

```
enum weekday {Mon, Tues, Wed, Thurs, Fri}
```

Subrange types define a contiguous sequence of ordinal values, e.g. In Pascal we can define the following sub-ranges.

```
type uppercase = 'A'..  'Z';
```

Types such as these are used to enhance readability of source code.

Reading Sebesta Section 6.4.

7.2 Compound Types

Today's languages provide a variety of ways in which the simple types can be combined to form structured types. The most common structured types are arrays and records. Some languages also provide set and union types.

In this section we discuss some of the issues involved in implementing these structured types, and look at examples of their implementation in various languages.

7.2.1 Arrays

You should all be familiar with the array concept. An array is a fixed length sequence of homogeneous elements. The value of the first and last possible position are the lower and upper bounds of the array. Elements of the array are accessed by position; i.e. by appending an array subscript to the array name.

Reading Sebesta Section 6.5 up to the end of sub-section 6.5.8.

Most languages use very similar syntax for array references. Although subscripts are usually integer types, some languages do allow other types.

Allocation of storage is either static, fixed stack-dynamic, stack-dynamic or heap-dynamic. Be sure you understand the difference between these storage allocation categories. We will meet them again later in this module.

Sebesta discusses other issues such as; most languages place no limit on the number of dimensions, some languages allow initialisation of arrays at declaration, whether or not operations are permitted on an array as a unit, and the number and type of permitted array operations.

The section on array slices should be read for further insight into subscripting, but will not be examined.

7.2.1.1 Implementing Arrays

As discussed previously, array elements are accessed by subscripting the array name with the position of the desired element. The address of the desired element must be computed using the base address of the array and the size of the elements in the array. The actual function used for this computation will depend largely on the storage category and the number of dimensions in the array. The design of the access function can adversely affect execution speed.

Reading Sebesta Section 6.5.9.

7.2.1.2 Associative Arrays

Associative arrays are a language feature which implements the familiar lookup table. An associative array is just an array of (key, value) pairs. You can add to, delete from, and look up entries in an associative array. The advantage of having an associative array as a built in language feature (rather than using some library routine which you may have written yourself) is that the syntax can be nicer, and you can be assured that it has been well implemented.

Reading Sebesta Section 6.6.

7.2.2 Record Types

Arrays provide a useful means of grouping data elements of the same type. However, we also often want to model collections of non-homogeneous data. Record types provide a reasonably straightforward solution to this need.

Reading Sebesta Section 6.7.

You may not have realised before just how closely records are related to arrays. You should read and understand Sebesta's comparison of these two types.

7.2.3 Unions

Unions describe a type that can store values of different types at different times during execution. Such types allow programmers a greater degree of flexibility, however, their inclusion in many languages provide a means of circumventing the type system.

Reading Sebesta Sections 6.8.

7.3 Pointer Types

A pointer type describes a class of objects whose values are the locations of other data objects. Pointers are not themselves a compound type, but may be used to link together as many component elements as the programmer wishes. There are three basic language features needed to make this possible.

- The data type pointer. A pointer object will contain either the address of another object of the null pointer e.g. `nil` in Pascal. If a pointer object has the null pointer as its value, this simply means that the pointer object is not ready for use (i.e. it does not contain the address of another data object).

- An operation that can be used to create data objects of fixed size, e.g. `new` in Pascal, `malloc` in C. The corresponding operation to destroy these dynamically created objects (e.g. `dispose` in Pascal) is not always provided.

- A dereferencing operation that allows recovery of the data object to which the pointer points e.g. `^` in Pascal.

There are two possible specifications for pointer types; either a pointer may reference objects of a single type, or a pointer may be able to reference objects of any type. In the latter case static type checking is not possible. If we allow the same pointer to reference objects of different types at different times throughout execution, then obviously we cannot determine the type of the object being referenced at compile time. For this reason, most languages adopt the former approach.

7.3.1 Dangling Pointers and Lost Objects

As a pointer simply contains a storage address, it is obviously quite possible to have several pointers that point to the same address. So the question arises of what should happen when one part of the program deallocates the storage being pointed to? If any of these other pointers are subsequently used to access the address, only garbage will be found, with possibly disastrous effects on the program. Such pointers are known as "dangling pointers".

There are two different methods that may be used to overcome these problems. One method uses the "locks and keys" approach, and the other uses "tombstones".

7.3.2 Implementation

Two representations are used for pointer values: the actual address of the object (absolute addressing), or an offset from the base address of the heap (relative addressing). Absolute addressing is efficient, as the pointer provides direct access, however, storage management becomes difficult as it is not possible to move objects if there are pointers to it stored elsewhere, unless we can also change those pointers to reflect the object's new position.

With relative addressing, an area of storage, known as the heap, is set aside for storage of dynamic objects. The starting address of the heap is called its base address. To access a data object, the absolute address is determined by adding the offset stored in the pointer variable to the base address of the heap. This is more costly than direct access. Now, however, it is possible to move the entire heap to another location, and none of the pointers need to be changed. Only the base address of the heap changes; the offsets stored in the pointers are still correct.

It is important for you to appreciate the complexities involved in heap management, so please read the followingsection carefully.

Reading Sebesta Section 6.9.

7.3.3 Garbage collection

The failure to release heap storage that is no longer needed by an application is a major source of error in the C family of languages. Programmers are expected to deallocate all memory that they dynamically allocate. As this often needs to occur at a very different location in the code to the allocation call, it is not surprising that this error is commonly made.

The solution adopted by many modern languages is to include a run time garbage collector system. This memory management subsystem automatically detects and frees unused dynamic memory. Java is an example of such a language that depends upon garbage collection.

7.4 Summary

In this module we looked at the most common structured data types provided by today's languages. These included arrays, records, sets and enumerated types. We also saw that there is much in common between arrays and records.

In the latter part of the module we discussed pointers, which are not themselves structured data types, but do provide a means of linking together arbitrary numbers of components. The inclusion of pointer types into a language raises a number of issues such as how should "dangling pointers" or "lost objects" be guarded against, and the problem of managing storage efficiently.

Exercises

1. Complete questions 1, 2, 5, 8, 9,10, 19, 20, 22, 23, 24, 27 and 28 from the Review Questions in Chapter 6 of Sebesta. Check your answers by referring to the text.

2. Complete questions 4, 5, 7, 9, 12 and 14 from the Problem Set in Chapter 6 of Sebesta.

3. In the language SIMSCRIPT a multidimensional homogeneous array is represented as a vector of pointers that point to other vectors of pointers etc., to as many levels as there are dimensions in the array. e.g. A 3 x 4 matrix of integers is represented by a vector of three pointers, each pointing to a vector of four integers. Give an algorithm for accessing A[i,j] for this representation. How does the efficiency of accessing using this representation compare with the usual sequential representation?

