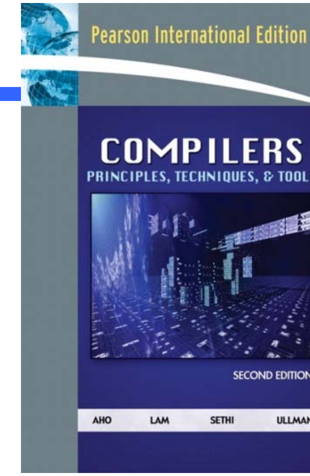
Introduction to Compiler Design

Lexical Analysis I

Professor Yi-Ping You
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Supplements to Textbook

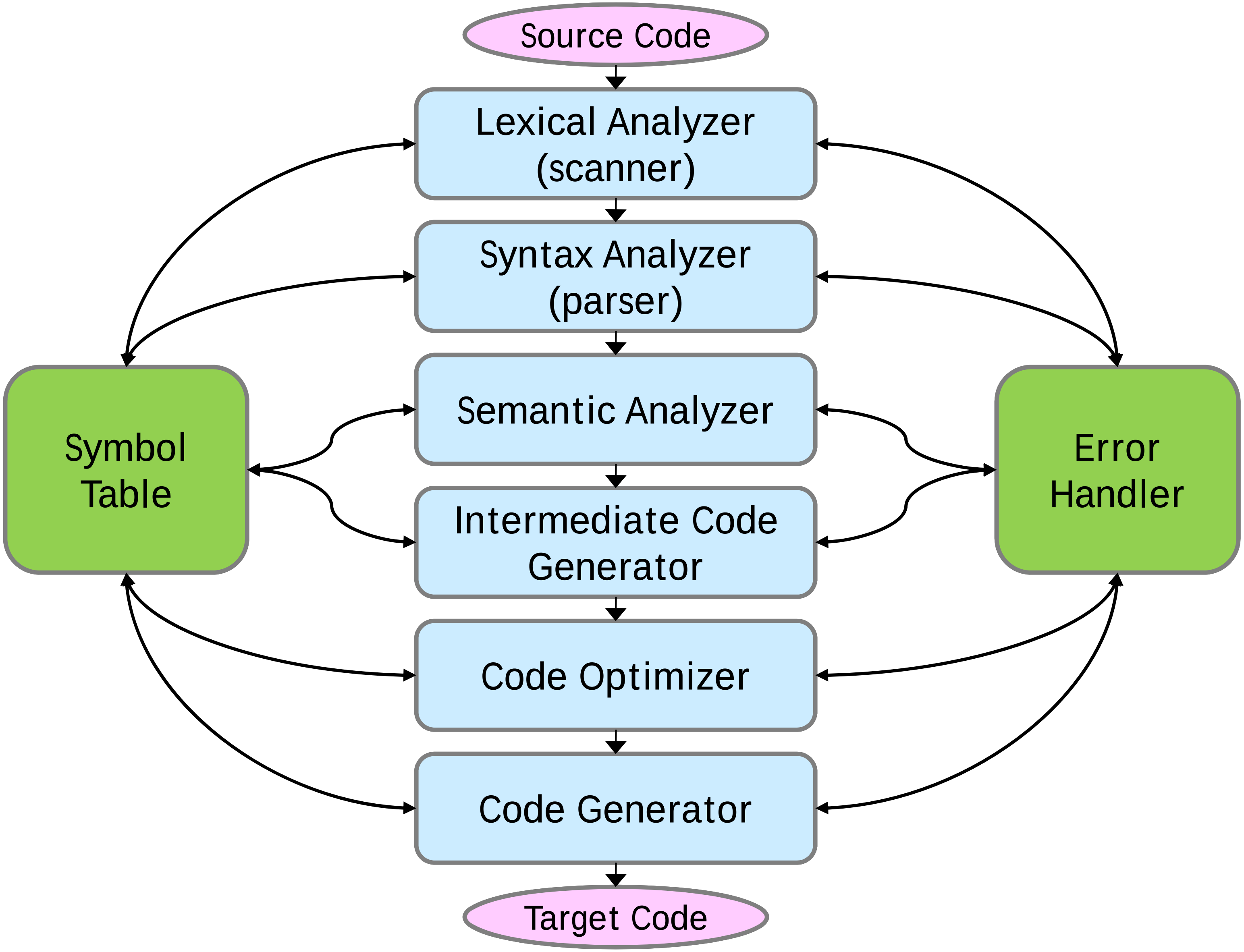


- http://dragonbook.stanford.edu/
- Errata sheet
  - http://infolab.stanford.edu/~ullman/dragon/errata1.html
  - http://infolab.stanford.edu/~ullman/dragon/errata.html

