University of Portsmouth
BSc (Hons) Computer Science
Second Year

Programming Applications and Programming Languages

(PAAPL)
M30205
September 2023 - June 2024
20 Credits

Thomas Boxall
up2108121@myport.ac.uk

Contents

I Programming Applications **2**

II Programming Languages **4**

1 Lecture - Introduction To Progrmaming Languages (2024-01-26) **5**

2 Lecture - Overview and Evaluation of Programming Languages (2024-01-26) **8**

3 Lecture - High Level Language Implementation (2024-02-02) **13**

4 Lecture - Lexical Analysis - Regular Expressions (2024-02-05) **16**

5 Lecture - Lexical Analysis - Deterministic Finite Automaton (2024-02-09) **21**

6 Lecture - Describing Language Syntax (2024-02-16) **26**

7 Lecture - Syntax Analysis and Parsing (2024-02-19) **31**

8 Lecture - LL1 Parsers (2024-02-23) **37**

9 Lecture - Syntax Analysis - Bottom Up Parsing (2024-03-01) **40**

10 Lecture - Names, Bindings, Scopes and Memory Allocation (2024-03-11) **42**

11 Lecture - Elementary Data Types (2024-03-15) **48**

12 Lecture - Compound & Composite Data Types (2024-03-22) **53**

13 Lecture - Expression and Assignment (2024-04-15) **59**

14 Lecture - Statement-Level Control Structures (2024-04-19) **66**

15 Lecture - Subprograms and Parameter Passing (2024-04-26) **70**

16 Lecture - Abstract Data Types & Concurrency (2024-04-29) **75**

Teaching Block I

Programming Applications

There are no notes for this Teaching Block. It was entirely practical with the coursework involving no written component.

Teaching Block II

Programming Languages

Page 1

Lecture - Introduction To Progrmaming Languages

2024-01-26 | 1400 | Jaicheng

This lecture will introduce us to the many different ways in which a programming language can be categorised.

1.1 Programming Domains

A *Programming Domain* is one way to think about & categorise a programming language. We have different different programming domains for different applications as each application requires a specialised instruction set to improve efficiency for the programmer. Everything humans do can be solved by a computer, the number of programming domains reflects this.

1.1.1 Scientific Applications

The *Scientific Applications* domain is concerned with lots of mathematical operations. These operations would be on a set of data, which would result in an output set of data. Applications could be used, for example, to model future weather conditions where the application is given the current weather conditions. Scientific Applications will use lots of floating-point computations and arrays, as well as matrices. An example of a language in this domain is Fortran (*For*mula *Tran*slating system, created by IBM).

1.1.2 Business Applications

Business Applications are designed to be used by businesses to complete business functions. For example, batch printing payslips. They use decimal numbers and characters. An example of a language in this domain is COBOL (COmmon Business-Oriented Language).

1.1.3 Artificial Intelligence

In the *Artificial Intelligence* domain, symbols are manipulated, rather than numbers and linked lists are used. Nowadays, this domain is now more talking about reasoning, facts and truth verification. An example of a language in this domain is LISP (LISt Processing).

1.1.4 Systems Programming

Systems Programming is concerned with the control of the hardware of the computer, the management of the storage, the display control and management of other components such as peripherals. The languages used, such as C, need to be specifically designed for this domain due to the required low level interactions between the program and the hardware.

1.1.5 Web Software

Web Software is arguably one of the most popular domains in these modern times. Much of the modern software is developed as a website, for easy use across multiple devices. The languages used are eclectic and each serve a particular purpose; for example, HTML for markup, PHP for scripting and JavaScript for adding interactivity.

1.2 Language Categories

Programming languages can be broken down into a series of categories based on their properties and uses.

1.2.1 Machine Languages

The *Machine Language* family of languages are hardware implemented languages; which means the instruction set available within them is the instruction set available on the CPU. This means the instruction set is limited in size and will be represented as binary (or hexadecimal) numbers.

1.2.2 Assembly Languages

The *Assembly Languages* family of languages are a simplification of Machine Languages. In essence, they are the machine language with a 'human-friendly' outside layer, meaning that they are legible to most people. To be executed, they require translating to machine code (which involves the use of a translator or interpreter). They come with labelled storage locations, jump targets and subroutine starting addresses in addition to the basic Machine Language instructions.

1.2.3 High Level Language

The *High Level Language* family is another step up from Assembly Languages. Their syntax is very close to natural language syntax, making it much more legible and easier for programmers to read, write, understand and memorise. They usually will come with variables, types, subroutines, functions, the ability to handle complex expressions, control structures, and composite types. Examples include: C and Java.

1.2.4 Systems Programming Language

The *Systems Programming Language* are effectively high level languages who also deal with the low level operations. For example C, C++, and Ada. They manage the memory & process management, I/O operations, device drivers, operating systems.

1.2.5 Scripting Languages

The *Scripting Languages* are a set of languages which exist to automate tasks, saving humans time. They will commonly be used to: analyse or transform a large amount of regular textual information; act as a glue between different applications; or bolt a front end onto an existing application. The languages used are often interpreted and will often include lots of string processing functions, such as in Python or PHP.

1.2.6 Domain Specific Languages

The *Domain Specific Languages* are highly specialised languages which are used in a specific area only. For example the Adobe PostScript language is used for creating vector graphics for electronic publishing.

1.3 Categories by Paradigm

Programming languages fit into one of three different paradigms.

1.3.1 Procedural

A program is built from one or more procedures (can also be called subroutines or functions) and the program will revolve around variables, assignment statements and iteration. Some languages will also support Object Oriented programming as well as some supporting scripting. Examples of languages include: C, Java, Perl, JavaScript, Python, Visual Basic, C++.

1.3.2 Functional

Functional languages work by applying a function to a given parameter. Languages include: Haskell, LISP, Scheme, F#, Java 8.

1.3.3 Logic

Logical rules are used to do reasoning over given facts which draws conclusions. The logical rules do not have to be defined in any particular order. Languages include: Prolog.

1.4 Categories by How Tasks are Specified

1.4.1 Imperative Languages

In imperative languages, you have to explicitly instruct the computer what it needs to do to reach the goal, computing tasks are defined as a sequence of commands which the computer performs. The program will state in step-by-step instructions what the computer needs to do. This means that the implementation of the algorithms, and therefore the efficiency of the algorithms is down to the developer. Procedural languages belong to this category.

1.4.2 Declarative Languages

In declarative languages, the computer gets told the desired results, without explicitly listing the the commands or steps which the program must undertake to reach its goal. Functional and logical programming languages belong to this category.

Page 2

Lecture - Overview and Evaluation of Programming Languages

2024-01-26 14:00 Jaicheng

2.1 The 'TPK' Algorithm

The TPK algorithm was designed by *Trabb, Pardo* and *Knuth* in the 1970s for illustration purposes. It is designed to:

1. read 11 numbers (entered by the user using their keyboard) into an array,
2. process the array in reverse order, applying a mathematical function to each value
3. then for each value - reporting the value or a message saying that the value is too large

The algorithm includes all the basic constructs which would be expected to exist in a modern language therefore making it useful to use when understanding how languages work. A pseudocode implementation of the TPK algorithm is below:

```
input 11 numbers into a sequence A
reverse sequence A
for each item in sequence A
    call a function to do an operation
    if result overflows
        alert user
    else
        print result
```

2.2 Fortran

Fortran (*For*mula *Tran*slation) is the first well-known high-level programming language. It was developed by a team at IBM led by John Backus with the goals: to lower the costs involved with programming and debugging; and to compete with "hand coded" assembly language programs in terms of execution speed. The first Fortran compiler, built for the IBM 704 mainframe, was completed in 1957.

Early source code had a strict, specific, format which was in part due to it being a punched-card program where the column and row position of the punch is important.

The TPK algorithm in Fortran is shown below:

```
C THE TPK ALGORITHM IN FORTRAN
  FUNF(T)=SQRTF(ABSF(T))+5.0*T**3
```

```
  DIMENSION A(11)
1 FORMAT(11F12.4)
  READ 1, A
  DO 10 J=1,11
  I=11-J
  Y=FUNF(A(I+1))
  IF(400.0-Y) 4,8,8
4 PRINT 5,I
5 FORMAT(I10,10H TOO LARGE)
  GOTO 10
8 PRINT 9,I,Y
9 FORMAT(I10,F12.7)
10 CONTINUE
  STOP
```

A letter `C` in the first column indicated that the card was a comment and as such it should be ignored by the compiler. Non-Compiler cards were divided into four fields:

1-5 is the label field; a sequence of digits here indiciates the purpoose of the card and therefore the instruction.

6 is a continuation field whereby a non-blank character here caused the card to be taken as a continuation of the statement on the previous card.

7-72 is the statement field

73-80 are ignored by the compiler. This means that they can be used for card identification purposes in the event that the cards were dropped.

The restrictions on the structure of the code were removed in Fortran 90, where it became a Free-Form language.

2.3 COBOL

COBOL (*CO*mmon *B*usiness *O*riented *L*anguage) was created at the end of the 1950s by the US Department of Defence. It was initially developed as a language for business data processing from which comes its verbose syntax that was designed with the ability for managers to be able to read it in mind. COBOL was never designed to be used as a scientific language and has many critics, where programmers felt that the verbosity of the language increased program length, not readability.

2.4 Algol

Algol (*ALGO*righmic *L*anguage) was originally designed to overcome the problems with Fortran in the late 1950s. Arguably, it is one of the most successful high level programming languages of the time because it was influential over the design of subsequent high level languages.

Algol 60 introduced the use of formal notation for syntax, block structure (with locally defined variables, whoop whoop), supported recursive procedures (until this point, you could do it however the languages didn't like it) and readable if and for statements. Ultimately, Algol died out with the rise in Fortran's popularity. The TPK algorithm in Algol is shown below.

```
begin
  comment TPK algorithm in Algol 60;
  integer i; real y; real array a[0:10];
  real procedure f(t); real t; value t;
```

```
    f := sqrt(abs(t))+5*t^3;
  for i := 0 step 1 until 10 do
    read(a[i]);
  for i := 10 step -1 until 0 do begin
    y := f(a[i]);
    if y > 400 then
      write(i, "TOO LARGE")
    else
      write(i, y);
  end
end
```

2.5 Pascal

Pascal is a direct descendant of Algol, which was intended to be more efficient in order to compete with Fortran as a general purpose language. An early Pascal compiler was designed to be portable, compiling the source code to a virtual machine (*P-Code*). Pascal was popularised in the late 1970s as a good teaching language as it enforced a "good" programming style, it was especially popular amongst Universities. Pascal is still in development, with more recent versions adding modules and classes (for example, the Object Pascal Language, which is sometimes known as Delphi). The TPK algorithm in Pascal is shown below:

```
program example(input, output); (* TPK alg in Pascal *)
var i : integer; y : real; a : array [0..10] of real;
function f(t : real) : real;
begin
  f := sqrt(abs(t)) + 5*t*t*t
end;
begin
  for i := 0 to 10 do read(a[i]);
    for i := 10 downto 0 do
      begin
        y := f(a[i]);
        if y > 400 then
          writeln(i,' TOO LARGE')
        else
          writeln(i, y)
      end
end.
```

2.6 Systems Programming: C

In the early 1970s, it was decided that a more efficient language would be required for systems programming (such as compilers, operating systems, etc) as the languages created during the 1960s (Fortran and COBOL) weren't sufficient. C was developed alongside the UNIX operating system at Bell Labs; and has proven to be a very successful in systems programming and as a general-purpose programming language. It combines high-level features, such as functions & loops with low-level operations, such as arithmetic on memory addresses. Critics argue that C code is less readable and it's weak typing causes problems. Shown below is the TPK algorithm written in C.

```
#include <stdio.h>/* TPK algorithm in ANSI C */
#include <math.h>
double f(double t) {
  return sqrt(fabs(t)) + 5*pow(t,3);
```

```
}
main() {
  int i; double y; double a[11];
  for (i = 0; i <= 10; i++)
    scanf("%lf", &a[i]); /* %lf means long double*/
  for (i = 10; i >= 0; i--) {
    y = f(a[i]);
    if (y > 400)
      printf("%d TOO LARGE\n", i); /* %d means double*/
    else
      printf("%d %lf\n", i, y);
  }
}
```

2.7 Object Oriented Languages

A major application of computers is the simulation of real-world systems (for example, hospital waiting lists, industrial production lines, etc). Early programming languages were developed specifically for simulation which include GPSS and Simula 1. The designers of Simula (Dahl and Nygaard) introduced the concept of a *class* to their programs to represent simulated entities. This was the first 'object-oriented language' as we know them, and therefore Simula 67 can be considered the parent OOL. Smalltalk in 1980 and Eiffel in 1986 were influential in the further development of OOLs; which was furthered in 1985 by Bell Labs who developed C++, the OOL cousin to C. Java and C# are considered descendants of C++.

2.8 Scripting Language

When Scripting Languages were first introduced, they were brought in to automated the task which a human operator could do, for example shell scripts. However nowerdays a scripting language loosely refers to high-level general-purpose programming languages such as: Pearl, Python, PHP, Ruby, JavaScript, and Matlab. Scripting languages are generally typeless with relatively simple syntax and semantics; they are usually interpreted and are, by design, fast to learn and write in. Shown below is the TPK algorithm in Python:

```
from math import sqrt
def f(t):
  return sqrt(abs(t))+5*t**3
a = [input() for i in range(11)]
for i in range(10,-1,-1): # range() is equiv to [10,-1)
  y = f(a[i])
  if y > 400:
    print i, "TOO LARGE"
  else:
    print i, y
```

2.9 Logical Programming (Prolog)

Logical Programming is a type of programming which is built upon logical statements which are comprised of variables, constants and structures. In Prolog, variables begin with capital letters, constants are either atoms or integers and structures consist of a functor and arguments.

A Prolog program consists of facts and rules. Prolog programs are used to answer queries, although simple arithmetic operations are possible. A query is a fact or rule that initiates a search for success in a Prolog program. It specifies a search goal by naming variables that are of interests.

2.10 Language Evaluation Criteria

There are a number of criteria on which the readability, writability and reliability of a programming language can be evaluated.

2.10.1 Simplicity

The simplicity of a language is defined based on the simplicity of the syntax and the number of constructs (small number of constructs is simple). Programmers will tend to only learn the part of a large language which suits them and therefore probably not use the best construct / syntax for what they are trying to achieve. It is, obviously, possible to make a language too simple for example Assembly Language.

2.10.2 Lexical Elements

The form that the individual lexical elements (i.e. words or symbols) or a language take can affect the languages' readability. The meaning of a symbol / keyword should ideally be obvious from it's name.

2.10.3 Orthogonality

Orthogonality in a programming language means that it has a relatively small number of control and data constructs which can be combined in a relatively small number of ways and every possible combination is legal and meaningful. When using an orthogonal language, a programmer is not required to remember a lot of "special cases" in the use of it's constructs. The orthogonality of a language influences both the readability and the writability of software; if a language's rules contain fewer special cases, it is easier to learn.

2.10.4 Data Types

It is also important for a language to have a rich set of data types; as well as having adequate mechanisms for combining types.

2.10.5 Expressiveness

The expressiveness of a programming language relates to how much code is required to implement computations.

Page 3

Lecture - High Level Language Implementation

2024-02-02 14:00 Jiacheng

3.1 Computers & Languages

A computer processor's circuitry provides a realisation of a set of primitive operations (or machine instructions) for arithmetic and logic operations. Some machine level instructions are called macroinstructions because they are implemented with a set of instructions at an even lower level called microinstructions. The machine language of a computer is the language which the computer understands directly, these are very simple as that is the most cost effective solution. It would *theoretically* be possible to design a processor to directly use a language that we would classify as 'high level' however this would add an extreme amount of complexity which isn't worth is.

