NCCU
Advanced Programming Languages
高等程式語言

Instructor:
資科系
陳 恭 副教授
Spring 2006

Lecture 2: Syntax

Agenda

- Defining a Programming Language
- Concepts from Formal Languages
- Grammar Specification
  - BNF & CFG
- Parsing and Syntax Ambiguities
- Some Syntactic Features of C (and Java)
- EBNF & Syntax Chart
- Lexical Elements of a PL
- AST

Defining Programming Languages

- 如何界定/定義一程式語言?
  - 語法Syntax、語意semantics
- 定義一個語言還是要用另一個語言!
  - Object language: 被定義的語言, 例如英文
  - Meta language: 用來定義的語言, 例如中文

- 程式語言的定義方式
  - Informal like natural language
    - 多義特性, 不容易界定清楚
  - 以Formal languages 嚴格界定 (Metalanguage)
    - syntax 已有成熟的技術
    - semantics 較複雜多了, 必須使用數學工具

Description of a Language

- Even in English,we can have
  - Wrong Syntax, ‘Correct Semantics’
    - Eg. “I is a University Student”
  - Correct Syntax, Wrong Semantics
    - Eg. Sentence = Subject + verb + object

      “A rectangle loves chicken rice”
  - Wrong Syntax, Wrong Semantics
    - Eg. “House worm chair long fast twenty-two”
  - Correct Syntax, Correct Semantics

Description of a Language

- Similarly, in programming we can have:
  - Semantics (Meaning)
    - You have in your mind, to express a conditional, when true do something, when false, do something else.
  - Syntax (Form/Structure)
    - C:

```
if (condition) {
   …part1…
} else {
   …part2…
}
```

    - Pascal:

```
if condition
   begin…part1…end
else
   begin…part2…end
```

    - LISP:

```
(if (condition) (part1) (part2))
```

Description of a Programming Language

- A programming language is a man-made artifact.
  - What is a legal program
    - For instance, X := A+B; (PASCAL)

      X = A+B (FORTRAN)

      X.Y[3].Z (C)
    - Different languages employ different notations.
    - We need to define precisely what constitute a legal program.
  - What is a program suppose to compute.
    - For instance, 1+2*3 (7 in C, but 9 in APL!)
    - It occurs frequently that a program does not compute what is intended because the programmer misunderstands the language.

Concepts from Formal Languages

語言的形式化處理

- 就形式而言, 一語言乃是一個由 “句子” (sentence)所組成的集合, (通常是一無限集合).
- 所謂 sentence 則是由一選定的 “字母集” (alphabet set) 的字母所組成的“字串” (string).

- Example: *A*(lphabet) = { ‘程式’, ‘語’, ‘言’, ‘PL’, ‘=‘, ‘ ‘ }
  - “程式語言 = PL” is a *sentence* (string) constructed from *A*.
  - { *“程式語言 = PL”, “程式 PL= 語言”* } is a trivial language based on *A*. –*Not an interesting language, though.*

•How to construct an interesting language (an infinite set)?

An Example Language: Floating Point Nmbers

- Let's try to construct the language of *positive floating point numbers.*
  - Alphabet = { '-', '.', '0', '1', …, '9' }
  - PFPN = {0, 0.00001, 0.00002, 0.00003, … }
- By enumeration?
  - Does "3.141596745" $\in$ PFPN?
- Can we do better? How?

Use Grammar!

--A finite set of rules for describing an infinite set.

A Grammar for PFPN

< PFP-number> ::= <integer-part> ‘ · ’ <fraction>

<integer-part> ::= <digit> | <integer-part> <digit>

<fraction> ::= <digit> | <digit> <fraction>

<digit> ::= ‘0’|’1’|’2’|’3’|’4’|’5’|’6’|’7’|’8’|’9’

Recursion in the rules

‘’: separate object language elements from meta language elements.

Grammars <T, N, S, P>

- Contain 4 components
  - terminal symbols
    - atomic components of statements in the language
      - appear in source programs
    - identifiers, operators, punctuation, keywords
  - non-terminal symbols
    - intermediate elements in producing terminal symbols
    - never appear in source program
  - start (or goal) symbol
    - a special nonterminal which is the starting symbol for producing statements
  - productions
    - rules for transforming nonterminal symbols into terminals or other nonterminals
    - <nonterminal> ::= terminals and/or nonterminals
    - < PFP-number> ::= <integer-part> ‘ · ’ <fraction> **→BNF notation**
    - each has lefthand side (LHS) and righthand side (RHS)

Backus-Naur Form (BNF): devised first for specifying Algol 60

BNF for Positive Floating-Point Numbers

T = { ‘0’,’1’,’2’,’3’,’4’,’5’,’6’,’7’,’8’,’9’, ‘ · ’ }

N = { PFP-number, integer-part, fraction, digit }

S = <real-number>

<integer-part>::= <digit>
<integer-part>::= <integer-part> <digit>

P ={ < PFP-number> ::= <integer-part> · <fraction>

<integer-part>::= <digit> | <integer-part> <digit>

<fraction> ::= <digit> | <digit> <fraction>

<digit> ::= ‘0’|’1’|’2’|’3’|’4’|’5’|’6’|’7’|’8’|’9’ }

- *Left* recursion or *right* recursion

Derivations Using Grammar Rules

Derivation (推導、衍): 從一nonterminal開始，相繼使用production rules 把nonterminals 替換爲其 production rules 的RHS的替換序列 。

<PFP-number> → <integer-part> · <fraction>
→ <integer-part> <digit> . <fraction>
→ <digit> <digit> · <fraction>
→ 2 <digit> · <fraction>
→ 2 1 · <fraction>
→ 2 1 · < digit> <fraction>
→ 2 1 · 8 <fraction>
→ 2 1 · 8 <digit>
→ 2 1 · 8 9

<PFP-number> →* 21.89

- Leftmost derivation - 每次選最左邊的nonterminal 來作替換

Generator Views of Grammars

- Given a grammar, we can derive a language (set of sentences) from it.
  - From the grammar of PFPN, we derive *L_pfpn*
    - 0.1 ∈ *L_pfpn* because <PFP_number> →* 0.1
    - 0.11 ∈ *L_pfpn* because <PFP_number> →* 0.11
    - 11.111 ∈ *L_pfpn* because <PFP_number> →* 11.111
    - …

給定 G = (T, N, P, S)，定義由此文法導出的語言

$$L(G) = \{ t \mid S \rightarrow^* t \text{ and } t \in T^* \}$$

I.e. 所有從S推導出的,僅由terminals所組成的字串稱爲此文法所定義之語言。

Theoretical Concerns

- 給定想界定的語言L, 如何定出G且證明 'L(G) = L' ?
  - Is *L_pfpn* the set of positive floating numbers we want?

< PFP-number> ::= <integer-part> ' · ' <fraction>
<integer-part> ::= <digit> | <integer-part> <digit>
<fraction> ::= <digit> | <digit> <fraction>
<digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

We have the following observations

- <PFP-number> →* "000.1234" , so "000.1234" ∈ *L_pfpn*
- <PFP-number> →* "0.100" , so "0.1000" ∈ *L_pfpn*
  - But ".38" ∉ *L_pfpn*
- Two grammars can generate the same language. I.e., L(G1) = L(G2). Prove it!

Recognizer View of Grammars

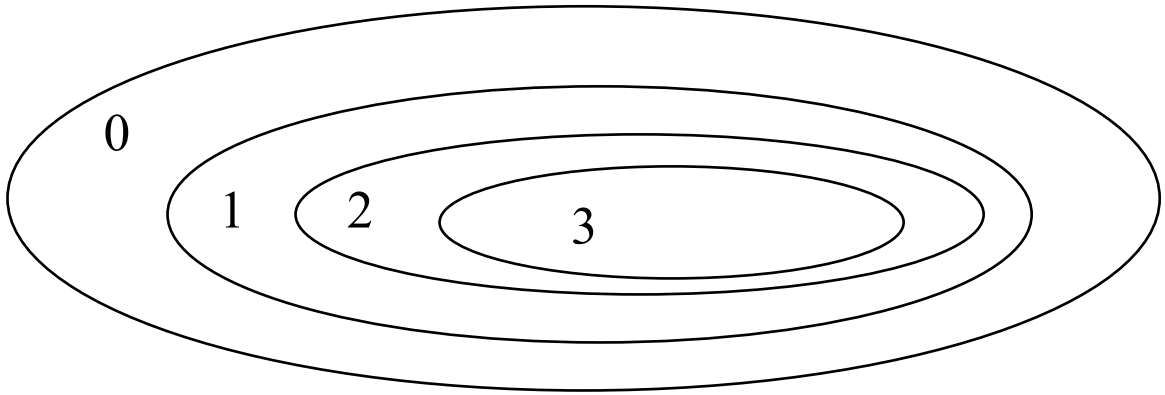
- Grammars can serve as “generators” or “recognizers”
  - Given a Grammar<T, N, S,, P> and a string (sentence) s ∈ T*, does s ∈ L(G)?
  - “000.1234” ∈ *L_pfpn?*

- Recognizer views are used in compilers
  - A program is simply a string input to a compiler
  - The parser component of a compiler will determine whether an input program is a legal sentence.

- 什麼形式的 grammar 所定義的語言一定可以以程式(parser)判斷一字串是否屬於此語言?

Theoretical Concerns

- 什麼形式的 grammar 所定義的語言一定可以以程式判斷一字串是否屬於此語言?
- 各式各樣的Grammar 是否也有結構上的關係?
- Type 0, type 1, type 2, type 3 grammars



•Regular expressions, Context-Free grammar, Context-Sensitive grammars,

Context-Free Grammar (CFG)

- Grammar G 的任一production rule 中， LHS 只能是單一的 nonterminal, 例如

  <PFP-number> ::= <integer-part> ． <fraction>

- 對照 *Context-Sensitive Grammar* (CSG)中之 production rules, 其<LHS>可能有含多個符號，包括terminals 與 nonterminals. 例如

  ‘r’ <PFP-number> ::= <integer-part> ． <fraction>

  <A><B> ::= a<B>a | b<A>b

An Example of CSG

例： T= {a,b}

N= { <after-a>,<s>}

P= { <s>::= <after-a>b

| a<after-a>,

a<after-a>::= aa <after-a> | b }

推導： <s> → <after-a> b → ???

