Compiler Design
Syntax Analysis
Top-Down Parsing

conf. dr. ing. Ciprian-Bogdan Chirila

chirila@cs.upt.ro

http://www.cs.upt.ro/~chirila

Outline

- Recursive-Descent Parsing
- FIRST and FOLLOW
- LL(1) Grammars
- Non-recursive Predictive Parsing
- Error Recovery in Predicting Parsing

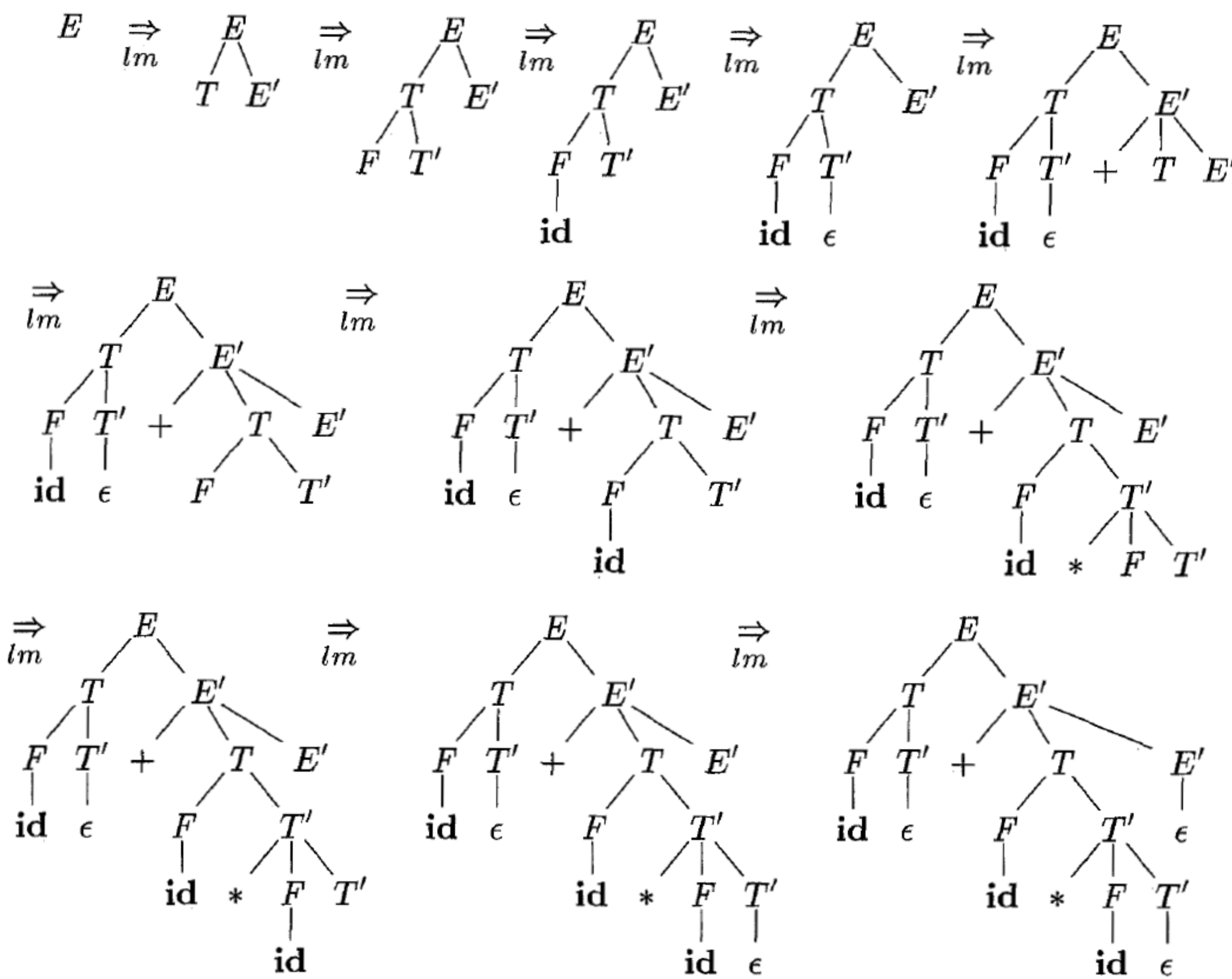
Top Down Parsing

- constructing a parse tree from the input string
  - starting from the root
  - creating the nodes in preorder
- finding the left-most derivation for an input string

