Describing Syntax and Semantics

CS 315 – Programming Languages
Pinar Duygulu
Bilkent University
Introduction

Providing a **precise description** of a programming language is important.

**Reasons:**
- Diversity of the people who need to understand
- Language implementors must determine how the expressions, statements, etc are formed, and their intended effects – clear description of language make their job easy
- Language users must understand the language by referring to the language manual

**ALGOL 60** was the **first** language with a precise description.
Introduction

**Syntax** of a PL: the *form* of its expressions, statements, and program units.

**Semantics** of a PL: the *meaning* of those expressions, statements, and program units

E.g. while statement in Java
syntax: `while (<boolean_expr>) <statement>`
semantics: when `boolean_expr` is true it will be executed

The meaning of a statement should be clear from its syntax
The general problem of describing syntax

**Language**: a set of strings of characters from some alphabet. Natural Languages/ Programming Languages/Formal Languages
Ex: English, Turkish / Pascal, C, FORTRAN / a*b*, 0n1n

**Strings of a language**: sentence / program (statement) / word

**Alphabet**: $\Sigma$, All strings: $\Sigma^*$, Language: $L \subseteq \Sigma^*$

**Syntax rules** specify which strings from $\Sigma^*$ are in the language.
Lexemes

Lower level constructs are given not by the syntax but by lexical specifications. These are called lexemes.

Examples: identifiers, constants, operators, special words.

total, sum_of_products, 1254, ++, (: 

So, a language is considered as a set of strings of lexemes rather than strings of chars.
Tokens

• A **token** of a language is a category of its lexemes.

• For example, **identifier** is a token which may have lexemes *sum* and *total*
Example in Java language

\[ x = (y + 3.1) \times z_5; \]

<table>
<thead>
<tr>
<th>Lexemes</th>
<th>Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>identifier</td>
</tr>
<tr>
<td>=</td>
<td>equal_sign</td>
</tr>
<tr>
<td>(</td>
<td>left_paren</td>
</tr>
<tr>
<td>)</td>
<td>right_paren</td>
</tr>
<tr>
<td>for</td>
<td>for</td>
</tr>
<tr>
<td>y</td>
<td>identifier</td>
</tr>
<tr>
<td>+</td>
<td>plus_op</td>
</tr>
<tr>
<td>3.1</td>
<td>float_literal</td>
</tr>
<tr>
<td>*</td>
<td>mult_op</td>
</tr>
<tr>
<td>z_5</td>
<td>identifier</td>
</tr>
<tr>
<td>;</td>
<td>semi_colon</td>
</tr>
</tbody>
</table>
Describing Syntax

• Higher level constructs are given by syntax rules.
• Examples: organization of the program, loop structures, assignment, expressions, subprogram definitions, and calls.
Elements of Syntax

• An alphabet of symbols
• Symbols are terminal and non-terminal
  – Terminals cannot be broken down
  – Non-terminals can be broken down further
• Grammar rules that express how symbols are combined to make legal sentences
• Rules are of the general form
  non-terminal symbol ::= list of zero or more terminals or non-terminals
• One uses rules to recognize (parse) or generate legal sentences
Recognizers vs Generators

- Automata (accept or reject) if input string is in the language
- Grammars (set of rules) easy to understand by humans
Formal Methods for Describing Syntax

- Noam Chomsky – linguist - 1950s
  - define four classes of languages

Programming languages are contained in the class of CFL’s.

ALGOL58 John Backus
ALGOL 60 Peter Naur

**Backus-Naur form**: A notation to describe the syntax of programming languages.
Fundamentals

• A **metalanguage** is a language used to describe another language.
• **BNF** (Backus-Naur Form) is a metalanguage used to describe PL’s.
• BNF uses abstractions for syntactic structures.

• `<LHS> → <RHS>`
• **LHS**: abstraction being defined
• **RHS**: definition

• Note: Sometimes ::= is used for →
• Example, Java assignment statement can be represented by the abstraction `<assign>`. Then the assignment statement of Java can be defined in BNF as

• `<assign> → <var> = <expression>`

• Such a definition is called a **rule** or **production**.

