CSCI312 Principles of Programming Languages

Chapter 2
Syntax

Xu Liu
Project: derive

Work in pairs: follow the rule

Terminal string length: number of terminals in the derivation:

e.g., ( id ) has length 3

( id + id ) has length 5
Recap
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 $\omega \in (N \cup T)^*$
Derivation of 352 from a BNF Grammar

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

\[
\begin{align*}
\text{Integer} & \Rightarrow \text{Integer Digit} \\
& \Rightarrow \text{Integer Digit Digit} \\
& \Rightarrow \text{Digit Digit Digit} \\
& \Rightarrow 3 \text{ Digit Digit} \\
& \Rightarrow 3 5 \text{ Digit} \\
& \Rightarrow 3 5 2
\end{align*}
\]
Parse Tree for 352 as an Integer

Figure 2.1
Ambiguity

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

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

Answer: either one!
The *Dangling Else* Ambiguity

**Figure 2.5**

```
if (Expression) Statement
  x<0
  y<0  y = y-1;
  y = y-1;
```
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
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
2.3 Syntax of a Small Language: *Clite*

Motivation for using a subset of C:

<table>
<thead>
<tr>
<th>Language</th>
<th>Grammar (pages)</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal</td>
<td>5</td>
<td>Jensen &amp; Wirth</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>Kernighan &amp; Richie</td>
</tr>
<tr>
<td>C++</td>
<td>22</td>
<td>Stroustrup</td>
</tr>
<tr>
<td>Java</td>
<td>14</td>
<td>Gosling, et. al.</td>
</tr>
</tbody>
</table>

The *Clite* grammar fits on one page (next 3 slides), so it’s a far better tool for studying language design.
Fig. 2.7  *Clite* Grammar: Statements

\[
\begin{align*}
\text{Program} & \rightarrow \text{int main ( ) } \{ \text{Declarations Statements} \} \\
\text{Declarations} & \rightarrow \{ \text{Declaration} \} \\
\text{Declaration} & \rightarrow \text{Type Identifier [ [ Integer ] ] } \{ \text{, Identifier [ [ Integer ] ] } \} \\
\text{Type} & \rightarrow \text{int | bool | float | char} \\
\text{Statements} & \rightarrow \{ \text{Statement} \} \\
\text{Statement} & \rightarrow ; \mid \text{Block} \mid \text{Assignment} \mid \text{IfStatement} \mid \text{WhileStatement} \\
\text{Block} & \rightarrow \{ \text{Statements} \} \\
\text{Assignment} & \rightarrow \text{Identifier [ [ Expression ] ] = Expression ;} \\
\text{IfStatement} & \rightarrow \text{if ( Expression ) Statement [ else Statement ]} \\
\text{WhileStatement} & \rightarrow \text{while ( Expression ) Statement}
\end{align*}
\]
Fig. 2.7 *Clite* Grammar: Expressions

Expression $\rightarrow$ Conjunction \{ | | Conjunction \}  
Conjunction $\rightarrow$ Equality \{ && Equality \} 

Equality $\rightarrow$ Relation [ EquOp Relation ]  
EquOp $\rightarrow$ == | !=  
Relation $\rightarrow$ Addition [ RelOp Addition ]  
RelOp $\rightarrow$ < | <= | > | >=  
Addition $\rightarrow$ Term \{ AddOp Term \}  
AddOp $\rightarrow$ + | -  
Term $\rightarrow$ Factor \{ MulOp Factor \}  
MulOp $\rightarrow$ * | / | %  
Factor $\rightarrow$ [ UnaryOp ] Primary  
UnaryOp $\rightarrow$ - | !  
Primary $\rightarrow$ Identifier [ [ Expression ] ] | Literal | ( Expression ) | Type ( Expression )
Fig. 2.7  *Clite* grammar: lexical level

\[
\text{Identifier} \rightarrow \text{Letter} \{ \text{Letter} \mid \text{Digit} \}
\]

