Chapter 3

(a) Syntax

Introduction

We usually break down the problem of defining a programming language into two parts.

- defining the PL’s syntax
- defining the PL’s semantics

Syntax - the form or structure of the expressions, statements, and program units

Semantics - the meaning of the expressions, statements, and program units

Note: There is not always a clear boundary between the two.

Some Preliminaries

- For the next several weeks we’ll look at how one can define a programming language.
- What is a language, anyway?
  Language is a system of gestures, grammar, signs, sounds, symbols, or words, which is used to represent and communicate concepts, ideas, meanings, and thoughts.
- Human language is a way to communicate representations from one (human) mind to another
- What about a programming language?
  A way to communicate representations (e.g., of data or a procedure) between human minds and/or machines.

Why and How

Why? We want specifications for several communities:
- Other language designers
- Implementers
- Machines?
- Programmers (the users of the language)

How? One ways is via natural language descriptions (e.g., user’s manuals, text books) but there are a number of techniques for specifying the syntax and semantics that are more formal.
This is an overview of the standard process of turning a text file into an executable program.

Syntax Overview

- Language preliminaries
- Context-free grammars and BNF
- Syntax diagrams

Introduction

A sentence is a string of characters over some alphabet.

A language is a set of sentences.

A lexeme is the lowest level syntactic unit of a language (e.g., *, sum, begin).

A token is a category of lexemes (e.g., identifier).

Formal approaches to describing syntax:

1. Recognizers - used in compilers
2. Generators - what we'll study

Lexical Structure of Programming Languages

- The structure of its lexemes (words or tokens)
  - token is a category of lexeme
- The scanning phase (lexical analyser) collects characters into tokens
- Parsing phase(syntactic analyser) determines syntactic structure
Grammars

Context-Free Grammars
- Developed by Noam Chomsky in the mid-1950s.
- Language generators, meant to describe the syntax of natural languages.
- Define a class of languages called context-free languages.

Backus Normal/Naur Form (1959)
- Invented by John Backus to describe Algol 58 and refined by Peter Naur for Algol 60.
- BNF is equivalent to context-free grammars.

BNF (continued)

A *metalanguage* is a language used to describe another language.

In BNF, *abstractions* are used to represent classes of syntactic structures -- they act like syntactic variables (also called *nonterminal symbols*), e.g.

```
<while_stmt> ::= while <logic_expr> do <stmt>
```

This is a *rule*; it describes the structure of a while statement.

BNF

- A rule has a left-hand side (LHS) which is a **single** non-terminal symbol and a right-hand side (RHS), one or more *terminal* or *nonterminal* symbols.
- A *grammar* is a finite, nonempty set of rules.
- A non-terminal symbol is “defined” by its rules.
- Multiple rules can be combined with the | symbol (read as “or”)
- These two rules:
  
  ```
  <stmts> ::= <stmt> 
  <stmts> ::= <stmnt> ; <stmts> 
  ```
  
  are equivalent to this one:
  
  ```
  <stmts> ::= <stmt> | <stmnt> ; <stmts> 
  ```
Non-terminals, pre-terminals & terminals

- A non-terminal symbol is any symbol that is in the RHS of a rule. These represent abstractions in the language (e.g., if-then-else-statement in)
  
  `if-then-else-statement ::= if `<test>` then `<statement>` else `<statement>``

- A terminal symbol is any symbol that is not on the LHS of a rule. AKA lexemes. These are the literal symbols that will appear in a program (e.g., `if`, `then`, `else` in rules above).

- A pre-terminal symbol is one that appears as a LHS of rule(s), but in every case, the RHSs consist of single terminal symbol, e.g., `<digit>` in
  
  `digit ::= 0 | 1 | 2 | 3 ... 7 | 8 | 9`

BNF

- Repetition is done with recursion

- E.g., Syntactic lists are described in BNF using recursion

- An `<ident_list>` is a sequence of one or more `<ident>`s separated by commas.

  `<ident_list> ::= <ident> | <ident>, <ident_list>`

BNF Example

Here is an example of a simple grammar for a subset of English.

A sentence is noun phrase and verb phrase followed by a period.

