Lecture 2 : Lexical Analysis

THE ROLE OF LEXICAL ANALYZER

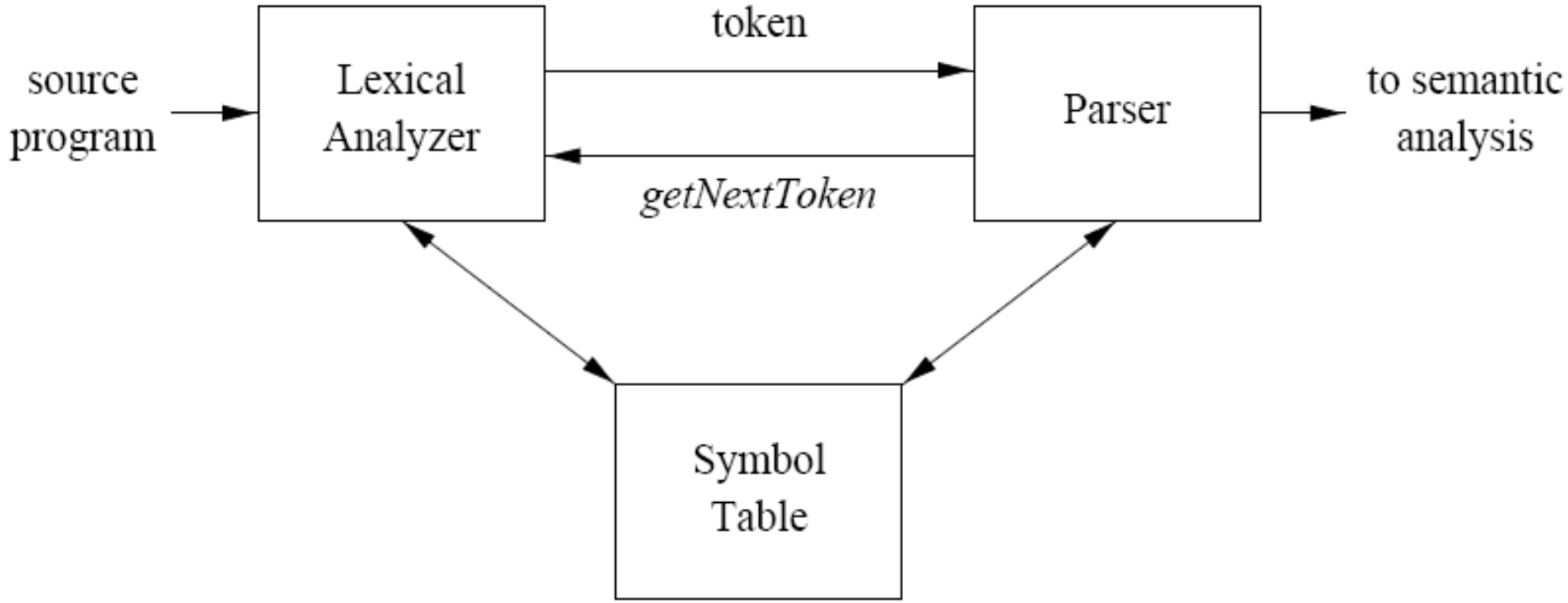

Figure 3.1: Interactions between the lexical analyzer and the parser

LEXICAL ANALYSIS

- Scan source program as a file of characters
- Divide character stream ➔ tokens.
  - A sequence of characters ➔ token
  - Ex: keyword, identifier, special symbol

Code

```
If x == y
   z = 1;
else
   z = 2;
```

A file of characters

/t If x==y /n /t z = 1;/n else /n/t z=2;/n

LEXICAL ANALYSIS

- Scan source program as a file of characters
- Divide character stream ➔ tokens.
  - A sequence of characters ➔ token
  - Ex: keyword, identifier, special symbol

Code

```
If x == y
   z = 1;
else
   z = 2;
```

A file of characters

/t If x==y /n /t z = 1;/n else /n/t z=2;/n

HOW TO DIVIDE A TOKEN?

- Use diagram to specify token.
- A special case of pattern matching
  - Pattern specification ➔ regular expression so it will be easy to modify or implement
- Regular expression ➔ NFA ➔ DFA

TOKEN, PATTERNS , LEXEMES

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

Figure 3.2: Examples of tokens

ตัวอย่าง

```
printf("Total=%d\n",score);
```

- printf และ score เป็น lexeme ที่ตรงกับรูปแบบของ token "id"
- "Total=%d\n" เป็น lexeme ที่ตรงกับรูปแบบของ token "literal"

INPUT BUFFERING

2 buffers + 2 pointers

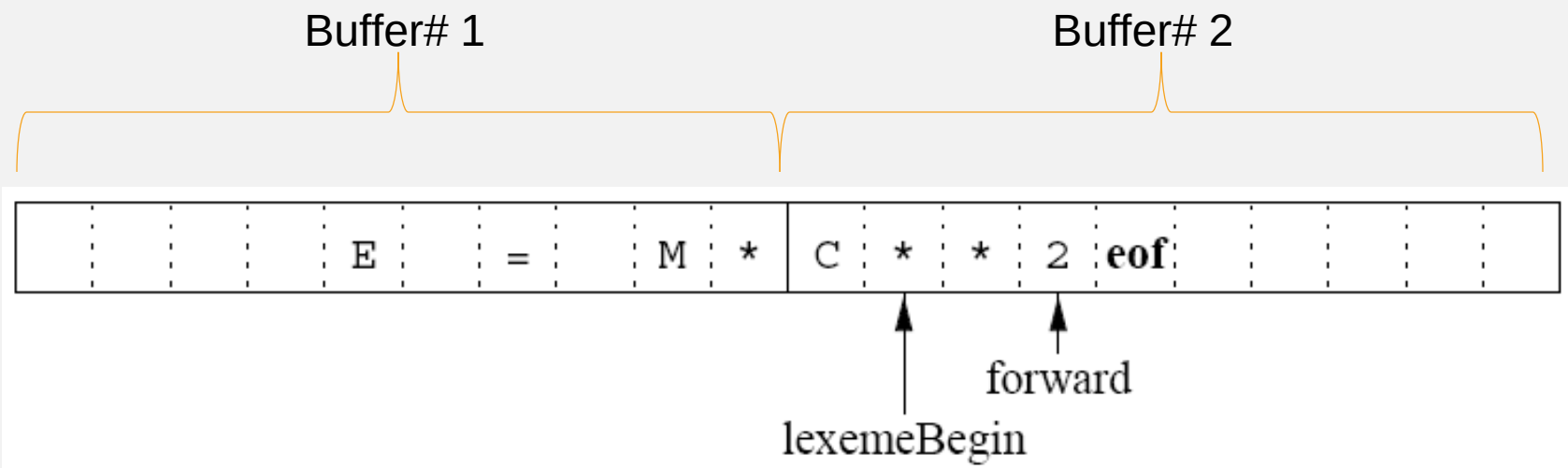

Figure 3.3: Using a pair of input buffers

SENTINEL

เครื่องหมายหรือสัญลักษณ์ ที่แสดงการเริ่มต้นหรือยุติลง.