Grammar Example

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

Derivation Example for id+id*id



LL(k) Grammars

- LL(k) – class of grammar for which we can construct predictive parsers looking k symbols ahead
- LL(1)
- FIRST and FOLLOW sets
  - are used to construct predictive parsing tables
  - make explicit the choice of production
  - are useful for bottom-up parsing

Recursive Descendant Parsing Program

- set of procedures
- one procedure for each non-terminal
- the start symbol
  - launches the execution
  - announces success if the body scans it's input string

Recursive Descendant Parsing

```
void A()
{
   choose an A-production, A->X1X2…Xk;
   for(i=1 to k)
   {
         if (Xi is a non-terminal)
                  call procedure Xi();
         else if (Xi equals the current symbol a)
                  advance the input to the next symbol;
         else /* an error has occurred */
   }
}
```

Recursive Descendent Parsing Pseudocode

- non-deterministic
  - the manner in which the A-production is chosen is not specified
- generally requires backtracking
  - repeated scans over the input
  - rarely needed to parse programming language constructs
  - not very efficient – tabular methods such as dynamic programming are preferred

Allowing Backtracking

void A()

{

choose an A-production, A->$X_1X_2\ldots X_k$;

for(i=1 to k)

{

if (X_i is a non-terminal)

call procedure X_i();

else if (X_i equals the current symbol a)

advance the input to the ne

else /* an error has occurred */

}

}

try each production in some order

try another A-production

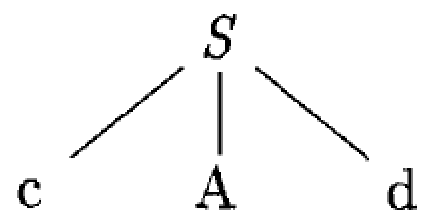
reset the input pointer

Top-Down Parse Tree

- S -> c A d
- A -> a b | a
- w=cad

Step 1

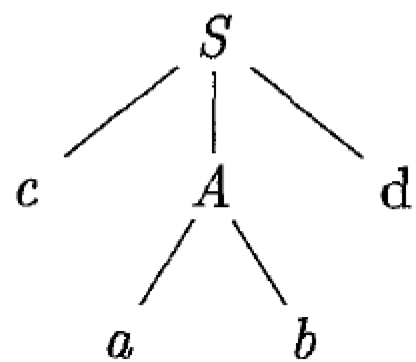
- S has only one production
- we expand S



- first character of input w=cad matches the leftmost leaf in the tree c

Step 2

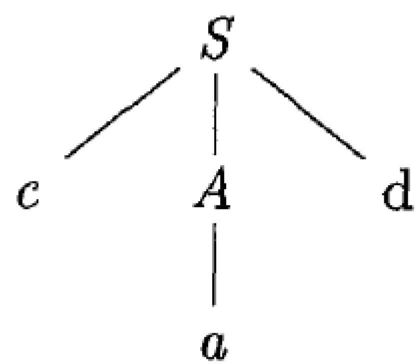
- we expand A->a b
- we have a match for second input character a



- we go to next symbol d
- b does not match d
- we report failure
- we go back to A to try another alternative
- we reset input pointer to position 2

Step 3

- the second alternative for A is A->a



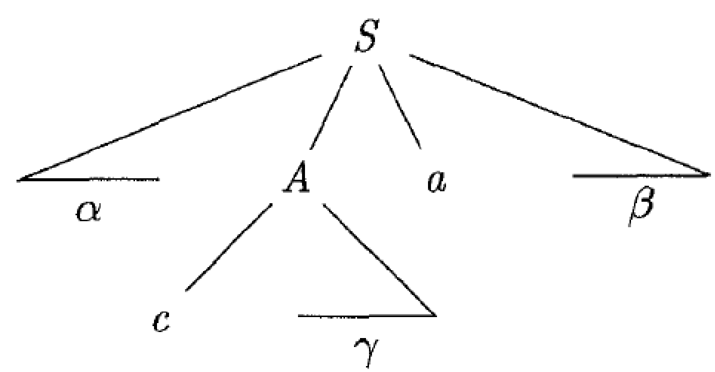
- leaf a matches second symbol
- leaf d matched the third symbol
- we halt with successful parsing message

