

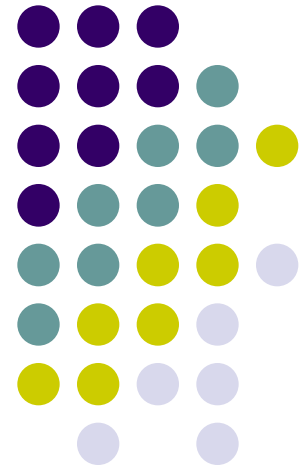
CS 460

Programming Languages

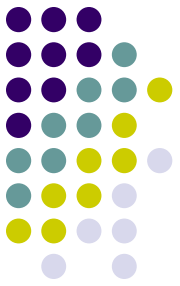
Fall 2021

Dr. Watts

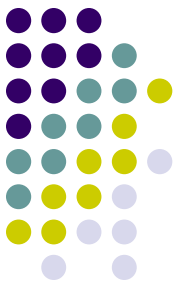
(30 October 2023)



Assignments

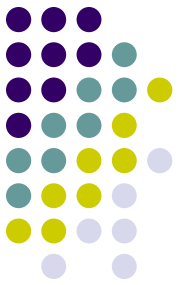


- Exercise 2
 - Script running so that your groups can improve their testing techniques
- Exercise 3
 - Posted – let me know if you see typos
 - Part 1 due Wednesday
 - Please bring a hard copy to class
- Project 2
 - Coming soon (Wednesday)
- Exercise 5
 - Preliminary exercise will be posted this week



What is Syntactic Analysis?

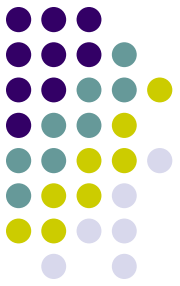
- Also known as Parsing
- Determining if the order of the tokens generated for the lexemes of the input are in a legal order according to some grammar
- Creation of a parse tree
 - Explicit or implicit
- Error recovery
 - When an error is detected, the parser must get back to a normal state and continue analysis of the input
- Basis for translation



VS Semantic Analysis

- The meaning of the expressions, statements and program units.
- Static semantics
 - At compile time
- Dynamic semantics
 - At run time
- Attribute grammars
- Denotational semantics

LL(1) Grammar for a Small Programming Language



$T = \{\text{begin, end, ;, =, A, B, C, +, *}\}$

$N = \{\langle \text{program} \rangle, \langle \text{stmt_list} \rangle, \langle \text{stmt} \rangle, \langle \text{var} \rangle, \langle \text{stmt_tail} \rangle, \langle \text{expression} \rangle, \langle \text{expr_tail} \rangle\}$

Start = $\langle \text{program} \rangle$

P =

- {
- 1. $\langle \text{program} \rangle \rightarrow \text{begin } \langle \text{stmt_list} \rangle \text{ end}$
- 2. $\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle \langle \text{stmt_tail} \rangle$
- 3. $\langle \text{stmt_tail} \rangle \rightarrow ; \langle \text{stmt_list} \rangle$
- 4. $\langle \text{stmt_tail} \rangle \rightarrow \lambda$
- 5. $\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$
- 6. $\langle \text{var} \rangle \rightarrow \mathbf{A}$
- 7. $\langle \text{var} \rangle \rightarrow \mathbf{B}$
- 8. $\langle \text{var} \rangle \rightarrow \mathbf{C}$
- 9. $\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
- 10. $\langle \text{expr_tail} \rangle \rightarrow + \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
- 11. $\langle \text{expr_tail} \rangle \rightarrow * \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
- 12. $\langle \text{expr_tail} \rangle \rightarrow \lambda$
- }

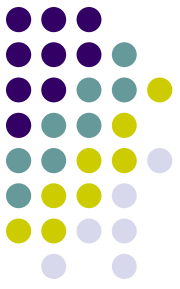
Example 1:

```
begin
    A = B + C;
    C = A * B
end
```

Example 2:

```
begin
    A = B + A * C;
    C = A * B;
end
```

Parsing of Example 1



From main parsing routine, call <program> function

From <program>, match **begin**; call <stmt_list> function

From <stmt_list>, see **A** call <stmt> function

From <stmt>, see **A** call <var> function

From <var>, match **A**; return

From <stmt>, match =; see **B** call <expression> function

From <expression>, see **B** call <var> function

From <var>, match **B**; return

From <expression>, see + call <expr_tail> function

From <expr_tail>, match +; see **C** call <var> function

From <var>, match **C**; return

From <expr_tail>, see ; call <expr_tail>

From <expr_tail>, see ; return

From <expr_tail>, return

From <expression>, return

...