Sitting on top of the machine language is the operating system, this is a collection of programs that supply higher-level primitives (including device and file system management, I/O operations, text/program editors, etc). Implementations of programming languages exist on top of the operating systems.

3.2 Language Implementation Methods

3.2.1 Compilation

A *Compiler* exists to translate high-level program (written in a source language) into machine code (machine language which the processor can understand). Compiling a program is slow as there is a number of checks and stages which have to be undertaken however as the output is instructions which can be directly executed on the processor, execution of the program is fast.

A Compiler is a program that translates a program in a source language into an equivalent program in a target language. The source language is a high-level language and the target language is a low-level language. Compilers are structured as an ordered series of steps which all make use of the 'symbol table' in different ways. The output from one step feeds into the input of the next step. The phases are outlined below.

3.2.1.1 Lexical Analysis

The *lexical analyser* reads the source program's text one character at a time and returns a sequence of tokens to send to the next phase. Tokens are symbolic names for the lexical elements of the source language and each token is associated with a pattern. The scanner matches patterns against sequences of input characters and when a match is found for one of the patterns, the corresponding token (and any additional required information) is output and passes to the next phase

3.2.1.2 Symbol Table

The symbol table is a data structure containing all the identifiers (together with their attributes) of a source program. For variables, the attributes could be size, type and scope; and for methods, procedures or functions - the attributes could be the number of arguments and their types and passing mechanisms and return type.

3.2.1.3 Syntax Analysis (Parsing)

The syntax analyser analyses the syntactic structure of the source program. The input to a parser is the sequence of output tokens from the lexical analyser. The Syntax Analyser applies the rules that define the language on the syntax tokens. During this process, the parser uses rules to derive the sequence of tokens. Parsers usually construct abstract syntax trees that still represent the source program's syntax, however are simpler than the corresponding parse trees.

3.2.1.4 Semantic Analysis

The syntax analyser checks whether the program is syntactically correct, but not that it is completely valid or semantically correct. The semantic analyser determines if the source is semantically valid. It uses the AST and Symbol table.

3.2.1.5 Code Optimisation

The code optimisation phase is used to shorten the time taken to run the program and improve it's space efficiency. It does this through trying to remove as much redundant code or through pre-executing code which will always return the same value in runtime, ie adding 3 and 7.

3.2.1.6 Code Generation

The last stage of compilation is to generate code for the specific machine. This phase involves: selecting which machine language instruction to use, scheduling these instructions in the most efficient order; allocating variables to processor registers and generating debug data if required. The output from this phase, and ultimately of compilation, is usually programs in machine language, or assembly language, or code for a virtual machine. The code generation can be found in compiler design texts.

3.2.2 Pure Interpretation

Pure Interpretation is a type of translation where the program we are planning on running is interpreted by another program called an interpreter. The interpreter directly executes the programs written in a high-level language, line-by-line as the program is executed. There is no pre-compilation or batch-compiling in pure interpretation.

Through Pure Interpretation - errors are more likely to occur during runtime, this is because there is no error checking as there is no pre-parsing of the code before execution. An interpreter parses the source code and executes it directly, examples include BASIC and early versions of LISP.

Pure Interpretation is much slower than compiled programs. This is because decoding high-level language statements is slower than decoded machine language instructions; which is further amplified where the high-level statement must be decoded every time the program is run.

Interpreted programs will often require more storage space to execute than a compiled program would - this is because the source code and symbol table will often both need to be present during interpretation.

Nowadays it is rare for traditional high level languages to utilise interpretation however it is making a comeback with some web languages (such as JavaScript and PHP).

3.2.3 Hybrid Implementation Systems

A Hybrid Implementation System is a mid-point between full compilation and pure implementation. A Hybrid system will compile the source code into an intermediary language which then gets interpreted at runtime through a virtual machine. The VM takes the intermediate language as it's machine language and can therefore interpret that at much greater speed.

3.2.4 Just In Time Implementation

In Just-In-Time (JIT), the source code is initially compiled to an intermediate language. The intermediate language is loaded into memory, and segments of the program are translated into machine code just before they are executed. The machine code version is kept for subsequent calls.

Page 4

Lecture - Lexical Analysis - Regular Expressions

2024-02-05 | 1400 | Jiacheng

4.1 Programming Language Definition

The full definition of a programming language is important for two different groups of people: language implementers (those who write compilers); and language users (programmers). The full definition of a programming language will include a number of definitions:

Lexical Structures which concern the forms of its symbols, keywords and identifiers

Syntax which define the structure of the components of the language, for example the structures of the programs, statements, expressions, terms

Semantics which define the meanings and usage of structures and and requirements that cannot be described by grammar (ie checking type consistency, arithmetic operations)

4.1.1 Language Analysis

A key process of a language implementation system is to analyze the source code, this is both the lexical and syntax structure. The code analysis system of a language generally consists of two parts:

Lexical analyser is a low-level component which is mathematically equivalent to a finite automaton which is based on regular grammar

Syntax Analyser is a high-level component, also known as a parser, which is mathematically equivalent to a push-down automaton that is based on a context-free grammar.

4.2 Lexical Analysis

A lexical analyser works by reading the source program a single character at a time. It outputs tokens to the next phase of the compiler (the parser). The lexical analyser works to identify substrings of the source program that belong together (lexemes). Lexemes match character patters, which are associated with a lexical category called a token.

4.2.1 Definitions: Alphabet

An alphabet (Σ) is a finite non-empty set of symbols, for example:

- the set $\Sigma_{ab} = \{a, b\}$ is an alphabet comprising symbols a and b.
- the set $\Sigma_{az} = \{a, \ldots, z\}$ is the alphabet of lowercase English letters
- the set Σ_{asc} of all ASCII characters is an alphabet

4.2.2 Definitions: Strings

A string or word over an alphabet (Σ) is a finite concatenation (or juxtaposition) of symbols from Σ. For example:

- $abba$, aaa and $baaaa$ are strings over Σ_{ab}
- $hello$, $abacab$ and $baaaa$ are strings over Σ_{az}
- $h\$(e'lo$, $PjM\#;$ and $baaaa$ are strings over Σ_{asc}

The length of a string, w is the number of symbols it has and is denoted as $|w|$. For example $|abba| = 4$.

The empty or null string is denoted ε and therefore $|\varepsilon| = 0$.

The set of all strings over Σ is denoted Σ^* for example

$$\Sigma_{ab}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aab, \ldots\}$$

For any symbol or string x, the notation x^n denotes the string of the concatenation of n copies of x:

$$a^4 = aaaa \qquad (ab)^4 = abababab$$

4.3 Regular Expressions

A regular expression specify a pattern of string of symbols. A regular expression, r, matches (or is matched by) a set of strings if the patterns of the strings are those specified by the regular expression. Regular expressions can be used in a variety of applications where more complex string matching is required.

The set of strings matched by a RegEx, r, is denoted by $L(r) \subseteq \Sigma^*$ (which translates to: those strings belong to all the strings over the alphabet, Σ) and is called the language determined or generated by r.

4.3.1 Definitions

4.3.1.1 Definition 1

The set for an empty set or empty language, $\emptyset$ is a regular expression. It matches no strings at all and will not be useful to us.

The empty string symbol ε is a regular expression which matches just the empty string ε.

The empty string ε should not be confused with the empty laguage $\emptyset$. $\emptyset$ is a formal language (e.g. a set of strings) that contains no strings, not even the empty string. The empty string is a string that has the properties:

- $|\varepsilon| =$ (it's length is 0)
- $\varepsilon + s = s + \varepsilon =$ (the empty string is the identity element of the concatenation operation)

4.3.1.2 Definition 2

Each symbol $c \in \Sigma$ in the alphabet Σ is a Regular Expression. This RegEx matches the string consisting of just the symbol c. For example, for the alphabet $\Sigma = \{a, b\}$ we have:

- RegEx a matches the string a
- RegEx b matches the string b
- Both symbols a and b are RegExs

4.3.1.3 Definition 3

If r and s are regular expressions, then $r|s$ (sometimes written as $r + s$, which is read as "r or s") is a RegEx. For example:

- RegEx $a|b$ matches the string a or b
- $a|\varepsilon$ matches the strings a or ε

4.3.1.4 Definition 4

If r and s are regular expressions, then concatenation rs (read "r followed by s") is a RegEx. This matches any concatenation of two strings where the first string matchces r and the second matches s. For example:

- RegEx ab matches the string ab
- RegEx $abba$ matches the string $abba$

As with arithmetic expressions, parentheses can be used in RegExs to make the meaning of a RegEx clearer. For example $(a|b)a$ matches the strings aa and ba.

4.3.1.5 Definition 5

The RegEx r^* (read "zero or more instances of r") is a Regular Expression. This matches all finite (possibly empty) concatenations of string matched by r.

4.3.1.6 Definition 6

The RegEx rr^* read as "one or more instances of strings matched by r" can also be written as r^+.

4.3.2 Examples

- RegEx a^* matches the strings ε, a, aa, aaa, ...
- RegEx $(ab)^*$ matches the strings ε, ab, $abab$, ...
- RegEx $(a|bb)^*$ matches the strings ε, a, bb, abb, baa, $abba$, ...
- RegEx $(a|b)^*aab$ matches any string ending with aab
- RegEx $(a|b)^*baa(a|b)$ matches any string containing the substring baa

4.3.3 Precedence

As with all 'operators', the different symbols in Regular Expressions have different precedences, they are shown below in the order highest to lowest.

1. ()
2. * or $^+$
3. concatenation
4. |

4.3.4 Using Regular Expressions for Lexical Analysis

Regular Expressions provide us with a way to describe the patterns of a programming language. This is useful as it can be used as a 'shorthand' rather than saying "a number is any combination of 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9".

We assume that the alphabet used here is Σ_{asc} since the source program takes the form of an ASCII (text) file. Some example patterns for a typical programming language are shown below:

- if for token `IF`
- ; for token `SEMICOLON`
- $(0|1|2|3|4|5|6|7|8|9)^+$ for a token `NUMBER`
- $(a|\ldots|z|A|\ldots|Z)(\_|a|\ldots|z|A|\ldots|Z|0|\ldots|9)^*$ for a token `IDENT`

4.3.5 Regular Definitions

We can give RegExs names which make them easier to read and write; and the names can be used to define other regular expressions. For Example:

- $letter = A|B|\ldots|Z|a|b|\ldots|z$
- $digit = 0|1|\ldots|9$
- $ident = letter(\_|letter|digit)^*$

4.4 Definition of Lexical Languages

Languages, L, are sets of strings (written in the form of a set $\{\ldots\}$), chosen from the strings over some alphabet Σ. This can formally be defined as:

A language L over an alphabet Σ is a subset of Σ^* (i.e., $L \subseteq \Sigma^*$).

For example:

- $\{\varepsilon, aab, bb\}$ is a language over Σ_{ab}
- The set of all Java programs is a language over Σ_{asc}; and so is the set of all C++ programs
- $\emptyset$ is the empty language (over any alphabet) with no strings
- $\{\varepsilon\}$ is a language (over any alphabet) containing just the empty string
- $\{a^n b | b \geq 0\}$ is a language over Σ_{ab} comprising all strings of 0 or more a followed by a single b
- Σ^* is a language over Σ for any alphabet Σ

We denote the language of RegEx's in the form $L(RE)$. For example:

- $L(a^*) = \{\varepsilon, a, aa, aaa, \ldots\}$
- $L(ba^*) = \{b, ba, baa, baaa, \ldots\}$
- $L(a|b) = L(a) \cup L(b)$

4.4.1 Decidability of Languages

Given a language L over some alphabet Σ, it is important to be able to write an algorithm that takes any string ($w \in \Sigma^*$) as an input and:

- outputs 'Yes' if $w \in L$ and
- outputs 'No' if $w \notin L$

An algorithm that does this is called a decision procedure for L.

4.4.2 Regular Expressions and Decision Procedures

There are two different algorithms which can be used to write a decision procedure for language:

- Deterministic Finite Automaton (DFA)
- Nondeterministic Finite Automaton (NFA)

Languages that can be denoted by a RegEx, and can have a DFA / NFA as a decision procedure, are known as *regular languages*. This means that if we can describe a program's language using RegExs and the RegExs have a DFA / NFA then, we can write the lexical analyser using a DFA (or NFA).

Page 5

Lecture - Lexical Analysis - Deterministic Finite Automaton

2024-02-09 14:00 Jiacheng

This lecture is a direct follow-on from the previous lecture looking at Regular Expressions (RegExs).

Rather than using Regular Expressions - we can use *state transition diagrams* to describe the process of recognising the patterns. State transition diagrams (or simply, state diagrams) are directed graphs which are represented of mathematical 'machines' called *finite state automata* (FSA) or *finite automata* (FA).

Finite automata can be *deterministic* (DFA) or *nondeterministic* (NFA). A FA is deterministic if it performs the same operation (a state transition) in a given situation (it's current state and input). A FA is nondeterministic if it can perform any of a set of state transitions in a given situation. We will only consider DFAs for now.

5.1 Finite State Automaton

A Finite State Automaton (FSA), or a finite automaton, has:

- a set of states
- a unique start state
- a set of one or more final / accepting states
- an input alphabet, augmented by a unique symbol representing end of input
- a state transition function - represented by directed edges from one node to another, labelled by one or more alphabet symbols.

Or more formally..., a finite automaton (FA) M consists of:

1. a finite set Q of states
2. a finite alphabet Σ of input symbols
3. a distinguished start state $q_1 \in Q$
4. a set of final states $F \subseteq Q$
5. a transition function $\delta : Q \times \Sigma \to Q$ that chooses a new state for M based on the current state, $s \in Q$, and the current input symbol $a \in \Sigma$.

5.1.1 DFA for Lexical Analysis

As far as lexical analysis is concerned, a DFA is a string processing machine. It reads an input string from left-to-right, one symbol at a time. Then at any step, it is in one of a finite number of states. When it reads each symbol, it moves into a new state determined by its current state and the symbol read. Ultimately, the string is either accepted or rejected depending on the state of the machine after reading the final symbol.

5.2 Transition Diagrams

DFAs are usually represented as diagrams called *transition diagrams*. For example, the diagram below shows a simple DFA that processes strings from the alphabet $\Sigma = \{a, b\}$.

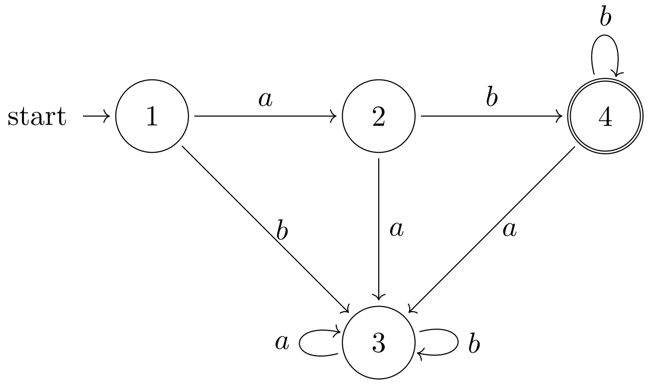

Figure 5.1: DFA Example 1

The above diagram shows a transition diagram. A string which is translated into it is read from left to right, for example *abbb* would be accepted as it finishes in the accept state however *abbab* would not be as it finishes at node 3 not at the finish state.

A DFA's states are represented as circles (or other shapes for convenience when it's meaning is clear) in the transition diagram. A DFA begins in the initial state, denoted by the arrow pointing into a shape - here the initial state is 1. It the reads the input string, one symbol at a time and for each symbol read it makes a transition to a new state according to the labelled arcs. One or more states can be accepting states - denoted by a double circle. Here only state 4 is an accepting state. If after reading the complete string, the DFA is in an accepted state - we say that the string is accepted and otherwise it is rejected.