  `<sentence> ::= <nounPhrase> <verbPhrase> .`
  `<nounPhrase> ::= <article> <noun>`
  `<article> ::= a | the`
  `<noun> ::= man | apple | worm | penguin`
  `<verbPhrase> ::= <verb>|<verb><nounPhrase>`
  `<verb> ::= eats | throws | sees | is`

Derivations

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence consisting of just all terminal symbols.

- It demonstrates, or proves that the derived sentence is “generated” by the grammar and is thus in the language that the grammar defines.

- As an example, consider our baby English grammar

  `<sentence> ::= <nounPhrase><verbPhrase>.`
  `<nounPhrase> ::= <article><noun>`
  `<article> ::= a | the`
  `<noun> ::= man | apple | worm | penguin`
  `<verbPhrase> ::= <verb>|<verb><nounPhrase>`
  `<verb> ::= eats | throws | sees | is`
Derivation using BNF

Here is a derivation for “the man eats the apple.”

\[ \text{<sentence>} \rightarrow \text{<nounPhrase><verbPhrase>} \]

\[ \text{<article><noun><verbPhrase>} \]

\[ \text{the<noun><verbPhrase>} \]

\[ \text{the man <verbPhrase>} \]

\[ \text{the man <verb><nounPhrase>} \]

\[ \text{the man eats <nounPhrase>} \]

\[ \text{the man eats <article> < noun>} \]

\[ \text{the man eats the <noun>} \]

\[ \text{the man eats the apple} \]

Derivation

Every string of symbols in the derivation is a sentential form.

A sentence is a sentential form that has only terminal symbols.

A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded.

A derivation may be either leftmost or rightmost or something else.

Another BNF Example

\[ \text{<program>} \rightarrow \text{<stmts>} \]

\[ \text{<stmts>} \rightarrow \text{<stmt>} \]

\[ | \text{<stmt>} ; \text{<stmts>} \]

\[ \text{<stmt>} \rightarrow \text{<var>} = \text{<expr>} \]

\[ \text{<var>} \rightarrow \text{a} | \text{b} | \text{c} | \text{d} \]

\[ \text{<expr>} \rightarrow \text{<term>} + \text{<term>} | \text{<term>} - \text{<term>} \]

\[ \text{<term>} \rightarrow \text{<var>} | \text{const} \]

Here is a derivation:

\[ \text{<program>} \Rightarrow \text{<stmts>} \]

\[ \Rightarrow \text{<stmt>} \]

\[ \Rightarrow \text{<var>} = \text{<expr>} \]

\[ \Rightarrow \text{a} = \text{<expr>} \]

\[ \Rightarrow \text{a} = \text{<term>} + \text{<term>} \]

\[ \Rightarrow \text{a} = \text{<var>} + \text{<term>} \]

\[ \Rightarrow \text{a} = \text{b} + \text{<term>} \]

\[ \Rightarrow \text{a} = \text{b} + \text{const} \]

Finite and Infinite languages

- A simple language may have a finite number of sentences.
- An finite language is the set of strings representing integers between -10**6 and +10**6
- A finite language can be defined by enumerating the sentences, but using a grammar might be much easier.
- Most interesting languages have an infinite number of sentences.
Is English a finite or infinite language?

- Assume we have a finite set of words
- Consider adding rules like the following to the previous example

\[
\begin{align*}
\text{<sentence>} &::= \text{<sentence>}\text{<conj>\text{<sentence>}}. \\
\text{<conj>} &::= \text{and} | \text{or} | \text{because}
\end{align*}
\]

- Hint: Whenever you see recursion in a BNF it’s a sign that the language is infinite.
  – When might it not be?

Parse Tree

A parse tree is a hierarchical representation of a derivation

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

Another Parse Tree

Grammar

A grammar is ambiguous iff it generates a sentential form that has two or more distinct parse trees.

Ambiguous grammars are, in general, very undesirable in formal languages.