FIRST and FOLLOW Functions

- two functions useful in creating parsers for both
  - top-down
  - bottom-up
- helps which production to apply based on next input symbol
- in panic mode error recovery tokens produced by FOLLOW are used for synchronization

The FIRST Function

- FIRST(α)
  - set of terminals that begin strings derived from α
  - α is any string of grammar symbols
  - if $\alpha \stackrel{*}{\Rightarrow} \varepsilon$ then ε is in FIRST(α)
- $A \stackrel{*}{\Rightarrow} c\gamma$
  - c is in FIRST(A)



How FIRST function works ?

- A->α|β
- FIRST(α) and FIRST(β) are disjoint sets
- input symbol **a** can be in one of the two sets
- if a is in FIRST(β) we can choose the production A->β

The FOLLOW Function

- FOLLOW(A)
  - the set of terminals a that can appear immediately to the right of A in some sentential form
  - the set of terminals **a** such that $S \overset{*}{\Rightarrow} \alpha A a \beta$ for some α and β

How to compute FIRST ?

- if X is terminal then FIRST(X)={X}
- if X is non-terminal $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some k>=1
  - place a in FIRST(X) if for some i
    - a is in FIRST(Y_i) and
    - ε is in FIRST(Y_1)…FIRST(Y_{i-1})
  - if ε is in all FIRST(Y_j) j=1,…,k
    - then add ε to FIRST(X)
- if X-> ε is a production
  - then add ε to FIRST(X)

How to compute FIRST ?

- input string $X_1X_2\ldots X_n$
- add to FIRST($X_1X_2\ldots X_n$)
  - all non-ε symbols of FIRST(X_1)
  - all non-ε symbols of FIRST(X_2) if ε is in FIRST(X_1)
  - all non-ε symbols of FIRST(X_3) if ε is in FIRST(X_1) and in FIRST(X_2)
  - …
  - ε, if ε is in all FIRST(X_i) $i=1,..,n$

How to compute FOLLOW ?

- place \$ in FOLLOW(S)
  - S is the start symbol
  - \$ is the right end-marker
- if there is a production A->αBβ
  - everything in FIRST(β) *except* ε is in FOLLOW(B)
- if there is a production A->αB or A->αBβ where first(β) contains ε
  - everything in FOLLOW(A) is in FOLLOW(B)

Example

- FIRST(F)={(,id}
- FIRST(T)=FIRST(F)={(,id}
- FIRST(E)=FIRST(T)=FIRST(F)={(,id}

- FIRST(E')={+,ε}
- FIRST(T')={*,ε}

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

Example

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

- FOLLOW(E)={),$}
  - E is the start symbol so it must include $
  - the body (E) tells that the) symbol must be included
- FOLLOW(E’)={),$}
  - E->TE’ so what follows after E will follow after E’
  - FOLLOW(E’)=FOLLOW(E)

Example

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

- FOLLOW(T)={+,),$}
  - E->TE’ so FOLLOW(T) includes FIRST(E’)={+} (except ε)
  - E->TE’ and E’ includes ε, so FOLLOW(E)={),$} is included in FOLLOW(T)
- FOLLOW(T’)={+,),$}
  - T->FT’ so FOLLOW(T) is included in FOLLOW(T’)

Example

- FOLLOW(F)={+,*,),$}
  - T'->*FT' so FOLLOW(F) includes FIRST(T')={*} (except ε)
  - T->FT' and T'->ε so FOLLOW(F) includes FOLLOW(T)={+,),$}

