Chapter 3
Describing Syntax and Semantics
Introduction

- Who must use language definitions?
  1. Other language designers
  2. Implementors
  3. Programmers (the users of the language)

We really need to have a clear language definition, because if not we find that a language may be hard to learn, hard to implement, and any ambiguity in the specification may lead to dialect differences which will weaken the language in most cases.
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

Syntax Example: Overly simple C if statement

```c
if (<expr> )
    <true-statement>
else
    <false-statement>
```

Semantics Example: if the expression evaluated to true (non-zero) execute the true statement (or block) otherwise execute the false statement (or block)

*Semantics should follow from syntax, the form of statements should be clear and imply what the statements do or how they should be used.*
Describing Syntax

- 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 Definition of Languages

• **Recognizers**
  - A recognition device reads input strings of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler

• **Generators**
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator
Formal Methods of Describing Syntax

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

- **Backus-Naur Form (1959)**
  - Invented by John Backus to describe Algol 58
  - BNF is equivalent to context-free grammars, we tend to use BNF in our discussions
    - Extended BNF - Improves readability and writability of BNF

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

Note: *We encounter meta languages all the time today (e.g. SGML/XML)*
Formal Methods of Describing Syntax (Continued)

- 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> ) <stmt>

This is a *rule* or *production* and 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 *terminal* and *nonterminal* symbols

- A *grammar* is a finite non-empty set of rules

*Be careful with BNF you will find that people are loose with it and instead of < > you may find italics for non-terminals. You may also see it extended with common regular expression constructs.*
Formal Methods of Describing Syntax (Continued)

- An abstraction (or nonterminal symbol) can have more than one RHS.

  \[ \text{<if-stmt>} \rightarrow \text{if <exp> then <stmt>} \]
  \[ \text{<if-stmt>} \rightarrow \text{if <exp> then <stmt>} \text{ else <stmt>} \]

  Could be become

  \[ \text{<if-stmt>} \rightarrow \text{if <exp> then <stmt>} | \text{if <exp> then <stmt>} \text{ else <stmt>} \]

  You could obviously unravel something like

  \[ \text{<stmt>} \rightarrow \text{<single_stmt>} \]
  \[ \text{ | begin <stmt_list> end} \]

- Syntactic lists are described using recursion (LHS appears on RHS)

  \[ \text{<ident_list>} \rightarrow \text{ident | ident, <ident_list>} \quad \{\text{note comma is a terminal}\} \]

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
- Every string of symbols in the derivation is a *sentential form* finally ending up in a *sentence* which 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, rightmost is the opposite. You could also do one that is not so consistent.

- Derivation order should have no effect on the language generated by a grammar.

- Exhaustively choosing all combos in rules should generate the whole language, but most programming language grammars are infinite and all sentences could not be generated in finite time.
Formal Methods of Describing Syntax (Continued)

- **An example grammar:**
  
  \[ <\text{sentence}> \rightarrow <\text{noun-phrase}> <\text{verb-phrase}> . \quad \{\text{note the period}\} \]
  
  \[ <\text{noun-phrase}> \rightarrow <\text{article}> <\text{noun}> \]
  
  \[ <\text{article}> \rightarrow a \mid \text{the} \]
  
  \[ <\text{noun}> \rightarrow \text{girl} \mid \text{dog} \]
  
  \[ <\text{verb-phrase}> \rightarrow <\text{verb}> <\text{noun-phrase}> \]
  
  \[ <\text{verb}> \rightarrow \text{sees} \mid \text{pets} \]

- **An example derivation (left most):** (\(\Rightarrow\) reads as derives)

  \[ <\text{sentence}> \Rightarrow <\text{noun-phrase}> <\text{verb-phrase}> . \]
  
  \[ \Rightarrow <\text{article}> <\text{noun}> <\text{verb-phrase}> . \]
  
  \[ \Rightarrow \text{the} <\text{noun}> <\text{verb-phrase}> \]
  
  \[ \Rightarrow \text{the girl} <\text{verb-phrase}> . \]
  
  \[ \Rightarrow \text{the girl} <\text{verb}> <\text{noun-phrase}> . \]
  
  \[ \Rightarrow \text{the girl sees} <\text{noun-phrase}> . \]
  
  \[ \Rightarrow \text{the girl sees} <\text{article}> <\text{noun}> . \]
  
  \[ \Rightarrow \text{the girl sees} \text{a} <\text{noun}> . \]
  
  \[ \Rightarrow \text{the girl sees} \text{a dog} . \]
Context Free?

