BSc (Hons) Computer Science

University of Portsmouth

Second Year

Programming Applications and Programming Languages

M30235

Semester 1&2

Hugh Baldwin

hugh.baldwin@myport.ac.uk

Contents

Part I

Teaching Block I

The first teaching block is 100% coursework, which makes up 50% of the overall grade. While there are lectures, they are informal, mostly teaching the basic concepts of Flutter, which is very practical, and so notes will not be made here.

Part II

Teaching Block II

Lecture - Intro 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 Σ^*.
  - 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.

Lecture - Deterministic Finite Automata

14:00 09/02/24 Jiacheng Tan

Rather than regular expressions, you can use state transition diagrams to describe patterns, or the process of matching said patterns. State transition diagrams (or state diagrams) are directed graphs which represent Finite State Automata (FSA) or Finite Automata (FA)

FSA

An FSA has

- A set of states
- A unique start state
- A set of one or more final/accepting states
- An input alphabet, including a unique symbol to represent the end of the input string
- A state transition function, represented by the edges of a directed graph from one state to another, labelled by one or more symbols of the alphabet

Mathematically speaking, an FSA M consists of

- A finite set, Q, of states
- A finite alphabet, Σ of input symbols
- A unique start state, $\mathrm{q}_1 \in \mathrm{Q}$
- A set of one or more final/accepting states, $\mathrm{F} \in \mathrm{Q}$
- A transition function $\delta : \mathrm{Q} \times \Sigma \to \mathrm{Q}$ which selects a new state for M based on the current state, $\mathrm{s} \in \mathrm{Q}$ and the current input symbol $\mathrm{a} \in \Sigma$

DFAs and NFAs

A finite automata can be either deterministic (DFA) or non-deterministic (NFA). For a FA to be deterministic, it must perform the exact same state transition in a given situation (it's current state and input). If the FA is non-deterministic, it can perform any state transition in a given situation

DFAs for Lexical Analysis

In the context of lexical analysis, a DFA is a string processing machine, using the following process, being in one of a finite set of states at any given step

- Read a string from left to right, one symbol at a time
- On reading a symbol, move to a new state determined by the current state and the symbol which was read
- Upon reading the final symbol, if the current state is an accepting state, then the string is valid. If not, the string is invalid

Transition Diagrams

A DFA is usually represented using a transition diagram. This is a directed graph in which each node represents a possible state, and each edge a transition between states. The label for each edge determines what input character is required for the transition to take place. The DFA begins in the initial state, represented by the small arrow pointing into state 1. Each edge is labelled with the character required to make that transition, such as requiring an 'a' to transition from state 1 to 2. Transitions can move to another state, or return to themselves. The accepting state, 4, is represented by a double circle.

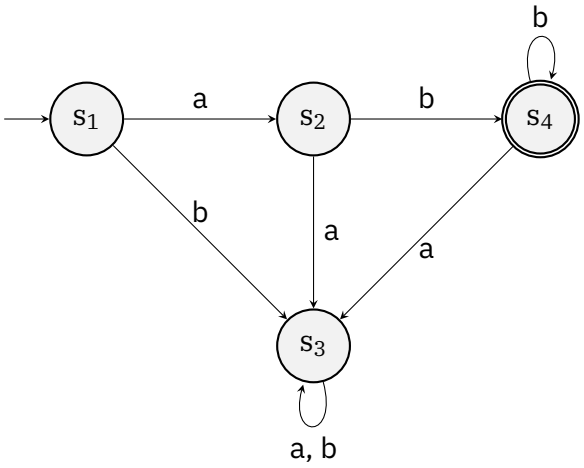

Figure 4.1: A simple DFA, M, which uses the alphabet $\Sigma = \{a, b\}$

For this DFA, the equivalent regular expression could be either $r = abb^*$ or $r = ab^+$. In the case of the transition diagram above, the DFA would be defined as

- $Q = \{s_1, s_2, s_3, s_4\}$ is the set of states
- $\Sigma = \{a, b\}$ is the DFA's input alphabet
- $q_1 = s_1, \in Q$ is the initial state
- $F = \{s_4\}$ is the set of accepting states
- The transition function can be represented as the following set of triples:
  $\{(s_1, a, s_2), (s_1, b, s_3), (s_2, a, s_3), (s_2, b, s_4), (s_3, \{a, b\}, s_3), (s_4, a, s_3), (s_4, b, s_4)\}$

Languages

The set of all strings which a DFA accepts is known as it's recognised language. 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 it's initial state, it ends up in an accepting state. For example, the language, $L(M)$ of the DFA above could be defined as $L(M) = \{ab^n \mid n \geq 1\}$, and therefore is the same as the regular expression $r = ab^+$. For any regular expression, r, there is a DFA or NFA, M, such that $L(r) = L(M)$. This makes DFAs and transition diagrams very useful for creating regular expressions, and testing that they work as intended.

Simplifying Transition Diagrams

Since most regular expressions, and therefore FAs, work with real languages such as English, each transition may have many characters for which it is valid. For example, a letter match would require 52 characters, one for each lower-case and capital letter. For this reason, as with regular expressions, you can define a set of symbols which are then used to label each transition, without rewriting the entire set of characters each time.

Building a Lexical Analyser

Lexical analysers tend to be built using one of three methods

- Write the formal description, e.g. a regular expression, of the token patterns, then use this as an input to a program such as **Lex**, which automatically generates a lexical analyser based upon the input
- Design DFAs which describe the patterns, then write a program to implement the DFAs
- Design DFAs which describe the patterns, then write a table-driven implementation of the DFAs

There are also algorithms which can be used to automatically construct a lexical analyser from the DFAs

Lex

Lex was originally written in the 70s, but since then several variants have been created, such as Quex which is a much faster implementation of the same algorithms, written in C and C++. The program takes an input file, called a lex file, which contains regular expressions for various tokens, and automatically generates the C source code for a lexical analyser.

In it's most basic form, a Lex file consists of a series of lines in the form `pattern action`, where pattern is a regular expression which should be matched, and action is a piece of C code.

Lecture - Describing Language Syntax

14:00 16/02/24 Jiacheng Tan

Context-Free Grammars

There are four classes of grammars for describing natural languages: regular, context-free, context-sensitive, and recursively enumerable. Of these, regular and context-free grammars have been found to be useful for describing programming languages. Context-Free Grammars are by far the most widely used in describing programming languages.

