

## Concepts of Programming Languages, CSCI 305, Fall 2021 Recursive Descent Parsing – Table Driven, Oct. 11

Hand-execute the recursive descent parsing code using the table driven parsing code on the next page:

```
read A
read B
sum := A + B
write sum
write sum / 2
$$ is sent to indicate the end-of-file'
```

```
terminal = 1 .. number_of_terminals
non_terminal = number_of_terminals + 1 .. number_of_symbols
symbol = 1 .. number_of_symbols
production = 1 .. number_of Productions
```

```
parse_tab : array [non_terminal, terminal] of record
  action : (predict, error)
  prod : production
prod_tab : array [production] of list of symbol
-- these two tables are created by a parser generator tool
```

```
parse_stack : stack of symbol
```

```
parse_stack.push(start_symbol)
```

```
loop
```

```
  expected_sym : symbol := parse_stack.pop()
```

```
  if expected_sym ∈ terminal
```

```
    match(expected_sym)           -- as in Figure 2.17
```

```
    if expected_sym = $$ then return  -- success!
```

```
  else
```

```
    if parse_tab[expected_sym, input_token].action = error
```

```
      parse_error
```

```
    else
```

```
      prediction : production := parse_tab[expected_sym, input_token].prod
```

```
      foreach sym : symbol in reverse prod_tab[prediction]
```

```
        parse_stack.push(sym)
```

Production table: This is the same grammar as used for the recursive descent parser using subroutines. This one, however, this is written without ‘or’s and the productions are numbered, so they can be referred to in the table.

1. program  $\rightarrow$  stmt\_list \$\$
2. stmt\_list  $\rightarrow$  stmt stmt\_list
3. stmt\_list  $\rightarrow \epsilon$
4. stmt  $\rightarrow$  id := expr
5. stmt  $\rightarrow$  read id
6. stmt  $\rightarrow$  write expr
7. expr  $\rightarrow$  term term\_tail
8. term\_tail  $\rightarrow$  add\_op term term\_tail
9. term\_tail  $\rightarrow \epsilon$
10. term  $\rightarrow$  factor factor\_tail
11. factor\_tail  $\rightarrow$  mult\_op factor factor\_tail
12. factor\_tail  $\rightarrow \epsilon$
13. factor  $\rightarrow$  ( expr )
14. factor  $\rightarrow$  id
15. factor  $\rightarrow$  number
16. add\_op  $\rightarrow$  +
17. add\_op  $\rightarrow$  -
18. mult\_op  $\rightarrow$  \*
19. mult\_op  $\rightarrow$  /

Parse table for above grammar:

Top-of-stack nonterminal	Current input token											
	id	number	read	write	:=	(	)	+	-	*	/	\$\$
<i>program</i>	1	-	1	1	-	-	-	-	-	-	-	1
<i>stmt_list</i>	2	-	2	2	-	-	-	-	-	-	-	3
<i>stmt</i>	4	-	5	6	-	-	-	-	-	-	-	-
<i>expr</i>	7	7	-	-	-	7	-	-	-	-	-	-
<i>term_tail</i>	9	-	9	9	-	-	9	8	8	-	-	9
<i>term</i>	10	10	-	-	-	10	-	-	-	-	-	-
<i>factor_tail</i>	12	-	12	12	-	-	12	12	12	11	11	12
<i>factor</i>	14	15	-	-	-	13	-	-	-	-	-	-
<i>add_op</i>	-	-	-	-	-	-	-	16	17	-	-	-
<i>mult_op</i>	-	-	-	-	-	-	-	-	-	18	19	-