프로그래밍언어의 Syntax와 Parsing

Contents

프로그래밍언어의 Syntax

- Syntax는 PL 문법의 정확한 표현을 의미한다. 어떤 방법으로?

- John Backus와 Peter Naur가 Algol 60의 정의에 언어학 (Linguistics)과 수리논리학에서 연구된 형식언어론 (Formal Languages)와 오토마타 (Automata) 이론을 PL 언어 표현에 이용함. BNF (Backus Naur Form)라고 불림. PL 문법 표현의 표준적 방법으로 이용되고 있음.

- 관련과목: 컴파일러, 형식언어론

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.

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 *(Context-Free Grammar)*

$$A \rightarrow \omega$$

where $A \in N$ and $\omega \in (N \cup T)^*$

Example: Binary Digits

Consider the grammar:

$$\textit{binaryDigit} \rightarrow 0$$
$$\textit{binaryDigit} \rightarrow 1$$

or equivalently:

$$\textit{binaryDigit} \rightarrow 0 \mid 1$$

Here, | is a metacharacter that separates *alternatives*.

Derivations

Consider the grammar:

Integer → *Digit* | *Integer Digit*
Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

We can *derive* any unsigned integer, like `352`, from this grammar. 여기서 Integer는 Start Nonterminal 임.

Derivation of `352` as an *Integer*

A 6-step process, starting with:

Integer

Derivation of 352 (step 1)

Use a grammar rule to enable each step:

$$\textit{Integer} \Rightarrow \textit{Integer Digit}$$

Derivation of 352 (steps 1–2)

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

$$\begin{aligned} \textit{Integer} &\Rightarrow \textit{Integer Digit} \\ &\Rightarrow \textit{Integer}\ 2 \end{aligned}$$

Derivation of 352 (steps 1–3)

Each step follows from the one before it.

$$
\begin{aligned}
&\textit{Integer} \\
&\Rightarrow \textit{Integer Digit} \\
&\Rightarrow \textit{Integer}\ 2 \\
&\Rightarrow \textit{Integer Digit}\ 2
\end{aligned}
$$

Derivation of 352 (steps 1–4)

$$
\begin{aligned}
&\textit{Integer} \Rightarrow \textit{Integer Digit} \\
&\quad \Rightarrow \textit{Integer}\ 2 \\
&\quad \Rightarrow \textit{Integer Digit}\ 2 \\
&\quad \Rightarrow \textit{Integer}\ 5\ 2
\end{aligned}
$$

Derivation of 352 (steps 1–5)

$$
\begin{aligned}
&\textit{Integer} \Rightarrow \textit{Integer Digit} \\
&\quad \Rightarrow \textit{Integer}\ 2 \\
&\quad \Rightarrow \textit{Integer Digit}\ 2 \\
&\quad \Rightarrow \textit{Integer}\ 5\ 2 \\
&\quad \Rightarrow \textit{Digit}\ 5\ 2
\end{aligned}
$$

Derivation of 352 (steps 1–6)

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

$$
\begin{aligned}
&\textit{Integer} \Rightarrow \textit{Integer Digit} \\
&\quad \Rightarrow \textit{Integer}\ 2 \\
&\quad \Rightarrow \textit{Integer Digit}\ 2 \\
&\quad \Rightarrow \textit{Integer}\ 5\ 2 \\
&\quad \Rightarrow \textit{Digit}\ 5\ 2 \\
&\quad \Rightarrow 3\ 5\ 2
\end{aligned}
$$

A Different Derivation of 352

$$
\begin{aligned}
\textit{Integer} &\Rightarrow \textit{Integer Digit} \\
&\Rightarrow \textit{Integer Digit Digit} \\
&\Rightarrow \textit{Digit Digit Digit} \\
&\Rightarrow 3\ \textit{Digit Digit} \\
&\Rightarrow 3\ 5\ \textit{Digit} \\
&\Rightarrow 3\ 5\ 2
\end{aligned}
$$

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

$Integer \Rightarrow^* 352$

Means that 352 can be derived in a finite number of steps using the grammar for *Integer*.

$352 \in L(G)$

Means that 352 is a member of the language defined by grammar *G*.

$L(G) = \{ \omega \in T^* \mid Integer \Rightarrow^* \omega \}$

Means that the language defined by grammar *G* is the set of all symbol strings ω that can be derived as an *Integer*.

문법 G에 의해서 생성될 수 있는 스트링들의 집합.