Outline

- Lexical Analysis and Tokens
- Regular Expressions
- Interaction between Scanner & Parser
- Implementation of Lexical Analysis
  - Transition Diagram

The Structure of a Compiler



Lexical Analysis Process

`if (b == 0) a = b;`

Preprocessed source code, read char by char

↓

Lexical Analysis/Scanner

↓

if	(	b	==	0	)	a	=	b	;
KWif	**(**	**ID**	**==**	**NUM**	**)**	**ID**	**=**	**ID**	**SEMI**

- Transform multi-character input stream to token stream
- Reduce length of program representation (remove spaces)

Examples of Tokens

- Token
  - smallest logically cohesive sequence of characters of interest in source program

Token	Lexeme
Single-character operators	`=   +   -   >`
Multi-character operators	`:=   ==   <>   ->`
Keywords	`if   else   while   break`
Identifiers	`my_variable   flag1`
Numeric constants/literals	`123   45.67   8.9e+05`
Character literals	`‘a’   ‘~’   ‘\’`
String literals	`“abcd”`

Tokens, Patterns and Lexemes

- A token is a pair a token name and an optional attribute value
  - E.g., *identifier*, *keyword*, *operator*
- A pattern is a description of the form that the lexemes of a token may take
  - Regular expression
  - E.g., **`[A-Za-z_][A-Za-z0-9_]*`**
- A lexeme is a sequence of characters in the source program that matches the pattern for a token
  - E.g., **`x`**, **`y`**; **`if`**, **`else`**, **`while`**; **`+`**, **`-`**, **`<`**

Attributes for Tokens: Example

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

<**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`**>

Lexical Errors

- Some errors are out of power of lexical analyzer to recognize:

```
fi (a == f(x)) …
d = 2r;
```

- However it may be able to recognize errors like:

```
d = "abc;
```

- Such errors are recognized when no pattern for tokens matches a character sequence

Error Recovery

- Why error recovery?

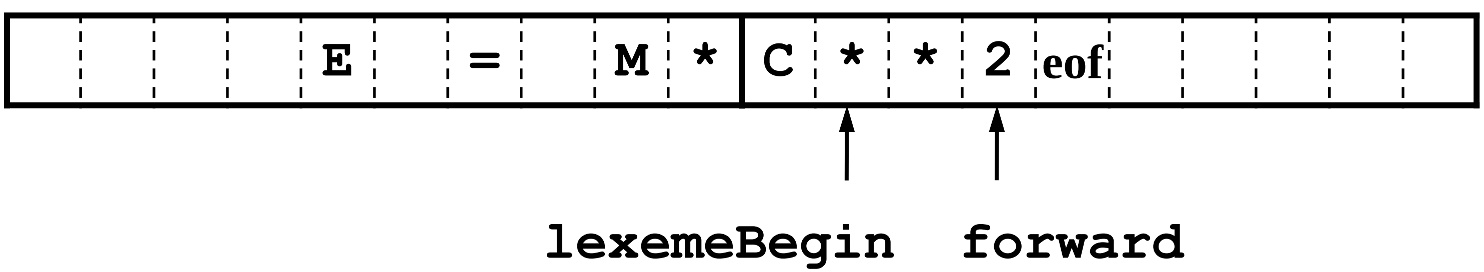
- Recovery strategies:
  - Most straightforward
    - Panic mode: successive characters are ignored until we reach to a well formed token
  - Aggressive
    - Delete one character from the remaining input
    - Insert a missing character into the remaining input
    - Replace a character by another character
    - Transpose two adjacent characters

Input Buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran:
    - `DO 5 I = 1.25` **(`DO5I` is an identifier)**
    - `DO 5 I = 1,25` **(`DO` is a keyword)**