A context-free grammar is usually defined as a tuple, $\mathrm{G} = (\mathrm{T}, \mathrm{N}, \mathrm{S}, \mathrm{P})$, where

- T - A finite, non-empty set of terminal symbols, which consist of strings referring to parts of sentences in the language
- N - A finite, non-empty set of non-terminal symbols, which refer to syntactic structures defined by other structures and rules
- $\mathrm{S} \in \mathrm{N}$ - The start symbol
- P - A set of (context-free) productions of the form $\mathrm{A} \to \alpha$ (A produces α) where $\mathrm{A} \in \mathrm{N}$ and $\alpha \in (\mathrm{T} \cup \mathrm{N})^*$

For example, $\mathrm{G}_1 = (\mathrm{T}, \mathrm{N}, \mathrm{S}, \mathrm{P})$ where

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

or $\mathrm{G}_2 = (\mathrm{T}, \mathrm{N}, \mathrm{S}, \mathrm{P})$ where

- $\mathrm{T} = \{\mathrm{a}, \mathrm{b}\}$
- $\mathrm{N} = \{\mathrm{S}, \mathrm{C}\}$
- $\mathrm{P} = \{\mathrm{S} \to \epsilon, \mathrm{S} \to \mathrm{C}, \mathrm{S} \to \mathrm{aSa}, \mathrm{S} \to \mathrm{bSb}, \mathrm{C} \to \mathrm{a}, \mathrm{C} \to \mathrm{b}\}$

As you can see, G_2 uses a recursive production to allow for more complex productions to be simplified.

Shorthand

Rules for each non-terminal can be written in an alternative shorthand notation, using |. For example, G_1 could also be written as $\mathrm{G}_1 \mid \mathrm{ab} \mid \mathrm{aSb}$.

Backus-Naur Form (BNF)

Another alternative nottaion for CFG definitions is the Backus-Naur Form (BNF). In BNF, non-terminal symbols are given a descriptive name, enclosed within `< >`. For example, you could define `<digit>` to represent $0, 1, \ldots, 9$. This is typically the for which programming languages are actually defined in.

As an example, you could use `<exp>`, `<number>` and `<digit>` as non-terminals, and $+, -, *, /, 0, 1, \ldots, 9$ as terminal symbols. Using these symbols, the syntactic structure for an arithmetic expression could 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
```

Derivations

You can use a context-free grammar to derive strings of terminal symbols. Starting with the start symbol S, you repeatedly apply the production rules until you are left with a string containing only terminal symbols, which is known as a sentence. This process is known as a derivation. Every string of symbols in a derivation is a sentential form.

For example, if we used the grammar G_2, we can derive the string abbba as follows

- Start at the symbol S
- Apply the rule $\mathrm{S} \to \mathrm{aSa}$, and replace S with aSa to obtain the string aSa
- Apply the rule $\mathrm{S} \to \mathrm{bSb}$, and replace the S in aSa with bSb to get the string abSba
- Apply the rule $\mathrm{S} \to \mathrm{C}$, and replace the S in abSba with C to get the string abCba
- Apply the rule $\mathrm{C} \to \mathrm{b}$, and replace the C in abCba with b to get the final string, abbba which consists only of terminal symbols

If we can get from α to β using a single production, you can say that α immediately derives β, which is written as $\alpha \Rightarrow \beta$. Therefore you can write the full derivation of abbba from S as

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

With this definition of a derivation, we can define a language as “A grammar is made up of exactly those sentences which can be derived from it”

Left- and Right-Most Derivations

A derivation can be either left- or right-Most, depending upon the order in which non-terminal symbols are resolved. If you start from the left and work rightwards, that is the left-most derivation of the sentence. If you were to instead start from the right and work leftwards, that would be a right-most derivation of the sentence. You can also have a neither left- nor right-most derivation, in which you start in the middle and work outwards.

For some grammars, the left- and right-most derivations of a given sentence could be different, i.e. have a different parse tree.

Parse Trees

You can also represent the structure of an expression given by a derivation as a parse tree. I will not give an example, but the internal nodes represent non-terminal symbols which are used in the derivation, and leaf nodes represent the terminal symbols.

Lecture - Syntax Analysis and Parsing

14:00 19/02/24 Jiacheng Tan

Ambiguity

Some grammars are ambiguous, such that there are multiple valid derivations of any given sentence. This means that the parse trees would be different, and therefore could produce different results. For example, using the same grammar as the previous lecture, a left-most derivation of the sentence `x + y * z` could be either

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

which would give you a parse tree equivalent to `x + (y * z)`, or it could be

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

which would give you a parse tree equivalent to `(x + y) * z`, which gives you a completely different value.

For almost any language, it is possible to completely remove the ambiguity by introducing new or extra non-terminals and rules. For example, if you were to add a new rule that forces the + operation to appear higher in parse trees than `*`. E.g.

```
<exp> -> <exp> + <term> | <term>
<term> -> <term> * <factor> | <factor>
<factor> -> x | y | z
```

where `term` and `factor` are new non-terminals which have been added to remove the ambiguity.

The Limits of Context-Free Grammars

Some programming languages cannot be fully described using only CFGs. For example, if a variable must be defined before it is referenced, the context is required to determine whether the reference or decelera-tion comes first. These are known as Context-sensitive properties, and must be resolved by the semantic analyser rather than the syntax analyser.

Syntax Analysis

Given some input source code, the goal of syntax analysis is to: find all syntax errors and produce a de-scriptive error for the user; and produce the parse tree for the program to be used in code generation. This process is completed by a syntax analyser, sometimes known as the parser. There are several algorithms which can be used for parsing, which fall into two categories - top-down and bottom-up parsers.

Top-Down Parsers

Starting at the root (the start symbol of the grammar), each node of the parse tree is visited before it's branches. The branches are visited from left-to-right, giving a left-most derivation. When manually performing the derivation, you start by replacing the start symbol with the right-hand-side of it's production. Then you replace the left-most non-terminal symbol with the right-hand-side of (one of) it's production(s). You repeat this process until the string consists only of terminal symbols.

With the grammar

```
S -> AB
A -> aA | Epsilon
B -> b | bB
```

and the string `aaab`

```
S
AB
aAB
aaAB
aaaAB
aaa{Epslion}B
aaaB
aaab
```

More on Top-Down Parsers

Different top-down parsers may use different information or rules to determine which production should be selected to replace a non-terminal symbol. Most compare the next input token with the first symbol of each production, these parsers are known as predictive parsers. These work using only the next input symbol and the current non-terminal.

Recursive-Descent Parsers (RDP)

A recursive descent parser is an implementation of a parser based upon the BNF of a grammar. An RDP consists of a collection of functions (or sub-programs), many of which are recursive. Each non-terminal symbol corresponds to a single function, which handles parsing that particular non-terminal symbol in the grammar. For example, if you wanted to implement the following grammar