\[
\text{Letter} \rightarrow a \mid b \mid \ldots \mid z \mid A \mid B \mid \ldots \mid Z
\]

\[
\text{Digit} \rightarrow 0 \mid 1 \mid \ldots \mid 9
\]

\[
\text{Literal} \rightarrow \text{Integer} \mid \text{Boolean} \mid \text{Float} \mid \text{Char}
\]

\[
\text{Integer} \rightarrow \text{Digit} \{ \text{Digit} \}
\]

\[
\text{Boolean} \rightarrow \text{true} \mid \text{false}
\]

\[
\text{Float} \rightarrow \text{Integer} \cdot \text{Integer}
\]

\[
\text{Char} \rightarrow ' \text{ASCII Char} '\n\]
Issues Not Addressed by this Grammar

• Comments
• Whitespace
• Distinguishing one token ≤ from two tokens < =
• Distinguishing identifiers from keywords like if

These issues are addressed by identifying two levels:

– lexical level
– syntactic level
2.3.1 Lexical Syntax

*Input*: a stream of characters from the ASCII set, keyed by a programmer.

*Output*: a stream of *tokens* or basic symbols, classified as follows:

- **Identifiers** e.g., Stack, x, i, push
- **Literals** e.g., 123, 'x', 3.25, true
- **Keywords** bool char else false float if int main true while
- **Operators** = || && == != < <= > >= + - * / !
- **Punctuation** ; , { } ( )
Whitespace

Whitespace is any space, tab, end-of-line character (or characters), or character sequence inside a comment. No token may contain embedded whitespace (unless it is a character or string literal).

Example:

\[ > = \textit{one token} \]
\[ > = \textit{two tokens} \]
Whitespace Examples in Pascal

while a < b do  legal - spacing between tokens
while a < b do  spacing not needed for <
while a < b do  illegal - can’t tell boundaries
while a < b do  between tokens
Comments

Not defined in grammar

*Clite* uses // comment style of C++
Identifier

Sequence of letters and digits, starting with a letter

if is both an identifier and a keyword

Most languages require identifiers to be distinct from keywords

In some languages, identifiers are merely predefined (and thus can be redefined by the programmer)
Redefining Identifiers can be dangerous

program confusing;
const true = false;
begin
    if (a<b) = true then
        f(a)
    else ...

Should Identifiers be case-sensitive?

Older languages: no

– *Pascal*: no
– *Modula*: yes
– *C, C++*: yes
– *Java*: yes
– *PHP*: partly yes, partly no
2.3.2 Concrete Syntax

Based on a parse of its *Tokens* ; is a statement terminator  

*(Algol-60, Pascal use ; as a separator)*

Rule for *IfStatement* is ambiguous:

“The else ambiguity is resolved by connecting an *else* with the last encountered else-less if.”

[Stroustrup, 1991]
Expressions in *Clite*

13 grammar rules

Use of meta braces – operators are left associative

C++ expressions require 4 pages of grammar rules
  [Stroustrup]

C uses an ambiguous expression grammar
  [Kernighan and Ritchie]
## Associativity and Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary - !</td>
<td>none</td>
</tr>
<tr>
<td>* /</td>
<td>left</td>
</tr>
<tr>
<td>+ -</td>
<td>left</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>none</td>
</tr>
<tr>
<td>== !=</td>
<td>none</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Clite Equality, Relational Operators**

... are non-associative.

(an idea borrowed from Ada)

Why is this important?

In C++, the expression:

```
if (a < x < b)
```

is *not* equivalent to

```
if (a < x && x < b)
```

But it is error-free!

