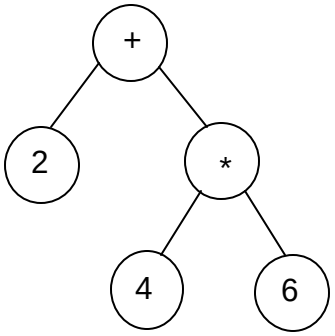
Lecture 16

Here is the output trace for the expression : **2+4*6**

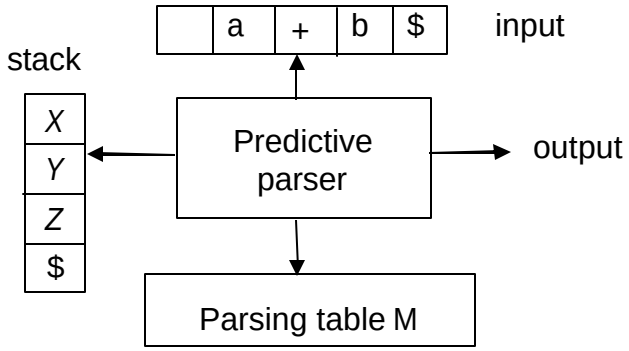
```
  >> Expr::isPresent()
    >> Term::isPresent()
      >> Factor::isPresent()
              token: 2 (257)
      << Factor::isPresent() return true
      >> Tprime::isPresent()
              token: + (267)
      << Tprime::isPresent() return false
    << Term::isPresent() return true
    >> Eprime::isPresent()
          token: + (267)
      >> Term::isPresent()
        >> Factor::isPresent()
                token: 4 (257)
        << Factor::isPresent() return true
        >> Tprime::isPresent()
                token: * (269)
          >> Factor::isPresent()
                  token: 6 (257)
          << Factor::isPresent() return true
          >> Tprime::isPresent()
                  token:  (0)
          << Tprime::isPresent() return false
        << Tprime::isPresent() return true
      << Term::isPresent() return true
      >> Eprime::isPresent()
             token:  (0)
      << Eprime::isPresent() return false
    << Eprime::isPresent() return true
  << Expr::isPresent() return true
```

**** AST ****
(2+(4*6))



Non-recursive Predictive Parsing

It is possible to build a non-recursive predictive parser. This is done by maintaining an explicit stack and using a table. Such a parser is called a table-driven parser. The non-recursive LL(1) parser looks up the production to apply by looking up a parsing table. The LL(1) table has one dimension for current non-terminal to expand and another dimension for next token. Each table cell contains one production.



Consider the expression grammar

1	*E*	?	*T E′*
2	*E′*	?	*+ T E′*
3		|	*e*
4	*T*	?	*F T′*
5	*T′*	?	** F T′*
6		|	*e*
7	*F*	?	*(E)*
8		|	id

Using the table construction algorithm that will be discussed later, we have the predictive parsing table

	id	+	*	(	)	$
E	*E* ? *TE′*			*E* ? *TE′*		
E′		*E′* ? *+TE′*			*E′* ? e	*E′* ? e
T	*T* ? *FT′*			*T* ? *FT′*		
T′		*T′* ? e	*T* ? **FT′*		*T′* ? e	*T′* ? e
F	*F* ? id			*F* ? *(E)*		

The rows are non-terminals and the columns are the terminals of the expression grammar. The predictive parser uses an explicit stack to keep track of pending non-terminals. It can thus be implemented without recursion.

LL(1) Parsing Algorithm

The *input buffer* contains the string to be parsed; $ is the end-of-input marker. The *stack* contains a sequence of grammar symbols. Initially, the stack contains the start symbol of the grammar on the top of $. The parser is controlled by a program that behaves as follows:

The program considers *X*, the symbol on top of the stack, and a, the current input symbol. These two symbols, *X* and a determine the action of the parser. There are *three* possibilities.

1. *X* = a = $, the parser halts and annouces successful completion.

2. *X* = a ¹ $ the parser pops *X* off the stack and advances input pointer to next input symbol.

3. If *X* is a nonterminal, the program consults entry M[*X*,a] of parsing table M.

   a. If the entry is a production M[*X*,a] = {*X* ? *UVW* }, the parser replaces *X* on top of the stack by *WVU* (with *U* on top). As output, the parser just prints the production used: *X* ? *UVW.* However, any other code could be executed here.

   b. If M[*X*,a] =**error**, the parser calls an error recovery routine

Example: let's parse the input string

id+id*id

using the non-recursive LL(1) parser

Stack	*Input*	*Ouput*
$*E*	id+id*id$	
$*E′ T*	id+id*id$	*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′* **?** *e*
$*E′ T* +	+id*id$	*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′* **?** *e*
$	$	*E′* **?** *e*