- In the previous example you might wonder about the idea of context.
- In a context-free grammar we find that replacements do not have any context which they cannot occur. For example you might imagine that pets as a verb should only be allowed in the case that girl is the subject
  - The dog pets the girl = wrong
  - The girl pets the dog = ok
- Of course this means that there are certain contexts that the rules don’t work, thus it would not be “context free”
- Adding more productions you might be able to work around simple issues, but be careful we are starting to confuse syntax and semantics and there are some things that will not be possible no matter how many productions we add.
Formal Methods of Describing Syntax (Continued)

- Another example grammar from the book this time:

\[
\begin{align*}
\text{<program>} & \rightarrow \text{<stmts>} \\
\text{<stmts>} & \rightarrow \text{<stmt>} \mid \text{<stmt>} ; \text{<stmts>} \\
\text{<stmt>} & \rightarrow \text{<var>} = \text{<expr>} \\
\text{<var>} & \rightarrow a \mid b \mid c \mid d \\
\text{<expr>} & \rightarrow \text{<term>} + \text{<term>} \mid \text{<term>} - \text{<term>} \\
\text{<term>} & \rightarrow \text{<var>} \mid \text{const}
\end{align*}
\]

- An example derivation:

\[
\begin{align*}
\text{<program>} & \Rightarrow \text{<stmts>} \Rightarrow \text{<stmt>} \\
& \Rightarrow \text{<var>} = \text{<expr>} \Rightarrow a = \text{<expr>} \\
& \Rightarrow a = \text{<term>} + \text{<term>} \\
& \Rightarrow a = \text{<var>} + \text{<term>} \\
& \Rightarrow a = b + \text{<term>} \\
& \Rightarrow a = b + \text{const}
\end{align*}
\]
Formal Methods of Describing Syntax (Continued)

- *Yet another example grammar:*
  
  \[
  \texttt{<expr>} \rightarrow \texttt{<expr> + <expr>} \mid \texttt{<expr> * <expr>} \mid ( \texttt{<expr>} ) \mid \texttt{<number>}
  \]

  \[
  \texttt{<number>} \rightarrow \texttt{<number> <digit>} \mid \texttt{<digit>}
  \]

  \[
  \texttt{<digit>} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
  \]

- *An example derivation:*

  \[
  \texttt{<number>} \Rightarrow \texttt{<number> <digit>}
  \]

  \[
  \Rightarrow \texttt{<number> <digit> <digit>}
  \]

  \[
  \Rightarrow \texttt{<digit> <digit> <digit>}
  \]

  \[
  \Rightarrow \texttt{2 <digit> <digit>}
  \]

  \[
  \Rightarrow \texttt{23 <digit>}
  \]

  \[
  \Rightarrow \texttt{234}
  \]
Parse Trees and Abstract Syntax Trees

- Syntax establishes structure, not meaning
- However, the meaning of a sentence (or program) must be related to its syntax.
- Given $\text{exprresult} \rightarrow \text{expr} + \text{expr}$
  we expect to add the values of the two right hands to get the left hand.
  - We just added meaning there, this is called syntax directed semantics (semantics directed syntax?)
- A parse tree is a hierarchical representation of a derivation

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

- Given “the girl sees a dog.”

```
sentence
  
  noun-phrase
  
  article
  The

  noun
  girl

  verb
  sees

  noun-phrase
  
  article
  a

  noun
  dog
```
Parse Tree Notes

• A parse tree is labeled by non-terminals at interior nodes and terminals at leaves
  - Interior nodes = production steps in derivation
• All terminals and non-terminals in a derivation are included in a parse tree
• Not everything may be necessary to determine syntactic structure, we can leave out some details creating an abstract syntax tree (AST) or just a syntax tree
  - Useful in compilers and to understand HTML/XML markup
Abstract Syntax Tree Example

- For $3 + 4 \times 5$ we have the parse tree
Abstract Syntax Tree Example Contd.

- As an AST we just need
Ambiguity

- Two different derivations can lead to the same the parse tree, this is good because the grammar is unambiguous

- Given 234 we have different derivations
  - \text{number} => \text{number digi}t
    - => \text{number 4}
    - => \text{number digi}t 4
    - => \text{number 3 4}
    - => \text{digi}t 3 4
    - => 234
  - \text{number} => \text{number digi}t
    - => \text{number digi}t digit
    - => \text{digi}t digit digit
    - => 2 digit digit
    - => 2 3 digit
    - => 234
Ambiguity Contd.