• Here, `<var>` and `<expression>` must also be defined.

• an instance of this abstraction can be 
  `total = sub1 + sub2`
Fundamentals

• These abstractions are called Variables, or Nonterminals of a Grammar.
• Grammar is simply a collection of rules.
• Lexemes and tokens are the Terminals of a grammar.
*An initial example*

- Consider the sentence
  - “Marry greets John”

- A simple grammar for it

\[
\begin{align*}
\text{<sentence>} & ::= \text{<subject>}\text{<predicate>} \\
\text{<subject>} & ::= \text{Mary} \\
\text{<predicate>} & ::= \text{<verb>}\text{<object>} \\
\text{<verb>} & ::= \text{greets} \\
\text{<object>} & ::= \text{John}
\end{align*}
\]
**Alternation**

- Multiple definitions can be separated by | to mean OR.

\[\text{<object>} ::= \text{John} | \text{Alfred}\]

This adds “Marry greets Alfred” to legal sentences

\[\text{<subject>} ::= \text{Marry} | \text{John}\]
\[\text{<object>} ::= \text{Marry} | \text{John}\]

**Alternatively**

\[\text{<sentence>} ::= \text{<subject>}<\text{predicate}>\]
\[\text{<subject>} ::= \text{noun}\]
\[\text{<predicate>} ::= \text{<verb>}<\text{object}>\]
\[\text{<verb>} ::= \text{greets}\]
\[\text{<object>} ::= \text{<noun>}\]
\[\text{<noun>} ::= \text{John} | \text{Mary}\]
*Infinite number of Sentences*

\[
<object> ::= \begin{align*}
& \text{John} \\ & \text{John again} \\ & \text{John again and again} \\ & \ldots
\end{align*}
\]

Instead use recursive definition

\[
<object> ::= \begin{align*}
& \text{John} \\ & \text{John <repeat factor>}
\end{align*}
\]

\[
<\text{repeat factor}> ::= \begin{align*}
& \text{again} \\ & \text{again and <repeat factor>}
\end{align*}
\]

- A rule is recursive if its LHS appears in its RHS
*Simple example for PLs*

\[
\begin{align*}
\langle \text{number} \rangle & ::= & \langle \text{number} \rangle & \langle \text{digit} \rangle & \mid & \langle \text{digit} \rangle \\
\langle \text{signed number} \rangle & ::= & + & \langle \text{number} \rangle & & - & \langle \text{number} \rangle
\end{align*}
\]
*Simple example for PLs*

- How you can describe simple arithmetic?

- `<expression> ::= <expr> <operator> <expr> | var`
- `<op> ::= + | - | * | /`
- `<var> ::= a | b | c | ...`
- `<var> ::= <signed number>`
*All numbers*

- $S := '-' FN | FN$
- $FN := DL | DL '.' DL$
- $DL := D | D DL$
- $D := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'$
*Identifiers*

\[
\text{<identifier> } \rightarrow \text{ <letter> }
\]

\[
| \text{ <identifier><letter> }
\]

\[
| \text{ <identifier><digit> }
\]
PASCAL/Ada If Statement

\[ \text{<ifStmt>} \rightarrow \text{if } \text{<logicExpr>} \text{ then } \text{<stmt>} \]

\[ \text{<ifStmt>} \rightarrow \text{if } \text{<logicExpr>} \text{ then } \text{<stmt>} \]

\[ \text{else } \text{<stmt>} \]

or

\[ \text{<ifStmt>} \rightarrow \text{if } \text{<logicExpr>} \text{ then } \text{<stmt>} \]

\[ | \text{if } \text{<logicExpr>} \text{ then } \text{<stmt>} \text{ else } \text{<stmt>} \]
Grammars and Derivations

• A grammar is a generative device for defining languages

• The sentences of the language are generated through a sequence of applications of the rules, starting from the special nonterminal called start symbol.

• Such a generation is called a derivation.