<s>→ a<after-a> → aa <after-a>

PL & CFG

- Major parts of programming languages are Context free BUT NOT ALL
- Example: a := b + c;
  - Assignment statement has the form

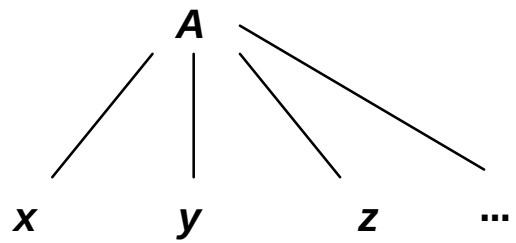
    <variable> := <expression>
- 程式語言的語法多以context-free的方式表示，但實際上有很多是context-sensitive的，例如變數需先宣告才能使用，函數呼叫所傳參數個數必須與宣告時的個數一致.
- 但是 CSG 的 recognizer 不可行，所以程式語言語法以 CFG + informal rules (static semantics) 表示

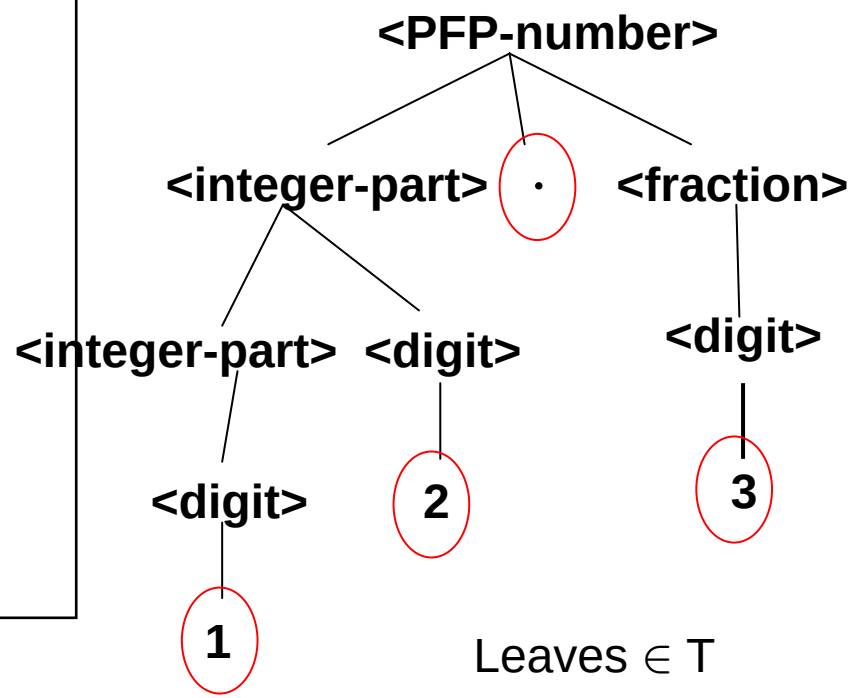
Parsing, Parse Trees, and Syntactic Ambiguity

Derivation Trees

- A derivation tree is the tree resulting from applying productions to rewrite *start symbol:*
  - *top-down view*
  - *From <PFP-number>*
- a parse tree is the same tree starting with *terminals* and building back to the *start symbol*
  - A bottom-up view
  - From “1.23” to the tree

Production: A → xyz…





Parsing & Parse Trees

- Parsing:由字組句的過程；反推是否存在 derivations such that "S →* *sentence*" .
  E.g. "1 . 23" is a legal PFP number.
- Derivations express the structure of syntax, but not very well: there can be lots of different derivations for the same structure, e.g. *Left-most derivation vs. Right-most derivation.*
- A parse tree better expresses the structure inherent in a derivation.(2D方式呈現)

A Grammar for Arithmetic Expressions

T = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, A, B, C, (,) }

N = { <id>, <expr>, <num>, <digit> }

P = <id> ::= A | B | C

<expr> ::= <expr> ‘+’ <expr>

| <expr> ‘-’ <expr>

| <expr> ‘*’ <expr>

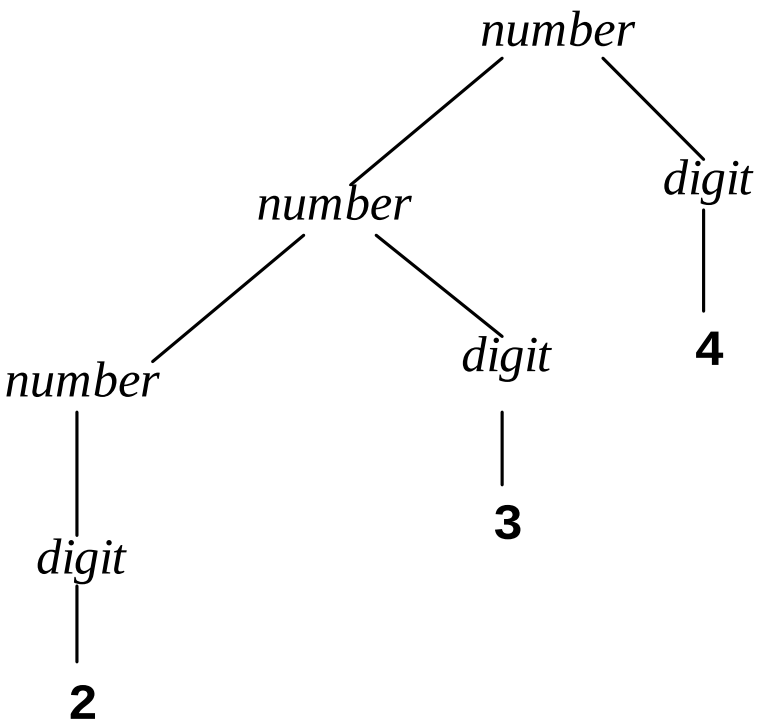
| <expr> ‘/’ <expr>

| ‘(‘ <expr> ‘)’ | <num> | <id>

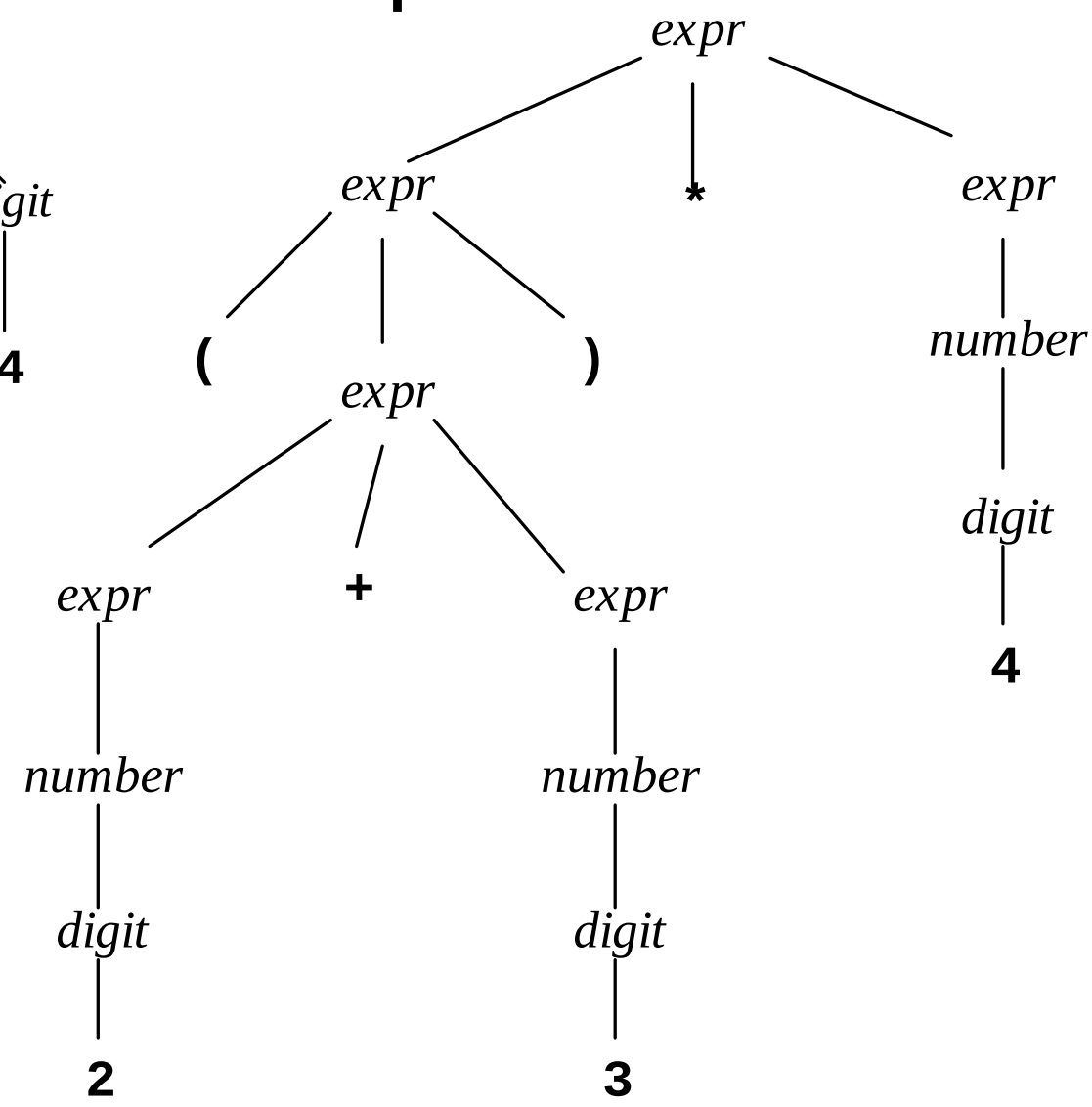
<num> ::= <digit> | <digit><num>

<digit> ::= ‘0’ | ‘1’ | … | ‘9’

Example 1:



Example 2:

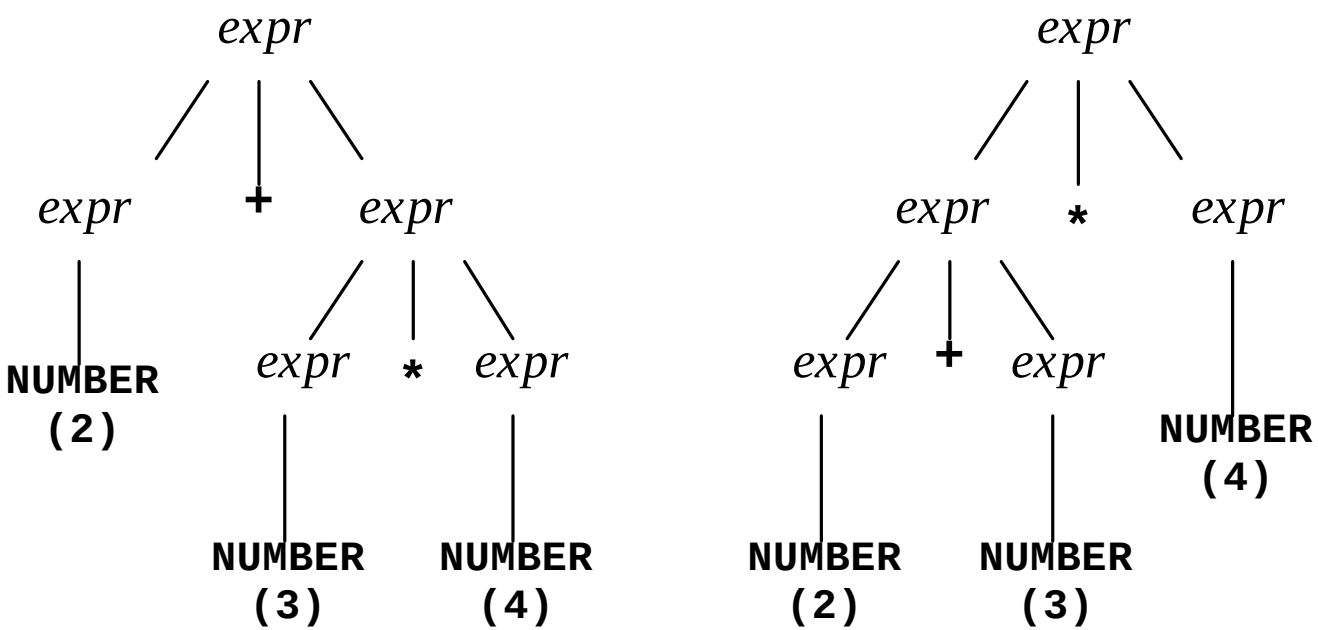


Syntactic Ambiguity

- Grammars don't always specify unique parse trees for every string in the language: a grammar is *ambiguous* *if some string has two distinct parse (or abstract syntax) trees (not just two distinct derivations).*