Input Buffering (Cont'd)

- We need to introduce a two-buffer scheme to handle large lookaheads safely
  - a buffer divided into two N-character halves (e.g., N=1024)
  - read N chars into buffer with system read command (vs. one-char at a time)
  - two pointers: <lexemeBegin, forward> enclose current lexeme
  - read next N-char if forward pointer pass current half



Input Buffering with Sentinels

- Need two tests for out-of-bound for every forward
  - one for 1st half, one for 2nd half
- Add one sentinel (EOF) at each half
  - need only one test (on EOF) for each move of forward pointer
  - do additional check on two halves only when EOF is encountered

			E		=		M	*	eof	C	*	*	2	eof					eof

Outline

- Lexical Analysis and Tokens
- Regular Expressions
- Interaction between Scanner & Parser
- Implementation of Lexical Analysis
  - Transition Diagram

How to Describe Tokens

- Use regular expressions (REs) to describe programming language tokens!
- A regular expression is defined inductively
  - $\varnothing$ empty set {}
  - **a** ordinary character stands for itself
  - ε empty string
  - $R|S$ either R or S (alteration or union), where R,S = RE
  - RS R followed by S (concatenation)
  - R^* concatenation of R zero or more times (Kleene closure)

> - Italics for symbols
> - Boldface for regular expression

Regular Expression: Examples

- **a** | **b** $\{a, b\}$
- (**a** | **b**)(**a** | **b**) $\{aa, ab, ba, bb\}$
- **a*** $\{\varepsilon, a, aa, aaa, \ldots\}$
- (**a** | **b**)* the set of all strings of a's and b's
  $\{\varepsilon, a, b, aa, ab, ba, bb, \ldots\}$
- **a** | **a*** **b** the set containing the string a and all strings consisting of zero or more a's followed by a b
  $\{a, b, ab, aab, aab, aaab, \ldots\}$

Language

- A regular expression R describes a set of strings of characters denoted $L(R)$, also called a regular set
- $L(R)$ = the language defined by R
  - $L($**abc**$) = \{ abc \}$
  - $L($**hello|goodbye**$) = \{ hello, goodbye \}$
  - $L($**1(0|1)***$)$ = all binary numbers that start with a 1
- Each token can be defined using a regular expression

- A definition of language does not require that any meaning be ascribed to the strings in the language

Regular Expressions

- ε is a RE denoting $\{\varepsilon\}$
- If $a \in$ alphabet, then **`a`** is a RE denoting $\{a\}$
- Suppose r and s are RE denoting $L(r)$ and $L(s)$
- (r) is a RE denoting $L(r)$
- $(r) \mid (s)$ is a RE denoting $L(r) \cup L(s)$
- $(r)\ (s)$ is a RE denoting $L(r)\ L(s)$
- $(r)^*$ is a RE denoting $(L(r))^*$

Algebraic Laws for REs

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 |
εr = rε = r	ε is the identity for concatenation
r* = (r|ε)*	ε is contained in *
r** = r*	* is idempotent

Extensions of Regular Expression

- Notational standards
  - R^+ one or more strings of R : $R(R^*)$
    - Positive closure
    - $R^* = R^+ | \varepsilon$
  - $R?$ optional R : $(R|\varepsilon)$
  - **[abcd]** one of listed characters: **(a|b|c|d)**
  - **[a-z]** one character from this range: **(a|b|c|d...|z)**
  - **[^ab]** anything but one of the listed chars
  - **[^a-z]** one character not from this range
- Example
  - C variable name: **[A-Za-z_][A-Za-z0-9_]***

Regular Definitions

- For simplicity, give names to regular expressions
  - format: name $\rightarrow$ regular expression
  - $d_1 \rightarrow r_1$
    $d_2 \rightarrow r_2$
    $\ldots$
    $d_n \rightarrow r_n$
    where r_i over alphabet $\cup \{d_1, d_2, \ldots, d_{i-1}\}$

- E.g.
  - $digit \rightarrow$ **0|1|2| · · · |9**
  - $letter\_ \rightarrow$ **a|b|c| · · · |z|A|B| · · · |Z|_**
  - $id \rightarrow letter\_ \,(letter\_ \mid digit)^*$

