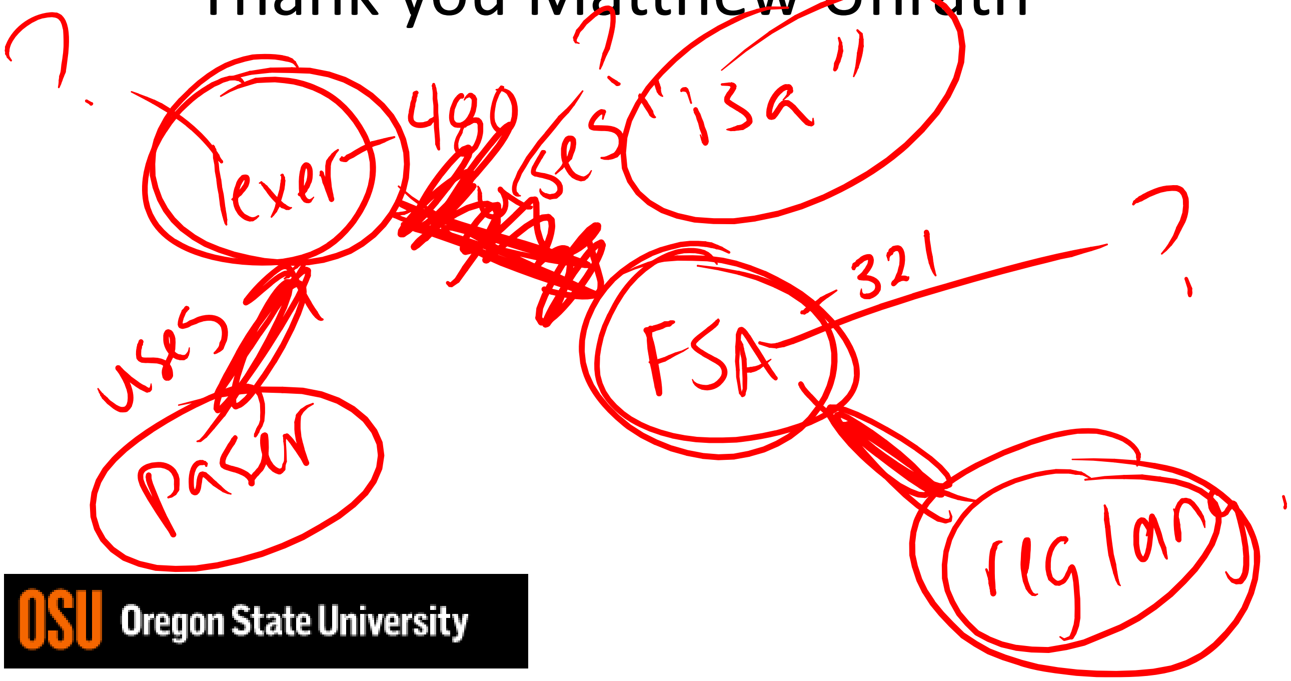
CS480 Translators

Top-down Parsing

Chap. 4

Why does our thinking fail?

- Have you seen the LEGO movie?
  - GO!!!
- http://en.wikipedia.org/wiki/Candle_problem
  - Thank you Matthew Unrath



