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

2.1 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: \( \text{if ( expr ) ... discard ( )} \)
- Ada: \( \text{if ( expr ) then discard then} \)
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:

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

We can derive any unsigned integer, like 352, from this grammar.

Derivation of 352 as an \textit{Integer}

A 6-step process, starting with:

\[ \text{Integer} \]
Derivation of 352 (step 1)

Use a grammar rule to enable each step:

\[ Integer \Rightarrow Integer \ Digit \]

Derivation of 352 (steps 1-2)

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

\[ Integer \Rightarrow Integer \ Digit \Rightarrow Integer \ 2 \]
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)

Integer ⇒ Integer Digit
⇒ Integer 2
⇒ Integer Digit 2
⇒ Integer 5 2
⇒ Digit 5 2

Derivation of 352 (steps 1-6)

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

Integer ⇒ Integer Digit
⇒ Integer 2
⇒ Integer Digit 2
⇒ Integer 5 2
⇒ Digit 5 2
⇒ 3 5 2
A Different Derivation of 352

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

This is called a leftmost derivation, since at each step the leftmost nonterminal is replaced.
(The first one was a rightmost derivation.)

Notation for Derivations

\[
\begin{align*}
\text{Integer} & \Rightarrow^* 352 \\
\text{Means that 352 can be derived in a finite number of steps using the grammar for Integer.}
\end{align*}
\]

\[352 \in L(G)\]
\[\text{Means that 352 is a member of the language defined by grammar } G.\]