Parse Trees

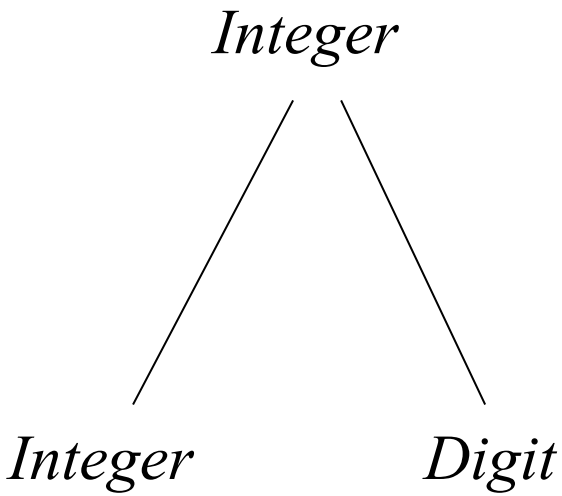
A *parse tree* is a graphical representation of a derivation.

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.

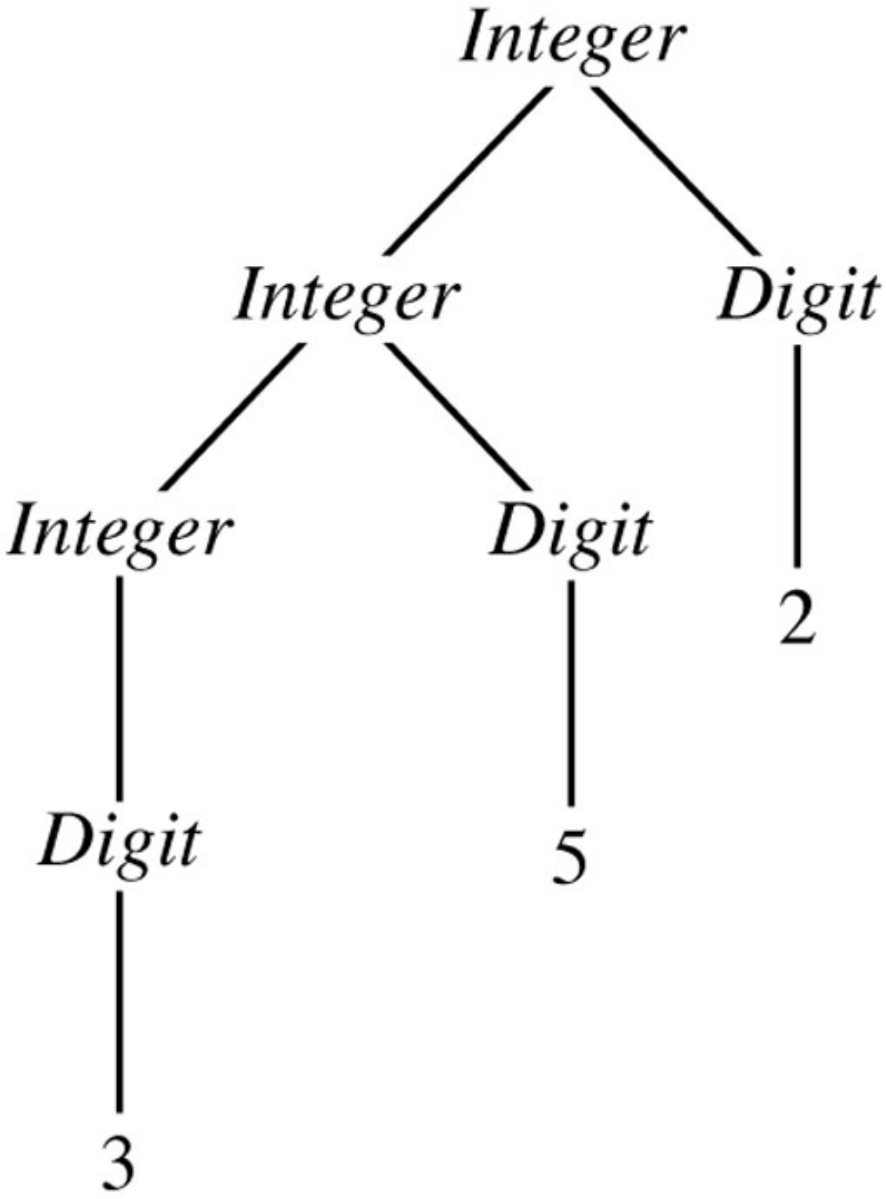
Each leaf node represents a symbol of the derived string, reading from left to right.