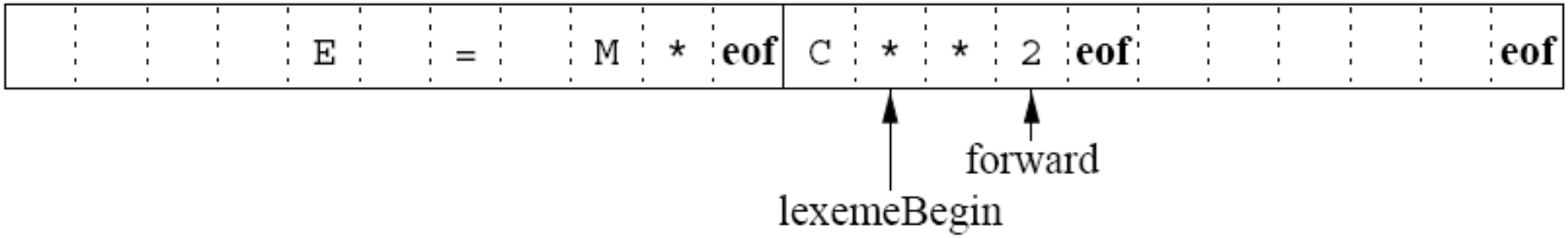

Figure 3.4: Sentinels at the end of each buffer

```
switch ( *forward++ ) {
        case eof:
                if (forward is at end of first buffer ) {
                        reload second buffer;
                        forward = beginning of second buffer;
                }
                else if (forward is at end of second buffer ) {
                        reload first buffer;
                        forward = beginning of first buffer;
                }
                else /* eof within a buffer marks the end of input */
                        terminate lexical analysis;
                break;
        Cases for the other characters
}
```

Figure 3.5: Lookahead code with sentinels

ATTRIBUTES FOR TOKENS

```
E = M * C ** 2
```

- สามารถเขียนเป็นคู่ลำดับของ **token name** กับ **attribute values**

<id, pointer to symbol-table entry for E>

<assign_op>

<id, pointer to symbol-table entry for M>

<mult_op>

<id, pointer to symbol-table entry for C>

<exp_op>

<number, integer value 2>

EXERCISE 1

- ลองตัดคำโปรแกรมภาษา C++ นี้

```
float limitedSquare(x) float x {
  /* return x-squared, but never >= 100 */
        return (x<=-10.0||x>=10.0)?100:x*x;
}
```

EXERCISE 2

- ลองเสนอวิธีการตัดคำ ภาษา HTML

Here is a photo of <B>my house</B>:

<P><Img Src = “house.gif”>

See <A HREF = “morePix.html”>More Pictures</A>

if you liked that one. <P>

LEXICAL ERRORS

- Some errors are hard for a lexical analyzer to recognize:

  fi (a == f(x)) …

- Lexical analyzer can not tell whether fi is a misspelling of the keyword "if" or an undeclared function identifier

ERROR RECOVERY

- Panic mode: delete successive characters until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Token Recognition Example

- Whitespace is ignored in Fortran
- indexed do loops fixed-format in Fortran90

DO label <*DO-var*> = <*expr1*>,<*expr2*> [,<*expr3* >]

< *exec-stmts* >

END DO

TOKEN RECOGNITION EXAMPLE

- The statement in Fortran

DO 5 I = 1.25 : lexeme “DO5I” is not recognized until we see the dot following the1.

DO 5 I = 1,25 : lexeme “DO” is not recognized until we see the comma(instead of dot) following the1

SPECIFICATION OF TOKEN

- เครื่องมือที่ใช้ในการกำหนดรูปแบบของ **lexeme** หรือ **token** คือ Regular expression(นิพจน์ปกติ)
- Regular แปลว่า ปกติ, ธรรมดา, สามัญ, เป็นประจำ, สม่ำเสมอ, เป็นกิจวัตร, ตามกฎ, **มีกฎเกณฑ์**, มีระเบียบ, ตามระเบียบ, ตามแบบแผน, ถูกต้องตามกฎเกณฑ์
- Regular expression ก็คือ รูปแบบของข้อความที่กำหนดขึ้นมาเพื่อใช้ในการอธิบายภาษา หรือการเปรียบเทียบกับข้อความที่ต้องการตรวจสอบ

คำนิยาม

- ชุดตัวอักษร(alphabet) คือ เซตของสัญลักษณ์ เขียนแทนด้วย Σ เช่น
  - สัญลักษณ์ประกอบด้วย ตัวหนังสือ, ตัวเลข และเครื่องหมายต่างๆ
  - เซตตัวอักษรเลขฐานสอง = {0,1}
- สตริง(String) คือสายอักขระ หรือชุดของตัวอักษรที่เรียงต่อกัน
  ในบางทีอาจเรียกว่า **word**(คำ)
- กำหนดให้ **s** เป็นสตริง
  จะได้ว่า **|s|** คือ ความยาวของสตริง

  เช่น **banana** คือสตริงที่ยาว **6** ตัวอักษร
- สตริงว่าง(Empty string) หรือสายอักขระว่าง
  เขียนแทนด้วย ε (epsilon)

LANGUAGE(ภาษา)

- เซตที่มีจำนวนนับได้ของสตริงบนชุดของตัวอักษร(**alphabet**) ที่กำหนด
- ตัวดำเนินการของภาษา(**operations on language**)
  - **Union** การรวมสตริงหรือสมาชิกเข้าด้วยกัน
  - **Concatenate** การนำสตริงหรือสมาชิกมาต่อกัน
  - **Kleene closure**(**L***) คือการนำสตริงมาต่อกันตั้งแต่ 0 ตัวขึ้นไป (**zero or more**)
  - **Positive closure**(L^+) คือการนำสตริงมาต่อกันตั้งแต่ **1** ตัวขึ้นไป (**one or more**)