• Start symbol represents a complete program. So it is named <program>.
Example grammar

<program> → begin <stmt_list> end
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>
<stmt> → <var> := <expression>
<var> → A | B | C
<expression> → <var>
| <var><arith_op> <var>
<arith_op> → + | - | * | /
Derivations

• In order to check if a given string represents a valid program in the language, we try to derive it in the grammar.

• Example string:

  \[ \text{begin A := B; C := A * B end;} \]

• Derivation starts from the start symbol \(<\text{program}>\).

• At each step we replace a nonterminal with its definition (RHS of the rule).
Example

If always the leftmost nonterminal is replaced, then it is called leftmost derivation.
Another example

\[ \texttt{<assign>} ::= \texttt{id} = \texttt{<exp>} \]
\[ \texttt{id} ::= A | B | C \]
\[ \texttt{<expr>} ::= \texttt{id} + \texttt{<exp>} \]
\[ \hspace{1em} | \texttt{id} * \texttt{<expr>} \]
\[ \hspace{1em} | (\texttt{<expr>}) \]
\[ \hspace{1em} | \texttt{id} \]
Derivation

A = B * (A + C)

<assign>  => <id> = <expr>
=> A = <expr>
=> A = <id> * <expr>
=> A = B * <expr>
=> A = B * (<id> + <expr>)
=> A = B * (A + <expr>)
=> A = B * (A + <id>)
=> A = B * (A+C)
Parse Trees

- Grammars naturally describe the hierarchical syntactic structure of the sentences of the languages that they define.
- These hierarchical structures are called parse trees.
- Every internal node is a nonterminal, and every leaf is a terminal symbol.
- A derivation can also be represented by a parse tree.
- In fact, a parse tree represents many derivations.
Parse trees

```
<program>
  begin  <stmt_list>  end
/
<stmt>
  ;
/
<var> := <expression>
  A
  B
<var>
C <var><arith_op> <var>
  A  *
  C
<stmt_list>
```
A parse tree for the simple statement $A = B \times (A + C)$
Ambiguous Grammar

A grammar that generates a sentential form for which there are two or more distinct parse trees is called as ambiguous

<assign> ::= <id> = <expr>
{id} ::= A | B | C
<expr> ::= <expr> + <expr>
    | <expr> * <expr>
    | (<expr>)
    | <id>
Two distinct parse trees for the same sentence, \( A = B + C \ast A \)
Ambiguity

The grammar of a PL must not be ambiguous

There are solutions for correcting the ambiguity

- Operator precedence
- Associativity rules
Operator precedence

In mathematics *, operation has a higher precedence than +. This can be implemented with extra nonterminals

<assign> ::= <id> = <expr>
{id} ::= A | B | C
<expr> ::= <expr> + <term>
    | <term>
<term> ::= <term> * <factor>
    | <factor>
<factor> ::= (<expr>)
    | <id>
A unique parse tree for $A = B + C \ast A$ using an unambiguous grammar
Leftmost derivation using unambiguous grammar

\[
\begin{align*}
<\text{assign}> &\Rightarrow <\text{id}> = <\text{expr}> \\
&\Rightarrow A = <\text{expr}> \\
&\Rightarrow A = <\text{expr}> + <\text{term}> \\
&\Rightarrow A = <\text{term}> + <\text{term}> \\
&\Rightarrow A = <\text{factor}> + <\text{term}> \\
&\Rightarrow A = <\text{id}> + <\text{term}> \\
&\Rightarrow A = B + <\text{term}> \\
&\Rightarrow A = B + <\text{term}> * <\text{factor}> \\
&\Rightarrow A = B + <\text{factor}> * <\text{factor}> \\
&\Rightarrow A = B + <\text{id}> * <\text{factor}> \\
&\Rightarrow A = B + C * <\text{factor}> \\
&\Rightarrow A = B + C * <\text{id}> \\
&\Rightarrow A = B + C * A
\end{align*}
\]
Rightmost derivation using unambiguous grammar

<assign> => <id> = <expr>

=> <id> = <expr> + <term>

=> <id> = <expr> + <term> * <factor>

=> <id> = <expr> + <term> * <id>

=> <id> = <expr> + <term> * A

=> <id> = <expr> + <factor> * A

=> <id> = <expr> + <id> * A

=> <id> = <expr> + C * A

=> <id> = <term> + C * A

=> <id> = <factor> + C * A

=> <id> = <id> + C * A

=> <id> = B + C * A

=> A = B + C * A
Associativity of Operators

What about equal precedence operators?