E.g., The step $Integer \Rightarrow Integer\ Digit$ appears in the parse tree as:



Parse Tree for 352 as an *Integer*

Figure 2.1



Arithmetic Expression Grammar

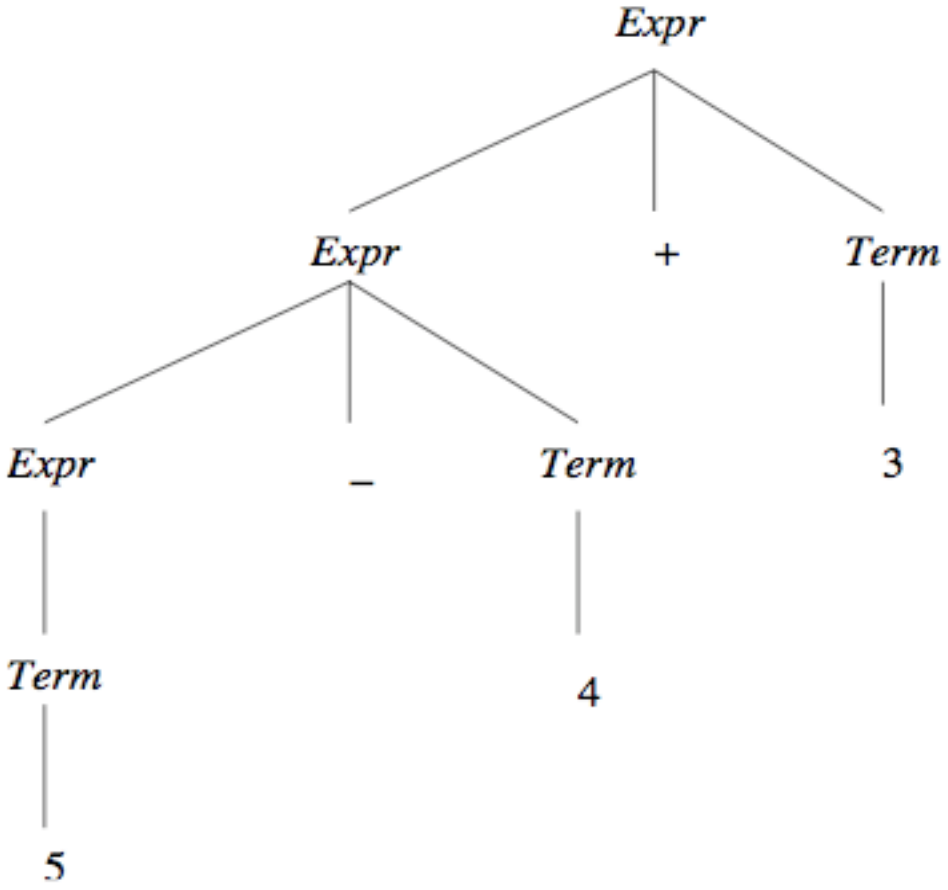
The following grammar defines the language of arithmetic expressions with 1-digit integers, addition, and subtraction.

Expr → *Expr* + *Term* | *Expr* − *Term* | *Term*

Term → *0* | ... | *9* | (*Expr*)

Parse of the String 5-4+3

Figure 2.2



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;