```
<exp> -> <exp> + <term> | <exp> - <term> | <term>
<term> -> <term> * <factor> | <term> / <factor> | <factor>
<factor> -> integer | (<exp>)
```

with an RDP, you would need to implement 3 functions - `exp()`, `term()` and `factor()`. Assuming that there is another function, `lex()` which updates the variable `nextToken` to be the next token in the sentence, each function will need to

- Check if the symbol is terminal, in which case make a call to `lex()`
- Check if the symbol is non-terminal within the current production, in which case make a call to the corresponding function
- If it is neither, then there is a syntax error and it should be raised with a helpful message for the user

Rules With Multiple Productions

When parsing a rule with more than one production, it is necessary to select which of the productions should be parsed. This can be done in several ways, but in the case of a predictive parser, the production should be selected based upon the next input token. The next input token is compared with the first token of each production until either a match is found, or all options are expended. If the token does not match any of the productions, there is a syntax error and an error should be raised with a helpful message for the user.

Rules with Left Recursion

If a grammar has left recursion, it cannot be directly used by a recursive-decent parser. This is because it leads to an indefinite or non-terminating recursion loop. A left-recursive grammar cannot be transformed into one which is not left- recursive. Instead, the grammar must be modified to remove any direct left recursion. For each non-terminal, A, group the A rules as $\mathrm{A} \rightarrow \mathrm{A}\alpha_1 \mid \mathrm{A}\alpha_2 \mid \cdots \mid \mathrm{B}_1 \mid \mathrm{B}_2 \mid \cdots \mid \mathrm{B}_\mathrm{n}$ where $\mathrm{A}\alpha_\mathrm{m}$ represents any rules with left-recursion, and B_n represents any rules without. To get rid of the direct recursion, you have to add a new non-terminal, such as $\mathrm{A}\prime$ and replace the original rules with $\mathrm{A} \rightarrow \mathrm{B}_1\mathrm{A}\prime \mid \mathrm{B}_2\mathrm{A}\prime \mid \cdots \mid \mathrm{B}_\mathrm{n}\mathrm{A}\prime$ and $\mathrm{A}\prime \rightarrow \alpha_1\mathrm{A}\prime \mid \alpha_2\mathrm{A}\prime \mid \cdots \mid \alpha_\mathrm{m}\mathrm{A}\prime \mid \epsilon$

Lecture - LL(k) Parsers

14:00 23/02/24 Jiacheng Tan

An LL(k) parser is a top-down, predictive parser. It's name means that it parses from (L)eft-to-right, (L)eft-most derivation, with (k) tokens of look-ahead. They are also known as a table-driven predictive parser, since they use a stack and a parsing table.

An LL(1) parser parses the input left-to-right, and always using a left-most derivation. In this case, it uses one token of lookahead to predict which production should be used. It also uses a stack to store the symbols of the right-hand-side of productions, in right-to-left order, as that way the left-most symbol is always at the top of the stack. A parsing table is also used to store the rules which the parser should use based upon the input token and which value is at the top of the stack.

Parse Tables

With the grammar

```
E  -> TE'
E' -> +TE' | Epsilon
T  -> FT'
T' -> *FT' | Epsilon
F  -> (E) | int
```

the parse table might look as below

Top of Stack Input	`int`	`+`	`*`	`(`	`)`	`$`
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \texttt{int}$			$F \to (E)$		

With this parse table, it is quite easy to parse a sentence, as it is a matter of simply picking the rule from the table, according to the current non-terminal (on the top of the stack) and the current input symbol and pushing the right-hand side of the production back onto the stack. $ is selected if the end of the input is reached. The parsing process begins with the start symbol (E) and it's right-hand side (TE') is pushed onto the stack, and so T is the top of the stack. If the next token is `int`, we would pick the rule $T \to FT'$, and so FT' is pushed onto the stack. As such, the top of the stack is now F.

This process is more generally written as

- If X and w are both the end symbol, $, stop and accept the input
- if X is a terminal, if $X = w$, pop X off the stack and get the next token, otherwise halt and give a descriptive error to the user
- If X is a non-terminal, if there is a production at position $[X, w]$, push the right-hand side onto the stack, otherwise halt and give a descriptive error to the user

Parse Table Construction

It is easy to perform an LL(1) parse if the parse table is already available. To construct the table, you must compute the first and follow sets of the non-terminals from the grammar. These are sets of terminal symbols. If these sets are available, the construction of the table is a simple procedure which can be performed automatically.

First Sets

For the grammar

- $\mathrm{E} \to \mathrm{TE}'$
- $\mathrm{E}' \to +\mathrm{TE}' \mid \epsilon$
- $\mathrm{T} \to \mathrm{FT}'$
- $\mathrm{T}' \to *\mathrm{FT}' \mid \epsilon$
- $\mathrm{F} \to (\mathrm{E}) \mid \mathrm{int}$

The first sets of its non-terminals are

- First(E) = First(T) = First(F) (Rules 1&3) = $\{(, \mathrm{int})\}$ (Rule 5)
- First(T′) = $\{*, \epsilon\}$ (Rule 4)
- First(E′) = $\{+, \epsilon\}$ (Rule 2)

The first set of a non-terminal symbol, A, is the set of terminals which start the sequences of symbols which can be derived from A. (Written as First(A)). To calculate First(A) where A is in the form $\mathrm{A} \to \mathrm{X}_1, \mathrm{X}_2, \ldots, \mathrm{x}_\mathrm{n}$, you must follow the process below

- If X_1 is a terminal, add X_1 to First(A), and that's it.
- Otherwise, add First(X_1) to First(A), as any symbol which starts X_1 also starts A
- If X_1 can go to ϵ, e.g. it is nullable, add First(X_2) to First(A). Repeat this for X_2 until you find the first non-nullable symbol.
- If all of these are nullable, add ϵ to the first set.

Follow Sets

Lecture - Bottom-Up Parsing

14:00 01/03/24 Jiacheng Tan

Bottom-up parsing works (shockingly enough) in the opposite direction as top-down parsing. A bottom-up parser starts with the string of terminals and works backwards to the start symbol, applying the productions in reverse as it goes.

With the grammar

- $\mathrm{S} \to \mathrm{AB}$
- $\mathrm{A} \to \mathrm{aA} \mid \varepsilon$
- $\mathrm{B} \to \mathrm{b} \mid \mathrm{bB}$

A bottom-up parse of the string aaab would look like

- Starting with the right-most symbol, we can apply the production $\mathrm{B} \to \mathrm{b} \mid \mathrm{bB}$ in reverse, to end up with the string aaaB
- Since neither a nor aB are the right-hand-side of a production, insert ε to give the string $\mathrm{aaa}\varepsilon\mathrm{B}$.
- Replace ε with A to get aaaAB
- Replace aA with A to get aaAB
- Replace aA with A to get aAB
- Replace aA with A to get AB
- Replace AB with S to get the start symbol, and therefore a valid sentence

vs Top-Down

Bottom-up parsers are typically more powerful than top-down parsers. There are excellent generator tools, such as `yacc` that can build a parser from an input specification, as `lex` does for scanners.

Shift-Reduce Parsing

A shift-reduce parser takes a stream of tokens as an input and creates the list of productions used to build a parse tree. It uses a stack to track the position in the parsing process and a parse table to determine the correct production to use. Shift-reduce parsing is typically the most common and most powerful method of bottom-up parsing.

One type of shift-reduce parser is an LR parser (Scans input from left-to-right, using a reversed right-most derivation)

The Shift-Reduce Process

When parsing a string of tokens, v, the input is initialised to v and the stack is empty. At each step, the parser can take one of four actions at each step - shift, reduce, accept or error. The first step of the process is always to shift the first token to the top of the stack.

For this example, the grammar will be as follows

- $\mathrm{S} \to \mathrm{E}$
- $\mathrm{E} \to \mathrm{T} \mid \mathrm{E} + \mathrm{T}$
- $\mathrm{T} \to \mathrm{id} \mid (\mathrm{E})$

Shift

The token at the start of the input string is *shifted* onto the top of the stack.

Reduce

Suppose that the contents of the stack are qw where w is a string of terminal and non-terminal symbols, and q may be an empty string. If there is a production such that $\mathrm{A} \to \mathrm{w}$, the stack can be *reduced* to qA. This means that the production for A is applied backwards, such that we replace the right-hand-side (w) with the left-hand-side non-terminal (A). In this case, w is known as a handle.

Accept

If the entire contents of the stack has been reduced to the start symbol (S), and there are no remaining input tokens, the input is a valid sentence in the parsers grammar. This means that the parser has ended in an acceptance state.

Error

If it is not possible to Shift, Reduce or Accept, then the parser must Error. In this case, the sequence on the stack cannot be reduced to the left-hand-side of any production, and any further shifting would be pointless as the input cannot form a valid sentence in the parsers grammar.

Lecture - Scopes and Memory Allocation

14:00 11/03/24 Jiacheng Tan

Variables

A variable is a place-holder for a run-time value. Each variable has multiple run-time attributes. These attributes include:

- Name - The name which is used in code to refer to the variable
- Address - The memory address which stores the value of the variable
- Value - The contents of the memory which corresponds to the variable
- Type - The range of values the variable can store, and the operations which can be performed upon it
- Lifetime - The time for which the variable is bound to the specific memory location
- Scope - The area in code which the variable is accessible

Implicit vs Explicit Declaration

A variable is introduced into the scope using a declaration. This can be done implicitly or explicitly. An explicit declaration is a statement in the source code which defines the type and name of a new variable, e.g.