We can eliminate ambiguity by revising the grammar.
An ambiguous grammar

Here is a simple grammar for expressions that is ambiguous

\[ <expr> \rightarrow <expr> <op> <expr> \]
\[ <expr> \rightarrow \text{int} \]
\[ <op> \rightarrow + | - | * | / \]

The sentence \(1+2*3\) can lead to two different parse trees corresponding to \(1+(2*3)\) and \((1+2)*3\)

Operators

- The traditional operator notation introduces many problems.
- Operators are used in
  - Prefix notation: E.g. Expression \((* (+ 1 3) 2)\) in Lisp
  - Infix notation: E.g. Expression \((1 + 3) * 2\) in Java
  - Postfix notation: E.g. Increment \(foo++\) in C
- Operators can have 1 or more operands
  - Increment in C is a one-operand operator: \(foo++\)
  - Subtraction in C is a two-operand operator: \(foo - bar\)
  - Conditional expression in C is a three-operand operator: \((foo == 3 ? 0 : 1)\)

Operator notation

- So, how do we interpret expressions like
  1. \(2 + 3 + 4\)
  2. \(2 + 3 * 4\)
- While you might argue that it doesn’t matter for (a), it can for different operators \(2 ** 3 ** 4\) or when the limits of representation are hit (e.g., round off in numbers, e.g., \(1+1+1+1+1+1+1+1+1+1+1+10**6\))
- Concepts:
  - Explaining rules in terms of operator precedence and associativity.
  - Realizing the rules in grammars.

Operators: Precedence and Associativity

- **Precedence and associativity** deal with the evaluation order within expressions
- **Precedence** rules specify the order in which operators of different precedence level are evaluated, e.g.:
  - \(**\) Has a higher precedence than \(\+\), so \(**\) groups more tightly than \(\+\)
- What is the results of \(4 * 5 ** 6\)?
- A language’s precedence hierarchy should match our intuitions, but the result’s not always perfect, as in this Pascal example:
  - \(if A<B \ and \ C<D \ then \ (*ouch*)\)
- Pascal’s relational operators have the lowest precedence!
### Operator Precedence: Precedence Table

<table>
<thead>
<tr>
<th>Fortran</th>
<th>Preval</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -- (post-inc., dec.)</td>
<td>++, -- (prec-incre, dec.)</td>
<td>abs (absolute)</td>
<td>abs (absolute)</td>
</tr>
<tr>
<td>**</td>
<td>not</td>
<td>+, - (Unary) &amp; (address of), value.</td>
<td>* (contents of), ! (logical not), not, ++</td>
</tr>
<tr>
<td>*, /</td>
<td>*, /, div, mod, * (binary), /, % (modulo division)</td>
<td>*, /, mod, rem</td>
<td>*, /, mod, rem</td>
</tr>
<tr>
<td>t, -</td>
<td>t, - (Unary and Binary), or</td>
<td>+, - (Binary)</td>
<td>+, - (Binary)</td>
</tr>
<tr>
<td>2, &lt;, &gt;, &lt;=, &gt;=</td>
<td>&lt;=&gt;, &lt;&gt;, &lt;=, &gt;=</td>
<td>&lt;=, &gt;=</td>
<td>&lt;=, &gt;=</td>
</tr>
<tr>
<td>.not.</td>
<td>==, != (equality tests)</td>
<td>==, != (equality tests)</td>
<td>==, != (equality tests)</td>
</tr>
</tbody>
</table>

### Operators: Associativity

- **Associativity** rules specify the order in which operators of the same precedence level are evaluated.
- Operators are typically either left associative or right associative.
- Left associativity is typical for +, - , * and /.
- So A + B + C
  - Means: (A + B) + C
  - And not: A + (B + C)
- **Does it matter?**
Precedence and associativity in Grammar

If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity.

An unambiguous expression grammar:

\[
<expr> \rightarrow <expr> - <term> \mid <term> \\
<term> \rightarrow <term> / const \mid const
\]

Grammar (continued)

Operator associativity can also be indicated by a grammar:

\[
<expr> \rightarrow <expr> + <expr> \mid const \text{ (ambiguous)} \\
<expr> \rightarrow <expr> + const \mid const \text{ (unambiguous)}
\]

Precedence and associativity in Grammar

Sentence: const – const / const

Derivation:
\[
<expr> \Rightarrow <expr> - <term> \\
\Rightarrow <term> - <term> \\
\Rightarrow const - <term> \\
\Rightarrow const - const / const
\]

Parse tree:

An Expression Grammar

Here's a grammar to define simple arithmetic expressions over variables and numbers.

