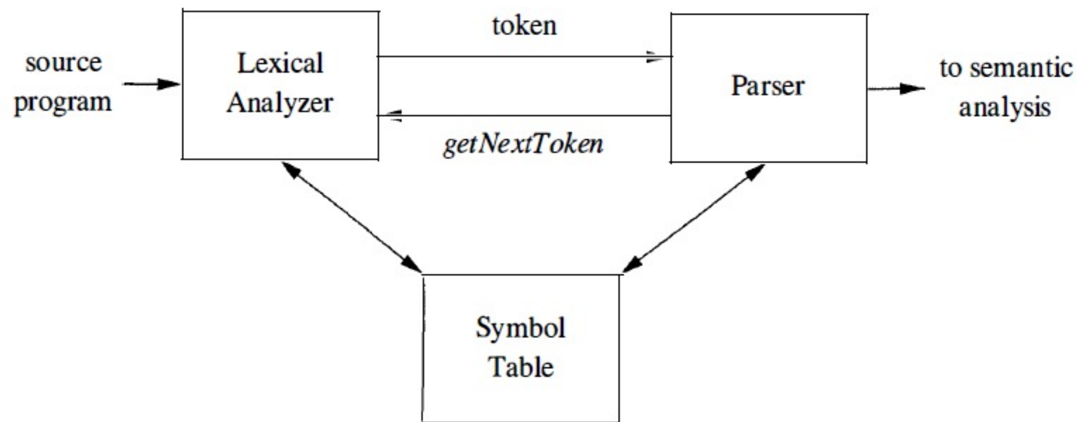
CSE 304 Compiler Design

Lexical Analysis [Supl]

YOUNGMIN KWON / TONY MIONE

The Role of the Lexical Analyzer



Why separating lexical analysis and parsing

- Simplify design (comments, white spaces…)
- Improve compiler efficiency (simpler algorithm)
- Improve compiler portability

Finite Automata

Tokens can be described with a directed graph.

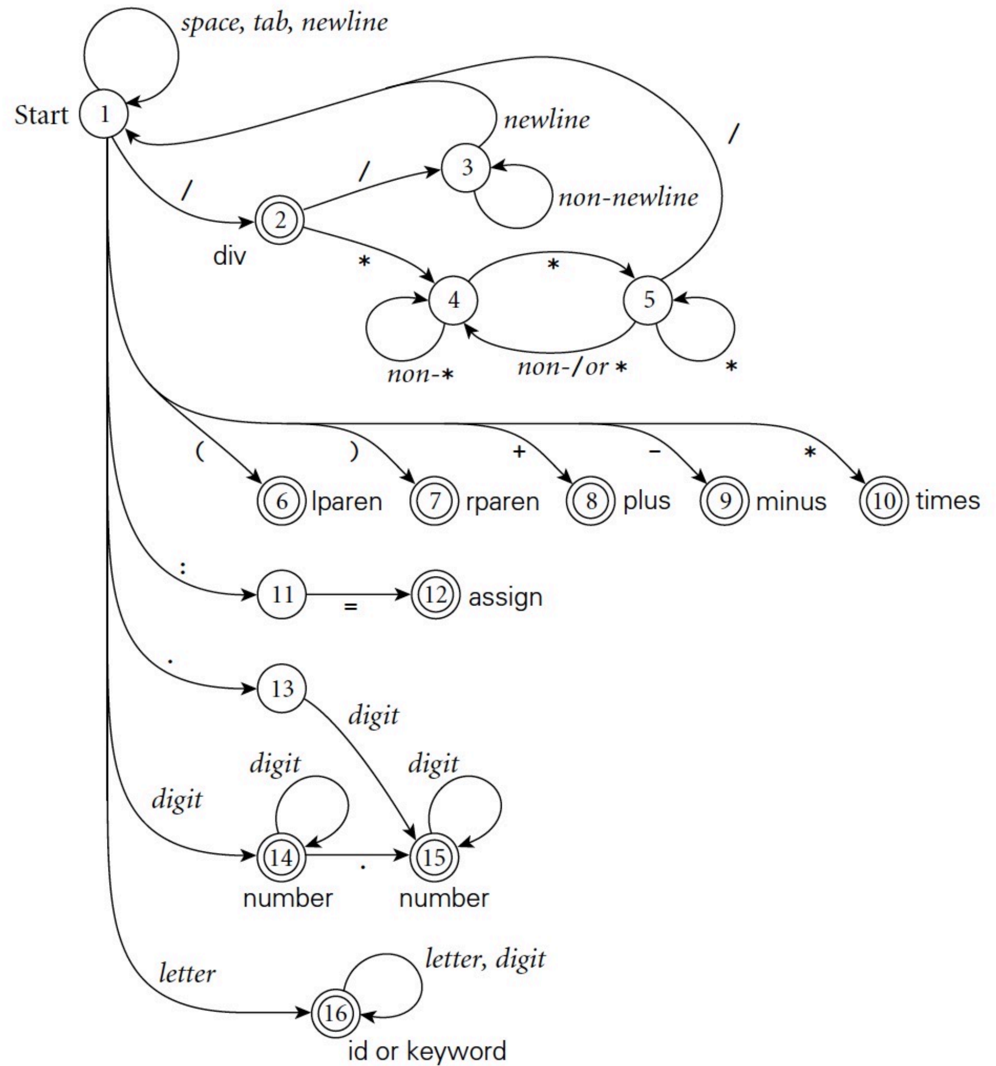
Nodes are ‘states’