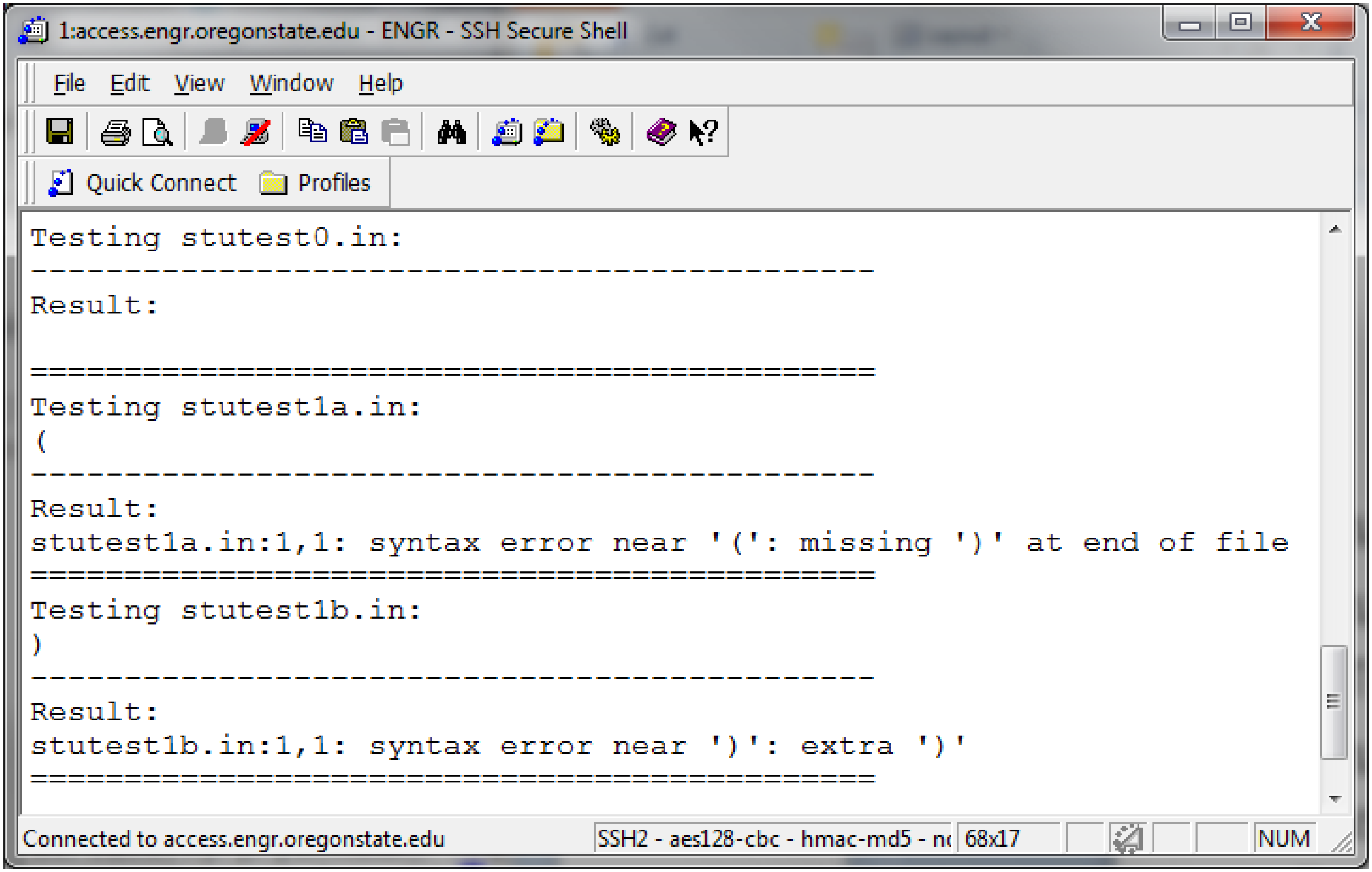
Milestone 3

build your tree

verify syntax matches the grammar

- What is the purpose of this milestone?
- What does this mean?
  - The parser produces a list encoding the input.
- What is accepted/not accepted by the grammar thus far? Why/Why not?
- Due date??? → Monday!!!

