ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 6a

Syntax Analysis

Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy

Top-Down Parsing

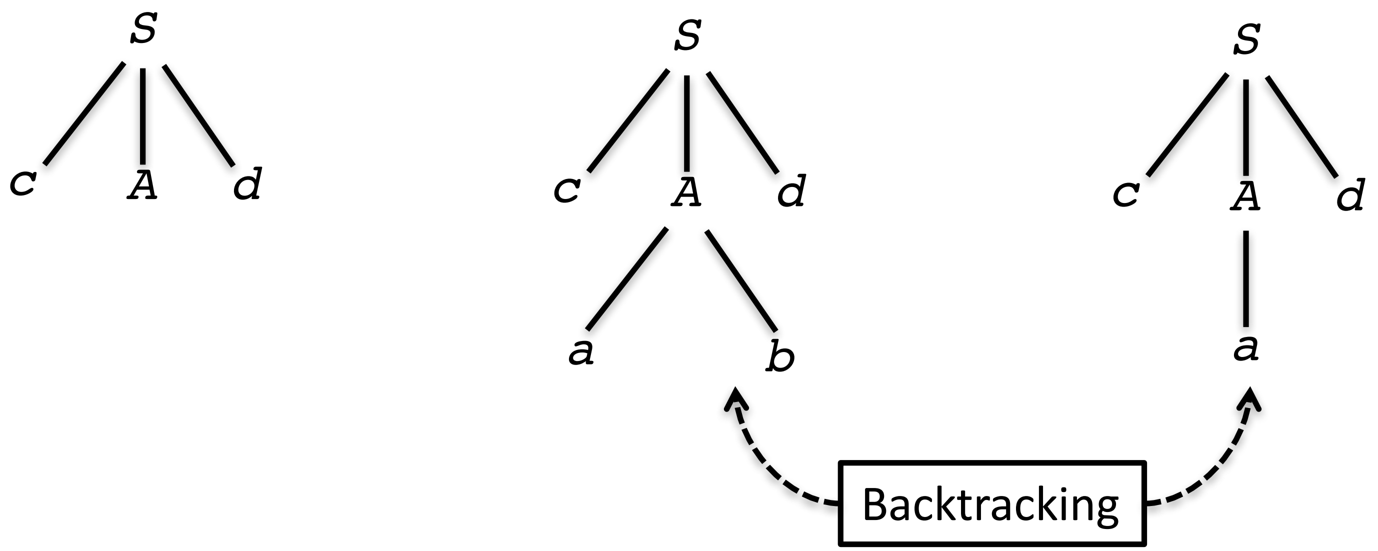
- Can be viewed as an attempt to find a leftmost derivation for an input string
- May involve **backtracking**
  - Use left-factoring to remove backtracking
- Two types of parsers
  - **Non recursive predictive parsing** is table driven
  - **Recursive-descent parsing**, where a procedure is associated with each non-terminal symbol

Backtracking

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Consider the input string $w = cad$

