CSE305
Programming Languages

Syntax

What is it?
How is it specified?
Who uses it?
Why is it needed?
Note

• Today’s notes are based on the Sebesta text.

• The tree diagrams in these slides are from the lecture slides provided in the instructor resources for the text, and were made by David Garrett.

• Cell phones off & name signs out
  – I’ll judge signs on Wednesday next week
Introduction: syntax and semantics

• **Syntax:** a formal description of the structure of programs in a given language.

• **Semantics:** a formal description of the meaning programs in a given language.

• Together the syntax and semantics define a language.
Who uses a language definition?

• Those who design a language
• Those who implement a language (e.g. write compilers for it)
• Those who use the language (i.e. software developers)
• Those who make tools for developers (e.g. JDT in Eclipse)
Language & grammar

• A given *language* can have more than one *grammar* which describes it.

• The grammar presented to a user is not necessarily the same as the grammar used in an implementation.
  – implementation requires a very detailed grammar
  – user needs a human-readable grammar
Syntax and semantics of programming languages

• I have cautioned against getting too hung up on the *syntax* of a programming language.
• But, you still need to learn the syntax of any language you work with so that you can read and write programs in the language.
• To understand the meaning of programs expressed in a language you also have know the *semantics* of the language.
General background

- Chomsky hierarchy
- Context-free grammars
- Backus-Naur form
Chomsky hierarchy

- Noam Chomsky defined a hierarchy of grammars and languages known as the Chomsky hierarchy:
  - regular languages (most restrictive)
  - context-free languages
  - context-sensitive languages
  - unrestricted languages (least restrictive)
Chomsky hierarchy

- unrestricted languages
- context-sensitive languages
- context-free languages
- regular languages
Context-free (CF) grammar

- A CF grammar is formally presented as a 4-tuple \( G = (T, NT, P, S) \), where:
  - \( T \) is a set of terminal symbols (the *alphabet*)
  - \( NT \) is a set of non-terminal symbols
  - \( P \) is a set of productions (or rules), where
    \( P \in NT \times (T \cup NT)^* \)
  - \( S \in NT \)
Example 1
A small formal language

$L_1 = \{ 0, 00, 1, 11 \}$

$G_1 = ( \{ 0, 1 \}, \{ S \}, \{ S \rightarrow 0, S \rightarrow 00, S \rightarrow 1, S \rightarrow 11 \}, S )$
Example 2
A small fragment of English

\[ L_2 = \{ \text{the dog chased the dog, the dog chased a dog, a dog chased the dog, a dog chased a dog, the dog chased the cat, \ldots } \} \]

\[ G_2 = (\{a, \text{the, dog, cat, chased}\}, \]
\[ \{S, \text{NP, VP, Det, N, V}\}, \]
\[ \{S \rightarrow \text{NP VP, NP } \rightarrow \text{Det N, Det } \rightarrow \text{a | the, N } \rightarrow \text{dog | cat, VP } \rightarrow \text{V | V NP, V } \rightarrow \text{chased}\}, \]
\[ S \} \]

Notes: \( S = \text{Sentence, NP = Noun Phrase, N = Noun, VP = Verb Phrase, V = Verb, Det = Determiner} \)
Language terminology
(from Sebesta, p. 125)

• A language is a set of strings of symbols, drawn from some finite set of symbols (called the alphabet of the language).
• “The strings of a language are called sentences”
• “Formal descriptions of the syntax […] do not include descriptions of the lowest-level syntactic units […] called lexemes.”
• “A token of a language is a category of its lexemes.”
• Syntax of a programming language is often presented in two parts:
  – regular grammar for token structure (e.g. structure of identifiers)
  – context-free grammar for sentence structure
Examples of *lexemes* and *tokens*

<table>
<thead>
<tr>
<th>Lexemes</th>
<th>Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>identifier</td>
</tr>
<tr>
<td>i</td>
<td>identifier</td>
</tr>
<tr>
<td>sum</td>
<td>identifier</td>
</tr>
<tr>
<td>-3</td>
<td>integer_literal</td>
</tr>
<tr>
<td>10</td>
<td>integer_literal</td>
</tr>
<tr>
<td>1</td>
<td>integer_literal</td>
</tr>
<tr>
<td>;</td>
<td>statement_separator</td>
</tr>
<tr>
<td>=</td>
<td>assignment_operator</td>
</tr>
</tbody>
</table>
Backus-Naur Form (BNF)

• Backus-Naur Form (1959)
  – Invented by John Backus to describe ALGOL 58, modified by Peter Naur for ALGOL 60
  – BNF is equivalent to context-free grammar
  – BNF is a metalanguage used to describe another language, the object language
  – Extended BNF: adds syntactic sugar to produce more readable descriptions
BNF Fundamentals

- Sample rules [p. 128]
  
  \[
  \begin{align*}
  \text{<assign>} & \rightarrow \text{<var>} = \text{<expression>} \\
  \text{<if_stmt>} & \rightarrow \text{if } \text{<logic_expr>} \text{ then } \text{<stmt>} \\
  \text{<if_stmt>} & \rightarrow \text{if } \text{<logic_expr>} \text{ then } \text{<stmt>} \text{ else } \text{<stmt>}
  \end{align*}
  \]

- non-terminals/tokens surrounded by < and >
- lexemes are not surrounded by < and >
- keywords in language are in **bold**
- → separates LHS from RHS
- | expresses alternative expansions for LHS
  
  \[
  \begin{align*}
  \text{<if_stmt>} & \rightarrow \text{if } \text{<logic_expr>} \text{ then } \text{<stmt>} \\
  & \mid \text{if } \text{<logic_expr>} \text{ then } \text{<stmt>} \text{ else } \text{<stmt>}
  \end{align*}
  \]

- = is in this example a lexeme
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 often given simply as a set of rules (terminal and non-terminal sets are implicit in rules, as is start symbol)