Example Past Input/Output

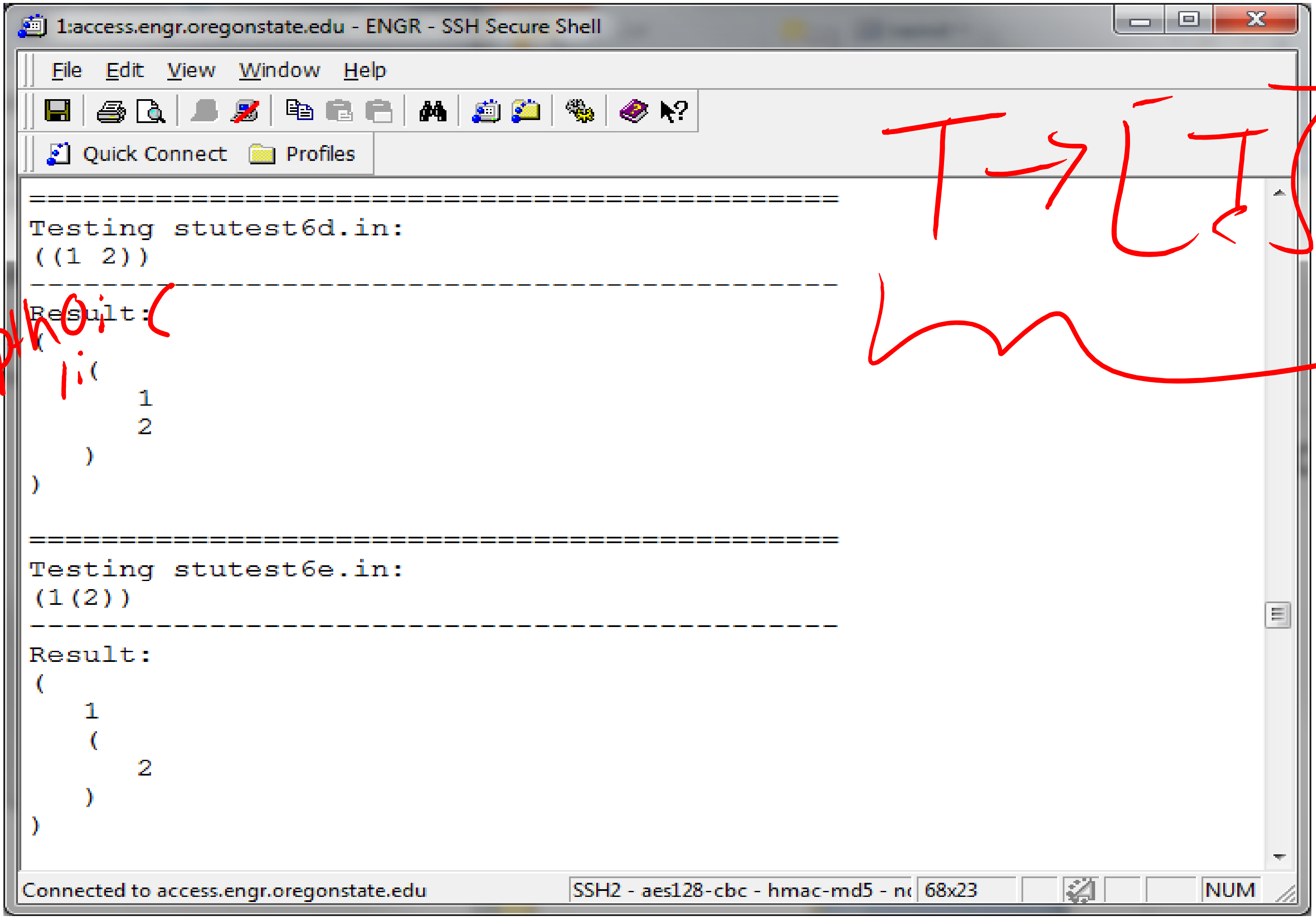


1:access.engr.oregonstate.edu - ENGR - SSH Secure Shell
File Edit View Window Help
Quick Connect
Profiles
Connected to access.engr.oregonstate.edu
SSH2 - aes128-cbc - hmac-md5 - nc
68x17
NUM

```
Testing stutest0.in:
---------------------------------------------
Result:

=============================================
Testing stutest1a.in:
(
---------------------------------------------
Result:
stutest1a.in:1,1: syntax error near '(': missing ')' at end of file
=============================================
Testing stutest1b.in:
)
---------------------------------------------
Result:
stutest1b.in:1,1: syntax error near ')': extra ')'
=============================================
```

OSU Oregon State University

Example Input/Output

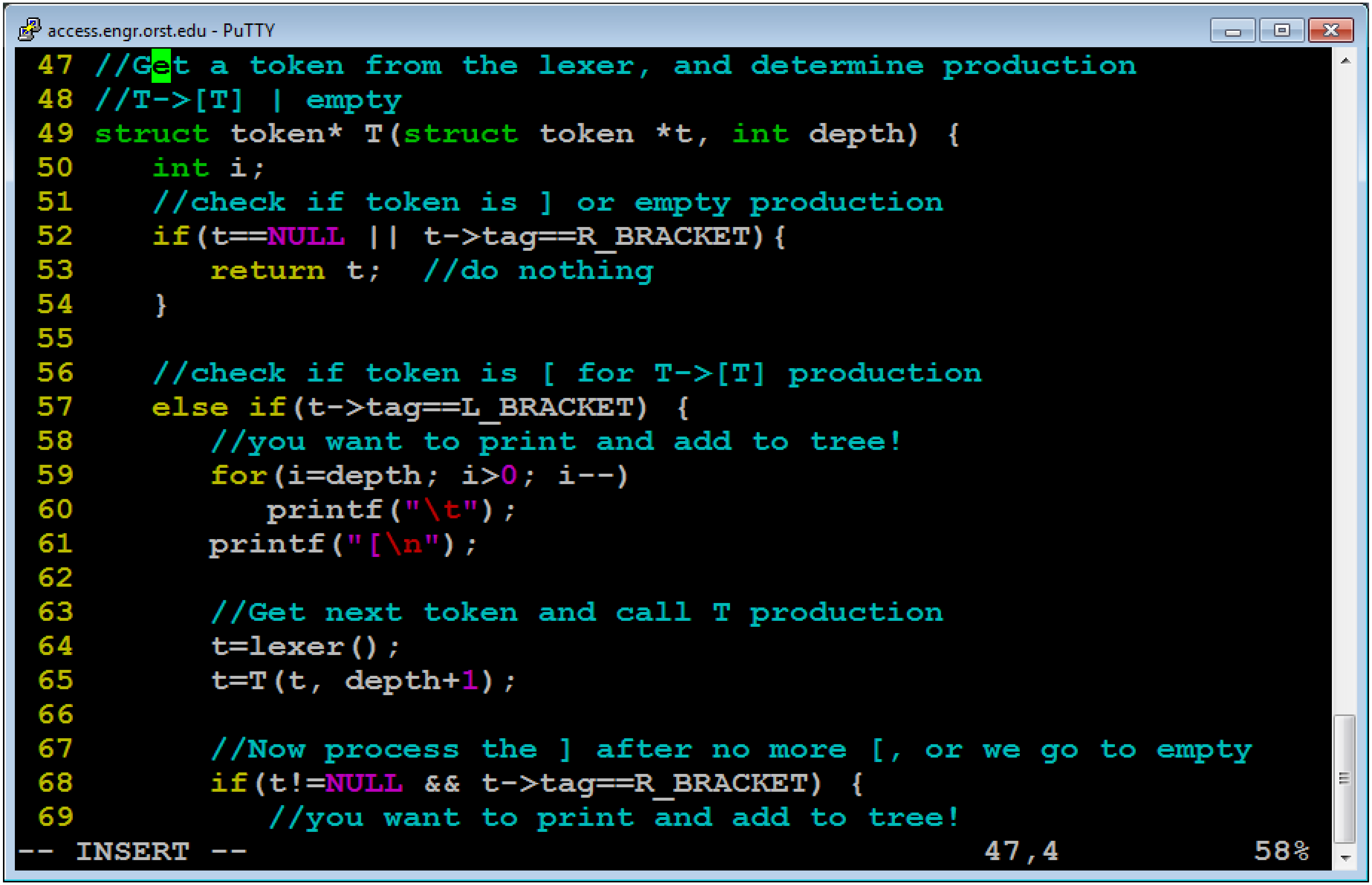

1:access.engr.oregonstate.edu - ENGR - SSH Secure Shell
File Edit View Window Help
Quick Connect
Profiles
Connected to access.engr.oregonstate.edu
SSH2 - aes128-cbc - hmac-md5 - n
68x23
NUM

```
===============================================
Testing stutest6d.in:
((1 2))
-----------------------------------------------
Result:
(
    (
        1
        2
    )
)

===============================================
Testing stutest6e.in:
(1(2))
-----------------------------------------------
Result:
(
    1
    (
        2
    )
)
```

