Syntax

In Text: Chapter 3
Outline

Syntax:
- Recognizer vs. generator
- BNF
- EBNF
Basic Definitions

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

Why write a language definition; who will use it?
- Other language designers
- Implementors (compiler writers)
- Programmers (the users of the language)
What is a “Language”? 

- 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).
Recognizers vs. Generators

We don’t want to use English to describe a language (too long, tedious, imprecise), so...

There are two formal approaches to describing syntax:

- Recognizers
  - Given a string, a recognizer for a language L tells whether or not the string is in L (ex: Compiler)

- Generators
  - A generator for L will produce an arbitrary string in L on demand. (ex: Grammar, BNF)

Recognition and generation are useful for different things, but are closely related
Grammars

- Developed by Noam Chomsky in the mid-1950s
- 4-level hierarchy (0-3)
- Language generators, meant to describe the syntax of natural languages
- Context-free grammars define a class of languages called context-free languages (level 2)
Backus- Naur Form

- Invented by John Backus and Peter Naur to describe syntax of Algol 58/60
- BNF is equivalent to context-free grammars
- A metalanguage: a language used to describe another language
BNF Nonterminals

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

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

- This is a rule; it describes the structure of a while statement
BNF Rules

- A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of terminal and nonterminal symbols.
- A grammar is a finite nonempty set of rules.
- An abstraction (or nonterminal symbol) can have more than one RHS:
  
  \[
  \langle\text{stmt}\rangle \rightarrow \langle\text{single\_stmt}\rangle \\
  \text{ | } \text{begin } \langle\text{stmt\_list}\rangle \text{end}
  \]

- Syntactic lists are described using recursion:

  \[
  \langle\text{ident\_list}\rangle \rightarrow \langle\text{ident}\rangle \\
  \text{ | } \text{ident, } \langle\text{ident\_list}\rangle
  \]
An Example Grammar

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

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols):

\[
\text{<program>} \Rightarrow \text{<stmts>}
\]
\[
\Rightarrow \text{<stmt>}
\]
\[
\Rightarrow \text{<var> = <expr>}
\]
\[
\Rightarrow a = \text{<expr>}
\]
\[
\Rightarrow a = \text{<term>} + \text{<term>}
\]
\[
\Rightarrow a = \text{<var>} + \text{<term>}
\]
\[
\Rightarrow a = b + \text{<term>}
\]
\[
\Rightarrow a = b + \text{const}
\]
Sentential Forms

- 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 next in the derivation.
- A rightmost derivation works right to left instead.
- Some derivations are neither leftmost nor rightmost.
A parse tree is a hierarchical representation of a derivation.

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

```plaintext
<program>
  <stmts>
    <stmt>
      <var> = <expr>
      a <term> + <term>
      <var> const
      b
```
Ambiguous Grammars

An ambiguous expression grammar:

\[
\text{expr} \rightarrow \text{expr} \op \text{expr} \mid \text{const} \\
\op \rightarrow / \mid -
\]

(Vertex diagrams for examples of multiple parses for the same expression.)
If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity:

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

\[
\text{<term>} \rightarrow \text{<term>} / \text{const} \mid \text{const}
\]

\[
\text{<expr>} \Rightarrow \text{<expr>} - \text{<term>}
\]

\[
\Rightarrow \text{<term>} - \text{<term>}
\]

\[
\Rightarrow \text{const} - \text{<term>}
\]

\[
\Rightarrow \text{const} - \text{<term>} / \text{const}
\]

\[
\Rightarrow \text{const} - \text{const} / \text{const}
\]
Operator associativity can also be indicated by a grammar

\(<expr>- \rightarrow <expr> + <expr> | \text{const} \) (ambiguous)

\(<expr>- \rightarrow <expr> + \text{const} | \text{const} \) (unambiguous)
Extended BNF (EBNF)

- Optional parts are placed in brackets ([ ])
  \[ \text{<proc\_call>} \rightarrow \text{ident} [ ( \text{<expr\_list>} ) ] \]

- Put alternative parts of RHS in parentheses and separate them with vertical bars
  \( \text{<term>} \rightarrow \text{<term>} (+ | -) \text{ const} \)

- Put repetitions (0 or more) in braces ( { {} )
  \( \text{<ident>} \rightarrow \text{letter} \{ \text{letter} | \text{digit} \} \)
BNF and EBNF Side by Side

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

- **EBNF:**
  \[ <expr> \rightarrow <term> {(+ | -) <term>} \]
  \[ <term> \rightarrow <factor> {(* | /) <factor>} \]
Recursive Descent Parsing

- Parsing is the process of tracing or constructing a parse tree for a given input string.
- Parsers usually do not analyze lexemes; that is done by a lexical analyzer, which is called by the parser.
- A recursive descent parser traces out a parse tree in top-down order; it is a top-down parser.
- 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.
Recursive Descent Example

- Example: For the grammar:
  - `<term> ::= <factor> {(* | /) <factor>}`

- Simple recursive descent parsing subprogram:
  ```c
  void term() {
    factor(); /* parse the first factor*/
    while (next_token == ast_code ||
            next_token == slash_code) {
      lexical(); /* get next token */
      factor(); /* parse the next factor */
    }
  }
  ```