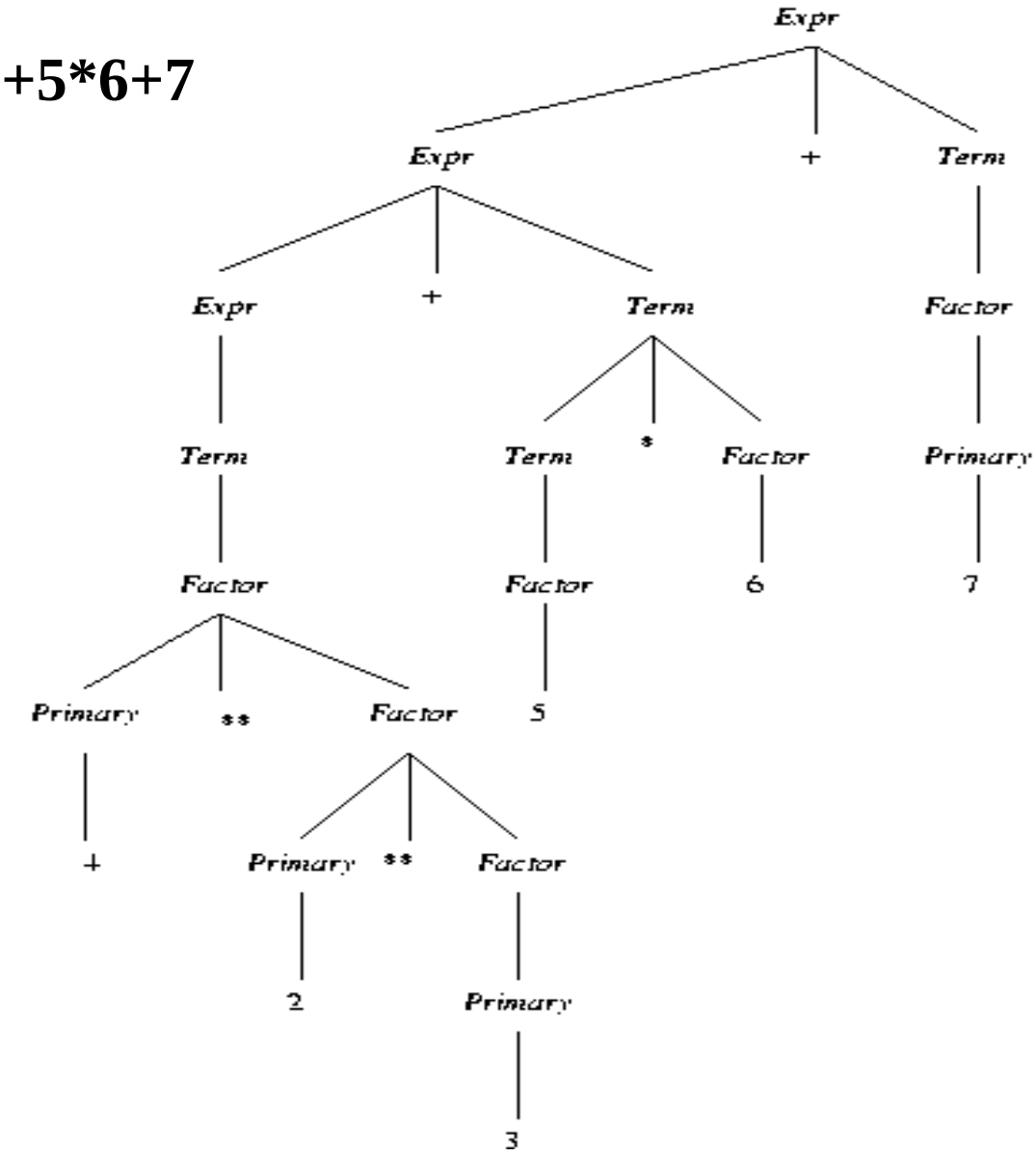
** and / have higher precedence than + and − .*

Consider the more interesting grammar G_1:

```
Expr → Expr + Term | Expr - Term | Term
Term → Term * Factor | Term / Factor |
   Term % Factor | Factor
Factor → Primary ** Factor | Primary
Primary → 0 | ... | 9 | ( Expr )
```

Parse of 42**3+5*6+7**
for Grammar G_1

Figure 2.3



Associativity and Precedence for Grammar G_1

Table 2.1

Precedence	Associativity	Operators
3	right	**
2	left	* / %
1	left	+ -

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.

