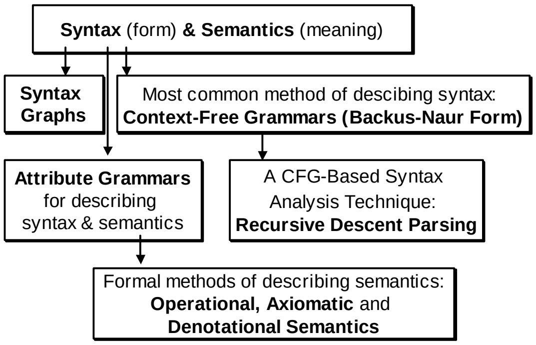
Chapter 3. Describing Syntax and Semantics



Syntax (form) **& Semantics** (meaning)

Syntax Graphs

Most common method of descibing syntax:
Context-Free Grammars (Backus-Naur Form)

Attribute Grammars for describing syntax & semantics

A CFG-Based Syntax Analysis Technique:
Recursive Descent Parsing

Formal methods of describing semantics:
Operational, Axiomatic and **Denotational Semantics**

Describing a Programming Language

The task of a concise yet understandable description of a PL is difficult but essential to the language’s success.

- – ALGOL 60 & ALGOL 68 are the first languages with concise descriptions.
- – What might be the result of imprecise description?

Who must use language definitions?

- Other language designers
- Implementors
- Programmers (the users of the language)

Syntax and Semantics

Study of PLs include examination of:

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

In a well-designed PL, semantics should follow directly from syntax.

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 progs 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.
- **Generators** - what we'll study
  - – A **language generator** is a device that can be used to generate the sentences of a language.
  - – more readable and understandable than recognizers
  - – Lang. recognizers are not useful as a language description mechanism.

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
```

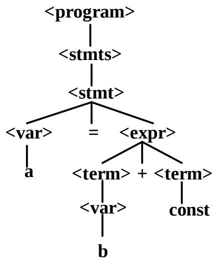
A derivation of a program in this language:

```
<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.

A grammar is **ambiguous** <u>iff</u> it generates a sentential form that has two or more distinct parse trees.



Every leaf is labelled with a terminal symbol.

A grammar is **ambiguous** <u>iff</u> it generates a sentential form that has two or more distinct parse trees.

- Ex: An ambiguous expression grammar:

```
<expr> -> <expr> <op> < expr>  |  const
<op> -> /  |  -
```

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity.
- Ex: An unambiguous expression grammar:

```
<expr> -> <expr> - <term> | <term>
<term> -> <term> /  const | const
```

Following derivation uses the above grammar:

```
<expr> => <expr> - <term> => <term>  - <term>
 => const - <term>
 => const - <term> / const
 => const - const / const
```

- Operator associativity can also be indicated by a grammar:

```
<expr> -> <expr> + <expr>  |  const  (ambiguous)
<expr> -> <expr> + const   |  const  (unambiguous)
```

Associativity of Operators

Make sure that the associativity is correctly described.

- Ex: A := B + C + A (See Figure 3.4)

In most cases, associativity of operators is irrelevant:

- In math, + is associative, i.e.,(A+B)+C = A + (B+C)
- In computers, + is sometimes <u>not</u> associative.
  Ex: Floating-point addition w/limited precision.
- (–) and (/) are not associative either in math or in a computer.

A **left (right) recursive BNF rule**: a rule where its LHS also appearing at the beginning (end) of its RHS.

- Left recursion specifies left associativity. (as in + - / *)
- Right recursion “ “ right associativity. (as in **)

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

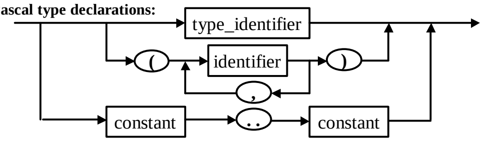
A **graph** is a collection of **nodes**, some of which are connected by lines, called **edges**.

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

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

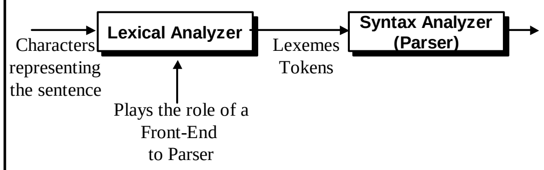
Syntax graphs (diagrams, charts) are directed graphs where **circle nodes** represent terminals and **rectangle nodes** represent non-terminals of a BNF grammar.

Pascal type declarations:



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 Analyzer

Syntax Analyzer (Parser)

Characters representing the sentence

Lexemes Tokens

Plays the role of a Front-End to Parser

- **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.

Static Semantics

(Have nothing to do with meaning but the legal forms of programs (syntax rather than semantics.))

Some characteristics of PLs:

1. Context-free but cumbersome (e.g., type checking)
   - Grammar would become too large to be useful. The size of the grammar determines the size of the parser.
2. Non-Context-free (e.g. variables must be declared before they are used)

Because of the inability to describe static semantics with BNF, a variety of more powerful mechanisms has been described for that task, such as attribute grammars.

Attribute Grammars (AGs) (Knuth, 1968)

CFGs cannot describe all of the syntax of programming languages. Additions to CFGs to carry some semantic info along through parse trees

Attribute grammars are grammars to which have been added:

- **Attributes**, which are associated with grammar symbols , are similar to variables that can be assigned values.
- **Attribute computation functions** (semantic functions) are associated with grammar rules to specify how attribute values are computed.
- **Predicate functions**, which state some of the syntax and semantic rules of the language, are associated with grammar rules.