```
C - int i;
Pascal - i : integer
```

An implicit declaration is a mechanism by which variables are automatically assigned a type based upon conventions of the language, rather than declaring it manually. For example, in Fortran the type of a variable is defined by the first character of the variable name. If the first character is I, J, K, L, M, or N then it is an integer, otherwise it is real.

Another form of implicit declaration is type inference. For example, if you define a `late` variable in Dart, it infers the type of the variable from the first value which is assigned to it, e.g.

```
late final variable;

variable = "This is a string";
```

In this case, the variable would be assigned the `String` type.

Binding

A binding is an association between an entity and attribute, such as between a variable and it's type, or a symbol and the operation it corresponds to. Binding can take place at many different times, but is always referred to as the *binding time*. An example of this are the following C statements

```
1. int x;
2. x = 1;
3. x = x + 1;
```

In this case,

- The type of x is bound at compile time, as C is a statically typed language
- The range of values which x can take is bound when the compiler is designed, as this is when they pick the number of bits, and therefore maximum value that an integer can be
- The operation of the + operator is bound at compile time, as this is when the types of it's operands are known
- The value of x is bound at run-time, since that is when the statement is executed

Binding can also take place at load time (when the variable is bound to a memory location) or link time (the variable in one module is bound to another module)

Static vs Dynamic Type Binding

A type binding can be either static or dynamic. It is static if it occurs before run-time, e.g. at compile time, and remains unchanged throughout the execution of a program. Declarations always have type information, and so binding is done at compile time. Languages with static type declaration are known as statically typed as, once set, the type cannot change. This gives the advantage of type errors being detected at compile time, and using less memory for each variable, but it is not very flexible and can lead to issues when user input is involved.

A dynamic type binding occurs when the type is bound during execution, or if it is able to change at run-time. This is the case in languages such as Python, JavaScript and PHP. These languages are known as dynamically typed, since the type of any given variable is unknown until run-time and can change at any point. This has the advantage of being more flexible (useful when user input is necessary) and allowing the developer to not need to know the type of a variable when the program is written, but has a much higher overhead due to dynamic type checking, and makes compile-time type error detection impossible.

Strongly and Weakly Typed Languages

A language is known as strongly typed if **all** type errors can be detected by the compiler. In this case, a language is also type safe, as it is impossible for the program to crash due to an invalid type. Some examples are Java, Haskell and Ada. Some languages may seem to be strongly typed languages, such as Fortran or C, but actually aren't. A strongly typed language can be either statically or dynamically typed.

A weakly typed language is any language which is not strongly typed, such as JavaScript. Once again, a weakly typed language can be either statically or dynamically typed.

Scopes

A block is a section of code, which defines the local environment for any given statement. Blocks are usually denoted by a start and end marker, such as in C, which uses { and }, but can also be denoted by indentation, if you are a masochist and/ or sadist (cough cough Python). A block can contain variables local to said block, and has it's own reference environment (the names and identifiers which are accessible to statements in the block).

Most languages allow blocks to be nested within each other, but some may restrict the type of blocks which can nest. In C and Java, methods can be nested within classes and other blocks, but not within other methods. In Python and Pascal, it is possible to nest a method within a method.

Since blocks contain variable definitions, the local reference environment for each statement must be determined, which also allows you to determine the **scope** of identifiers. The scope determines where each identifier can be used, and if an error should be thrown if there are multiple identifiers with the same

name. In most languages, the scope of an identifier is the block in which it is declared and, by extension, any nested blocks. There is also a scope known as the global scope, which allows identifiers within it to be accessed anywhere within the program. These are known as global identifiers, and are typically only used when they are absolutely necessary, to reduce the chance of overlapping identifiers.

Duplicated Identifiers

Most languages have a rule or set thereof to determine which declaration of an identifier takes precedence. In languages such as Pascal, the local variable *hides* any variables with a larger scope. In this case, it is said that there is a hole in the scope of the hidden variable, as it's scope is everywhere aside from any blocks in which it is hidden.

Some languages just flat out refuse to allow any duplicated identifiers, as this reduces the confusion since it will cause an error at compile time, rather than a mysterious run-time error.

Static and Dynamic Scoping

In the previous example, the scope of variables is determined by their visibility, and therefore the scope depends upon the lexical structure of the program, and the compiler can therefore determine the scope of all variables. This is known as static or lexical scoping. On the other hand, with dynamic scoping, the reference environment depends upon the sequence of sub-programs, rather than the layout of the nested blocks. This means that the scope of any given identifier can only be determined at run-time.

Dynamic scoping is typically less used than static scoping, as it is less reliable due to the unknown order of execution and results in poor code readability. It is however, easier to implement using stacks and resolving a variable does not require tracing the structure of the entire program.

Lifetimes

The lifetime of a variable is the period of time for which the variable exists, and has a value that is only known at the time of execution. The lifetime of a local variable is the execution of the method. Each recursive execution of a method has its own copy of any local variables. The lifetime of a global variable is the execution of the entire program.

This is related to the way that memory is allocated and deallocated. There are several basic mechanisms which a programming language may use to allocate memory

- Static allocation is when a fixed memory address is retained throughout the variables lifetime
- Stack-based allocation is done on a last-in first-out basis, and is used for function calls, as the lifetime of the variable is that of the function
- Heap-based allocation is used for variables that are dynamically allocated. They often have no identifier, and are therefore known as anonymous variables, and can only be referenced by pointers

Lecture - Elementary Data Types

14:00 15/03/24 Jiacheng Tan

An elementary data type is one which has a fixed size in memory. Typically, languages have a few elementary data types, such as numeric types (Integer, Floating Points, etc), Boolean, Character, etc. There are also usually "enumerated" types, which can take any of a set of values, which are usually represented internally as Integers.

Most imperative languages also provide built-in operators for computing using these elementary data types. These typically include arithmetic, relational and boolean operations, such as
`+ - * / == < <= > >= || &&`, etc.

Enumerated Types

Enumerated types are *ordinal* types, in which the possible values can be associated with the set of positive integers. This means that they can be used to represent order, and that you can effectively 'index' the values using an integer.

Some built-in types are enumerated, but they can also be user-defined. This is useful for storing information such as the day of the week, month of the year, or season. For example, you could represent the seasons in C++ as follows