หมายเหตุ

closure หมายถึง คุณสมบัติปิด

เซต **A** มีสมบัติการปิดภายใต้โอเปอเรชัน * ใด

ถ้า **a, b** เป็นสมาชิกใน **A** แล้ว สมาชิกที่เกิดขึ้นใหม่จาก **a * b** จะต้องเป็นสมาชิกใน **A** ด้วย

OPERATIONS

Operation	Definition and Notation
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

Figure 3.6: Definitions of operations on languages

- ตัวอย่างของการ **concatenate** เช่น
  ให้ **x** และ **y** เป็นสตริง โดยที่ **x=dog , y=house**

  จะได้ว่า **xy = doghouse**
- เมื่อนำสตริงว่างไปต่อกับ สตริงใดๆ จะได้สตริงนั้นๆ
  เช่น $\varepsilon X = X\,\varepsilon = X$ เป็นต้น และ $|\varepsilon| = 0$

CONCATENATE

- ถ้าเปรียบ **concatenate** กับ การคูณ(**product**) ทางคณิตศาสตร์ เราสามารถเขียน **exponential** แสดงถึงการคูณกันได้
- s^0 หมายถึง ε
- $s^1 = s$
- $s^2 = ss$
- $s^3 = sss$
- เมื่อ $i > 0$ จะได้ว่า

$$s^i = s^{i-1}s$$

อะไรคือ regular expressions?

- ในคณิตศาสตร์ เราใช้ตัวดำเนินต่างๆ เช่น + และ × ในการสร้างนิพจน์ (expression) ทางคณิตศาสตร์ เช่น

$$(5 + 3) \times 4$$

- ในทำนองเดียวกัน เราสามารถใช้ตัวดำเนินปกติ (regular operations) ในการสร้างนิพจน์เพื่ออธิบายภาษาได้ ซึ่งเรียกว่า regular expressions ตัวอย่างเช่น

$$(0 \cup 1)\, 0^*$$

From Dr.Suthee's Lecture notes

อะไรคือ regular expressions?

- ค่าของ regular expression คือภาษา ภาษาหนึ่ง

$$(0 \cup 1)\ 0^*$$

- สัญลักษณ์ 0 และ 1 ย่อมาจากเซต {0} และ {1}.
  - $(0 \cup 1) = (\{0\} \cup \{1\}) = \{0,1\}$
  - $0^* = \{0\}^* = \{\varepsilon, 0, 00, 000, \ldots\}$
- เครื่องที่ใช้แทนตัวดำเนิน concatenation $\circ$ มักจะถูกละเอาไว้ใน regular expressions
  - $(0 \cup 1)\ 0^* = (0 \cup 1) \circ 0^*$

From Dr.Suthee's Lecture notes

ตัวอย่าง

$$(0 \cup 1)^*$$

- สำหรับค่าของ RE นี้คือภาษาที่ประกอบด้วยสตริงทุกตัวที่มี 0 หรือ 1
- ถ้า $\Sigma = \{0,1\}$, เราสามารถใช้ Σ เขียนย่อสำหรับ $(0 \cup 1)$ ได้ ดังนั้น $(0 \cup 1)^* = \Sigma^*$
- โดยทั่วไป Σ^* อธิบายภาษาที่ประกอบด้วยสตริงทั้งหมดสำหรับ alphabet นั้น เช่น
  - Σ^*1 คือภาษาที่ประกอบสตริงทั้งหมดที่ลงท้ายด้วย 1
  - $(0\ \Sigma^*) \cup (\Sigma^*1)$ คือภาษาที่ประกอบด้วยสตริงที่ขึ้นต้นด้วย 0 หรือสตริงที่ลงท้ายด้วย 1

From Dr.Suthee's Lecture notes

ลำดับความสำคัญของตัวดำเนินการ

- Unary operator * มีลำดับความสำคัญสูงสุด และเป็น left associative
- Concatenation มีลำดับความสำคัญรองลงมา และเป็น left associative
- Union มีลำดับความสำคัญรองลงมา และเป็น left associative

หมายเหตุ เครื่องหมาย | (เส้นตรง) ใช้แทน เครื่องหมาย union ได้

นิยามอย่างเป็นทางการของ RE

Say that R is a regular expression if R is

1. *a* for some *a* in the alphabet Σ,
2. ε, (the empty string)
3. $\emptyset$, (the language that doesn't contain any strings)
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expression.
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expression.
6. (R_1^*), where R_1 is a regular expression.

From Dr.Suthee's Lecture notes

ตัวอย่าง

ให้ $\Sigma = \{a, b\}$

1. Regular expression a|b แสดงถึงภาษา {a,b}
2. (a|b)(a|b) แสดงถึง {aa,ab,ba,bb}
   คือภาษาที่สตริงยาว 2 ตัวอักษร บนเซตของตัวอักษร Σ
   เขียน Regular expression ได้อีกแบบว่า aa|ab|ba|bb
3. a* แสดงถึงภาษาที่ประกอบด้วยสตริงของตัวอักษร a
   ที่มีความยาวตั้งแต่ 0 ตัวอักษรขึ้นไป
   เขียนได้ดังนี้ {ε, a, aa,aaa,…}
4. (a|b)* แสดงถึงเซตของสตริงที่ประกอบด้วย ตัว a หรือ b
   ที่มีความยาวตั้งแต่ 0 ตัวอักษรขึ้นไป
   เขียนได้ดังนี้ {ε, a, b,aa,ab, ba, bb, aaa,…}
   เขียน Regular expression ได้อีกแบบว่า (a*b*)*
5. a|a*b แสดงถึงภาษา {a,b,ab,aab,aaab,…}
   นั่นคือ สตริง a และสตริงที่ประกอบด้วยตัว a ตั้งแต่ 0 ตัวขึ้นไปลงท้ายด้วย b

แบบฝึกหัด

- จงเขียน RE สำหรับอธิบายภาษาดังต่อไปนี้

โดยที่ $\sum = \{a,b\}$

- L = $\{a^n$: n เป็นผลคูณของ 3 หรือ เป็นผลคูณของ 5$\}$
- L = $\{a^n$: n ≥ 0, n ≠ 4$\}$
- L = $\{ab^5wb^2: w \in \{a, b\}^*\}$
- L = $\{w : |w|$ mod 3 = 0$\}$
- L = $\{w : n_a(w)$ mod 3 = 0$\}$