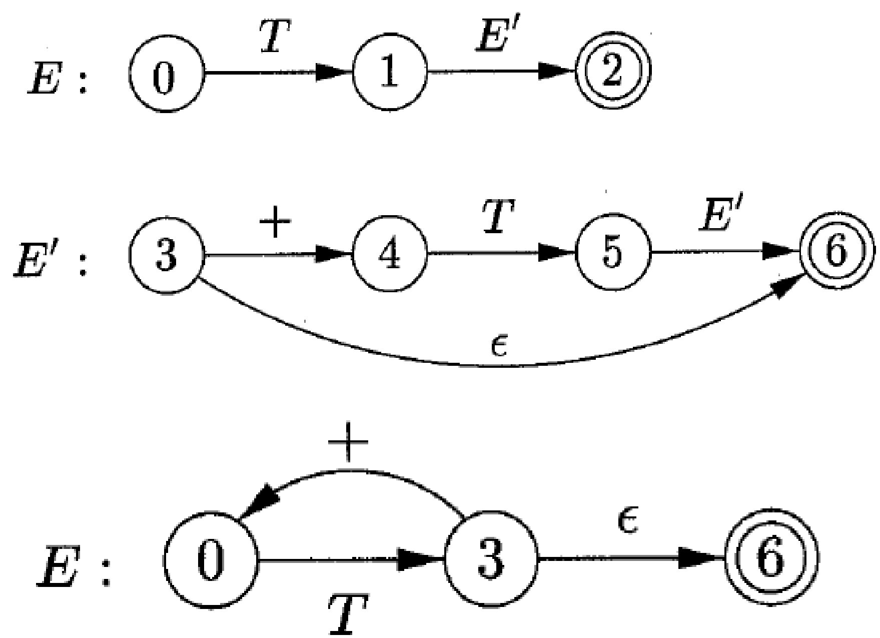
E->TE'
E'->+TE'|ε
T->FT'
T'->*FT'|ε
F->(E) | id

LL(1) Grammars

- predictive parsers
  - recursive descendant with no backtracking
- can be constructed for LL(1) grammar class
  - first L stands for scanning the input **from left to right**
  - second L for producing **leftmost derivation**
  - the 1 is for using one input symbol of lookahead at each step to make parsing actions decisions

Transition Diagrams for Predictive Parsers

- useful for visualizing predictive parsers
  - E->TE’
  - E’->+TE’|ε



Building a Transition Diagram

- eliminate left recursion
- left factor the grammar
- for each non-terminal
  - create an initial and a final state
  - for each production $A \text{->} X_1 X_2 \ldots X_k$
    - create a path from initial state to final state with edges labeled $X_1, X_2, \ldots, X_k$
    - if $A \text{->} \varepsilon$ the path is an edge labeled ε
- label of edges can be tokens or non-terminals
- ε-transitions are the default choice

LL(1) Grammar Definition

- rich enough to cover most programming constructs
- a grammar G is LL(1) iff A->α|β
  - for no terminal a do both α and β derive strings beginning with a
  - at most one of α and β can derive the empty string
  - if $\beta \overset{*}{=}> \varepsilon$
    - α does not derive any string beginning with a terminal in FOLLOW(A)

LL(1) Grammar Definition

- FIRST(α) and FIRST(β) are disjoint sets
- If ε is in FIRST(β) then FIRST(α) and FOLLOW(A) are disjoint sets
- vice versa if ε is in FIRST(α)

Example

- control flow constructs having distinguishable keywords generally satisfies the LL(1) constraints
- stmt-> **if** (expr) stmt **else** stmt

  | **while**(expr) stmt

  | {stmt_list}
- keywords like: if, while, { tells which alternative to take in order to succeed in finding a statement

The Construction of a Predictive Parsing Table

- to collect information from FIRST and FOLLOW
- to store them into a predictive parsing table M[A,a] – two dimensional array
  - A – non-terminal
  - a – terminal or the $ end marker
- main idea
  - A->α is chosen if the next input symbol a is in FIRST(α)
  - if $\alpha => \varepsilon$ or $\alpha \overset{*}{=>} \varepsilon$ production A->α is chosen when the current input symbol or $ is in FOLLOW(A)

The Construction Algorithm

- Input
  - Grammar G
- Output
  - Parsing table M
- Method
  - for each production A->α
    - for each terminal a in FIRST(A) 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 ε is in FIRST(α) and $ is in FOLLOW(A) the add A->α to M[A,$]
  - after filling the table if there is no production in M[A,a] then set M[A,a] to error, represented by an empty entry in the table

Example

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

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

Example

- E->TE'
  - FIRST(TE')=FIRST(T)={(,id}
  - added to M[E,(] and M[E,id]
- E'->+TE'
  - FIRST(+TE')={+}
  - added to M[E',+]
- E'->ε
  - FOLLOW(E')={),$}
  - added to M[E',)] and M[E',$]