DFAs are just another way to represent something that we could use a RegEx to represent. The RegExs:

$$r = abb^*$$
$$r = ab^+$$

could be used to represent the DFA shown above.

5.2.1 A More Mathematical Example...

Considering the DFA that accepts strings of decimal digits containing a single decimal point:

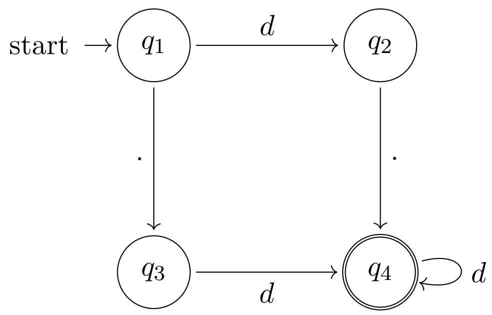

Figure 5.2: DFA Example 2

The DFA above is defined as:

$$\Sigma = \{1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$$
$$Q = \{q_1, q_2, q_3, q_4\}$$

q_1 is the initial stage, and $F = \{q_4\}$ is the set of final states. The transition functions can be represented by a set of triples:

$$\begin{aligned} \delta = \{ & \\ & (q_1, 0, q_2)., \dots, (q_1, 9, q_2), \\ & (q_1, ., q_3) \\ & (q_2, 0, q_2), \dots, (q_2, 9, q_2) \\ & (q_2, ., q_4) \\ & (q_3, 0, q_4), \dots, (q_3, 9, q_4) \\ & (q_4, 0, q_4), \dots, (q_4, 9, q_4) \\ \} & \end{aligned}$$

In each triple (q_i, a, q_j), q_i is the current state, a is the input and q_j is the state which the DFA will transit to. For example: $\delta(q_i, a) = q_j$.

5.3 Language of a DFA

The set of all strings accepted by the DFA is known as the *language recognised* by the DFA. This means that for a DFA, M, the language $L(M)$ is defined as 'the set of all strings $w \in \Sigma^*$ such that when the DFA starts processing w from its initial state it ends up in an accepting state'.

For example, the language for the string processing DFA (Fig 5.1) is a set of strings:

$$L(M) = \{ab^n | n \geq 1\}$$

There are in fact algorithms that, given a regular expression r, builds a DFA (or NFA) M such that:

$$L(M) = L(r)$$

5.3.1 Simplification to DFA Diagrams

We often have a large number of transitions between two states in a DFA. For example - in an identifier recognition DFA, there are 52 arcs for English letters (labelled by each of the lower and upper-case letters); and 10 for digits. Drawing that many transition arcs to a DFA is not a good choice. For this reason, we name a set of symbols:

$$letter = \{a, b, c, \dots, z, A, B, C, \dots, Z\}$$
$$digit = \{0, \dots, 9\}$$

and replace their arcs by a single arc labelled *letter* or *digit* (or simply *d* as in our example).

Therefore, the regular expression

$$ident = letter(\_|letter|digit)^*$$

can be represented with the following DFA

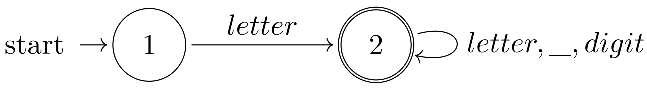

Figure 5.3: Example DFA 3

5.3.2 A More Sophisticated Example

If we take the example of a DFA for the lexical analyser for a minimal language that includes:

- identifiers
- the symbols: `:=`, `+`, `;`
- keywords `program`, `begin`, `end`, `input` & `output`

It should output tokens `END OF INPUT`, `ERROR`, `IDENTIFIER`, `ASSIGN`, `PLUS`, `SEMI COLON`, `PROGRAM`, `BEGIN`, `END`, `INPUT` & `OUTPUT`

The DFA begins in its initial state `START`, and a token is recognised as soon as the accepting state `DONE` is recognised. For example, if the arc labelled `+` is taken, then token `PLUS` has been recognised. The re-read means that the input character resulted in this particular transition being taken should be read again (because it is the first character of the next token). The state names `IN_ID` and `IN_ASSIGN` can be replaced with any other meaningful names.

5.4 Building a Lexical Analyser

Lexical analysers tend to be built in three ways:

1. Write a formal definition of the token patterns, which are used as an input to a software tool (i.e. *Lex*) which automatically generates a lexical analyse

or, design DFAs that describe the token patterns, then

2. write a program to implement the diagram
3. write a table-driven implementation of the DFA.

There are algorithms that construct lexical analysers from DFAs automatically.

5.4.1 Lex

The *Lex* program is a lexical analyser generator. It was written in the 70s and new versions are available.

- Flex: a variant of the classic “lex” (C/C++)
- JLex: lexical analyser generator for Java, written in Java
- Quex: A fast universal lexical analyser generator for C and C++

The program *Lex* takes as input a source file (called a Lex file) comprising regular expressions for various tokens and automatically generates (most of) a lexical analyser in C. Therefore, the work of creating a lexical analyser using Lex is most about preparing Lex files.

In its most basic form - a Lex file comprises a series of lines of the form

```
pattern                 action
```

where `pattern` is a regular expression and `action` is a piece of code. For example the following is a complete Lex file that displays token names rather than returning tokens to the syntax analyser.

```
%%
[ \n\t]+                { ; }
if                      { printf("IF\n"); }
then                    { printf("THEN\n"); }
[0-9]+                  { printf("NUMBER\n"); }
[a-zA-Z][_0-9a-zA-Z]*   { printf("IDENT\n"); }
.                       { printf("ERROR\n"); }
%%
```

The patterns on the left are all regular expressions, although in a slightly different notation to what has been discussed in this module so far. Running this Lex file on Lex produces a C file, which is then compiled by a C compiler to produce the lexical analyser.

Page 6

Lecture - Describing Language Syntax

2024-02-16 14:00 Jiacheng

6.1 Context-Free Grammars

The linguist Noam Chomsky introduced four classes of formal grammars for describing natural languages:

- regular
- context-free
- context-sensitive
- recursively enumerables

Of the above listed, two have been found to be useful to describe programming languages:

- regular grammars (equivalently, regular expressions) are useful for describing languages' lexical structure
- context-free grammars (CFG) for defining their syntax.

By far, CGF is the most widely used way to describe a programming language.

6.1.1 What is A CFG?

A CFG is a tuple: $G = (T, N, S, P)$ where:

T is a finite non-empty set of terminal symbols, which consist of strings in the language (for example `while`), which refer to parts of the text of sentences in the language

N is a finite non-empty set of non-terminal symbols, disjoint from T. These refer to syntactic structures defined by other structures and rules (for example `<exp>`)

S the start symbol, where $S \in N$

P is a set of (context-free) productions of the form $A \to \alpha$ (which reads A produces α) where $A \in N$ and $\alpha \in (T \cup N)$

6.1.2 Example of a CFG

Taking the CFG as $G_1 = (T, N, S, P)$ where:

$$T = \{a, b\}$$
$$N = \{S\}$$
$$P = \{S \to ab, S \to aSb\}$$

We know that this is a CFG because it has T which is a set of symbols available in the language; N containing the start-state (therefore non-terminal) S and a set of rules of production P.

To take the CFG $G_2 = (T, N, S, P)$ where:

$$T = \{a, b\}$$
$$N = \{S, C\}$$
$$P = \{S \rightarrow \varepsilon, S \rightarrow C, S \rightarrow aSa, s \rightarrow bSb, C \rightarrow a, C \rightarrow b\}$$

6.1.3 Shorthand Notation

It's lovely writing the CFGs out in full however this takes up quite a lot of space. So, instead of writing each individual rules of a given non-terminal, we are able to group the alternative right hand sides and separate them using |. For example, G_2 can be written as follows:

$$S \rightarrow \varepsilon|C|aSa|bSb$$
$$C \rightarrow a|b$$

6.1.4 Symbol Choice Conventions

As we have seen in the above examples, there is a standard naming conventions for the symbols used:

- $a, b, c, \ldots$ for members of T (single terminal symbols)
- $A, B, C, \ldots$ for members of N (single non-terminal symbols)
- $\ldots, X, Y, Z$ for members of $T \cup N$ (single symbols, terminal or non-terminal)
- $w, x, y, z, \ldots$ for members of T^* (strings of terminal symbols)
- $\alpha, \beta, \gamma, \ldots$ for members of $(T \cup N)^*$ (mixed strings of terminals and / or non-terminal symbols)

Here $a, b, c, \ldots$ refer to letters at the beginning of the alphabet (the set of all symbols) and $w, x, y, z, \ldots$ to letters at the end of the alphabet.

6.2 Backus-Naur Form (BNF)

Backus-Naur Form (BNF) is a popular notation for CFG definitions of real programming languages. BNF uses angle brackets to denote non-terminal symbols, in a similar way to XML tags. For example <exp>, <number> and <digit> are non-terminal; while +, -, *, /, 0, 1, … 9 are terminal symbols.

6.2.1 Syntactic Structures

Using the above symbols, the syntactic structure for arithmetic expression would be defined by the following productions:

```
<exp> → <exp> + <exp> | <exp> + <exp> |
         <exp> * <exp> | <exp> / <exp> | (<exp>) | <number>

<number> → <digit> | <digit> <number>

<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

In the above example, <exp>, <number> and <digit> are non-terminals while +, -, *, /, 0, 1, … 9 are terminals.

6.2.2 Grammar for a Simple Language

Below is a complete example of a BNF definition for a simple language:

```
     <program> → begin <stmt-list> end
   <stmt-list> → <stmt> | <stmt> ; <stmt-list>
        <stmt> → <assign>
                 | <input-stmt>
                 | <output-stmt>
      <assign> → <ident> := <exp>
         <exp> → <ident> | <exp> + <exp>
  <input-stmt> → input <ident>
 <output-stmt> → output <ident>
       <ident> → x | y | z
```

6.3 Derivations

We can use a Context-Free Grammar to *derive* strings of terminal symbols. Starting with the start symbol, S, we repeatedly apply the production rules until we obtain a string comprising only of terminal symbols, which is called a sentence. This process is called a derivation. Every string of symbols in a derivation is a sentential form.

The language defined by a grammar is made up of exactly those sentences that can be derived from it.

6.3.1 An Example

If we consider the grammar $G_2(T, N, S, P)$ where:

$$T = \{a, b\}$$
$$N = \{S, C\}$$
$$P = \{S \to \varepsilon, S \to C, S \to aSa, s \to bSb, C \to a, C \to b\}$$

We are able do derive the string $abbba$ for G_2 in the following steps:

1. We begin with the start symbol S
2. Applying the rule $S \to aSa$, we replace the S by aSa to obtain the string aSa.
3. Applying the rule $S \to bSb$ on the new string aSa, we replace the S by bSb to obtain the string $abSba$
4. Applying the rule $S \to C$, we obtain the string $abCba$
5. Applying the rule $C \to b$, we obtain the string $abbba$ of terminal symbols.

6.3.2 Notation

As we can see above, that is quite a long handed approach to righting out a derivation. There is a shorthand way of writing out a derivation as we will see below.

If we can get from α to β by applying a single application rule, we say *alpha* immediately derives β written

$$\alpha \Rightarrow \beta$$

(note the double line'd arrow used here, not a single line as we have seen above for production rules)

We can therefore write the full derivation of *abba* from S as:

$$\begin{aligned} S &\Rightarrow aSa \\ &\Rightarrow abSba \\ &\Rightarrow abCba \\ &\Rightarrow abbba \end{aligned}$$

6.3.3 Full Derivation Example

We've now seen all the components to derivation, so now we will see a full example. The grammar of our language is as follows:

```
<program> → <stmts>
  <stmts> → <stmt> | <stmt> ; <stmts>
   <stmt> → <var> = <expr>
    <var> → a | b | c | d
   <expr> → <term> + <term> | <term> - <term>
   <term> → <var> | const
```

In the above grammar, <program> is the start symbol, making it non-terminal; <stmt>, <stmts>, <expr>, <term>, and <var> are non-terminals; whereas a, b, c, d, +, - and const are terminals.

We can derive a single line program:

```
a = b + const
```

from this grammar:

```
<program> ⇒ <stmts>
          ⇒ <stmt>
          ⇒ <var> = <expr>
          ⇒ a = <expr>
          ⇒ a = <term> + <term>
          ⇒ a = <var> + <term>
          ⇒ a = b + <term>
          ⇒ a = b + const
```

Lines 5 and 6 are in Sentential Form and the final line (ln. 8) is the finished sentence.

There is a second example of a derivation in the lecture06 slides on Moodle.

6.3.4 Leftmost & Rightmost Derivations

Considering the following grammar for arithmetic expression

```
<exp> → <exp> + <exp> | <exp> * <exp> | x | y | z
```

A derivation of the sentence x + y * z from this grammar could be:

```
<exp> ⇒ <exp> + <exp>
      ⇒ x + <exp>
      ⇒ x + <exp> * <exp>
      ⇒ x + y * <exp>
      ⇒ x + y * z
```

Which is known as a *leftmost* derivation because, at each step, the leftmost non-terminal symbol is resolved.

It is also possible to have a rightmost derivation:

<exp> ⇒ <exp> + <exp>
⇒ <exp> + <exp> * <exp>
⇒ <exp> + <exp> * z
⇒ <exp> + y * z
⇒ x + y * z

It is also possible to have a neither left-nor right-most:

<exp> ⇒ <exp> + <exp>
⇒ <exp> + <exp> * <exp>
⇒ <exp> + y * <exp>
⇒ x + y * <exp>
⇒ x + y * z

6.4 Parse Trees

We are able to illustrate the structure of an the expression given by a derivation as a parse tree.

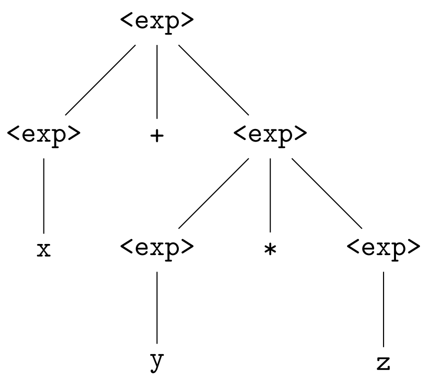

Figure 6.1: Parse Tree for derivation of `x + y * z`

The derivation of `x + y * z` can be seen below

<exp> ⇒ <exp> + <exp>
⇒ x + <exp>
⇒ x + <exp> * <exp>
⇒ x + y * <exp>
⇒ x + y * z

The internal nodes of the parse tree contin non-terminal symbols whereas leaf nodes contin terminal symbols.

It should be noted that for some grammars, the derivation of a given sentence can be different. Meaning they have different parse trees. This will commonly be the case when applying different ordered derivations.

Page 7

Lecture - Syntax Analysis and Parsing

2024-02-19 1400 Jiacheng

7.1 Ambiguities in Grammars

Unsurprisingly, given the complex and custom nature of grammars - it is possible to make them ambiguous. This would mean that the derivation of given sentences could be different, therefore the parse trees are different.

For example if we take a left-most derivation for the sentence `x + y * z`:

```
<exp> ⇒ <exp> + <exp>
      ⇒ x + <exp>
      ⇒ x + <exp> * <exp>
      ⇒ x + y * <exp>
      ⇒ x + y * z
```

However, we can first apply the rule:

```
<exp> → <exp> * <exp>
```

which would yield:

```
<exp> ⇒ <exp> * <exp>
      ⇒ <exp> + <exp> * <exp>
      ⇒ x + <exp> * <exp>
      ⇒ x + y * <exp>
      ⇒ x + y * z
```

These two derivations would produce different parse trees:

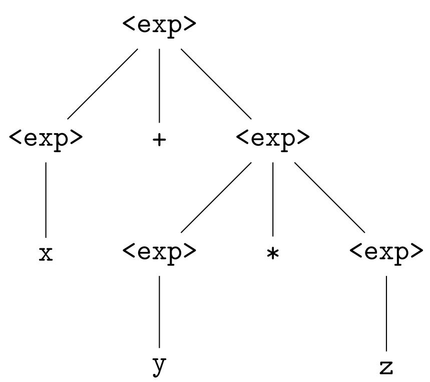

Figure 7.1: Parse Tree for derivation of `x + y * z`

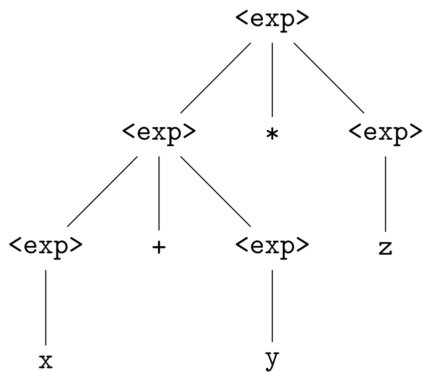

Figure 7.2: Parse Tree for alternate derivation of `x + y * z`

An *Abstract Syntax Tree* only shows the terminal symbols, without showing expressions.

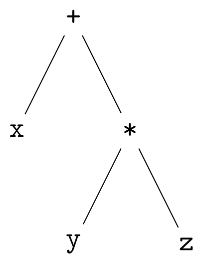

Figure 7.3: Example of an Abstract Syntax Tree

7.1.1 Avoiding Ambiguity

Ambiguity in grammars should be avoided, as we know from primary school - ambiguity in mathematical expressions has been removed through having an order of precedence which we can use brackets as part of.

7.1.2 Removing Ambiguity

For nearly all programming languages, the ambiguities in a grammar can be removed. To do this, extra non-terminals and rules are added. For example, taking the example grammar below:

<exp> → <exp> + <exp> | <exp> * <exp> | x | y | z

which can be disambiguated by adding rules that force certain operations (`+`) to appear above other operators (`*`) in parse trees. Thus giving the standard precedence of `+` then `*`. The new grammar can be seen below:

<exp> → <exp> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → x | y | z

Note that `term` and `factor` are newly introduced non-terminals.

7.1.3 Limits of Context-Free Grammars

There are some aspects of programming language syntax that cannot be captured using a context-free grammar. For example, the rule that variables must be declared before they are used is a *context sensitive* property. Context-sensitive properties are resolved by the semantic analyser (which is beyond the scope of this module).

7.2 Syntax Analyser

For any given input program, the goals of syntax analysis (also known as parsing) are to:

- find all syntax errors, and for each discovered - produce an appropriate diagnostic message and recover quickly
- produce the parse tree for the program for code generation

These two functions are carried out by the syntax analyser (also known as the parser). There are different algorithms for parsing which are completed by different parsers.

7.2.1 An Example

If we take the grammar:

```
<exp> → <exp> + <exp> | <exp> * <exp> | x | y | z
```

the source code:

```
x + y * z
```

It will produce the tokens outputted by the scanner:

```
IDENT PLUS IDENT MULTI IDENT
```

And the following Parse Tree:

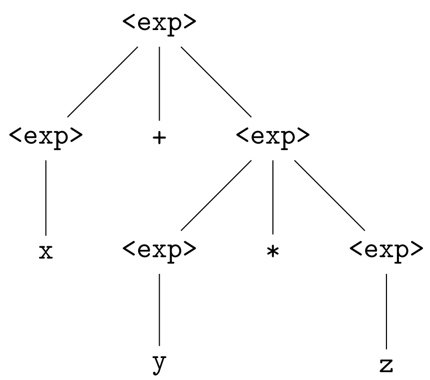

Figure 7.4: Example Parse Tree for derivation of `x + y * z`

With the following abstract syntax tree:

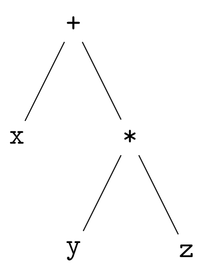

Figure 7.5: AST For example

7.3 Parsers

There are two categories of parsers.

Top-Down Parsers begin with the root (the start symbol of the grammar rules) then visit each node (of the parse tree) before the branches are followed. For a left-most derivation, the branches from a given node are visited in left-to-right order.

Bottom-Up Parsers begin at the leaves of the parse tree (which are terminal symbols) and progress towards the root. The order is that of the reverse of a rightmost derivation.

7.3.1 Parsing: An Example

If we consider the following grammar:

$$S \to AB$$
$$A \to aA|\varepsilon$$
$$B \to b|bB$$

The above grammar defines strings consisting of any number (0 included) of a's followed by at least one (with the possibility of more) b's.

If we consider parsing the string $aaab$ using the grammar by top-down parsing:

1. begin with the start symbol S and read the sentence one character at a time from the left
2. at each step - expand the leftmost non-terminal by replacing it with the right side of one of its productions
3. repeat until only terminals remain

Shown below is the full parsing for the string:

$\boldsymbol{S}$ Begin with S (start symbol)

$\boldsymbol{A}B$ $S \to AB$ (replace S with the right hand side of $S \to AB$)

$a\boldsymbol{A}B$ $A \to aA$ (the leftmost non-terminal is A. On seeing first input a, replace A with the right hand side of $A \to aA$)

$aa\boldsymbol{A}B$ $A \to aA$ (on seeing 2nd a)

$aaa\boldsymbol{A}B$ $A \to aA$ (on seeing 3rd a)

$aaa\varepsilon\boldsymbol{B}$ $A \to \varepsilon$ (on seeing b, use $A \to \varepsilon$ to make A disappear as there is no rule for $A \to b$ and we can't work on non-terminal B yet)

$aaa\boldsymbol{B}$

$aaab$ $B \to b$ (on seeing b)

The top-down parse of the string $aaab$ is a left-most derivation of the sentence, which verifies that $aaab$ is a legal sentence.

At each step of top-down parsing:

- Give a general sentential form, $xA\alpha$ where x is a string of terminal symbols, A is a non-terminal symbol and α is a mixed string of terminal & non-terminal symbols.

- This means that A is the leftmost non-terminal that must be expanded to get the next sentential form in a leftmost derivation.
- Determining the next sentential form is a matter of choosing the correct grammar rule that has A as its left hand side.

For example, with the current sentential form, $xA\alpha$, suppose the grammar has three rules for A:

$$A \to bB$$
$$A \to cBb$$
$$A \to a$$

Depending upon the next input being a, b or v, a top-down parser must choose among these three rules to generate the nex sentential form (from $xA\alpha$).

7.3.2 Predictive Parsers

Different top-down parsing algorithms may use different information to make parsing decision (choosing the correct rules). Most parsers compare the next input token with the first symbols that can be generated by the right hand side of those rules, therefore called predictive parsers. A predictive parser is characterized by its ability to choose the production rule to apply solely based on the next input symbol and the current non-terminal being processed:

- Current non-terminal ⇒ choose the candidate rules
- Next input ⇒ choose one rule among the candidates

Over this lecture and the next one, we will explore two different implementations of a predictive (top-down) parser.

- Recursive descent parsers - a coded implementation of a syntax analyser based on BDF description of syntax
- LL (1) parsers - driven by a parsing table (created from BNF grammar) with: 1st L is a left-to-right scan of input; 2nd L is a leftmost derivation; and the '1' meaning one input symbol of lookahead (i.e. predictive)

7.4 Recursive-Descent Parsers (RDP)

A Recursive-Descent Parser (RDP) is made up of a collection of subprograms, many of which are recursive (hence where its name comes from) and produces a parse tree in top-down order.

Within an RDP - there is a subprogram for each non-terminal in the grammar.

7.4.1 RDP Example

If we take the following grammar for a small set of simple expressions:

```
     <exp> → <term> + <term> | <term> - <term>
    <term> → <factor> * <factor> | <factor> / <factor>
  <factor> → id | int_constant | (<exp>)
```

We can re-write the grammar for <exp> and <term> in a more compact form, because the only difference in the two alternatives in both cases is a terminal symbol (+ or -, * or /):

```
     <exp> → { ( + | - ) <term> }
    <term> → { ( * | / ) <factor> }
  <factor> → id | int_constant | (<exp>)
```

To implement a programmed recursive-descent parser, we need to implement three subprograms:

- `exp()`
- `term()`
- `factor()`

We would also assume that we have a lexical analyser (for example `lex()`) that puts the next token in a variable called `nextToken`. Upon receiving a token, the parser will decide:

- if it is a terminal of current rule (i.e, `+`, `-`, `*`, `/`, `id`, …), continue to have the next token (make a call to `lex()` to get `nextToken`).
- if it is a non-terminal symbol fo the right hand side of the current rule (i.e. `<term>`, `<factor>`), call it's associated parsing subprogram for that non-terminal
- if it is neither terminal nor non-terminal of the current rule or something else, then there is a syntax error.

7.4.2 Rules with Multiple Right Hand Sides

Where a rule ha multiple right hand sides, a decision has to be taken as to which one to use. The correct RHS could be chosen based on the next token of input - which is compared with the first token of each RHS until a match is found. If no match is found, it is a syntax error.

Page 8

Lecture - LL1 Parsers

2024-02-23 1400 Jiacheng

8.1 Top Down Parsing: Recursive Descent Parsers

Last lecture, we saw the *Recursive Descent Parsers* which are the coded implementation of a syntax analyser. We saw that the RDP consists of a collection of subprograms, and that there is a subprogram for each non-terminal in the grammar. Many of these subprograms are recursive, hence it's name.

8.2 Left Recursion Rules

If a grammar makes uses of left recursion, either directly or indirectly, it cannot be used directly by a recursive-descent parse. A left recursion is where the definition is defined in terms of itself, for example $A \to A\alpha$. There are two different types of Left Recursion.

- **Direct Left Recursion**, where the rule can directly invokes itself without making any progress in the parse string. For example:
$$A \to S\beta$$
- **Indirect Left Recursion**, where there are multiple stages to the left recursion as seen below:
$$S \to A\beta$$
$$A \to S$$

Ultimately, left recursion leads to indefinite / non-terminating recursion. However, we are able to transform a left-recursive grammar into one that is not.

8.2.1 Left Recursion Removal

For each non-terminal involved in the Left Recursion,A, there are two steps which have to be undertaken.

1. Group the A-rules as: $A \to A\alpha_1|\ldots A\alpha_m|\beta_1|\beta_2|\ldots|\beta_n$ where $A \to A\alpha_1|\ldots|A\alpha_m$ are rules with left recursion and $A \to \beta_1|\beta_2|\ldots|\beta_n$ are rules without left recursion.
2. Introduce a new non-terminal, A', and replace the original rules with:
$$A \to \beta_1A'|\beta_2A'|\ldots|\beta_nA'$$
$$A' \to \alpha_1A'|\alpha_2A'|\ldots|\alpha_mA'|\varepsilon$$

8.3 Top Down Parsing: LL(1) Parsers

LL(1) Parsers are table-driven Predictive Parsers. The LL(1) stands for the what the parser does:

- 1st L: left-to-right scan of input.
- 2nd L: leftmost derivation
- "1": means one input symbol of lookahead

A LL(1) parser utilises: a *stack* to store the symbols on the right-hand side of the productions in right-to-left order so that the leftmost symbol is on top of the stack; and a *parsing table* which stores the actions (ie the rules) the parser should take based on the input token and what value is on top of the stack.

8.3.1 Example

If we consider the following grammar:

$$
\begin{aligned}
E &\to T\ E' \\
E' &\to +TE'|\varepsilon \\
T &\to FT' \\
T' &\to *FT'|\varepsilon \\
F &\to (E)|int
\end{aligned}
$$

Here, we have non-terminal symbols E, T and F which may stand for `<exp>`, `<term>`, `<factor>` or other structures. E is our start symbol.

Top of parse stack	**int**	+	*	(	)	**$**
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to^* FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to$ int			$F \to (E)$		

Table 8.1: Parse Table

T
E'
$

Figure 8.1: Stack

(	int	+	int	$*$	int	$\cdots$	$

Figure 8.2: Input Strings (tokens)

8.3.2 Parsing Table

For LL(1) parsing, the grammar is arranged into a parsing table.

- The first column has the non-terminal symbols of the grammar,
- The first row contains the terminal symbols and $ is for the end of input
- The table entries give the rules of choice based on the current input (a terminal symbol) and the current non-terminal symbols (the symbol on top of the stack)

8.3.3 Parsing Procedure / Algorithm

Each step in parsing is about choosing a rule from the table according to the current non-terminal (which is on top of the stack) and current input; then pushing it's right-hand side into the stack. The $ symbol represents the bottom of the stack.

The process begins with the start symbol (E in our ongoing example). We push the right-hand side of the E expression ($E \to TE'$) to into the stack, working right-to-left. This means that the top of our stack is now T.

Now, if we take the current input to be *int*. According to the table, $T \to FT'$ is selected and it's right hand side is pushed into the stack. Now the top of the stack is F.

Page 9

Lecture - Syntax Analysis - Bottom Up Parsing

2024-03-01 1400 Jiacheng

9.1 Parse Table Construction

LL(1) parse is easy if the action table is available. The construction of the action table of the LL(1) parser requires computing the first and follow sets (sets of non-terminal symbols) of the non-terminals of the grammar.

NB: There's lots of information on how the First sets of Non-Terminals work on the slides on Moodle.

9.2 Bottom-Up Parsing

Bottom-Up parsing works in the opposite direction of top-down parsing. This means that it will start with the string of terminal symbols and then works backwards to the start symbol by applying the productions in reverse.

1. Begin with the rightmost symbol for the sentence of terminals
2. Apply a production in reverse at each step
3. Replace a substring with the non-terminal on the left of a production THiswhose right side matches the substring
4. Continue until we have substituted our way back to the start symbol

For example, using the following grammar:

$$S \to AB$$
$$A \to aA|\varepsilon$$
$$B \to b|bB$$

We can then follow the parsing through:

$aaa\boldsymbol{b}$ (the sentence of terminals)

$aaa\boldsymbol{B}$ $B \to b$

$aaa\boldsymbol{\varepsilon}B$ (insert ε because a or aB is not RHS of any rule)

$aa\boldsymbol{aA}B$ $A \to \varepsilon$

$a\boldsymbol{aA}B$ $A \to aA$ (we cannot use $S \to AB$ yet because it terminates the parsing with some terminal symbols not being parsed)

$\boldsymbol{aA}B$ $A \to aA$

$\boldsymbol{AB}$ $A \to aA$

$\boldsymbol{S}$ $S \to AB$

An example of parsing the same string with Top-Down Parsing is available in the *Parsing and Syntax Analysis* lecture.

9.3 Top-Down vs Bottom-Up

Bottom-Up parsing algorithms are more powerful than top-down methods. There are excellent parser generators like *yacc* that build a parser from an input specification (a bit like how *lex* builds a scanner).

9.4 Shift-Reduce Parsing

Shift-Reduce Parsing is a bottom-up parsing technique which takes an input as a stream of tokens and develops the list of productions (the grammar rules) that are used to build the parse tree. It makes use of a stack to keep track of the position in the parsing process and a parsing table to determine what to do next.

Shift-Reduce parsing is the most used and most powerful bottom-up parsing techniques.

9.4.1 The LR Parser

The LR parser is a type of shift-reduce parser which scans the input from left-to-right and operates using a reversed rightmost derivation.

The LR Parser works as follows

- When parsing a string of tokens v, the input is initialised to v (ended with the end-marker \$) and the stack is initialised to empty.
- Parsing starts by shifting the first (leftmost) token to the top of the stack
- A shift-reduce parser proceeds by taking one of three actions at each step

  shift where we transfer a token from the input onto the stack. In an ideal world, we would want to run a reduce or acceptance however if neither are possible - we run a shift.

  reduce where we apply the production for a non-terminal backwards (replacing the RHS with the LHS of a grammar rule). For example if we have a production $A \to w$, we can produce A from any ws we have at the top of the stack

  acceptance where we reduce the entire contents of the stack to the start symbol, with no remaining input means the input is a valid sentence. We would then say that the string is "accepted".

- If none of the above cases apply, we have an error. An error is a case where we have a sequence on the stack that cannot eventually be reduced to the LHS of any production; and any further shift would be futile and the input cannot form a valid sentence.

Page 10

Lecture - Names, Bindings, Scopes and Memory Allocation

2024-03-11 | 1400 | Jiacheng

10.1 Variables & Attributes

A *Variable* is a place holder (a named memory location) for a value at run-time. Variables have several attributes at run-time which includes:

name which provides the means for accessing a variable

address (also known as it's l-value) which is what is required when the name of the variable appears in the left side of an assignment

value which is the contents of the memory cell(s) (also known as the r-value of a variable)

type which determines the range of values that the variable can store and the operations that are defined for the values

lifetime which defines the time during which a variable is bound to a specific memory location (i.e. it's address)

scope which is the range of statement in which the variable is visible / accessible

10.2 Declaration

In most languages, variables are introduced using a declaration. This can either be done implicitly or explicitly.

10.2.1 Explicit Declaration

An Explicit Declaration is a program statement used for declaring the type of the variable. For example, a variable, `i`, is declared in different languages as follows:

- in Pascal `i : integer`
- in Java `int i;`
- in Fortran `INTEGER :: Count`

10.2.2 Implicit Declaration

An implicit declaration is a default mechanism for specifying types of variables through default conventions rather than declaration statements. For example, Fortran has both explicit and implicit declarations:

Explicit