  - Expression: "2 + 3 * 4"

precedence



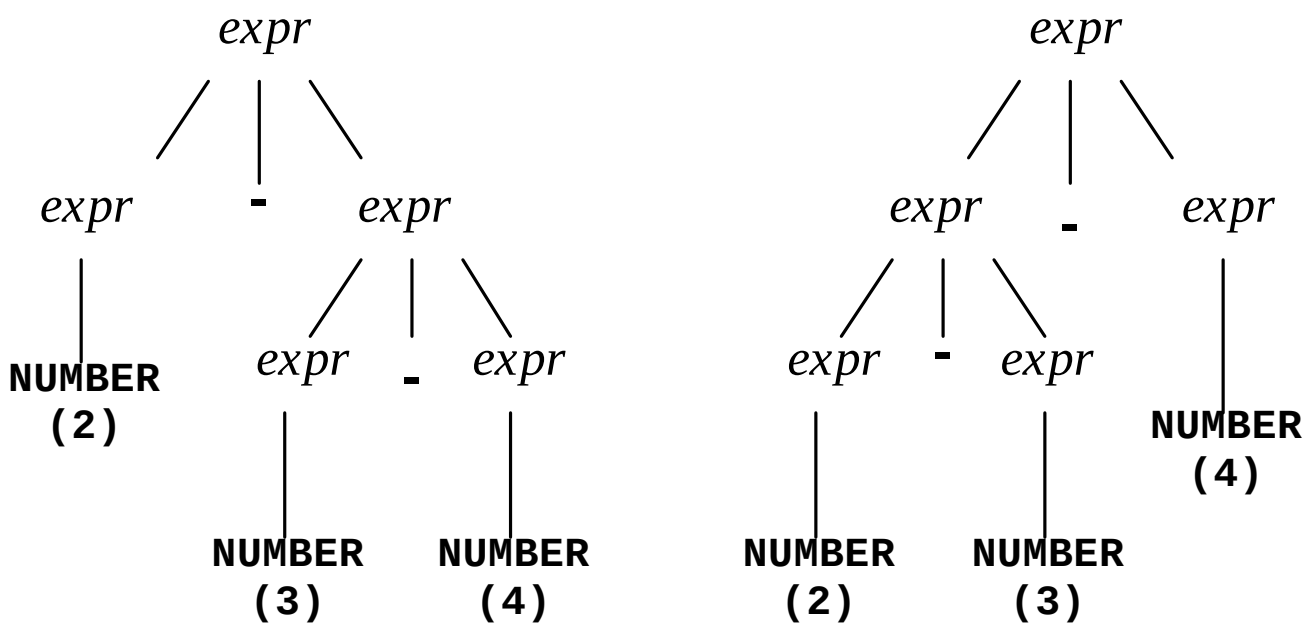
Syntactic Ambiguity, Cont'd

- Grammar (with subtraction):

  expr **::=** ***expr*** + ***expr*** **|** ***expr*** - ***expr***
  | (***expr***) **|** NUMBER

- Expression: `2 - 3 - 4`
- Parse trees:

associativity



Syntactic Ambiguity, Cont'd

- Ambiguity is *usually* bad and must be removed.
  - Syntactic ambiguity 可能導致 semantic ambiguity (ill-definedness):

    3 * (3+4) vs. (3 * 3) + 4

  - But 3 * (3 * 4) = (3 * 3) * 4
- Often the grammar can be rewritten to make the correct choice. Or the specification must define some "*disambiguating rules*."

Avoiding Syntactic Ambiguity

1. 簡單改寫文法，但也改變語言
2. 複雜改寫文法，但不改變語言
   - Arithmetic expressions 需考慮運算子(operator)
     - 優先順序 (precedence): 先乘除後加減
     - 結合性 (associativity): 左結合 or 右結合
   - 解決之道:
   - 優先順序 (precedence): 一層級一non-terminal
   - 結合性 (associativity): 左結合算子用 left-recursion

1. Rewrite the Grammar yet Change the Language

- (Delimiter) Fully-parenthesized expressions:
  expr ::= **(** ***expr*** + ***expr*** **)** **|** **(** ***expr*** - ***expr*** **)**
  | NUMBER

  so: **(** **(** 2 - 3) - 4)
  and: (2 + (3 * 4))

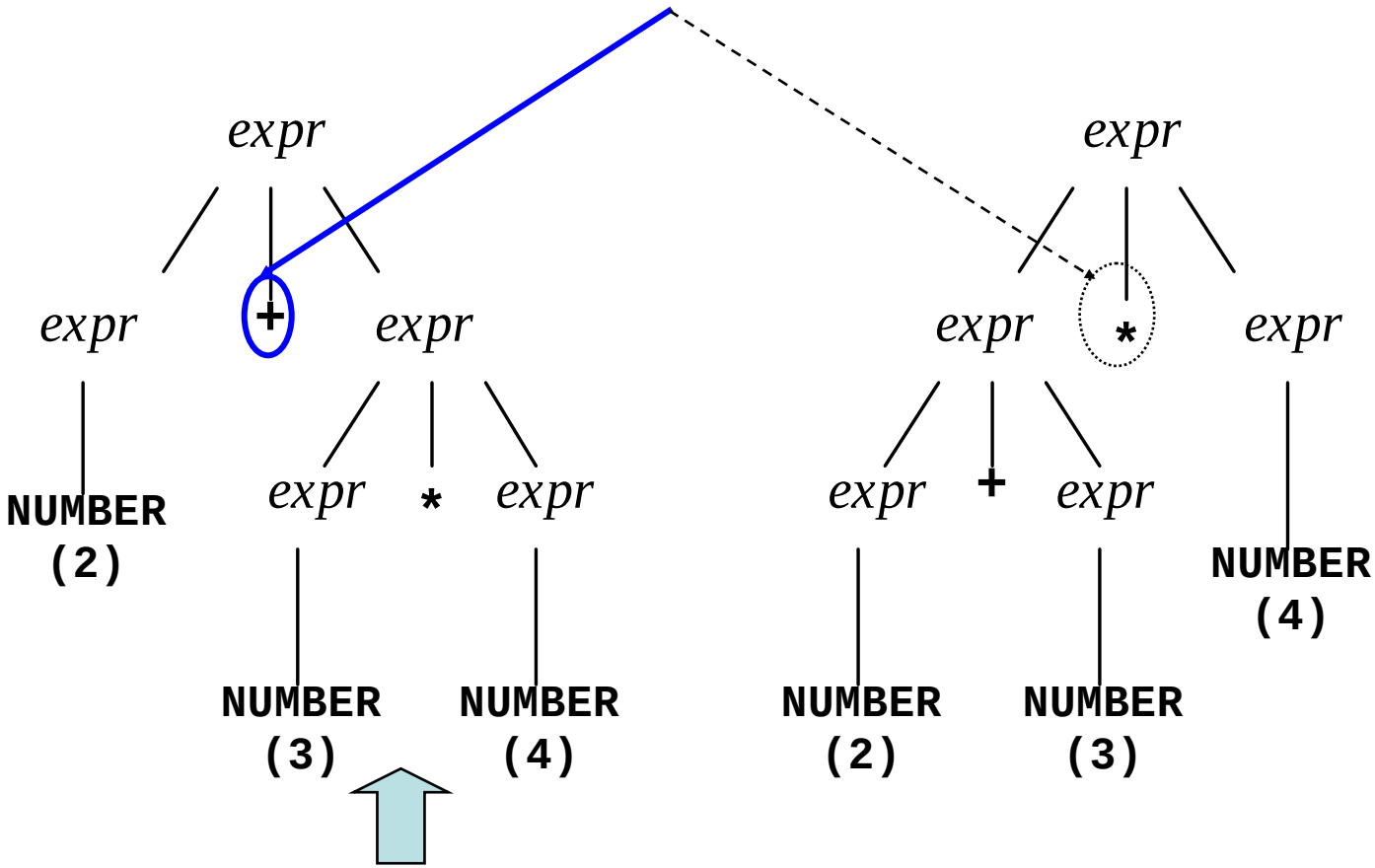
- Prefix expressions:
  expr ::= **+** ***expr expr*** **|** **-** ***expr expr***
  | NUMBER

  so: - - 2 3 4
  and: + 2 * 3 4

2. Rewrite the Grammar yet Keep the Language

Revisit the example:

優先順序低者先浮現

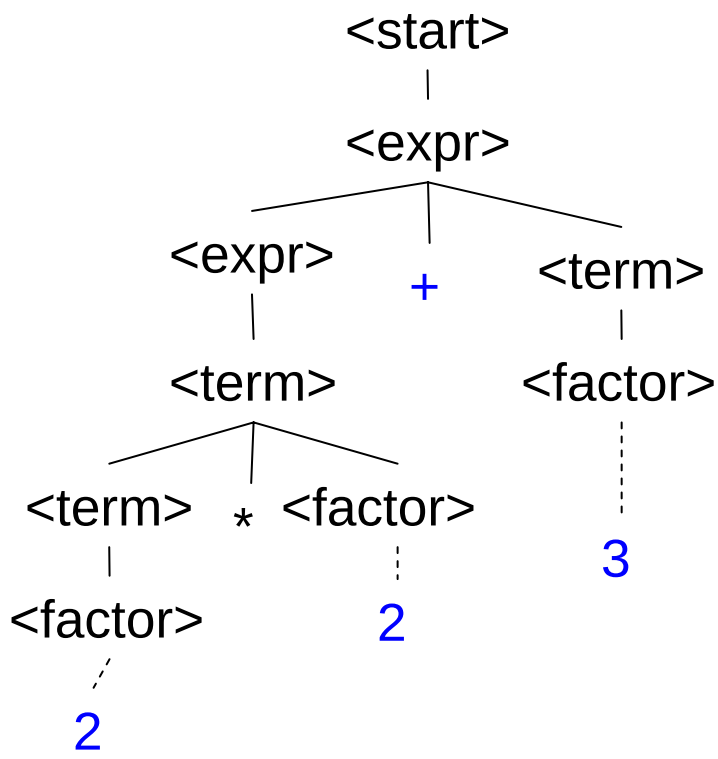


2. Resolving Ambiguity by Imposing Precedence

一優先層級一non-terminal

```
<start>  ::= <expr>
<expr>   ::= <expr> - <term>
           |  <expr> + <term>
           | <term>
<term>   ::= <term> * <factor>
           | <term> / <factor>
           | <factor>
<factor> ::= <number> | <id>
           | ( <expr> )
```

Parse "**2 * 2 + 3**"



Resolving Ambiguity by Imposing Precedence

Only one possible derivation for “2 * 2 + 3” from grammar (therefore, only one possible parse tree)

1. <start> ::= <expr>
2. <expr> ::= <expr> - <term>
   | <expr> + <term>
   | <term>
3. <term> ::= <term> * <factor>
   | <term> / <factor>
   | <factor>
4. <factor> ::= <number> | <id>
   | (<expr>)
5. <number> ::= …

<start>	
<expr>	[rule 1]
<expr> + <term>	[rule 2b]
<term> + <term>	[rule 2c]
<term> * <factor> + <term>	[rule 3a]
<factor> * <factor> + <term>	[rule 3c]
…	
2 * <factor> + <term>	[rule 5]
…	
2 * 2 + <term>	[rule 5]
2 * 2 + <factor>	[rule 3c]
…	
2 * 2 + 3	[rule 5]

“Dangling-Else” Ambiguity

<start> ::= <stmt>
<stmt> ::= <if-stmt> | <assign>

<if-stmt> ::= **if** <expr> **then** <stmt>
| **if** <expr> **then** <stmt> **else** <stmt>

