Syntax and Semantics

Describing Syntax

- Syntax
  - Form or structure of the expressions, statements and program units
  - What a program looks like
  - Defined using a formal grammar
  - Backus-Naur Form (BNF)
  - Notation to describe syntax

Describing Syntax

- Semantics
  - Meaning of the expressions and expressions
  - Describes execution behavior
  - Static Semantics
    - Semantics described at compile time
    - \texttt{C++/C/Java: int a = 3;}
      - Type of a is integer
  - Dynamic Semantics
    - Semantics determined during execution
    - \texttt{Scheme: (+ a b)}
      - Function result type determined by types of a and b

Describing Syntax

- Importance of Syntax and Semantics
  - Programmers using the language
  - Language Implementers
  - Language Designers
  - Well designed languages provide:
    - Readability - self-documenting
    - Syntax reflecting semantics
    - Writeability - concise statements
    - Ease of Verifiability & Translation
    - Lack of Ambiguity

Describing Syntax

- Lexemes
  - Characters or alphabet of a language
- Token
  - Lowest level syntactic unit of a language
- Category of Lexemes
  - Identifiers, delimiters
- Sentence
  - String of characters over some alphabet
- Language
  - Set of sentences
Describing Syntax

Sentence

\[ \text{sum} = 2 \times \text{count} + 17; \]

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>identifier</td>
</tr>
<tr>
<td>=</td>
<td>equal-sign</td>
</tr>
<tr>
<td>2</td>
<td>number</td>
</tr>
<tr>
<td>*</td>
<td>mult-op</td>
</tr>
<tr>
<td>count</td>
<td>identifier</td>
</tr>
<tr>
<td>+</td>
<td>plus-op</td>
</tr>
<tr>
<td>17</td>
<td>number</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
</tr>
</tbody>
</table>

Describing Syntax

Common Syntactic Elements
- Character Set
- Identifiers
- Operator Symbols
- Keywords and reserve Words
- Noise Words
- Comments
- Blanks (spaces)
- Delimiters and Brackets
- Free and Fixed Field Formats
- Expressions
- Statements

Formally describing syntax:
- Generators
  - Used by programmers
  - Rules to generate sentences in a language
- Recognizer
  - Used in Compilers
  - "Is this sentence in my Language?"

Context-Free Grammars
- Grammar
  - Finite Non-empty Set of Rules
- Rules
  - Describe the structure of sentences
- Language Generators
  - Intent is to describe the syntax of natural languages
  - Developed by Noam Chomsky in the mid-1950s
  - Define context-free languages
- A meta-language
  - A rule is not dependent on previous rules (context)
  - A language used to describe other languages

An English Grammar

A sentence is:
- a noun phrase, a verb, and a noun phrase.
  \[ <S> ::= <NP> <V> <NP> \]

A noun phrase is:
- an article and a noun.
  \[ <NP> ::= <A> <N> \]

A verb is:
- \[ <V> ::= \text{loves} | \text{hates} | \text{eats} \]

An article is:
- \[ <A> ::= \text{a} | \text{the} \]

A noun is:
- \[ <N> ::= \text{dog} | \text{cat} | \text{rat} \]
How Grammar Works

- The grammar is a set of rules that say how to build a tree—a parse tree.
- You put $<S>$ at the root of the tree.
- The grammar's rules say how children can be added at any point in the tree.
- For instance, to expand the rule, add nodes $<NP>$, $<V>$, and $<NP>$, in that order, as children of $<S>$.

A Parse Tree

```
<NP> ::= <A> <N>
<V> ::= loves | hates | eats

<NP> ::= <A> <N>
<NP> ::= <NP> <V> <NP>

<s> ::= <NP> <V> <NP>
```

BNF Grammar Definition

- **Backus-Naur Form (BNF)**
  - Used by John Backus to describe ALGOL 58.
  - Equivalent to Context-free Grammar.
- A BNF grammar consists of four parts:
  - The set of tokens or terminals.
  - The set of non-terminal symbols.
  - The start symbol.
  - The set of productions.