```
INTEGER :: Count, Total
REAL :: Average, Sum
COMPLEX :: c
```

Implicit The default implicit typing rule is that if the first letter of the name is `I`, `J`, `K`, `L`, `M`, or `N`, then the data type is integer, otherwise it's real.

10.2.3 Type Inference

Another kind of implicit type declaration is type inference. This is popular in *modern, edgy* language such as JavaScript and C# whereby a variable can be declared with `var` or `let` and set with an initial value. This initial value sets the type of the variable. Python takes this one step further whereby a variable is just defined. That's it, there is no keyword required to say that you're defining a variable. For this to work, an initial value must be provided.

Type Inference (a way to declare a type) is different from reference type (a data type, e.g. pointer type)

10.3 Binding

A binding is an association between an entity and an attribute, for example - between a variable and it's type, or value, or memory location; or between a symbol (e.g. `+`) and an operation.

Binding time is the time at which a binding takes place, which can be at different times.

10.3.1 Binding Time

If we take the following Java Statements:

```
int count;
count = count + 5;
```

We can then review the attributes of the statements and when they're bound.

type of `count` is at *compile time* (because is a typed language, a variable must be declared before it's used)

range of possible values of `count` is bound at *compiler design time*

operation (the meaning) of the operator `+` is bound at *compile time* when the types of its operands have been determined

value of `count` is bound at the execution of the statement, i.e. *runtime*

It is also possible for binding to take place at load time or link time.

Load time whereby we are binding a C or C++ `static` variable to a memory location

Link time whereby we are binding the variables in one module with those in another

10.3.2 Type Binding: Static vs Dynamic

A type binding is static if it occurs before runtime and remains unchanged throughout program execution. For example, declarations (implicit or explicit) always carry the type information, and binding is done at compile time. Languages which conform to this are said to be statically typed and include Java, C, Fortran, Pascal, etc. It is advantageous to use one because they are more efficient and the type errors can be detected by the compiler. However, they are disadvantageous because they're not very flexible for interactive applications.

Languages which complete type-binding during execution, or where they can change during execution of the program (specified through an assignment statement) use *dynamic* type binding. Examples of languages which do this include Python, JavaScript and PHP. It is advantageous to use a dynamically typed language as they have greater flexibility (given the possibility to produce generic program units); and allow for rapid prototyping. However they are disadvantageous as they have a higher resource cost (given the dynamic type checking at run-time); and that the type-error detection by the compiler is difficult.

10.3.3 Strong vs Weak Typing

Languages in which all the type errors (where the use of an operator to an operand of an inappropriate type) can be detected by the compiler, that language is known as *type safe*. For example, Java, Haskell and Ada are type safe; however FORTRAN and C are not. Type safe languages are said to be *strongly typed*. Strongly typed languages can either be static of dynamic typed.

Languages which aren't type safe are said to be *weakly typed*. Dynamic Typed does not necessarily mean weakly typed, for example Python's typing is dynamic but it is strongly typed.

Some languages as a whole are not typed safe, however they might have a subset of instructions that are type safe - for example Mesa for Systems Programming.

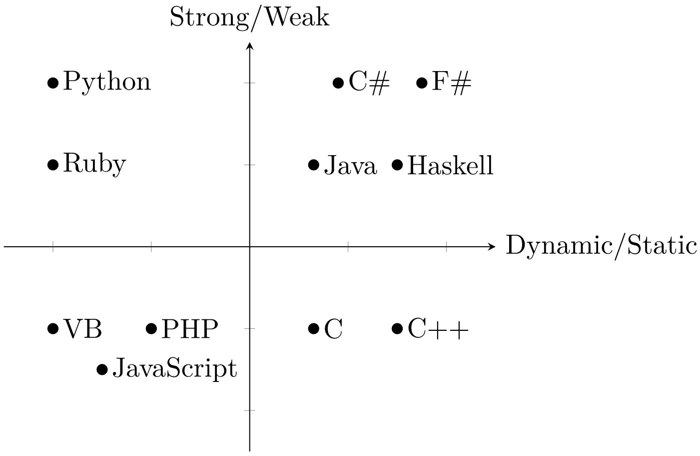

Figure 10.1: Strong-Weak vs Dynamic-Static Language Typing

10.4 Block Structure of Programs

Modern programming languages use *blocks*. These are subsections of code which have a specific purpose and they define a local reference environment which is useful in relation to variable binding & scope. Blocks are denoted by start and end markers, in sensible languages (such as C, C++, Java) this is braces (`{` and `}`); however in silly languages such as Python, this is through indentation.

A block can contain declaration of variables local to that region and have its own *reference environment* (the accessible names / identifiers)

10.4.1 Nesting Blocks

Blocks are typically nested. This is seen in Java and C in that method blocks are always nested within class blocks and other code blocks can be nested within these. However in Java and C, it is not possible to nest methods within methods. We can also see nesting blocks in Python and pascal, where procedures (which have no return value) and functions (which have a return value) can be nested.

When designing an algorithm, we have to consider the local reference environment to decide the scope identifiers in name resolution. This involves considering: where these identifiers can be used / accessed; and what happens if we have two identifiers which have the same name.

10.5 Scope of Identifiers

The scope of an identifier (e.g. a variable or a procedure) is the part of the program text that can access the identifer. If we take the following Pascal program as an example:

```
program my_prog;
var
    i, j : integer;
procedure f (k : integer);
var
    j : integer;
begin
    j := k + i;
end
procedure g (k : integer);
begin
    f(k+1);
end
begin
    i := 5; j := 3; g(j);
end.
```

In most languages, an identifiers scope is the block in which it's declared. For example, in the above program:

- the variables, `i`, `j` declared in the outermost block are *global variables* (their scope is the whole program)
- the scope of parameter `k` and local variable `j` is the body of procedure `f`. The use of variable `j` (in the body of `f`) is bound to its local declaration (not the `j` that has been globally declared)
- procedures `f` and `g` are declared in the outermost block and are therefore global

10.5.1 Visibility Rules for Duplicated Identifiers

A declaration in an inner block hides a declaration of a variable in an enclosing block with the same name - this *visibility rule* is adopted by most languages within blocks.

Using the example above - the local variable `j` in procedure `f` hides the global variable `j`. Where this is the case - we say that there is *a hole in the scope* of the global variable `j` (as its scope is the whole program except procedure `f`).

10.5.2 Static and Dynamic Scoping

In the Pascal program above, the scopes of variables are governed by the visibility rules. Therefore the scopes of variables depend on the lexical structure (the layout) of the program which means that it is possible for a compiler to determine the scopes of all variables. This is known as *static* or *lexical* scoping.

By contrast, *dynamic scoping* is where the reference environment (the visibility of identifiers) depends on the calling sequence of subprograms, not the layout of the nesting structures. Therefore the scopes of identifiers can be determined only at runtime.

Dynamic scoping is not popular: it gives less-reliable programs because subprograms are always executed in the environment of all previously called subprograms that have not yet completed their execution. Dynamic scoping also leads to programs of poor readability. However, it is easier to implement using stacks and resolving a variable does not require tracing the syntactic structure of the entire program. Dynamic Scoping was used by early versions of LISP and is still in operation in Perl and Common LISP.

10.5.2.1 An Example

If we consider the following pseudocode (based on JavaScript syntax) function, `big`, in which two functions `sub1` and `sub2` are nested:

```
function big(){
    function sub1(){
        var x = 7;
        sub2();
    }
    function sub2(){
        var y = x;
    }
    var x = 3;
    sub1();
}
```

The `x` which is referenced within `sub2()` depends on the scoping type.

In Static Scoping

- When resolving the identifier `x`, the search for `x` begins in the body of function `sub2()`. Here it is found that there is a reference for `x`, but no declaration for `x` is found.
- With static scoping, the search continues to the static parent of `sub2()` (the function `big()`), in which `x` is declared (in the line `var x = 3;`).
- If there is no declaration of `x` in `big()`, then an error is generated. The `x` declared in `sub1()` is ignored.

In Dynamic Scoping

- The process of resolving `x` in `sub2()` depends on the calling sequence of the functions. This cannot be determined at compile time.
- In this case, the searching for `x` also begins in the body of `sub2()`, where no declaration for `x` is found.
- After that, the search for `x` is rather different to static scoping - in that the call stack is searched until a declaration of `x` is located. In the above example, the search for `x` begins in `sub2()`, then moves to it's caller (also known as it's dynamic parent) `sub1()`, where `x` is declared (with the line `x = 7;`) and therefore is found. The value of `x` is then assigned to `y`

10.6 Lifetime of Variables

The *lifetime* of a variable is the period in which it "exists" and has a value assigned to it during the given program execution. For local variables within procedures, the lifetime is the current execution of the procedure; which means the variable is created when the procedure is called and destroyed when the procedure exits. Each (recursive) execution of a procedure has its own copy of the local variables. The lifetime of a global variable is the entire duration of the execution of the program.

10.6.1 Lifetime and Memory Allocation

The lifetime of a name / variable is related to how memory is allocated and deallocated for its values. There are three basic *memory allocation mechanisms*, each associated with a different area of memory.

static allocation is when a fixed memory address is retained throughout program execution (meaning the lifetime of the variable is the entire program execution)

stack-based allocation is done on a first-in-last-out basis and is used for procedure / function calls (the lifetime of the variable is the execution of the function)

heap-based allocation is used when storing variables in the *heap* (memory area where storage is dynamically allocated). Variables that are allocated from the heap are called *heap-dynamic* variables and they often do not have identifiers associated with them (making them *anonymous variables*) therefore they an only be referenced by pointers or by reference type variables (for example, a variable of Java object is a pointer to a memory location rather than some values).

10.6.2 Memory Allocation

Using the example of the above Pascal program:

- Memory is statically allocated for the global variables `i` and `j`
- Memory addresses for these global variables are determined at compile time, as are the locations of the machine code instructions
- Parameters and local variables are allocated on the stack; after the main body calls `g` which calls `f`.

Page 11

Lecture - Elementary Data Types

2024-03-15 14:00 Jiacheng

Elementary Data Types have fixed values that can be stored in a memory space of fixed size. Typically, languages provide the following kinds of elementary types:

- Numeric (these are integer / floating points)
- Boolean
- Character
- Enumerated
- Reference or pointer

We will not look at all of these data types in great detail.

Imperative languages provide built-in operators for computing over these elementary data types. A selection of these operators is provided in the table below.

Kind of Operator	Pascal	Python	C
Arithmetic	`+ - * /`	`+ - * /`	`+ - * /`
Int division	`div mod`	`/ %`	`/ %`
Relational	`= < > < <= > >=`	`== != < <= > >=`	`== != < <= > >=`
Boolean	`and or not`	`and or !`	`&& || !`
Bitwise	`-`	`& |`	`& |`

Table 11.1: Operations on Elementary Data Types

11.1 Enumerated Types

Enumerated types are *ordinal* type that the possible values can be associated with the set of positive integers. Enumerated types are used to represent order, for example used as indices of arrays.

In Java - the primitive ordinal types are integer and char. The char data type is a single 16-bit Unicode character, which has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535).

Enumerated types can be user-defined. For example, if we wanted to define a type to represent the four seasons in Pascal:

```
type season = (spring, summer, autumn, winter);
```

We can then declare a variable of type `season`:

```
var s : season;
```

11.1.1 Operations

Typically, enumerated types have a small set of operations - comparison and retrieval of the value of successor / predecessor value. We can see this in Java, which has included Enumerated Types (`enum`) since version 1.5. The example below defines an `enum` type `Season` and then sets `s` and displays the ordinal of it.

```
public enum Season {SPRING, SUMMER, AUTUMN, WINTER};
Season s = Season.SUMMER;
System.out.println(s.ordinal()); // outputs '2'
```

11.2 Pointers

Pointer (or reference) variables have a memory address as their value (they "point" to another data item). Originally, pointers were designed to work with memory locations, enabling efficient use of limited memory space - a feature of assembly language. They provide a convenient way to manage (allocate and de-allocate) dynamic storage (heap memory). Memory spaces that are dynamically allocated from a heap are heap-dynamic variables. They often are anonymous (meaning they do not have identifiers associated with them) and thus can be referenced only be pointers (memory addresses). C and C++ natively include lots of support for pointers.

11.2.1 Pointer Type Declaration

The declaration of a pointer is given by preceding a variable name with a `*`:

```
int number, *numPtr;
```

The above code snipped declares `number` as an `int` and `numPtr` as a "pointer to int". At runtime, this means:

- `number` is the name of a location in memory where an integer is stored
- `numPtr` is the name of a location in memory where a memory address is stored.

11.2.2 "&" Operator

The `&` operator retrieves the address of a variable. This can be used as follows:

```
number = 42;
numPtr = &number;
```

where the value of `numPtr` is the address of variable `number`.

11.2.3 Pointer Dereferencing

The dereferencing operator (`*`) of pointers to access the value pointed to by a pointer, for example if we wanted to print the value:

```
printf("%d\n", *numPtr);
```

The dereferencing operator is also used to modify the value of the memory location that the pointer points to:

```
*numPtr = 7
```

The line above updates the value pointed to by `numPointer` to be the value `7`.

11.2.4 Null Pointers

A special value `null` (actually stored as `0`) is assigned to pointer variables to signify that it doesn't point to any location. For example:

```
numPtr = null;
```

results in `numPtr` not actually pointing to anything.

11.2.5 Assignments Involving Pointers

If we take the following code:

```
int *p;
int *q;
int a = 100;
p = &a;

*q = *p;
```

The code above involves pointer de-referencing, whereby it copies the content of the memory location pointed to by `p` (`100`) to the location pointed to by `q`. This is distinctly different to the line

```
q = p;
```

which copies the value of `p` (a memory address) to `q`.

11.2.6 Dynamic Memory Allocation

Pointers are commonly used with dynamic memory allocation (where the memory allocation takes place at runtime). The following code allocates 400 bytes (100 spaces of 4 bytes each) and sets the variable `data` to point to the first byte. Note that both of the first lines below do effectively the same thing:

```
int *date = calloc(100, sizeof(int));
int *data = malloc(400);
```

There is, however, a difference between `malloc` and `calloc`

calloc allocates a block of memory of "n elements", each of `sizeof` bytes and initializes the contents of the memory to zeroes. It contains info about memory organisation. Example above has created an integer array of 100 elements.

malloc allocates specified size, with no organisation information, of memory however does not erase the memory space. This may lead to the memory containing garbage values.

11.2.7 Semantics of Pointer Allocation

When reading pointer arithmetic, it's important to understand it with reference to the pointer type on which they operate. For example:

```
int *p;
int *q;
p = (int *) malloc(sizeof(int));
q = (char *) malloc(sizeof(char));
p = p + 1;
q = q + 1;
```

If two `malloc` functions return `addr1` for `p` and `addr2` for `q`, after the increment operations - then the values of `p` and of `q` are not `addr+1` and `addr2+1` but instead are `addr1+sizeof(int)` and `addr2+sizeof(char)`.

11.2.8 Array Indices and Pointers

Arrays in C are essentially pointers to large blocks of memory. For example, if we take:

```
int a[10], *iPtr, sum, i;
iPtr = &a[0];
```

It would then be possible to access the individual elements of the array by using the indices of the array:

```
a[0], a[1], a[2], ..., a[9]
```

or by dereferencing the pointers:

```
*(iPtr + 0), *(iPtr + 1), *(iPtr + 2), ..., *(iPtr + 9)
```

This means the the following two for loops have the same effect

```
for(i = 0, sum = 0; i < 10; i++){
    sum += a[i];
}
```

```
for(i = 0, sum = 0; i < 10; i++){
    sum += *(iPtr + i);
}
```

It is also possible to use indices on pointers:

```
iPtr[2] is the same as
*(iPtr + 2)
```

When an array is declared, the memory is allocated and the array name holds the address to the first byte of this block of memory. The name of the array is like a pointer which points to the first element of the array (for example `a=&a[0]`)

```
int a[10];
int *iPtr;
iPtr = a + 2;
```

The last statement is the same as saying:

```
iPrt = &a[0] + 2;
```

it is also possible to do pointer arithmetic with array names. From this we can see that the following two statements are equivalent:

```
*(a + 2) = 11;
a[2] = 11;
```

11.2.9 Memory Deallocation

Deallocation of memory can be explicit or implicit

```
int *p = (int *) malloc(sizeof(int));
*p = 5;
p = null;
```

In the above example, we see that assigning `null` to `p` destroys the only way to access the allocated memory (which is still in existence). The runtime environment can recognise the inaccessibility of this piece of memory and free it implicitly (through a process called garbage collection).

However, it is also possible to explicitly deallocate memory using the `free(p)` statement as seen below:

```
int *p = (int *) malloc(sizeof(int));
int *q;
*p = 5;
q = p;
free(p);
p = null;
```

It is regarded as good practice to assign `null` to `p` after freeing it. Note that it is a semantic error (with an unpredictable result) to invoke `free` on a pointer which does not point to an object allocated using `malloc`

11.2.10 Dangling Pointers

Taking the above example, after freeing `p` - the pointer `q` points to a heap-dynamic variable that has been deallocated. `q` becomes a *dangling pointer* that can cause unexpected errors at runtime. Another example:

```
int *arrayPtr1;
int *arrayPtr2 = new int[100];
arrayPtr = arrayPtr2;
delete [] arrayPtr2; // returns memory to heap
```

Now, `arrayPtr1` is dangling because the heap storage to which it was pointing has been deallocated.

Page 12

Lecture - Compound & Composite Data Types

2024-03-22 14:00 Jiacheng

A *compound data type* is any data type which can be constructed using a programming languages' primitive data types. Common examples of compound types include: arrays, string, records, structs and variant records. Note that *records* aren't directly found in OO languages, rather classes can be seen as extensions of records.

12.1 Arrays

The *array* is the most common compound type found in programming languages. Generally speaking, an array has the following attributes:

- The *type* of its elements. This is also the type of the composite type
- The type of its *indices*. Integers are normally used, however it is possible to use other types (for example, enumerated type)
- The *index range* (number of elements)

As could be expected, the syntax and characteristics of arrays vary from language to language.

12.1.1 Arrays in Different Languages

We will consider two languages, Java and Pascal, to explore the differences between different languages implementations of arrays.

The first major difference is that in Pascal, the *size* and *indices* of an array are part of it's type definition, for example:

```
var rainfall: array [11 .. 20, 1..5] of real;
```

The type of the variable `rainfall` is an array of `reals` indexed by integers between 11 and 20 (1st dimension), and between 1 and 5 (2nd dimension).

Pascal arrays allow any ordinal type as its indices, for example the enumerated type as indices:

```
type month = (jan, feb, march, ..., dec);
var rainfall : array [jan .. dec] of real;
```

Or characters (128 ASCII characters, or 256 ANSI256 characters) as indices

```
var anArray : array ['a' .. 'z'] of integer;
```

Whereas in Java, arrays use integer type as its index and the index always starts from `0`:

```
double[] rainfall;
```

Java supports a number of different index sizes:

- Short (16-bit signed, -32768 to 32767)
- Byte (-128 to 127)
- Char (2 bytes, 65535)

These smaller types can be promoted (converted) to Integers which are bigger.

12.1.2 Static v Dynamic Arrays

As we saw in the Pascal definition example above, we now that the sizes of Pascal arrays are fixed at *compile* time. We say that Pascal arrays are "static". By contrast, we saw that Java allows "dynamic" arrays. This means their sizes are determined at *run-time* when they are created. For example:

```
rainfall = new double[numDays];
```

Dynamic arrays have the obvious advantage of allowing for the creation of more suitably-sized arrays.

12.1.2.1 Static Arrays

The subscript ranges (size of arrays) are statically bound and storage allocation is static at compile time. For example, all Pascal Arrays or C/C++ arrays which include the `static` modifier. A key advantage is that they are more efficient as there is no dynamic allocation.

12.1.2.2 Stack-Dynamic Arrays

The subscript ranges are dynamically bound and the storage is allocated at run-time. It lives on the stack and the size of it is fixed once it is created. For example, C/C++ arrays without the `static` modifier:

```
double balance [100];
myType stackArray[100];
```

A key advantage is that dynamic arrays are more flexible (as the size of an array need not be known until the array is to be used).

12.1.2.3 Heap-Dynamic Arrays

As the name would suggest, a heap-dynamic array lives on the heap. This means that it's size can change. C++, JavaScript, Python, etc support Heap-Dynamic arrays. For example, in C++:

```
myType *heapArray = new myType[100];
delete [] heapArray;
```

A key advantage is that the Heap-Dynamic arrays are hugely flexible, meaning that they can grow or shrink during program execution.

12.1.2.4 Java Arrays

Java Arrays are objects and live on the heap (in the same way that other Java objects do), but they behave like stack-dynamic arrays. It's seen that they have fixed sizes once initialised:

```
int [] a_array = new int (100);
```

12.1.3 Heterogeneous Arrays

A *heterogeneous* array is one in which the elements do not need to be of the same type. They are supported by Perl, Python and JavaScript. Shown below is an example of a heterogeneous array in JavaScript:

```
var mixArray1 = new Array();
mixArray1 [0] = "Hello, World!";
mixArray1 [1] = 100;
```

As would be expected, a heterogeneous array can be an element of another heterogeneous array:

```
var mixArray2 = new Array(
    200,
    300,
    "Hello Again",
    mixArray1
);
```

12.1.4 Rectangular & Jagged Arrays

A rectangular array is a multi-dimensional (for example, 2D) array in which all the rows have the same number of elements and all the columns have the same number of elements. This is supported by Fortran and C#. (Note that C# also supports jagged arrays).

A jagged array has rows with varying number of elements, however has no concept of columns. Therefore jagged multiple-dimensional arrays are actually arrays of arrays. C, C++, C# and Java support jagged arrays. For example, a jagged array in Java:

```
int [] [] myArray = { { 3, 4, 5 }, { 6, 7 } };
```

Jagged arrays are more flexible and memory efficient. For example the second dimension of array `a` can be allocated separately:

```
int[][] a = int[3][];
a[0] = new int[1]; // row 1 has 1 element
a[1] = new int[2]; // row 2 has 2 element
a[2] = new int[3]; // row 3 has 3 elements
```

If `a` turns out to be a big triangular array, almost half of the memory space can be saved by using a Jagged Array.

12.2 Strings

Strings can be regarded as arrays / collections of characters.

In (standard) Pascal, strings need to be declared as arrays:

```
var arr : packed array [1 .. 10] of char;
```

`packed` tells the compiler to use the minimum amount of memory.

In C, strings are declared as arrays:

```
char name[10]; // memory allocated
```

or as pointers to character:

```
char *name; // memory not yet allocated
```

In modern languages, strings are viewed as important enough to be built-in and a set of typical operations are defined for the string type such as:

- finding the length / size of a string
- string concatenation

Note that the char array in C does not have these operations built in.

12.3 Record

A *record* is composed of a number of named elements of data. For example in Pascal:

```
type date = record
    day: integer;
    month: integer;
    year: integer;
end;
```

We can see below how a variable `d` is declared to be of type `date`:

```
var d: date;
```

and how this has the *fields* `day`, `month` and `year`, which are accessed and modified using the dot notation / operator:

```
writeln(d.year);  // access
d.day = 24;       // modify
```

12.3.1 Structs

A similar similar to record in C (and C++) are called *structs*, for example:

```
struct date {
    int day;
    int month;
    int year;
}
```

However! A struct in C++ is more powerful, as they may include *member functions*:

```
struct date {
    int day;
    int month;
    int year;
    void display();
}
```

As we can see above, it only declares the member function `display()`, which means it will need to be defined elsewhere in the program. To display a date, we might call:

```
date d;
d.display();
```

12.3.2 Structs vs Classes

Structurally, structs are similar to classes (without constructors) and member functions is an alternative term for method. A C++ class is shown below:

```
class Date {
    int day;
    int month;
    int year;
public:
    Date (int, int, int);  // constructor
    void display();
}
```

Another difference between structs and classes in C++ is that, by default, members are public in structs and private in classes.

12.3.3 Whats the difference: Records, Structs & Classes

A Class is the concept of an Object Oriented language. It would be considered as an extension of records, given that procedural languages typically have records (containing fields), and object oriented languages have classes that contain fields (attributes) and methods (operators).

12.3.4 Variant Records

Records can be inflexible and memory-inefficient, because all values of a record type have a fixed set of fields. For example, if we are creating a record to store students in; containing name, registration number, graduation status; then if they've graduated also storing their year of graduation and if not, storing their course number and year of enrolment.

Depending on the implementation of this, it could result in wasted fields which must be present in each record even if they're not needed.

However, in such cases - Pascal provides a *variant record* to implement the alternative choices:

```
type student = record
    name: array [1..30] of char;
    reg_no: integer;
    case graduated: boolean of;
    true (year_of_grad: integer);
    false (course_no: integer;
           year_of_enrol: integer);
end;
```

In the above example, we used a `boolean` as the tag to determine which fields to include - other types (for example `byte`) can be used in circumstances where we need more than two cases.

12.3.5 Unions

C & C++ makes use of "Unions" instead of "Variant Records". Unions are designed for storing data items of multiple types in the same memory space. For example in C:

```
typedef enum (INTEGER, DOUBLE) MyType;

typedef struct {
    MyType a_type;
    union {
```

```
            int i;
            double d;
        } myUnion;
    } Value;
```

Then we can either use the integer value:

```
    Value v;
    v.a_type = INTEGER;
    v.myUnion.i = 5;
```

or we can use the double value:

```
    Value v;
    v.a_type = DOUBLE;
    v.myUnion.d = 5.4321;
```

Here, we see that union is the storage for either an integer or a double but not both at the same time.

12.3.6 Variant Records vs Union

The main difference between the C implementation and the Pascal version is that in the C version (Union), it is possible to have both options set at the same time. This not the case in Pascal's Variant Records.

Page 13

Lecture - Expression and Assignment

2024-04-15 14:00 Jiacheng

13.1 Expressions

An *expression* is a combination of values, variables, operators and function calls. Expressions perform operations on data and move data around. Expressions can be of the type:

- arithmetic (`4 * I + func(5)`)
- relational (`a >= b`)
- boolean (`A && B`)
- assignment (`i = 4`)

Some expressions are very important and some of them are instrumental for implementing various algorithms. To correctly evaluate the expressions we need to know the rules of precedence and rules of associativity of operators.

13.1.1 Arithmetic Expressions

Arithmetic evaluation was one of the motivations for the development of the first programming languages. Arithmetic expressions consist of:

- operators
- operands
- parentheses
- function calls

13.2 Operators

A *unary* operator has one operand. For example:

- The Negation operator "`-`", used as `-b` where `-` is the operator and `b` is the operand
- The *Integer Promotion* operator "`+`", used as `+b` converts `b` to `int` if it is a `short`, `byte` or `char`.

When the operators precede their operands, we say the operators are "prefix"; and when the operators follow follow their operands, we say the operators are "postfix". Unary operators can be prefix or postfix. For example the increment operator `++` can either be used as `i++` or `++i`. A Binary operator has two operands - for example `+`, `-`, `*`, `/`. In most languages, binary operators are "infix" which means they appear between the operands.

13.2.1 Operator Precedence Rules

The *operator precedence rules* define the order in which operations are evaluated within an expression. Typical precedence levels (from high to low) are shown below, with examples in brackets:

- Parentheses (`(` and `)`)
- Unary Operators (`++` and `--`)
- Unary Operators (`+` and `-`)
- Binary Operators (`*` and `/`)
- Binary Operators (`+` and `-`)

13.2.2 Operator Associativity Rules

The operator associativity is a property that determines how operators of the same precedence are grouped in the absence of parentheses. For example, in the expression `3/5*0`, operand `5` is preceded and followed by operators `/` and `*` (which both have equal precedence). Which operator to apply to the operand is determined by the associativity of the operators. The operators may be:

associative the operations can be grouped arbitrarily. For example `a * b * c` could be grouped as `(a * b) * c` or as `a * (b * c)`

left-associative the operations are grouped from the left

right-associative the operations are grouped from the right

If we consider the expression `a - b - c`. If the operator has left associativity, this expression would be interpreted as `(a - b) - c`. Whereas if the operator has right associativity, this expression would be interpreted as `a - (b - c)`.

However, it should be noted that some mathematical operators have inherent associativity. For example:

- Subtraction (`-`) and division (`/`) are inherently left-associative
- Addition (`+`) and multiplication (`*`), by contrast are both left and right associative.

Many programming language manuals (also known as their documentation), provide a table of operator precedence and associativity. Normally this would be from *left to right*. However there are some which are different - for example the Exponentiation operator is normally *right to left*.

13.2.3 Operator Overloading

Use of an operator for more than one purpose is called *operator overloading*. This can be seen in Java, for example, where `+` is used for `int` and `float` additions but it is also the operator for String concatenations. The semantics of the operator could be evident in program context.

C++ and C# allow user-defined overloaded operators. These, when sensibly used, can be an aid to readability (whereby they avoid method calls, and expressions appear natural).

Overloading could cause problems or difficulties. For example the ampersand (`&`) in C and C++ specify both bitwise AND (binary) operation and "address of" (unary) operations. This could lead to a compiler error detection being affected (for example missing the operand `x` in the expression `x&y`).

13.2.4 Compound Assignment Operators

Compound means combining assignment with arithmetic operators. For example, `+=`, `-=`, `*=`, `/=`, `<<=`, … (general form `operator=`). It is a shorthand method of specifying a commonly needed form of assignment.

Originally introduced in ALGOL; and then adopted by C and inherited by the C-derived languages. It means we can take an expression `a = a + b` and rewrite it as `a += b`, using the compound assignment operator `+=`.

As you might come to expect, there are some edge cases to these helpful operators. For example:

```
a[i++] *= 2;
a[i++] = a[i++] * 2;
```

In the first expression, `i++` (and therefore `a`) is evaluated once. Whereas in the second expression, `i++` (and therefore `a`) is evaluated twice:

- index of `i++` in `a[i++]` on the right hand side is evaluated
- value of `a[i++]` is retrieved; addition is performed
- `i++` in `a[i++]` on the left hand side is evaluated
- assignment operation is performed

This is bad, and whilst it will always execute to the same thing - it may not be what the programmer is expecting to happen.

13.2.5 Unary Assignment Operators

Unary assignment operators in C-based languages combine increment and decrement operations with assignment. For example the line `count++` would be equal to `count = count + 1`; and `-count++` would be equal to count getting incremented then negated (as `++` has a higher level of precedence than negation).

The location of the `++` in the expression is important:

- `sum = count++` means that `count` is incremented by 1, then assigned to `sum`
- `sum = ++count` means that `count` is assigned to `sum`, then incremented by 1

13.3 Expressions & Assignments

13.3.1 Side Effect of Expressions

We say that an expression has a side effect it also modifies variables or causes something else to happen, in addition to returning a value (the main aim of an expression). Side effects come about when an expression involves:

- an assignment
- increment / decrement operations
- function call
- method invocation

For example, a function might modify a global or static variable, or modify one of it's arguments.

13.3.2 Side Effects & Assignment

In the presence of side effects, a program's behavior may depend on (execution) history; that is, the order of evaluation matters. In programming, sometimes the side effect is exactly what we want to use, for example in an assignment.

13.3.3 Assignment as an Expression

In the C-derived languages (for example Java, Perl and JavaScript), the assignment operator (`=`) is treated in the same way as other binary operators (for example `+`). This means that assignment returns a value, but it has the side effect of changing its left operand (the effect of the assignment). For example in Java and C, the expression

```
x = y + 1;
```

returns a value (which equals the value assigned to the variable on the left, therefore equals `y+1`) and assigning the value to `x` is just a side effect.

13.3.4 Use / Abuse of Assignment as Expression

In the following statement

```
while ((ch = getchar())!= EOF){...}
```

where `ch = getchar()` is an expression (assignment) it returns a value as the result of evaluating the expression. In addition, it assigns a value to variable `ch`. Such uses of assignment may cause a loss of error detection, for example (in C), if we write:

```
if (x=y) ...
```

where `x=y` is an assignment and the returned numeric value form evaluation of assignment `x=y` could be treated as a boolean value; instead of:

```
if (x==y) ...
```

where `x==y` us a relational expression. This mistake is not detectable by the compiler.

It is also possible to use assignment as an expression to assign a value to more than one variable at once, for example

```
z = x = 1
```

When multiple assignment operators appear in an expression, they are evaluated from right-to-left (because `=` is right associative).

13.3.5 Type Conversions in Assignment

If we consider the following assignment:

```
aFloat := aInt
anInt := aFloat
```

We would need to convert types of values during it. Different languages take different approaches to the type compatibility between the left and right hand sides. For some languages, this has to be done explicitly. For example in Pascal and Java.

In Java - this is done using a *cast*. A *cast* refers to type conversion that is explicitly requested by the programmer. For example, in Java:

```
int a;
float b, c;
c = (float) a*b;
```

In other languages, for example C, the conversion is done automatically by the compiler by *coercion*. Coercion is an *implicit type conversion* that is initiated by the compiler. The compiler checks the type compatibility of the assignment and changes the type of the right side of the assignment, if needed.

13.3.6 Type Conversion in Arithmetic Expressions

There are two key conversion rules in arithmetic expressions:

Widening Conversions are conversions in which an object is converted to a type that allows any possible value of the original data (means that converting to a type takes a larger memory space). This can be seen in the `int` to `float` conversion. Widening Conversions are safe and are widely used.

Narrowing Conversions are conversions in which the data type is converted in the opposite direction to widening. It is usually done explicitly as a request from programmers, who feel such a conversion is necessary. The conversion is not safe because there is a chance that overflow may happen.

In arithmetic expressions, all numeric types are coerced using *widening conversions*, for example

```
int a;
float b, c;
...
c = b * a;
```

In the example above, `a` is widened to `float` before multiplication. Coercion weakens the type error detection ability of the compiler, for example if `a` is a typing error (it was supposed to be `c`), the compiler would not be able to detect it.

13.3.7 Relational Expressions

A *relational operator* is a binary operator that compares the values of its operands and the result is a boolean value, either `True` or `False`. Operator symbols vary somewhat among languages, for example the "not equal" operator could be one of these: `!=`,`/=`, `~=`, `.NE`, or `<>`.

JavaScript and PHP have two additional relational operators `===` and `!==`. Whilst similar to `==` and `!=`, they do not prevent the operands being coerced, as `==` would result in coercion:

- `"7" == 7` would evaluate to as `True`, where string "7" is coerced to number 7; whereas
- `"7" === 7` is `False`

13.3.8 Boolean Expressions

In *Boolean expressions* - both the operands and results are both Boolean. Boolean Operators include the following, note that every language uses a different symbol to represent them:

- OR (`||`)
- AND (`&&`)
- NOT (`!`)

C89 does not include a Boolean type, rather programmers are reduced to using `int` type with `0` for `False` and a nonzero value for `True`. A *quirk* of C's expressions:

```
a < b < c
```

is, shockingly, a legal expression. However the result is not what you would express at first glance. The *left* operator (`a < b`) is evaluated, producing 0 or 1, which is then treated as a *numeric value* (rather than a boolean, as apparently this makes sense) and is then compared with the third operand (`c`). [1]

13.4 Notation

In the absence of associativity and precedence rules, the *infix notation is inherently ambiguous.* For example the following expression has two possible values:

```
a = 3
b = 4
c = 5
print a + b * c
```

The value could either be `35` if evaluated from left to right and `23` if evaluated right to left.

So here we have a problem - infix notation isn't actually perfect and we need to consider using alternative notation styles. There are three more notations we will explore in this lecture.

13.4.1 Prefix Notation

In this notation, the operators appear before their operands. For example, the usual mathematical expression `a+b-c*d` would be written in prefix form as `-+ab*cd`. then it would be evaluated from right to left by combining the two operands with the operator in front of them (we will use parenthesis to show the order of evaluation)

- `- (+ab) * cd` then
- `- (+ab) (*cd)` then
- `(- (+ab) (*cd))`

As we can see above, prefix notation is inherently unambiguous.

13.4.2 Postfix Notation

In postfix notation, operators appear after their operands. Using postfix notation, the expression `a+b-c*d` would be expressed as `ab+cd*-` or `abcd*-+`. It is also evaluated from right to left but by combining the two operands with the operator after them:

- `(ab+) (cd*) -` then
- `((ab+)(cd*)-)`

Postscript (a Printer Control Language from Adobe) uses this notation.

13.4.3 Cambridge Prefix Notation

Cambridge notation introduces parentheses into prefix notation. For example `a+b-c*d` would be written as `(-(+ab) (*cd))`. An advantage of this is that it makes operators like `+` and `-` become n-nary (for example `a+b+c+d` would be written as `(+abcd)`). Lisp uses this notation.

[1] I lost braincells understanding this.

13.5 Short Circuit Evaluation

Short Circuit Evaluation is a way of expression evaluation in which the result is determined without evaluating all the operands and / or operators. For example:

```
(13 * a) * (13 / b - 1)
```

If `a` is 0, the whole expression evaluates to zero therefore there is no need to evaluate the second half. However, such a case is difficult to detect, which means that short circuit evaluation like this is never taken.

We can also see this in the following expression:

```
(a >= 0) && (b < 10)
```

The value of the Boolean expression is independent of the second relational expression if `(a<0)` is `False` because `False & (b<0)` is `False` for all values of `b`. Therefore, there is no need to evaluate `(b<0)`. Unlike the case of arithmetic expressions, this shortcut can be easily discovered and taken during execution.

Many programming languages provide both standard and short-circuit versions of the boolean operators. For example in C++, C#, Java, Matlab, etc, provide both:

- `&`, `|` (standard) and
- `&&`, `||` (short-circuit version)

However, some languages do not; for example C, Python, Lisp. Some allow compilers to choose between standard and short-circuit evaluation; for example, Fortran operator `.and.`, `.or.` are neither short-circuit or standard.

13.5.1 Use of Short Circuit Evaluation

If we consider a Java loop for searching for an item (`valueToFind`) in a `list`:

```
index = 0;
while (index < length) && (list[index] != valueToFind)
    index++;
```

Without the use of short-circuit, the right hand side of the program will always be executed. This will cause the program with a *subscript out of range* exception, if the searched item is not in the list.

13.5.2 Potential Problems

Short-Circuit evaluation of Boolean expressions allows subtle errors to occur, for example:

```
(a>b || b++>3)
```

In this expression, `b++>3` is evaluated (therefore `b` changes) only when `a>b` is `False`. If the programmer assumes that `b++>3` is evaluated every time (therefore treating it as standard evaluation), the program will fail. In short-circuit evaluation, the order of the expression being typed by the programmer is important. It does not allow the compiler to reorder and prune expressions as it sees fit.

Page 14

Lecture - Statement-Level Control Structures

2024-04-19 14:00 Jiacheng

14.1 Levels of Flow Control

Flow Control in programs, or execution sequence, makes it possible to implement algorithms to do various things. The flow control of programs can be explored at different levels:

- Within Expression - governed by associativity and precedence rules
- Program Statements
- Program Units

Statements that provide capabilities such as selecting among control flow paths or causing the repeated execution of certain collection of statements are called control statements.

Control Flow comes form the early days of programming - where they would use statements, such as `goto`, which sent the program to a different line to execute. This would result in programs which are hard to read and maintain.

14.2 Programming Theorem

Structured Programming Theorem states that all algorithms which can be expressed by flowcharts can be coded in programming languages that are equipped with two control statements:

- selection (choosing from two, or more, control flow paths)
- pre-test logical loop, or iterative statement (the condition is tested before the body of the loop is executed)

These two control structures are necessary for any imperative programming language. In programming language design & implementations, variations of these two types of control statements are often provided.

14.3 Selection Statements

Selection statements choose between two or more execution paths in a program. The two-way variety select between one of two execution paths; and the multiple-selection statements select one of many execution paths.

14.3.1 Two-Way Selection

The general form (syntax) of a two-way selector is as follows:

```
if control_expression
    then
        do something
    else
        do something
```

The control expression, within contemporary languages, is a boolean expression which is evaluated. Older languages did not support a boolean type and therefore would use arithmetic expressions, some languages still support this. The expressions within the `then` or `else` blocks could either be single statements or compound statements.

Many languages use braces to form compound statements (blocks):

```
if (control_expression) then{
    ...
} else{
    ...
}
```

However in true quirky fashion, Python uses indentation and a colon:

```
if control_expression:
    ...
else:
    ...
```

14.3.2 Nesting Selectors

Selector nesting is supporting and used. But there should be no ambiguity in the semantics of nesting selectors. For example:

```
if (control_expression){
    if (another_control_expression){
        ...
    } else{
        ...
    }
} else{
    ...
}
```

14.3.3 Shorthand Selection

A full selection statement can be quite big and obtuse - especially in the case that it's controlling part of an output to the user. Conveniently, there exists shorthand selectors which enable more succinct code. For example, the normal Java expression:

```
int x;
if (expression) {
    x = -1;
} else {
    x = 1;
}
```

Can be condensed to:

```
int x = (expression) ? -1 : 1;
```

14.3.4 Multiple-Way Selection Statements

Multiple-Way Selection Statements allow the selection of one of any number of statements or statement groups. In C, C++, Java and JavaScript - it takes the form:

```
switch (expression){
    case const_expr:
        statement;
        break;
        ...
    default:
        statement;
}
```

In most languages the control expression can be a character, integer, string or enumerated type. Once the case is matched, the statements starting from the matched case are executed until a `break` statement or until the end of the switch statement is reached (the latter being referred to as fall-through behaviour). C# does not implement fall-through behaviour; it requires that every segment must end with an explicit unconditional branch statement (`break`, `goto`, or `return`).

The question of why you would want to fall-through is an interesting one. It is occasionally useful to allow control to flow from one selectable segment to another. For example, where the same code would be executed for multiple cases as using fall-through would minimise repeated code.

14.4 Iterative Statements

Iterative Statements are a mechanism for the repeated execution of a statement or compound statement. It is implemented either by various loops (iteration) or recursion. Common loop would either be the `for` loop (which is counter controlled) or `while` loop (which is logically controlled).

14.4.1 For Loop

In C-derived languages, for loops take the form:

```
for (init; loopControl; stepSize){
    loopBody
}
```

The `init` value is the initialisation value for the loop and is executed once at the start of the loops execution. The second expression (`loopControl`) is the loop control and is evaluated before every execution of the loop body; it produces a Boolean value as the condition to continue or stop. The last expression (`stepSize`) is the control for how big step sizes to take, and is executed after each execution of the body.

Different to Java and C#, C and C++ allow for a non-numeric `loopControl`. To them, a zero means false and all non-zero values mean true. Therefore if the value of `loopControl` is zero, the for loop is terminated, otherwise the loop body is executed.

In the event that `loopControl` is absent, the loop will be an infinite loop.

14.4.2 Logically Controlled Loops

Logically controlled loops come in two forms:

Pre-Test where the loop condition is tested before execution of the loop body In most languages this is the `while` loop however in Pascal, this is the `while ... do` loop

Post-Test where the loop body is executed before the condition is tested, meaning that the loop body is always executed at least one time. In most languages, this is the `do ... while` loop however in Pascal, this is the `repeat ... until` loop.

All for loops can be rewritten as while loops.

C and C++ have both pre-test and post-test forms, in which the control expression can be arithmetic. Java has both pre-test and post-test forms, however the control expression bust be boolean.

14.4.3 Unconditional Branching

Unconditional Branching statements are statements that chance control flow without any condition. They include `break`, `continue` and `return`. They are usually used within loop / iteration structures:

`break` provides unconditional exits from loops

`continue` skips the remainder of the current iteration but does not exit the loop

`return` terminates the function / method call

Some of these unconditional branching statement come with labelled and unlabelled varieties:

- Unlabelled `break` - terminates the innermost 'breakable' statement when nested. Can break: `switch`, `for`, `while` or `do-while`
- Labelled `break` terminates the labelled statement / loop and then program control flow is transferred to the statement immediately following the labelled (and now terminated) statement
- Unlabelled `continue` statement terminates the current iteration of the innermost loop and skips to the end of loop, then evaluates the boolean expression that controls the loop.
- Labelled `continue` skips the current iteration of a loop marked with the given label

The `return` statement, in addition to terminating function calls, can have a return value. Pascal does not have a return statement.

Page 15

Lecture - Subprograms and Parameter Passing

2024-04-26 14:00 Jiacheng

15.1 Subprograms

A common technique used to manage complexity of a larger problem is to decompose the problem into a series of smaller sub-problems. These smaller problems are easier to solve; however it is necessary that the programming language supports decomposition.

The fundamental concept, provided by all modern programming languages is the subprogram, procedure or function.

A *subprogram* is a piece of code that is identified by a name, is given a local reference environment of its own, and can exchange information with the rest of the code using parameters. Subprograms come in two flavours - procedures and functions.

Both *procedures* and *functions* are types of subprogram. Functions normally have a return value however procedures do not. They both accept parameters and perform code operations on those parameter values. During this lecture - we will only consider functions, however the concepts are also applicable to procedures.

15.2 Functions & Abstraction

Functions provide one of the two fundamental abstraction facilities of programming languages:

Process / Control abstraction in which programmers can hide procedural detail and only be concerned with a procedures interface, rather than its implementation

Data abstratction in which sophisticated data types can be used without knowing how they are implemented. This separates concerns and promotes highly reusable code as well as making code more maintainable. This will be covered further in a different lecture

An example of a function is shown below:

```
int foo (int n, int a){
    int temp = a;
    if (temp == 0){
        return n;
    } else{
        return n + 1;
    }
```

```
}
...
int x;
x = foo(3, 0);
x = foo(x+1, 1);
```

The definition of the function refers to the actual implementation of the function. It describes the interface (header) and the actions of the function (body). The header compeises of the name, return type and the formal parameters of a function. The parameter profile of a function is the number, order and types of its parameters, for example:

```
(int n, int a)
```

A function call is an explicit request that the function be executed. For example:

```
y = foo(3, 0);
```

15.2.1 Function Declarations

A function definition (known as a *prototype* in C and C++) provides the header but not the body of a function. The parameter names can be omitted and it could be saved in a separate file, which contains only the declaration of the functions.

The function definition comes after the declaration of the function.

15.2.2 C & C++ Prototypes and Header Files

Prototypes are often written in a separate header file which can be included in other C source files that wish to use the functions. For example, the header file `stdio.h` contains prototypes for the standard input & output files `scanf` and `printf`. The actual source code is stored in a library elsewhere on the computer, however the header file says how to interface with the functions.

Parameters & Return Values

A parameter is a way in which data can be provided to the function. There are a number of different types:

formal parameters are dummy variables listed in the function header and used in the function body

actual parameters represent a value or address used in the function call statement

Functions use *return values* to return values to the calling program at the end of the function's execution. This calling program, which called the function, is suspended during execution of the called function. Control always return to the caller when function's execution terminates. Note that the calling program could be another function.

15.2.3 Parameter Binding

There are two ways to bind the actual parameters to the formal ones:

By Position means that the binding of actual parameters to formal parameters is by their order of appearance, therefore the first parameter is bound to the first formal parameter and so forth. This is safe and effective

By Keyword means that the names of the formal and actual parameters are used. This means that parameters can appear in any order, thereby avoiding parameter correspondence errors; however the user must remember the formal parameters names.

An example of by position is shown below:

```
x = foo(3, 0);
```

An example of By Keyword is shown below:

```
func(value=my_value, array=my_array)
func(array=my_array, value=my_value)
```

15.2.4 Local Referencing Environment

When a function is called, a local referencing environment which includes all local variables is created. In most contemporary languages, local variables are created on the stack and are stack-dynamic (dynamic meaning the variables are created on demand, and stack meaning there is a natural implementation of recursion).

Local variables, however, can be static. This is especially helpful when their values are to be shared between different calls. For example, in C and C++, locals are by default stack-dynamic however cna be declared `static`.

15.3 Parameter Passing Methods

The *parameter passing method* is the method in which parameters are transmitted to and / or from the called subprograms. The relationship between the actual parameters is characterised by one of three semantic models:

- formal parameters receive data from the corresponding actual parameters;
- formal parameters transmit data to the actual parameters;
- do both

These models are called: in mode, out mode and inout mode, respectively. Further information on these models is below:

In Mode is implemented as pass by value. This is where the values are copied to the formal parameters

Out Mode is implemented as pass by result. This is where the values are copied to the actual parameters

Inout Mode is implemented as pass by value and result, which is a combination of pass by value (on function entry at the moment when functions are called) and pass by result (on termination of functions); or implemented as pass by reference where an access path (for example, an address) is transmitted to a formal parameter.

15.3.1 Pass by Value

Pass by Value works by transmitting the value when the function is called, and the value of the actual parameter is used to initialise the corresponding formal parameter. For example:

```
void foo (int x){
    x = x + 1;
}
int y = 1;
foo(y + 1);
```

The above code works by, during the execution of `foo`, local variable `x` assumes the initial value `2` by the effect of passing the parameter. It is then increased to `3` (by the operations of the function body) and is finally destroyed on termination of the function call (execution control returns to the caller), therefore the value of `3` is lost.

Through this method, we take a copy of the values of the parameters which separates the actual parameters from the formal parameters which could be modified in the function; meaning that the original values (in the calling program) are not altered during the function execution. It is simple and fast for transmitting a few scalar (elementary types) values. Pass by Value is the default mechanism in many languages, for example C, C++, Java and pascal.

However, additional storage is required (as the parameter values end up being stored twice) and the actual move of data can be costly (for parameters which contain a large amount of data).

15.3.2 Pass by Result

Pass by Result works by transmitting the value of the parameters back to the calling program when execution terminates. Note that no value is transmitted from the caller to the function.

During the execution of the function, the formal parameters acts as local variables of the function and their values are copied to the real parameters when control is returned to the caller. An example of this is shown below:

```
void foo (result int x){
    x=8
}
...
int y = 1;
foo(y);
```

After the function call, the value of `y` becomes `8`. When calling such functions in proper code, you shouldn't use actual values as the parameters to call them.

A major limitation to this solution is that the actual parameters must be variables. However, there is a potential problem to this solution. In that the following C# method specifies the pass by result by putting `out` specifiers on it's formal and actual parameters. At the end of execution of the method, if `x` is copied to `a` first, then the value of `a` will be `30`. If `y` is copied first, then `a` will be `15`. This shows that the value of `a` depends on the order of copying, which varies from language to language and implementation to implementation.

```
void myMethod (out int x, out int y){
    x = 15;
    y = 30;
}
...
int a = 0;
someObj.myMethod(out a, out a);
```

15.3.3 Pass by Value-Result

Unsurprisingly, as the name would suggest, Pass by Value-Result is a combination of pass by value and pass by result. It is sometimes known as pass-by-copy, and an example of it can be seen below:

```
void foo (valueresult int x) {
    x = x + 1;
}
```

```
...
int y = 8;
foo(y);
```

At the point of call, the value of actual parameters are copied to and stored in the formal parameters; and at the end of the call, the values of formal parameters are copied to the actual parameters.

The pass by value-result method facilities bi-directional data exchange, however isolates the formal parameters from the actual parameters.

15.3.4 Mixture Mode of Parameter Passing

In a function, multiple different parameter passing mechanisms may be used depending upon languages and implementations. For example, in Ada:

```
Procedure ADDER (A: in out INTEGER;
                 B: in INTEGER;
                 C: out FLOAT;)
```

15.3.5 Pass by Reference

In Pass by Reference, instead of passing values, access paths are passed. This is also called pass-by-sharing (referring to sharing a memory address between the calling program and the function). In this mode, the formal parameter and it's corresponding actual parameters are both referring to the same variable (and therefore the same memory address), even if they use different names.

This allows for bi-directional data exchange, whereby each modification of the formal parameter is a modification of the actual parameter. This can be seen in the following example:

```
void foo (reference int x){
    x = x + 1
}
...
int y = 0;
foo(y);
```

During the execution of `foo`, `x` is a name for `y`. After the call `y` becomes `1`.

An advantage of this methodology is that the passing process is efficient, in that there is no copying and no duplicated storage. However, there is no separation between formal and actual parameter; and the accessed formal parameters might be slower (when compared to pass-by-value, where local variables are used).

Pass-by-reference is a low-level operation (for example only an address needs to be stored). It has been excluded from some modern languages - for example Java, C and C++. In these languages, some forms of bidirectional communication between the caller and callee can still be achieved.

Page 16

Lecture - Abstract Data Types & Concurrency

2024-04-29 1400 Jiacheng

16.1 Abstraction

Abstraction is the concept of separating the top-level usefulness of a given thing from the details of its implementation; thus freeing the programmer from thinking about its details. For example, after abstraction - you may only be concerned about what the function can do, and how to interact with it, but you don't care about how it does it's job.

The idea of abstraction is to promote readability, maintainability, reusability, and security of software.

Modern programming languages provide two types of mechanisms for abstraction: a mechanisms for data abstraction; and a mechanism for control / process abstraction via subprogram functions or procedures. We saw the later in the previous lecture, where it is necessary for achieving program modularity, which means dividing up a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system.

This lecture, will look at *abstraction as a mechanism for data abstraction.* Nearly all programming languages designed sing 1980 support data abstraction. Some achieve abstraction by borrowing the OO concepts (ie constructors, accessors, and mutators) for example OO Python, JavaScript, C#, etc. However some languages provide data abstraction through classes / interfaces such as Java and C++.

16.1.1 Data Abstractions

The use of *data abstraction* enforces a clear separation between the abstract properties (the uses of data) or a data type and the actual details of its implementation. This means that the abstract properties are visible to the client code, that uses the data type, however the actual implementation is kept entirely private and can change.

Data abstraction is achieved by defining *abstract data types* (ADTs), which are user-defined data types that consist of:

- a set of data
- necessary operations on the data

For example, we could define an ADT called `lookupTable` which uniquely associates `keys` with `values`; and the values may be retrieved specifying their corresponding keys. The lookup table may be implemented in a number of different ways; for example as a: hash table, a binary search tree, or a linear list of key-value pairs. As far as the client code is concerned, the abstract properties aof the type are the same in each case.

16.2 ADTs and Data Structures

An ADT often implements a data structure, which is a representation of data and the operations allowed on the data. For example, a List is a widely used data structure containing:

data a collection of items, each item has a position in a sequential order

operations add / remove an item, sort the items, etc.

An interface is an agreement / contract between two parties:

- the *implementer* of the ADT, who is concerned with making the operations correct and efficient
- the *applications programmer* who just wants to use the ADT to get a job done

It doesn't matter if you are both of these parties (ie, you have built an ADT for personal use), the contract is sill essential for good quality code. The separation of concerns is essential in any large project.

16.2.1 Support for ADTs

Many high-level languages (C++, Java, C#, Ada) provide encapsulation mechanisms that support data abstraction and provide built-in ADTs. This is seen as the Standard Template Library (STL) in C++; or in Java as it's interfaces and implementations in the standard library.

These libraries provide many widely used ADTs, for example:

stacks access to the most recent item in the collection

queues access to the least recent item

16.2.2 Build Your Own ADT

It is, of course, possible to build your own ADT - the first step of this process is designing the ADT.

Designing an ADT involves choosing a set of core operations for users to use the data type. The core operations support the basic uses of the data. For example, every ADT should provide a way to:

- add an item
- remove an item
- find, retrieve, or access an item

As well as many more non-essential but useful operations:

- check if a collection is empty
- make a collection empty
- retrieve a sub-set of th collection

The next stage of building your own ADT is to implement the ADT. To implement the ADT, you need to choose a *data representation* and *algorithms*. Data Representation refers to the way in which the data is stored - usually through an internal storage container. The users do not need to know this representation, and they should not be able to tamper with the representation. For this reason, data is generally made private; then users use accessors and mutators to access the data.

16.3 Concurrency

Concurrency is the existence of multiple simultaneously active execution contexts.

Concurrency can occur at a number of different levels:

- Machine instruction level (via processor design)
- Statement level: statements of high-level languages that support parallel computing
- Subprogram level: a single program has multiple concurrently running subprograms/threads on a single computer
- Program level, where either: multiple programs run concurrently on a single computer; or a program / application runs on multiple (networked) computers or on multi-cores of high performance computers.

Each of these different levels have different names and behave slightly differently.

At program level, when concurrency exists between multiple programs on a single computer, we call it multitasking (e.g. the way most operating systems control multiple tasks). However, whenever a program implements concurrency on multiple computers, we call it a distributed program - a program of which different pieces are on computers connected by a network.

At subprogram level, when a program implements concurrency on a single computer, we call the program concurrent / parallel program. This program is said to be *multithreaded*. The rest of this lecture will focus on concurrency at subprogram level.

16.4 Subprogram-Level Concurrency

A task, process or thread of a program is a program unit that can be in concurrent execution with other program units. A task / processes / thread differs from ordinary subprograms in that:

- A process / task is implicitly stated
- When a multithreaded program starts the execution of a thread, the rest of the program is not necessarily suspended
- When a thread's execution is completed, control may not return to the caller

A task / thread is *disjoint* if it does not communicate with (or affect the execution of) any other tasks in the program; otherwise it is a joint task and needs to be synchronised. Inter-thread communication is necessary for joint tasks, because a thread may need either exclusive access to some resource or to exchange data with another thread. If not synchronised, joint tasks can lead to race conditions or deadlocks.

16.4.1 Race Conditions

A race condition occurs when the resulting value of a variable (or the status of an attribute) depends on the execution order of two or more threads. Therefore, depending on the timing order of the instructions, there could be different results.

16.4.2 Deadlock

A deadlock is a situation in which two or more competing threads are each waiting for the other to finish while holding resources that the other needs, and therefore neither ever finishes.

For deadlock to occur, all of the following conditions must hold simultaneously in a system:

mutual exclusion where resources may only be under mutual exclusion - only one process can use the resource at any time

hold and wait where a process is allowed to hold resources wile waiting for others which are being held by other processes

no preemption where resources cannot be forcibly removed once it is granted to a process

circular wait where a circular chain of threads exists in which each thread holds a resource needed by the next thread in the chain

16.4.3 Task Synchronisation

To remove race conditions and deadlocks, the order of execution of the tasks need to be controlled (synchronised). Synchronisation requires communication between the tasks, which can be provided by:

- shared non-local variables
- message passing
- special data types (semaphores & monitors)

Synchronisation can be cooperative, or competitive, depending upon the nature of the resource that must be shared.

16.4.3.1 Cooperative Synchronisation

Cooperative Synchronisation ensures two (or more) tasks work in a cooperative way to avoid deadlock. This means that one task must wait for the other task to finish execution before it may proceed; and that there are more than one unit of resources to be shared.

This is best illustrated by the producer-consumer problem where one task produces something that the other task consumes.

16.4.3.2 Competitive Synchronisation

Competitive Synchronisation ensures the exclusive access to critical resources by a single task at any one time - to avoid race conditions.

16.4.4 Synchronisation Implementation

Synchronisation can be implemented using special data structures, called *semaphores* and *monitors*. Semaphores can be used to implement both cooperative and competitive synchronisations; while Monitors wrap the resources with the mechanism for exclusive access control and implement competitive synchronisation.

16.4.5 Semaphores

A semaphore is a data structure consisting of:

- a counter - which is an integer variable, sometimes known as the value of the semaphore and is used for keeping track of the units of available resources (for example, this might start at 10)
- a queue - which is used for storing the processes that request access to the resources
- two operations - which are used by processes
  - wait (or P) - which is used by a process to indicate that it would like access to the shared resource

 - signal (or release or V) - which is used by a process to indicate that it has finished with the shared process

When a process needs access to the semaphore-guarded resource, it must call the `wait()` method. When this is called, the method first decrements the value of the counter variable by 1. After this decrement - if the counter value is non-negative (ie ≥ 0), then the calling process is granted access to the resource and put on the ready list for the CPU to execute; however, if the counter value becomes negative, the calling process / thread is added to the semaphore's waiting queue (we then say that "the thread is blocked").

When a process finishes using the resource, it must release the resource by calling the `signal()` method. When called, the `signal()` method increments the value of the semaphore's counter variable by 1. After the increment, where the counter value is equal to or less than 0, from which we know that there are threads on the waiting queue - it transfers a blocked thread from the semaphore's waiting queue to the ready list.

16.4.6 Monitors

A *monitor* is an object that encapsulates the shared data / resources and guarantees that at any point in time - at most one thread may access the shared data. This is the monitor's mutual exclusion property.

Monitors implement competition synchronisation.

Mutual exclusion can be achieved by equipping an object (a resource) with a private lock. The lock, initially open (or unlocked), is locked at the start of each public call of the object and is unlocked on the return from each public method call.

Monitors are used for synchronisation control by Java, C#, concurrent Pascal, Python, Ada, etc.