\[L(G) = \{ \omega \in T^* \mid \text{Integer} \Rightarrow^* \omega \}\]
\[\text{Means that the language defined by grammar } G \text{ is the set of all symbol strings } \omega \text{ that can be derived as an Integer.}\]
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
```
Arithmetic Expression Grammar

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

\[
Expr \rightarrow Expr + Term \mid Expr - Term \mid Term
\]
\[
Term \rightarrow 0 \mid ... \mid 9 \mid ( Expr )
\]
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 $G_1$:

\[
\begin{align*}
Expr & \rightarrow \text{Expr} + \text{Term} | \text{Expr} - \text{Term} | \text{Term} \\
\text{Term} & \rightarrow \text{Term} * \text{Factor} | \text{Term} / \text{Factor} | \text{Term} \% \text{Factor} | \text{Factor} \\
\text{Factor} & \rightarrow \text{Primary} ** \text{Factor} | \text{Primary} \\
\text{Primary} & \rightarrow 0 | \ldots | 9 | (\text{Expr})
\end{align*}
\]
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.

*E.g.*, Grammar $G_1$ above is unambiguous.

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 $G_2$

$\text{Expr} \rightarrow \text{Expr } \text{Op} \ \text{Expr} \ | \ ( \text{Expr} ) \ | \ \text{Integer}$

$\text{Op} \rightarrow + \ | \ - \ | \ * \ | \ / \ | \ % \ | \ **$

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

IfStatement $\rightarrow$ if ( Expression ) Statement |
if ( Expression ) Statement else Statement
Statement $\rightarrow$ Assignment | IfStatement | Block
Block $\rightarrow$ { Statements }
Statements $\rightarrow$ Statements Statement | Statement
Example

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

\[
\begin{align*}
&\text{if} \ (x < 0) \\
&\quad \text{if} \ (y < 0) \quad y = y - 1; \\
&\quad \text{else} \ y = 0;
\end{align*}
\]

Answer: \textit{either one!}
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:

   IfThenStatement -> if (Expression) Statement
   IfThenElseStatement -> if (Expression) StatementNoShortIf
                           else Statement

   The category StatementNoShortIf includes all except 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 -

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

\[ \text{IfStatement} \rightarrow \text{if} ( \text{Expression} ) \text{Statement} [ \text{else} \text{Statement} ] \]

C-style EBNF lists alternatives vertically and uses \textit{opt} to signify optional parts. E.g.,

\[ \text{IfStatement:} \quad \text{if} ( \text{Expression} ) \text{Statement} \text{ElsePart}_{opt} \]

\[ \text{ElsePart:} \quad \text{else} \text{ Statement} \]

EBNF to BNF

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

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

can be rewritten:

\[ A \rightarrow x A' z \]

\[ A' \rightarrow | \ y A' \]

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

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

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

Fig. 2.7 *Clite* Grammar: Expressions

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

\[
\begin{align*}
\text{Identifier} & \rightarrow \text{Letter} \{ \text{Letter} | \text{Digit} \} \\
\text{Letter} & \rightarrow \text{a} | \text{b} | \ldots | \text{z} | \text{A} | \text{B} | \ldots | \text{Z} \\
\text{Digit} & \rightarrow 0 | 1 | \ldots | 9 \\
\text{Literal} & \rightarrow \text{Integer} | \text{Boolean} | \text{Float} | \text{Char} \\
\text{Integer} & \rightarrow \text{Digit} \{ \text{Digit} \} \\
\text{Boolean} & \rightarrow \text{true} | \text{False} \\
\text{Float} & \rightarrow \text{Integer} . \text{Integer} \\
\text{Char} & \rightarrow \text{'ASCII Char '} \\
\end{align*}
\]

Issues Not Addressed by this Grammar

- Comments
- Whitespace
- Distinguishing one token \(<=\) from two tokens \(<\ =\)
- Distinguishing identifiers from keywords like \(\text{if}\)

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 <

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. 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>Clite 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:

\[
\text{if (a < x < b)}
\]

is not equivalent to

\[
\text{if (a < x && x < b)}
\]

But it is error-free!

So, what does it mean?

2.4 Compilers and Interpreters
Lexer

- Input: characters
- Output: tokens
- Separate:
  - Speed: 75% of time for non-optimizing
  - 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
\[ z = x + 2\times y; \]
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:

Pascal

\[
\text{while } i < n \text{ do begin }
\begin{align*}
&i := i + 1; \\
&\text{end;}
\end{align*}
\]

C/C++

\[
\text{while } (i < n) \{ \\
&i = i + 1; \\
\}
\]

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

Abstract Syntax of \textit{Clite} Assignments

\[
\text{Assignment} = \text{Variable} \, \text{target}; \, \text{Expression} \, \text{source} \\
\text{Expression} = \text{VariableRef} \mid \text{Value} \mid \text{Binary} \mid \text{Unary} \\
\text{VariableRef} = \text{Variable} \mid \text{ArrayRef} \\
\text{Variable} = \text{String} \, \text{id} \\
\text{ArrayRef} = \text{String} \, \text{id}; \, \text{Expression} \, \text{index} \\
\text{Value} = \text{IntValue} \mid \text{BoolValue} \mid \text{FloatValue} \mid \text{CharValue} \\
\text{Binary} = \text{Operator} \, \text{op}; \, \text{Expression} \, \text{term1}, \, \text{term2} \\
\text{Unary} = \text{UnaryOp} \, \text{op}; \, \text{Expression} \, \text{term} \\
\text{Operator} = \text{ArithmeticOp} \mid \text{RelationalOp} \mid \text{BooleanOp} \\
\text{IntValue} = \text{Integer} \, \text{intValue} \\
\ldots
\]
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

\[\begin{align*}
\text{Program} &= \text{Declarations} \text{ decpart}; \text{ Statements} \text{ body;} \\
\text{Declarations} &= \text{Declaration}^* \\
\text{Declaration} &= \text{VariableDecl} | \text{ ArrayDecl} \\
\text{VariableDecl} &= \text{Variable} \ v; \ \text{Type} \ t \\
\text{ArrayDecl} &= \text{Variable} \ v; \ \text{Type} \ t; \ \text{Integer} \ \text{size} \\
\text{Type} &= \text{int} | \text{bool} | \text{float} | \text{char} \\
\text{Statements} &= \text{Statement}^* \\
\text{Statement} &= \text{Skip} | \text{Block} | \text{ Assignment} | \text{ Conditional} | \text{ Loop} \\
\text{Skip} &= \\
\text{Block} &= \text{Statements}' \\
\text{Conditional} &= \text{Expression} \ \text{test}; \ \text{Statement} \ \text{thenbranch}, \ \text{elsebranch} \\
\text{Loop} &= \text{Expression} \ \text{test}; \ \text{Statement} \ \text{body}
\end{align*}\]