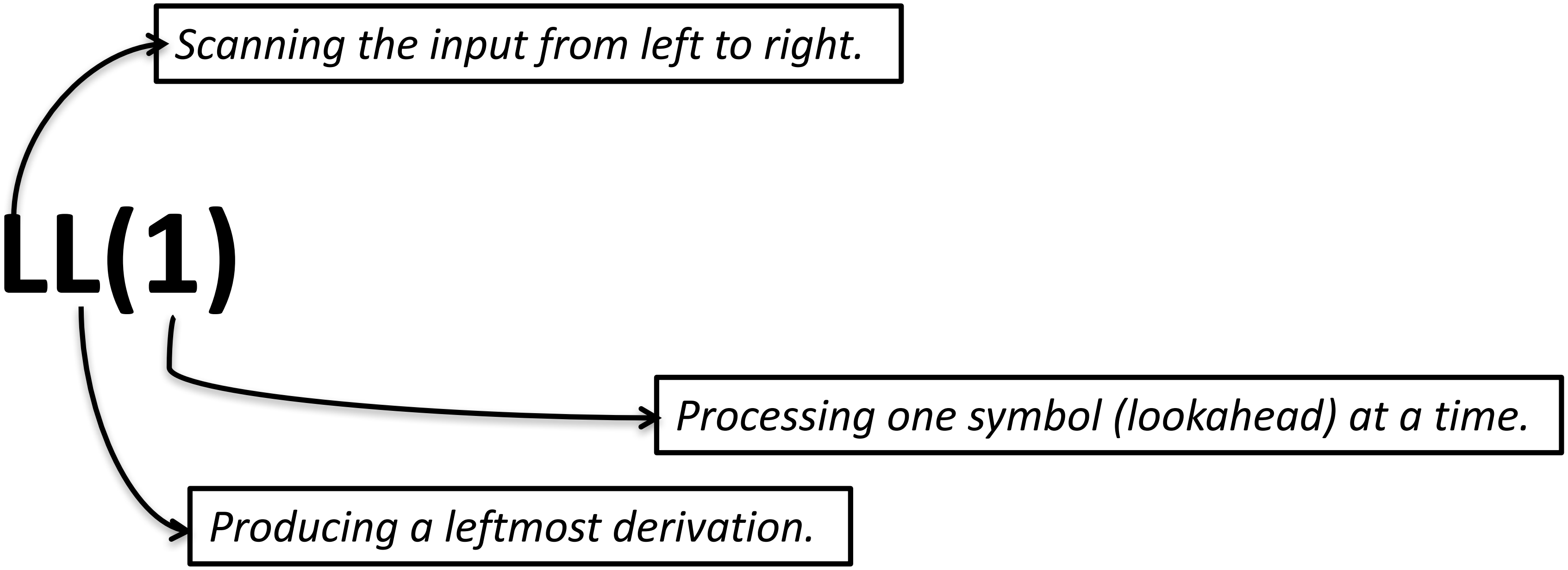


Backtracking

Predictive parsing

- If we carefully re-write the grammar by eliminating left recursion and by left factoring the grammar, then the result can be parsed with no backtracking

```
stmt → if expr then stmt else stmt
      | while expr do stmt
      | begin stmt_list end
```