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

Expr → *Expr Op Expr* | (*Expr*) | *Integer*

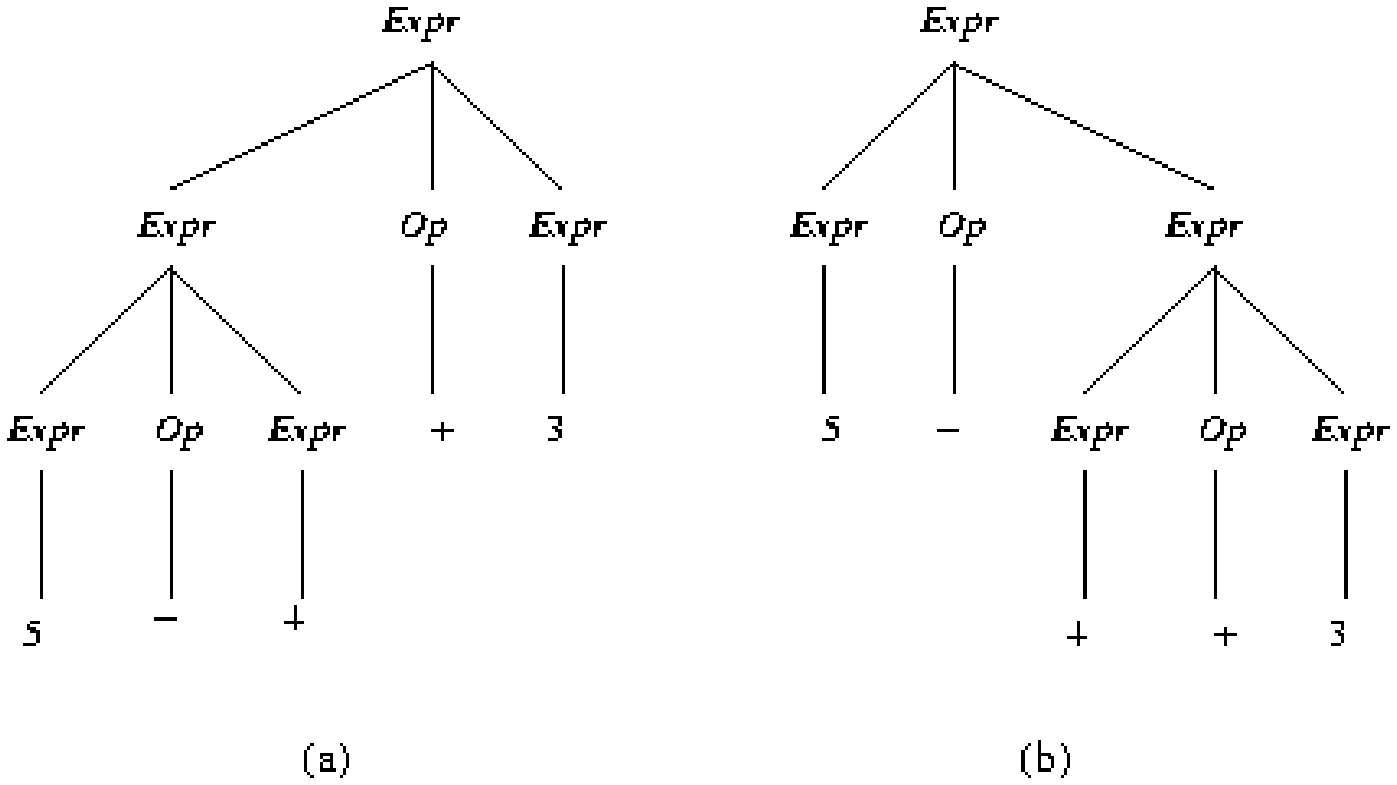
Op → + | - | * | / | % | **

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 → if (*Expression*)*Statement* |
    if (*Expression*) *Statement* else *Statement*
Statement → *Assignment* | *IfStatement* | *Block*
Block → { *Statements* }
Statements → *Statements Statement* | *Statement*

Example

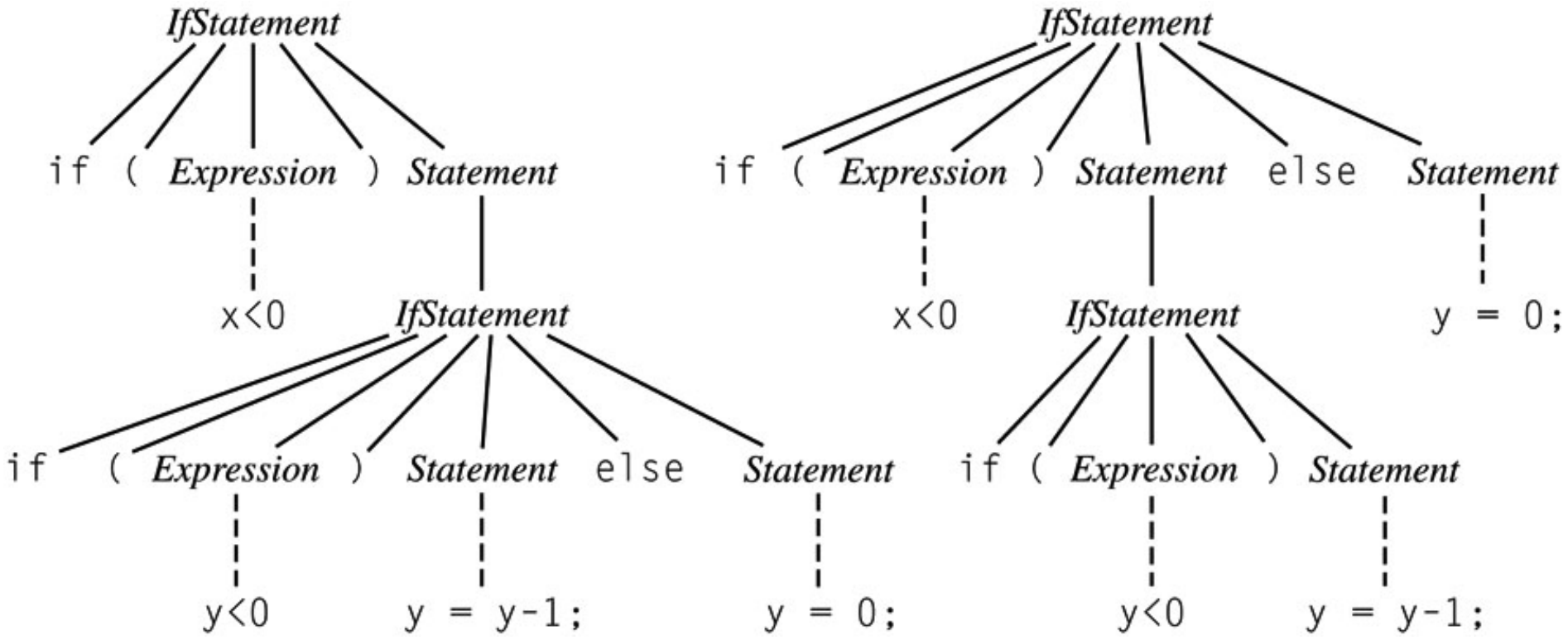
With which '`if`' does the following '`else`' associate