Solutions to Selected Exercises

Exercise 2

Q 5 How do you access the entire record? In Ada you have to use the keyword all to specify you want the entire record rather than a particular field e.g.

```
type employee is
    record
        name : string(1..30);
        age : integer range 15 ..100;
    end record;

type emp_ptr is access employee;
t1 : emp_ptr;

t1 := new(employee);
```

To access just one field, use the standard dot notation e.g. `t1.age`, but to access the entire record you must use `t1.all`.

Q 7 The constructs `(*p).field_name` and `p->field_name` are equivalent. The `->` operator combines the effects of `.` and `*`. the combined operator is somewhat easier to write than the other construct; by using an explicit operator the combined operation is clearly indicated as such; the form of the operator implies its effect i.e. follow this arrow to access the specified field.

Module 8

Expressions, Statements and Flow of Control

Overview

Programs written in imperative languages consist of expressions and statements to be executed. These expressions and statements are executed sequentially, unless some control structure is present to redefine the order of execution. In this module we focus on the semantics of expressions, control structures and the various possible combinations that may be formed.

8.1 Expressions

An expression is a nest of function calls or operators that return a value. Expressions are the means by which computations are specified. The associativity and precedence rules of a language determine order of evaluation within operator expressions. Expressions may be classified as arithmetic, relational, boolean or conditional.

8.1.1 Function Evaluation

The usual form of a function call is a function name, a list of formulas representing actual arguments and an indication that the function should be applied to those arguments. Many languages also permit the arguments to contain function calls, thus producing a nest of calls.

For traditional languages the rule for evaluating elements within a function call has been to determine the value of every argument before beginning to evaluate the function. Arguments may be evaluated left-to-right, right-to-left or in some other order determined by the writer of the particular compiler.

The newer functional languages such as Miranda and Haskell (see Module 2) start evaluating the function first. Arguments are evaluated only when the value of that argument is required. The value of the argument is then saved in case it is needed again. This is often called lazy-evaluation.

8.1.2 Operator Expressions

Operators may be thought of as syntactic variants of functions. Both describe an action to be applied to some objects (arguments) to produce another object (the result). One difference, however, is that while functions may have any number of arguments, operators are usually limited to one or two.

An operator with two arguments is called a binary operator. Binary operators are usually called using infix notation. Single argument operators are known as unary operators and are generally called with prefix syntax, but postfix is also possible. e.g. C has both prefix and postfix unary operators, binary operators, and a two part conditional operator (?) with three operands, called a ternary operator.

As we have already seen in Module 4, the order of evaluation of operators is determined by the language's precedence and associativity rules. The language designer determines an hierarchy of precedence levels and assigns a level to each operator. Operators with the highest precedence are evaluated before those of a lower level. If an expression contains adjacent operators of the same precedence, then the associativity rules determine which operator is evaluated first. Many languages allow the precedence and associativity rules to be altered by enclosing sub-expressions that must be evaluated first in parentheses.

8.1.3 Side Effects

When an expression contains functions as operands, the order of evaluation of the operands becomes an important consideration, especially if functions are permitted to have side effects.

A function has a side effect if it can change the value of it's arguments, or change the value of a global variable. In either case, the variable passed as an argument, or the global variable, will have a different value after the function completes than it had before the function began executing. Clearly then, if both the function and the variable being changed appear as operands in an expression, then the result of the expression will differ depending on whether the function is evaluated first, then the variable's value is accessed, or the evaluations are carried out in the reverse order. In the former case the variable will contain the changed value, in the latter the original value.

8.1.4 Coercion

As we saw in Module 6, the declared type of a variable determines what operations are permitted on that variable. Now consider the arithmetic operators +, - and *. These should be legal operations for a variable of any numeric type. Also, there will be occasions when a program may need to perform an arithmetic operation with operands of differing numeric type. The question arises though whether different numeric types (e.g. reals and integers) may be mixed in the same expression, and if so, what is the type of the result?

Languages that allow such mixed-mode expressions, must define a set of rules to govern the implicit type conversions required. If mixed-mode expressions are not allowed, explicit type conversions can be provided, allowing conversion between different types.

Reading Sebesta Sections 7.1 to 7.6.

The short-circuit evaluation of expressions discussed in Section 7.6 of Sebesta, is similar to the lazy evaluation of functions mentioned above in Section 8.1.1.

8.2 Assignment Statements

As we saw in Module 1, the central feature of the imperative languages is the use of variables. As a consequence, one of the central constructs in an imperative language is the assignment statement, whereby a value is assigned to a variable.

You may wonder why you need to study assignment statements. The form of the simple assignment statement in the imperative languages has the same general syntax:

<*target_variable*><*assignment_operator*><*expression*>

However, some languages provide additional forms e.g. multiple targets, and some treat assignment as an expression, rather than a statement.

Reading Sebesta Sections 7.7 and 7.8.

Exercises

1. Complete questions 3, 4, 6, 7, 8 and 14 from the Review Questions in Chapter 7 of Sebesta. Check your answers by referring to the text.

2. Is there a difference between operators and functions? Explain your answer.

3. Complete questions 4, 5, 6, 9, 10 and 13 from the Problem Set in Chapter 7 of Sebesta.

8.3 Sequence Control

Programs written in an imperative language consist of statements and operations that are to be executed. In most languages the physical sequence of statements controls the order of execution. Often, however, we wish to have more control, selecting different paths through the program, or repeating the same block of code a different number of times for different runs. Control structures provide the ability to control the order of execution.

Sequence control structures may be categorised as:

- Structures used within the expressions forming statements, such as precedence rules and parentheses. We have already discussed these methods of sequence control in the first part of this module.

- Structures used between statements or groups of statements such as conditional and iteration statements. These are the structures we will be studying in this second half of this module.

- Structures used between subprograms such as function or procedure calls. We will study these in the next module.

The usual control structures provided at the statement level allow selection between alternative statement sequences, iteration (repetition) of a statement sequence and explicit transfer of control to a named statement i.e. a `GOTO`.

8.3.1 Selection Statements

A selection statement allows execution to proceed via alternate statement sequences, or optional execution of a single sequence. The choice between alternatives is controlled by testing some condition, thus selection statements are also known as conditional statements.

The most common forms of selection statement are the if and case statements. If statements are used to optionally execute a single statement (single-way selection) or to select between two alternate statements (two-way selection). Case statements are multi-way selectors, allowing control of more than two paths.

Reading Sebesta, Chapter 8 to the end of 8.3.