- However the parse tree is the same in either case
Ambiguity Contd.

- This isn’t always the case consider $3+4*5$ we might have two different simplified parse trees:

As you can see here we seem to have a precedence problem now!
Removing Ambiguity

- A grammar that produces different parse trees depending on derivation order is considered *ambiguous*.
- We can try to revise the grammar and introduce a disambiguating rule to establish which of the trees we want.
- In the previous example we want multiplication to take precedence over addition, thus we tend to write a special grammar rule that establishes a precedence cascade to force the * at the lower point in the tree.
Removing Ambiguity Contd.

• To remove the ambiguity you might add
  - \(<\text{expr}> \rightarrow <\text{expr}> + <\text{expr}> \mid <\text{term}>\)
  - \(<\text{term}> \rightarrow <\text{term}> * <\text{term}> \mid ( <\text{expr}> ) \mid <\text{number}>\)

• This doesn’t quite do it because \(3 + 4 + 5\) can be \((3 + 4) + 5\) or \(3 + (4 + 5)\)
  - Addition becomes left or right associative, this isn’t so bad with addition but associativity can be a problem with other operators. We can fix this with some new rules where we find that
    • Left recursive rules become left associative
    • Right recursive rules become right associative
Removing Ambiguity Contd.

- The revised grammar is as follows

\[
<\text{expr}> \rightarrow <\text{expr}> \ + \ <\text{term}> \ | \ <\text{term}>
\]
\[
<\text{term}> \rightarrow <\text{term}> \ * \ <\text{factor}> \ | \ <\text{factor}>
\]
\[
<\text{factor}> \rightarrow ( \ <\text{expr}> ) \ | \ <\text{number}>
\]
\[
<\text{number}> \rightarrow <\text{number}> \ <\text{digit}> \ | \ <\text{digit}>
\]
\[
<\text{digit}> \rightarrow 0 \ | \ 1 \ | \ 2 \ | \ 3 \ | \ 4 \ | \ 5 \ | \ 6 \ | \ 7 \ | \ 8 \ | \ 9
\]

- This should be unambiguous, try it and see with some derivations and parse trees
- Extended BNF (just abbreviations):
  1. Optional parts are placed 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}
     ```

- BNF:
  
  ```
  <expr> → <expr> + <term> | <expr> - <term> | <term>
  <term> → <term> * <factor> | <term> / <factor> | <factor>
  ```

- EBNF:
  
  ```
  <expr> → <term> {(+ | -) <term>}
  <term> → <factor> {(* | /) <factor>}
  ```

There are even more BNF like forms out there if you look around
Augmented BNF forms (http://www.ietf.org/rfc/rfc2234.txt)
You may also see people using basic RegExes for at least portions of languages
Formal Methods of Describing Syntax (Continued)

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

Example here Pascal type declarations

```
type_identifier
 (        identifier          )
 constant          ..        constant
```

```
\begin{center}
\begin{tikzpicture}
    \node [rectangle, draw] (type) {type_identifier};
    \node [circle, draw] (identifier) at (type -| 0,0) {identifier};
    \node [circle, draw] (constant) at (type -| -1,0) {constant};
    \node [circle, draw] (constant2) at (type -| 1,0) {constant};
    \node [circle, draw] (comma) at (identifier -| 0,0) {$,$};
    \node [circle, draw] (function) at (type -| -1,1) {\textbf{(}};
    \node [circle, draw] (function2) at (type -| 1,1) {\textbf{)}};
    \draw [->] (type) -- (identifier);
    \draw [->] (identifier) -- (comma);
    \draw [->] (comma) -- (constant);
    \draw [->] (constant) -- (function);
    \draw [->] (function) -- (type);
    \draw [->] (function2) -- (type);
    \draw [->] (type) -- (type2);
\end{tikzpicture}
\end{center}
```
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
- Primary value of AGs:
  1. Static semantics specification
  2. Compiler design (static semantics checking)
- Def: 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
Attribute Grammars (continued)

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule.

- Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$ define synthesized attributes.

- Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, for $i \leq j \leq n$, define inherited attributes.

- Initially, there are intrinsic attributes on the leaves.

