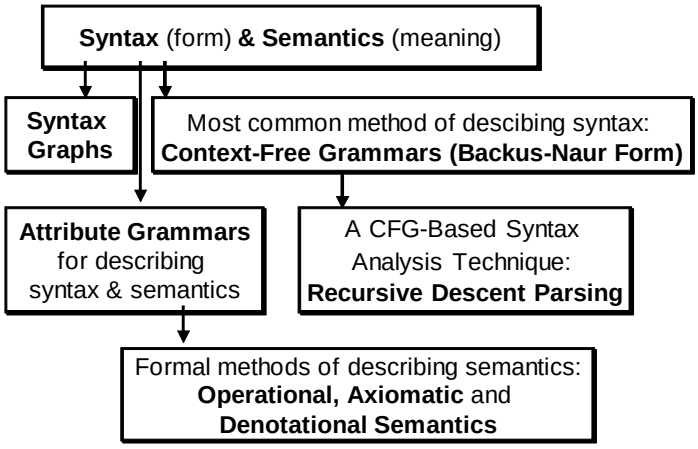
UNIX Systems Programming

Compilers

(Sabesta, chp.3)

Dr. Kivanç Dinçer
CENG-332 Lectures
Spring 2000



Syntax and Semantics

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

Describing syntax is easier than describing semantics.

Ex: An if statement in C language:

```
if ( <expr> ) <statement>
```


The General Problem of Describing Syntax

- A **sentence** is a string of characters over some alphabet.
- A **language** is a set of sentences.
  - Syntax rules specify which sentences are in the language.
- A **lexeme** is the lowest level syntactic unit of a language (e.g., *, sum, begin.)
  - Description of lexemes is given by a lexical specification, and separate from the syntactic description of the lang.
  - Lexemes include identifiers, constants, operators and special words.
- A **token** is a category of lexemes (e.g., identifier, semicolon, or equal_sign) **[Example]**

You can think of programs as strings of lexemes rather than chars

Formal Approaches to Describing Syntax

- **Recognizers** - used in syntax analysis part of compilers
  - A language L that uses alphabet Σ of characters.
  - We construct a recognition device, R, which is capable of
    - inputting strings of chars. from the alphabet Σ and
    - indicating whether a given input string is in L or not.

Backus-Naur Form and Context-Free Grammars

Grammars are formal language generation mechanisms commonly used to describe syntax of PLs.

Context-Free Grammars (CFG) (mid-1950s)

- Developed by Noam Chomsky.
  - Defined a class of languages called **context-free langs.**
- **Context-free grammars** can describe whole languages, with minor exceptions.
- **Regular grammars** can describe langs of tokens of PLs.

Backus-Naur Form (BNF) (1959)

- Invented by John Backus to describe Algol 58.
- BNF is equivalent to context-free grammars.
- BNF is a very natural notation for describing syntax.

Fundamentals

- A **metalanguage** is a language used to describe another language. (ex. BNF is a metalang. for PLs)
- 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** (or production); it describes the structure of a while statement.
- A rule has a **left-hand side** (LHS) and a **right-hand side** (RHS), and consists of **nonterminal** and **terminal** (lexemes and tokens) **symbols**.

- A **grammar** is a finite nonempty set of rules.
- An abstraction (or nonterminal symbol) can have more than one RHS (i.e., definitions):

```
<stmt> -> <single_stmt>
        | begin <stmt_list> end
```

- **Syntactic lists** are described in BNF using recursion:

```
<ident_list> -> ident
              | ident, <ident_list>
```

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

- Each of the strings in the derivation, including start symbol is called a **sentential form**.
  - A **sentence** is a sentential form that has only terminal symbols, or lexemes.
- A **leftmost derivation** is one in which the leftmost nonterminal in each sentential form is the one that is expanded:

```
<term> -> <term> * <factor>
```

- A derivation may be leftmost, rightmost, or neither of them.
  - Derivation order has no effect on the language generated by a grammar.
  - By exhaustively choosing all combinations of alternative RHSs of rules, the entire language can be generated.

Examples

An example grammar for a small language:

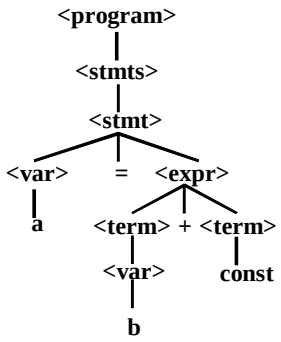
```
<program> -> <stmts>
<stmts> -> <stmt> | <stmt> ; <stmts>
<stmt> -> <var> = <expr>
<var> -> a | b | c | d
<expr> -> <term> + <term> | <term> - <term>
<term> -> <var> | const
```

A derivation of a program in this language:

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


Parse Trees

A **parse tree** is a hierarchical representation of a derivation.