Be sure you understand what is meant by multiple entries/exits to/from a block. Note the differences between nested ifs, else-if sequences and the case forms of multi-way selection. You should also study carefully the effects different forms allowed for conditional expressions may have. Finally, note that C's `break` is a restricted `GOTO`. We will discuss this further in Section 8.3.3.

8.3.2 Iterative Statements

Iteration (or looping) is the basic mechanism whereby a program may execute the same sequence of statements more than once. There are two very different main methods of controlling loops. The first method increments (decrements) a counter each iteration and exits when that counter reaches a terminal value. In the second method, exit from the loop is controlled by a Boolean expression.

Reading Sebesta, Section 8.4.

8.3.3 Explicit Sequence Control

A statement of the form

```
goto NEXT
```

transfers control to the statement labelled `NEXT`. The instruction to branch to `NEXT` is unconditional. Control will be transferred every time the statement is executed. A `goto may` be combined with an if statement to produce a conditional branch such as

```
if condition then goto NEXT
```

where `condition` is a Boolean expression.

Some of the advantages cited for `goto`s are:

- direct hardware support provided labels are simple tags on statements.
- simple and easy to use in small programs.
- completely general purpose and may be used to simulate any other control structure we have already studied.

The disadvantages, which are generally considered to outweigh these advantages, are all related to program comprehension and thus to ease of maintenance. They include:

- Lack of program structure.

  Larger programs are much easier to read and understand if the statements making up the program are organised into groups, with each groups representing a single conceptual unit of the computation. When a design is structured in this way it easier for others to comprehend the design and understand the intent of the original programmer.

- Order of execution no longer corresponds to the physical order of the statements in the code.

  Just as the lack of structure makes it difficult to understand a program, the same can be said of a program where the programmer uses lots of `goto` s so that the program jumps around irregularly.

- The same group of statements may serve multiple purposes.

  Sometimes two separate groups of statements contain a number of statements that are identical. We can use `goto` s so that we only have to write the identical statements once, transferring control to the common statements as required. This is similar in some ways to the use of sub-programs, however, a sub-program computes a clearly defined (and named) part of the entire computation. Using `goto` s in this way creates a program that is less readable and therefore less easily modified than when subprograms are used.

  Although most languages defines labels and goto statements, their use has fallen into disfavour, and they have been eliminated from some newer languages.

Reading Sebesta, to the end of Chapter 8.

Note we will meet guarded commands again when we study concurrency.

8.3.4 Limited GOTOs

Sebesta concludes Section 8.5 by arguing that restricted or limited `goto` s "improve readability because to avoid their use results in convoluted and unnatural code ...". We will now look at some examples where the restricted `goto` is useful.

8.3.4.1 Multiple exit loops

Consider the problem of searching an array containing K elements till the first element that meets some condition is found. Clearly we have two exit conditions;

either the desired element is found, or the end of the array has been reached. To iterate through the elements of an array it is natural to use a for loop, so the desired form is:

```
for I := 1 to K do
   if THIS_ARRAY[I] = 0 then goto outside
....
outside:
```

We have already seen that to do this in Pascal we must use a `while` loop, or we can use the above code with a `goto` statement. When we use the `while` loop version, the information about the size of the array becomes obscured.

Ada's `exit` and C's `break` statements provide a useful means for expressing these kinds of loop exits.

```
for I in 1..K loop
    exit when THIS_ARRAY(I) = 0;
end loop;
```

8.3.4.2 do-while-do

It can often be more natural to test an exit condition in the middle of the loop, rather than at the beginning or the end. e.g.

```
loop
    read(X)
    if end_of_file then goto outside {outside the loop}
    process(X)
end loop;
```

What we need is the following loop form, which is often called a "do while do" as a `while` at the midpoint will serve our purposes.

```
dowhiledo
    read(X)
    while (not end_of_file)
    process(X)
end dowhiledo
```

No common languages implement such a loop. Ada's `exit when` and C's `if condition break` are close.

8.3.4.3 Exception handling

Exceptions are error conditions such as end of file or subscript out of range. Commonly the code to handle such exceptional conditions is placed in a special place in the program e.g. at the end of the subprogram where the exception might occur. Unless a language has implemented some special form of exception handling, a goto is the easiest way to transfer control from the point in the program where the error occurred.

We will look at exception handling in more detail in Module 12.

Exercises

1. Complete questions 3, 6, 8, 10, 11, 14 and 16 from the Review Questions in Chapter 8 of Sebesta. Check your answers by referring to the text.

2. Explain the difference between `BREAK` and `GOTO`. Why is `BREAK` a superior control structure?

3. Complete questions 4 and 8 from the Problem Set in Chapter 8 of Sebesta.

Solutions to Selected Exercises

Exercise 2 A `GOTO` allows a control to be transferred to anywhere else within the program; a `BREAK` only transfers control to the end of the current block. Thus the `BREAK` does not detract from readability as much as a `GOTO`. In fact using a `BREAK` to control exits from loops can enhance readability as without them the code controlling such exits would need to be more complex.

Exercise 3

Q 4 Unique closing reserved words for compound statements enhance readability by clearly indicating exactly which compound statement is being terminated in a number of nested statements, however this adds to the number of reserved words the programmer must learn to use.

Module 9

Subprograms and Parameter Passing

Overview

Subprograms are one of the most important facilities available in a programming language. In this module we discuss the mechanisms for sequencing between subprograms, and mechanisms for controlling data passing between subprograms.

9.1 Subprograms

A subprogram is a (named) collection of expressions and statements which is invoked explicitly. Any invocation of a subprogram guarantees that control will return to the statement immediately following the call. These properties provide the programmer with a further means of directing flow of control in addition to the control structures discussed in the previous module.

Subprograms may be parameterised. This is the usual way of passing data between subprograms. The parameters in a subprogram declaration are called formal parameters, those in the call are actual parameters. Traditionally parameter information is provided by following the subprogram name with a list of all actual parameters between parentheses. These are called positional parameters. With two or more parameters, this scheme becomes ambiguous. The call `insert (a, b)` may mean insert `a` into `b`, or `b` into `a`, depending on the definition of `insert`.

Improvements over positional parameters include *keyword parameters* and *default parameters*. Default parameters may be omitted from the call. The subprogram definition provides the default value.

Most subprograms fall into the category of either function or procedure. An operator can also be considered a subprogram. A function differs from a procedure in that it returns a value. A function call must therefore appear in an expression rather than as a standalone statement. In most imperative languages the function body must execute a return statement to specify and return the value. e.g. C uses an explicit return statement, while Pascal assigns the value to be returned to the function name.

```
int sq(int n){ return n*n;
             }
```

```
function sq(n: integer):integer;
begin
       sq := n*n
end;
```

Reading Sebesta Sections 9.1 to 9.4.