Edges are character inputs during the scan

Double circled states are ‘accepting’ or ‘final’ states

Finite Automata

Finite Automata graph for simple calculator language



Scanning

Consider an ad-hoc (hand-written) scanner for a Calculator:

***assign* → :=**

***plus* → +**

***minus* → -**

***times* → ***

***div* → /**

***lparen*→ (**

***rparen*→)**

id* → letter (letter | digit)

number* → digit digit

| digit * (. digit | digit .) digit *

***comment* → /* (non-* | * non-/)* */**

| // (non-newline)* newline

Scanning

Read characters one at a time with look-ahead

```
skip initial white space (spaces, tabs, and newlines)

 if cur_char∈{'(', ')', '+', '-', '*'}
        return the corresponding single-character token
if cur_char= ':'
        read the next character
        if it is '=' then return assign else announce an error
 if cur_char= '/'
        peek at the next character
        if it is '*' or '/'
               read additional characters until "*/" or newline is seen, respectively
               jump back to top of code
        else return div
```

Scanning

```
if cur_char = .
        read the next character
        if it is a digit
                read any additional digits
                return number
        else announce an error
if cur_char is a digit
        read any additional digits and at most one decimal point
        return number
if cur_char is a letter
        read any additional letters and digits
        check to see whether the resulting string is read or write
        if so then return the corresponding token
        else return id
else announce an error
```

Table Driven Scanning

- Scanning can be done with a relatively simple algorithm and a state table
- State table has state number on left and charcter/character class across the top
- Cell contents are the ‘next state’.

Table Driven Scanning the pseudo-code

```
state = 0..number_of_states
token = 0..number_of_tokens
scan_tab : array [char, state] of record
   action : (move, recognize, error)
   new_state : state
token_tab : array[state] of token  -- what to recognize
keyword_tab : set of record
   k_image : string
   k_token : token
tok : token
cur_char : char
remembered_chars : list of char
repeat
   cur_state ;  state := start_state
   image : string := null
   remembered_state : state := 0
```

Table Driven Scanning the pseudo-code

```
loop
  read cur_char
  case scan_tab[curr_char, cur_state].action
     move:
       if token_tab[cur_state] != 0
          --- this could be a final state
          remembered_state := cur_state
          remembered_chars := epsilon
       add cur_char to remember_chars
       cur_state := scan_tab[cur_char, cur_state].new_state
     recognize:
       tok := token_tab[cur_state]
       unread cur_char  -- push back into input stream
       exit inner loop
```

Table Driven Scanning the pseudo-code

```
    error:
        if remembered_state != 0
            tok := token_tab[remembered_state]
            unread remembered_chars
            remove remembered_chars from image
            exit inner loop
        -- else print error message and recover; probably start over
    append cur_char to image
  -- end inner loop
until tok !element_of {white_space, comment}

look image up in keyword_tab and replace tok with approparite keyword if found

return (tok, image)
```

Table Driven Scanning the table

State	space,tab	newline	/	*	(	)	+	-	:	=	.	digit	letter	other	
1	17	17	2	10	6	7	8	9	11		13	14	16		
2			3	4											div
3	3	18	3	3	3	3	3	3	3	3	3	3	3	3	
4	4	4	4	5	4	4	4	4	4	4	4	4	4	4	
5	4	4	18	5	4	4	4	4	4	4	4	4	4	4	
6															lparen
7															rparen
8															plus
9															minus
10															times
11										12					
12															assign
13												15			
14											15	14			number
15												15			number
16												16	16		identifier
17	17	17													white_space
18															comment

Questions?