From Dr.Suthee's Lecture notes

กฎพีชคณิตสำหรับ REGULAR EXPRESSION

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

Figure 3.7: Algebraic laws for regular expressions

REGULAR DEFINITIONS

- เพื่อความสะดวกในการใช้งาน เราจะมีการกำหนดชื่อให้กับ **expression**
- **Regular definition** คือ ชุดลำดับของคำนิยามของ **regular expression** ที่ใช้ใน **grammar** นี้ มีรูปแบบดังข้างล่างนี้

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\ldots\ldots$$
$$d_n \rightarrow r_n$$

โดยที่

d_i เป็นสัญลักษณ์ใหม่ที่ไม่มีอยู่ใน Σ และ d_i จะมีชื่อไม่ซ้ำกัน

r_i เป็น **regular expression** บน

$$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$$

ตัวอย่าง C IDENTIFIER

letter_ → A|B|…|Z|a|b|…|z|_

digit → 0 | 1 |… | 9

id → letter_ (letter_ | digit)*

ตัวอย่าง unsigned numbers

digit → 0 | 1 |… | 9

digits → digit digit*

optionalFraction → . digits | ε

optionalExponent → (E(+|-) digits | ε

Number → digits optionalFraction optionalExponent

ตัวอย่างเช่น 5280, 0.01234, 6.336E4, หรือ 1.89E-4

EXERCISE

เขียน regular definitions สำหรับภาษาต่างๆ ดังนี้

- All strings of lowercase letters that contain the 5 vowels in order.
- Comments, consisting of a string surrounded by /* and */ โดยไม่มี */ อยู่ระหว่างกลาง

EXTENSIONS OF REGULAR EXPRESSIONS

- One or more instances: เครื่องหมาย + เป็น unary, postfix operator แสดง positive closure
  มีลำดับความสำคัญและการจัดหมู่(precedence and associativity)เหมือนกับ operator *
  สำหรับ r ที่เป็น regular expression
  $(r)^+$ แสดงถึงภาษา $(L(r))^+$
  มีกฎ algebraic law เพิ่มเติมดังนี้

$$r^* = r^+ \mid \varepsilon$$
$$r^+ = rr^* = r^*r$$

EXTENSIONS OF REGULAR EXPRESSIONS

- **Zero or one** instance: เครื่องหมาย **?** เป็น **unary, postfix operator** หมายถึง **regular expression** ส่วนนี้ปรากฏ 0 หรือ 1 ครั้ง, นั่นคือ เป็นทางเลือกว่าส่วนนี้จะมีหรือไม่ก็ได้
- มีลำดับความสำคัญและการจัดหมู่(**precedence and associativity**) เหมือนกับ **operator** * และ +
- r? = r| ε
- L(r?) = L(r) ∪ {ε}

EXTENSIONS OF REGULAR EXPRESSIONS

- **Character classes**:
  ใช้สัญลักษณ์ $[a_1 a_2 ... a_n]$ แทน $a_1 | a_2 | \cdots | a_n$
- ในกรณีที่ $a_1, a_2, \cdots, a_n$ เป็นข้อมูลที่ต่อเนื่องกันเช่น ตัวอักษรตัวใหญ่ ตัวเล็ก, ตัวเลข เราสามารถเขียนแทนในรูปแบบ $a_1 - a_n$
- ดังนั้น [abc] ใช้แทน a|b|c
  และ [a-z] ใช้แทน a|b|…|z

ตัวอย่าง C identifier

letter_ → [A-Za-z_]

digit → [0-9]

id → letter_ (letter_ | digit)*

ตัวอย่าง unsigned numbers

digit → [0-9]

digits → $digit^+$

Number → digits (. digits)? ((E(+|-) digits)?

EXERCISES

ให้เขียน **character classes** สำหรับเซตของตัวอักษรต่อไปนี้

- The first 10 letters (up to “j”) in either upper or lower case
- The “digits” in a hexadecimal number (choose either upper or lower case for the digits above 9).

LEX

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
$\backslash c$	character c literally	`\*`
`"`s`"`	string s literally	`"**"`
`.`	any character but newline	`a.*b`
`^`	beginning of a line	`^abc`
`$`	end of a line	`abc$`
$[s]$	any one of the characters in string s	`[abc]`
$[\hat{}s]$	any one character not in string s	`[^abc]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one r	`a?`
$r\{m,n\}$	between m and n occurrences of r	`a{1,5}`
$r_1 r_2$	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

Figure 3.8: Lex regular expressions

3.4 RECOGNITION OF TOKENS

- How to take "Pattern" to examines the input string
- How to find a prefix that is a lexeme matching one of the patterns

ตัวอย่าง

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	ϵ
expr	→	*term* **relop** *term*
	|	*term*
term	→	**id**
	|	**number**

Figure 3.10: A grammar for branching statements

digit	→	`[0-9]`
digits	→	$digit^+$
number	→	*digits* (`.` *digits*)? (`E` `[+-]`? *digits*)?
letter	→	`[A-Za-z]`
id	→	*letter* (*letter* | *digit*)*
if	→	`if`
then	→	`then`
else	→	`else`
relop	→	`<` | `>` | `<=` | `>=` | `=` | `<>`

Figure 3.11: Patterns for tokens of Example 3.8

Regular definitions

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Figure 3.12: Tokens, their patterns, and attribute values

3.4.1 TRANSITION DIAGRAMS

- เราจะทำการแปลง **regular expression** ให้อยู่ในรูปของ **transition diagram**
- **Transition diagram** ประกอบด้วย
  - **States** สัญลักษณ์วงกลม หรือ **node** แสดงถึง**สถานะ**เมื่ออ่าน **input** แต่ละตัวเข้ามา
  - **Edges** สัญลักษณ์ลูกศร เป็นเส้นเชื่อมจากสถานะ(วงกลม)หนึ่งไปยังอีกสถานะหนึ่ง
- การทำงานของ **transition diagram** จะอยู่บนสมมติฐานว่า ทุกๆ **transition diagram** เป็น **deterministic** นั่นคือ สำหรับสัญลักษณ์ **input** 1 ตัว จะมีเพียง 1 **edge**(เส้นเชื่อม) ไปยัง **state**(สถานะ)ถัดไปได้เพียง 1 สถานะเท่านั้น
- If we are in some **states s** and the next **input** symbol is **a**, we look for an **edge** out of state s labeled by a. If we find such an **edge**, we advance the ***forward pointer*** and enter the **state** of the transition diagram to which that **edge** leads.

