Hugh Baldwin
up2157117

Programming Applications and Programming Languages

M30235
TB1&2

University of Portsmouth
BSc Computer Science
2nd Year

Contents

Lecture - Introduction

14:00 22/01/24 Jiacheng Tan

- No content from TB1 is assessed as part of TB2

Lecture - Introduction to Programming Languages

14:00 22/01/24 Jiacheng Tan

Since there are many different types of application, there are also many types of programming language. The main programming domains are as follows

- Scientific (e.g. ForTran)
- Business (e.g. COBOL)
- AI (e.g. LISP)
- Systems Programming (e.g. C, C++)
- Web Software (e.g. HTML, JavaScript)

Language Categories

There are several ways to categorise programming languages, such as by uses, paradigms, abstraction level, etc

Machine Languages

- Machine languages directly run on the hardware, using the instruction set of the processor
- Machine code is usually written in hexadecimal as this is a more efficient way of displaying the binary which represent the instructions
- It is very hard for programmers to directly write machine code, as it is not easy to remember instructions and it lacks features such as jump targets, subroutines, etc

Assembly Languages

- A slight abstraction over machine languages
- Each instruction is replaced with an alphanumeric symbol which is easier for programmers to remember and understand
- They also include features such as subroutines, jump targets, etc which make it much easier to create complex programs

System Programming Languages

- More abstracted from machine languages, but you are still concerned with low-level functions such as memory management
- Used to create operating systems, and for embedded applications where low system requirements do not allow the use of high-level languages

High Level Languages

- Languages that are machine-independent (are not written directly in machine code, and are therefore portable between CPU architectures)
- Need to be compiled or otherwise translated from text to machine code before they can be run

Scripting Languages

- Used to create programs which perform a single, simple task
- These are used for system administration
- Usually interpreted languages
- More akin to pseudocode than other programming languages

Domain-Specific Languages

- Some languages are designed to perform a specific task much more efficiently
- The specific purpose could be just about anything, but are specific to that task and either cannot be used otherwise or are not well suited for it

Programming Paradigms

There are several different paradigms which are used in programming

- Procedural
  - Most programming languages are procedural
  - A program is made up of one or more routines which are run in a specific order
- Functional
  - Applies mathematical functions to inputs to get a result
  - Useful for data processing applications such as data analysis or big data
- Logical

There are also two major types of programming languages, which are designed for different purposes

- Imperative Languages
  - Programs are defined as a sequence of commands for the computer to perform
  - Like a recipe for exactly how to get the desired output
- Declarative Languages
  - Programs describe the desired results without actually specifying how the program should complete the task
  - Functional and logical programming languages are examples of this

Lecture - Implementation and Compilation

14:00 02/02/24 Jiacheng Tan

There are 3 main methods of implementing a language:

- Compilation - Programs are translated into machine language, either before (Compilation) or during (JIT) execution
- Pure Interpretation - Programs are interpreted by another program, known as an interpreter
- Hybrid Interpretation - A compromise between the two, code is compiled into an intermediary language, which is then interpreted with a Language Virtual Machine

Compilation

- High-level code is translated into machine code for a specific platform
- This results in slow translation, but much faster execution
- The compilation process has multiple stages
  - Lexical Analysis - Converts characters in the source into lexical units
  - Syntax Analysis - Transforms lexical units into parse trees which represent the syntactic structure of the program
  - Semantics Analysis - Generate intermediary code
  - Code Generation - Intermediary code is translated into platform-specific machine code
- The program which completes this process is known as the Compiler
- During this process, the compiler uses a ”Symbol Table”, which each stage interacts with

Lexical Analysis

The scanner reads the source code one character at a time and returns a sequence of tokens which are sent to the next phase. Tokens are symbolic names for elements of the source language. An example of a token in C++ is the keyword ‘void‘, which is a type definition, another example is ‘;‘, which delimits the end of a statement. Each token is also stored in the symbol table, along with it’s attributes.

Symbol Table

The symbol table stores all of the identifiers of a source program, along with their attributes. These attributes include information such as the type of a variable, the size or length of a string or array, the arguments to be used with a function and the types of each argument, etc.

Syntax Analysis or Parsing

The parser analysis the structure of the source code. The parser takes the output of the lexical analyser as a sequence of tokens. It attempts to apply the syntactic rules (or grammar) of the language to the sequence of tokens. The parser uses the language's grammar to derive a parse tree for each statement. Parsers usually construct Abstract Syntax Trees (ASTs), which are slightly simpler and easier to represent with a computer, but which still represent the same syntax. If the syntax tree is invalid for the language's grammar, a syntax error is generated and the compilation process stops

Semantic Analysis

The semantic analyser catches any other issues that are still valid syntax. For example, if you attempt to add a string to a float, it could still be syntactically correct, but semantically makes no sense and is not possible to compute. It is also able to find issues with the variable types of function arguments, such as attempting to use a string in the place of an integer or float.

Code Generation and Optimisation

The code optimiser attempts to improve the time and space efficiency of the program. It can do this in several ways, such as simplifying constants (e.g. replacing 10 * 10 with 100), removing unreachable code, optimising the flow of code, etc.

