LEXICAL ANALYSIS

if (x = 5) then x := 10;

Goal: assist parsing
How: break up input into tokens

(if) (() [id, 32] (=) [const, 45] ()) (then) [id, 32] (:=) [const, 57] (;)

symbol table pointer

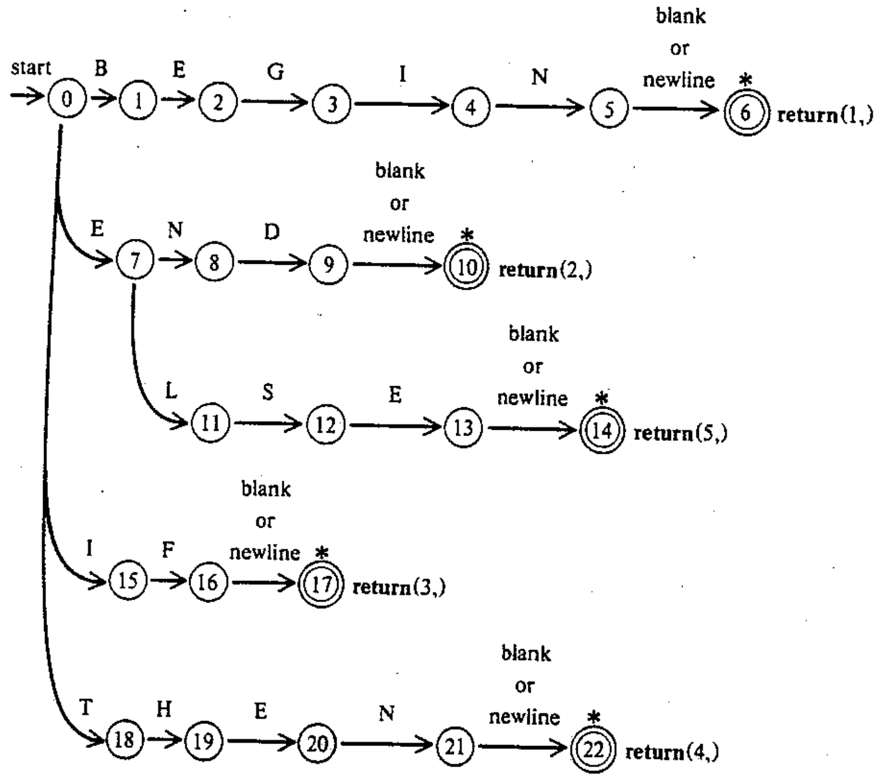
Tokens are defined by Regular Expression.

→ finite automata recognizable

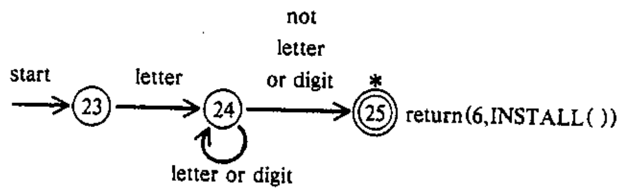
Symbol Table

	Entry	
32	x	→
45	const	→ 5
57	const	→ 10

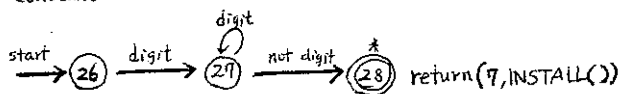
keywords:



identifier:



constant



Transition diagrams

SYNTAX ANALYSIS (PARSING)

parser (syntax analyzer) ≡ deterministic pda

Assume lanuage is defined by a context-free grammar.

Goal: Given input string, produce parse tree in the grammar

Convention:

Terminals:	**a, b, c, …**
Nonterminals:	**A, B, C, …**
Mixed string:	$\alpha, \beta, \gamma \ldots$

G:

$S \rightarrow AB$
$A \rightarrow Ab \mid a$
$B \rightarrow bD$
$D \rightarrow d$

input: abbbd

Top-down	**Bottom-up**
S	abbbd
AB	Abbbd
AbB	Abbd
AbbB	Abd
abbB	AbD
abbbD	AB
abbbd	S

Recursive-Descent Parsing

1970's: operator precedence (expression)
recursive descent (the rest)

current: LL(k) – predictive parser (parse table)
LR(k) – shift/reduce (parse table)

Note:

(1) left-recursion

$$A \rightarrow A + B$$

or → **get rid of left-recursion**

$$A \rightarrow BaA$$
$$B \rightarrow Ab$$

(2) pairwise disjointness test

G1: $A \rightarrow aB$
$A \rightarrow bAb$
$A \rightarrow c$

Depending on next input character, the parser knows which production rule to use.