From <program>, match end; return

From main parsing routine, print error count and terminate

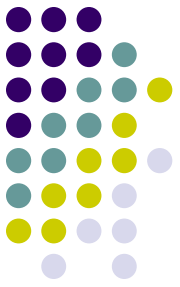
Recursive Descent Parser for a Small Programming Language



1) `<program>` → **begin** `<stmt_list>` **end**

```
program ()
{
    if (current_token == begin)
    { // rule 1
        get_next_token;
        call stmt_list;
        if (current_token == end)
            get_next_token;
        else
            call error_routine;
    }
    else
        call error_routine;
    return;
}
```

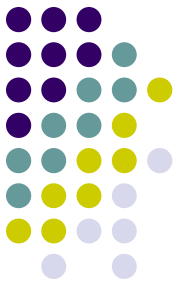
Recursive Descent Parser for a Small Programming Language



2) $\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle \langle \text{stmt_tail} \rangle$

```
stmt_list ()
{ // rule 2
    call stmt;
    call stmt_tail;
    return;
}
```


Recursive Descent Parser for a Small Programming Language

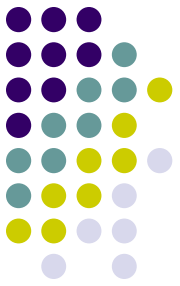


3) $\langle \text{stmt_tail} \rangle \rightarrow ; \langle \text{stmt_list} \rangle$

4) $\langle \text{stmt_tail} \rangle \rightarrow \lambda$

```
stmt_tail ()
{
    if (current_token == ;)
    { // rule 3
        get_next_token;
        call stmt_list;
    }
    else if (current_token == end)
    { // rule 4
    }
    else
        call error_routine;
    return;
}
```

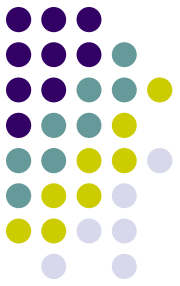
Recursive Descent Parser for a Small Programming Language



5) `<stmt> → <var> = <expression>`

```
stmt ()
{ // rule 5
  call var;
  if (current_token == =)
  {
    get_next_token;
    call expression;
  }
  else
    call error_routine;
  return;
}
```

Recursive Descent Parser for a Small Programming Language



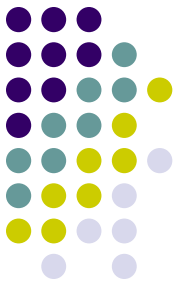
6) <var> → A

7) <var> → B

8) <var> → C

```
var ()
{ // rule 5
    if (current_token == A)
    { // rule 6
        get_next_token;
    }
    else if (current_token == B)
    { // rule 7
        get_next_token;
    }
    else if (current_token == C)
    { // rule 8
        get_next_token;
    }
    else
        call error_routine;
    return;
}
```

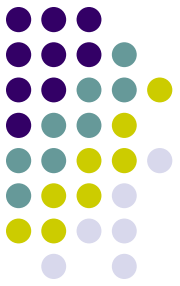
Recursive Descent Parser for a Small Programming Language



9) $\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle \langle \text{expr_tail} \rangle$

```
expression ()
{ // rule 9
    call var;
    call expr_tail;
    return;
}
```

Recursive Descent Parser for a Small Programming Language

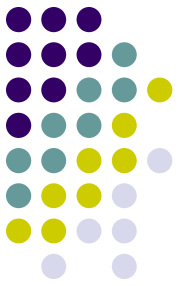


- 10) `<expr_tail> → + <var><expr_tail>`
- 11) `<expr_tail> → * <var><expr_tail>`
- 12) `<expr_tail> → λ`

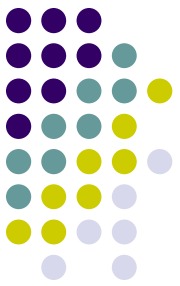
```
expr_tail ()
{
    if (current_token == +)
    { // rule 10
        get_next_token;
        call var;
        call expr_tail;
    }
    else if (current_token == *)
    { // rule 11
        get_next_token;
        call var;
        call expr_tail;
    }
    else if (current_token == ; or current_token == end)
    { // rule 12
    }
    else
        call error_routine;
    return;
}
```