<assign> ::= <ident> := <digit>

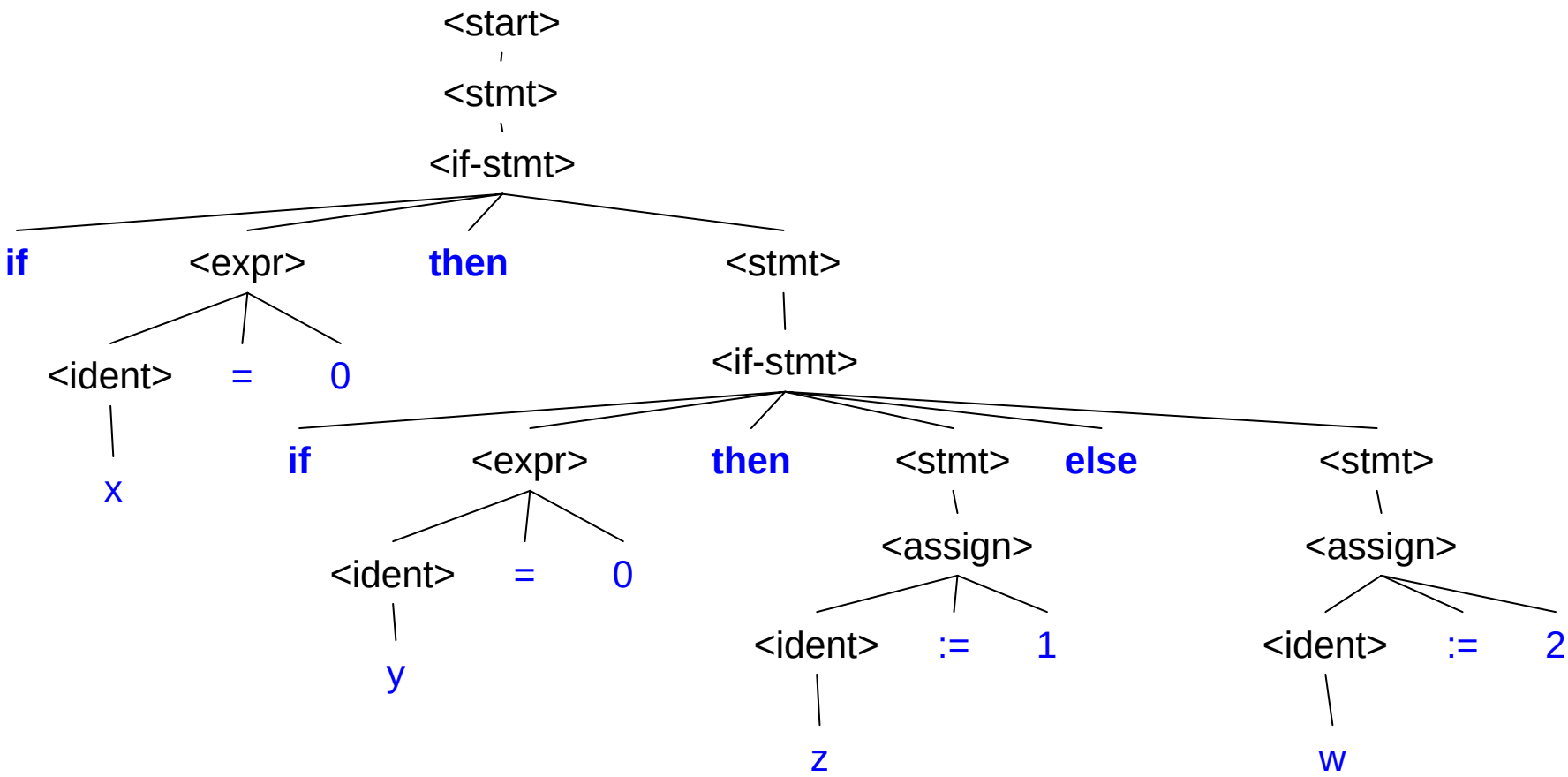
<expr> ::= <ident> = 0

<digit> ::= 0 | 1 | … | 9
<ident> ::= a | b | … | z

Dangling-Else Ambiguity

- How are compound if statements parsed with this grammar?

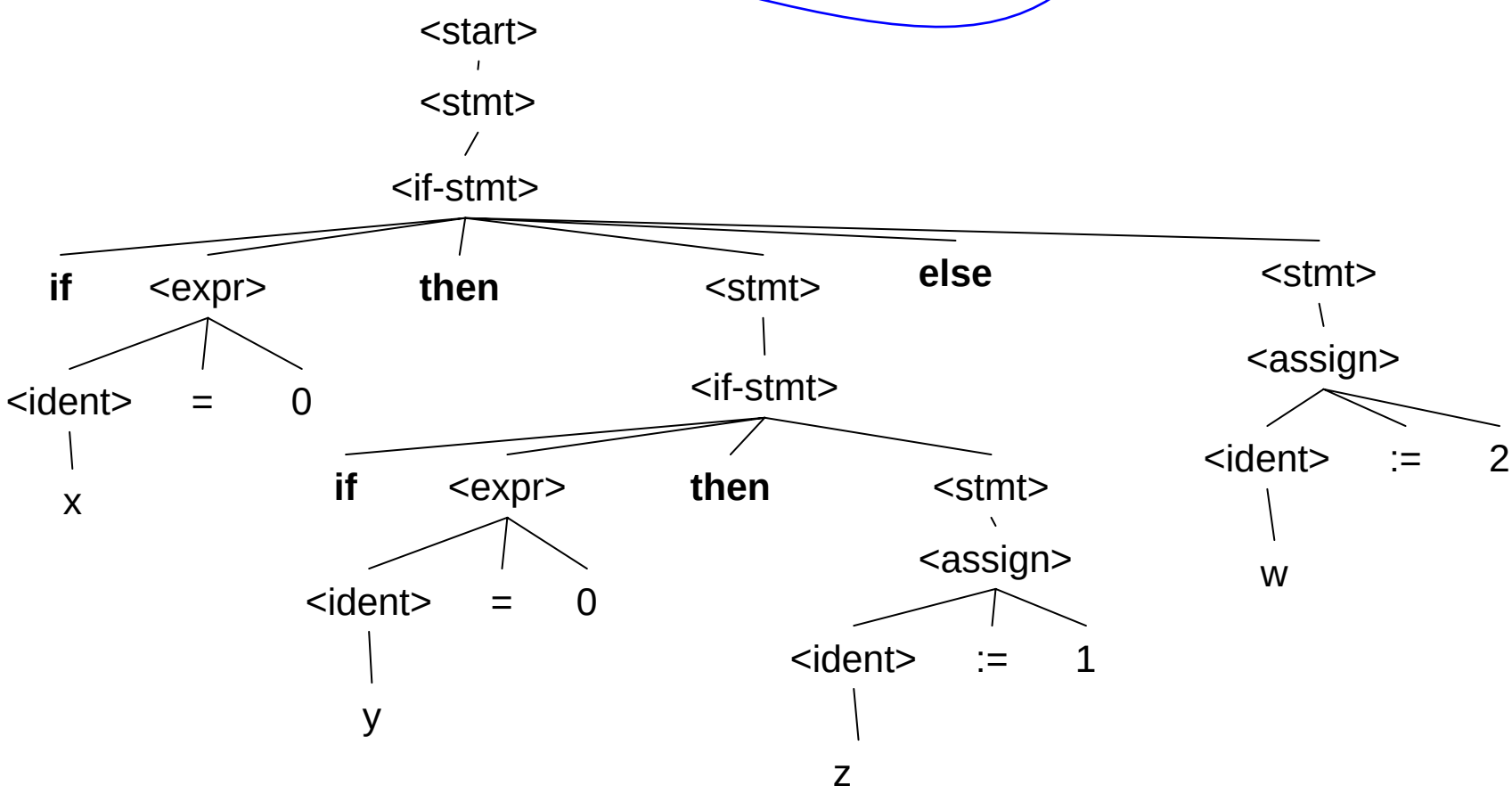
E.g., parse "***if*** *x = 0* ***then*** ***if*** *y = 0* ***then*** *z := 1* ***else*** *w := 2*"



Dangling-Else Ambiguity

Alternatively,

parse “**if** x = 0 **then** **if** y = 0 **then** z := 1 **else** w := 2”



如何避免Dangling-Else Ambiguity?

- Use additional keywords (Modula-2, VB)

  <Statement>::= …

  | if <expr> then <stmt_list> end

  | if <expr> then <stmt_list>
  else <stmt_list> end

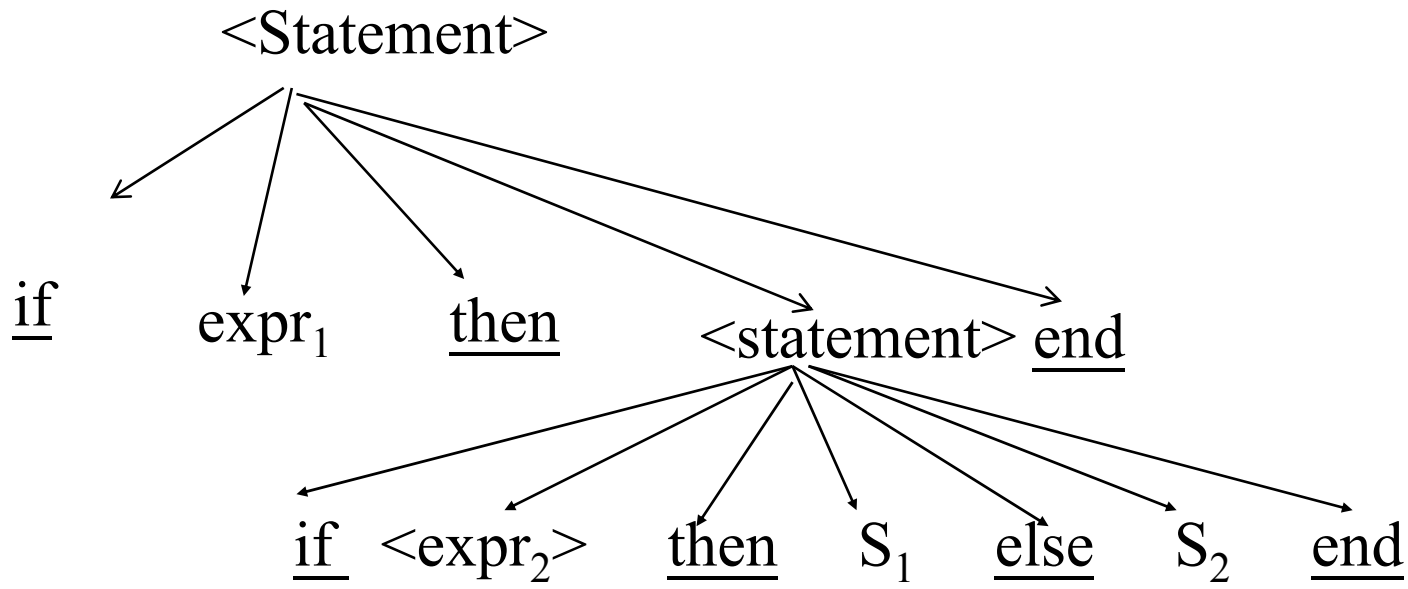
  | …

- Rewrite the grammar according to a given rule
  - 每個“else”應與最近尚未配對的“then”來配

Additional Key Word

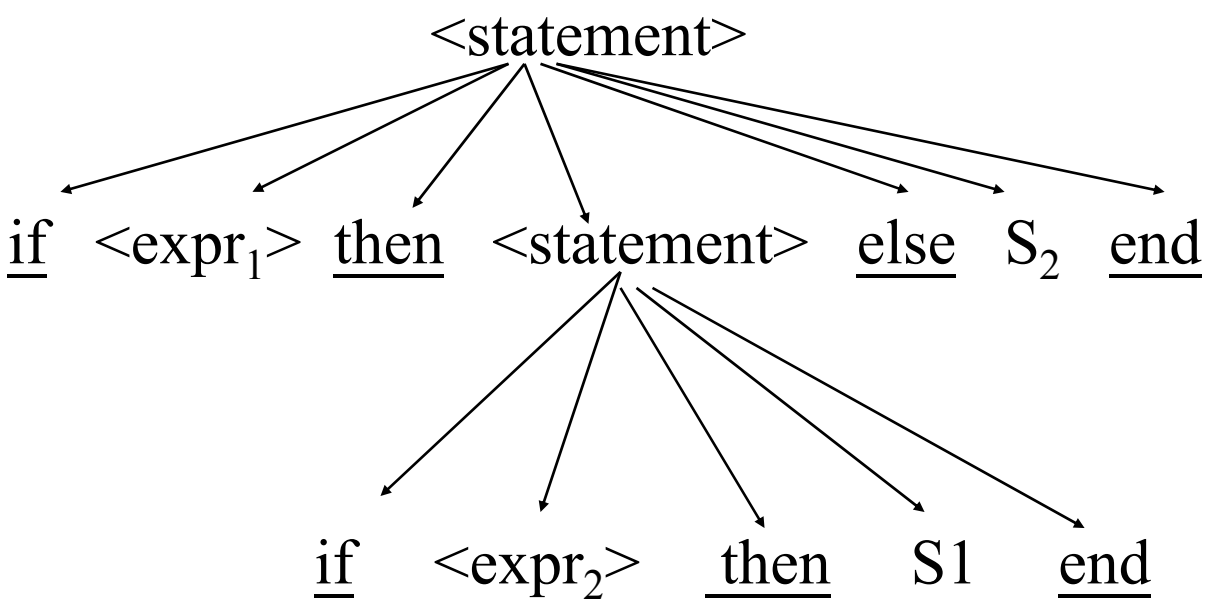
- 必須寫成

  if < $expr_1$> then if <$expr_2$> then S_1 else S_2 end end



或是

if <expr$_1$> then if <expr$_2$> then S_1 end else S_2 end



而不會有ambiguity

Rewrite the Grammar

- 每個“else”應與最近尚未配對的“then”來配. 所以前例
  “if <expr$_1$> then if <expr$_2$> then S_1 else S_2” 應解讀成

