MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Programming Languages (Part 2)

Daniel Playne
d.p.playne@massey.ac.nz

Programming Languages

Models of computation and *programming paradigms* are useful when we encounter a new programming language.

They can help us to understand programming languages and guide our expectations of how a particular language will behave and what it will/won't provide (and why).

But what actually makes up a programming language?

Language Structure

In a ***natural language*** we usually think in terms of lexical, syntactic and semantic layers.

- ***Lexical*** layer - concerns the vocabulary (set) of words.
- ***Syntax*** layer - says how words are allowed to be strung together to make sentences.
- ***Semantic*** layer - addresses the meaning of the language.

Programming languages follow a similar structure.

Language Structure

Programming language also have the same lexical, syntactic and semantic layers but also usually include a context layer.

- ***Lexical*** layer - the keywords, symbols allowed in the language.
- ***Syntax*** layer - how keywords/symbols are structured in a valid program.
- ***Context*** layer - rules about allowable programs (e.g. type checking).
- ***Semantic*** layer - addresses the meaning of the program

Language Structure - Lexical Layer

Lexical Layer

Language Structure - Lexical Layer

The ***Lexical Layer*** in a natural language represents the words (nouns, verbs etc) of the language.

In programming languages, these are special reserved keywords such as **BEGIN**, **END**, **FOR** and includes symbols such as **; { }** etc.

Rules defining valid identifier (variable names) would also be defined in the lexical layer.

Lexical Structure

Lexical Symbols / Units / Tokens

- ***Identifiers*** - variable or function names - `mName`
- ***Keywords*** - reserved words - `for`, `while`, `if`
- ***Operators*** - special symbols - `+, *`
- ***Separators*** - more special symbols - `;`
- ***Literals*** - hard-wired constants - `123, "abc"`
- ***Comments*** - removed before compilation
- ***Layout & Whitespace*** - breaks input into tokens

Lexical Analysis

Lexical analysers are essentially pattern matchers, given an input of characters the pattern matcher attempts to find a substring that matches a known pattern.

The lexical analysis groups characters into tokens which are then passed to the rest of the syntax analysis.

Technically lexical analysis is a part of the syntax analysis at the lowest level.

Lexical Analysis

A ***lexical analyser*** or ***lexer*** is a program that takes a string of characters as input and converts the characters into a string of ***tokens***.

Lexical analysis can greatly simplify the task of writing the parser (syntax analyser) as it need only consider a sequence of ***tokens*** rather than reading raw characters.

Lexers are often written using ***regular expressions***.

Lexical Analysis - Regular Expressions

Regular expression syntax:

- `a` - the single character '`a`'
- ϵ - empty, nothing
- **s**|**t** - either the expression **s** or the expression **t**
- **st** - the expression **s** followed by **t**
- **s*** - zero or more of the expression **s**.
- **s**$^+$ - one or more of the expression **s**.
- **s?** - either the expression **s** or nothing.
- `[abc]` - any of the characters '`a`', '`b`' or '`c`'
- `[a-z]` - any of the characters from '`a`' to '`z`'

*some of these are shorthand that simplify the expressions.

Lexical Analysis - Keywords

Keywords are pretty easy to define as patterns, the pattern is simply the keyword itself.

if = `if`
for = `for`

Other tokens such as a variable or function name (often called identifiers) require more general patterns. The first step is to determine the rules of what makes a valid identifier in the language.

Lexical Analysis - Identifiers

A commonly used rule is that an identifier must start with an underscore, a lowercase letter or an uppercase letter and can then be followed by any number of underscores, letters (uppercase or lowercase) and digits.

This can be written as follows:

identifier = `[_a-zA-Z]` `[_a-zA-Z0-9]`*

Lexical Analysis - Constants

The pattern for integer constants are a series of digits (there must be at least one).

integer $=$ `[0-9]`$^+$

Depending on the definition, the sign in front of the integer may be included.

integer $=$ `[-+]?[0-9]`$^+$

Lexical Analysis - Constants

How would you write a pattern for floats?

Lexical Analysis - Constants

How would you write a pattern for floats?

float $=$ `[-+][0-9]`$^+$`.[0-9]`$^+$`[eE][-+]?[0-9]`$^+$

This definition has compulsory sign, integer part, fraction and exponent.

