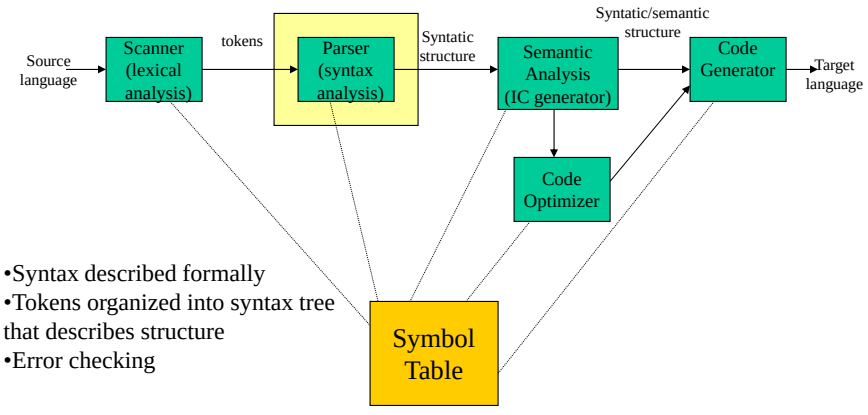
Lecture 4b: LL Parsing

COSC 4316

(grateful acknowledgement to Robert van Engelen and Elizabeth White for some of the material from which these slides have been adapted)

Parsing



- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking

Constructing an LL(1) Predictive Parsing Table

- **for** each entry $M[A,a]$ **do**
  $M[A,a] \leftarrow \emptyset$
  endfor
- **for** each production $A \rightarrow \alpha$ **do**
  for each $a \in$ FIRST(α) **do**
  $M[A,a] \leftarrow M[A,a] \cup \{A \rightarrow \alpha\}$
  endfor
  if $\varepsilon \in$ FIRST(α) **then**
  for each $b \in$ FOLLOW(A) **do**
  add $A \rightarrow \alpha$ to $M[A,b]$
  endfor
  endif
  endfor

for each entry $M[A,a] = \emptyset$ **do**
$M[A,a] \leftarrow$ ***error***
endfor

Example

$E \rightarrow E + T \mid T$
$T \rightarrow T + F \mid F$
$F \rightarrow$ **(E)** | **id**

becomes

$E \rightarrow T\ E_R$
$E_R \rightarrow$ **+** $T\ E_R \mid \varepsilon$
$T \rightarrow F\ T_R$
$T_R \rightarrow$ ***** $F\ T_R \mid \varepsilon$
$F \rightarrow$ **(E)** | **id**

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$E \rightarrow T\ E_R$	**(id**	**$)**
$E_R \rightarrow$ **+** $T\ E_R$	**+**	**$)**
$E_R \rightarrow \varepsilon$	ε	**$)**
$T \rightarrow F\ T_R$	**(id**	**+ $)**
$T_R \rightarrow$ ***** $F\ T_R$	*****	**+ $)**
$T_R \rightarrow \varepsilon$	ε	**+ $)**
$F \rightarrow$ **(** E **)**	**(**	*** + $)**
$F \rightarrow$ **id**	**id**	*** + $)**

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$E \rightarrow T\,E_R$	**(id**	**$)**
$E_R \rightarrow +\,T\,E_R$	**+**	**$)**
$E_R \rightarrow \varepsilon$	ε	**$)**
$T \rightarrow F\,T_R$	**(id**	**+ $)**
$T_R \rightarrow *\,F\,T_R$	*****	**+ $)**
$T_R \rightarrow \varepsilon$	ε	**+ $)**
$F \rightarrow (\,E\,)$	**(**	*** + $)**
$F \rightarrow$ **id**	**id**	*** + $)**

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow T\,E_R$			$E \rightarrow T\,E_R$		
E_R		$E_R \rightarrow +\,T\,E_R$				
T	$T \rightarrow F\,T_R$			$T \rightarrow F\,T_R$		
T_R			$T_R \rightarrow *\,F\,T_R$			
F	$F \rightarrow$ **id**			$F \rightarrow (\,E\,)$		

for each production $A \rightarrow \alpha$ **do**
  for each $a \in$ FIRST(α) **do**
    add $A \rightarrow \alpha$ to $M[A,a]$
  enddo
  if ε ∈ FIRST(α) **then**
    for each $b \in$ FOLLOW(A) **do**
      add $A \rightarrow \alpha$ to $M[A,b]$
    enddo
  endif
enddo
Mark each undefined entry in M error