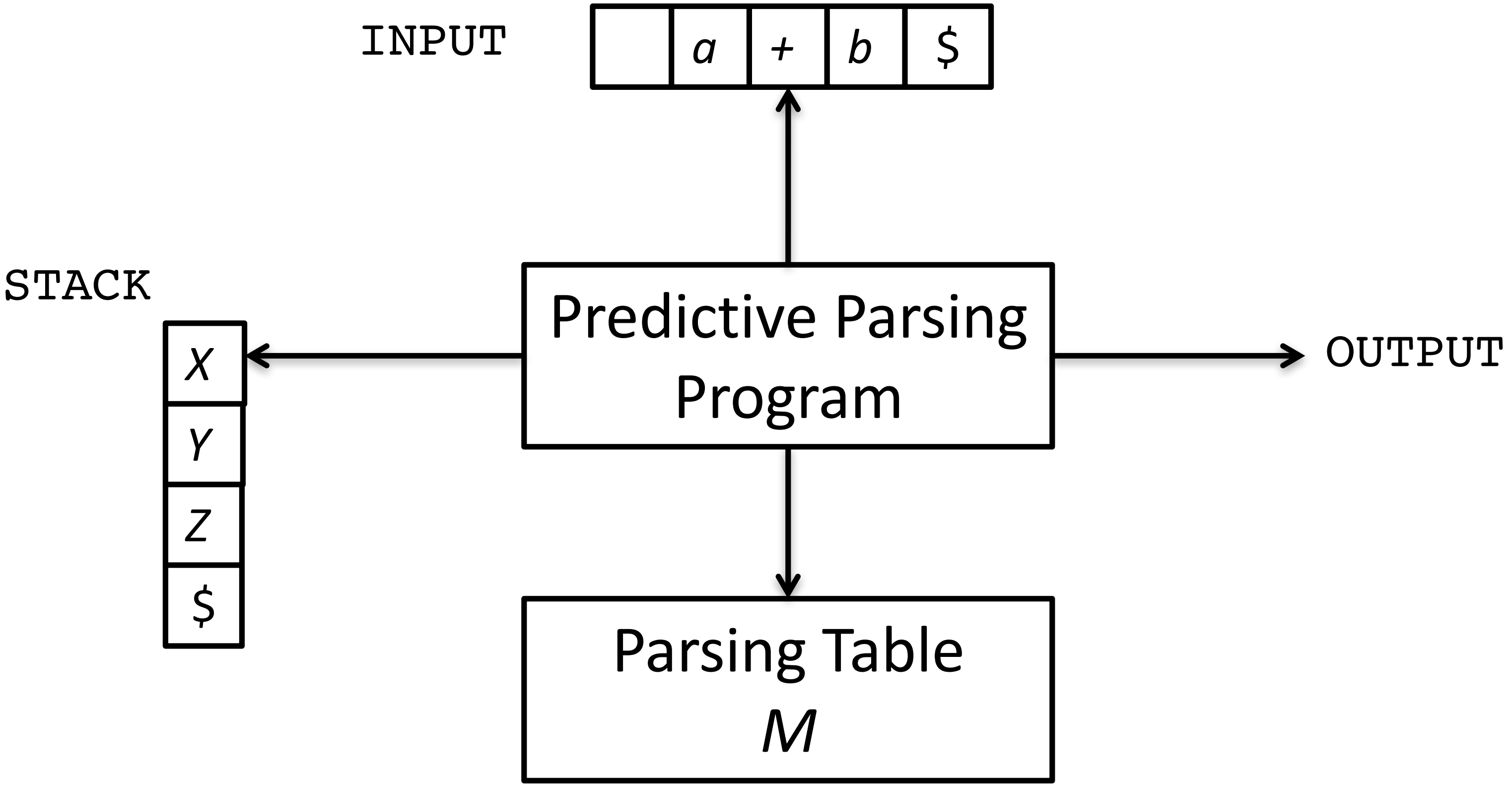
LL(1)

Scanning the input from left to right.

Processing one symbol (lookahead) at a time.

Producing a leftmost derivation.

Non recursive predictive parser



$: end symbol
X, *Y*, *Z*: non-terminals or terminals

Parsing Table *M*

NONTERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \textbf{id}$			$F \rightarrow (E)$		

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid \textbf{id}$

Stack for

id + id*id

STACK	INPUT	OUTPUT
$E	**id + id * id**$	
$E′T	**id + id * id**$	*E*→*TE′*
$E′T′F	**id + id * id**$	*T* →*FT′*
$E′T′**id**	**id + id * id**$	*F*→**id**
$E′T′	**+ id * id**$	
$E′	**+ id * id**$	*T′*→ε
$E′T+	**+ id * id**$	*E′*→+*TE′*
$E′T	**id * id**$	
$E′T′F	**id * id**$	*T* →*FT′*
$E′T′**id**	**id * id**$	*F*→**id**
$E′T′	*** id**$	
$E′T′F*	*** id**$	*T′*→**FT′*
$E′T′F	**id**$	
$E′T′**id**	**id**$	*F*→**id**
$E′T′	$	
$E′	$	*T′*→ε
$	$	*E′*→ε

FIRST()

If a is any string of grammar symbols, let FIRST(a) be the set of terminals that begin the strings derived from a.

