22c:111 Programming Language Concepts

Fall 2008

Syntax I

These notes were originally developed by Allen Tucker, Robert Noonan and modified by Cesare Tinelli. They are copyrighted materials and may not be used in other course settings outside of the University of Iowa in their current form or modified form without the express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.
Contents

2.1 Grammars
   2.1.1 Backus-Naur Form
   2.1.2 Derivations
   2.1.3 Parse Trees
   2.1.4 Associativity and Precedence
   2.1.5 Ambiguous Grammars

2.2 Extended BNF

2.3 Syntax of a Small Language: Clite
   2.3.1 Lexical Syntax
   2.3.2 Concrete Syntax

2.4 Compilers and Interpreters

2.5 Linking Syntax and Semantics
   2.5.1 Abstract Syntax
   2.5.2 Abstract Syntax Trees
   2.5.3 Abstract Syntax of Clite
Thinking about Syntax

The *syntax* of a programming language is a precise description of all its grammatically correct programs.

Precise syntax was first used with Algol 60, and has been used ever since.

Three levels:

- *Lexical syntax*
- *Concrete syntax*
- *Abstract syntax*
Levels of Syntax

Lexical syntax = all the basic symbols of the language (names, values, operators, etc.)
Concrete syntax = rules for writing expressions, statements and programs.
Abstract syntax = internal representation of the program, favoring content over form. E.g.,

- C: \texttt{if (expr) ... discard ()}
- Ada: \texttt{if (expr) then discard then}
2.1 Grammars

A *metalanguage* is a language used to define other languages.

A *grammar* is a metalanguage used to define the syntax of a language.

*Our interest*: using grammars to define the syntax of a programming language.
2.1.1 Backus-Naur Form (BNF)

- Stylized version of a context-free grammar (cf. Chomsky hierarchy)
- Sometimes called Backus Normal Form
- First used to define syntax of Algol 60
- Now used to define syntax of most major languages
BNF Grammar

Set of productions: $P$

terminal symbols: $T$

nonterminal symbols: $N$

start symbol: $S \in N$

A production has the form

$$A \rightarrow \omega$$

where $A \in N$ and $\omega \in (N \cup T)^*$
Example: Binary Digits

Consider the grammar:

\[
\text{binaryDigit} \rightarrow 0 \\
\text{binaryDigit} \rightarrow 1
\]

or equivalently:

\[
\text{binaryDigit} \rightarrow 0 \mid 1
\]

Here, \( \mid \) is a metacharacter that separates alternatives.
2.1.2 Derivations

Consider the grammar:

\[ \text{Integer} \rightarrow \text{Digit} \mid \text{Integer} \text{ Digit} \]
\[ \text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

We can derive any unsigned integer, like 352, from the start symbol \text{Integer} in this grammar.
Derivation of 352 as an *Integer*

A 6-step process, starting with:

*Integer*
Derivation of 352 (step 1)

Use a grammar rule to enable each step:

\[ \text{Integer} \Rightarrow \text{Integer Digit} \]
Derivation of 352 (steps 1-2)

Replace a nonterminal by a right-hand side of one of its rules:

\[
\text{Integer} \Rightarrow \text{Integer Digit} \\
\Rightarrow \text{Integer } 2
\]
Derivation of 352 (steps 1-3)

Each step follows from the one before it.

\[ \text{Integer} \Rightarrow \text{Integer Digit} \Rightarrow \text{Integer 2} \Rightarrow \text{Integer Digit 2} \]
Derivation of 352 (steps 1-4)

\[ \text{Integer} \Rightarrow \text{Integer Digit} \]
\[ \Rightarrow \text{Integer 2} \]
\[ \Rightarrow \text{Integer Digit 2} \]
\[ \Rightarrow \text{Integer 5 2} \]
Derivation of 352 (steps 1-5)

\[\begin{align*}
\text{Integer} & \Rightarrow \text{Integer Digit} \\
& \Rightarrow \text{Integer 2} \\
& \Rightarrow \text{Integer Digit 2} \\
& \Rightarrow \text{Integer 5 2} \\
& \Rightarrow \text{Digit 5 2}
\end{align*}\]
Derivation of 352 (steps 1-6)