9.2 Parameter Passing Methods

The simplest way in which data can be passed to a subprogram is to copy the value of the actual parameter then use this as the initial value of the corresponding formal parameter. This mechanism is called call by value or pass by value. As this mechanism transmits data to the formal parameter, but does not return data to the actual parameter, it is said to have in mode semantics. In some languages such as Ada, the in mode semantics are strictly adhered to, and any attempt to change the value of a formal parameter will be produce a compile time error. Other languages, notably C, treat a formal parameter variable just like any other variable within the subprogram i.e. its value may be changed. This does not affect the actual parameter in any way, as only a copy of the value was transferred.

Of course we don't just want to pass data to a subprogram, we also want to have data returned so that the results of any processing carried out by the subprogram are available for use by the rest of the program. The pass by result mechanism has out mode semantics directly opposite to the in mode semantics of call by value. No data is passed to the subprogram when called. The formal parameter is treated just like any other (municipalised) variable. When the subprogram finishes executing, and just prior to control being returned to the caller, the current value of the formal parameter variable is copied to the actual parameter.

Obviously we will not always want to use a parameter in just in-mode or out-mode. We will often wish to pass the value of an actual parameter for use by a subprogram in updating the value of the actual parameter. Such a mechanism would have in-out mode semantics. The simplest way to do this is to combine the above two mechanisms into a pass-by-value-result (call-by-value-result) mechanism. As the name suggests, the value of the actual parameter is copied to the formal parameter, the subprogram may modify the formal parameter, then, just before the subprogram returns, the value of the formal parameter is copied back to the actual parameter.

An alternative to waiting for the subprogram to return before changing the value of the actual parameter is to have any change in the value of the formal parameter reflected immediately in the actual parameter. This can be achieved using pass by reference (call by reference). In call by reference, the formal parameter is an alias for the actual parameter so that everything that happens to the formal parameter also happens to the actual parameter. Pascal and Modula 2 provide call by reference via their `var` parameters.

Note that using pointers as parameters is not the same as pass by reference, although it can be used to achieve the same affect. In fact, call by pointer is actually a special case of call by value. The address stored in the actual parameter is copied to the formal parameter when the subprogram is called. No data is returned to the caller via the parameter mechanism. Instead, because the formal parameter refers to an

address, we can use it to change the value stored at that address. In C, call by pointer must be used to achieve in-out semantics, as C provides only call-by value parameter passing. Note also that, with call by pointer, the pointer variable must be explicitly dereferenced in the subprogram.

Reading Sebesta Sections 9.5 to 9.6.

Sebesta discusses both the implementation of the various parameter passing models, and the advantages and disadvantages of those methods. You should be sure you understand how the various problems such as parameter collision can occur.

A number of issues related to aliasing are also discussed. As a further illustration of the potential for errors consider the following Modula 2 code segment.

```
VAR i: INTEGER; (*declares i of type integer*)
PROCEDURE TestVar (VAR x : INTEGER); (* x is called by reference*)
BEGIN
    WriteInt(i, 1); (* print i, using 1 position*)
    x := 3;
    WriteInt(i, 1); (*print i again*)
END TestVar;
...
...
i := 1;
TestVar(i);
```

Although both calls to `WriteInt` are identical, the first will print a `1`, the second, `3`. If we simply study the code in the procedure `TestVar`, we can see no reason for this difference. There has been no change to `i` between the two calls to `WriteInt` - or has there? In this case, because `x` is an alias for `i`, the statement `x:= 3` has in fact changed the value of `i`.

9.3 Program Composition

We have seen that programs are made up declarations and statements that may be grouped together into blocks and subprograms. These groupings actually fall into four levels of hierarchy:

- blocks;
- procedures, functions and operators;
- modules/packages/tasks; and
- programs

Anything we have discussed about subprograms in general, also refers to procedures. The two main issues specific to functions are whether or not to allow side effects and what types of values can be returned?

In this section we will focus on issues related specifically to user-defined operators and also look at other groupings not provided by all languages.

9.3.1 Overloaded Operators/Subprograms

Operators may be viewed as a different notation for functions with just one or two parameters. In most languages the arithmetic operators are overloaded e.g. we can use + to add both reals or integers or, in some cases, concatenate strings. The correct code to execute is determined by the type of the operands.

In a similar fashion other subprograms may be overloaded i.e. two or more subprograms can have the same name within the same referencing environment. Again the correct version of the subprogram to be executed is determined by the type of the actual parameters and the return type if the subprogram is a function.

Most languages do not permit users to define their own operators, however, both Ada and C++ provide the facility to overload existing operators. This facility has the advantage of increasing orthogonality, allowing the use of operators in natural applications such as + for vector and matrix multiplication. However, careful consideration must be given to precedence and associativity rules for any new operators to prevent resulting expressions from being unreadable.

9.3.2 Modules and Packages

Related declarations of types, variables, functions and procedures can be grouped together in a module or package. Modules may then restrict access to their contents. Usually a module is divided into a specification and an implementation part. Only the specification part, the part that describes the module's interface is visible to the user.

Separation of modules is usually explicit; in Ada the package and end keywords are used; in C they correspond to file boundaries.

We will now look at an example of a small module to implement two operations on dates; `create_date` and `last_of_the_month`. In C we would use two files, `date.h` for the specification part, and `date.c` for the implementation. The contents of `date.h` might be:

```
typedef struct { int dt_year, dt_month, dt_day; } date_type;
extern date_type create_date(int, int, int);
extern last_of_the_month( date_type);
```

The implementation of `create_date` and `last_of_the_month` and any auxiliary functions and data appear in the `date.c` file. In Ada the specification would look like:

```
package Date is type Date_type is private;
    function Create_Date(Y, M, D:Integer) return Date_Type;
    function Last_Of_The_Month(Date: Date_Type) return Boolean;
    private type Date_Type is record
        Year, Month, Day : Integer;
    end record;
end Date;
package body Date is
                -- implementation details go here
end date;
```

The implementation unit is the so-called package body. It may appear in the same file as the above package declaration and the two parts are recognised syntactically.

To use these modules requires a `with Date` clause in Ada and an `#include ‘‘date.h’’` compiler directive in C. In Ada we still need to use the dot notion, `Date.Create_Date`, to access a public name. However, we can also include a `use Date` clause in Ada which makes all public names from the Date package visible and means we can access `Create_Date` directly. (It should be fairly obvious that this can easily lead to overloading.)

Ada uses the `private` keyword to prevent the internal structure of `Date_Type` becoming part of the interface. In C we can use `static` for private names and `extern` for public names.