Regular Expression: Expressions

- Regular Expression, R
  - `a`
  - `ab`
  - `a|b`
  - `(ab)*`
  - `(a|ε)b`
  - *digit* = `[0-9]`
  - *posint* = *digit*$^+$
  - *int* = `-?`*posint*
  - *real* = *int*`(ε|(.`*posint*`))`

  = `-?[0-9]+(ε|(.[0-9]+))`

- Strings in $L(R)$
  - "*a*"
  - "*ab*"
  - "*a*", "*b*"
  - " ", "*ab*", "*abab*", ...
  - "*ab*", "*b*"
  - "0", "1", "2", ..., "9"
  - "8", "412", "03", ...
  - "-23", "34", ...
  - "-1.56", "12", "1.056", ...

- Note, ".45" is not allowed in this definition of *real*

Restrictions on REs

- Regular expressions are not capable of describing most complete languages
- They describe languages composed of sets of strings of the from

  $S \to \alpha B$

  - α is a basic symbol
  - B is a regular expression
  - S is a regular definition and is not a part of B
- They cannot describe balanced nesting constructs
  - E.g., **`if`** ... **`then`** ... **`else`** ...
  - Recognized by context free grammar

$$
\begin{aligned}
stmt &\to \textbf{if}\ \ expr\ \ \textbf{then}\ \ stmt \\
&\ \mid\ \textbf{if}\ \ expr\ \ \textbf{then}\ \ stmt\ \textbf{else}\ stmt \\
&\ \mid\ \varepsilon \\
expr &\to term\ \textbf{relop}\ term \\
&\ \mid\ term \\
term &\to \textbf{id} \mid \textbf{number}
\end{aligned}
$$

$$
\begin{aligned}
digit &\to \texttt{[0-9]} \\
digits &\to digit^{+} \\
number &\to digits(\mathrm{.digits})?(\mathrm{E}[+\text{-}]?\mathrm{digits})? \\
letter &\to \texttt{[A-Za-z]} \\
id &\to letter(letter|digit)\text{*} \\
if &\to \textbf{if} \\
then &\to \textbf{then} \\
else &\to \textbf{else} \\
relop &\to \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{aligned}
$$

Restrictions on REs (Cont'd)