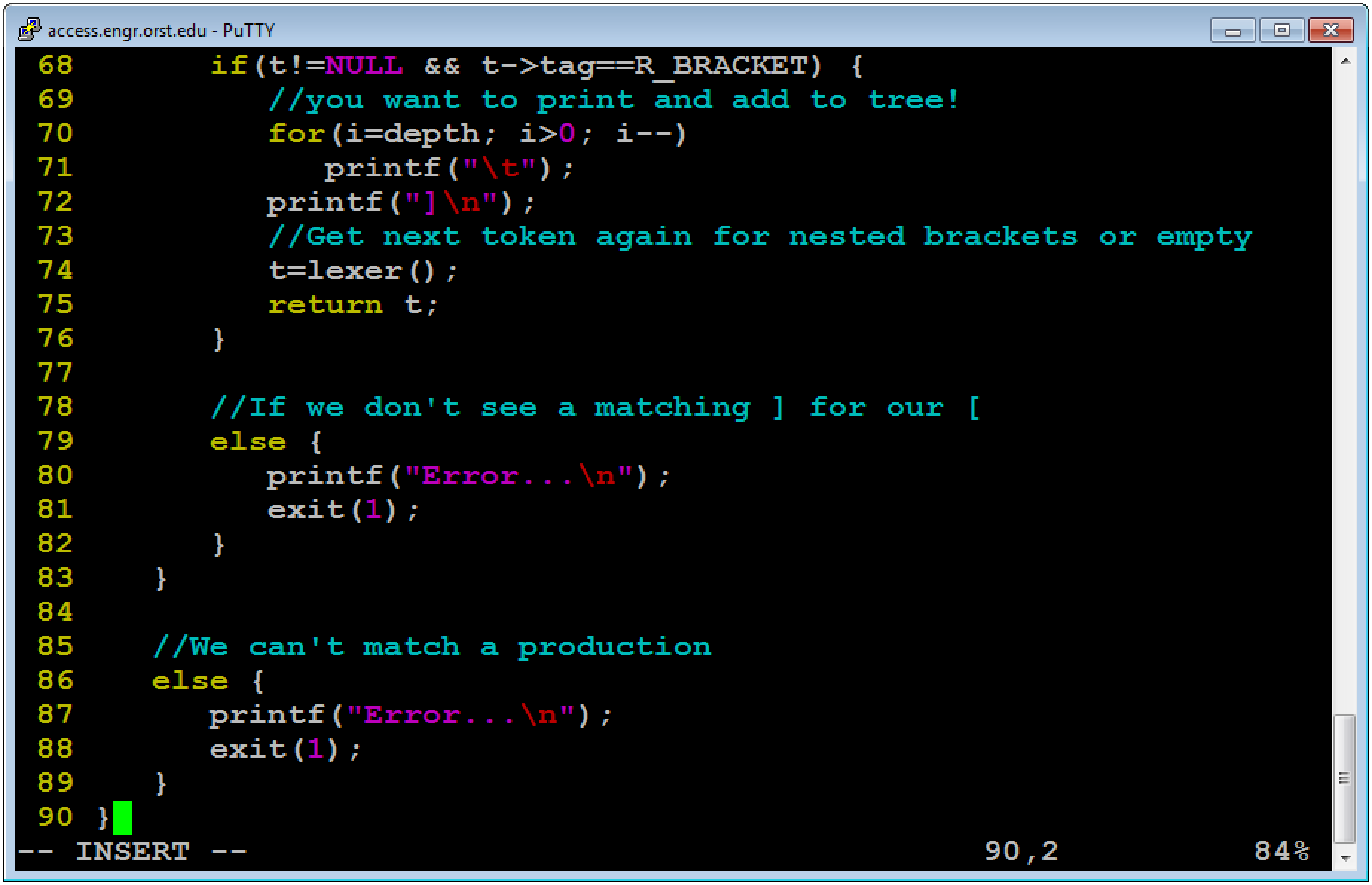
depth 0: (

1: (

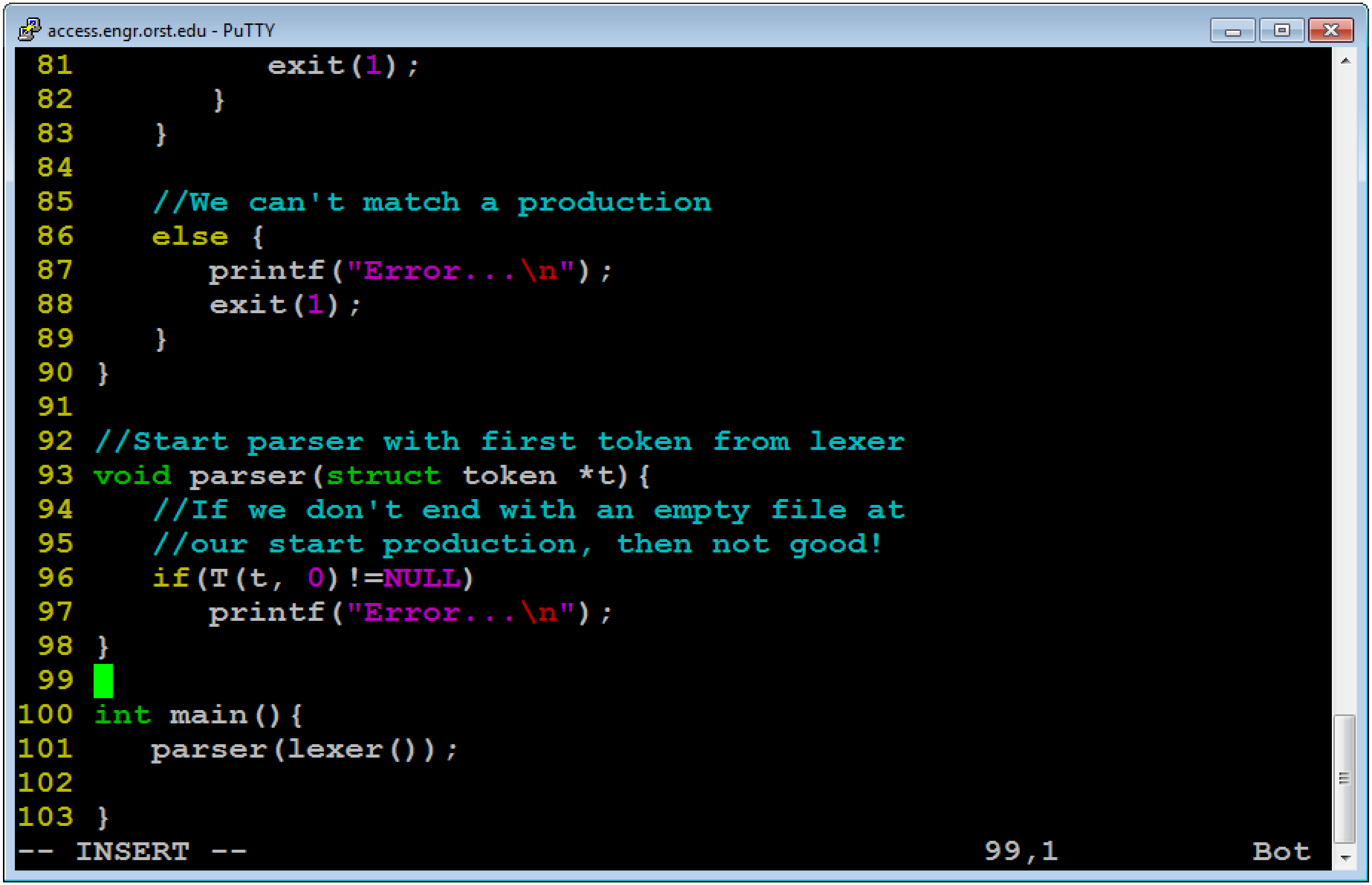
T → [T] | ε



```
//Get a token from the lexer, and determine production
//T->[T] | empty
struct token* T(struct token *t, int depth) {
   int i;
   //check if token is ] or empty production
   if(t==NULL || t->tag==R_BRACKET){
      return t;  //do nothing
   }

   //check if token is [ for T->[T] production
   else if(t->tag==L_BRACKET) {
      //you want to print and add to tree!
      for(i=depth; i>0; i--)
         printf("\t");
      printf("[\n");

      //Get next token and call T production
      t=lexer();
      t=T(t, depth+1);

      //Now process the ] after no more [, or we go to empty
      if(t!=NULL && t->tag==R_BRACKET) {
         //you want to print and add to tree!
-- INSERT --                                            47,4          58%
```



```
      if(t!=NULL && t->tag==R_BRACKET) {
         //you want to print and add to tree!
         for(i=depth; i>0; i--)
            printf("\t");
         printf("]\n");
         //Get next token again for nested brackets or empty