Modula 2 also provides specific support for module definition. The `export` keyword is used to denote those names that are visible from outside the module. To use a Modula 2 module we use the`import clause`, however, we can actually specify specifically which of a module's public names we wish to import.

```
FROM InOut IMPORT WriteInt;
```

This is a direct contrast to the Ada and C approach where all public names became visible.

We will study packages and modules further in Module 10.

9.3.3 Generic Subprograms

Another two components of program composition are generics and abstract data types. We will discuss generics here.

Consider the advantages of being able to write down common algorithms such as sort routines, or the operations on a linked list without knowing what type of elements will be manipulated, then subsequently putting that code to use by simply specifying the type for the elements we wish to manipulate this time. We write the sort code once, then, when we want to use it, all we do is specify in some way that this time we will sort integers, another time strings etc. We don't have to rewrite the code for each type of element. This facility is provided by generic subprograms.

9.3.4 Separate Compilation

An important issue that arises when large software systems are to be constructed from many separate modules is whether or not the entire system must be recompiled when only a single module is changed. Obviously, such a requirement would considerably lengthen the time to completion of any large project. Fortunately most languages now allow for separate or independent compilation of modules. The separate units are then collected together into a program using the linker.

Reading Sebesta To the end of Chapter 9.

Exercises

1. Complete questions 1, 4, 7, 9, 10, 13, 14, 15 and 17 from the Review Questions in Chapter 9 of Sebesta. Check your answers by referring to the text.

9.4 Implementing Subprograms

As we have already seen, sequencing of subprograms in the imperative languages follows strict controls. Subprograms must be explicitly called, a subprogram must execute to completion at each call, control is transferred to the subprogram immediately it is called and, finally, although subprogram calls may be nested to any depth only one subprogram has control at any one point in time.

To view these controls in a more concrete way: at any one moment only one subprogram, P, is active; this subprogram was called by exactly one other subprogram Q; Q called exactly one subprogram, P; Q is suspended while P is executing; Q was called by exactly one subprogram R; R called exactly one subprogram Q; R is suspended until Q returns; and so on to the top level call in the program.

This creates a chain of one to one relations between caller and callee which can be implemented very efficiently using a stack of activation records.

An activation record holds the local data and linkage information for one subprogram call. In order to completely record the state of an active subprogram we need to know:

1. the current instruction being (about to be) executed the referencing environment, which includes parameters and local variables as well as a static link which provides access to variables in non-local scope.
2. a dynamic link which points to the start of the activation record of the calling subprogram and indicates the point past which stack data must be retained when execution of this subprogram completes.
3. the return address i.e. the address of the instruction to which control will return when the current subprogram finishes.

9.4.1 Procedure Call and Return

When a subprogram is called there are three basic functions to be performed

- the state of the caller must be saved
- a new activation record for the callee must be created
- the callee must be entered in the context of the new activation record

Saving the state of the caller is accomplished by storing the address of the instruction to which control will return in the return address of the callee's activation record, setting the dynamic link in the callee's activation record to the start of the caller's activation record so that the caller's locals stored in its activation record are not destroyed when the callee returns, and, saving the caller's non-locals in the static link of the callee.

Returning from a subprogram reverses the call's effects.

- delete the callee's activation record
- restore the state of the caller

Note these steps cannot be done independently as the information needed to restore the caller's state is contained in the dynamic link in the callee's activation record.

Reading Sebesta Chapter 10 up to the end of Section 10.5.

Sebesta discusses the two major techniques used to implement non-locals in the statically scoped languages; static chains and displays. You should make sure you fully understand the differences between them, as well as the advantages and disadvantages of both.

You should also be able to draw the stack showing the activation records for any uncompleted subprograms at any point during execution of a simple program.

9.5 Dynamic Scoping

The activation records we have just studied relate to static scoping i.e. where the scope of a variable is determined at compile time. With dynamic scoping the situation becomes a little more complex. As a non-local environment will depend on the calling sequence, we need to be able to search through the declarations of all currently suspended subprograms. Sebesta calls this method of dynamic chaining deep access. It is somewhat analogous to following the static chain in the static-scoped languages, however is usually more costly in terms of both storage and speed. Sebesta also discusses an alternative implementation for dynamic scoping he terms shallow access.

Note that the chain of activation records created when a recursive subprogram repeatedly calls itself in a statically scoped language closely resembles a dynamic chain. In both cases the subprogram activation are chained together in order of their dynamic creation during program execution.

Reading Sebesta: Section 10.6.

9.6 Summary

A subprogram defines a procedural abstraction; a group of statements describing a process which is replaced in a program by a call to the subprogram. The benefits of this facility include: the code in the subprogram to be reused without having to be rewritten, and, where the code is too long, chopping into named chunks enhances readability. Two categories of subprograms are usually recognised; functions, which return a value to the calling routine, and procedures. A defining property of the imperative languages is that only one subprogram can be active at any one time; all other activations are suspended.

We also studied the main parameter passing methods and their effects. We saw that the most common parameter passing methods are call by value or call by reference and that call by pointer is a special case of call by value.

Finally we studied a very important concept in implementing subprograms; the activation record. When a new subprogram is called, a new activation record is created which holds all the information relevant to that subprogram. In many languages the activation record is destroyed when the subprogram exits.

Exercises

1. Complete questions 3, 4, 5, 6, 8 and 13 from the Review Questions in Chapter 10 of Sebesta. Check your answers by referring to the text.