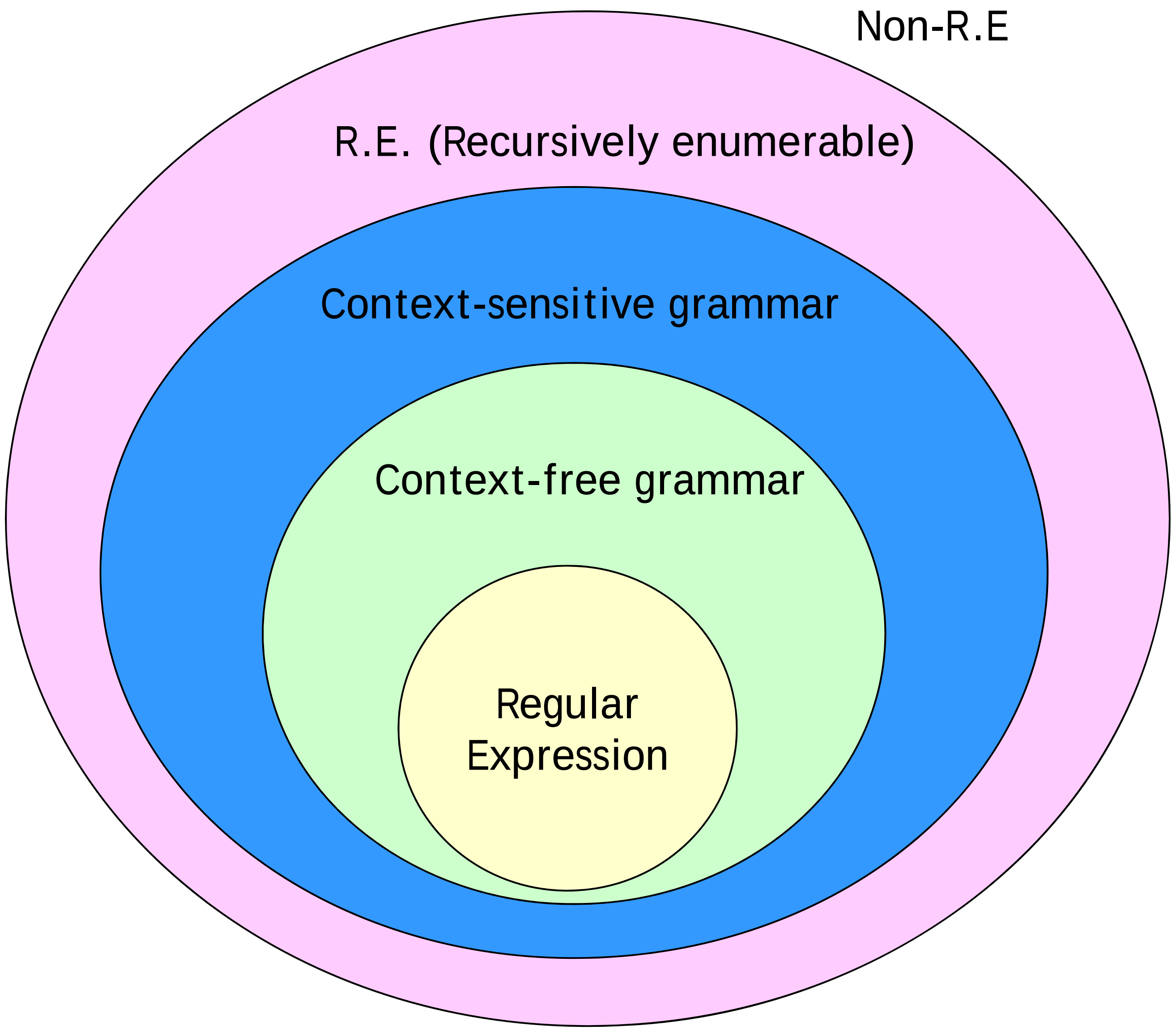
- Repetition of the same string cannot be described
  - E.g., $\{wcw \mid w$ is a string of a's and b's$\}$
  - Cannot be recognized even using context free grammars
- Constructs where the number of repetitions is fixed by the value of a part of the string cannot be described
  - E.g., $nHa_1a_2...a_n$
  - Cannot be recognized even using context free grammars

- Remark: anything that needs to "memorize" "non-constant" amount of information happened in the past cannot be recognized by regular expressions

The Chomsky Hierarchy

- Unrestricted languages (type 0)
  - Turing machines
- Context-sensitive languages (type 1)
  - Linear bounded automata
- Context-free languages (type 2)
  - Pushdown automata
- Regular languages (type 3)
  - Finite automata