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

Answer: *either one!*

The *Dangling Else* Ambiguity

Figure 2.5



Solving the dangling else ambiguity

Algol 60, C, C++: associate each else with closest if; use `{}` or `begin…end` to override.

Algol 68, Modula, Ada: use explicit delimiter to end every conditional (e.g., `if…fi`)

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*.

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 *−*

```
Exp → Term { ( + | - )  Term }
IfSt → if ( Exp ) Stat [ else Stat ]
```

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

```
IfSt:  if ( Exp ) Stat ElsePart_opt
ElsePart:  else 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:

$A \rightarrow x\ A'\ z$

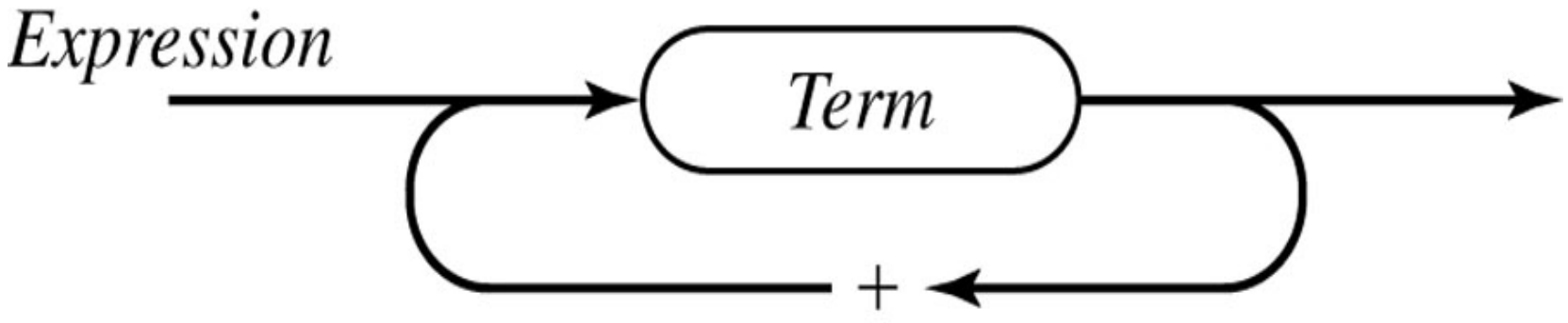
$A' \rightarrow \mid y\ A'$

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

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

Syntax Diagram for *Expressions* with Addition

Figure 2.6



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.