- Example: 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.
Attribute Grammars (continued)

- **BNF:**
  \[
  \text{<expr> } \rightarrow \text{<var> + <var>}
  \]
  \[
  \text{<var> } \rightarrow \text{id}
  \]

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

- **The Attribute Grammar:**
  1. Syntax rule: \( \text{<expr> } \rightarrow \text{<var>[1] + <var>[2]} \)
     Semantic rules:
     \( \text{<expr>.actual_type} \leftarrow \text{<var>[1].actual_type} \)
     Predicate:
     \( \text{<var>[1].actual_type == <var>[2].actual_type} \)
     \( \text{<expr>.expected_type == <expr>.actual_type} \)
  2. Syntax rule: \( \text{<var> } \rightarrow \text{id} \)
     Semantic rule:
     \( \text{<var>.actual_type} \leftarrow \text{lookup (<var>.string)} \)
Attribute Grammars (continued)

- 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.

1. <expr>.expected_type ← inherited from parent

2. <var>[1].actual_type ← lookup (A)
   <var>[2].actual_type ← lookup (B)
   <var>[1].actual_type =? <var>[2].actual_type

3. <expr>.actual_type ← <var>[1].actual_type
   <expr>.actual_type =? <expr>.expected_type
Semantics Overview

- Specifying the semantics of a programming language is a much more difficult task than specifying syntax.
- We need formal semantics to define all the props of a language that are not specified with a BNF (declaration before use, some type issues, etc.)

- Three approaches
  - Reference Manual Approach (in English as precise as we can make things)
  - Define a translator - see what the language does by experimentation
    - Seems ridiculous but is this how HTML/CSS/JS is often dealt with, the popular browser(s) being the translators?
  - Formal definition - very precise, but complex and may not be useful to the groups describes early in the slide
Semantics Overview

- The advantages of the formal definition is that it is so precise that programs can be proven correct and translators validated to produce the defined behavior.

- Formal definitions for semantics have not met with complete acceptance, multiple approaches are pushed and many are not well understood and most not used with common languages (at least not initially).

- Interestingly formal semantics are often after the fact to a language or may not be applied to the whole language
  - The story of HTML and SGML
Semantics Overview

• Three methods for formal semantics
  1. Operational Semantics - defines a language by describing its actions in terms of operations on an actual or hypothetical machine

  2. Denotational semantics - uses mathematical functions on programs to specify semantics

  3. Axiomatic semantics - applies mathematical logic to language definition. Assertions or predicates are used to describe desired outcomes and initial assumptions. Constructs transform new assertions out of old ones reflecting the action of the construct. These transforms can prove the desired outcome follows from the initial conditions (a correctness proof)
Semantics

First off, note there is no single widely acceptable notation or formalism for describing semantics

1. Operational Semantics
   - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement

   - To use operational semantics for a high-level language, a virtual machine is needed

   - A *hardware* pure interpreter would be too expensive to create
Semantics (continued)

- A software pure interpreter also has problems:
- A possible alternative: A complete computer simulation

- The process:
  1. Build a translator (translates source code to the machine code of an idealized computer)
  2. Build a simulator for the idealized computer

- Evaluation of operational semantics:
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g., VDL)
2. **Axiomatic Semantics**

- Based on formal logic (first order predicate calculus)
- *Original purpose:* formal program verification
- *Approach:* Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)
Semantics (continued)

3. **Denotational Semantics**

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)

The process of building a denotational spec for a language (not necessarily easy):
1. Define a mathematical object for each language entity
2. Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
Semantics (continued)

- The difference between denotational and operational semantics: In operational semantics, the state changes are defined by coded algorithms; in denotational semantics, they are defined by rigorous mathematical functions
Semantics (continued)

- Evaluation of denotational semantics:
  - Can be used to prove the correctness of programs
  - Provides a rigorous way to think about programs
  - Can be an aid to language design
  - Has been used in compiler generation systems
  - Probably way beyond what most folks will get involved with
Encounters with Syntax and Semantics

- JavaScript specification

- Various markup language specifications particular those based upon XML
  - [http://www.w3.org/TR/xhtml1/](http://www.w3.org/TR/xhtml1/)

- Are there specifications for your favorite languages?
  - Real specification complete with grammar or de facto specification in form of a popular book or site?
Summary

• BNF and context-free grammars are equivalent metalinguages
  - Well-suited for describing the syntax of programming languages
• An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
• Three primary methods of semantics description
  - Operation, axiomatic, denotational
• Actual application of these ideas in common PLs varies
  - BNFs or some variant - common, formal semantic definitions beyond reference manual style - fairly uncommon