In math addition and multiplication are associative

\[ A + B + C = (A + B) + C = A + (B + C) \]

However computer arithmetic may not be associative

e.g: for floating point addition where floating points values store 7 digits of accuracy, adding eleven numbers together where one of the numbers is \(10^7\) and the others are 1 result would be \(1.000001 \times 10^7\) only if the ten 1s are added first

Subtraction and division are not associative

\[ A / B / C / D = ? \quad ((A / B) / C) / D \neq A / (B / (C / D)) \]
In a BNF rule, if the LHS appears at the beginning of the RHS, the rule is said to be left recursive

Left recursion specifies left associativity

<expr> ::= <expr> + <term>  
    | <term>

Similar for the right recursion

In most of the languages exponentiation is defined as a right associative operation

<factor> ::= <expr> ** <factor>  
    | <expr>  
<expr> ::= (<expr>)  
    | <id>
A parse tree for \( A = B + C + A \) illustrating the associativity of addition

Left associativity
Left addition is lower than the right addition
Two distinct parse trees for the same sentential form

\[
\text{<if_stmt>} ::= \text{if <logic_expr>} \text{ then <stmt>} \\
\quad | \quad \text{if <logic_expr>} \text{ then <stmt>} \text{ else <stmt>}
\]

If C1 then if C2 then A else B
An Unambiguous grammar for if then else

To design an unambiguous if-then-else statement we have to decide which if a dangling else belongs to

Dangling else problem: there are more if then else
Most PL adopt the following rule:
“an else is matched with the closest previous unmatched if statement”
(unmatched if = else-less if)

<stmt> ::= <matched> | <unmatched>
<matched> ::= if <logic_expr> then <matched> else <matched>
            | any non-if-statement
<unmatched> ::= if <logic_expr> then <stmt>
               | if <logic_expr> then <matched> else <unmatched>

there is a unique parse tree for this if statement
**BNF and Extended BNF**

**EBNF:** same power but more convenient

[X] : X is optional (0 or 1 occurrence)
Equivalent to X|empty

<writeln> ::= WRITELN [(<item_list>)]

<selection> ::= if (<expression>)<statement>[else<statement>]

{X} : 0 or more occurrences

A::={X} is equivalent to A::=XA|empty

<identlist> = <identifier> {,<identifier>}

{X1|X2|X3} : choose X1 or X2 or X3

A::=B(X|Y)C is equal to A::=AXC | AYC

<for_stmt> ::= for <var>::=<exp>(to|downto) <exp> do <stmt>

<term>::=<term>(*|/|%)<factor>
**BNF and Extended BNF**

BNF:

<expr> ::= <expr> + <term>
   | <expr> - <term>
   | <term>

<term> ::= <term> * <factor>
   | <term> / <factor>
   | <factor>

<factor> ::= <expr> ** <factor>
   | <expr>

<expr> ::= (<expr>)
   | <id>
BNF and Extended BNF

< expr> ::= <term> {(+ | -) <term>}
<term> ::= <factor> {(*|/) <factor>}
<factor> ::= <expr> {**<expr>}
<expr>::=(<expr>)
    | id
*Extended BNF*

- `<number> ::= \{ <digit> \}`
- `<signed number> ::= [+ | - ] <number>`
*Syntax Graphs*

- Another form for representation of PL syntax
- Syntax graphs = syntax diagrams = syntax charts
- Equivalent to BNF in the power of representation, but easier to understand
- A separate graph is given for each syntactic unit (for each nonterminal)
- A rectangle represents a nonterminal, contains the name of the syntactic unit
- A circle (ellipse) represents a terminal
A syntax graph consists of one entry and one or more exit points. If there exists a path from the input entry to any of the exit points corresponding to the string, then the string represents a valid instance of that unit. There may exist loops in the path.