SPECIFYING SYNTAX

Terminology

The **syntax** of a programming language determines the well-formed or grammatically correct programs of the language.

Its **semantics** describes how or whether such programs will execute.

- syntax is how things look
- semantics is how things work (the meaning)

Example: “a = b+c” is correct Java syntax apparently.

Questions:

1) Have b and c been declared as a type that allows the + operation?

   Note that Java has five different + operations.

2) Is “a” assignment compatible with the result of the expression “b+c”?

3) Do “b” and “c” have values?

Compile-time or **static** issues: 1) and 2)

Run-time or **dynamic** issues: 3)

Copyright © 2001 by Ken Slonneger

Syntax

A **terminal** is a symbol in the language.

Java terminals: **while**, **static**, (, ;, 5, b

A **syntactic category** or **nonterminal** is a set of objects (strings) that will be defined in terms of symbols in the language.

Java nonterminals: <statement>, <expression>

A **metalanguage** is a higher-level language used to discuss, describe, or analyze another language. English is used as a metalanguage for describing programming languages, but because of the ambiguities in English, more formal metalanguages have been proposed.

Backus-Naur Form (BNF)

BNF was first used by John Backus and Peter Naur to describe Algol in 1963.

It consists of a set of **rules** or **productions** of the form:

<syntactic category> ::= a sequence of terminals,
syntactic categories, and the
symbol "|"

"::=" means "is defined as" (sometimes written as "→")

juxtaposition means concatenation

"|" means alternation (or)

 Copyright © 2001 by Ken Slonneger

BNF Examples from Java

<integral type> ::=
byte | **short** | **int** | **long** | **char**

<argument list> ::= <expression>
| <argument list> **,** <expression>

<statement> ::= <statement without trailing substatement>
| <labeled statement> | <if then statement>
| <if then else statement> | <while statement >
| <for statement>

<method declaration> ::=
<method modifiers> <result type>
<method declarator> <throws> <method body>
| <method modifiers> <result type>
<method declarator> <method body>
| <result type> <method declarator>
<throws> <method body>
| <result type> <method declarator> <method body>

Note that “|” corresponds to set union.

<type> ::= <primitive type> | <reference type>

The set of types is the union of the sets of primitive types and reference types.

Copyright © 2001 by Ken Slonneger

CBL (Cobol-like) Grammars

These were originally used in the description of Cobol; they are not as formal as BNF.

1) Optional elements are enclosed in brackets: [ ].
2) Alternate elements are vertically enclosed in braces: { }.
3) Optional alternates are vertically enclosed in brackets.
4) Repeated elements are indicated by listing one element followed by an ellipsis: ….
5) Required key words are underlined; optional noise words are not.
6) Items supplied by the user are written as lower case or as syntactic categories from which an item my be taken.

Examples

```
                         [ <letter> ]
<identifier> ::= <letter> [ <digit>  ] …
```

```
<Cobol ADD statement> ::=

       { <identifier> }  [ ,<identifier> ]
   ADD { <number>     }  [ ,<number>     ] …  TO

   <identifier> [ ROUNDED ] [ ,<identifier> [ ROUNDED ] ] …

   [ ON SIZE ERROR <statement> ]
```

ADD A, 5 TO B ROUNDED, D

ON SIZE ERROR PERFORM E-ROUTINE

 Copyright © 2001 by Ken Slonneger

Extended BNF (EBNF)

Since a BNF description of the syntax of a programming language relies heavily on recursion to provide lists of items, many definitions use these extensions:

1) item$^{?}$ or [item] means item is optional.
2) item* or { item } means to take zero or more occurrences of item.
3) item^{+} means to take one or more occurrences of item.
4) Parentheses are used for grouping.

Examples

<method declaration> ::=
    <method modifiers>$^{?}$ <result type>
        <method declarator> <throws>$^{?}$ <method body>

<method modifier> ::= **public** | **protected** | **private** | **static**
| **abstract** | **final** | **synchronized** | **native**

<method modifiers> ::= <method modifier>
| <method modifiers> <method modifier>

may be written

<method modifiers> ::= <method modifier>$^{+}$

Or better, eliminate nonterminal <method modifiers> and write:

<method declaration> ::=
    <method modifier>* <result type>
        <method declarator> <throws>$^{?}$ <method body>

<argument list> ::= <expression> (**,** <expression>)*

Copyright © 2001 by Ken Slonneger

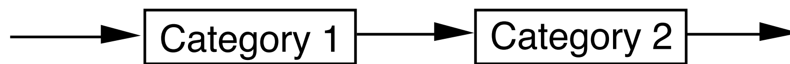
Syntax Diagrams (or Charts or Graphs)

Syntax diagrams are associated with Pascal, but they have been used to describe command languages and, recently, many other programming languages.

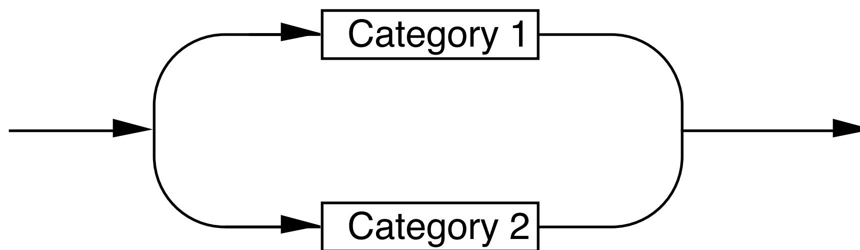
1) A terminal symbol is shown in a circle or oval.

2) A syntactic category is placed in a rectangle.

