Programming Languages

2nd edition

Tucker and Noonan

Chapter 2
Syntax

A language that is simple to parse for the compiler is also simple to parse for the human programmer.

N. Wirth

Contents

2.3 Syntax of a Small Language: *Clite*

Motivation for using a subset of C:

Language	*Grammar (pages)*	*Reference*
Pascal	5	Jensen & Wirth
C	6	Kernighan & Richie
C++	22	Stroustrup
Java	14	Gosling, et. al.

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

$$\begin{aligned}
\mathit{Program} &\rightarrow \texttt{int main ( ) \{ } \mathit{Declarations}\ \mathit{Statements}\ \texttt{\}} \\
\mathit{Declarations} &\rightarrow \{\ \mathit{Declaration}\ \} \\
\mathit{Declaration} &\rightarrow \mathit{Type}\ \mathit{Identifier}\ [\ \texttt{[}\ \mathit{Integer}\ \texttt{]}\ ]\ \{\ ,\ \mathit{Identifier}\ [\ \texttt{[}\ \mathit{Integer}\ \texttt{]}\ ]\ \} \\
\mathit{Type} &\rightarrow \texttt{int}\ |\ \texttt{bool}\ |\ \texttt{float}\ |\ \texttt{char} \\
\mathit{Statements} &\rightarrow \{\ \mathit{Statement}\ \} \\
\mathit{Statement} &\rightarrow \texttt{;}\ |\ \mathit{Block}\ |\ \mathit{Assignment}\ |\ \mathit{IfStatement}\ |\ \mathit{WhileStatement} \\
\mathit{Block} &\rightarrow \texttt{\{}\ \mathit{Statements}\ \texttt{\}} \\
\mathit{Assignment} &\rightarrow \mathit{Identifier}\ [\ \texttt{[}\ \mathit{Expression}\ \texttt{]}\ ]\ \texttt{=}\ \mathit{Expression}\ \texttt{;} \\
\mathit{IfStatement} &\rightarrow \texttt{if (} \mathit{Expression}\ \texttt{)}\ \mathit{Statement}\ [\ \texttt{else}\ \mathit{Statement}\ ] \\
\mathit{WhileStatement} &\rightarrow \texttt{while (} \mathit{Expression}\ \texttt{)}\ \mathit{Statement}
\end{aligned}$$

Fig. 2.7 *Clite* Grammar: Expressions

$$
\begin{aligned}
\textit{Expression} &\to \textit{Conjunction}\ \{\ \texttt{||}\ \ \textit{Conjunction}\ \} \\
\textit{Conjunction} &\to \textit{Equality}\ \{\ \texttt{\&\&}\ \ \textit{Equality}\ \} \\
\textit{Equality} &\to \textit{Relation}\ [\ \textit{EquOp Relation}\ ] \\
\textit{EquOp} &\to\ \texttt{==}\ \mid\ \texttt{!=} \\
\textit{Relation} &\to \textit{Addition}\ [\ \textit{RelOp Addition}\ ] \\
\textit{RelOp} &\to\ \texttt{<}\ \mid\ \texttt{<=}\ \mid\ \texttt{>}\ \mid\ \texttt{>=} \\
\textit{Addition} &\to \textit{Term}\ \{\ \textit{AddOp Term}\ \} \\
\textit{AddOp} &\to\ \texttt{+}\ \mid\ \texttt{-} \\
\textit{Term} &\to \textit{Factor}\ \{\ \textit{MulOp Factor}\ \} \\
\textit{MulOp} &\to\ \texttt{*}\ \mid\ \texttt{/}\ \mid\ \texttt{\%} \\
\textit{Factor} &\to [\ \textit{UnaryOp}\ ]\ \textit{Primary} \\
\textit{UnaryOp} &\to\ \texttt{-}\ \mid\ \texttt{!} \\
\textit{Primary} &\to \textit{Identifier}\ [\ \texttt{[}\ \textit{Expression}\ \texttt{]}\ ]\ \mid \textit{Literal} \mid\ \texttt{(}\ \textit{Expression}\ \texttt{)}\ \mid \\
&\quad\ \textit{Type}\ \texttt{(}\ \textit{Expression}\ \texttt{)}
\end{aligned}
$$

Fig. 2.7 *Clite* grammar: lexical level

$$\begin{aligned}
\textit{Identifier} &\to \textit{Letter} \; \{ \; \textit{Letter} \mid \textit{Digit} \; \} \\
\textit{Letter} &\to \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \\
\textit{Digit} &\to \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\textit{Literal} &\to \textit{Integer} \mid \textit{Boolean} \mid \textit{Float} \mid \textit{Char} \\
\textit{Integer} &\to \textit{Digit} \; \{ \; \textit{Digit} \; \} \\
\textit{Boolean} &\to \texttt{true} \mid \texttt{False} \\
\textit{Float} &\to \textit{Integer} \;\; \texttt{.} \;\; \textit{Integer} \\
\textit{Char} &\to \texttt{'} \;\; \textit{ASCII Char} \;\; \texttt{'}
\end{aligned}$$

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:

$>=$ *one token*

$>\ \ =$ *two tokens*

Whitespace Examples in Pascal

while a < b do	*legal* - spacing between tokens
while a<b do	spacing not needed for <
whilea<bdo	*illegal* - can't tell boundaries
whilea < bdo	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. 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

Clite Operator	Associativity
Unary - !	none
* /	left
+ -	left
< <= > >=	none
== !=	none
&&	left
||	left

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?