Context Free Grammar

Definition



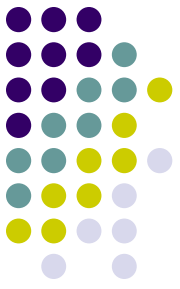
- Given a Context Free Grammar of the form:
 - Terminals = $\{T_1, T_2, T_3, \dots\}$
 - Non-terminals = $\{\langle nt_1 \rangle, \langle nt_2 \rangle, \langle nt_3 \rangle, \dots\}$
 - A Start symbol from the set of non-terminals
 - A set of Production rules of the form
 $\langle nt_i \rangle \rightarrow$ string of T and $\langle nt \rangle$ symbols



First and Follow Sets

- **Firsts**
 - A terminal symbol T_i is a member of the First Set of non-terminal symbol $\langle nt_j \rangle$ if T_i can become the first terminal symbol in a complete expansion of $\langle nt_j \rangle$.
- **Follows**
 - A terminal symbol T_i is a member of the Follow Set of non-terminal symbol $\langle nt_j \rangle$ if T_i can become the first terminal symbol immediately following a complete expansion of $\langle nt_j \rangle$.

Short Project Grammar



Character Sets

α = upper or lower alphabetic characters

η = digits 0 to 9

Θ = all typeable characters

Lexeme Regular Expression

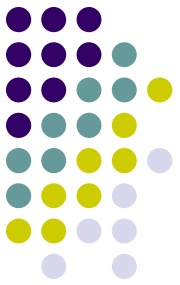
```
define | ( | ) |  $\alpha(\alpha|\eta|_)*$  | (+|-|\lambda) ( $\eta^+$  |  $\eta^*.\eta^+$  |  $\eta^+.\eta^*$  |  $\eta^+/\eta^+$ ) | " $\Theta^*$ "  
| #f | #t | display | newline
```

$T = \{\text{DEFINE_T, LPAREN_T, RPAREN_T, IDENT_T, NUMLIT_T, STRLIT_T, FALSE_T, TRUE_T, DISPLAY_T, NEWLINE_T, EOF_T}\};$

$NT = \{\langle\text{program}\rangle, \langle\text{more_defines}\rangle, \langle\text{define}\rangle, \langle\text{stmt_list}\rangle, \langle\text{stmt}\rangle, \langle\text{literal}\rangle, \langle\text{logical_lit}\rangle, \langle\text{param_list}\rangle, \langle\text{else_part}\rangle, \langle\text{action}\rangle\}$

$S = \langle\text{program}\rangle$

Short Project Grammar



P = {

1. `<program>` -> LPAREN_T `<define>` LPAREN_T `<more_defines>` EOF_T
2. `<more_defines>` -> `<define>` LPAREN_T `<more_defines>`
3. `<more_defines>` -> IDENT_T `<stmt_list>` RPAREN_T
4. `<define>` -> DEFINE_T LPAREN_T IDENT_T `<param_list>` RPAREN_T `<stmt>` `<stmt_list>`
RPAREN_T
5. `<stmt_list>` -> `<stmt>` `<stmt_list>`
6. `<stmt_list>` -> λ
7. `<stmt>` -> `<literal>`
8. `<stmt>` -> IDENT_T
9. `<stmt>` -> LPAREN_T `<action>` RPAREN_T
10. `<literal>` -> NUMLIT_T
11. `<literal>` -> STRLIT_T
12. `<literal>` -> `<logical_lit>`
13. `<logical_lit>` -> TRUE_T
14. `<logical_lit>` -> FALSE_T
15. `<param_list>` -> IDENT_T `<param_list>`
16. `<param_list>` -> λ
17. `<action>` -> IDENT_T `<stmt_list>`
18. `<action>` -> DISPLAY_T `<stmt>`
19. `<action>` -> NEWLINE_T

}

Short Project Grammar

- Write a PL460 program that
 - Uses all possible lexemes (tokens)
 - Uses all 19 production rules
 - Is Syntactically correct

- Submit as Team[A-Z].pl460