Lexical Analysis - Constants

How would you write a pattern for floats?

float = `[-+][0-9]`$^+$`.[0-9]`$^+$`[eE][-+]?[0-9]`$^+$

This definition has compulsory sign, integer part, fraction and exponent.

float = `[-+]?[0-9]`$^+$`((.[0-9]`$^+$`([eE][-+]?[0-9]`$^+$`)?)` | `(.?[eE][-+]?[0-9]`$^+$`))`

A more general version that has an optional sign, a compulsory integer part and optional fraction and exponent parts (but must have at least one).

Language Structure - Syntax Layer

Syntax Layer

Syntax Layer

The syntax layer defines what order tokens are allowed to appear in.

These syntax rules are (usually) designed to be close to natural language constructs to make programming languages as easy to understand as possible.

e.g. `if ... then ... else` construct.

Syntax Analysis

Syntax analysis is called ***parsing***. A parser takes a stream of tokens from the lexical analyser and performs two important functions:

- First it checks the input against the rules of the language to determine if the input program is correct or not.
- Second it will construct a parse tree (or at least traverse a parse tree) which is used to generate the output.

Syntax Analysis

The ***syntax rules*** of programming language are usually written as a ***context-free grammar***.

Extended Backus-Naur Form (EBNF) is one commonly used way to express a context-free grammar. A grammar written in EBNF consists of terminal symbols and non-terminal production rules which describe how terminal symbols can be legally combined.

Context-free grammars mean that production rules can be applied regardless of the context (other preceding or surrounding symbols).

Extended Backus-Naur Form

EBNF can be used to write a grammar which can be used to define a language (used to work out whether an input program is valid or not).

There are several different variants of EBNF which use some different notation (this can be confusing). We will use:

`=`	is defined as
`|`	alternation (or)
`[...]`	optional
`{...}`	repetition
`(...)`	grouping
`"..."`	terminal string

Extended Backus-Naur Form

Using this notation we could write a simple grammar (note that in this case we are defining terminal strings rather than tokens from a lexer*).

```
program = "program" { statement } "end"
statement = assignment | loop
assignment = identifier "=" expression ";"
loop = "while" expression "do" { statement } "done"
expression = value | value "+" value | value "<=" value
value = identifier | number
identifier = letter { letter | digit}
number = { digit }
digit = "0" | "1" | .. | "8" | "9"
letter = "a"|"b"|..|"y"|"z"|"A"|"B"|..|"Y"|"Z"
```

*Using a lexer can often simplify the process of writing a grammar.

Extended Backus-Naur Form

This EBNF grammar defines whether a program is valid or not. It also specifies what a compiler or interpreter needs to make sense of a program.

```
program
    n = 1;
    while n <= 10 do
        n = n + 1;
    done
end
```

Consider this example program, does it conform to the EBNF grammar from the previous slide?

Extended Backus-Naur Form

If the rules of our grammar can produce the same sequence of characters as the input, then the grammar accepts that input (it is a valid program).

```
program
"program" { statement } "end"
"program" statement statement "end"
"program" assignment loop "end"
"program" identifier "=" expression ";" loop "end"
"program n =" value ";" loop "end"
"program n =" number ";" loop "end"
"program n =" { digit } ";" loop "end"
"program n = 1;" loop "end"
"program n = 1; while" expression "do" { statement } "done end"
"program n = 1; while" value "<=" value "do" { statement } "done end"
"program n = 1; while" identifier "<=" number "do" { statement } "done end"
"program n = 1; while n <= 10 do" { statement } "done end"
"program n = 1; while n <= 10 do" assignment "done end"
"program n = 1; while n <= 10 do" identifier "=" expression "; done end"
"program n = 1; while n <= 10 do n =" value "+" value "; done end"
"program n = 1; while n <= 10 do n =" identifier "+" number "; done end"
"program n = 1; while n <= 10 do n = n +" number "; done end"
"program n = 1; while n <= 10 do n = n +" { digit } "; done end"
"program n = 1; while n <= 10 do n = n + 1; done end"
```

Syntax Analysis

If the input program matches the grammar then it is considered to be a valid program.

Most of the error messages your compiler gives you are generated during the parsing process (they're easy to detect).

As you well know, just because a program compiles doesn't mean it can run successfully or accomplish what you want it to.

Summary

- Programming Language Layers
  - Lexical
  - Syntax
  - Context
  - Semantic
- Lexical Analysis
  - Regular Expressions
- Syntax Analysis
  - Extended Backus-Naur Form