```
enum Season = {
  spring,
  summer,
  autumn,
  winter,
};
```

Then, a variable which has the type `Season` could store any of the four seasons.

Most languages also automatically define operators for enumerated types. These include basic functionality such as the equality operator, but more complex functions like relational operators, which allow you to do things like checking if the season is 'greater' than `summer`. You can usually also interact with the real data type behind the enumerated values, which is typically an integer. This allows you to increment or decrement, as well as performing arithmetic with your enumerated values.

Pointer Types

Pointer (or reference) variables store a location in memory as their value. This allows them to point to a piece of data in memory. Pointers are meant to work with memory locations, allowing for more efficient use of limited memory space. As this is less of an issue nowadays, fewer and fewer languages allow you to access the raw memory locations, and is typically limited to lower-level languages like C and C++.

Memory is often dynamically allocated from the heap, and are known as heap-dynamic variables. Typically, they do not have an identifier, and can therefore only be accessed by using a pointer to the memory address.

Pointer *dereferencing* is the act of accessing the value stored in the memory pointed to by a pointer. With the dereferenced value, you can read the current value or overwrite a new one. This is useful, as it allows you to pass a pointer to a memory location, and modify it in a function, without having to pass a

value in and return the modified value. This is especially useful in low-level languages which don't support compound data types.

There is also a special `null` pointer, which is usually stored as `0`. This is used to signify that a pointer does not point to any location, and is typically the value which pointers are initialised to.

Dynamic Memory Allocation

Since a pointer is a reference to a location in memory, you can also use them to dynamically allocate space in memory for values at runtime. This allows you to both store as many values as you need to, but also to store arbitrarily large values in memory. For example, if you wanted to store a string, you might not know how long the string will be at compile time. Therefore, you can dynamically allocate a space in memory which is large enough to store the string, and then access it using a pointer.

Memory Deallocation

Memory addresses need to be *deallocated* when they are no longer needed, otherwise the program's memory space would fill up over time, with unused values being stored unnecessarily. Deallocation can be done either implicitly or explicitly. Assigning `null` to a pointer removes the only way of accessing the allocated memory, which still contains the last value which was assigned to it. The runtime environment can then 'garbage collect' the memory address and deallocate it so it can be used again. To deallocate memory explicitly, you call the `free` function, with a pointer to the address you wish to free. This will mark it as available for reuse. It is good practice to also set the value of the pointer to `null`, so that you don't have a *dangling pointer*.

Array Indices

In languages such as C, arrays are actually just pointers to a block of memory. For example, an array of 10 integers would be stored in memory as a contiguous block of memory which is 10 times the size of an integer. Then, when indexing the array, the specified index acts like an offset, adding to the pointer to the first element to get the address of the value you're interested in. Since they are accessed via a pointer, it is also possible to dynamically allocate an array of arbitrary length, useful for storing a list of input values.

Dangling Pointers

A dangling pointer is one which points to a memory address which has already been deallocated. This is dangerous and liable to cause the program to crash, as it may attempt to read from or write to the memory of another program. On Unix systems, accessing a dangling pointer guarantees a crash as the kernel will notice the program attempting to read outside of its assigned memory and kill it, giving a `SEGMENTATION FAULT` error. On Windows systems, this could still cause a crash as the program may write into another programs memory, or read an invalid value placed there by a different program. Because of this, it's important to destroy the pointer when it is deallocated, typically by setting its value to a `null` pointer.

Lecture - Compound Data Types

14:00 22/03/24 Jiacheng Tan

A compound (or structured) data type is one which is made up of simpler types. This includes types such as arrays, strings, records, structs and maps.

Arrays

Arrays or lists are the most common compound data type and are found in most programming languages. In general, an array has several attributes, namely

- The type of its elements (This is also the compound type)
- The type of its indices (This is usually integers, but it is possible to use an enumerated type in some languages)
- The number of elements in the array

Different languages define each of these attributes in different ways. For example, some languages like Dart have dynamically sized and allocated arrays, which allows you to increase or decrease the size of the array at runtime, but others, such as C or C++, have statically sized and allocated arrays. This means that the size of the array is determined at compile time, and cannot change at runtime. Different languages might also use a different starting index for their arrays. For example, C uses 0-indexed arrays, but higher level languages such as Lua use 1-indexed arrays.

Dynamic Arrays

There are two methods of creating dynamic arrays - stack-dynamic and heap-dynamic arrays. A stack-dynamic array has its size dynamically set, and the storage is allocated at runtime. They live on the stack, and the size of the array is fixed after it's created. Heap-dynamic arrays live on the heap and are able to dynamically change size at runtime.

Heterogeneous Arrays

A heterogeneous array is one in which the type of the elements is not necessarily the same. This is only supported by a few high-level languages: Perl, Python and JavaScript. A heterogeneous array can also be the element of another heterogeneous array, which allows for some very funky stuff.

Rectangular & Jagged Arrays

A rectangular array is a multi-dimensional array in which each row has the same number of elements and each column has the same number of elements.

A jagged array is a multi-dimensional array in which each row has a varying number of elements, columns don't really exist as they would be impossible to align meaningfully. This is simply an array of arrays. Supported by C, C++ and Java.

Strings

In the vast majority of languages, a string is an array or list of characters. More literally in some languages, such as C/C++ and Haskell, but in higher-level languages, it is usually a built-in data type which hides its true type behind a 'fake' type. These languages also tend to have a set of logical operations such as finding the length of a string, and functions such as concatenation.

Records

A record is a compound data type which is composed of a number of named elements. These are useful for storing information about an object, such as a Person or Student, while encapsulating them into one pseudo-type.

Variant Records

Records are typically very inflexible, and may be memory-inefficient since all of the values of a record have a fixed size and set of fields. For example, if you wanted to store a student record, you would need to store a name, registration number, information about their course, and a year of entry and graduation. This means that even if the student has not graduated, you still need to store a null value for their year of graduation. Pascal has a *variant record*, which allows you to store a value for one element, if and only if another element has a specific value. In the case of a student record, you could store a boolean value for their graduation status, and then only store a graduation date if the status is true.

Unions

C & C++ have a type known as a Union rather than variant records. Unions are designed to store data of multiple types in the same memory space. When you define a union, you specify the types that should be stored for that value, but then you have to set the type and reference it as such elsewhere in the code. This allows you to store values for different purposes without wasting memory. You can have only one of the values stored at a time, since they would use the same space in memory. The main difference between this and a variant record is that the two values stored in a union are not related in any way, and it is possible to ignore the precedent set elsewhere in the code, which can cause issues.

Structs

Languages such as C, C++, C# and Rust all have a similar concept to a record, known as a *struct*. These perform a similar function to records, but can also include member functions. A member function does not directly access the data in the struct (unlike a class), but is used to group a function with the data type it operates on.

Classes

Structurally, classes and structs are very similar, but a class has a constructor (a function which is called when the class is instantiated), and methods rather than member functions. A method is able to directly access the data stored in the class, and is therefore able to act as a getter/setter. Another difference is that in most languages, member functions in structs are public by default, but methods in classes are private by default.

Lecture - Expressions and Assignments

14:00 15/04/24 Jiacheng Tan

Expressions

An expression is a combination of values, variables, operators and/ or function calls. Expressions can be used to evaluate mathematical values, move data around and more.

The main types of expressions are

- Arithmetic
- Relational
- Boolean
- Assignment

These expressions can then be used and chained together to create various algorithms and programs. To be able to correctly evaluate expressions, you must know the rules of precedence and the rules of associativity of operators.

Rules of Precedence

The rules of precedence determine the order in which operations should be evaluated in an expression. The typical order is along the lines of

1. Parentheses
2. The unary operators ++ and `--`
3. The unary operators + and `-`
4. The binary operators `*` and `/`
5. the binary operators + and `-`

Rules of Associativity

The rules of associativity determines how operators with the same precedence are grouped together, if they have been left in an ambiguous state with no parentheses. For example, the expression `3 / 5 * 0` is ambiguous as `/` and `*` have the same precedence. Therefore, the order of operations is determined using the associativity of each operator.

Any given operator may be

- Associative - The operations can be grouped in any order, e.g. `a * b * c` could be `(a * b) * c` or `a * (b * c)`
- Left-associative - The operators must be grouped together from left-to-right
- Right-associative - The operators must be grouped together from right-to-left

Most mathematical operators inherently have associativity, such as subtraction and division being left-associative and addition and multiplication being associative.

Many programming languages include a table of operator precedence and associativity in their documentation, but in general, most operators are left-associative apart from the assignment operator, which is necessarily right-associative.

Operator Overloading

When a language uses the same operator for more than one purpose, it is *overloading* the operator. For example, in Dart + is used for adding `int`s and `float`s, but is also the operator for string concatenation. This means that the function of the operator is determined by the semantics of the language, as they determine which variant of the operator should be used. Some languages, such as C++ and C# allow user-defined operator overloads, which can be used to improve the readability of user-defined classes and data types.

Side Effects

An expression is said to have a *side effect* if as well as returning a value, it also modifies a variable or changes the flow control of the program. The four main causes of side effects are

- Assignment operators
- Increment/ Decrement operators
- Function calls
- Method invocation

Assignment as an Expression

In some languages, the assignment operator (=) is treated the same as any other binary operator, and as such still returns a value. However, it also has the side effect of changing the value of the left operand. For example, the expression `x = y + 1` both returns the value `y + 1`, and changes the value of x to be `y + 1`.

This has the advantage of allowing you to do things like `x = y = z + 1`, but also has issues. For example, if you were to use an if statement with the condition x=y, you might be trying to use it as an assignment and be treating the returned integer as a boolean value, which is perfectly valid, or you might have missed the second = to use the relational operator ==. This mistake is impossible to detect by the compiler, but may occasionally be detected by the IDE or Linter as a logic error.

Compound Assignment Operators

A compound assignment operator is one which combines an assignment with some form of arithmetic operator. This includes operators such as += and `-=`, which add and subtract the right operand from the current value of the left operand, but also assign the new value to the left operand.

Unary Assignment Operators

Some languages also include unary assignment operators, which both perform some arithmetic and assign that value to the only operand. This is operators such as ++ and `--`, which increment and decrement the value respectively. The order of assignment and increment is important, as `++count` and `count++` have two different meanings. In C, `sum = ++count` would increment `count` by one, then assign it to `sum` (pre-increment), but `sum = count++` would assign `count` to `sum` and then increment `count` by one (post-increment).

Type Conversions

Different languages follow different approaches when it comes to the compatibility of different types, when used in an expression. For example, some languages such as Dart require you to *explicitly* convert the types using a built- in function, but some languages such as JavaScript allow you to *cast* values between types. There are also some languages which allow casting for some types, but require explicit type conversion for others.

Casting is type conversion which is explicitly requested by the code, but which does not make use of a function. For example, in C you can cast an unsigned integer to a signed integer using