Formal Definition

An **attribute grammar** is a CFG **G = (S, N, T, P)** with the following additions:

1. For each grammar symbol x there is a set A(x) of attribute values.
2. Each rule has a set of functions that define certain attributes of the nonterminals in the rule.
3. Each rule has a (possibly empty) set of predicates to check for attribute consistency.

Primary value of AGs:

1. Static semantics specification
2. Compiler design (static semantics checking)

Attributes and Attribute Computation Functions

Let **X_0** -> **X_1 ... X_n** be a rule.

Associated with each grammar symbol **X** is a set of attributes **A(X)** that consists of two disjoint sets:**S(X)** & **I(X)**

- Functions of the form **$S(X_0)$** = f(A(X_1), ... A(X_n)) define **synthesized attributes.**
  - used to pass semantic info up a parse tree.
  - f is a semantic function and value of X_0 depends only on the values of attributes on that node's children.
- Functions of the form **$I(X_j)$** = f(A(X_0), ... , A(X_n)), for i <= j <= n, define **inherited attributes.**
  - used to pass semantic info down a parse tree.
  - f is a semantic function and value of X_j depends on the values of attributes on that node's parent & siblings.

Predicate Functions

- A **predicate function** has the form of a Boolean expression on the attribute set {A(X_0), ... A(X_n)}.
  - Only derivations allowed with an attribute grammar are those in which the predicates associated with every nonterminal are all true.
  - A false predicate function value indicates a violation of the syntax or static semantics rules of the language.

Parse Tree of an Attribute Grammar

- Parse tree is based on its underlying BNF grammar, with a possibly empty set of attribute values attached to each node.
- If all the attribute values in a parse tree have been computed, the tree is said to be **fully attributed**.
- Assume that attribute values are computed after the complete unattributed tree has been constructed.

Intrinsic Attributes

Intrinsic attributes are synthesized attributes of leaf nodes whole values are determined outside the parse tree.

Example 1: Ada procedure names.

Rule: In Ada language, the name on the end of a procedure should match the procedure's name.

Syntax rule:

```
<proc_def> ® procedure <proc_name>[1]
                  <proc_body> end <proc_name>[2];
```

Semantic rule:

```
<proc_name>[1].string = <proc_name>[2].string
```

Example 2: Type Constraints

Rule: The syntax and semantics of an arithmetic statement are as follows:

- The only variable names are A, B, and C.
- The RHS of assignments can be: `<var> | <var> + <var>`
- There are only two variable types: `real` and `int`.
- When there are two variables on RHS, they need not be the same type:
  - The type of expression becomes `real` if types of two variables do not match.
  - When both variables have the same type, the expression type is assigned that type.
  - LHS's type in assignment must match the type of RHS.

BNF:

```
<assign> ® <var> := <expr>
<expr> ® <var> | <var> + <var>
 <var> ® A | B | C)
```

Attributes:

	<assign>	**< var>**	**< expr>**
synthesized	lhs_type	actual_type	actual_type
inherited	env	expected_type,env	env

The environment variable, **env**, is a pointer to the compiler's symbol table and is inherited from above the root of the parse tree in this grammar. The declarations in the language cause the compiler to generate a symbol table. **(See Example 3.6)**

Example 3: Simple Expression

Expressions of the form: `id + id`

- `id`'s can be either **int_type** or **real_type**
- types of the two `id`'s must be the same
- type of the expression must match it's expected type

BNF:

```
<expr> -> <var> + <var>
<var> -> id
```

Attributes:

- **actual_type** - synthesized for **`<var>`** and **`<expr>`**
- **expected_type** - inherited for **`<expr>`**

Attribute Grammar:

1. Syntax rule: `<expr> -> <var>[1] + <var>[2]`

   Semantic rules:

   `<var>[1].env ¬ <expr>.env`
   `<var>[2].env ¬ <expr>.env`
   `<expr>.actual_type ¬ <var>[1].actual_type`

   Predicate:

   `<var>[1].actual_type = <var>[2].actual_type`
   `<expr>.expected_type = <expr>.actual_type`

2. Syntax rule: `<var> -> id`

   Semantic rule:

   `<var>.actual_type ¬ lookup (id, <var>.env)`

How are Attribute Values Computed?

1. If all attributes were inherited, the tree could be decorated in **top-down order**.

2. If all attributes were synthesized, the tree could be decorated in **bottom-up order**.

3. In many cases, both kinds of attributes are used, and it is some **combination** of top-down and bottom-up that must be used.

Attribute Evaluation Order

1. `<expr>.env ¬` inherited from parent
`<expr>.expected_type ¬` inherited from parent

2. `<var>[1].env ¬ <expr>.env`
`<var>[2].env ¬ <expr>.env`

3. `<var>[1].actual_type ¬ lookup (A, <var>[1].env)`
`<var>[2].actual_type ¬ lookup (B, <var>[2].env)`
`<var>[1].actual_type =? <var>[2].actual_type`

4. `<expr>.actual_type ¬ <var>[1].actual_type`
`<expr>.actual_type =? <expr>.expected_type`

Dynamic Semantics

No single widely acceptable notation or formalism for describing semantics, all are complicated and very theoretical.

Three common types:

1. **Operational Semantics**
2. **Axiomatic Semantics**
   - Based on formal logic (first order predicate calculus)
   - Original purpose: formal program verification
3. **Denotational Semantics**
   - Based on recursive function theory
   - The most abstract semantics description method.

Homework 2

Due: March 2nd, 1999 Tuesday

1-) Answer the following Review Questions:
2.5, 3.5, 3.9, and 3.12 (Each 10 points)

2-) Solve the following problems in the Problem Sets:
2.1, 3.5, 3.7, 3.8 (Each 15 points)