G2: $A \rightarrow aBb$ $\xrightarrow{\textit{left factoring}}$ $A \rightarrow aN$
$A \rightarrow a$ $\qquad N \rightarrow Bb \mid \lambda$

LL(k) parser (Left-to-right read, Leftmost derivation)

Lookahead (k symbols)

Ex G: S → cABc
A → aAa
A → c
B → bBb
B → c

input: cacabcbc

S
cABc
caAaBc
cacaBc
cacabBbc
cacabcbc ($)

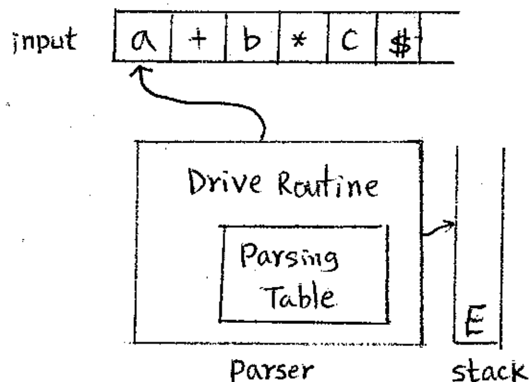
	a	b	c
S			cABc
A	aAa		c
B		bBb	c

```
procedure LL(1)
    push (S)
    read (symbol)
    while stack not empty do
       case stack[top] of
          terminal: if stack[top] = symbol
              then pop stack and read (symbol)
              else exit to error-rtn
          nontermianl: if table[stack[top],symbol] ≠ error
              then replace stack[top] with table[stack[top, symbol]
              else exit to error-rtn
       endcase
    endwhile
    if symbol ≠ eos then exit to error-rtn
```

* eos ≡ end-of-string marker ($)

LR(k) Parser (Left-to-right read, Rightmost derivation)

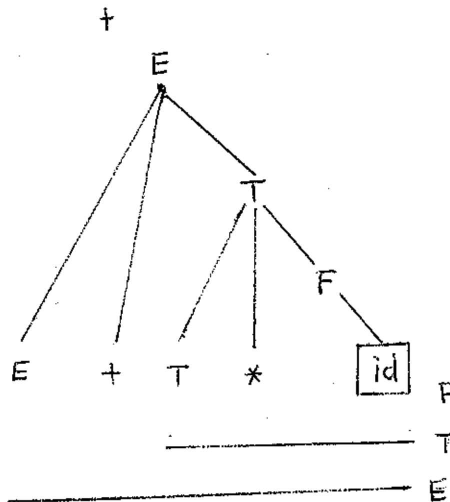
- Bottom-up parsing
- more powerful than LL(k)
- shift/reduce



E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

E → E + T
→ E + T * F
→ E + T * id †
→ E + F * id
→ E + id * id
→ T + id * id
→ F + id * id
→ id + id * id

rightmost derivation in reverse



1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Stack	*Input*	*Action*
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept

Intermediate Code Generation

Syntax Directed Translation Scheme

$id_1 + id_2 * id_3$

Symbol Table

id_1	
id_2	
id_3	

- For each reduction, value associated with RHS becomes associated with LHS
- Temporaries, T_i, is used each time a value is associated with an object

Ex. T → F // T gets F's value

F → (E) // F gets E's value

E → E + T // values associated with E and T are added and

// assigned to E on LHS

Stack	*Input*	*Action*	Code
0	id + id * id $	Shift 5	
0id5	+ id * id $	Reduce 6	T_1 = code(id_1)
0F3	+ id * id $	Reduce 4	T_2 = T_1
0T2	+ id * id $	Reduce 2	T_3 = T_2
0E1	+ id * id $	Shift 6	T_4 = T_3
0E1+6	id * id $	Shift 5	
0E1+6id5	* id $	Reduce 6	T_5 = code(id_2)
0E1+6F3	* id $	Reduce 4	T_6 = T_5
0E1+6T9	* id $	Shift 7	T_7 = T_6
0E1+6T9*7	id $	Shift 5	
0E1+6T9*7id5	$	Reduce 6	T_8 = code(id_3)
0E1+6T9*7F10	$	Reduce 3	T_9 = T_8
0E1+6T9	$	Reduce 1	T_{10} = $T_7 * T_9$
0E1	$	Accept	T_{11} = $T_4 + T_{10}$