You know you’re finished when there are only terminal symbols remaining.

\[
\begin{align*}
\text{Integer} & \Rightarrow \text{Integer Digit} \\
& \Rightarrow \text{Integer 2} \\
& \Rightarrow \text{Integer Digit 2} \\
& \Rightarrow \text{Integer 5 2} \\
& \Rightarrow \text{Digit 5 2} \\
& \Rightarrow 3 5 2
\end{align*}
\]
A Different Derivation of 352

\[
\begin{align*}
\text{Integer} & \Rightarrow \text{Integer Digit} \\
& \Rightarrow \text{Integer Digit Digit} \\
& \Rightarrow \text{Digit Digit Digit} \\
& \Rightarrow 3 \text{ Digit Digit} \\
& \Rightarrow 3 \text{ 5 Digit} \\
& \Rightarrow 3 \text{ 5 2}
\end{align*}
\]

This is called a \textit{leftmost derivation}, since at each step the leftmost nonterminal is replaced.
(The first one was a \textit{rightmost derivation}.)
Notation for Derivations

$\text{Integer} \Rightarrow^* 352$

Means that 352 can be derived in a finite number of steps using the grammar for $\text{Integer}$.

$352 \in L(G)$

Means that 352 is a member of the language defined by grammar $G$.

$L(G) = \{ \omega \in T^* | \text{Integer} \Rightarrow^* \omega \}$

Means that the language defined by grammar $G$ is the set of all symbol strings $\omega$ that can be derived as an $\text{Integer}$.
2.1.3 Parse Trees

A *parse tree* is a graphical representation of a derivation.

*Each internal node of the tree corresponds to a step in the derivation.*

*Each child of a node represents a right-hand side of a production.*

*Each leaf node represents a symbol of the derived string, reading from left to right.*
E.g., The step $\text{Integer} \Rightarrow \text{Integer Digit}$ appears in the parse tree as:

```
        Integer
       /   \
   Integer  Digit
```
Parse Tree for 352 as an \textit{Integer}

Figure 2.1
Arithmetic Expression Grammar

The following grammar defines the language of arithmetic expressions with 1-digit integers, addition, and subtraction.

\[
Expr \rightarrow Expr + Term \mid Expr - Term \mid Term
\]

\[
Term \rightarrow 0 \mid \ldots \mid 9 \mid (Expr)
\]
Parse of the String 5-4+3

Figure 2.2
2.1.4 Associativity and Precedence

A grammar can be used to define associativity and precedence among the operators in an expression.

E.g., + and - are left-associative operators in mathematics;
* and / have higher precedence than + and -.

Consider the more interesting grammar \( G_1 \):

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Expr} + \text{Term} \mid \text{Expr} - \text{Term} \mid \text{Term} \\
\text{Term} & \rightarrow \text{Term} \ast \text{Factor} \mid \text{Term} / \text{Factor} \\
& \quad \mid \text{Term} \% \text{Factor} \mid \text{Factor} \\
\text{Factor} & \rightarrow \text{Primary} \ast \ast \text{Factor} \mid \text{Primary} \\
\text{Primary} & \rightarrow 0 \mid \ldots \mid 9 \mid ( \text{Expr} )
\end{align*}
\]
Parse of $4**2**3+5*6+7$ for Grammar $G_1$

**Figure 2.3**
### Associativity and Precedence for Grammar $G_1$

**Table 2.1**

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Associativity</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>right</td>
<td>**</td>
</tr>
<tr>
<td>2</td>
<td>left</td>
<td>* / %</td>
</tr>
<tr>
<td>1</td>
<td>left</td>
<td>+ -</td>
</tr>
</tbody>
</table>

**Note:** These relationships are shown by the structure of the parse tree: highest precedence at the bottom, and left-associativity on the left at each level.
2.1.5 Ambiguous Grammars

A grammar is *ambiguous* if one of its strings has two or more different parse trees (equivalently, if it has two or more different left-most derivations).