  if <expr$_1$> then

  if <expr$_2$> then S_1 else S_2

- 這個慣例可藉修改grammar,以*Additional Nonterminals* 的方式來定義清楚.
- Idea: #(then) >= #(else) 恆真

  number_of

<statement> ::= <matched_stmt> (‘=’)

| <unmatched_stmt> (‘>’)

<matched_stmt> ::= if <expr> then <matched_stmt>

else <matched_stmt>

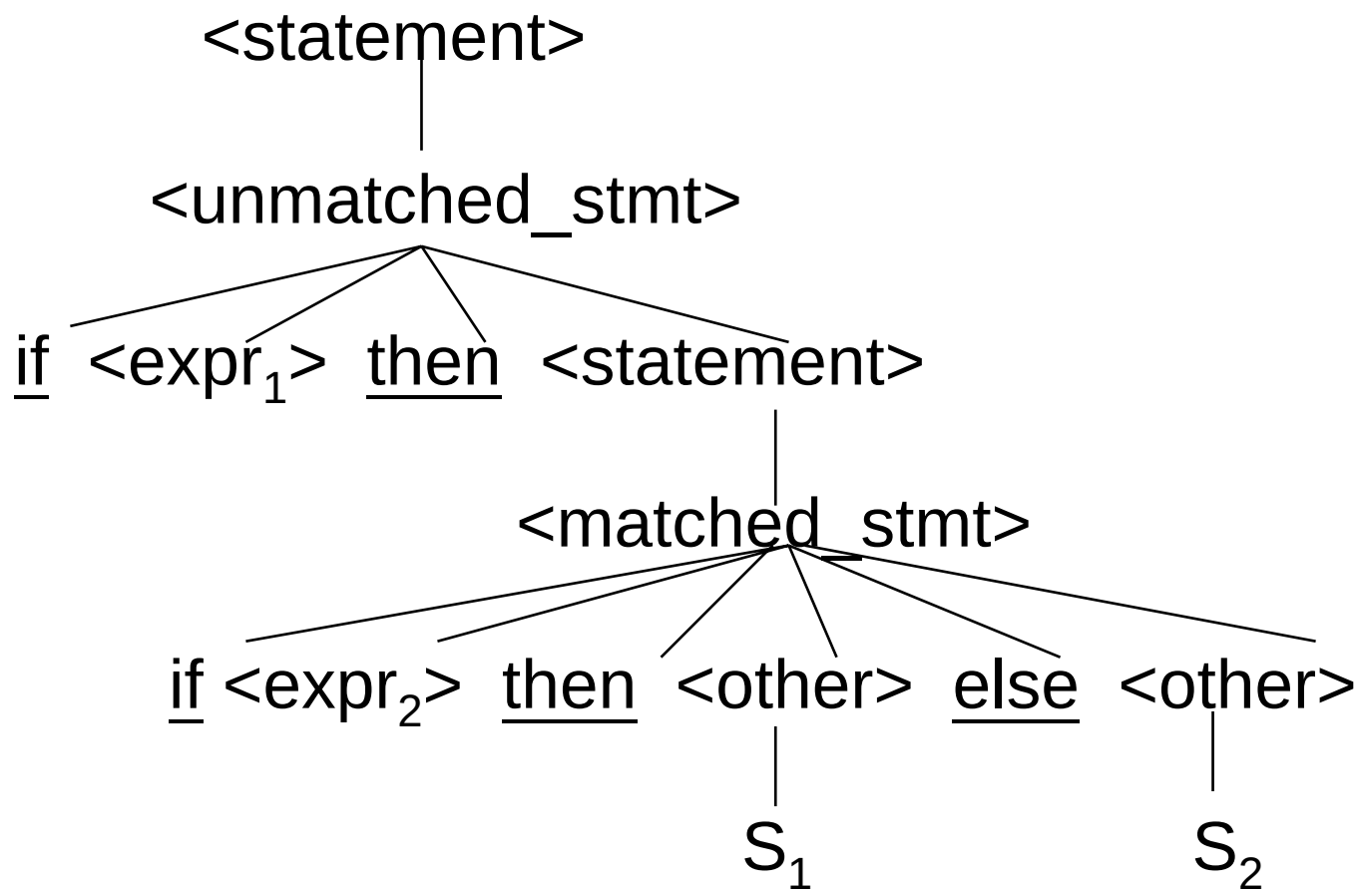
| <other_stmt>

<unmatched_stmt> ::= if <expr> then <statement>

| if <expr> then <matched_stmt>

else <unmatched_stmt>

- 前例:


<statement>
<unmatched_stmt>
if <expr1> then <statement>
<matched_stmt>
if <expr2> then <other> else <other>
S1
S2

EBNF and Syntax Chart

E(xtended) BNF

- Alternative notation for specifying syntax; use repetition and optional features instead of recurion.
- Braces, { and }, represent zero or more repetitions.
  - E.g., `float_num ::= {digit} . digit {digit}`
- Brackets, [and], represent an optional construct.
  `float_num ::= [-]{digit}.digit{digit}`
- A vertical bar,'|', represents a choice.
- Parentheses, (and), are used for grouping.

Arithmetic Expressions

E ::= E + T

展開

E → E +T
→ E +T +T
→ E +T +T +T
→ …
→ T +T +T … +T

EBNF

E ::= T { + T }

EBNF for Arithmetic Expressions: (省略 <,>)

BNF-

<E> ::= <E> + <T> | <E> - <T> | <T>

T ::= T * F | T / F | F

F ::= ‘(’ E ’)’ | name | number

EBNF-

F ::= ‘(’ E ’)’ | name | number

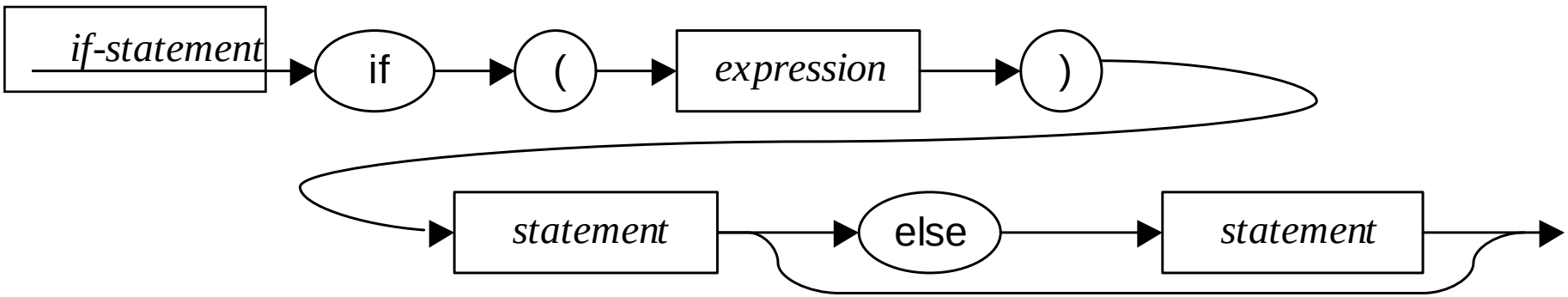
T ::= F { (* | /) F }

E ::= T { (+ | -) T}

Syntax Diagrams (Chart, Graph)

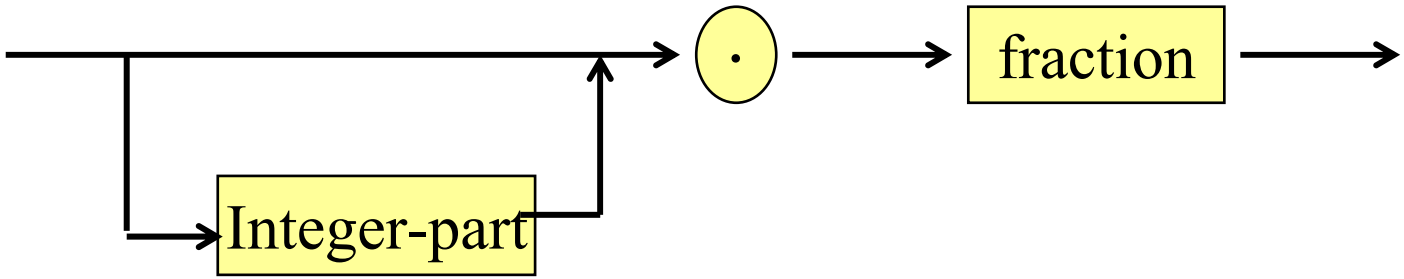
- An alternative to EBNF using cycles.
- Rarely seen any more: EBNF is much more compact.

  nonterminal terminal production rules



Syntax Charts (Graphs, Diagrams)