- Rule 1
  - If X is a terminal then FIRST(X) = {X}
- Rule 2
  - If $X \rightarrow \varepsilon$, then we add {ε} to FIRST(X).
- Rule 3
  - If X is nonterminal, and $X \rightarrow Y_1Y_2...Y_m$ then add FIRST($Y_1Y_2...Y_m$) to X
    - FIRST(Y_1) if FIRST(Y_1) does not contain ε
    - FIRST(Y_1) and FIRST($Y_2...Y_m$) (excluding ε) if FIRST(Y_1) contains ε
    - If FIRST(Y_m) contains ε, then add also ε

Example

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid \textbf{id}$

FIRST(+) ={+}

FIRST(*) ={*}

FIRST(() ={(}

FIRST()) ={)}

FIRST(**id**)={**id**}

FIRST(E) ={(,id}

FIRST(T) ={(,id}

FIRST(F) ={(,id}

FIRST(E′)={+,ε}

FIRST(T′)={*,ε}

FOLLOW( )

FOLLOW(*A*), for a nonterminal *A*, is the set of terminals a that can appear immediately to the right of *A*.

- Rule 1
  - Add $ to the follow set of the starting symbol.
- Rule 2
  - If A→aBb, then FOLLOW(B) contains at least FIRST(b) (with ε excluded)
- Rule 3
  - If (A→aB) or (A→aBb and ε is in FIRST(b)) then FOLLOW(A) contains at least FOLLOW(B)

Example

```
E   →  TE′
E′  →  +TE′|ε
T   →  FT′
T′  →  *FT′|ε
F   →  (E)  |id
```

```
FOLLOW(E) ={),$}
FOLLOW(E′)={),$}
FOLLOW(T) =(+,),$)
FOLLOW(T′)=(+,),$)
FOLLOW(F) =(*,+,),$)
```

Constructing the parsing table

1. For each $A \rightarrow a$ production of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(a), add $A \rightarrow a$ to $M[A, a]$.
3. Ifε is in FIRST(a), add $A \rightarrow a$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ε is in FIRST(a) and \$ is in FOLLOW(A), add $A \rightarrow a$ to M[A, \$].
4. Make each undefined entry of *M* be **error**.

Error Recovery with Synchronizing Symbols

NONTERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	*E*→*TE*′			*E*→*TE*′	synch	synch
E′		*E*′→+*TE*′			*E*′→ε	*E*′→ε
T		synch		*T* →*FT*′	synch	synch
T′		*T*′→ε	*T*′→**FT*′		*T*′→ε	*T*′→ε
F	*F*→**id**	synch	synch	*F* →(*E*)	synch	synch

FOLLOW(E) ={),$}

FOLLOW(E′)={),$}

FOLLOW(T) =(+,),$)

FOLLOW(T′)=(+,),$)

FOLLOW(F) =(*,+,),$)

Stack for **id*++id**

STACK	INPUT	OUTPUT
$E	**id***+**+id**$	
$E’T	**id***+**+id**$	
$E’T’F	**id***+**+id**$	
$E’T’**id**	**id***+**+id**$	
$E’T’	*+**+id**$	
$E’TF*	*+**+id**$	
$E’TF	+**id**$	synch, pop()
$E’T’	+**id**$	
$E’	**+id**$	
$E’T+	**+id**$	
$E’T	**id**$	
$E’TF	**id**$	
$E’T’**id**	**id**$	
$E’T’	$	
$E’	$	
$	$	

Recursive Descent Example

```
type      →  simple
            |^id
            | array [simple] of type
simple    →  integer
            |char
            |num dotdot num
```

match()

```
match(token t) {
  // lookahead global variable
  // with current token
  if lookahead == t
    then lookahead = getnext();
  else
    error();
}
```

simple()

```
simple() {
   if lookahead == integer
     match(integer);
   else if lookahead == char
     match(char);
   else if lookahead == num {
     match(num); match(dotdot); match(num);
   }
   else
     error();
}
```

type()

```
type() {
    if (lookahead == integer ||
        lookahead == char ||
        lookahead == num)
        simple();
    else if (lookahead == ‘^’) {
        match(“^”); match(id);
    }
    else if (lookahead == array) {
        match(array);
        match(’[‘); simple();match(’]’);
        match(of); type();
    }
    else
        error();
}
```