สิ่งสำคัญสำหรับ TRANSITION DIAGRAM

- จะต้องมี **accepting** หรือ **final state** ซึ่งเป็นวงกลมซ้อนกัน 2 วง เป็นตัวบอกว่าเราพบ **lexeme** แล้ว
- ถ้าเกิดกรณีต้องย้อน **forward pointer** คืนไป 1 ตำแหน่ง นั่นคือ **lexeme** จะไม่นับตัวอักขระตัวล่าสุดที่ทำให้เราไปยัง **accepting state** เราจะเขียนเครื่องหมายดอกจัน * ที่ **accepting state** เพื่อแสดงว่าเกิดกรณีนี้ขึ้น
- **Starting** หรือ **initial state** จะมีป้าย **start** ชี้เข้ามา

ตัวอย่าง TOKEN RELOP

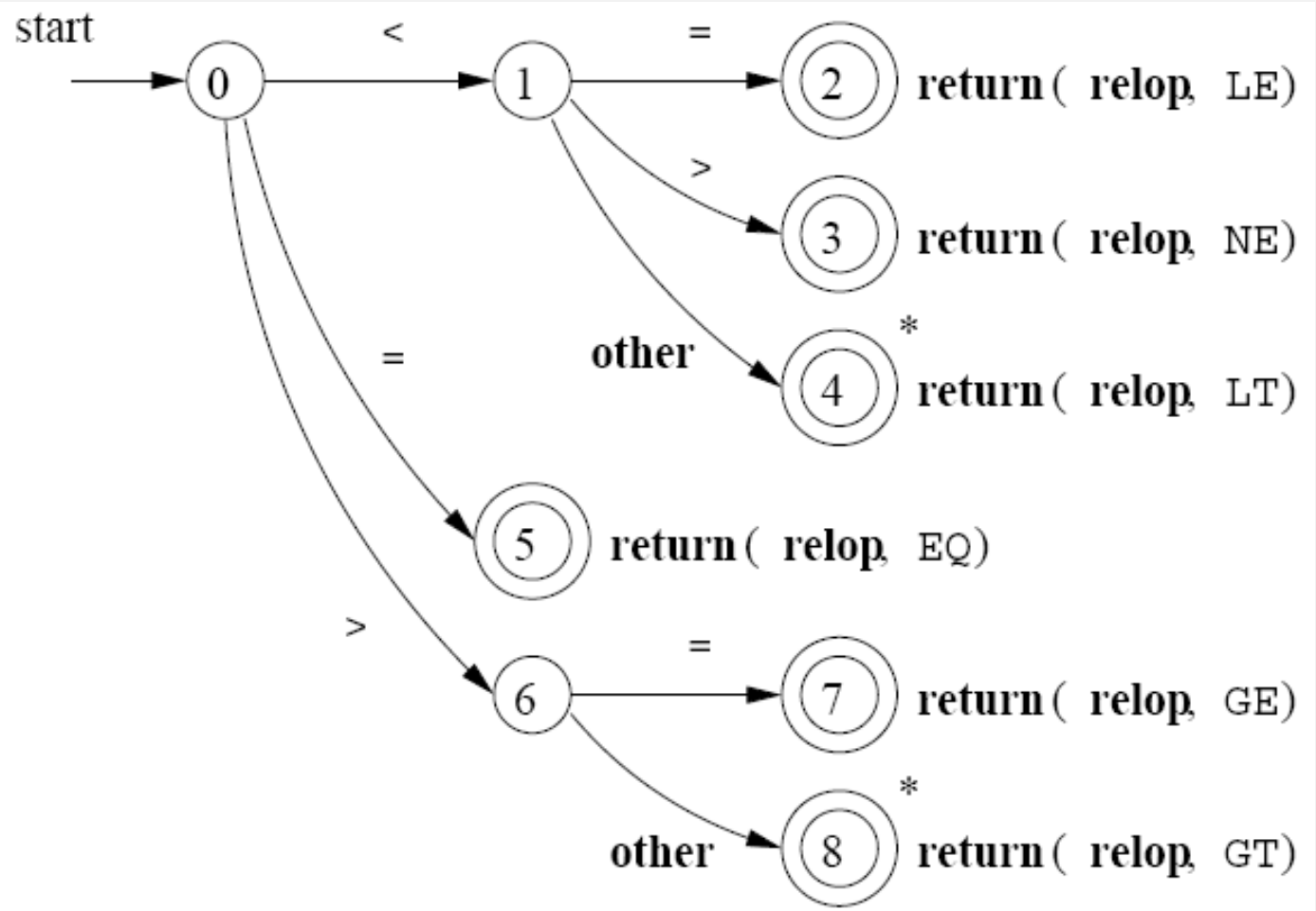

Figure 3.13: Transition diagram for **relop**

ตัวอย่าง TOKEN ID

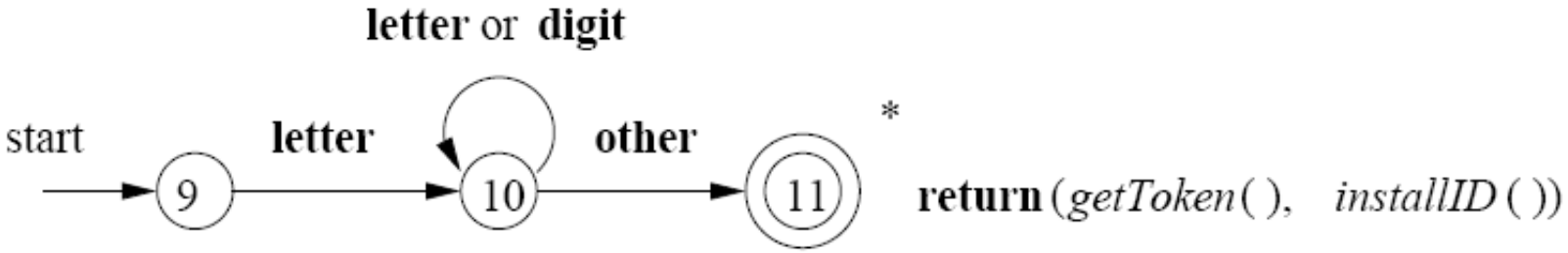

Figure 3.14: A transition diagram for **id**'s and keywords

- ถึงแม้ แผนภาพ 3.14 นี้จะวิเคราะห์ คีย์เวิร์ดอย่าง **if, then** หรือ **else** ได้ แต่เราไม่ถือว่าคำเหล่านี้เป็น **identifier lexeme**
- วิธีการจัดการแยกคำสงวนจาก **identifier** ทำได้ดังนี้
  - ตอนเริ่มต้น ให้ใส่คำสงวนเหล่านี้ใน **symbol table** มี **field(**เขตข้อมูล**)** ในตารางสัญลักษณ์ ที่บอกว่า คำเหล่านี้ไม่ใช่ **identifier** แต่เป็น **Token** เฉพาะ **installID()** เป็นฟังก์ชันที่ทำการเพิ่ม **Token ID** ในตารางสัญลักษณ์ ถ้าหากว่าคำนั้นไม่เป็นคำสงวน
  - สร้าง **transition diagram** สำหรับคำสงวนแต่ละอัน ดังรูป 3.15