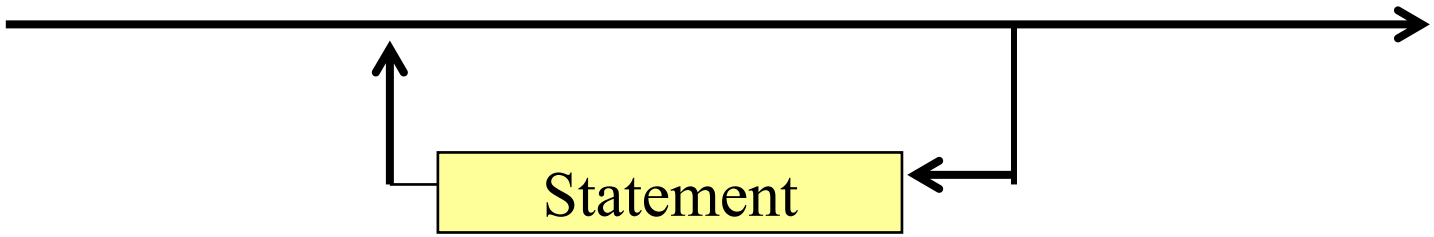
例如：<real_numbers>



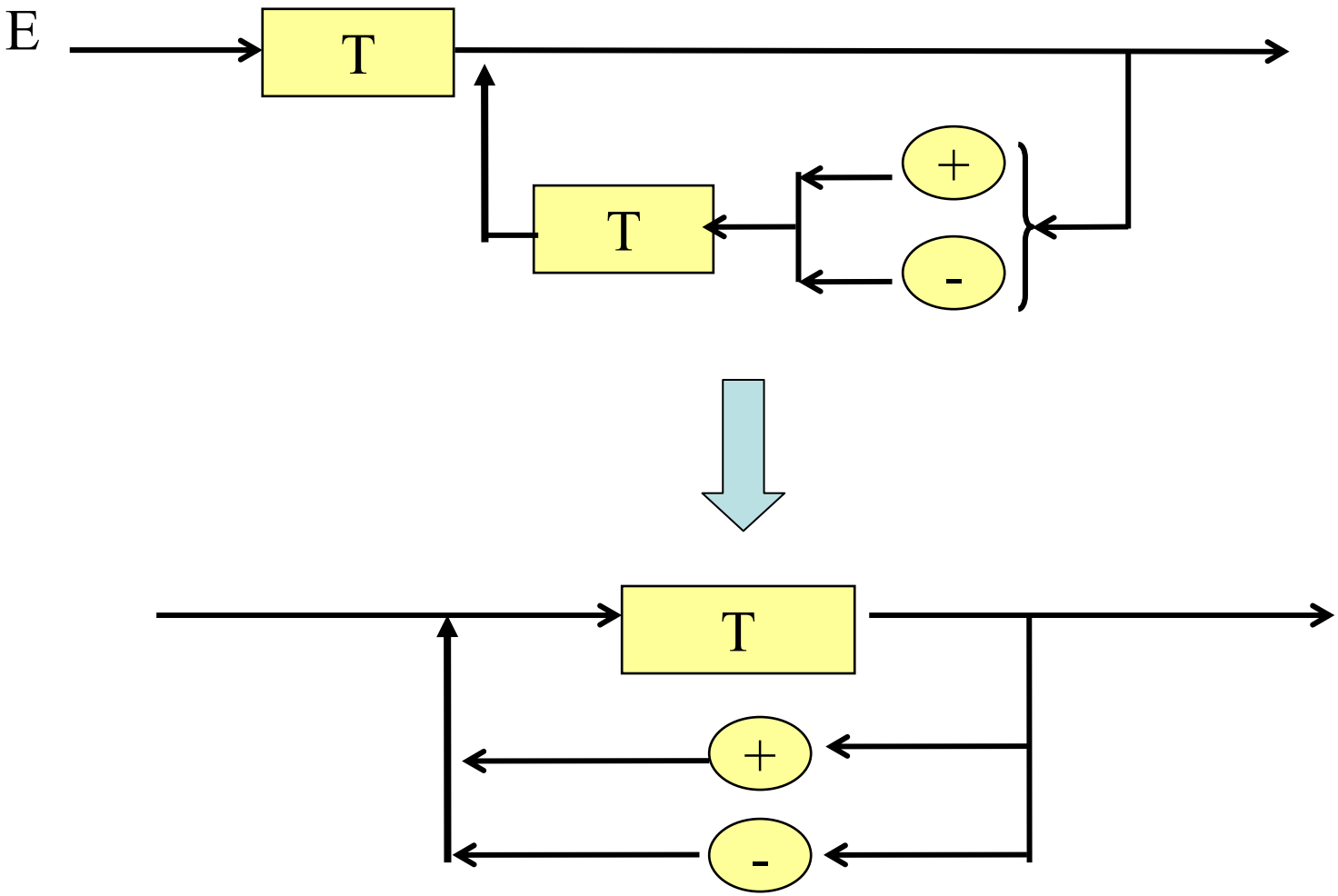
注意箭頭的方向

例如：Statement lists

<statement_list>

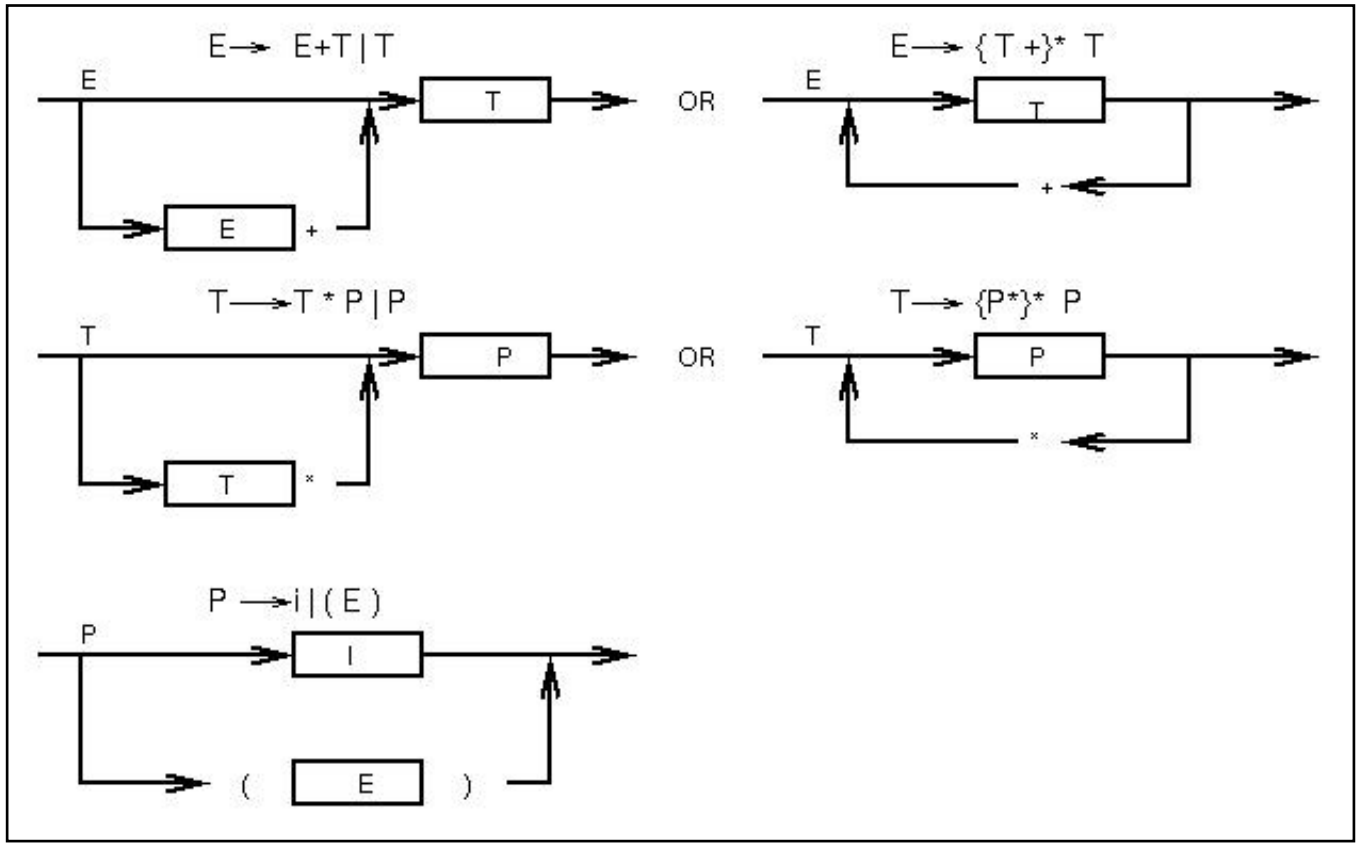


Expressions (*E ::= T { (+|-) T }*)



Syntax Chart Examples

Syntax Charts



Syntactic Idiosyncrasy of C and Java

- Syntax specification for ANSI C:

http://eli-project.sourceforge.net/c_html/c.html

http://www.cs.colorado.edu/~eliuser/c_html/c.html

- Syntax specification for Java:

http://java.sun.com/docs/books/jls/second_edition/html/grammars.doc.html#44271

Basic Syntactic Categories

- Declarations
  - `int i, j, k;`
- Expressions
  - `j + k * f( i)`
- Statements
  - `if (j>k) i =10; else i =5;`
  - `while (i) { …};`

Are Assignments statements?
x = x + 1;

ExpressionStatement

Examples of C & Java Expressions

C (Java) has 15 levels of precedence for its operators!

- **`++a[3]` means `++(a[3])`**
- **`a >> b + 3` means `a >> (b+3)`**
- **`a +++ b` means `(a++) + b`**
- **`a | 4 + c >> b & 7 || b > a % 3`**
  means

`(a | ((4 + c) >> b) & 7)) || (b > a % 3)`

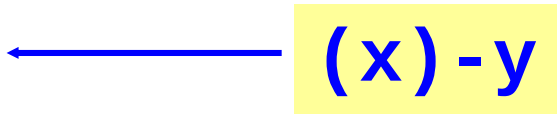
- Is: `I = 1 && 2 + 3 | 4;`

legal in C? What is assigned to I if it is?

Operator Precedence in C

- the higher in the table an operator appears, the higher its precedence

postfix operators	[] . (params) expr++ expr--
unary operators	++expr --expr +expr -expr ~ !
creation or cast	new (type) expr
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof



Operator Precedence in C

equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
conditional	? :
assignment	= += -= *= /= %= ^= &= |= <<= >>= >>>=

```
if (a==b & c==d)…
```

What's Wrong with C's Declaration Syntax

- Unclear: three or one pointers?
  - `int*  first, next, last;`
- Unreadable
  - `int *(*(*(*x)())[10])();`
- Precedence among Function, Pointers, Arrays
  - `double *fn(double);`
  - `double (*fn)(double);`

The flip side of this is that you have to deal with old mistakes and with compatibility problems. For example, I consider the C declarator syntax an experiment that failed.
- Bjarne Stroustrup, in his SlashDot interview on 2/25/2000.

Syntax for C's Declarations

declaration:

decl-specifiers declarator-list$_{opt}$;

decl-specifiers:

type-specifier
sc-specifier
type-specifier sc-specifier
sc-specifier type-specifier

sc-specifier:

auto
static
extern
register

type-specifier:

int
char
float
double
struct { *type-decl-list* }
struct *identifier* { *type-decl-list* }
struct *identifier*

declarator-list:

declarator
declarator , *declarator-list*

declarator:

identifier
* *declarator*
declarator ( )
declarator [*constant-expression*$_{opt}$]
(*declarator*)

type-decl-list:

type-declaration
type-declaration type-decl-list

type-declaration:

type-specifier declarator-list ;

Expressions & Statements in C

- In C (Java) all expressions can act as statements.