         t=lexer();
         return t;
      }

      //If we don't see a matching ] for our [
      else {
         printf("Error...\n");
         exit(1);
      }
   }

   //We can't match a production
   else {
      printf("Error...\n");
      exit(1);
   }
}
-- INSERT --                                         90,2          84%
```



```
         exit(1);
      }
   }

   //We can't match a production
   else {
      printf("Error...\n");
      exit(1);
   }
}

//Start parser with first token from lexer
void parser(struct token *t){
   //If we don't end with an empty file at
   //our start production, then not good!
   if(T(t, 0)!=NULL)
      printf("Error...\n");
}

int main(){
   parser(lexer());

}
-- INSERT --                                            99,1          Bot
```

Defining an LL Grammar

top-down

- Need two definitions:
- **First** and **Follow**

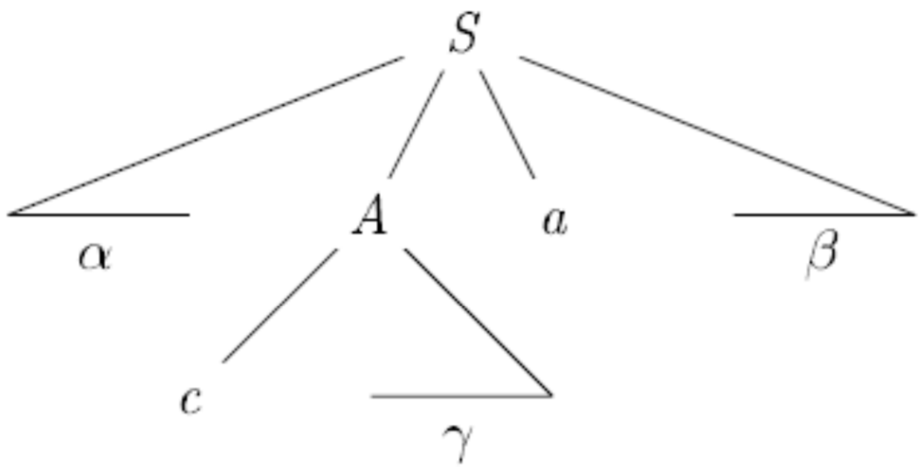


Figure 4.15: Terminal c is in FIRST(A) and a is in FOLLOW(A)

Example First and Follow

E-> T E’

E’-> + T E’ | ε

T-> F T’