3) The concatenation of two objects is indicated by a flowline:

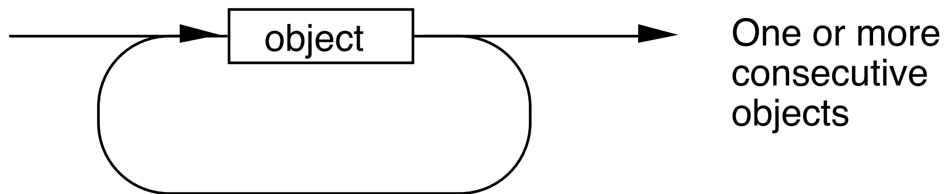


4) The alternation of two objects is shown by branching:

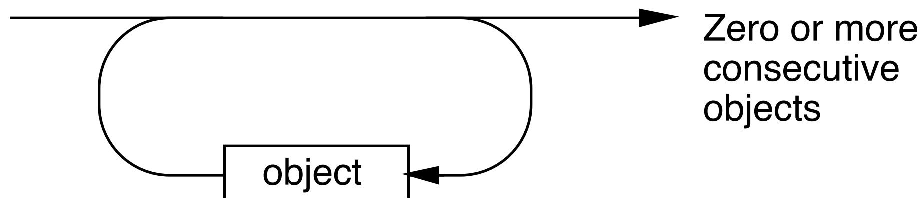


 Copyright © 2001 by Ken Slonneger

5) Repetition of objects is represented by a loop:



<thing> ::= <object> | <thing> <object>



<thing> ::= <empty> | <thing> <object>

<empty> ::=

Example: Simple Expressions in Pascal

BNF

<simple expr> ::= <term> | <sign> <term> |
<simple expr> <adding operator> <term>

<sign> ::= + | –

<adding operator> ::= + | – | **or**

CBL

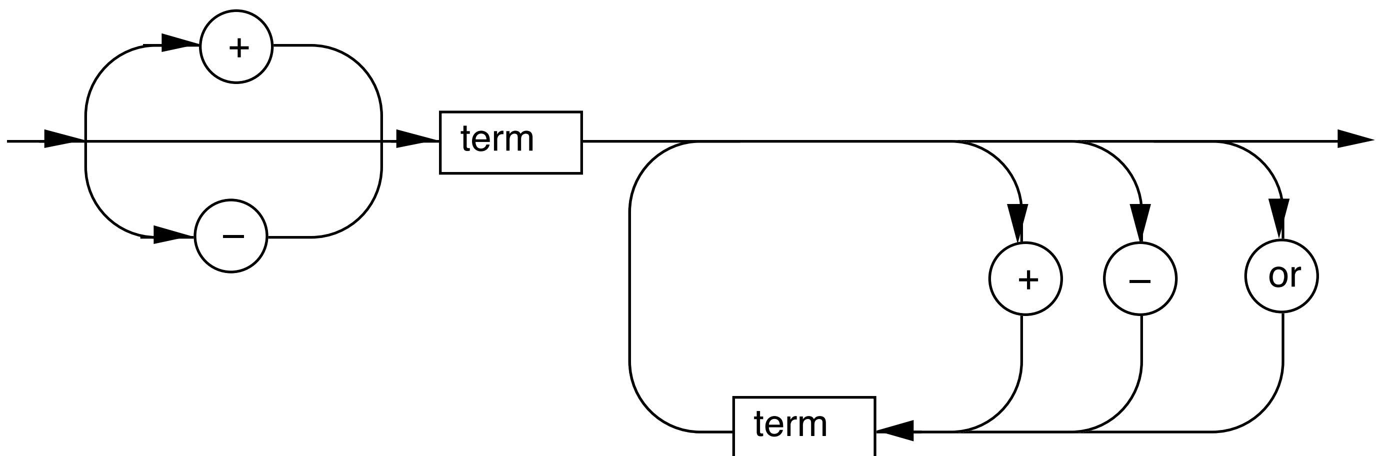
$$\left[\begin{matrix}+\\-\end{matrix}\right]\langle\text{term}\rangle\left[\left\{\begin{matrix}+\\-\\\textbf{or}\end{matrix}\right\}\langle\text{term}\rangle\right]\ \ldots$$

EBNF

<simple expr> ::=

[<sign>] <term> { <adding operator> <term> }

Syntax Diagram



Exercise: According the the syntactic specification, which of these terminal strings are simple expressions, assuming that a, b, and c are legal terms:

1) a + b – c

2) – a **or** b + c

3) b – – c

 Copyright © 2001 by Ken Slonneger

Derivation or Parse Trees

The syntactic structure of a string of terminals as generated by a given grammar can be depicted as a **derivation tree**.

- Leaf nodes are labeled with terminal symbols.
- Each interior node is labeled by a nonterminal whose children represent the right side of a production for that nonterminal.
- The root is labeled with the syntactic category being derived, called the **start** symbol.