Clite Grammar: Statements

Program → `int main ( )` { *Declarations Statements* }

Declarations → **{** *Declaration* **}**

Declaration → *Type Identifier* **[** [*Integer*] **]** **{** , *Identifier* **[** [*Integer*] **]** **}**

Type → `int | bool | float | char`

Statements → **{** *Statement* **}**

Statement → `;` | *Block* | *Assignment* | *IfStatement* | *WhileStatement*

Block → { *Statements* }

Assignment → *Identifier* **[** [*Expression*] **]** = *Expression* ;

IfStatement → `if` (*Expression*) *Statement* **[** `else` *Statement* **]**

WhileStatement → `while` (*Expression*) *Statement*

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*)

Clite grammar: lexical level

Identifier → *Letter* { *Letter* | *Digit* }

Letter → `a` | `b` | `...` | `z` | `A` | `B` | `...` | `Z`

Digit → `0` | `1` | `...` | `9`

Literal → *Integer* | *Boolean* | *Float* | *Char*

Integer → *Digit* { *Digit* }

Boolean → `true` | `false`

Float → *Integer* `.` *Integer*

Char → `'` *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*

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*

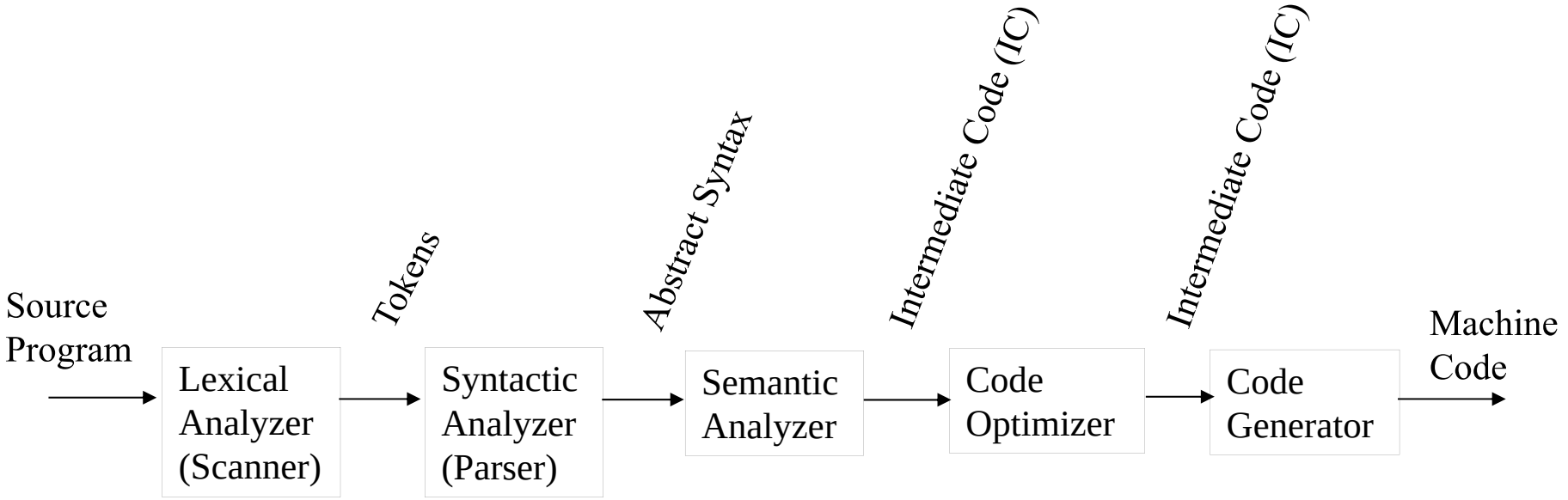
Identifier

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)

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*

Linking Syntax and Semantics

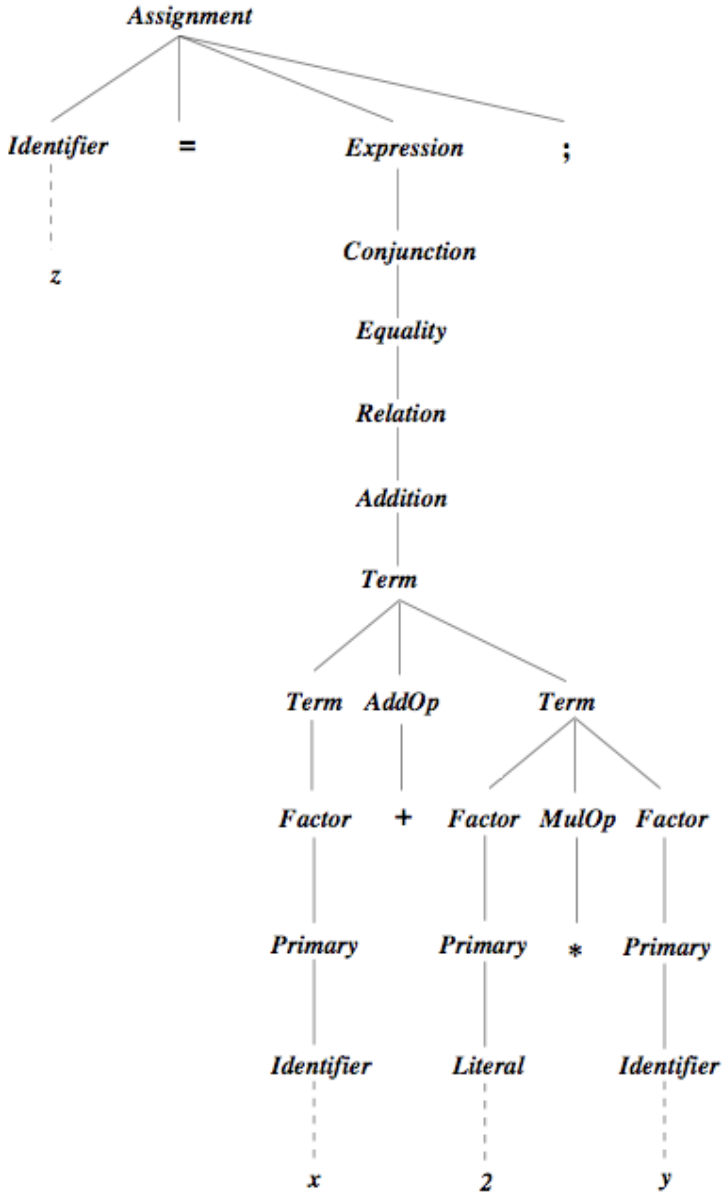
Output: parse tree is inefficient

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.

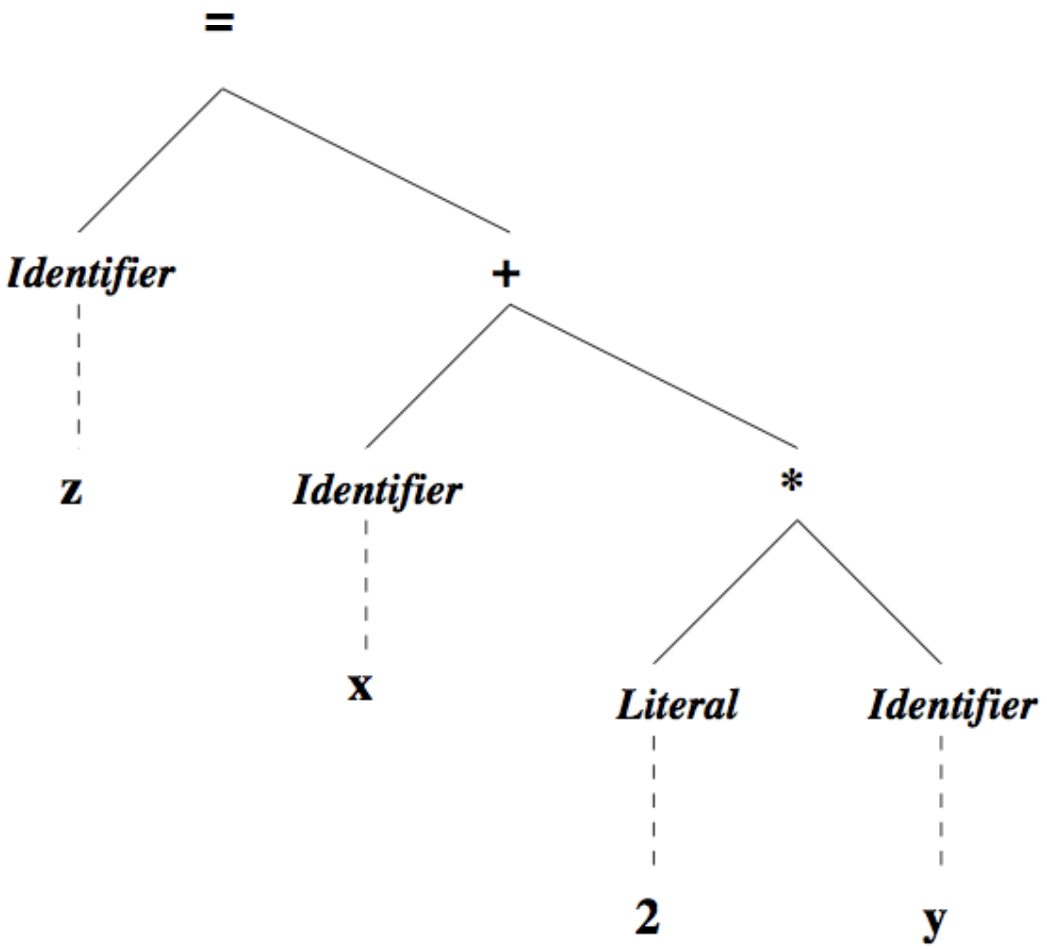
- *Remove separator/punctuation terminal symbols*
- *Remove all trivial root nonterminals*
- *Replace remaining nonterminals with leaf terminals*

Example: Fig. 2.10

Abstract Syntax Tree

```
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

```
while i < n do begin
 i := i + 1;
end;
```

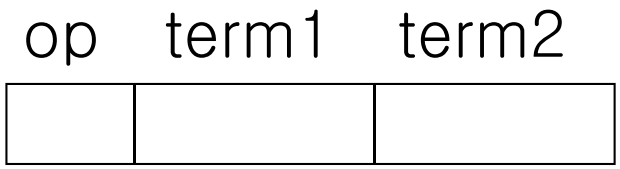
C/C++

```
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`.

Example of Abstract Syntax Tree

Binary node



Abstract Syntax Tree
for `x+2*y` (Fig 2.13)