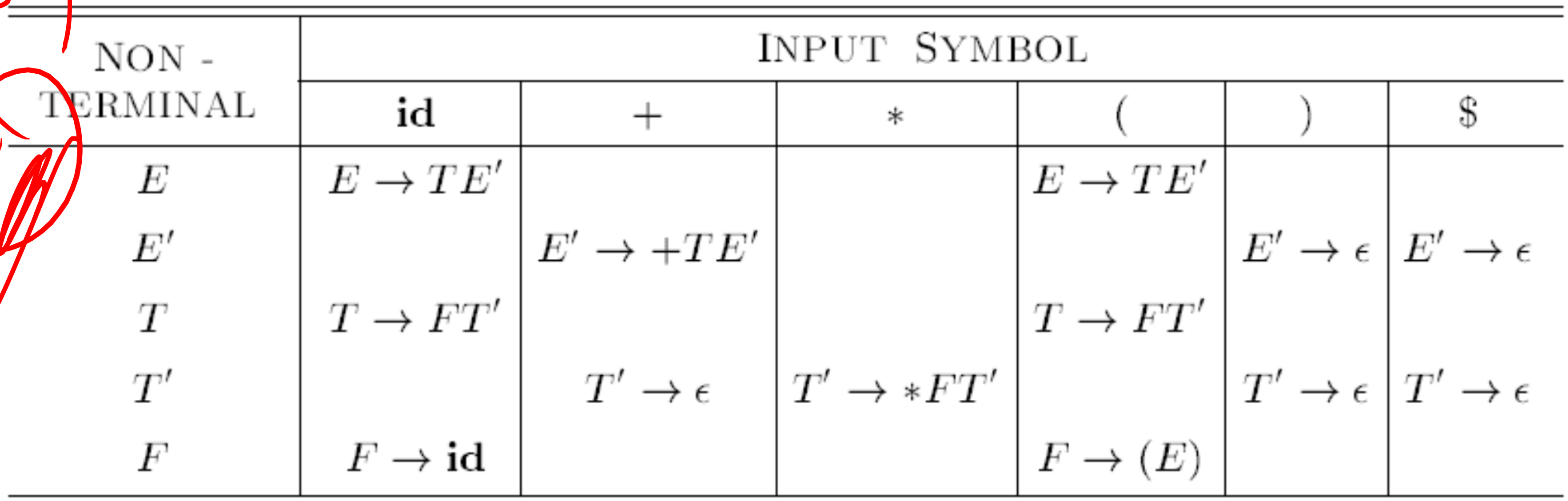
*T’-> * F T’* | ε

F-> (*E*) | **id**

- First(E), First(E’), First(T), First(T’), First(F)?
- Follow(E), Follow(E’), Follow(T), Follow(T’), Follow(F)?

Predictive Parsing Table

- For each production A->α in the grammar:
  - For each terminal **a** in First(α), add A->α to M[A, a]
  - If ε is in First(α), then for each terminal **b** in Follow(A), add A->α to M[A, b]. If $ is in Follow(A), add A->α to M[A, $] as well



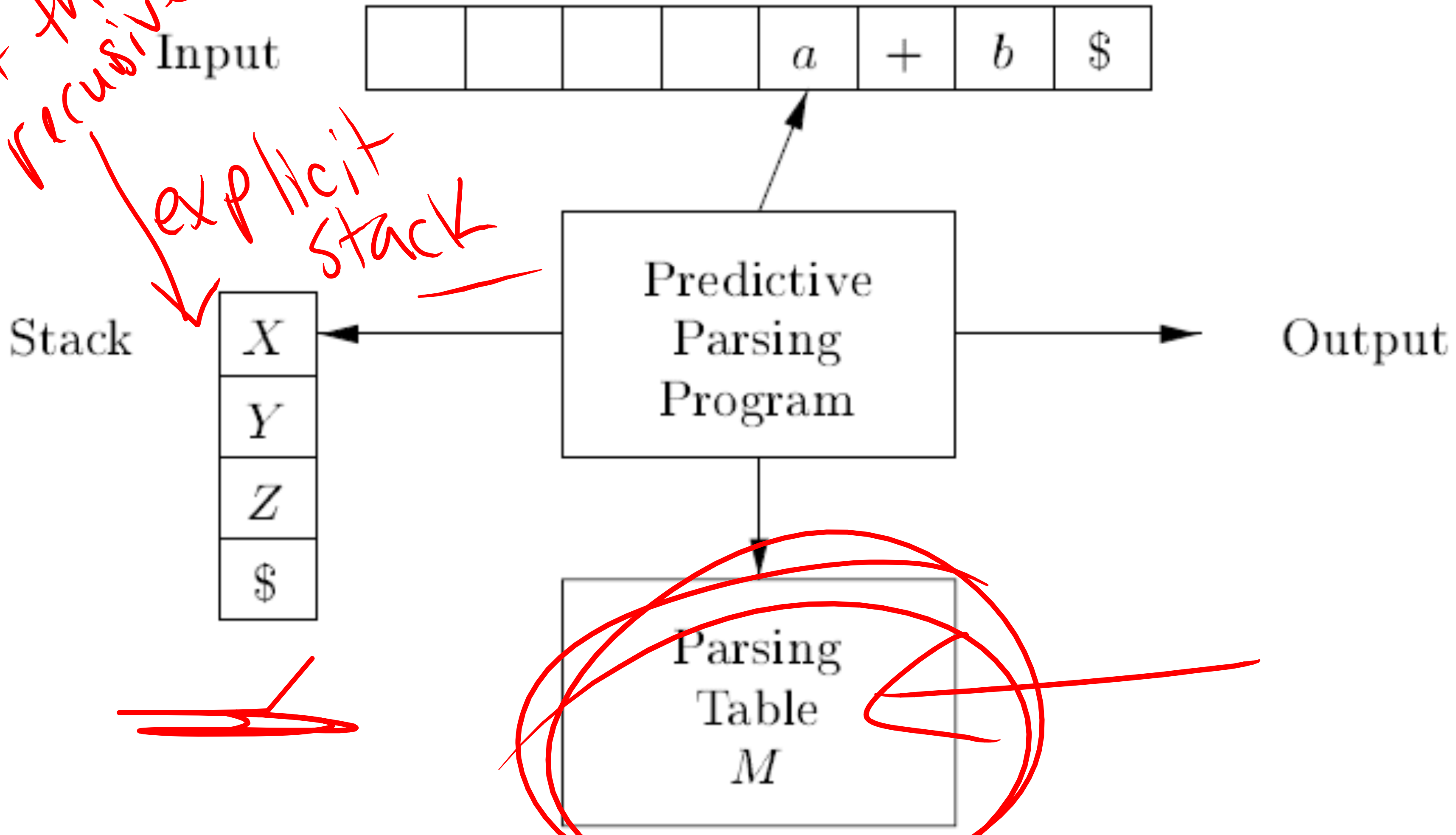
NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