Example: A Restricted English Grammar

<sentence> ::= <subject> <predicate> **.**

<subject> ::= <simpsub> | <simpsub> **and** <subject>

<simpsub> ::= <qualifiers> <noun> | <noun>

<qualifiers> ::= <adjective> | <adjective> **,** <qualifiers>

<adjective> ::= **big** | **little**

<noun> ::= **dinners** | **books** | **Snoopy** | **Linus**

<predicate> ::=

<verb> | <verb> <dirobj> | <verb> <dirobj> <adverb>

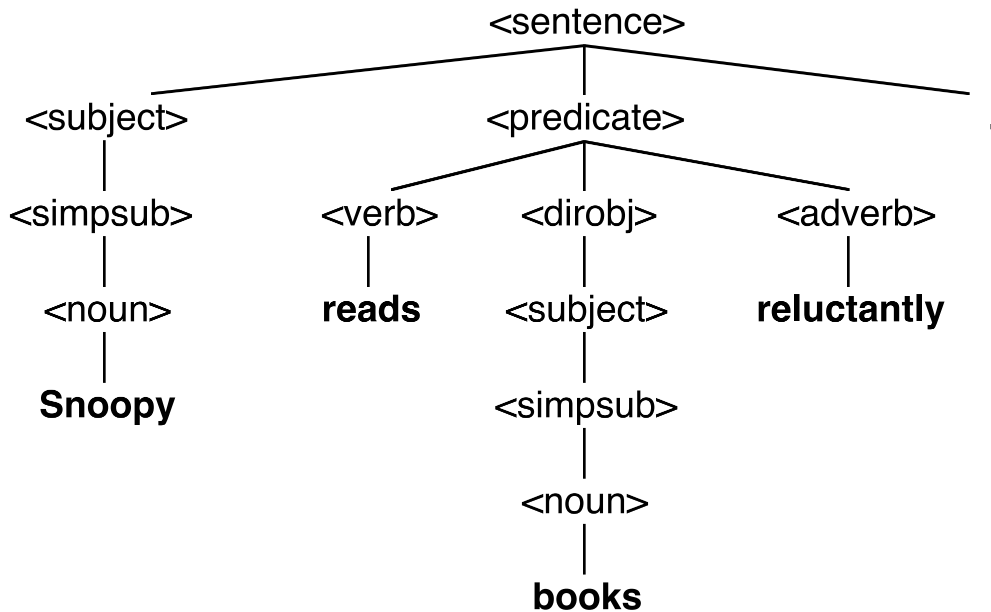
<dirobj> ::= <subject>

<verb> ::= **eats** | **reads**

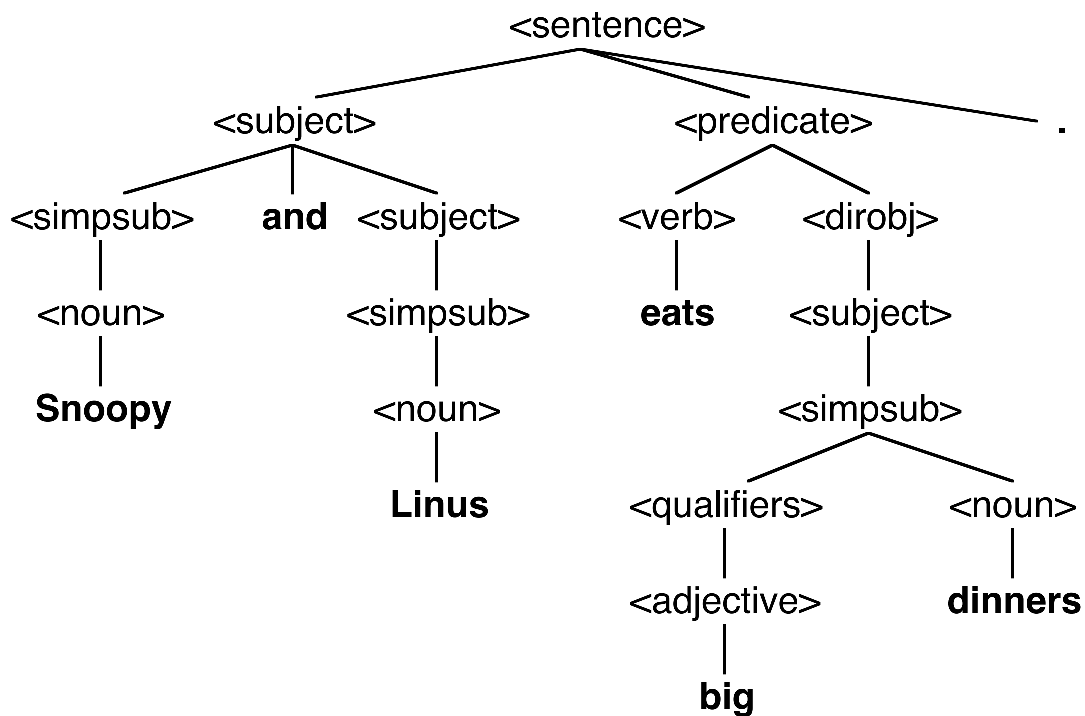
<adverb> ::= **with feeling** | **reluctantly**

Copyright © 2001 by Ken Slonneger

Snoopy reads books reluctantly.



Snoopy and Linus eats big dinners.



 Copyright © 2001 by Ken Slonneger

Exercises:

1. Draw a parse tree for the sentence:

   little Snoopy eats Linus and books with feeling.

2. How many legal sentences are in the language?

3. Write descriptions of the restricted English grammar using:

   a) CBL b) EBNF c) Syntax Diagrams

Expressions

Expressions are a vital component in programming languages.

Here is a simple language of expressions:

<expr> ::= <name> | <expr> <operator> <expr>

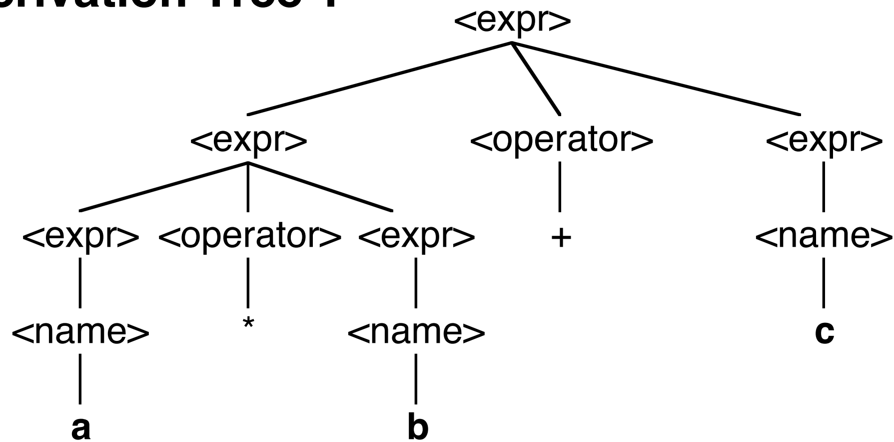
<name> ::= **a** | **b** | **c**

<operator> ::= + | *

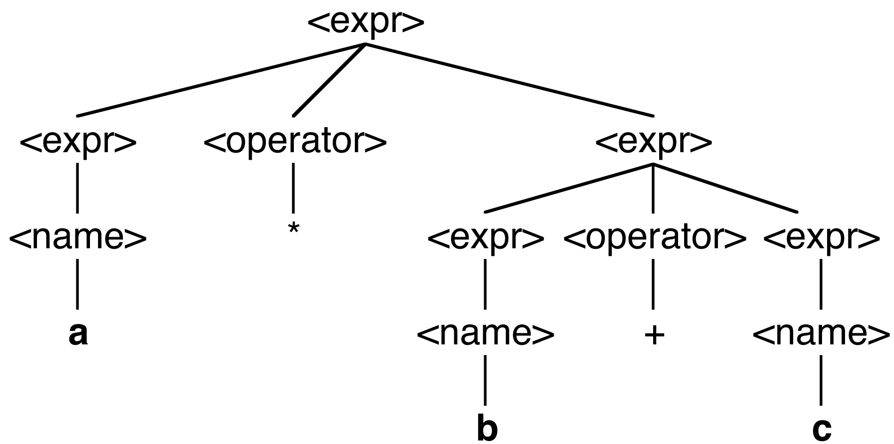
Consider the terminal string “**a** * **b** + **c**”.

It allows two derivations in this language of expressions:

Derivation Tree 1



Derivation Tree 2



A grammar is **ambiguous** if some terminal string has two or more distinct structural descriptions (parse trees).

Copyright © 2001 by Ken Slonneger

Revision 1: <expr> ::= <name> | <expr> <operator> <name>

- called left recursion
- gives left-to-right evaluation of operators
- yields the first derivation tree

Revision 2: <expr> ::= <name> | <name> <operator> <expr>

- called right recursion
- gives right-to-left evaluation of operators
- yields the second derivation tree
- this is the order of execution of all binary operators in APL

Note: Some (very strange) languages are inherently ambiguous—every grammar for the language is ambiguous.

Exercise: Look at <expression> in Java

a) How are precedence levels provided for operators?

b) How is the order of execution within a level determined?

c) How are parentheses handled?

Another Syntactic Ambiguity: The dangling "**else**"

What is the meaning of this Java or C statement:

if (E_1) **if** (E_2) S_1; **else** S_2;

This ambiguity can be solved in several ways:

Java: A semantic solution specified using English—the statement is considered equivalent to

if (E_1) { **if** (E_2) S_1; **else** S_2; }

Algol: A syntactic solution

<conditional statement> ::=

if <Bool expr> **then** <unconditional statement> |

if <Bool expr> **then** <unconditional statement>

else <statement>

Note: A compound statement is unconditional.

Ada: A syntactic solution

if <condition> **then**

<sequence of statements>

[**else** <sequence of statements>]

end if;

Each **if** needs an **end if**.