Chomsky Hierarchy



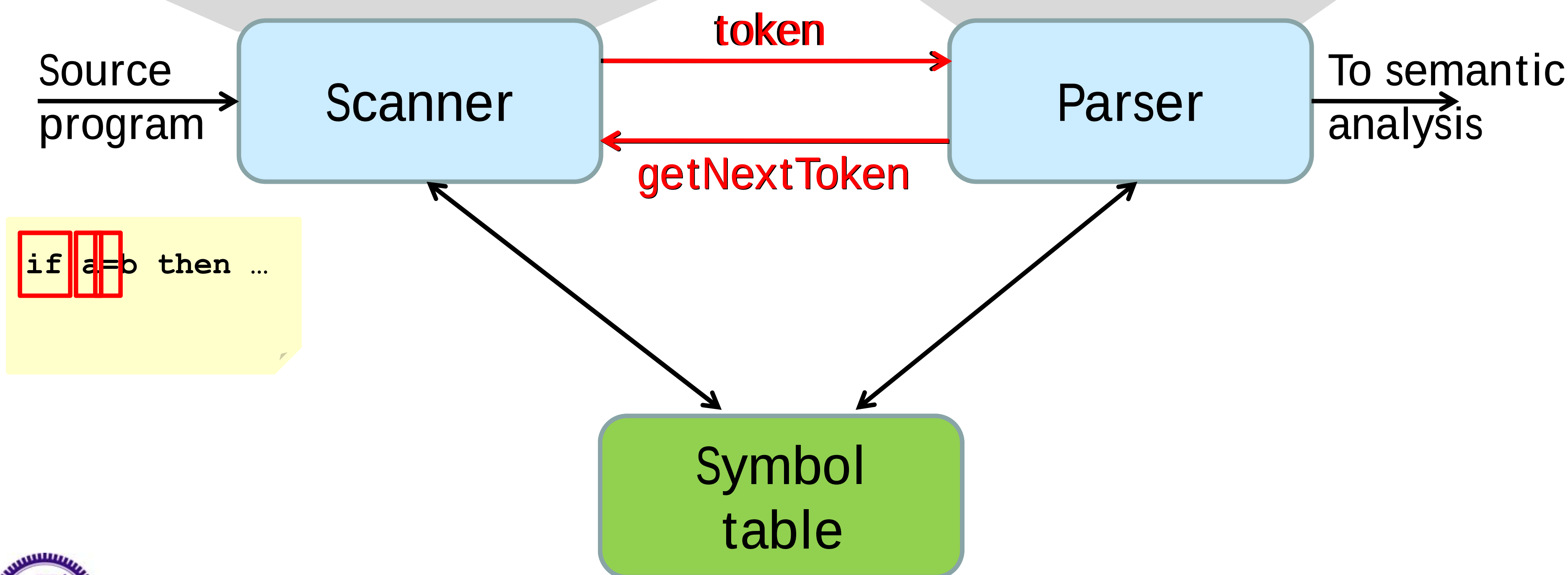
Outline

- Lexical Analysis and Tokens
- Regular Expressions
- Interaction between Scanner & Parser
- Implementation of Lexical Analysis
  - Transition Diagram

Interaction between Scanner & Parser

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 → < | > | <= | >= | = | <>

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



Attributes of Tokens

Lexemes	Token Name	Attribute Value
`if`	**if**	-
`then`	**then**	-
`else`	**else**	-
id	**id**	Pointer to table entry
number	**number**	Pointer to table entry (or the value of the number)
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