top-down

Table-Driven Predictive Parsing

opposite of that recursive

explicit stack



Input

a + b $

Stack

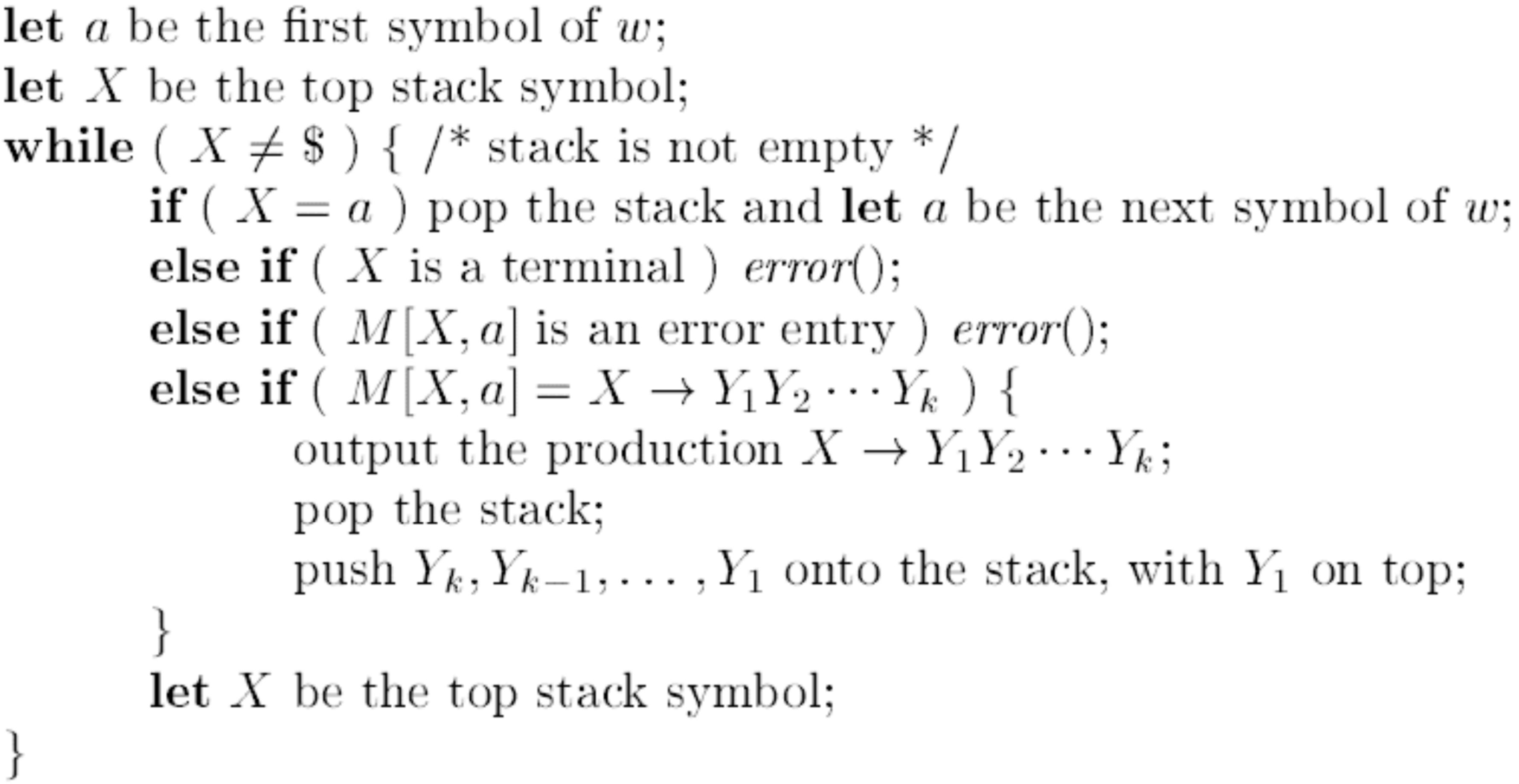
X
Y
Z
$

Predictive Parsing Program

Output

Parsing Table M

Nonrecursive Predictive Parsing



```
let $a$ be the first symbol of $w$;
let $X$ be the top stack symbol;
while ( $X \neq \$$ ) { /* stack is not empty */
        if ( $X = a$ ) pop the stack and let $a$ be the next symbol of $w$;
        else if ( $X$ is a terminal ) $error()$;
        else if ( $M[X, a]$ is an error entry ) $error()$;
        else if ( $M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$ ) {
                output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
                pop the stack;
                push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with $Y_1$ on top;
        }
        let $X$ be the top stack symbol;
}
```