2. Complete questions 1, 3 and 5 from the Problem Set in Chapter 10 of Sebesta.

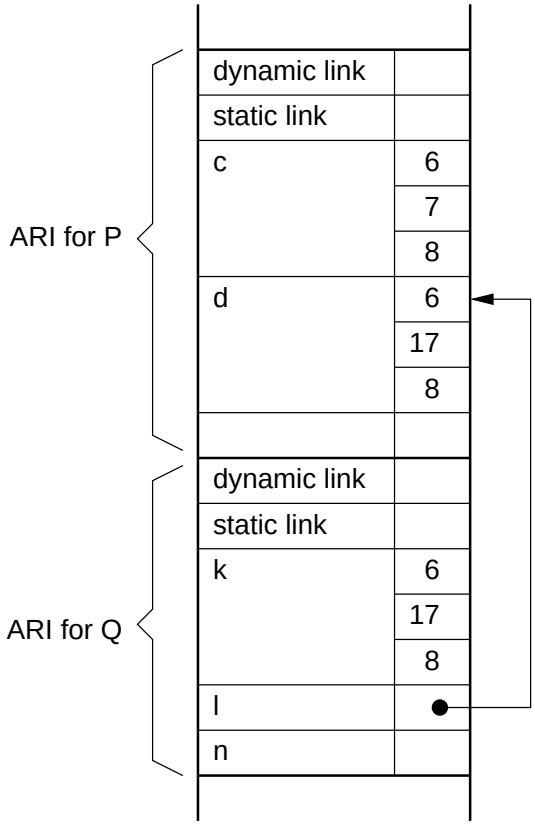
3. In the following Pascal code sample, two arrays are passed to the procedure Q which is called from procedure P. One array is passed by value and the other by reference.What is the output when P is executed? Draw the state of the run-time stack just before Q terminates. You may ignore any reference to the caller of P.

```
type vect = array[1..3] of integer;
procedure Q(k:vect; var l:vect);
    var n: integer;
begin
    k[2]:=k[2] + 10;
    l[2]:=l[2] = 10;
    for n := 1 to 3 do write(k[n]);
    for n := 1 to 3 do write(l[n])
end;

procedure p;
    var
    c, d : vect;
    m : integer;
begin
    c[1]:=6, c[2]:= 7; c[3] := 8;
    d[1]:=6, d[2]:= 7; d[3] := 8;
    Q(c,d);
    for m := 1 to 3 do write(c[m]);
    for m := 1 to 3 do write(d[m])
end
```

Solution to selected exercise

Exercise 3



Note that formal parameter `k` is a copy of `c`, while formal parameter `l` points to `d`.

When P executes, the values printed are: 6 17 8 6 17 8 6 7 8 6 17 8

Module 10

Abstract Data Types

Overview

In our previous discussion on types we introduced the idea that major activity for today's programmers is designing new data types as a better way to represent the data being manipulated within a program. An abstract data type is a further extension of this idea.

The concept of abstract data types (ADTs) forms a basis for the design of another class of languages; the object oriented languages. In this module we study ADTs in a more general sense, prior to our discussion of object oriented languages in Module 11.

10.1 Data Abstraction

Data abstraction is the process of fully characterising some category of objects by describing the essential attributes of that category. e.g. A cup is a container for liquid from which we can drink the liquid. This description fits all cups, whether or not the cup has a handle, holds only hot or cold liquid, includes a lid and a drinking spout (a trainer cup) or any other variation. If you are thirsty, you can use any cup irrespective of any other properties it may have. As long as you can fill (or partly fill) the cup and then drink from it, your requirements are met. It is irrelevant whether the particular cup you use is blue, made of plastic, has a handle or whatever. In other words, it is the operations you can perform with it that defines a cup, not how it is constructed. This is similar to the idea we saw in our earlier discussion an types, that a type defines the operations that can be performed on a variable.

As programmers, one of the common design decisions we must make is a choice of data structure representation. e.g. we can implement a stack as an array or as a linked list. Within a well designed program that uses several stacks the stack data structure will be defined in one unit (a module or package etc.). Placing the definition of a data structure within a single unit accords with the principle of encapsulation. The definition will provide the usual stack operations such as push and pop. When we want to use that unit we can only use these abstract operations, the concrete operations to implement push and pop (array subscripts or a pointer operations) or not available. Prohibiting users access to the data representation accords with the principle of data hiding. A unit that defines a set of abstract operations on a data structure in this way is called an abstract data type.

Reading Sebesta Chapter 11.

Sebesta discusses support for ADTs in Ada, C++ and Java. Of the major imperative languages, only these three provide the necessary modularisation to encapsulate the definitions and hide the representation.

Note that Ada packages are not true ADTs. In Module 9 we saw an example of an Ada package definition for `Date_Type`. Why is this is not a true ADT? An ADT is a type which we can use to declare items; then each item carries with it the operations it needs to manipulate itself. In the `Date_Type` example, the package contains a type definition and also contains the operations to manipulate items of that type. However, although we can declare items to be of `Date_Type`, we still need the `use Date` clause to access these operations. The item itself does not carry with it this information.

The classes of C++ and the other object oriented languages we discuss in Module 11 provide true support for abstract data types.

Exercises

1. Complete questions 1, 2, 3, 4, 8 and 25 from the Review Questions in Chapter 11 of Sebesta. Check your answers by referring to the text.

2. Complete questions 1 and 2 from the Problem Set in Chapter 11 of Sebesta.

Solutions to Selected Exercises

Exercise 2

Q 2 By returning a pointer to the object rather than a copy, the program has direct access to the object e.g. the object's value can be changed by assigning a new value to the memory location being pointed to. A true abstraction will only permit the data object to be changed by invoking an operation defined for the object; no direct access to the data should be possible.

Module 11

Object Oriented Languages

Overview

We saw in Module 10 there are many advantages to be gained when a language fully supports the definition of abstract data types and provides a means of both encapsulating that definition and hiding implementation details from the user of the ADT. Module 10 also foreshadowed the ultimate support for ADTs; the object oriented languages. This is where we now focus our attention.

Reading Sebesta section 12.1 to 12.3, 12.10

Sebesta discusses object oriented programming as supported by a number of languages: Smalltalk, C++, Java, C#, and Ada95.

Although these are probably the two most widely used OO languages they are far from being the only ones. Not all OO languages exhibit exactly the same characteristics as Sebesta discusses. In the remainder of this module we will take a more general look at OO and some of the other possible features that may be found in an OO language.

Reading Sebesta section 12.4 to 12.9. Browse these sections to gain some insight into differences between different object oriented languages. This material will not be examined.

11.1 ADTs & OOP

Initially ADT definitions in the OO languages appear little different from those we have already studied; complex data objects can be built and a limited set of functions to operate on that structure provided. The unit of encapsulation in the common OO languages is the class, but note that not all languages that can be charactered as OO include classes.

In our previous study of ADTs, there was no mechanism for representing ADTs that describe similar sets of objects. e.g. An ADT that described a linked list could not be used as part of the definition of an ADT that described a circularly linked list. We needed to define a new set of operations in the new ADT, even though all the operations on linked list are also applicable to a circularly linked list.

Now, in the OO languages we can actually describe a relationship between these ADTs and use all the operations defined in the linked_list ADT in the ADT for a circularly linked list. This relationship is created by deriving a new ADT from another and is known as inheritance. The inheritance relationships form a tree-like hierarchy.

11.1.1 Classes and Typing

The class definitions in class based OO languages describe a new type. Inheritance relationships introduce the concept of subtyping into the OO languages. Thus the type `circularly_linked_list` is a subtype of `linked_list`.