BNF Grammar Definition

- **Start symbol**
  - The non-terminal that forms the root of any parse tree for the grammar.
- **Productions**
  - Tree-building rules.
  - The left-hand side is a single non-terminal.
  - The right-hand side is a sequence of one or more things, each of which can be either a token or a non-terminal.
  - A production gives one possible way of building a parse tree: it permits the non-terminal symbol on the left-hand side to have the things on the right-hand side, in order, as its children in a parse tree.

BNF Grammar Definition

- **Tokens**
  - Smallest units of syntax.
  - Strings of one or more characters of text.
  - Atomic: not composed from smaller parts.
- **Non-terminal symbols**
  - Stand for larger pieces of syntax.
  - Not strings occurring literally in program text.
  - Grammar defined how they can be expanded.
- **BNF Grammar Definition**
  - They are strings enclosed in angle brackets.

BNF Grammar Definition

- **Rules (productions) are comprised of**
  - A Left-hand side (LHS).
  - A Right-hand side (RHS).
  - Terminal Symbols.
  - Non-terminal Symbols.
  - Represent syntactic structures.
  - Used as syntactic variables.
Alternative Notations

When there is more than one production with the same left-hand side, an abbreviated form can be used:

\[
\begin{align*}
\text{<exp>} & \text{ ::= } \text{<exp>} + \text{<exp}> \\
\text{<exp>} & \text{ ::= } \text{<exp>} \ast \text{<exp}> \\
\text{<exp>} & \text{ ::= } \{ \text{<exp>} \} \\
\text{<exp>} & \text{ ::= } \text{a} \\
\text{<exp>} & \text{ ::= } \text{b} \\
\text{<exp>} & \text{ ::= } \text{c}
\end{align*}
\]

Note that there are six productions in this grammar. It is equivalent to this one:

\[
\begin{align*}
\text{<exp>} & \text{ ::= } \text{<exp>} + \text{<exp}> \\
\text{<exp>} & \text{ ::= } \text{<exp>} \ast \text{<exp}> \\
\text{<exp>} & \text{ ::= } \{ \text{<exp>} \} \\
\text{<exp>} & \text{ ::= } \text{a} \\
\text{<exp>} & \text{ ::= } \text{b} \\
\text{<exp>} & \text{ ::= } \text{c}
\end{align*}
\]

Definitions

- start symbol
  \[
  \text{<S>} \text{ ::= } \text{<NP>} \text{<V>} \text{<NP>}
  \]
- a production
  \[
  \text{<V>} \text{ ::= } \text{loves} | \text{hates} | \text{eats}
  \]
- non-terminals
  \[
  \text{<NP>} \text{ ::= } \text{a} | \text{the} \\
  \text{<NP>} \text{ ::= } \text{dog} | \text{cat} | \text{rat}
  \]
- tokens

Abstractions

- LHS nonterminals
- Can have more than one RHS
  \[
  \text{<stmt>} \rightarrow \text{<single_stmt>} \\
  \text{begin \text{<stmt_list> end}} \\
  \text{<stmt>} \rightarrow \text{<single_stmt> | begin \text{<stmt_list> end}}
  \]
- Lists can be described using recursion
  \[
  \text{<ident_list> \rightarrow ident | ident <ident_list>}
  \]

Derivations

- Derivation
  - begins with the language start symbol
  - "Top-Level" nonterminal
  - repeated application of rules
  - ends with a sentence in the language
  - all terminal symbols
- Technique for generating sentences
  - "What sentences does this grammar generate?"
  - "Show the derivation for . . ."