Example 2

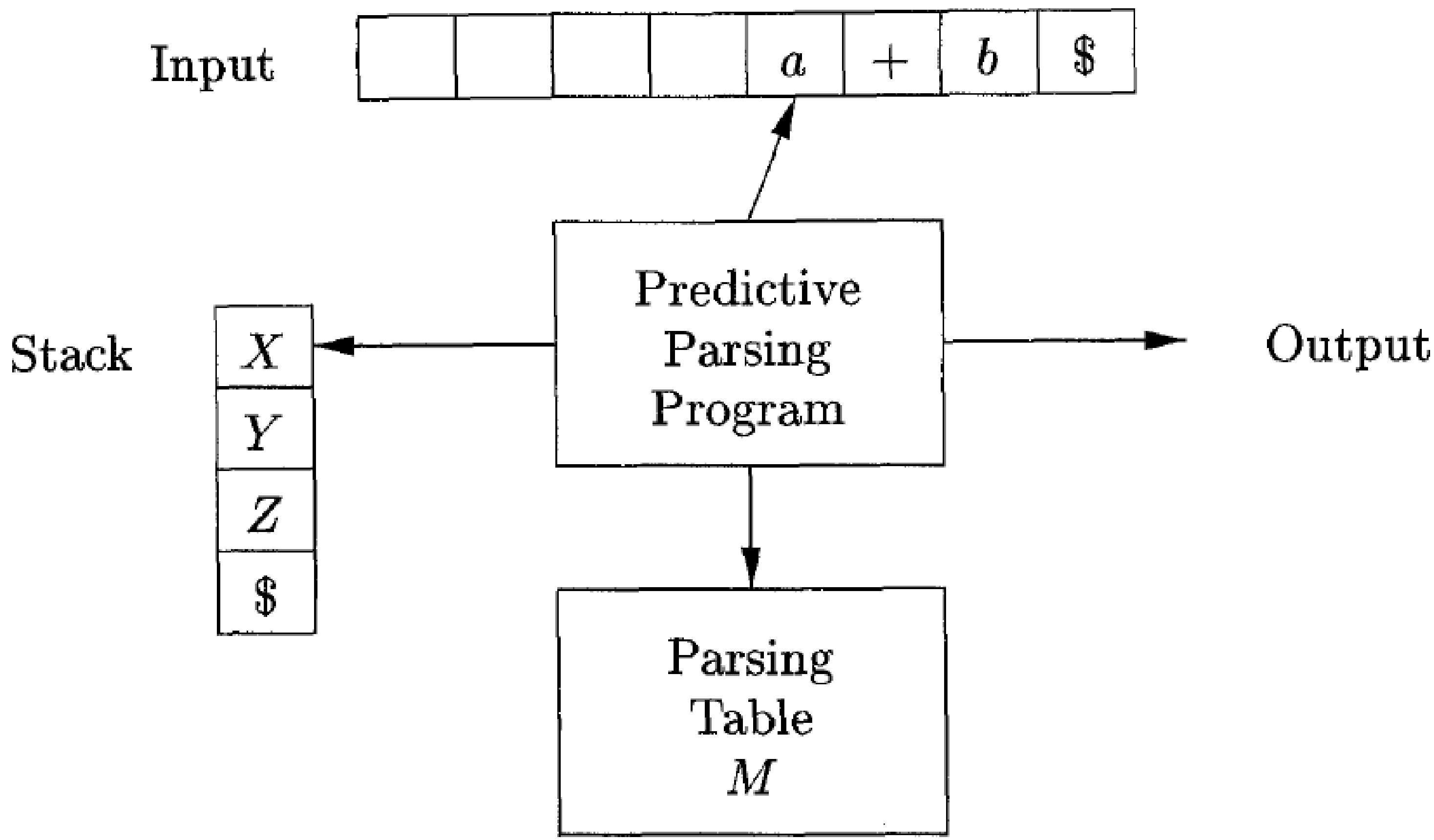
- S->iEtSS' | a
- S'->eS | ε
- E->b

	a	b	e	i	t	$
S	S->a			S->iEtSS'		
S'			S'->ε **S'->eS**			S'->ε
E		E->b				

Non-recursive Predictive Parsing

- to maintain a stack explicitly
- rather than implicitly by recursive calls
- the parser simulates the leftmost derivation
- if w is the input matched so far
  - then the stack holds a sequence of grammar symbols α such that $S \overset{*}{\underset{lm}{\Rightarrow}} w\alpha$