\[
\text{Exp ::= num} \\
\text{Exp ::= id} \\
\text{Exp ::= UnOp Exp} \\
\text{Exp ::= }'(\text{ Exp }') \\
\text{UnOp ::= }'+' \\
\text{UnOp ::= }'-' \\
\text{BinOp ::= }'+' | '-' | '*' | '/'
\]

Here's another common notation variant where single quotes are used to indicate terminal symbols and unquoted symbols are taken as non-terminals.
A derivation

Here’s a derivation of a+b*2 using the expression grammar:

\[
\begin{align*}
\text{Exp} &\Rightarrow & \text{Exp} ::= \text{Exp} \text{ BinOp} \text{ Exp} \\
\text{Exp BinOp Exp} &\Rightarrow & \text{Exp} ::= \text{id} \\
\text{id BinOp Exp} &\Rightarrow & \text{BinOp} ::= '+' \\
\text{id + Exp} &\Rightarrow & \text{Exp} ::= \text{Exp BinOp Exp} \\
\text{id + Exp BinOp Exp} &\Rightarrow & \text{Exp} ::= \text{num} \\
\text{id + Exp BinOp num} &\Rightarrow & \text{Exp} ::= \text{id} \\
\text{id + id BinOp num} &\Rightarrow & \text{BinOp} ::= '*' \\
\text{id + id * num} &\Rightarrow & a + b * 2
\end{align*}
\]

A parse tree

A parse tree for a+b*2:

\[
\begin{align*}
\text{Exp} &\Rightarrow & \text{Exp} ::= \text{id} \\
\text{id + Exp} &\Rightarrow & \text{Exp} ::= \text{num} \\
\text{id + id BinOp num} &\Rightarrow & \text{BinOp} ::= '*' \\
\text{id + id * num} &\Rightarrow & \text{id + id * num} &\Rightarrow & a + b * 2
\end{align*}
\]

Precedence

- Precedence refers to the order in which operations are evaluated.
- Usual convention: exponents > mult div > add sub.
- So, deal with operations in categories: exponents, mulops, addops.
- Here’s a revised grammar that follows these conventions:

\[
\begin{align*}
\text{Exp} &::= \text{Exp AddOp Exp} \\
\text{Exp} &::= \text{Term} \\
\text{Term} &::= \text{Term MulOp Term} \\
\text{Term} &::= \text{Factor} \\
\text{Factor} &::= \text{Factor} + \text{Exp} + \text{Factor} \\
\text{Factor} &::= \text{num} | \text{id} \\
\text{AddOp} &::= '+' | '-' \\
\text{MulOp} &::= '*' | '/'
\end{align*}
\]

Associativity

- Associativity refers to the order in which 2 of the same operation should be computed
  \( -3+4+5 = (3+4)+5, \) left associative (all BinOps)
  \( -3^4^5 = 3^{(4^5)}, \) right associative
- Conditionals right associate but have a wrinkle: an else clause associates with closest unmatched if
  \( \text{if a then if b then c else d} = \text{if a then (if b then c else d)} \)
Adding associativity to the grammar

Adding associativity to the BinOp expression grammar

\[
\begin{align*}
\text{Exp} & \ ::= \text{Exp AddOp Term} \\
\text{Exp} & \ ::= \text{Term} \\
\text{Term} & \ ::= \text{Term MulOp Factor} \\
\text{Term} & \ ::= \text{Factor} \\
\text{Factor} & \ ::= \text{'}(\text{Exp} \text{')}\text{'} \\
\text{Factor} & \ ::= \text{num} \mid \text{id} \\
\text{AddOp} & \ ::= \text{'}+\text{'} \mid \text{'}-\text{'} \\
\text{MulOp} & \ ::= \text{'}\ast\text{'} \mid \text{'}/\text{'}
\end{align*}
\]

Another example: conditionals

- Goal: to create a correct grammar for conditionals.
- It needs to be non-ambiguous and the precedence is else with nearest unmatched if.

\[
\begin{align*}
\text{Statement} & \ ::= \text{Conditional} \mid \text{'}whatever\text{'}
\text{Conditional} & \ ::= \text{'}if\text{' test 'then' Statement 'else' Statement}
\text{Conditional} & \ ::= \text{'}if\text{' test 'then' Statement}
\end{align*}
\]

- The grammar is ambiguous. The 1st Conditional allows unmatched 'if's to be Conditionals.
  - if test then (if test then whatever else whatever) = correct
  - if test then (if test then whatever) else whatever = incorrect

- The final unambiguous grammar.

\[
\begin{align*}
\text{Statement} & \ ::= \text{Matched} \mid \text{Unmatched}
\text{Matched} & \ ::= \text{'}if\text{' test 'then' Matched 'else' Matched}
\text{Unmatched} & \ ::= \text{'}if\text{' test 'then' Statement}
\end{align*}
\]

Extended BNF

Syntactic sugar: doesn’t extend the expressive power of the formalism, but does make it easier to use (more readable and more writable).