Start

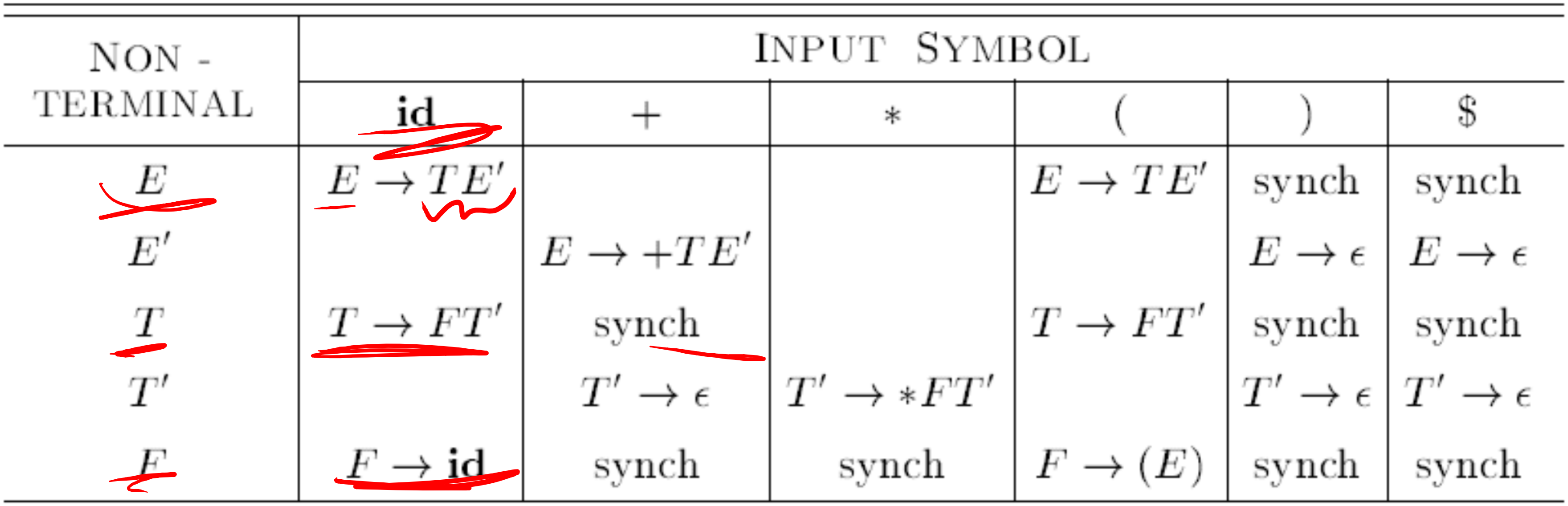
replaced & don't consume

Matched	Stack	Input	Action
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id}\ +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id}\ +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id}\ +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id}\ *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id}\ *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Figure 4.21: Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Error Recovery

- Use Follow sets for synch tokens
- Specify rules for synch tokens



NON - TERMINAL	INPUT SYMBOL					
	id	$+$	$*$	(	)	\$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	synch	synch	$F \to (E)$	synch	synch

Figure 4.22: Synchronizing tokens added to the parsing table of Fig. 4.17

STACK	INPUT	REMARK
$E\ \$$	$)\ \mathbf{id} * + \mathbf{id}\ \$$	error, skip)
$E\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	**id** is in $\text{FIRST}(E)$
$TE'\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$FT'E'\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$\mathbf{id}\ T'E'\$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$T'E'\ \$$	$* + \mathbf{id}\ \$$	
$* FT'E'\ \$$	$* + \mathbf{id}\ \$$	
$FT'E'\ \$$	$+ \mathbf{id}\ \$$	error, $M[F, +] =$ synch
$T'E'\ \$$	$+ \mathbf{id}\ \$$	F has been popped
$E'\ \$$	$+ \mathbf{id}\ \$$	
$+ TE'\ \$$	$+ \mathbf{id}\ \$$	
$TE'\ \$$	$\mathbf{id}\ \$$	
$FT'E'\ \$$	$\mathbf{id}\ \$$	
$\mathbf{id}\ T'E'\ \$$	$\mathbf{id}\ \$$	
$T'E'\ \$$	$\$$	
$E'\ \$$	$\$$	
$\$$	$\$$	

Figure 4.23: Parsing and error recovery moves made by a predictive parser

What happens here?

First $S = \{i, a\}$ $S' = \{e, \varepsilon\}$ $E = \{b\}$

S -> **i***E***t***SS'* | **a**

S' -> **e***S* | ε

E -> **b**

Follow $S = \{\$, e\}$ $S' = \{\$, e\}$ $E = \{t\}$

start

S' first

\$

NON - TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	\$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

we do not have an LL(1) grammar

Quiz #5

- Eliminate left recursion from the S production in IBTL.
- For each grammar below, calculate First and Follow sets for each nonterminal and construct a parsing table.

  (a) S -> 0 S'

  S' -> S 1 | 1

  (b) S -> (S) S | ε
- What do we need to do to our grammar to use top-down parsing? Is it LL(1), LL(2), etc.?