```
uint a = 10;
int b;
b = (int)a;
```

Type coercion is when casting is done implicitly by the compiler. This means that the compiler checks the compatibility of the types of the left and right operand, and automatically casts them if they are compatible. This is used heavily in scripting languages such as JavaScript and Python.

Arithmetic Type Conversions

There are two types of type conversion - widening and narrowing. A widening conversion is one in which an object is converted to another type which can store all values of the original type, and more. One example of this is casting an integer to a long integer (32-bit integer to 64-bit integer). This type of conversion is completely safe and used all the time. A narrowing conversion is one in which the new type can store only a subset of the values the original type could store. An example of this is converting an integer to an unsigned integer. This type of conversion is unsafe since an overflow or other issue could occur if the programmer is not careful. Typically, compilers will only coerce types when it is safe to do so, e.g. a widening conversion.

Notations

There are several different ways of writing any given expression, which are known as different notations. The notation used in mathematics and most programming languages is known as *Infix Notation*, but the issue with it is that it's inherently ambiguous, at least when not used in conjunction with the rules of associativity and precedence. Given the inherent issues with infix notation, several other notations have been developed.

Prefix Notation

With prefix notation, operators appear before the operands. For example, the expression `a + b - c * d` would be written as `-+ab*cd`. It is evaluated from left-to-right by combining the operator with the two operands in front of it, meaning that is inherently unambiguous. When evaluating the previous expression, you would evaluate the operators in the following order

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

Below is the same expression, but using numbers to make it easier to see the order. (Numbers are shown in brackets for ease of reading)