In many languages this means we can substitute a variable of type `circularly_linked_list` for a variable of type `linked_list`. We can do this because a variable of type `circularly_linked_list` has all the properties of `linked_list` plus the extensions that make it a circularly linked list. This is similar in some ways to being able to assign an integer to a real variable. The difference is that the coercion rules of most languages will convert the integer to a real. In OO languages the information that the object is a circularly linked list is not necessarily lost when the value is assigned to a linked list variable.

11.2 Inheritance

Because inheritance implies some kind of familial relationship we use terms such as child, parent, ancestor etc. when describing inheritance relationships. Smalltalk also uses the terms subclass and superclass.

One way to view inheritance is that it provides a means to implicitly pass information between program components. An ADT that inherits from another receives all the properties and characteristics of its parent. Looked at this way, we can say that the scope rules of the block structured languages are a form of inheritance. Names in an inner block are inherited from the outer block. Indeed both scope diagrams and inheritance hierarchies are depicted using similar tree structures.

We can extend this analogy further. Consider:

```
{
  int i, j;
    {
      float j, k;
      k = i + j;
    }
}
```

As we know, the declaration of `j` inside the inner block obsures the definition of `j` in the outer block. The `i` referred to in the inner block is actually the `i` declared in the outer block.

Now consider a class `B` which is derived from a class `A`. The class `A` defines two data objects; `ob1` and `ob2`. Providing `B` does not redefine either `ob1` and `ob2`, any reference to them from within `B` actually refers to the objects `ob1` and `ob2` in class `A`. If `B` chooses to redefine `ob2`, then the redefinition obscures the reference to `ob2`

```
#include<iostream.h>

class A     { public: virtual void f() { cout << "A\n"; }; };

class B : A { public: void f()         { cout << "B\n"; } };

class C : A { public: void f()         { cout << "C\n"; } };

main () {
    A  a;
    B  b;
    C  c;
    int x;

    cin >> x;
    a = x ? (A *) &b : (A *) &c;
    a -> f();
}
```

Figure 11.1: Dynamic binding in C++

in class `A` in much the same way that the local declaration of `j` in the inner block above, obscured the declaration in the outer block.

In this way classes can choose which part of its parent it has access to.

11.2.1 Single vs multiple inheritance

When a class can inherit from only one other class we call this single inheritance. Smalltalk permits single inheritance only. Many of the newer OO languages permit multiple inheritance i.e. allow a class to have two or more parents.

11.2.2 Dynamic Binding

According to Sebesta, dynamic binding, along with ADTs and inheritance are the three properties that usually characterise object oriented languages. As we have seen, child classes can inherit the parent's implementations or they may choose to provide their own implementation. We also saw that an object whose value is a subtype of the declared type of a variable may be assigned to that variable. Say we now call an operation on that object. Which version of the operation is to be performed; the version defined in the subtype or the version defined for the variable's declared type?

Figure 11.1 shows a minimal working C++ program which illustrated dynamic binding. The function `f` from either class `B` or `C` is executaed at runtime depending on the value of `x` read from input. Note that dynamic binding is only possible because pointer `a`, declared as pointer to superclass `A`, can be assigned the address of pointers to subclasses `B` or `C`.

With dynamic binding, any call causes the class describing the actual dynamic type of the current object to be searched for an operation of that name. Thus the dynamic type of the object, rather than the declared type, determines which version of an operation is executed. If no definition is found for the operation, the

search continues in the parent or parents, and so on. In this way we eventually search the class describing the declared type of the variable and, if still no definition is found, continue to search the ancestor classes.

Most OO languages are strongly typed so we are guaranteed to find a definition for the operation somewhere in the inheritance tree. If no such definition appeared, the statement calling the operation would not have compiled. Smalltalk variables are not typed, so this search will continue all the way up the inheritance tree and may eventually fail, causing a run-time error.

Note that dynamic binding must be made explicit in C++ as static binding is the default.

11.3 Garbage Collection

In Module 7 we studied problems associated with the allocation and deallocation of memory. Program objects created dynamically can become "lost" i.e. unreachable by the program, leaving the memory allocated for the variable unavailable for use.

Clearly this will also be a problem with an OO program which, during execution, may create and discard numerous objects. The task of determining whether an "object" is reachable is compounded by the fact that objects are often aggregates of other objects, and a number of aggregate objects may contain a reference to the same object.

Of the mainstream OO languages, only Java implements garbage collection. This is an enormously helpful feature.

11.4 Summary

Object oriented programming includes object oriented design. The major OO languages implement ADTs in the programming unit called a class. Inheritance describes a relationship between classes whereby a new class may be derived from an existing class. In most languages, classes define a new type.

Polymorphism and dynamic binding may be regarded as complementary concepts. Polymorphism allows variables to hold values of different types that are related by inheritance; dynamic binding provides the means of resolving any name conflicts by searching the inheritance structure of the variable's dynamic type.

Garbage collection and assertions are two other issues often raised in the context of object oriented programming languages.

Exercises

1. Complete questions 1, 2, 3, 6, 7, 8, 12, 15, 21, 23 and 28 from the Review Questions in Chapter 12 of Sebesta. Check your answers by referring to the text.

2. In Ada we can write a generic function that accepts parameters of different types. Compare this approach to polymorphic operations in OO languages.

3. Compare and contrast abstract data types and classes.

4. Complete questions 1, 4 and 6 from the Problem Set in Chapter 12 of Sebesta.

Solutions to Selected Exercises

Exercise 3 Both are similar in that they describe a new type and the operations that can be performed on an object of that type. They are dissimilar in that we can use inheritance between classes to describe relationships between similar sets of objects. You should be able to find some other minor points of difference and similarity, without discussing specifics of syntax.

Module 12

Exception Processing and Event Handling

Overview

In Module 8 exception handling was briefly mentioned as another means of altering the flow of control within a program. An exception is any error that occurs during execution such as an array subscript being out of bounds or an attempt to read past the end of file. The mechanisms provided to take action or that allow the programmer to define what should be taken when such errors are detected, are known as exception handling.

Event handling has some similarities with exception handling, but it deals specifically with external physical events like those generated by mouse clicks or movements. Mostly, event handling is used to implement graphical user interfaces.

12.1 Error Conditions

In most of our discussions so far we have assumed a smooth flow of control, however, as we all know, our programs can and do encounter problems during execution. We can consider the main causes of these problems as falling into three main categories:

- data errors
- resource exhaustion
- loss of facilities

A data error occurs when we attempt to perform an operation on a data value that cannot handle that operation e.g. divide by zero and integer overflow. As programmers we can test for zero before performing division but integer overflow is machine dependant and thus more difficult.

