University of Portsmouth
BSc (Hons) Computer Science
Second Year

Programming Applications and Programming Languages

(PAAPL)
M30205
September 2023 - June 2024
20 Credits

Thomas Boxall
up2108121@myport.ac.uk

Contents

I Programming Applications 2

II Programming Languages 4

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

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

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

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

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

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

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

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

Teaching Block I

Programming Applications

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

Teaching Block II

Programming Languages

Page 1

Lecture - Introduction To Progrmaming Languages

📅 2024-01-26 🕘 1400 🎓 Jaicheng

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

1.1 Programming Domains

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

1.1.1 Scientific Applications

Scientific Applications of a programming language would be to do mathematical operations on some data which would result in an output. This could be used in applications such as weather forecasting where data on the current weather is fed into it - and a simulation is used to simulate the future weather conditions. Scientific applications will complete a large number of floating-point computations and use arrays. An example of a language in this domain is *Fortran* (FORmula TRANslating system, created by IBM).

1.1.2 Business Applications

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

1.1.3 Artificial Intelligence

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

1.1.4 Systems Programming

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

1.1.5 Web Software

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

1.2 Language Categories

1.2.1 Machine Languages

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

1.2.2 Assembly Languages

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

1.2.3 High Level Language

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

1.2.4 Systems Programming Language

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

1.2.5 Scripting Languages

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

1.2.6 Domain Specific Languages

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

1.3 Categories by Paradigm

There are three different categories.

1.3.1 Procedural

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

1.3.2 Functional

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

1.3.3 Logic

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

1.4 Categories by How Tasks are Specified

1.4.1 Imperative Languages

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

1.4.2 Declarative Languages

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

Page 2

Lecture - Overview and Evaluation of Programming Languages

📅 2024-01-26 🕑 14:00 🎓 Jaicheng

2.1 The 'TPK' Algorithm

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

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

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

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

2.2 Fortran

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

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

The TPK algorithm in Fortran is shown below:

```
C THE TPK ALGORITHM IN FORTRAN
  FUNF(T)=SQRTF(ABSF(T))+5.0*T**3
  DIMENSION A(11)
```

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

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

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

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

7-72 is the statement field

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

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

2.3 COBOL

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

2.4 Algol

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

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

```
begin
  comment TPK algorithm in Algol 60;
  integer i; real y; real array a[0:10];
  real procedure f(t); real t; value t;
    f := sqrt(abs(t))+5*t^3;
  for i := 0 step 1 until 10 do
    read(a[i]);
  for i := 10 step -1 until 0 do begin
    y := f(a[i]);
    if y > 400 then
      write(i, "TOO LARGE")
    else
      write(i, y);
  end
end
```

2.5 Pascal

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

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

2.6 Systems Programming: C

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

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

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

2.7 Object Oriented Languages

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

2.8 Scripting Language

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

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

2.9 Logical Programming (Prolog)

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

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

2.10 Language Evaluation Criteria

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

2.10.1 Simplicity

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

2.10.2 Lexical Elements

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

2.10.3 Orthogonality

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

2.10.4 Data Types

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

2.10.5 Expressiveness

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

Page 3

Lecture - High Level Language Implementation

📅 2024-02-02 🕑 14:00 🎓 Jiacheng

3.1 Computers & Languages

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

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

3.2 Language Implementation Methods

3.2.1 Compilation

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

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

3.2.1.1 Lexical Analysis

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

3.2.1.2 Symbol Table

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

3.2.1.3 Syntax Analysis (Parsing)

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

3.2.1.4 Semantic Analysis

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

3.2.1.5 Code Optimisation

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

3.2.1.6 Code Generation

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

3.2.2 Pure Interpretation

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

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

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

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

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

3.2.3 Hybrid Implementation Systems

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

3.2.4 Just In Time Implementation

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

Page 4

Lecture - Lexical Analysis - Regular Expressions

📅 2024-02-05 🕘 1400 🎓 Jiacheng

4.1 Programming Language Definition

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

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

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

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

4.1.1 Language Analysis

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

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

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

4.2 Lexical Analysis

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

4.2.1 Definitions: Alphabet

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

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

4.2.2 Definitions: Strings

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

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

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

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

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

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

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

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

4.3 Regular Expressions

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

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

4.3.1 Definitions

4.3.1.1 Definition 1

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

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

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

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

4.3.1.2 Definition 2

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

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

4.3.1.3 Definition 3

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

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

4.3.1.4 Definition 4

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

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

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

4.3.1.5 Definition 5

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

4.3.1.6 Definition 6

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

4.3.2 Examples

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

4.3.3 Precedence

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

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

4.3.4 Using Regular Expressions for Lexical Analysis

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

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

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

4.3.5 Regular Definitions

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

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

4.4 Definition of Lexical Languages

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

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

For example:

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

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

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

4.4.1 Decidability of Languages

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

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

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

4.4.2 Regular Expressions and Decision Procedures

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

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

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

Page 5

Lecture - Lexical Analysis - Deterministic Finite Automaton

📅 2024-02-09 🕘 14:00 🎓 Jiacheng

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

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

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

5.1 Finite State Automaton

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

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

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

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

5.1.1 DFA for Lexical Analysis

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

5.2 Transition Diagrams

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

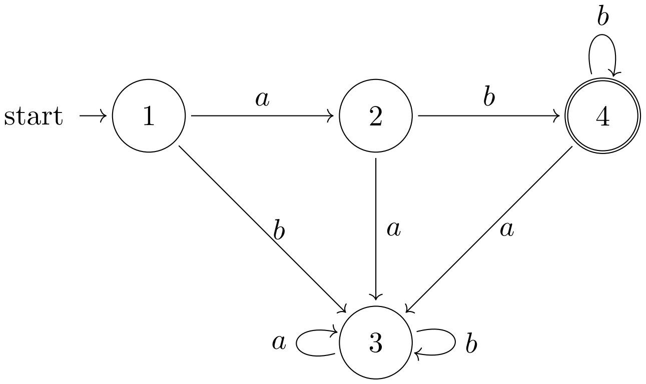

Figure 5.1: DFA Example 1

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

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

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

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

could be used to represent the DFA shown above.

5.2.1 A More Mathematical Example...

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

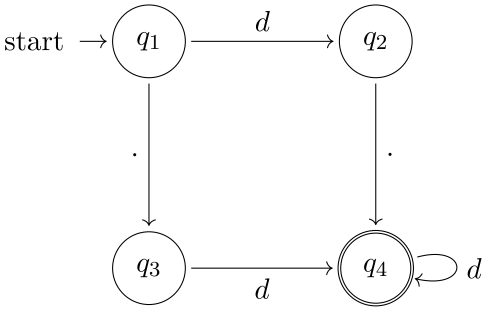

Figure 5.2: DFA Example 2

The DFA above is defined as:

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

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

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

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

5.3 Language of a DFA

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

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

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

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

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

5.3.1 Simplification to DFA Diagrams

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

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

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

Therefore, the regular expression

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

can be represented with the following DFA

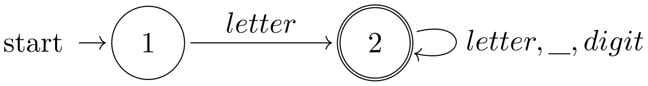

Figure 5.3: Example DFA 3

5.3.2 A More Sophisticated Example

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

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

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

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

5.4 Building a Lexical Analyser

Lexical analysers tend to be built in three ways:

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

or, design DFAs that describe the token patterns, then

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

There are algorithms that construct lexical analysers from DFAs automatically.

5.4.1 Lex

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

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

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

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

```
pattern                 action
```

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

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

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

Page 6

Lecture - Describing Language Syntax

📅 2024-02-16 🕘 14:00 🎓 Jiacheng

6.1 Context-Free Grammars

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

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

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

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

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

6.1.1 What is A CFG?

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

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

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

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

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

6.1.2 Example of a CFG

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

$$
\begin{aligned}
T &= \{a, b\} \\
N &= \{S\} \\
P &= \{S \to ab, S \to aSb\}
\end{aligned}
$$

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

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

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

6.1.3 Shorthand Notation

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

$$\begin{aligned} S &\to \varepsilon|C|aSa|bSb \\ C &\to a|b \end{aligned}$$

6.1.4 Symbol Choice Conventions

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

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

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

6.2 Backus-Naur Form (BNF)

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

6.2.1 Syntactic Structures

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

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

<number> → <digit> | <digit> <number>

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

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

6.2.2 Grammar for a Simple Language

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

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

6.3 Derivations

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

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

6.3.1 An Example

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

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

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

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

6.3.2 Notation

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

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

$$\alpha \Rightarrow \beta$$

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

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

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

6.3.3 Full Derivation Example

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

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

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

We can derive a single line program:

`a = b + const`

from this grammar:

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

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

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

6.3.4 Leftmost & Rightmost Derivations

Considering the following grammar for arithmetic expression

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

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

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

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

It is also possible to have a rightmost derivation:

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

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

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

6.4 Parse Trees

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

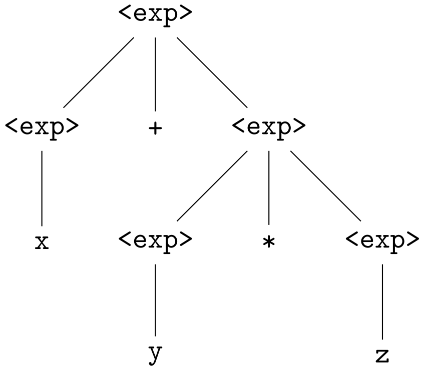

Figure 6.1: Parse Tree for derivation of `x + y * z`

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

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

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

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

Page 7

Lecture - Syntax Analysis and Parsing

📅 2024-02-19 🕑 1400 🎓 Jiacheng

7.1 Ambiguities in Grammars

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

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

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

However, we can first apply the rule:

```
<exp> → <exp> * <exp>
```

which would yield:

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

These two derivations would produce different parse trees:

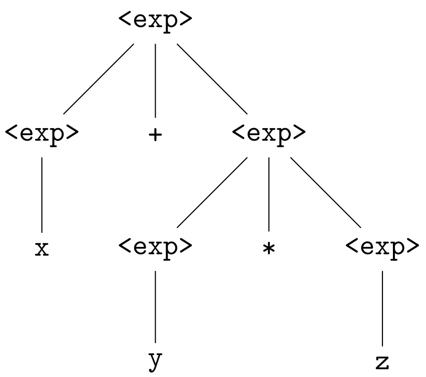

Figure 7.1: Parse Tree for derivation of `x + y * z`

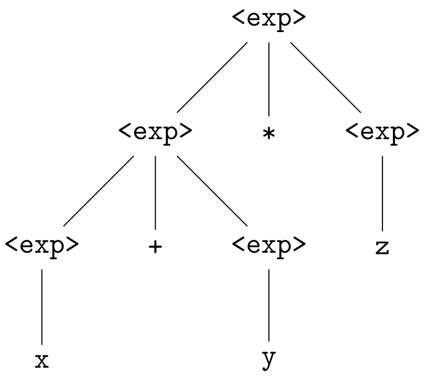

Figure 7.2: Parse Tree for alternate derivation of `x + y * z`

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

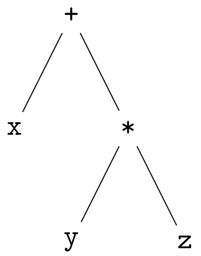

Figure 7.3: Example of an Abstract Syntax Tree

7.1.1 Avoiding Ambiguity

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

7.1.2 Removing Ambiguity

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

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

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

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

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

7.1.3 Limits of Context-Free Grammars

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

7.2 Syntax Analyser

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

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

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

7.2.1 An Example

If we take the grammar:

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

the source code:

```
x + y * z
```

It will produce the tokens outputted by the scanner:

```
IDENT PLUS IDENT MULTI IDENT
```

And the following Parse Tree:

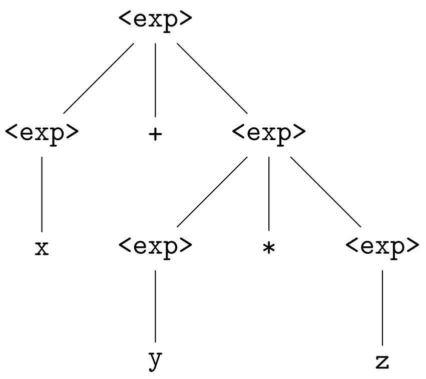

Figure 7.4: Example Parse Tree for derivation of `x + y * z`

With the following abstract syntax tree:

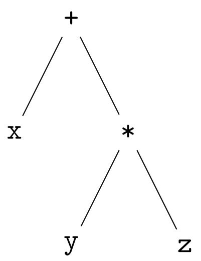

Figure 7.5: AST For example

7.3 Parsers

There are two categories of parsers.

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

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

7.3.1 Parsing: An Example

If we consider the following grammar:

$$\begin{aligned} S &\to AB \\ A &\to aA|\varepsilon \\ B &\to b|bB \end{aligned}$$

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

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

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

Shown below is the full parsing for the string:

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

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

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

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

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

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

$aaa\boldsymbol{B}$

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

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

At each step of top-down parsing:

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

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

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

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

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

Predictive Parsers

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

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

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

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

7.4 Recursive-Descent Parsers (RDP)

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

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

7.4.1 RDP Example

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

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

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

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

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

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

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

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

7.4.2 Rules with Multiple Right Hand Sides

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

Page 8

Lecture - LL1 Parsers

📅 2024-02-23 🕑 1400 🎓 Jiacheng

8.1 Top Down Parsing: Recursive Descent Parsers

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

8.2 Left Recursion Rules

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

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

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

8.2.1 Left Recursion Removal

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

1. Group the A-rules as: $A \rightarrow A\alpha_1|\dots A\alpha_m|\beta_1|\beta_2|\dots|\beta_n$ where $A \rightarrow A\alpha_1|\dots|A\alpha_m$ are rules with left recursion and $A \rightarrow \beta_1|\beta_2|\dots|\beta_n$ are rules without left recursion.
2. Introduce a new non-terminal, A', and replace the original rules with:
$$A \rightarrow \beta_1 A'|\beta_2 A'|\dots|\beta_n A'$$
$$A' \rightarrow \alpha_1 A'|\alpha_2 A'|\dots|\alpha_m A'|\varepsilon$$

8.3 Top Down Parsing: LL(1) Parsers

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

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

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

8.3.1 Example

If we consider the following grammar:

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

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

8.3.2 Parsing Table

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

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

8.3.3 Parsing Procedure / Algorithm

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