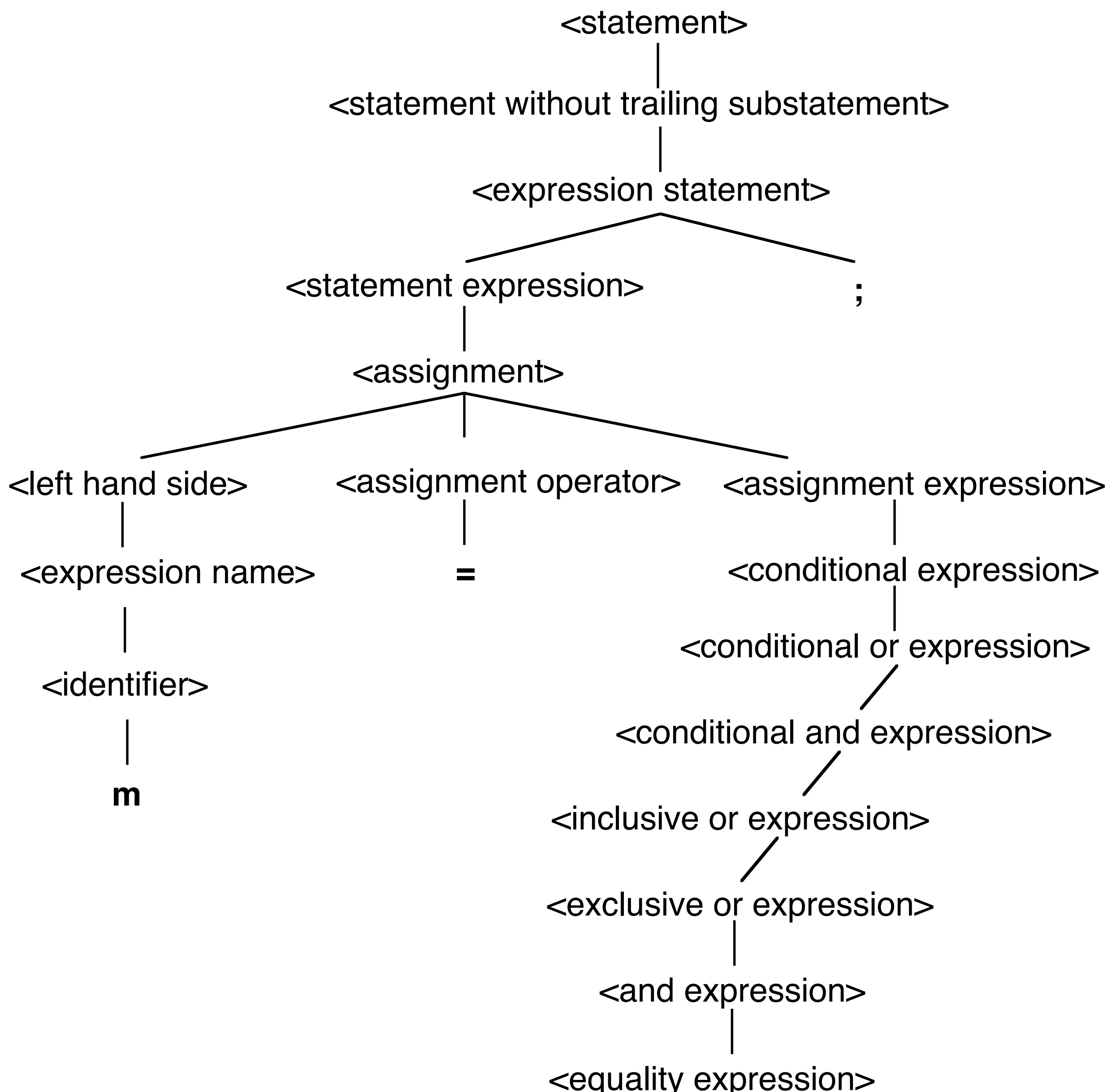
 Copyright © 2001 by Ken Slonneger

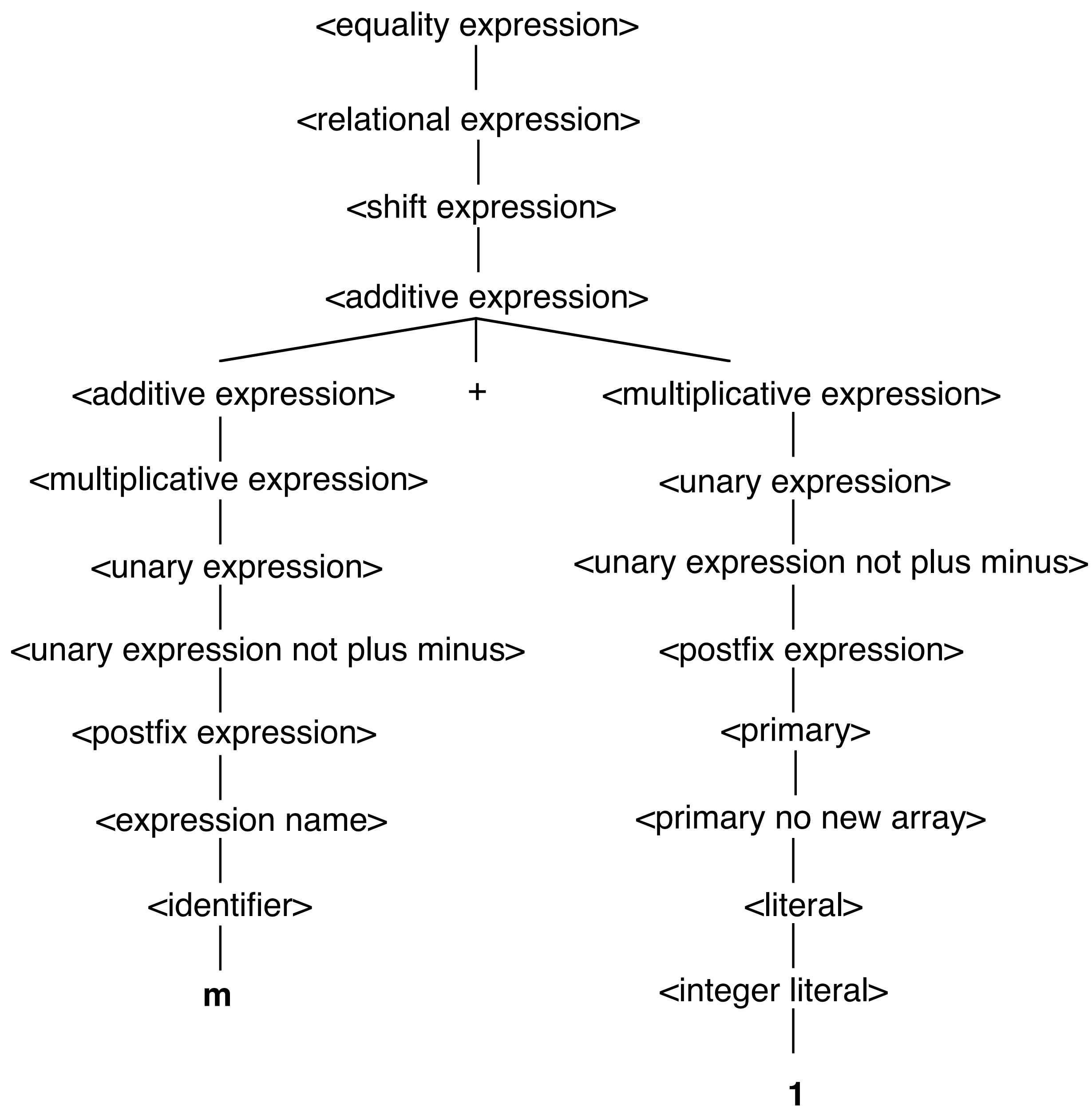
Parse Tree in Java

Each correctly formed construct in Java has a derivation tree defined by its BNF syntax.

This example uses the Concrete Syntax of Java at the end of this part.

Example: m = m + 1; is a correct <statement>





 Copyright © 2001 by Ken Slonneger

Limitations of Syntactic Definitions

- The Concrete Syntax for Java is an incomplete description of Java—not all the terminal strings generated are regarded as valid Java programs, although all legal programs can be derived from the BNF grammar.
- In fact, there is no BNF (or EBNF or Syntax Diagram) grammar that generates only the legal Java programs.
- A BNF grammar defines a **context-free** language: the left-hand side of each BNF rule contains only one syntactic category—it is replaced by one of its alternative definitions regardless of the context in which it occurs.
- Example of a **context-sensitive** grammar:

<sentence> ::= **abc** | **a**<thing>**bc**

<thing>**b** ::= **b**<thing>

<thing>**c** ::= <other>**bcc**

a<other> ::= **aa** | **aa**<thing>

b<other> ::= <other>**b**

The language generated = { **abc**, **aabbcc**, **aaabbbccc**, … }

Sample Derivation:

<sentence> => **a**<thing>**bc** => **ab**<thing>**c**

=> **ab**<other>**bcc** => **a**<other>**bbcc** => **aabbcc**

Exercise: Derive the <sentence> **aaabbbccc**

- The set of Java programs is not a context-free language; it has context-sensitive features.