An example grammar:

\[
\begin{align*}
\text{<program>} & \rightarrow \text{<stmt_list>} \\
\text{<stmt_list>} & \rightarrow \text{<stmt>} | \text{<stmt> ; <stmt_list>} \\
\text{<stmt>} & \rightarrow \text{<var>} = \text{<expr>} \\
\text{<var>} & \rightarrow \text{a} | \text{b} | \text{c} | \text{d} \\
\text{<expr>} & \rightarrow \text{<term>} + \text{<term>} | \text{<term>} - \text{<term>} \\
\text{<term>} & \rightarrow \text{<var>} | \text{<const>} \\
\text{<const>} & \rightarrow 1 | 2 | 3
\end{align*}
\]

Start Symbol

<program>
**Grammars & Derivations**

An example derivation:

\[ \langle \text{program} \rangle \Rightarrow \langle \text{stmt_list} \rangle \]
\[ \Rightarrow \langle \text{stmt} \rangle \quad | \quad \langle \text{stmt} \rangle \; ; \; \langle \text{stmt_list} \rangle \]
\[ \langle \text{stmt} \rangle \Rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle \]
\[ \Rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle \]
\[ \Rightarrow \langle \text{var} \rangle = \langle \text{term} \rangle + \langle \text{term} \rangle \]
\[ \Rightarrow \langle \text{var} \rangle = \langle \text{var} \rangle + \langle \text{term} \rangle \]
\[ \Rightarrow \langle \text{var} \rangle = \langle \text{var} \rangle + \langle \text{term} \rangle \]
\[ \Rightarrow \langle \text{var} \rangle = \langle \text{var} \rangle + \langle \text{const} \rangle \]
\[ \Rightarrow \langle \text{var} \rangle = \langle \text{var} \rangle + 2 \]

**Derivations**

- Every string of symbols is a sentential form.
- Leftmost derivation:
  - Leftmost nonterminal in each sentential form is expanded first.
  - Derivation may be neither leftmost nor rightmost.
- Sentence:
  - Sentential form having only terminal symbols.

**Empty**

- The special nonterminal \( \langle \text{empty} \rangle \) is for places where you want the grammar to generate nothing.
- For example, this grammar defines a typical if-then construct with an optional else part:

\[ \langle \text{if-stmt} \rangle ::= \text{if} \; \langle \text{expr} \rangle \; \text{then} \; \langle \text{stmt} \rangle \; \langle \text{else-part} \rangle \]
\[ \langle \text{else-part} \rangle ::= \text{else} \; \langle \text{stmt} \rangle \quad | \quad \langle \text{empty} \rangle \]

**Parse Trees**

- To build a parse tree, put the start symbol at the root.
- Add children to every non-terminal, following any one of the productions for that non-terminal in the grammar.
- Done when all the leaves are tokens.
- Read off leaves from left to right—that is the string derived by the tree.

**Grammars & Derivations**

Parse tree from example grammar:

\[ \langle \text{program} \rangle \rightarrow \langle \text{stmt_list} \rangle \]
\[ \langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle \; | \; \langle \text{stmt} \rangle \; ; \; \langle \text{stmt_list} \rangle \]
\[ \langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expr} \rangle \]
\[ \langle \text{var} \rangle \rightarrow \text{a} \; | \; \text{b} \; | \; \text{c} \; | \; \text{d} \]
\[ \langle \text{expr} \rangle \rightarrow \langle \text{term} \rangle + \langle \text{term} \rangle \; | \; \langle \text{term} \rangle - \langle \text{term} \rangle \]
\[ \langle \text{term} \rangle \rightarrow \langle \text{var} \rangle \; | \; \langle \text{const} \rangle \]
\[ \langle \text{const} \rangle \rightarrow \text{1} \; | \; \text{2} \; | \; \text{3} \]

**Start Symbol**

\[ \langle \text{program} \rangle \]
Grammars & Derivations

Parse Tree
- Hierarchical representation of a derivation

Nodes are non-terminals

Leaves are terminals

Compiler Note

- What we just did is parsing
  - try to find a parse tree for a given string
- Key step in compiling
  - try to build a parse tree for program,
  - using grammar for language programmer used
- Take a course in compiler construction to learn about algorithms for doing this efficiently

Language Definition

- We use grammars to define the syntax of programming languages
- The language defined by a grammar is the set of all strings that can be derived by some parse tree for the grammar
- As in the previous example, that set is often infinite (though grammars are finite)
- Constructing grammars is a little like programming...