So, what does it mean?
2.4 Compilers and Interpreters

Source Program → Lexical Analyzer → Syntactic Analyzer → Semantic Analyzer → Code Optimizer → Code Generator → Machine Code
Lexer

• Input: characters
• Output: tokens
• Separate:
  – Speed: 75% of time for non-optimizing compilers
  – Simpler design
  – Character sets
  – End of line conventions
Parser

- Based on BNF/EBNF grammar
- Input: tokens
- Output: abstract syntax tree (parse tree)
- Abstract syntax: parse tree with punctuation, many nonterminals discarded
Semantic Analysis

- Check that all identifiers are declared
- Perform type checking
- Insert implied conversion operators
  (i.e., make them explicit)
Code Optimization

• Evaluate constant expressions at compile-time
• Reorder code to improve cache performance
• Eliminate common subexpressions
• Eliminate unnecessary code
Code Generation

• Output: machine code
• Instruction selection
• Register management
• Peephole optimization
Interpreter

Replaces last 2 phases of a compiler

Input:
- *Mixed*: intermediate code
- *Pure*: stream of ASCII characters

Mixed interpreters
- *Java, Perl, Python, Haskell, Scheme*

Pure interpreters:
- *most Basics, shell commands*
2.5 Linking Syntax and Semantics

Output: parse tree is inefficient

Example: Fig. 2.9
Parse Tree for
\( z = x + 2*y; \)
Fig. 2.9
Finding a More Efficient Tree

The *shape* of the parse tree reveals the meaning of the program.

So we want a tree that removes its inefficiency and keeps its shape.

- Remove separator/punctuation terminal symbols
- Remove all trivial root nonterminals
- Replace remaining nonterminals with leaf terminals

Example: Fig. 2.10
Abstract Syntax Tree for
\[ z = x + 2*y; \]
Fig. 2.10
Abstract Syntax

Removes “syntactic sugar” and keeps essential elements of a language. E.g., consider the following two equivalent loops:

\begin{verbatim}
Pascal
while i < n do begin
  i := i + 1;
end;
\end{verbatim}

\begin{verbatim}
C/C++
while (i < n) {
  i = i + 1;
}
\end{verbatim}

The only essential information in each of these is 1) that it is a \textit{loop}, 2) that its terminating condition is i < n, and 3) that its body increments the current value of i.
Abstract Syntax of Clite Assignments

Assignment = Variable target; Expression source
Expression = VariableRef | Value | Binary | Unary
VariableRef = Variable | ArrayRef
Variable = String id
ArrayRef = String id; Expression index
Value = IntValue | BoolValue | FloatValue | CharValue
Binary = Operator op; Expression term1, term2
Unary = UnaryOp op; Expression term
Operator = ArithmeticOp | RelationalOp | BooleanOp
IntValue = Integer intValue
...

Abstract Syntax as Java Classes

abstract class Expression {
}
abstract class VariableRef extends Expression {
}
class Variable extends VariableRef {
    String id;
}
class Value extends Expression {
    ...
}
class Binary extends Expression {
    Operator op;
    Expression term1, term2;
}
class Unary extends Expression {
    UnaryOp op;
    Expression term;
}
Example Abstract Syntax Tree

*Binary node*

Abstract Syntax Tree for $x+2*y$ (Fig 2.13)
Remaining Abstract Syntax of *Clite* (Declarations and Statements)

Fig 2.14

\[
\begin{align*}
\text{Program} &= \text{Declarations \ depar \ Statements \ body;} \\
\text{Declarations} &= \text{Declaration}^* \\
\text{Declaration} &= \text{VariableDecl} \mid \text{ArrayDecl} \\
\text{VariableDecl} &= \text{Variable \ v; Type \ t} \\
\text{ArrayDecl} &= \text{Variable \ v; Type \ t; Integer \ size} \\
\text{Type} &= \text{int \mid bool \mid float \mid char} \\
\text{Statements} &= \text{Statement}^* \\
\text{Statement} &= \text{Skip \mid Block \mid Assignment \mid Conditional \mid Loop} \\
\text{Skip} &= \\
\text{Block} &= \text{Statements} \\
\text{Conditional} &= \text{Expression \ test; Statement \ thenbranch, elsebranch} \\
\text{Loop} &= \text{Expression \ test; Statement \ body}
\end{align*}
\]