_E.g._, Grammar $G_1$ above is _not_ ambiguous.

C, C++, and Java have a large number of

- operators and
- precedence levels

Instead of using a large grammar, we can:

- _Write a smaller ambiguous grammar, and_
- _Give separate precedence and associativity (e.g., Table 2.1)"_
An Ambiguous Expression Grammar $G_2$

$$Expr \rightarrow Expr\ Op\ Expr \mid (\ Expr) \mid Integer$$

$$Op \rightarrow + \mid - \mid * \mid / \mid \% \mid **$$

Notes:

- $G_2$ is equivalent to $G_1$. I.e., its language is the same.
- $G_2$ has fewer productions and nonterminals than $G_1$.
- However, $G_2$ is ambiguous.
Ambiguous Parse of 5-4+3
Using Grammar $G_2$

Figure 2.4
The Dangling Else

Block → \{ Statements \}

Statements → Statements Statement | Statement

Statement → Assignment | IfStatement | Block

IfStatement → if ( Expression ) Statement | if ( Expression ) Statement else Statement
Example

With which ‘if’ does the following ‘else’ associate

```java
if (x < 0)
    if (y < 0) y = y - 1;
else y = 0;
```

Answer: *either one!*
The **Dangling Else Ambiguity**

**Figure 2.5**

```
IfStatement
  if ( Expression ) Statement
    x<0
    y<0 y = y-1;
  y = y-1;

IfStatement
  if ( Expression ) Statement else Statement
    x<0
    y = 0;

IfStatement
  if ( Expression ) Statement
    x<0
    y<0 y = y-1;

IfStatement
  if ( Expression ) Statement
    y = 0;
```
Solving the dangling else ambiguity

1. Algol 60, C, C++: associate each else with closest if; use {} or begin...end to override.

2. Algol 68, Modula, Ada: use explicit delimiter to end every conditional (e.g., if...fi)

3. Java: rewrite the grammar to limit what can appear in a conditional:

   \[
   \begin{align*}
   \text{IfThenStatement} & \rightarrow \text{if ( Expression ) Statement} \\
   \text{IfThenElseStatement} & \rightarrow \text{if ( Expression ) StatementNoShortIf} \\
   & \quad \text{else Statement}
   \end{align*}
   \]

   The category \text{StatementNoShortIf} includes all except \text{IfThenStatement}. 
2.2 Extended BNF (EBNF)

BNF:

- recursion for iteration
- nonterminals for grouping

EBNF: additional metacharacters

- {  } for a series of zero or more
- (  ) for a list, must pick one
- [  ] for an optional list; pick none or one
EBNF Examples

*Expression* is a list of one or more *Terms* separated by operators + and -

\[
Expression \rightarrow Term \{ ( + \mid - ) Term \}
\]

*IfStatement* → if ( *Expression* ) *Statement* [ else *Statement* ]

C-style EBNF lists alternatives vertically and uses _opt_ to signify optional parts. E.g.,

*IfStatement*:

if ( *Expression* ) *Statement* _ElsePart_ ~opt

*ElsePart*:

else *Statement*
EBNF to BNF

We can always rewrite an EBNF grammar as a BNF grammar. E.g.,

\[ A \rightarrow x \{ y \} z \]

can be rewritten:

\[ A \rightarrow x A' z \]
\[ A' \rightarrow \varepsilon \mid y A' \]

(Rewriting EBNF rules with ( ), [ ] is left as an exercise.)

While EBNF is no more powerful than BNF, its rules are often simpler and clearer.