Resources a programmer must request explicitly, such as memory and disk space, may be exhausted. In some languages a failed request for either will return an error code and the programmer can thus check the operation's success. In Ada, however a failed call to *new* does not return an error code. Other resources such as CPU time and stack space may also become exhausted. In neither case are these

requested explicitly so error codes are not applicable, so the programmer has no way of guarding against the failed request.

Circumstances that lead to loss of facilities include a broken network connection, a disk head crash loss of power or a user interrupt. None of these are related in any way to an action taken by the program.

Even if the programmer checks for all situations under his/her control, most of these problems will cause the program to terminate. An informative error message on termination is all that is needed in some cases. However, in other situations this is not sufficient. e.g. A program that has brought a database into an inconsistent state should restore the database to a proper state before terminating. The programmer needs to regain control after an error so that either an alternative action might be taken or any necessary clean up operations can be done before exiting.

12.1.1 Signals

One way in which the programmer can regain control after an error is using signals. C on UNIX systems provide signals.

Errors are grouped into named categories called error conditions. e.g. An error in a floating point operation on a C-UNIX system is a `SIGFPE` error. Programs may contain signal statements, which name the error condition and a function to be called if that error occurs. e.g.

```
signal (SIGFPE, close_down);
```

If the error `SIGFPE` occurs, the function `close_down` will be called and clean up the system before the program terminates.

Possible problems with this approach include:

- The procedure `close_down` has no access to the local variables in the subprogram that was active when the error occurred. As these are the most recent and therefore probably contain precisely the data needed for a graceful close down, the programmer is forced to use globals for any information that might be needed.

- What happens if the `close_down` procedure does not terminate the program but returns instead? Depending on the language implementation we may enter an infinite loop of signal-returns or the program may continue with data of doubtful integrity.

12.2 Exception Processing

As we have already seen, many exceptions can be detected in hardware. When a hardware error occurs the operating system raises a status flag signalling the error that occurred. These status flags can be read by application programs. e.g. COBOL provides an `ON SIZE ERROR` clause that can be included in arithmetic expressions to pass control to an error routine if overflow occurs. However, not all languages provide the primitives to read these status flags e.g. Standard Pascal.

The error codes returned by failed requests for resources are another primitive means of detecting exceptions. C detects I/O errors in this way.

Although both mechanisms allow us to detect some errors, and perhaps exit gracefully, the need to include error checking code can produce complex subprograms that are difficult to read. As an example, consider the two versions of a C routine that appear below. Both perform the same task; the first includes error checking; the second none.

```
    /*
     *  input WITH error checking
     */
    void test_input () {
    int age, code;
    char * name;
    FILE * infile, outfile;

      if (!(infile = fopen (``myinput'', ``r'') ) {
        printf(stderr, ``Cannot open input file.\n'');
        exit(1);
      }
      if (!(infile = fopen (``myoutput'', ``r'') ) {
        printf(stderr, ``Cannot open output file.\n'');
        exit(1);
      }
      if (fscanf(infile, ``%s%d'', name \&age) <2)
        fprintf (``Bad data, record ignored\n'');
      if (fprintf(outfile, (``Hi %s; I hear you are %d years old!\n'',
                           name, age) == EOF {
        fprintf (stderr, ``Cannot write to output file !\n'');
        exit (1);
      }
    }
    /*
     *  input WITHOUT error checking
     */
    void test_input ( ) {
    int age, code;
    char * name;
    FILE * infile, outfile;
    infile = fopen (``myinput'', ``r'');
    infile = fopen (``myoutput'', ``r'');
    fscanf(infile, ``%s%d'', name &age);
    fprintf(outfile,(``Hi %s; I hear you are %d years old!\n'',
                     name, age);
}
```

The code with error checking is cluttered, harder to write and more difficult to read. Moreover, errors often need to be passed back up a chain of calls to the program level to ensure a graceful termination. Each level requires error checking code to handle the error. A more elegant solution is needed.

The signal mechanism is an improvement, however, we have already seen it is not

without problems of its own.

12.3 Exception Handlers

A more integrated approach to handling error conditions is provided by an exception handler. Exception handlers associate exception conditions with the actions to be performed when that condition is met. Note that not all exceptions need be errors; some unusual events that require special processing e.g. a user interrupt, can be treated as exceptions.

Integrated exception handlers consist of a control structure and, often, a list of predefined exception names. The programmer can define exception handling routines for any of these predefined exception names. The exception handler then associates these actions with the appropriate exception name. The programmer can test for the exception names in the program, and, if the exception occurs, the handler passes control to the user-defined exception-handling code.

Some languages permit the user to also define new exceptions which are then added to the list of predefined exception names.

Reading Sebesta sections 14.1 to 14.4.

Sebesta first discusses exception handling in general and then looks at the exception handling mechanisms of Java, Ada and C++. While it is not essential that you fully understand the syntax of the mechanism for each language, you should appreciate the different mechanics of these approaches.

12.4 Event Handling

An event occurs as a result of some external action. Typically this occurs when a user clicks a mouse button or keyboard key. It is asynchronous—the program does not know in advance when any kind of event will occur or, in general, the order of events.

Reading Sebesta sections 14.5 and 14.6.

Sebesta discusses event handling, and the event model used by Java. Note that the event handling implementation does not commonly require any extra language features. All that is required is a library or set of classes to provide the facility of connecting a user-written event handler with one of a number of system defined events.

Many other languages such as Visual Basic, and packages like GTK+, use similar schemes to that employed in the Java example.

12.5 Summary

Exception handling is an integrated approach to handling program errors and other unusual events. Few languages fully support exception handling, and many provide only limited facilities for detecting and handling errors.

Event handling solves a simpler problem than exception handling. It is most commonly used to implement graphical user interfaces, and is usually implemented as a separate package written using standard language features.

Exercises

1. Complete questions 1, 2, 3, 10, 14, 18 and 22 from the Review Questions in Chapter 14 of Sebesta. Check your answers by referring to the text.

2. Complete questions 1, 5, 6 and 7 from the Problem Set in Chapter 14 of Sebesta.

Solutions to Selected Exercise

Exercise 2

Q 1 Standard Pascal provides no way of testing hardware status codes; therefore the only runtime errors that can be handled relate to I/O e.g. code can be written specifically to cover input of a character when an integer is expected etc. Subrange errors, subscript errors, division by zero and arithmetic overflow cause the program to terminate. (Extensions to Pascal usually include some means of testing status flags.)

Q 6 The Ada mechanism is flexible and powerful; usually providing more information regarding the nature of the exception, thus allowing the handling mechanism to fit the exception. However, the major advantage of the Ada mechanism is that an exception will propagate outwards till a suitable handler is encountered; rather than having to include exception handling code everywhere exceptions are possible.