การจัดการกับคำสงวน

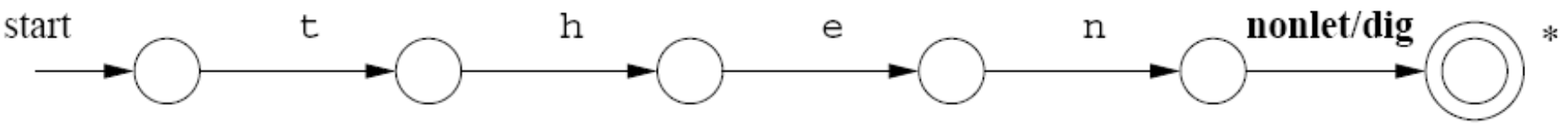

Figure 3.15: Hypothetical transition diagram for the keyword then

- สิ่งสำคัญคือ ตอนท้ายของ **transition diagram** จะต้องมี อักขระที่ไม่ใช่ตัวอักษรหรือตัวเลข(**nonletter-or-digit**)
- นั่นคือทดสอบว่าจะต้องไม่มีตัวอักษรหรือตัวเลขต่อท้ายซึ่งจะทำให้คำนั้นเป็น **identifier** แทนที่จะเป็น คำสงวน
- ถ้ามีคำว่า **thenextvalue** เข้ามา
  จะเห็นได้ว่า **then** เป็น **prefix** ของ **lexeme** โดยที่เราจะได้ **token id** ที่มี **attribute value** เป็น **thenextvalue**

TRANSITION DIAGRAM สำหรับ UNSIGNED NUMBERS

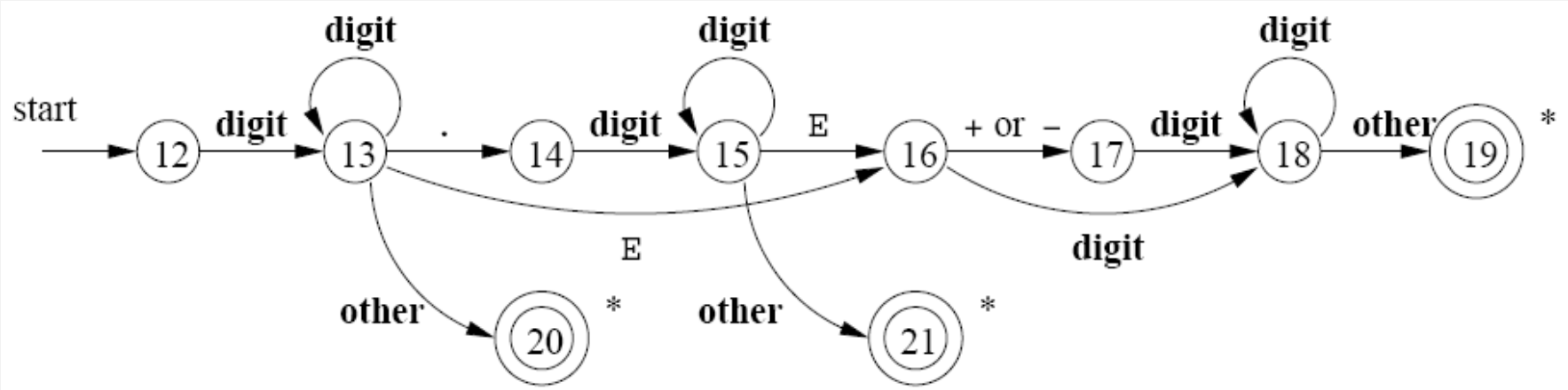

Figure 3.16: A transition diagram for unsigned numbers

- ลองทดสอบว่า ไดอะแกรมนี้ รับตัวเลขดังต่อไปนี้หรือไม่

  12, 13.4, 18E+7, 14E8, 5.5E-2, .4E+8

TRANSITION DIAGRAM สำหรับ WHITESPACE

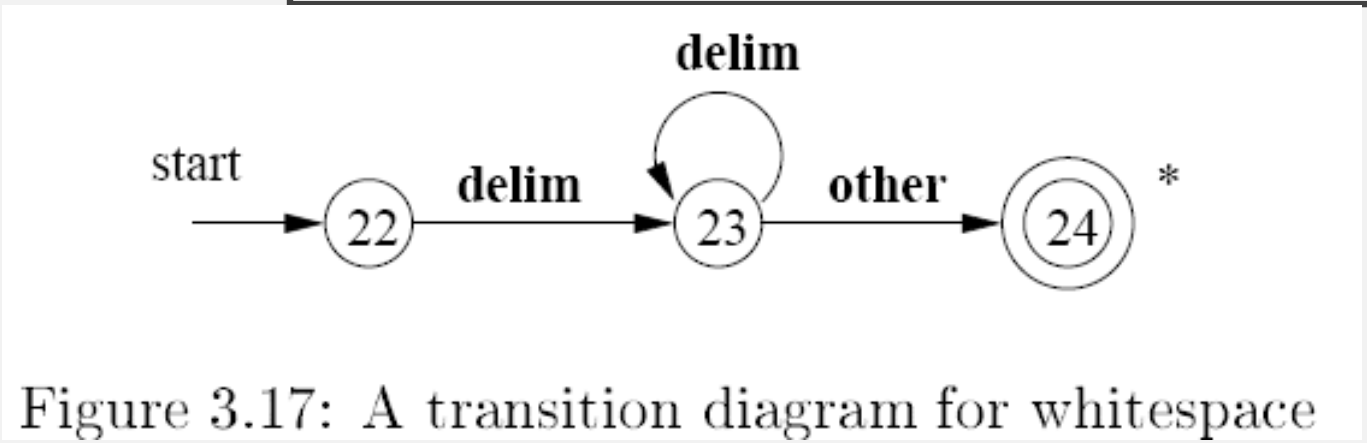

Figure 3.17: A transition diagram for whitespace

- Delim จะเป็นคำที่ใช้แทน ช่องว่าง(whitespace) ทั้งหลาย เช่น ตัวอักษรขึ้นบรรทัดใหม่ แท็บ ช่องว่าง(blank) เป็นต้น
- Delim จะมีจำนวน one or more
- เราไม่มีการส่ง token ของ whitespace กลับไป