Every leaf is labeled with a terminal symbol.

Extended BNF (EBNF)

Extensions do not enhance the power of BNF but bring abbreviations and increase its readability writability.

1. Place optional parts in brackets: **[ ]**

```
<proc_call> -> ident [ (<expr_list>) ]
```

2. Put alternative parts of RHSs in parentheses and separate them with vertical bars:

```
<term> -> <term> (+ | -) const
```

3. Put repetitions (0 or more) in braces***: { }**

```
<ident> -> letter {letter | digit}
```

{ }⁺ indicates one or more repetions.

This is a replacement of the recursion by a form of implied iteration. Sometimes an ellipsis (. . .) (i.e., more of the same) is used instead:

```
<ident_list> -> <identifier> [,<identifier>]...
```


- Metasymbols: The brackets, braces, and parantheses in the EBNF extensions.
  - Metasymbols are notational tools and not terminal symbols in the syntactic entities they help describe.
  - If these metasymbols are also terminal symbols in the language being described, the instances that are terminal symbols are underlined.

BNF:

```
<expr>  -> <expr> + <term>
         | <expr> - <term>
         | <term>
<term>  -> <term> * <factor>
         | <term> / <factor>
         | <factor>
```

EBNF:

```
<expr>-> <term> {(+|-)<term>}

<term>-><factor>{(*|/)<factor>}
```

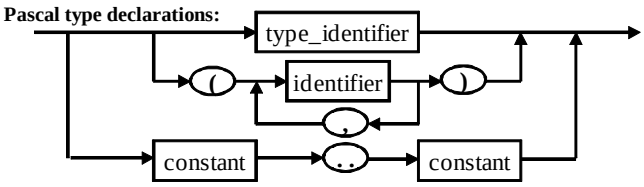

Syntax Graphs

A **graph** is a collection of **nodes**, some of which are connected by lines, called **edges**.

A **directed** graph is one in which the edges are directional.

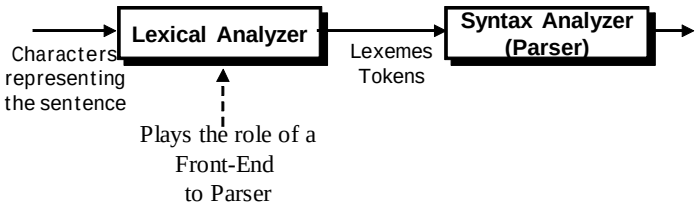
- (Ex: A parse tree is a restricted directed graph)

Syntax graphs (diagrams, charts) are directed graphs where **circle nodes** represent terminals and **rectangle nodes** represent non-terminals of a BNF grammar.



Recursive Descent Parsing

- A CFG can serve as a syntax analyzer, or parser, of a compiler. **Recursive descent** is a grammar-based top-down parser.
- **Parsing** is the process of tracing or constructing a parse tree for a given input string.
- Each nonterminal in the grammar has a subprogram associated with it;
  - Given an input string, it traces out the parse tree whose leaves match the input string.
  - The subprogram parses all sentential forms that the nonterminal can generate. In effect, it is a parser for the language that can be generated by its nonterminal.
  - These subprograms are built directly from the grammar rules, and they are usually recursive.



- **lexical()** gets leftmost token of input and puts it into global variable **next_token**.

Recursive descent parsers, like other top-down parsers, cannot be built from left-recursive grammars.

Example

Given the grammar:

```
<expr>   -> <term> {(+|-) <term> }
<term>   -> <factor>{(*|/)<factor>}
<factor> -> <id> | ( <expr> )
```

The recursive descent subprogram in C for the second rule:

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



```
void factor () {
  if (next_token == id_code ) {
      lexical();
      return;
  }
  else if (next_token == left_ paren_code) {
      lexical();
      expr();
      if (next_token == right_ paren_code) {
         lexical();
         return;
      else error();  /*expecting right  paranthesis*/
  }

  else
      error(); /*it was neither an id or a left  paranthesis*/
}
```

Parsers of real compilers report a diagnostic message when an error is detected, and recover from the error so that the parsing process can continue.

Homework 5

Due: May 12th, 2000 Friday

1-) Answer the following Review Questions:
3.4, 3.5, 3.6, 3.7 (10 points each)

2-) Solve the following problems in the Problem Sets:
3.2, 3.3, 3.4, and 3.8 (15 points each)