Model of a Table Driven Predictive Parser



Model of a Table Driven Predictive Parser

- input buffer
  - string to be parsed
  - end marker $
- stack containing grammar symbols
  - it’s bottom is marked by $
- parsing table
- output stream

Model of a Table Driven Predictive Parser

- **X** is the symbol on top of the stack
- **a** is the current input symbol
- if **X** is non-terminal
  - the parser chooses a production by consulting the entry M[X,a]
  - semantic actions can be added to build a node in the parse tree
- if **X** is a terminal
  - a match is checked between **X** and input symbol **a**

Model of a Table Driven Predictive Parser

- parser configurations
  - stack content
  - remaining input

Table Driven Predictive Parsing

- Input
  - a string w
  - parsing table M for a grammar G
- Output
  - if w is in L(G) then
    - a leftmost derivation of w
    - otherwise error indication
- Method
  - initially the parser has
    - w$ in the input buffer
    - start symbol S of G on the stack top, above $

Predictive Parsing Algorithm

```
set ip to point the first symbol a of w
set X to the top stack symbol
while(X!=$)
{
   if (X is a) then pop the stack and advance ip
   elseif (X is a terminal) error();
   elseif (M[X,a] is an error entry) error();
   elseif (M[X,a]=X->Y1,Y2,…,Yk)
   {
      output the production X->Y1,Y2,…,Yk
      pop the stack
      push Yk,Yk-1,…,Y1 onto the stack with Y1 on top
   }
   set X to the top stack symbol
}
```

Example

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

	id	**+**	*****	**(**	**)**	**$**
E	E->TE’			E->TE’		
E’		E’->+TE’			E’->ε	E’->ε
T	T->FT’			T->FT’		
T’		T’->ε	T’->*FT’		T’->ε	T’->ε
F	F->id			F->(E)		

Moves Made by a Predictive Parser on id+id*id

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 $E' \to \epsilon$

Error Recovery in Predictive Parsing

- error recovery refers to the stack of the table driven predictive parser
- is makes explicit the terminals and non-terminals the parser hopes to match
- the techniques can be used with recursive-descendant parsing
- an error is detected when:
  - stack top terminal does not match the next input symbol
  - M[A,a] is error (empty)
    - A is the non-terminal on the top of the stack
    - a is the next input symbol

Panic Mode

- skipping input symbols until a set of synchronizing symbols appear
- effectiveness depend on the chosen set
- the sets should be chosen so the parser recovers quickly from errors that are likely to occur in practice

Some Heuristics

- all symbols in FOLLOW(A) will be added to the synchronizing set for A non-terminal
- skip tokens until an element of FOLLOW(A) is seen
- pop A from the stack
- the parsing is likely to continue

Some Heuristics

- only FOLLOW(A) set is not enough
- because semicolons terminate statements in C
- keywords that begin statements may not appear in the FOLLOW set for expression non-terminal
- a missing semicolon after an assignment may result in the keyword beginning next statement to be skipped
- expressions appear within statements
- we need to add
  - to the synchronizing symbols of lower level constructs
  - the synchronizing symbols of higher level constructs
- we can add symbols that begin statements to the synchronizing sets for the non-terminals generating expressions

Some Heuristics

- all symbols from FIRST(A) will be added to the synchronizing set for A non-terminal
  - it is possible to resume parsing according to A
  - if a symbol from FIRST(A) appears
- if a non-terminal can generate the empty string
  - then the production deriving in ε can be used by default
  - we may postpone some error detection
  - cannot cause an error to be missed
  - reduces the number of non-terminals to be considered during error recovery

Some Heuristics

- if a terminal on the top can not be matched
  - pop the terminal
  - issue a message
  - continue parsing
  - the synchronization set of a token consists in all other tokens

Phrase Level Recovery

- filling in the blank cells pointers to error routines
  - change, insert, delete symbols
  - pop from the stack
- stack alteration is questionable
  - modifying the stack might not enable derivation at all
  - risk of infinite loop
  - to check the stack size after modifying it
    - it should decrease

Bibliography

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman – Compilers, Principles, Techniques and Tools, Second Edition, 2007