3.4.4 ARCHITECTURE OF A LEXICAL ANALYZER BASED ON A TRANSITION DIAGRAM

- จากรูป 3.13 เราสามารถเขียนโปรแกรม โดยใช้คำสั่ง **switch** ได้ดังรูป 3.18
- **State** เป็นตัวแปรที่ใช้ในการเลือก **case**
- คำสั่ง **switch** เป็น **multiway branch**

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

Figure 3.18: Sketch of implementation of **relop** transition diagram

วิธีการเขียนโปรแกรม LEXICAL ANALYZER จาก TRANSITION DIAGRAM ทั้งหมด

1. เขียนโปรแกรมจาก transition diagram สำหรับแต่ละ token แล้วทดสอบตามลำดับ
   - พร้อมกับ reset pointer forward ใหม่ทุกครั้งเมื่อเริ่มทดสอบ transition diagram ถัดไป
   - วิธีการนี้ทำให้เราสร้าง transition diagram สำหรับ keyword ต่างๆ ดังรูปที่ 3.15 โดยเราตรวจสอบที่ diagram เหล่านี้ก่อนตรวจสอบ token id
2. ประมวลผล transition diagram ต่าง ๆ แบบ**คู่ขนาน**
   - วิธีการนี้ต้องทำงานด้วยความระมัดระวัง
3. รวม transition diagram ทุกอันเป็นหนึ่งเดียว
   เราจะอ่าน input จนกระทั่ง no possible next state
   เราจะเลือก longest lexeme ที่ตรงกับรูปแบบใดรูปแบบหนึ่ง
   ดังนั้นเราจะรวม state 0, 9, 12, 22 เป็น start state เดียวกัน

ตัวอย่าง EXPRESSION TRANSLATOR

- ตัวอย่าง Input : 9-5+2
- Lexical analyzer อนุญาตให้เราใช้ ตัวเลข(number), ตัวแปร(identifier) เครื่องหมาย +, -, *, / และ white space (ช่องว่าง, tabs, ขึ้นบรรทัดใหม่)ภายใน expression

ที่มาของ TOKEN

expr	→	*expr* + *term*	{ print('+') }
	|	*expr* − *term*	{ print('−') }
	|	*term*	
term	→	*term* * *factor*	{ print('*') }
	|	*term* / *factor*	{ print('/') }
	|	*factor*	
factor	→	(*expr*)	
	|	**num**	{ print(**num**.*value*) }
	|	**id**	{ print(**id**.*lexeme*) }

Figure 2.28: Actions for translating into postfix notation

num หมายถึง terminal symbol ใน parser

id (identifier) หมายถึง terminal symbol ชื่อตัวแปรใน parser

REMOVAL OF WHITE SPACE AND COMMENTS

- ถ้า Expression translator รับทุกๆ ตัวอักษรที่ส่งเข้ามา, ตัวอักษรอื่น ๆ เช่น ช่องว่าง จะทำให้โปรแกรมทำงานไม่ถูกต้อง
- การกำจัด white space และ คอมเมนต์ ทำโดย lexical analyzer
- การกำจัดสามารถทำได้ดังนี้

```
for ( ; ; peek = next input character ) {
        if ( peek is a blank or a tab ) do nothing;
        else if ( peek is a newline ) line = line+1;
        else break;
}
```

Figure 2.29: Skipping white space

Reading ahead

- **Lexical analyzer** อาจต้องการอ่านตัวอักษร(มากกว่า **1** ตัว)เข้ามาก่อนตัดสินว่าจะส่ง **token** ไหนกลับไปให้ **parser**
- ตัวอย่างเช่น เมื่ออ่านตัวอักษร **>**

  ถ้าตัวอักษรถัดไปเป็น **=** เราจะได้ **lexeme** “มากกว่าหรือเท่ากับ”

  ถ้าตัวอักษรถัดไปไม่ใช่ **=** เราจะได้ **lexeme** “มากกว่า”และต้องคืนตัวอักษรที่เกินมากลับไป
- วิธีการ
  - **input buffer** ไว้จัดเก็บเราสามารถอ่านและ **push back** ได้
  - **One-character read-ahead** เช่นมีตัวแปรชื่อ **peek**

Constants

- Input: 31+28+59
- เราสามารถแปลงเป็นลำดับของ **token** ดังนี้

  <num,31> <+> <num,28> <+> <num,59>

  โดยที่ **tuple** หรือ คู่อันดับ (วงเล็บ **< >**) ประกอบด้วย

  num เป็น **terminal symbol**
  และ **integer-valued attribute**
- **Lexical analyzer** จะทำการอ่านตัวอักษรเข้ามาแล้วเปลี่ยนให้เป็นเลขจำนวนเต็ม

การจัดการกับตัวเลขมากกว่า 1 หลัก

- Peek เก็บ ตัวอักษรแต่ละตัว(character) ที่อ่านเข้ามา
- ถ้า peek นั้นเป็นตัวเลข แล้วจะดำเนินการดังนี้

```
if ( peek holds a digit ) {
        v = 0;
        do {
                v = v * 10 + integer value of digit peek;
                peek = next input character;
        } while ( peek holds a digit );
        return token ⟨num, v⟩;
}
```

Figure 2.30: Grouping digits into integers

Recognizing keywords & identifiers

- **Keywords(คีย์เวิร์ด)/Reserved words(**คำสงวน**)**
  เป็นคำที่สงวนไว้ห้ามนำมาใช้สร้างเป็นชื่อฟังก์ชัน หรือ ตัวแปร
  ถ้าใช้แล้วจะทำให้การผิดพลาดในการคอมไพล์(**compile**)
- คีย์เวิร์ด ได้แก่ **for, do, if** หรืออื่น ๆ จะเป็นสายอักขระ
  (**character string**) ที่แน่นอนชัดเจน
- สายอักขระที่ไม่ใช่ คีย์เวิร์ด ใช้เป็น **identifier** ชื่อตัวแปร,ชื่อฟังก์ชัน
- **Identifier(id)** เป็น **terminal** ใน **parser**

identifiers

- ตัวอย่าง input :

  count = count + increment;