<statement> ::= *<expression> ;*
| if statement | …

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

```
int a = 0;
if  (a=1)
   printf(“a is one\n”);
```

Lexical Elements of Language Syntax

Components of Language Syntax

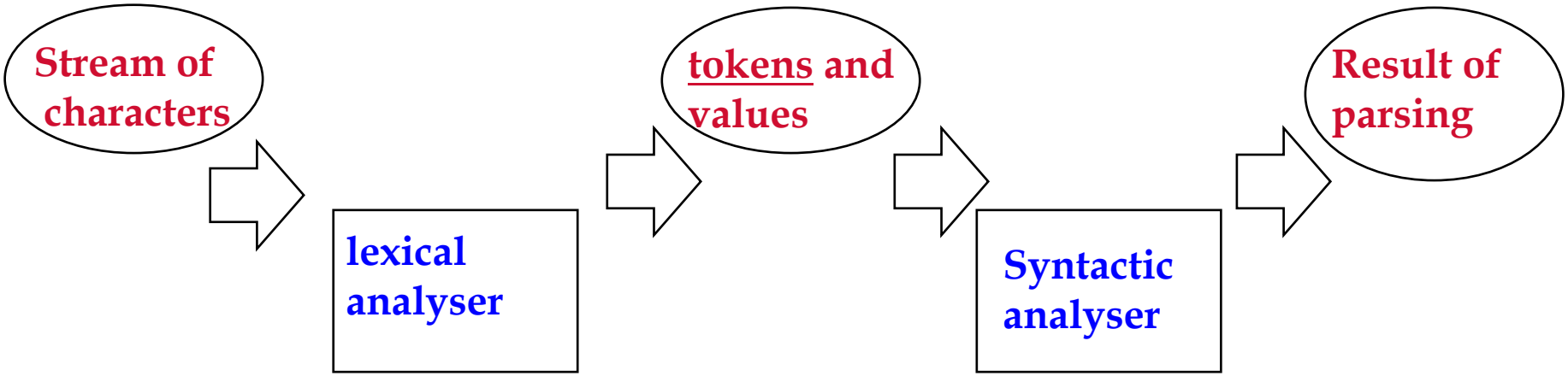
- Alphabet
- Token (Terminal symbols, Word)
  - Identifiers
  - Reserve Words (Keywords)
  - Delimiters

Lexical structure

- Sentence
- Grammar

Syntax Processing of PL's

- Syntax processing is divided into two phases:
- The Scanning phase (lexical analyser) collects characters into tokens (斷字)
- Parsing phase (syntactic analyser) determines syntactic structure (組句)



Tokens

- Tokens (Terminal symbols)
  - smallest “atomic” units of syntax
  - used to build all other constructs
  - E.g., in Pascal:

    A B C…z

    0 1 … 9

    + - * / = <> > < >= <=

    ( ) [ ] ; := . , …

    program begin end if then while…
- Based on alphabet
  - Most PL’s: English Letters, digits, selected symbols
  - Java: include Unicode characters (例如：可取中文變數名字)

Token Categories

- **Identifiers**
  - names of variables, functions, etc.
  - sequence of terminals with restricted form (e.g., must start with letter in Pascal)
- **Reserved words**
  - special identifiers whose use is restricted
  - e.g., Pascal: **if** is reserved

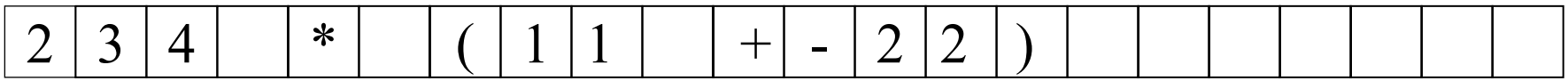
    FORTRAN: “IF” is not reserved **(Keyword only)**
    E.g., IF (IF .LT. 0) IF = IF + 1
  - Positives:
    - enhances readability
    - helps with syntactic error recovery
  - Negatives:
    - must be learned
    - language extensions may invalidate old programs

Tokens Categories

- Literals or constants, such as `42` (a numeric literal) or `"hello"` (a string literal)
- Symbols:
  - Operators, such as “+”, “`-`”, “*”, “<”, “>”, etc.
  - Misc such as “`,`” “`;`” , “(“, “)”, “[“, “]”, etc.
  - Delimiters
    - punctuation in programming languages; “Internal” tokens of the scanner that are matched and discarded
    - white space: newlines, tabs, spaces
    - FORTRAN is an exception:
      - `DO I = 1,5` is beginning of a loop
      - `DO I = 1.5` is *assignment* to variable “DOI”
    - Comments

Lexical Analyzer (Scanner)

Program: a stream of characters (斷字)



Num(234)

mul_op

lpar_op

Num(11)

add_op

Num(-22)

rpar_op

Lexical Conventions

- *Principle of longest substring:* If the input stream has been separated into tokens up to a given character, the next token is the *longest* string of characters that could constitute a token.
- Examples:
  - "`ifelse`" is an identifier, not two keywords.
  - `i = j+++k;`
- Comments are generally not nested.
- Multi-character token
  - `x = y/*p;`

Defining Tokens Using **Regular Expressions**

	RE	**Language**
Terminal symbol **a**	**a**	{ **a** }
Empty string ε	ε	{ ε }
Union	**R | S**	$L(R) \cup L(S)$
Concatenation	**RS**	$\{ rs \mid r \in L(R) \wedge s \in L(S) \}$
Non-empty closure	$\mathbf{R^+}$	$R \cup RR \cup RRR \cup \ldots$ (1 or more Rs concatenated)
(Kleene) closure	**R***	$\{ \varepsilon \} \cup R \cup RR \cup RRR \cup \ldots$ ($R^* = R^+ \cup \{ \varepsilon \}$)
Grouping	**(S)**	L(S)

all RE operators *left-associative*, shown in order of increasing precedence

Regular Expression Examples

RE	Language over an alphabet $\sum$
a | bc	{ **a, bc** }
(a | b)c	{ **ac, bc** }
a ε	{ **a** }
a^* | b	{ ε , **a, aa, aaa, aaaa, …**} $\cup$ {**b**}
ab^*	{**a, ab, abb, abbb, abbbb, …**}
ab^* | c^+	{**a, ab, abb, abbb, abbbb, …**} $\cup$ {**c, cc, ccc, …**}
(a | b)*	{ ε , **a, b, aa, ab, ba, bb, aaa, aab, …**}
(0 | 1)* 1	binary numbers ending in 1 (i.e., the odd binary numbers)

Examples of RE

- Keywords: “else” or “if” or “while” or …
  - `else` + `if` + `while` + …
  - `else` abbreviates `e` `l` `s` `e`
  - keywords = { `else`, `if`, `then`, `while`, … }

- Integer:
  - digit = `0`+`1`+`2`+`3`+`4`+`5`+`6`+`7`+`8`+`9`
  - integer = *digit digit**
    - is `000` an integer?

Examples of RE (cont.)

- Identifier: strings of letters or digits, starting with a letter
  - letter = `A` + … + `Z` + `a` + … + `z`
  - Identifier = *letter (letter + digit)**
    - Is (letter* + digit*) the same ?

- Whitespace: a non-empty sequence of blanks, newlines and tabs
  - (` `+`\n`+`\t`)$^+$

Examples of RE Around

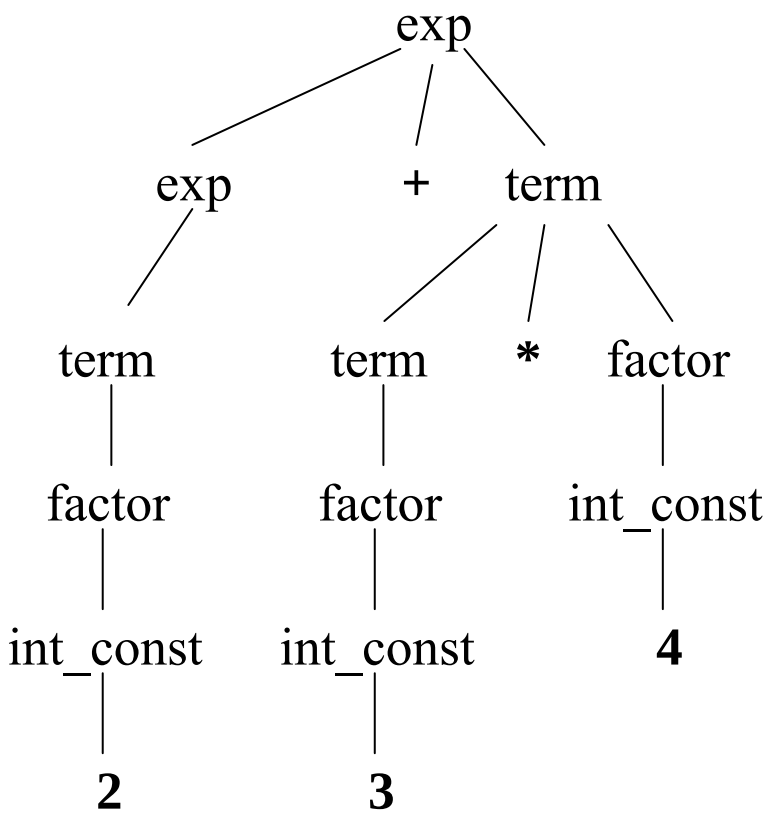
- Phones number: consider (02) 883-0000
  - $\sum$ = digit $\cup$ { -, (,) }
  - exchange = $digit^3$
  - phone = $digit^4$
  - area = $digit^2$
  - phone_number = '(' area ')' exchange '-' phone

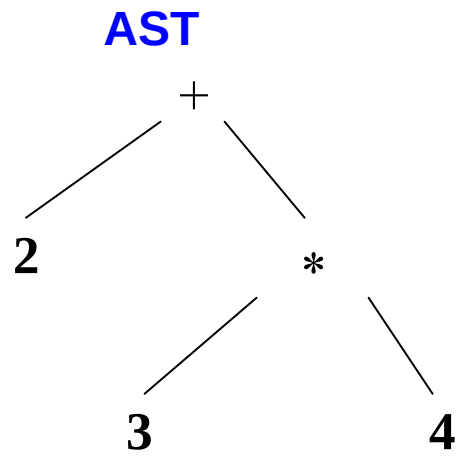
- Email address: student@nccu.edu.tw
  - $\sum$ = letter $\cup$ { . , @ }
  - name = $letter^+$
  - address = name`@`name`.`name `.` name

Abstract Syntax Trees (AST)

5. Abstract vs. Concrete Syntax

- Concrete = What is written down.
- Abstract Syntax identifies the meaningful components of a language construct, independent of its actual form (Concrete Syntax).





Abstract vs. Concrete Syntax

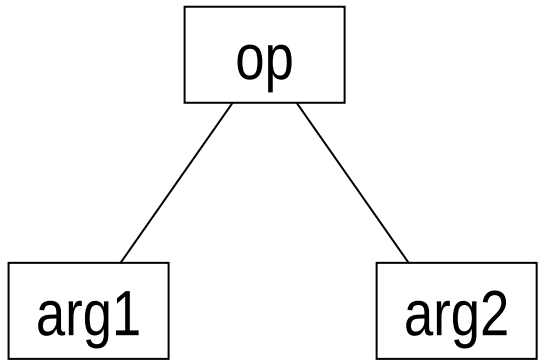
- Concrete Syntax – what is written down

1. a + b
2. + a b
3. a b +

- Abstract Syntax – meaningful components

All have 3 meaningful components: *an operator, two operands.*

Abstract Syntax Tree:



AST example

- expression grammar

  expression → expression ‘+’ term | expression ‘-’ term | term

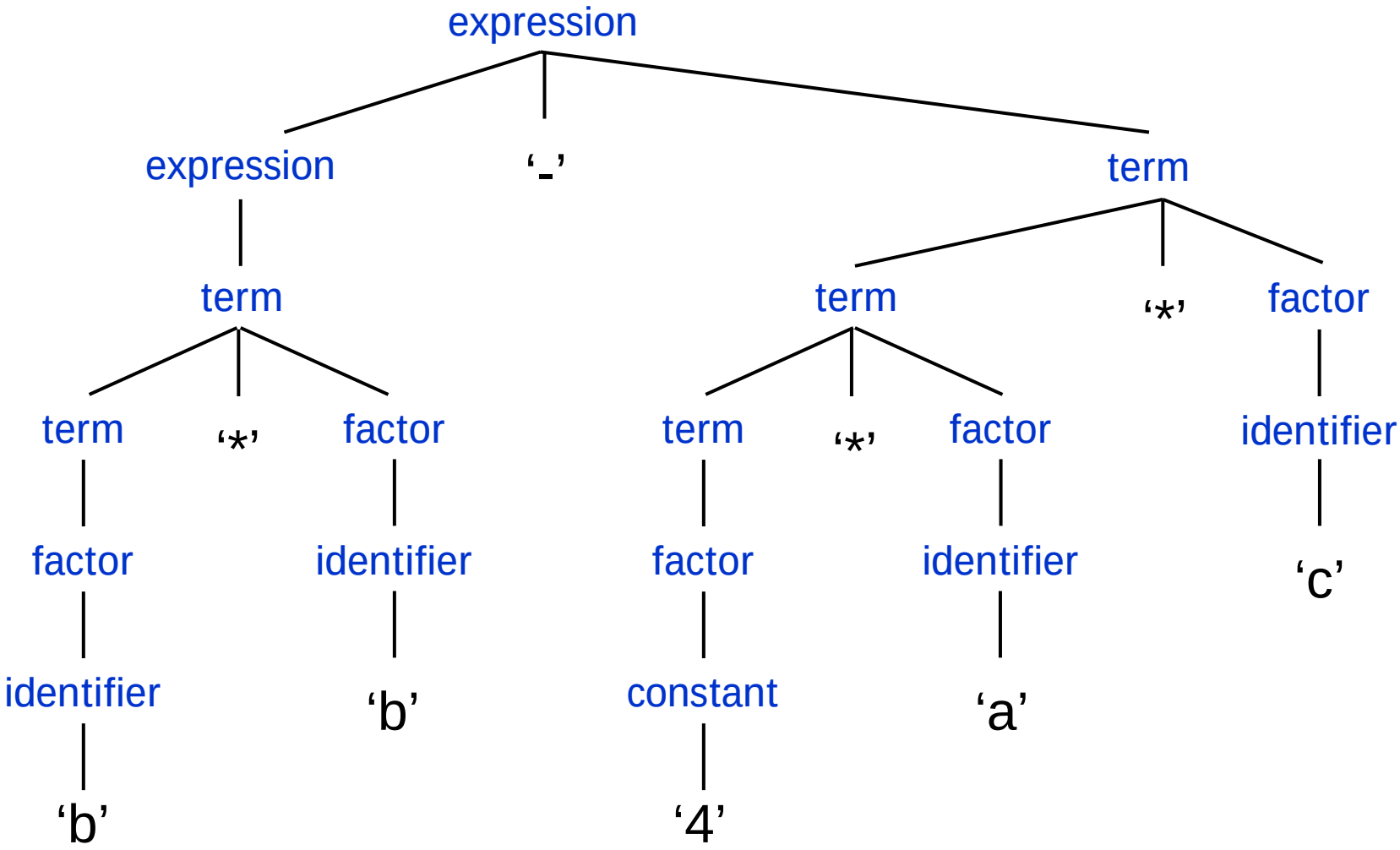
  term → term ‘*’ factor | term ‘/’ factor | factor

  factor → identifier | constant | ‘(‘ expression ‘)’

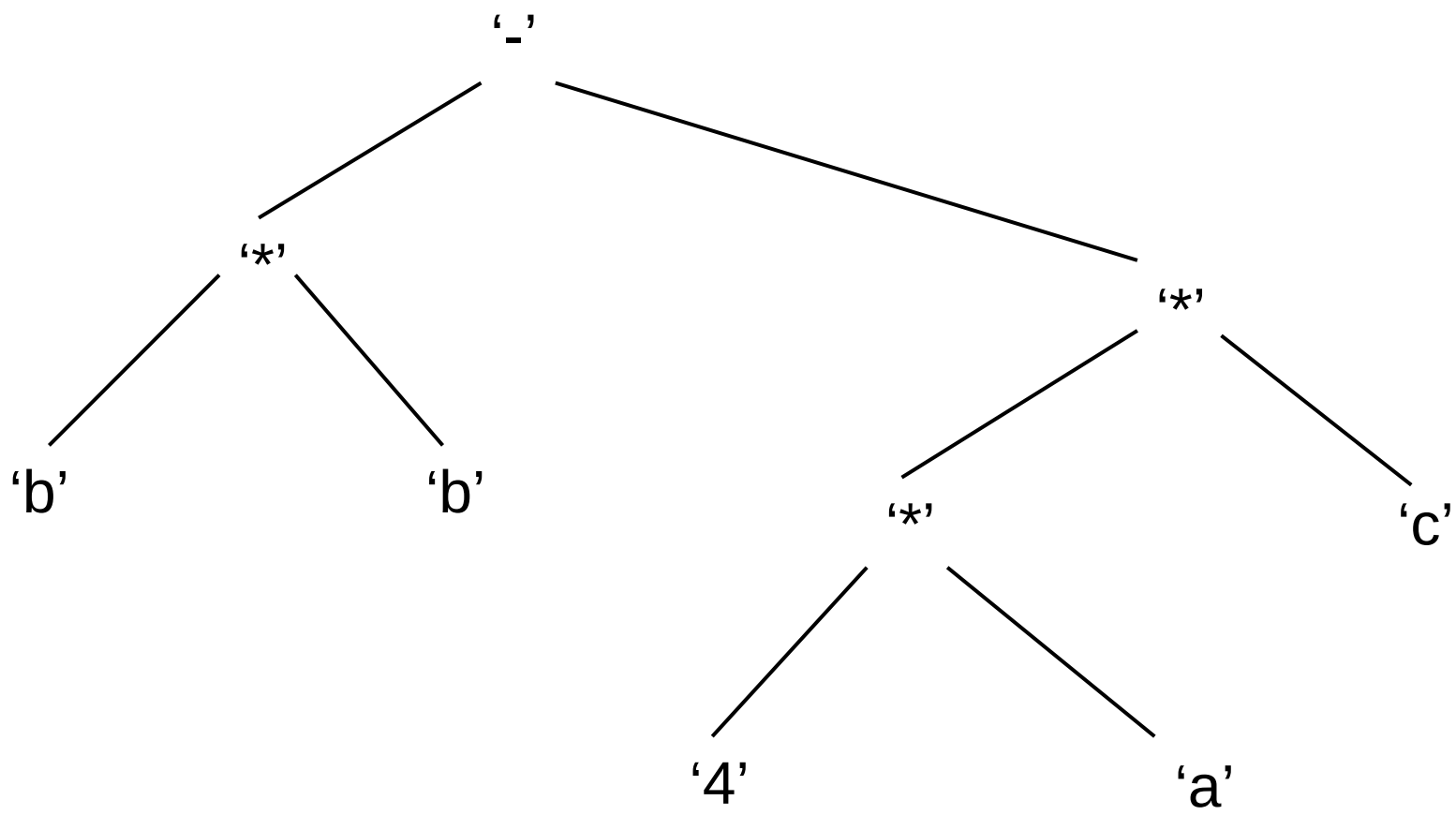
- example expression

  b*b – 4*a*c

Parse tree: b*b – 4*a*c



AST: b*b – 4*a*c



Abstract Syntax Tree

- Rationale
  - Not all the information in the parse tree is needed for translation
- Abstract Syntax Tree (AST)
  - condensed parse tree
  - nonterminal nodes are removed and some terminal nodes become internal nodes

Concrete vs. abstract syntax

grammar for concrete syntax

```
exp ::= term
      | exp + term
      | exp - term
term ::= factor
       | term * factor
       | term / factor
factor ::= ( exp )
         | int_const
         | fun_name args
```

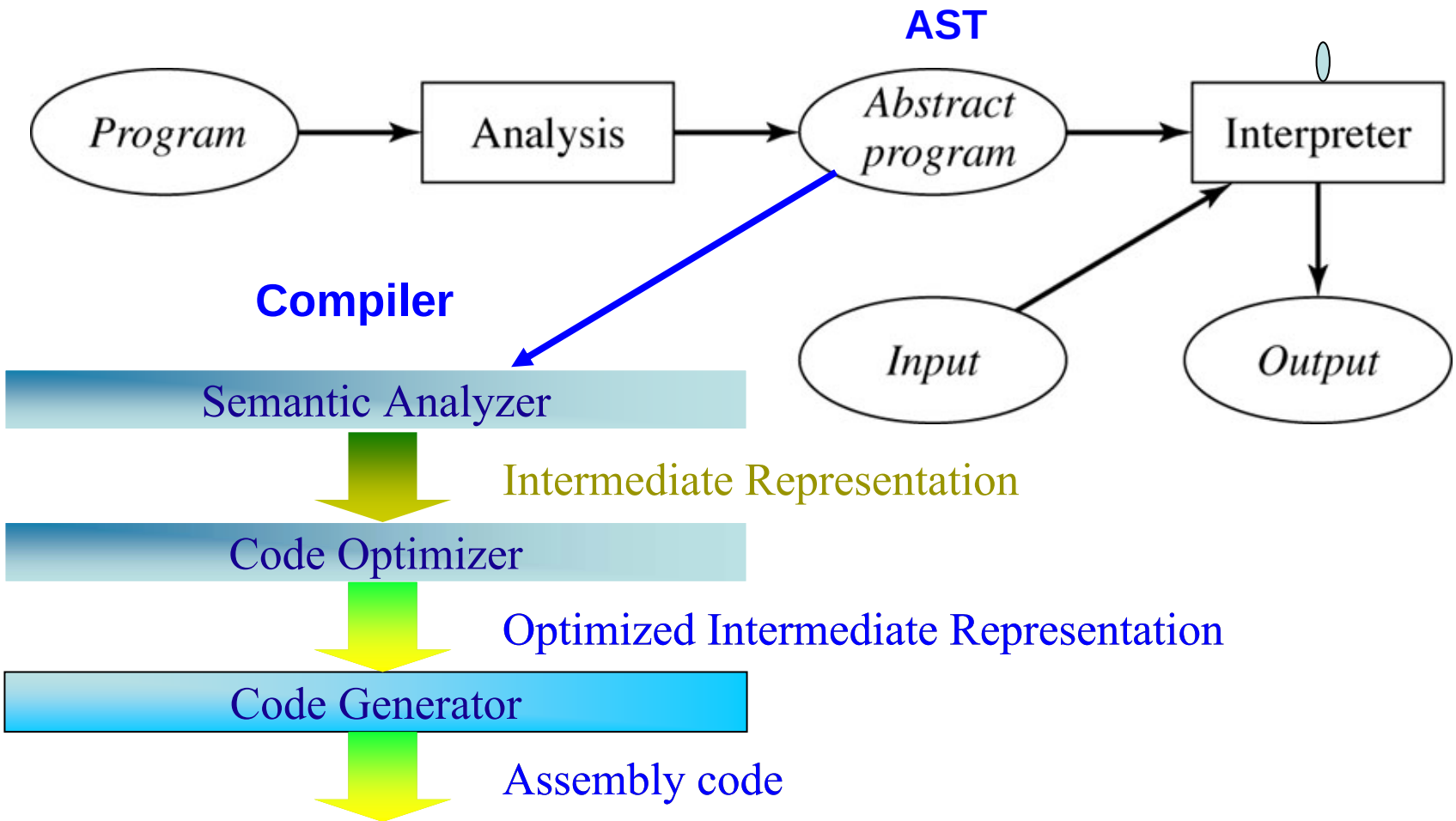
grammar for abstract syntax

```
e = + | e e | - | e e
  | * | e e | / | e e
  | i
  | Apply
      funName  args
```

Ignore the precedence etc. issues

AST for Interpreter/Compiler

- AST is the main data structure built and used by interpreters and compilers



Design and Specification of a PL

- 設計一程式語言時要面對許多課題
  - 這語言所寫的程式的外表 (surface syntax)
  - 語法結構：嚴格界定一個程式語言所包含的句子或字串
  - 語意內涵
  - 語法結構與語意內涵的關聯性 *(Notation as a Tool for Thought)*
  - Feature Interaction
  - Implementability: Compiler/Interpreter以及其複雜度
  - …
- 本單元著重在語法構造上

Write-Only Languages?

Notation as a Tool for Thought,

--K. Iverson's Turing Award Lecture

- APL (矩陣運算)

```
(.tilde N .contains N .circle . .product N)/N .leftarrow 1 .downarrow .iota N
```

- Perl (字串處理)

```
perl  -le  '$_ = 1; (1 x $_) !~ /^(11+)\1+$/  &&  print while $_++'
```