```
-+(7)(9)*(2)(5)
-(16)*(2)(5)
-(16)(10)
(6)
```

This is the same expression, but with `a = 7`, `b = 9`, `c = 2` and `d = 5`

Postfix Notation

With postfix notation, operators appear after their operands. For example, the expression `a + b - c * d` would be written as either `ab+cd*-` *or* `abcd*-+`. (I'm not sure about that second one so I'll just use the first here on out). It is also evaluated left-to-right, but by combining two operands with the operator after them. This notation is also inherently unambiguous, but is harder to convert to since there are several valid options. To evaluate the previous expression, you would do the following

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

Once again, below is the same expression but using the numbers `a = 7`, `b = 9`, `c = 2` and `d = 5`

```
(7)(9)+(2)(5)*-
(16)(2)(5)*-
(16)(10)-
(6)
```

Cambridge Prefix Notation

Cambridge prefix notation (or just Cambridge notation) is a variant of Prefix notation which introduces parentheses. This has the advantage of allowing operators like + and - to be n-ary, e.g. `a + b + c` would be written as `++abc` in standard prefix notation, but as `(+abc)` in Cambridge notation.

Lecture - Control Structures

14:00 19/04/24 Jiacheng Tan

The flow control or execution sequence allows you to implement complex algorithms, and can be examined on several levels, namely

- Within expressions (i.e. rules of Associativity and Precedence)
- At the statement level
- At the program unit level

Statements which enable the program to select between different execution paths are known as *control statements*. This includes conditionals such as if and case blocks, as well as while loops.

In previous years, unconditional branching statements like `break` and `goto` were the only options for controlling the flow of execution, but these led to poor readability and maintainability. Since then, it was realised that such unconditional branching statements are actually unnecessary, as long as other mechanisms such as functions and procedures are available.

Structured Programming Theorem

The *Structured Programming Theorem* states that all algorithms that can be expressed using a flowchart can also be implemented in programming languages with two basic control statements – selection and pre-test logical loops. This also means that these control statements are necessary for any imperative programming language. In most programming languages, there are several variations of each type of control statement.

Selection Statements

Selection statements pick between two or more execution paths in a program. These typically fall under one of the two sub-types, two-way selection statements or multiple-selection statements.

Two-Way Selection

This mainly consists of basic selection statements such as `if-then-else` blocks. The general syntax of a two-way selection is as follows–

```
if {control expression}
then {execution path}
else {execution path}
```

In this case, the control expression could be any expression which evaluates to a boolean type, since there are only two possible execution paths. Each of the execution paths could be as simple or complex as needed, and are typically enclosed within parenthesis (in sane languages) or within an indented block in python.

Most languages also support nesting selectors, which is made easier in cases which use parenthesis. In cases which don't, it is typically assumed that an `else` matches to the nearest previous `if` statement.

The vast majority of languages also support a short-hand `if` statement, which in most C-derived languages, is written as follows:

```
({control expression}) ? {execution path} : {execution path};
```

This is usually only used when assigning a value to a variable based upon a condition, but could theoretically be used anywhere the conventional notation is used.

Multiple-Selection

A *multiple-selection* allows the selection of any number of execution paths based upon the value of the control expression. These usually take the form of a `switch-case` block, as shown below–

```
switch {control expression}
  case {value1} {execution path}
  case {value2} {execution path}
  ...
  default {execution path}
```

The control expression can typically take any of many types, but they are typically a number, character, string or enumerated type.

Most languages use the *fall-through* behaviour once reaching the end of a cases execution path. This means that they fall-through to the next case and execute that path as well. This is useful since it allows you to specify multiple values which use the same branch, but might not always be desired. In these cases, you can end the execution path with a `break` statement, which immediately skips to the end of the switch block.

Pre-Test Logical Loops

A pre-test logical loop is a mechanism for repeatedly executing a statement based upon a simple condition. There are typically two types of loop, `for` loops and `while` loops, which are controlled by a counter and logic statement respectively.

For Loops

A for loop typically takes the following form–

```
for ({expression 1}; {expression 2}; {expression 3})
  loop {execution path}
```

With this syntax, the first expression is for initialisation and is only evaluated once, before the loop begins. The second expression is the loop control, and is evaluated before each execution of the loop. It must take a boolean value, and determines if the loop should execute again or terminate. The final expression is for stepping, and is executed after every execution of the loop body.

Logically Controlled Loops

There are two sub-types of logically controlled loops, typically known as `while` and `do-while` loops. These pre-test and post-test the control expression, respectively. This means that a `do-while` loop will always execute *at least* once, and a `while` loop may not execute at all. They typically take the following forms–