- The context-sensitive features may be informally described as a set of restrictions or context conditions.

- They deal primarily with the declarations of identifiers, the types of variables and expressions, and the compatibility of objects and operations.

- Some examples:

  1) Operands for the **&&** operation must be of boolean type.

  2) In a method definition of a function, the return expression must be compatible with the return type of the method.

  3) When a method is called, the actual parameters must match the formal parameters in number and type.

Exercise: List four more context conditions in Java or C++.

 Copyright © 2001 by Ken Slonneger

Basic Parts of a Programming Language

1. Expression

   Construct that produces (defines) a value.

2. Command (statement)

   Construct that changes the state of the computer.

   - change in memory
   - produce output of some form
   - change control state (position of execution in the program)

3. Declaration

   Construct that binds an identifier (a name) to a programming language artifact.

Java Bindings

Identifier	**Binding**
variable	memory location
method	method definition (code segment)
class, interface	Class object
package	set of classes and interfaces and directory path
constant	value

Expressions May Act As Commands

Some languages allow expressions to serve as commands.

Java

75. <statement> ::= <statement without trailing substatement>

77. <statement without trailing substatement> ::= <expression statement>

81. <expression statement> ::= <statement expression> **;**

82. <statement expression> ::= <assignment> | <preincrement expression>
    | <postincrement expression> | <predecrement expression>
    | <postdecrement expression> | <method invocation>
    | <class instance creation expression>

113. <assignment> ::= <left hand side> <assignment operator>
    <assignment expression>

129. <predecrement expression> ::= **--** <unary expression>

130. <preincrement expression> ::= **++** <unary expression>

132. <postdecrement expression> ::= <postfix expression> **--**

133. <postincrement expression> ::= <postfix expression> **++**

135. <method invocation> ::= <method name> **(** <argument list>? **)**
    | <primary> **.** <identifier> **(** <argument list>? **)**
    | **super .** <identifier> **(** <argument list>? **)**

139. <class instance creation expression> ::= **new** <class type> **(** <argument list>? **)**

Copyright © 2001 by Ken Slonneger

Assignment

The BNF for Java shows that an assignment in Java is really an expression.

Its value is the value assigned to the variable to the left of the equal sign.

Having assignment be an expression allows code such as the following:

```
m = n = 0;

if ((count = count - 1) > 0) …

add(b = new JButton("Press"));
```

Other Statement Expressions

The other statement expressions can be viewed as normal expressions.

When used as a command, the value produced by such an expression is simply discarded.

Examples

```
num++;

--count;

Math.sqrt(100.0);        // a useless command

new JFrame();
```

C (and C++)

In C *all* expressions can act as commands.

105. <statement> ::= <expression statement>

106. <expression statement> ::= <expression> ;

This design decision leads to some very strange programs.

```
main ()
{
    int n;
    n = 5;
    n++;
    n+1;
    123;
    printf("n = %d\n",n);
}
```

```
/**** Output:

n = 6

********************/
```

 Copyright © 2001 by Ken Slonneger

Concrete Syntax for Java

Programs

1. <goal> ::= <compilation unit>

2. <compilation unit> ::= <package declaration>$^?$ <import declarations>$^?$
   <type declarations>$^?$

Declarations

3. <package declaration> ::= **package** <package name> **;**

4. <import declarations> ::= <import declaration>
   | <import declarations> <import declaration>

5. <import declaration> ::= <single type import declaration>
   | <type import on demand declaration>

6. <single type import declaration> ::= **import** <type name> **;**

7. <type import on demand declaration> ::= **import** <package name> **. * ;**

8. <type declarations> ::= <type declaration>
   | <type declarations> <type declaration>

9. <type declaration> ::= <class declaration> | <interface declaration> | **;**

10. <class declaration> ::= <class modifiers>$^?$ **class** <identifier> <super>$^?$
    <interfaces>$^?$ <class body>

11. <class modifiers> ::= <class modifier> | <class modifiers> <class modifier>

12. <class modifier> ::= **public** | **abstract** | **final**

13. <super> ::= **extends** <class type>

14. <interfaces> ::= **implements** <interface type list>

15. <interface type list> ::= <interface type> | <interface type list> **,** <interface type>

16. <class body> ::= **{** <class body declarations>$^?$ **}**

17. <class body declarations> ::= <class body declaration>
    | <class body declarations> <class body declaration>

18. <class body declaration> ::= <class member declaration>
    | <static initializer> | <constructor declaration>

19. <class member declaration> ::= <field declaration> | <method declaration>

Copyright © 2001 by Ken Slonneger

20. <static initializer> ::= **static** <block>

21. <constructor declaration> ::= <constructor modifiers>$^{?}$ <constructor declarator>
    <throws>$^{?}$ <constructor body>

22. <constructor modifiers> ::= <constructor modifier>
    | <constructor modifiers> <constructor modifier>

23. <constructor modifier> ::= **public** | **protected** | **private**

24. <constructor declarator> ::= <simple type name> **(** <formal parameter list>$^{?}$ **)**

25. <formal parameter list> ::= <formal parameter>
    | <formal parameter list> **,** <formal parameter>