- Parser จะเห็นเป็น สายของ terminal ดังนี้

  <id,"count"> < = > <id,"count"> <+>

  <id, "increment"> <;>

– Token id จะมี attribute value บอกให้รู้ชื่อของตัวแปร

String table

- ช่วยในการเก็บสายอักขระที่อ่านเข้ามา
- การที่มีตัวชี้(**pointer**) ชี้ไปยังตารางที่เก็บสายอักขระ จะมีประสิทธิภาพดีกว่าเก็บเป็น **string** เฉยๆ
- คำสงวนจะถูกเก็บไว้ใน **string table** นี้
- เมื่อ **lexical analyzer** อ่านสายอักขระหรือ **lexeme** เข้ามา จะทำการตรวจสอบว่า **lexeme** นี้อยู่ใน **string table** หรือไม่
  - ถ้าอยู่จะส่ง **token** ของคำสงวนกลับจากตารางนี้
  - ถ้าไม่มี จะส่ง **token id** กลับ

การจัดการกับตัวอักษร

- ฟังก์ชัน **get()** จะเข้าไปตรวจสอบคำ(รวมทั้ง **reserved words**)ใน ตาราง **hash table** ที่ชื่อว่า **words** ซึ่งใช้ในการ **map** จาก **lexemes** ไปยัง **token**
- ตาราง **hash table** มีการกำหนดไว้ดังนี้

```
Hashtable words = new Hashtable();
```

```
if ( peek holds a letter ) {
        collect letters or digits into a buffer b;
        s = string formed from the characters in b;
        w = token returned by words.get(s);
        if ( w is not null ) return w;
        else {
                Enter the key-value pair (s, ⟨id, s⟩) into words
                return token ⟨id, s⟩;
        }
}
```

Figure 2.31: Distinguishing keywords from identifiers

2.6.5 A lexical analyzer

ฟังก์ชัน scan() จะส่ง token object กลับ

```
Token scan( ) {
    skip white space, as in section 2.6.1;
    handle numbers, as in section 2.6.3;
    handle reserved words and identifiers, as in section 2.6.4;
    /* if we get here, treat read-ahead character peek as a token */
    Token    t = new Token(peek);
    peek = blank /*initialization, as discussed in section2.6.2 */;
    return   t;
}
```

2.6.5 A lexical analyzer

- ฟังก์ชัน **scan()** เป็นส่วนหนึ่งของ **java package** ที่ชื่อว่า **Lexer** สำหรับ **lexical analysis**
- **Lexer package** ประกอบด้วย คลาส **Token**(รูป2.32) และ **Lexer**(รูป2.34)
- คลาส **Token** ประกอบด้วย คลาสย่อย **Num** และ **Word** ดังภาพ

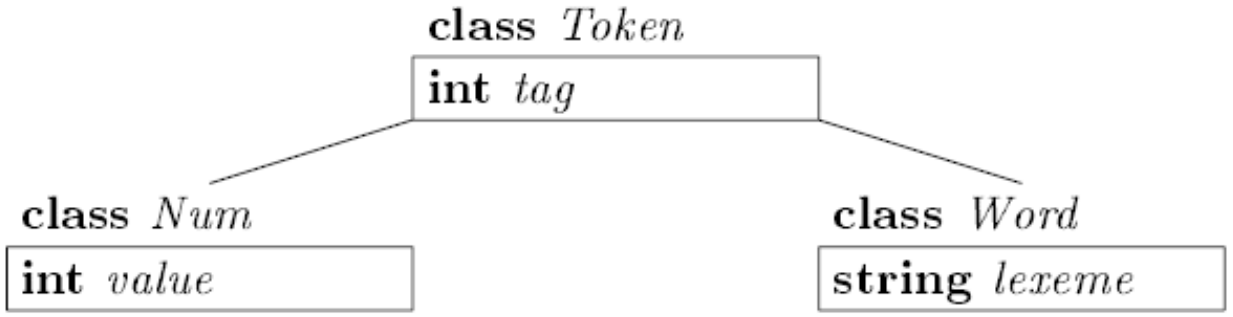

Figure 2.32: Class *Token* and subclasses *Num* and *Word*

```
1) package lexer;                          // File Token.java
2) public class Token {
3)         public final  int  tag;
4)         public Token(int t)  { tag = t ;}
5) }
1) package lexer;                          // File Tag.java
2) public class Tag {
3)         public final  static int
4)              NUM = 256, ID = 257, TRUE = 258, FALSE = 259;
5) }
```

```
1) package lexer;                          // File Num.java
2) public class Num extends Token {
3)    public final int value;
4)    public Num(int v) { super(Tag.NUM); value = v; }
5) }
```

```
1) package lexer;                          // File Word.java
2) public class Word extends Token {
3)    public final String lexeme;
4)    public Word(int t, String s) {
5)       super(t); lexeme = new String(s);
6)    }
7) }
```

Figure 2.33: Subclasses Num and Word of Token

```
 1) package lexer;                          // File Lexer.java
 2) import java.io.*; import java.util.*;
 3) public class Lexer {
 4)    public int line = 1;
 5)    private char peek = ' ';
 6)    private Hashtable words = new Hashtable();
 7)    void reserve(Word t) { words.put(t.lexeme, t); }
 8)    public Lexer() {
 9)       reserve( new Word(Tag.TRUE,  "true")  );
10)       reserve( new Word(Tag.FALSE, "false") );
11)    }
12)    public Token scan() throws IOException {
13)       for( ; ; peek = (char)System.in.read() ) {
14)          if( peek == ' ' || peek == '\t' ) continue;
15)          else if( peek == '\n' ) line = line + 1;
16)          else break;
17)       }
          /* continues in Fig. 2.35 */
```

Figure 2.34: Code for a lexical analyzer, part 1 of 2

```
18)         if( Character.isDigit(peek) ) {
19)             int v = 0;
20)             do {
21)                 v = 10*v + Character.digit(peek, 10);
22)                 peek = (char)System.in.read();
23)             } while( Character.isDigit(peek) );
24)             return new Num(v);
25)         }
26)         if( Character.isLetter(peek) ) {
27)             StringBuffer b = new StringBuffer();
28)             do {
29)                 b.append(peek);
30)                 peek = (char)System.in.read();
31)             } while( Character.isLetterOrDigit(peek) );
32)             String s = b.toString();
33)             Word w = (Word)words.get(s);
34)             if( w != null ) return w;
35)             w = new Word(Tag.ID, s);
36)             words.put(s, w);
37)             return w;
38)         }
39)         Token t = new Token(peek);
40)         peek = ' ';
41)         return t;
42)     }
43) }
```

Figure 2.35: Code for a lexical analyzer, part 2 of 2