Constructing Grammars

- Most important trick: divide and conquer
- Example: the language of Java declarations: a type name, a list of variables separated by commas, and a semicolon
- Each variable can be followed by an initializer:
  
  float a;
  boolean a, b, c;
  int a = 1, b, c = 1 + 2;

- Primitive type names:
  
  <type-name> ::= boolean | byte | short | int | long | char | float | double

  (Note: skipping constructed types: class names, interface names, and array types)
Constructing Grammars

That leaves the comma-separated list of variables with initializers.

Postpone defining variables with initializers, and just do the comma-separated list part:

\[
<\text{declarator-list}> ::= <\text{declarator}>
\quad|\quad <\text{declarator}>, <\text{declarator-list}>
\]

Constructing Grammars

That leaves the variables with initializers:

\[
<\text{declarator}> ::= <\text{variable-name}>
\quad|\quad <\text{variable-name}> = <\text{expr}>
\]

For full Java, we would need pairs of square brackets after the variable name.

There is also a syntax for array initializers.

And definitions for \( <\text{variable-name} > \) and \( <\text{expr} > \)

Where are Tokens from?

Tokens are pieces of program text that we do not choose to think of as being built from smaller pieces.

Identifiers (count), keywords (if), operators (==), constants (123.4), etc.

Programs stored in files are just sequences of characters.

How is such a file divided into a sequence of tokens?

Lexical Structure And Phrase Structure

Grammars so far have defined phrase structure: how a program is built from a sequence of tokens.

We also need to define lexical structure: how a text file is divided into tokens.

One Grammar For Both

You could do it all with one grammar by using characters as the only tokens.

Not done in practice: things like white space and comments would make the grammar too messy to be readable.

\[
<\text{if-stmt}> ::= \text{if} <\text{white-space} > <\text{expr} > <\text{white-space} > \text{then} <\text{white-space} > <\text{stmt} > <\text{white-space} > <\text{else-part} >
\text{else-part} ::= \text{else} <\text{white-space} > <\text{stmt} > <\text{empty} >
\]

One Grammar For Both

You could do it all with one grammar by using characters as the only tokens.

Not done in practice: things like white space and comments would make the grammar too messy to be readable.

\[
<\text{if-stmt}> ::= \text{if} <\text{white-space} > <\text{expr} > <\text{white-space} > \text{then} <\text{white-space} > <\text{stmt} > <\text{white-space} > <\text{else-part} >
\text{else-part} ::= \text{else} <\text{white-space} > <\text{stmt} > <\text{empty} >
\]
**Separate Grammars**

- Usually there are two separate grammars
  - One says how to construct a sequence of tokens from a file of characters
  - One says how to construct a parse tree from a sequence of tokens

\[
\text{<program-file>} ::= \text{<end-of-file>} | \text{<element>} \text{<program-file>}
\]
\[
\text{<element>} ::= \text{<token>} | \text{<one-white-space>} | \text{<comment>}
\]
\[
\text{<one-white-space>} ::= \text{<space>} | \text{<tab>} | \text{<end-of-line>}
\]
\[
\text{<token>} ::= \text{<identifier>} | \text{<operator>} | \text{<constant>} | \ldots
\]

**Separate Compiler Passes**

- The scanner (Lexical Analyzer) reads the input file and divides it into tokens according to the first grammar
- The scanner discards white space and comments
- The parser constructs a parse tree (or at least goes through the motions—more about this later) from the token stream according to the second grammar

**Historical Note #1**

- Early languages sometimes did not separate lexical structure from phrase structure
  - Early Fortran and Algol dialects allowed spaces anywhere, even in the middle of a keyword
  - Other languages like PL/I allow keywords to be used as identifiers
- Makes them harder to scan and parse
- It also reduces readability

**Historical Note #2**

- Some languages have a fixed-format lexical structure—column positions are significant
  - One statement per line (i.e. per card)
  - First few columns for statement label
- Early dialects of Fortran, Cobol, and Basic
- Almost all modern languages are free-format: column positions are ignored