26. <formal parameter> ::= <type> <variable declarator id>

27. <throws> ::= **throws** <class type list>

28. <class type list> ::= <class type> | <class type list> **,** <class type>

29. <constructor body> ::= **{** <explicit constructor invocation>$^{?}$ <block statements>$^{?}$ **}**

30. <explicit constructor invocation> ::= **this (** <argument list>$^{?}$ **)**
    | **super (** <argument list>$^{?}$ **)**

31. <field declaration> ::= <field modifiers>$^{?}$ <type> <variable declarators> **;**

32. <field modifiers> ::= <field modifier>
    | <field modifiers> <field modifier>

33. <field modifier> ::= **public** | **protected** | **private** | **static** | **final** | **transient** | **volatile**

34. <variable declarators> ::= <variable declarator>
    | <variable declarators> **,** <variable declarator>

35. <variable declarator> ::= <variable declarator id>
    | <variable declarator id> **=** <variable initializer>

36. <variable declarator id> ::= <identifier> | <variable declarator id> **[ ]**

37. <variable initializer> ::= <expression> | <array initializer>

38. <method declaration> ::= <method header> <method body>

39. <method header> ::=
    <method modifiers>$^{?}$ <result type> <method declarator> <throws>$^{?}$

40. <result type> ::= <type> | **void**

41. <method modifiers> ::= <method modifier>
    | <method modifiers> <method modifier>

 Copyright © 2001 by Ken Slonneger

42. <method modifier> ::= **public** | **protected** | **private** | **static** | **abstract** | **final**
    | **synchronized** | **native**

43. <method declarator> ::= <identifier> **(** <formal parameter list>$^{?}$ **)**

44. <method body> ::= <block> | **;**

45. <interface declaration> ::= <interface modifiers>$^{?}$ **interface** <identifier>
    <extends interfaces>$^{?}$ <interface body>

46. <interface modifiers> ::= <interface modifier>
    | <interface modifiers> <interface modifier>

47. <interface modifier> ::= **public** | **abstract**

48. <extends interfaces> ::= **extends** <interface type>
    | <extends interfaces> **,** <interface type>

49. <interface body> ::= **{** <interface member declarations>$^{?}$ **}**

50. <interface member declarations> ::= <interface member declaration>
    | <interface member declarations> <interface member declaration>

51. <interface member declaration> ::= <constant declaration>
    | <abstract method declaration>

52. <constant declaration> ::= <constant modifiers> <type> <variable declarator>

53. <constant modifiers> ::= **public** | **static** | **final**

54. <abstract method declaration> ::=
    <abstract method modifiers>$^{?}$ <result type> <method declarator> <throws>$^{?}$ **;**

55. <abstract method modifiers> ::= <abstract method modifier>
    | <abstract method modifiers> <abstract method modifier>

56. <abstract method modifier> ::= **public** | **abstract**

57. <array initializer> ::= **{** <variable initializers>$^{?}$ **,**$^{?}$ **}**

58. <variable initializers> ::= <variable initializer>
    | <variable initializers> **,** <variable initializer>

59. <variable initializer> ::= <expression> | <array initializer>

Types

60. <type> ::= <primitive type> | <reference type>

61. <primitive type> ::= <numeric type> | **boolean**

62. <numeric type> ::= <integral type> | <floating-point type>

63. <integral type> ::= **byte** | **short** | **int** | **long** | **char**

64. <floating-point type> ::= **float** | **double**

65. <reference type> ::= <class or interface type> | <array type>

66. <class or interface type> ::= <class type> | <interface type>

67. <class type> ::= <type name>

68. <interface type> ::= <type name>

69. <array type> ::= <type> **[ ]**

Blocks and Commands

70. <block> ::= **{** <block statements>? **}**

71. <block statements> ::= <block statement> | <block statements> <block statement>

72. <block statement> ::= <local variable declaration statement> | <statement>

73. <local variable declaration statement> ::= <local variable declaration> **;**

74. <local variable declaration> ::= <type> <variable declarators>

75. <statement> ::= <statement without trailing substatement>
| <labeled statement> | <if then statement> | <if then else statement>
| <while statement> | <for statement>

76. <statement no short if> ::= <statement without trailing substatement>
| <labeled statement no short if> | <if then else statement no short if>
| <while statement no short if> | <for statement no short if>

77. <statement without trailing substatement> ::= <block> | <empty statement>
| <expression statement> | <switch statement> | <do statement>
| <break statement> | <continue statement> | <return statement>
| <synchronized statement> | <throws statements> | <try statement>

78. <empty statement> ::= **;**

79. <labeled statement> ::= <identifier> **:** <statement>

80. <labeled statement no short if> ::= <identifier> **:** <statement no short if>

81. <expression statement> ::= <statement expression> **;**

82. <statement expression> ::= <assignment> | <preincrement expression>
| <postincrement expression> | <predecrement expression>

 Copyright © 2001 by Ken Slonneger

| <postdecrement expression> | <method invocation>
| <class instance creation expression>

83. <if then statement> ::= **if (** <expression> **)** <statement>

84. <if then else statement> ::= **if (** <expression> **)** <statement no short if>
else <statement>

85. <if then else statement no short if> ::=
if (<expression> **)** <statement no short if> **else** <statement no short if>

86. <switch statement> ::= **switch (** <expression> **)** <switch block>

87. <switch block> ::= **{** <switch block statement groups>$^?$ <switch labels>$^?$ **}**

88. <switch block statement groups> ::= <switch block statement group>
| <switch block statement groups> <switch block statement group>

89. <switch block statement group> ::= <switch labels> <block statements>

90. <switch labels> ::= <switch label> | <switch labels> <switch label>

91. <switch label> ::= **case** <constant expression> **:** | **default :**

92. <while statement> ::= **while (** <expression> **)** <statement>

93. <while statement no short if> ::= **while (** <expression> **)** <statement no short if>