```
while {control expression}
  loop {execution path}
```

and

```
do
  loop {execution path}
while {control expression}
```

respectively.

Any `for` loop can be re-written as a while loop, with an external and manually incremented counter.

Unconditional Branching

Any statement which changes the flow of execution without a condition is *branching unconditionally*. These usually include `break`, `continue` and `return`. Within the previously discussed looping structures,

- `break` unconditionally exits the loop immediately
- `continue` unconditionally skips the remainder of the current iteration, but allows the loop to continue
- `return` terminates the current function or method call and immediately returns a value to the caller

In some languages, such as Java, a break or continue can be either labelled or unlabelled.

Unlabelled & Labelled Breaks

An unlabelled break terminates the innermost nested switch, for, while or do-while structure. On the other hand, a labelled break terminates a correspondingly labelled structure, as listed before. Execution then jumps to the statement immediately after the labelled structure, and continues as usual. This may be useful when searching through a two- dimensional structure.

Unlabelled & Labelled Continues

Similarly, an unlabelled continue terminates the current iteration of the innermost nested for, while or do-while loop, and evaluates the control expression before continuing normal execution. A labelled continue, on the other hand, skips the current iteration of the correspondingly labelled loop.

Lecture - Functions and Parameter Passing

14:00 26/04/24 Jiacheng Tan

A *subprogram* is a fundamental part of program flow control, and are present in almost every language. Decomposing a problem in to sub-problems makes it much easier to handle, as the complexity of any given part is much lower. Subprograms go by many names, but are usually called a *function*, *procedure* or *subprogram*. It is any piece of code which is identified by a name, and has its own local reference space. There is usually some facility to exchange data with the other code using parameters. Procedures and functions are both types of subprogram, where a function returns a value, and a procedure does not.

Abstraction

Subprograms provide one of the two fundamental abstractions in programming languages

- Process control abstraction – Allows details of procedures to be hidden, and only exposes the interface and not the implementation
- Data abstraction – Allows the use of sophisticated data types without needing to know their implementation, only the interface

Function Declarations & Definitions

The *definition* of a function refers to the actual implementation of the function. This includes both the header or interface, as well as the body of the function, the actual code that runs. The header consists of the name, return type, and parameter profile of the function. The parameter profile of a function is the number, order and types of parameters, as well as their names in languages which support named parameters.

The *declaration* of a function (sometimes known as the prototype) is the header of the function on its own. This is used in languages like C and C++ to expose functions to other files in the program. If a function needs to be used in another file, it's declaration is placed in a header file, which is then included in other files. This allows the header of the function to be exposed, without exposing the implementation as well.

Parameters and Return Values

A *formal parameter* is a dummy variable in the function header, which can then be referred to by the implementation of the function. The *actual parameters* are then filled in at runtime, and is usually either the value or address which the caller used. Functions are able to report results by returning a value, typically of a type which was specified in the declaration of the function. When a return statement is called, the value is passed back to the caller, and execution resumes at the caller.

Parameter Binding

Actual parameters can be bound to the formal parameters in one of two ways – by position or by keyword/name. When bound by position, the values are bound in the order that they appear in the function call, which is safe and easy. When bound by keyword, the names of the formal parameters are specified manually in the function call, and the order they appear is entirely irrelevant. This means that it is impossible for a programmer to accidentally put the wrong values into the parameters, but it does mean that they have to remember the names of the formal parameters.

Parameter Passing

There are several ways in which a parameter can be passed into a function. These determine how the values are sent to the function, and if it is possible for the function to modify them. The relationship between the actual and formal parameters can be one of three models

- In (Put) Mode – Formal parameters receive data from the actual parameters, aka pass by value
- Out (Put) Mode – Formal parameters transmit data to the actual parameters, aka pass by result
- Inout Mode – Both occur, can be implemented either using both pass by value and result, or by passing the reference

The vast majority of languages support passing values by value, result or both and default to passing by value. Only some languages support passing values by reference, and most are low level, such as C or C++.

Pass by Value

The value of the actual parameter is used to initialise the corresponding formal parameter, at the time of function call. During the execution of the function, the local variable uses the value passed to it, and can be modified, but is then destroyed when the function returns.

This is very simple and fast for sending elementary data types, and is the default in many languages, such as C, C++ & Java. It does have the downside of needing to use more memory, as the value is stored twice, and needing to move data in memory, which can be expensive, depending upon the type of the variable.

Pass by Result

The value is transmitted to the variable when control returns to the caller, and no value is sent from the caller to The function. During execution of the function, the formal parameter acts as a local variable within the function, and their values are copied to the real parameters when control returns to the caller.

This is also simple, but does have quite a few limitations and downsides. Each of the parameters must be a variable, and the order which the values are copied back to the actual parameters does matter. This is a bigger issue in languages which have multiple implementations, such as C#.

Pass by Value and Result

Sometimes known as pass-by-copy, this is a combination of the two previous methods. When the function is called, the values of the actual parameters are stored in the formal parameters using pass-by-value, and at the end of the function call, the values of the formal parameters are copied back to the actual variables.

This facilitates bi-directional data exchange, but does have the disadvantage of needing to copy the data twice, and store two copies of it the whole time.

Pass by Reference

Rather than passing the values, a pointer to the address in memory is used instead. This is sometimes known as pass-by- sharing. The formal parameters and actual parameters are simply treated as two pointers to the same address in memory, which can be written to by both the caller and the function.

This also allows bi-directional data exchange, but without having to make multiple copies of the data. This makes it much more efficient, and more convenient than passing by value and result, but could be seen as insecure, as it gives the function direct access to the memory space.

Local Reference Space

When a function is called, a new *local reference space* is created, which contains all of the local variables and actual parameters. In most languages, local variables are created on the stack, as stack-dynamic variables, but if the variable is labelled as static, it must be assigned statically.

Lecture - ADTs and Concurrency

14:00 26/04/24 Jiacheng Tan

Abstraction

Abstraction means separating the interface of a thing from the details of its implementation, such that the programmer only needs to know how to use it, but not how it does its job. This improves readability, maintainability, re-usability and security of software and libraries.

Modern languages typically provide two types of abstraction– data abstraction and process/ control abstraction.

Data Abstraction

Data Abstraction enforces a clear separation between the abstract properties of a data type, and the concrete details of its implementation. The abstract properties are visible to *client code* (any code which makes use of the abstracted item) which makes use of the data type. This includes information about methods which can be used to manipulate the data, and how items are arranged on a stack. The concrete details of the implementation are kept entirely private, and can change version-to-version without affecting any client code.

Data abstraction is achieved in most languages by defining an Abstract Data Type (ADT)– user defined data types which consist of a set of data, and a set of operations which can be performed upon the data. ADTs are often used to implement a data structure, which acts as a representation of the data and provides implementations for the operations which are allowed on the data. This includes some structures included in languages, such as lists, dictionaries, and more.

ADT Interfaces

An *interface* is a method of interaction between two parties– the implementer of the ADT, and the user of the ADT. The implementer is responsible for the code which performs the operations in a correct and efficient way, and the user is responsible only for the code which interacts with the interface of the ADT. The interface defines the representation of the ADT which the user will actually interact with, but abstracts away the implementation details.

When implementing a new ADT, you must select a number of *core operations* which users will need to interact with your data type. The standard set of operations which almost all ADTs need are: a method to add an item; remove an item; find or retrieve an item. There are also some operations which some ADTs will not need, but are typically implemented anyway, such as checking empty/ fullness, retrieving a subset, etc.

Another thing to consider when implementing a new ADT is how you will actually store the data. For example, you need to select an *internal storage container*, which is used to actually hold the items in the data structure. Users should not need to know or be able to interfere with the internal representation of the data. If users actually need to interact with the data, *accessors* and *mutators* should be used.