The final task of the compiler is to generate the final output code. This could be in the form of platform-specific machine code, or intermediary code for use with a virtual machine. This stage also deals with scheduling and assigning registers for use during execution

Pure Interpretation

- High level code is directly executed by another program known as the interpreter
- There is no syntax or semantics analysis, and there is no optimisation
- Only really suitable for small, non-real-time applications
- It also often requires more space as it needs to store the symbol table during execution
- Very few modern languages use interpreters, other than Python, JavaScript and PHP

Hybrid Interpretation

- A compromise between compilers and pure interpreters
- High-level languages are translated or compiled into an intermediary language, using the same compilation steps as before
- The intermediary code is then run by a platform-specific virtual machine, which interprets the code into machine language

Just-in-Time

- Programs are initially translated into an intermediary language
- This is then loaded into memory and segments of the program are then translated into machine code just before execution
- The machine code is then kept in case the function is called again somewhere else in the program
- This drastically improves the execution speed as compared to pure interpretation, but is still slower and typically less space and memory efficient than a compiled program

Lecture - Regular Expressions

14:00 05/02/24 Jiacheng Tan

The full definition of a language includes definitions of it's lexical structures, syntax and semantics. The lexical structures of a language are the form and structure of the individual symbols, such as keywords, identifiers, etc. The syntax determines the structure of the language, such as how a statement is defined, how to structure an expression, and so on. The semantics of a language determine how you can use each operator, what types they support, checking for type consistency in strongly typed languages, etc. The semantics of a language also define it's "grammar", which is how the compiler enforces the semantics.

Language Analysis

The implementations of a language must analyse the lexical and syntactic structure of the source code to determine if it is valid or not. This is usually implemented using two separate systems, the lexical analyser and syntax analyser. If the analyser is implemented using regex, it is a finite automaton, based on a regular grammar (that of the language)

Lexical Analysis

A lexical analyser reads the source code one character at a time and outputs a list of tokens to the next stage of the compiler. These tokens are made up of smaller substrings of source code, known as lexemes. Each lexeme matches a character pattern from the language's grammar.

The lexical analyser can be implemented in several ways, but the most common are by using regular expressions (Regex), or a deterministic finite automata (DFA).

Definitions

- The Alphabet
  - Each language has it's own alphabet, which is the set of all characters which could be used in a lexeme
  - An alphabet is usually represented using Σ
- String or Word
  - A string or word *over* an alphabet is a finite string of symbols from the alphabet
  - The length of a string is the number of symbols which make up the string
  - An empty or null string is denoted by ε, and so $\mid \varepsilon \mid = 0$
  - The set of all strings over Σ is denoted by $\Sigma*$.
  - For a symbol or string x, x^{n} represents a string of that symbol, n times, e.g. $\mathrm{a}^4 = \mathrm{aaaa}$

Regular Expressions

Regular expressions specify patterns which can be used to match strings of symbols. A regular expression, r matches or is matched by a set of strings if the strings conform to r's pattern. The set of strings matched

by r is denoted by $\mathrm{L}(\mathrm{r}) \subseteq \Sigma*$, i.e. all strings which are over the alphabet Σ. This is known as the language generated by r.

$\emptyset$ is in and of itself a regular expression, but does not match any strings at all, and is only very rarely useful. ε is also a regular expression, which matches only the empty string ε.

Since ε is an empty string, it can be used as the identity element for concatenation, and as such, $\varepsilon + \mathrm{s} = \mathrm{s} + \varepsilon = \mathrm{s}$.

For each symbol where $\mathrm{c} \in \Sigma$, c is a regular expression over Σ. In this case, the expression only matches a single instance of the symbol.

If r and s are both regular expressions, then $\mathrm{r} \mid \mathrm{s}$ is also a regular expression. $\mathrm{a} \mid \mathrm{b}$ would match a single instance of either a or b. $\mathrm{a} \mid \varepsilon$ would match a single instance of a or ε.

If r and s are both regular expressions, then rs is also a regular expression. This would match a single instance of the string rs, as the string would have to match both the regular expression r **and** s. As with arithmetic expressions, brackets can be used to make the meaning of a regular expression clearer. e.g. $(\mathrm{a} \mid \mathrm{b})\mathrm{a}$ matches the strings aa and ba

If r is a regular expression, then $\mathrm{r}*$ would match any number of rs in a row. Specifically, it means zero or more instances of r. $\mathrm{r}+$ would match one or more instances of r, which could also be written as $\mathrm{rr}*$.

As with arithmetic expressions, there is a specific order of operations which the symbols must follow. The order is as follows: (), * or +, concatenation, |. This is similar to arithmetic as anything inside parentheses must be processed before everything else.

A regular definition is a named regular expression, which can be used to make up more complex regular expressions, without re-writing the same expression several times. For example, you might define number as $\mathrm{number} = 0 \mid \cdots \mid 9$

Regular Expressions for Lexical Analysis

Regular expressions provide a method to describe the patterns which make up the lexical structure of a language, as well as restricting the alphabet which can be used to write source code. In most cases, languages use a standard alphabet, such as ASCII or UTF-8. An example of a regular expression used in a typical language could be if for the token of IF, ; for a semicolon, $(0 \mid \cdots \mid 9)+$ for a number, etc.

Languages are sets of strings chosen from some alphabet Σ. More formally, a language L over an alphabet Σ, $\mathrm{L} \subseteq \Sigma*$.

Given a language L over some alphabet Σ, it is necessary to be able to write an algorithm which takes any input string $\mathrm{w} \in \Sigma*$, and outputs True if $\mathrm{w} \in \mathrm{L}$ and False if $\mathrm{w} \notin \mathrm{L}$. This algorithm is known as a decision procedure for L. A decision procedure can be written using either a Deterministic Finite Automaton (DFA) or a Non-deterministic Finite Automaton (NFA). Any language which can be denoted by a regular expression is known as a regular language.