Syntax II
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) Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal</td>
<td>5: Jensen &amp; Wirth</td>
</tr>
<tr>
<td>C</td>
<td>6: Kernighan &amp; Richie</td>
</tr>
<tr>
<td>C++</td>
<td>22: Stroustrup</td>
</tr>
<tr>
<td>Java</td>
<td>14: 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

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

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

\[
\text{Identifier} \rightarrow \text{Letter} \{ \text{Letter} \mid \text{Digit} \} \\
\text{Letter} \rightarrow \text{a} \mid \text{b} \mid \ldots \mid \text{z} \mid \text{A} \mid \text{B} \mid \ldots \mid \text{Z} \\
\text{Digit} \rightarrow 0 \mid 1 \mid \ldots \mid 9 \\
\text{Literal} \rightarrow \text{Integer} \mid \text{Boolean} \mid \text{Float} \mid \text{Char} \\
\text{Integer} \rightarrow \text{Digit} \{ \text{Digit} \} \\
\text{Boolean} \rightarrow \text{true} \mid \text{False} \\
\text{Float} \rightarrow \text{Integer} \cdot \text{Integer} \\
\text{Char} \rightarrow ' \text{ASCII Char} ' 
\]
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 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:

\[ \geq \quad \text{one token} \]
\[ > \quad = \quad \text{two tokens} \]
Whitespace Examples in Pascal

while a < b do \textit{legal} - spacing between tokens
while a<b do \textit{spacing not needed for <}
while a<b do \textit{illegal} - can’t tell boundaries
while a < b do \textit{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, keywords are merely predefined identifiers (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. Why?

- *Pascal*: no.
- *Modula*: yes
- *C, C++*: yes
- *Java*: yes
- *PHP*: partly yes, partly no. What about orthogonality?
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>Clitic 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?