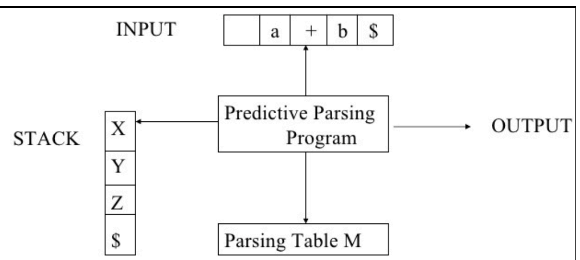
Table-Driven Parsing

- It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls [1]
- The non-recursive parser looks up the production to be applied in a parsing table.
- The table can be constructed directly from LL(1) grammars

1

Table-Driven Parsing



- An input buffer
  - Contains the input string
  - The string can be followed by $, an end marker to indicate the end of the string
- A stack
  - Contains symbols with $ on the bottom, with the start symbol initially on the top
- A parsing table (2-dimensional array M[A, a])
- An output stream (production rules applied for derivation)

2

Input: a string w, a parsing table M for grammar G
Output: if w is in L(G), a leftmost derivation of w; otherwise, an error indication
Method:

```
set ip to point to the first symbol of w$
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $, then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else                        /* X is a non-terminal */
        if M[X, a] = X->Y1Y2…Yk, then
            pop X from the stack
            push Yk, …, Y2, Y1 on to the stack
            output the production X->Y1Y2…Yk
        end
        else error()
until X = $
```

3

An Example

- Input String: id + id * id
- Input parsing table for the following grammar
  E -> TE'
  E' -> +TE' | ε
  T -> FT'
  T' -> *FT' | ε
  F -> (E) | id

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	$F \to \text{id}$			$F \to (E)$		

4

LL Parsing

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	$F \to \text{id}$			$F \to (E)$		

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' -> ε

5

Construction of Parsing Table

- Two functions used to fill in a predicative parsing table for G
  - FIRST
    - For non-terminal A, FIRST(A) is the set of terminals that begin the strings derived from A
  - FOLLOW
    - For non-terminal A, FOLLOW(A) is the set of terminals that appear immediately to the right of A. If A can be the rightmost symbol, $ can be included in FOLLOW(A)

6

Algorithm to compute FIRST(X)

- If X is terminal, then FIRST(X) = {X}
- If X -> ε is a production, then ε∈FIRST(X)
- If X is non-terminal, and X -> $Y_1 Y_2 \ldots Y_{k,}$ then place a in FIRST(X), if for some i, a is in FIRST(Y_i), and ε is in all of FIRST(Y1), …, FIRST(Y_{i-1}). Place ε in FIRST(X) if for all i, FIRST(X_i) contains ε

7

Revisit the example

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | id

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
FIRST(E')={+, ε}
FIRST(T')={*, ε}

8

Algorithm to compute FOLLOW(X)

- Place $ in FOLLOW(S)
- If there is a production A -> αBβ, then {FIRST(β) – ε}⊆ FOLLOW(B)
- If there is a production A -> αB, or a production A -> αBβ, where FIRST(β) contains ε, then FOLLOW(A) ⊆ FOLLOW(B)

9

Revisit the example

E -> TE'
E' -> +TE' | ε
T -> FT'
T' -> *FT' | ε
F -> (E) | id

FIRST(E) = FIRST(T) = FIRST(F) = {(, id}
FIRST(E')={+, ε}
FIRST(T')={*, ε}

FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(T) = FOLLOW(T')
= FIRST(E') - ε U FOLLOW(E')
= {+,), $}
FOLLOW(F) = FIRST(T') - ε U FOLLOW(T')
= {*, +,) $}

10

Algorithm to create a parsing table

Input: Grammar G

Output: Parsing table M

Method:

1. for each production A -> α, do steps 2 and 3
2. for each terminal a in FIRST(α), add A -> α to M[A, a]
3. if ε is in FIRST(α), add A -> α to M[A, b] for each terminal b in FOLLOW(A). If $ is in FOLLOW(A), add A -> α to M[A, $]
4. make each undefined entry of M be error

11

Revisit the example

FIRST(E) = FIRST(T) = FIRST(F) = {(, id} FIRST(E')={+, ε} FIRST(T')={*, ε}	FOLLOW(E) = FOLLOW(E') = {), $} FOLLOW(T) = FOLLOW(T') = {+,), $} FOLLOW(F) = {*, +,) $}	E -> TE' E' -> +TE' | ε T -> FT' T' -> *FT' | ε F -> (E) | id

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

12

Bottom-up Parsing

- Construct a parse tree for an input string beginning at the leaves, and working up towards the root
  - E.g., reducing a string w to the start symbol

13

An Example

- Consider the grammar:
  S -> aABe
  A -> Abc | b
  B -> d
- Input string: abbcde
- How to build a parse tree bottom-up?

14

Bottom-up Parsing

S -> aABe
A -> Abc | b
B -> d

abbcde

- Scan the string to look for a substring that matches the right side of some production
  - E.g., b matches A, while d matches B
- Choose the leftmost b and replace it with A, obtaining “aAbcde”
- Now “Abc”, “b”, and “d” match the right side of some rules
- Choose the leftmost longest substring to replace, obtaining “aAde”

15

Bottom-up Parsing

S -> aABe
A -> Abc | b
B -> d

abbcde

- Replace d with B, getting “aABe”
- Replace the whole string with S

16

LR(1) Parsing

- LR(1) Grammar
- Input String: id + id * id
- There is still a parsing table involved (not shown here)
- A stack is also used to help parsing

```
E  -> E + T
   -> T
T  -> T * F
   -> F
F  -> id
```

17

LR Parsing

"." represents lookup

Stack	Input	Action
	id + id * id$	shift
id .	+ id * id$	Reduce by F->id
F	+ id * id$	Reduce by T->F
T	+ id * id$	Reduce by E->T
E	+ id * id$	shift
E +	id * id$	shift
E + id .	* id$	Reduce by F->id
E + F	* id$	Reduce by T->F
E + T	* id$	shift
E + T *	id$	shift
E + T * id	$	Reduce by F->id
E + T * F	$	Reduce by T->T*F
E + T	$	Reduce by E->E+T
E	$	accept

18

Homework

- Exercises 1.1, 2.4, 2.12
- Hint:
  - For 2.4, please refer to slides about conversions from RE to minimized DFA
- Due Date: 09/20 11:59pm
- Submit the electronic copy to Canvas.