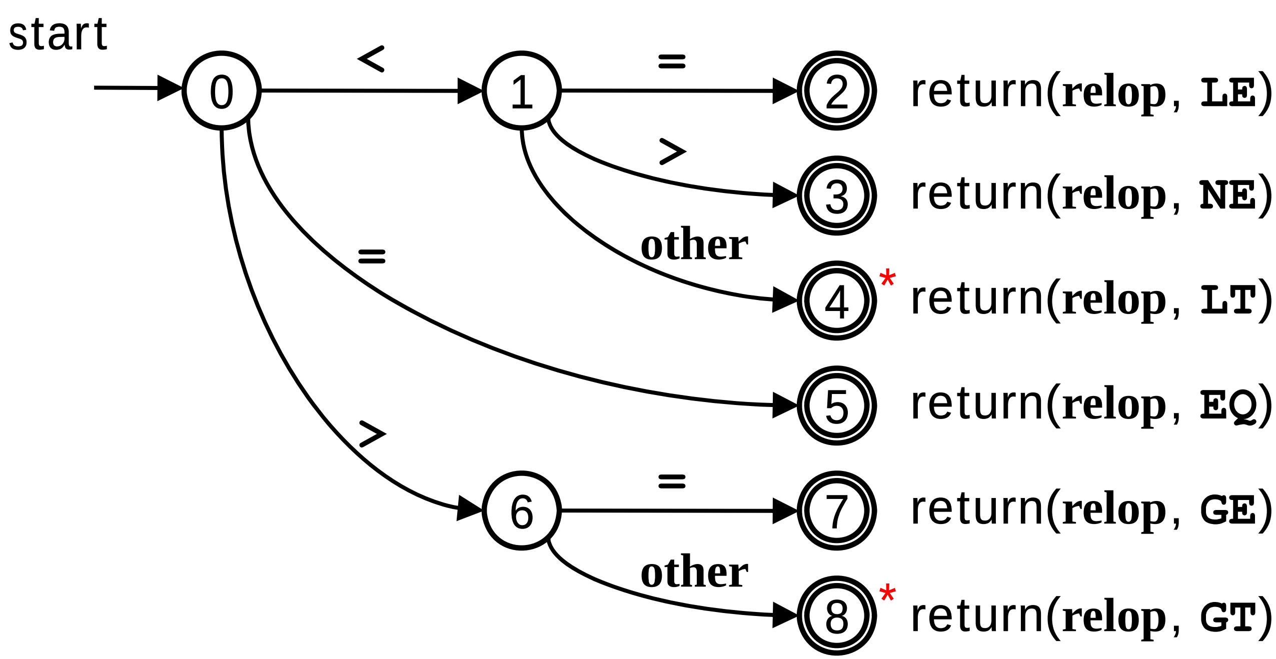
Outline

- Lexical Analysis and Tokens
- Regular Expressions
- Interaction between Scanner & Parser
- Implementation of Lexical Analysis
  - Transition Diagram

Recognition of Tokens

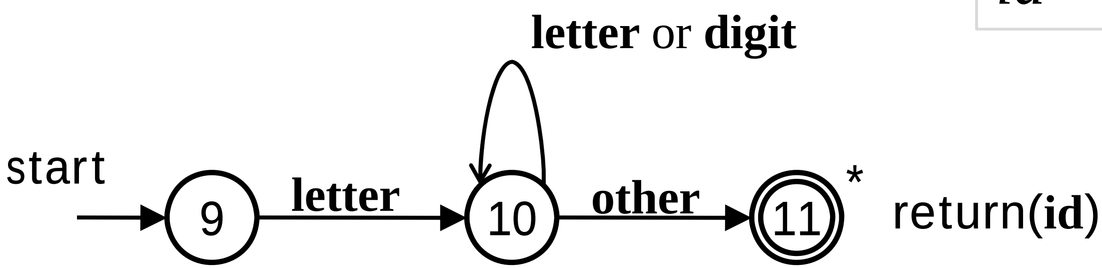
- Use Transition Diagram

$relop \rightarrow$ **< | > | <= | >= | = | <>**



Transition Diagram: Examples

$id \rightarrow letter\ (letter \mid digit)^*$

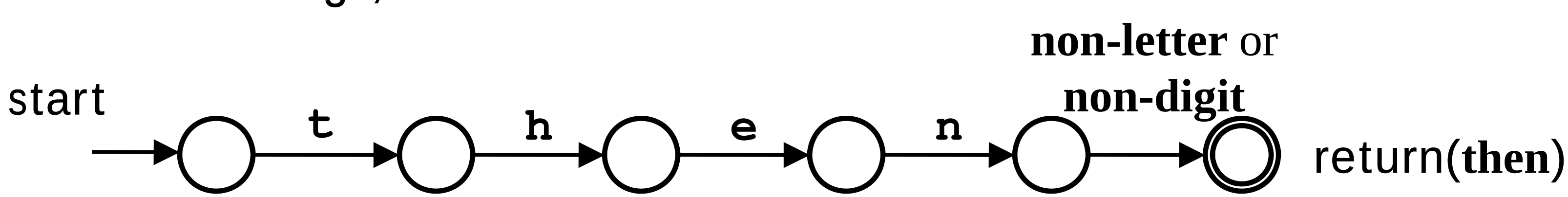



```
TOKEN getID()
{
  TOKEN retToken = new (ID)
  while (1) {     /* repeat character processing until a return or
                     failure occurs */
    switch (state) {
      case 9: c = nextchar();
              if (isletter( c) ) state = 10; else state = failure();
              break;
      case 10: …
      case 11: retract(1);
               retToken.attribute = insert(id);
               return(retToken);
    }
  }
}
```

Transition Diagram: Examples

$$id \rightarrow letter\ (letter \mid digit)^*$$

- The RE also recognizes reserved words
  - e.g., **`if`**, **`then`**, **`else`**
- Two solutions
  - Install the reserved words in the symbol table initially
  - Create separate transition diagrams for each keyword
    - Must prioritize the tokens so that the reserved-word tokens are recognized in preference to id
      - E.g., **`thenextvalue`**



Transition-Diagram-Based Scanner

- Three ways to implement a lexical analysis using transition diagrams
  1. Run the transition diagrams one by one (with prioritization)
  2. Run the transitions diagrams “in parallel”
     - Longest match
  3. Combine all transition diagrams into one (preferred)