Optional parts are placed in brackets ([])  
<proc_call> -> ident [ ( <expr_list>)]

Put alternative parts of RHSs in parentheses and separate them with vertical bars  
<term> -> <term> (+ | -) const

Put repetitions (0 or more) in braces ({})  
<ident> -> letter {letter | digit}
BNF vs EBNF

BNF:

\[<\text{expr}> \rightarrow <\text{expr}> + <\text{term}>\]
| \[<\text{expr}> - <\text{term}>\]
| \[<\text{term}>\]

\[<\text{term}> \rightarrow <\text{term}> * <\text{factor}>\]
| \[<\text{term}> / <\text{factor}>\]
| \[<\text{factor}>\]

EBNF:

\[<\text{expr}> \rightarrow <\text{term}> \{ (+ | - ) <\text{term}>\}\]
\[<\text{term}> \rightarrow <\text{factor}> \{ (* | / ) <\text{factor}>\}\]

Syntax Graphs

Syntax Graphs - Put the terminals in circles or ellipses and put the nonterminals in rectangles; connect with lines with arrowheads

e.g., Pascal type declarations

Provides an intuitive, graphical notation.

Parsing

- A grammar describes the strings of tokens that are syntactically legal in a PL
- A recogniser simply accepts or rejects strings.
- A generator produces sentences in the language described by the grammar
- A parser construct a derivation or parse tree for a sentence (if possible)
- Two common types of parsers:
  - bottom-up or data driven
  - top-down or hypothesis driven
- A recursive descent parser is a way to implement a top-down parser that is particularly simple.

Parsing complexity

- How hard is the parsing task?
- Parsing an arbitrary Context Free Grammar is \(O(n^3)\), e.g., it can take time proportional the cube of the number of symbols in the input. This is bad!
- If we constrain the grammar somewhat, we can always parse in linear time. This is good!
- Linear-time parsing
  - LL parsers
    » Recognize LL grammar
    » Use a top-down strategy
  - LR parsers
    » Recognize LR grammar
    » Use a bottom-up strategy

LL(n) : Left to right, Leftmost derivation, look ahead at most \(n\) symbols.
LR(n) : Left to right, Right derivation, look ahead at most \(n\) symbols.
Recursive Decent Parsing

- Each nonterminal in the grammar has a subprogram associated with it; the subprogram parses all sentential forms that the nonterminal can generate
- The recursive descent parsing subprograms are built directly from the grammar rules
- Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars (why not?)

Recursive Decent Parsing Example

Example: For the grammar:

\[
\langle \text{term} \rangle \rightarrow \langle \text{factor} \rangle \{ (* | / ) \langle \text{factor} \rangle \}
\]

We could use the following recursive descent parsing subprogram (this one is written in C)

```c
void term() {
    factor(); /* parse first factor*/
    while (next_token == ast_code ||
            next_token == slash_code) {
        lexical(); /* get next token */
        factor(); /* parse next factor */
    }
}
```

The Chomsky hierarchy

- The Chomsky hierarchy has four types of languages and their associated grammars and machines.
- They form a strict hierarchy; that is, regular languages < context-free languages < context-sensitive languages < recursively enumerable languages.
- The syntax of computer languages are usually describable by regular or context free languages.
Summary

- The syntax of a programming language is usually defined using BNF or a context free grammar.
- In addition to defining what programs are syntactically legal, a grammar also encodes meaningful or useful abstractions (e.g., block of statements).
- Typical syntactic notions like operator precedence, associativity, sequences, optional statements, etc. can be encoded in grammars.
- A parser is based on a grammar and takes an input string, does a derivation and produces a parse tree.