94. <do statement> ::= **do** <statement> **while (** <expression> **) ;**

95. <for statement> ::= **for (** <for init>$^?$ **;** <expression>$^?$ **;** <for update>$^?$ **)** <statement>

96. <for statement no short if> ::= **for (** <for init>$^?$ **;** <expression>$^?$ **;** <for update>$^?$ **)**
<statement no short if>

97. <for init> ::= <statement expression list> | <local variable declaration>

98. <for update> ::= <statement expression list>

99. <statement expression list> ::= <statement expression>
| <statement expression list> **,** <statement expression>

100. <break statement> ::= **break** <identifier>$^?$ **;**

102. <continue statement> ::= **continue** <identifier>$^?$ **;**

103. <return statement> ::= **return** <expression>$^?$ **;**

104. <throws statement> ::= **throw** <expression> **;**

105. <synchronized statement> ::= **synchronized (** <expression> **)** <block>

106. <try statement> ::= **try** <block> <catches> | **try** <block> <catches>? <finally>

107. <catches> ::= <catch clause> | <catches> <catch clause>

108. <catch clause> ::= **catch (** <formal parameter> **)** <block>

109. <finally > ::= **finally** <block>

Expressions

110. <constant expression> ::= <expression>

111. <expression> ::= <assignment expression>

112. <assignment expression> ::= <conditional expression> | <assignment>

113. <assignment> ::= <left hand side> <assignment operator> <assignment expression>

114. <left hand side> ::= <expression name> | <field access> | <array access>

115. <assignment operator> ::= **=** | ***=** | **/=** | **%=** | **+=** | **-=** | **<<=** | **>>=**
| **>>>=** | **&=** | **^=** | **|=**

116. <conditional expression> ::= <conditional or expression>
| <conditional or expression> **?** <expression> **:** <conditional expression>

117. <conditional or expression> ::= <conditional and expression>
| <conditional or expression> || <conditional and expression>

118. <conditional and expression> ::= <inclusive or expression>
| <conditional and expression> **&&** <inclusive or expression>

119. <inclusive or expression> ::= <exclusive or expression>
| <inclusive or expression> **|** <exclusive or expression>

120. <exclusive or expression> ::= <and expression>
| <exclusive or expression> **^** <and expression>

121. <and expression> ::= <equality expression>
| <and expression> **&** <equality expression>

122. <equality expression> ::= <relational expression>
| <equality expression> **==** <relational expression>
| <equality expression> **!=** <relational expression>

123. <relational expression> ::= <shift expression>
| <relational expression> **<** <shift expression>
| <relational expression> **>** <shift expression>
| <relational expression> **<=** <shift expression>
| <relational expression> **>=** <shift expression>
| <relational expression> **instanceof** <reference type>

 Copyright © 2001 by Ken Slonneger

124. <shift expression> ::= <additive expression>
| <shift expression> **<<** <additive expression>
| <shift expression> **>>** <additive expression>
| <shift expression> **>>>** <additive expression>

125. <additive expression> ::= <multiplicative expression>
| <additive expression> **+** <multiplicative expression>
| <additive expression> **-** <multiplicative expression>

126. <multiplicative expression> ::= <unary expression>
| <multiplicative expression> * <unary expression>
| <multiplicative expression> **/** <unary expression>
| <multiplicative expression> % <unary expression>

127. <cast expression> ::= **(** <primitive type> **)** <unary expression>
| **(** <reference type> **)** <unary expression not plus minus>

128. <unary expression> ::= <preincrement expression>
| <predecrement expression>
| **+** <unary expression> | **-** <unary expression>
| <unary expression not plus minus>

129. <predecrement expression> ::= **--** <unary expression>

130. <preincrement expression> ::= **++** <unary expression>

131. <unary expression not plus minus> ::=
<postfix expression> | ~ <unary expression>
| **!** <unary expression> | <cast expression>

132. <postdecrement expression> ::= <postfix expression> **--**

133. <postincrement expression> ::= <postfix expression> **++**

134. <postfix expression> ::= <primary> | <expression name>
| <postincrement expression> | <postdecrement expression>

135. <method invocation> ::= <method name> **(** <argument list>$^?$ **)**
| <primary> **.** <identifier> **(** <argument list>$^?$ **)**
| **super .** <identifier> **(** <argument list>$^?$ **)**

136. <field access> ::= <primary> **.** <identifier> | **super .** <identifier>

137. <primary> ::= <primary no new array> | <array creation expression>

138. <primary no new array> ::= <literal> | **this** | **(** <expression> **)**
| <class instance creation expression> | <field access>
| <method invocation> | <array access>

139. <class instance creation expression> ::= **new** <class type> **(** <argument list>$^?$ **)**

140. <argument list> ::= <expression> | <argument list> **,** <expression>

141. <array creation expression> ::= **new** <primitive type> <dim exprs> <dims>$^{?}$
  | **new** <class or interface type> <dim exprs> <dims>$^{?}$

142. <dim exprs> ::= <dim expr> | <dim exprs> <dim expr>

143. <dim expr> ::= **[** <expression> **]**

144. <dims> ::= **[ ]** | <dims> **[ ]**

145. <array access> ::= <expression name> **[** <expression> **]**
  | <primary no new array> **[** <expression> **]**

Tokens

146. <package name> ::= <identifier> | <package name> **.** <identifier>

147. <type name> ::= <identifier> | <package name> **.** <identifier>

148. <simple type name> ::= <identifier>

149. <expression name> ::= <identifier> | <ambiguous name> **.** <identifier>

150. <method name> ::= <identifier> | <ambiguous name> **.** <identifier>

151. <ambiguous name> ::= <identifier> | <ambiguous name> **.** <identifier>

152. <literal> ::= <integer literal> | <floating-point literal> | <boolean literal>
  | <character literal> | <string literal> | <null literal>

 Copyright © 2001 by Ken Slonneger