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$E \rightarrow T\,E_R$	**(id**	**$)**
$E_R \rightarrow +\,T\,E_R$	**+**	**$)**
$E_R \rightarrow \varepsilon$	ε	**$)**
$T \rightarrow F\,T_R$	**(id**	**+ $)**
$T_R \rightarrow *\,F\,T_R$	*****	**+ $)**
$T_R \rightarrow \varepsilon$	ε	**+ $)**
$F \rightarrow (\,E\,)$	**(**	*** + $)**
$F \rightarrow$ **id**	**id**	*** + $)**

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow T\,E_R$			$E \rightarrow T\,E_R$		
E_R		$E_R \rightarrow +\,T\,E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\,T_R$			$T \rightarrow F\,T_R$		
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\,F\,T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (\,E\,)$		

for each production $A \rightarrow \alpha$ **do**
  for each $a \in$ FIRST(α) **do**
    add $A \rightarrow \alpha$ to $M[A,a]$
  enddo
  if ε ∈ FIRST(α) **then**
    for each $b \in$ FOLLOW(A) **do**
      add $A \rightarrow \alpha$ to $M[A,b]$
    enddo
  endif
enddo
Mark each undefined entry in M **error**

Non-LL(1) Examples

Grammar	*Not LL(1) because:*
$S \rightarrow S\ \mathbf{a} \mid \mathbf{a}$	Left recursive
$S \rightarrow \mathbf{a}\ S \mid \mathbf{a}$	FIRST(**a** S) $\cap$ FIRST(**a**) $\neq \varnothing$
$S \rightarrow \mathbf{a}\ R \mid \varepsilon$ $R \rightarrow S \mid \varepsilon$	For R: $S \Rightarrow^* \varepsilon$ and $\varepsilon \Rightarrow^* \varepsilon$
$S \rightarrow \mathbf{a}\ R\ \mathbf{a}$ $R \rightarrow S \mid \varepsilon$	For R: FIRST(S) $\cap$ FOLLOW(R) $\neq \varnothing$

7

LL(1) Grammars are Unambiguous

Ambiguous grammar
$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ SR \mid \mathbf{a}$
$SR \rightarrow \mathbf{e}\ S \mid \varepsilon$
$E \rightarrow \mathbf{b}$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S_R$	**i**	**e** **$**
$S \rightarrow \mathbf{a}$	**a**	**e** **$**
$S_R \rightarrow \mathbf{e}\ S$	**e**	**e** **$**
$S_R \rightarrow \varepsilon$	ε	**e** **$**
$E \rightarrow \mathbf{b}$	**b**	**t**

Error: duplicate table entry

	a	**b**	**e**	**i**	**t**	**$**
S	$S \rightarrow \mathbf{a}$			$S \rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S_R$		
S_R			$S_R \rightarrow \varepsilon$ $S_R \rightarrow \mathbf{e}\ S$			$S_R \rightarrow \varepsilon$
E		$E \rightarrow \mathbf{b}$				

8

Error Recovery in Predictive Parsing

- Panic Mode Recovery:
  (Skip symbols until a set of synchronizing tokens appears)
- How to choose a synchronizing set? One option (among several):
  1. Place all terminal symbols in FOLLOW(*A*) into the synchronizing set for *A*
  2. Wherever *A* has an undefined entry corresponding to a synchronizing token, add a synchronizing action to that entry.
  3. If a synch action is found, pop the current nonterminal *A* from the stack and skip until a synchronizing token is found, ***or***
  4. Don't pop *A*, but skip input until a symbol in FIRST(*A*) is found.

Panic Mode Recovery

Add synchronizing actions to undefined entries based on FOLLOW

Pro: Can be automated
Cons: Error messages are needed

FOLLOW(*E*) = { **)** **$** }
FOLLOW(*ER*) = { **)** **$** }
FOLLOW(*T*) = { **+** **)** **$** }
FOLLOW(*TR*) = { **+** **)** **$** }
FOLLOW(*F*) = { **+** ***** **)** **$** }

	id	**+**	*****	**(**	**)**	**$**
E	$E \to T\,E_R$			$E \to T\,E_R$	***synch***	***synch***
E_R		$E_R \to +\,T\,E_R$			$E_R \to \varepsilon$	$E_R \to \varepsilon$
T	$T \to F\,T_R$	***synch***		$T \to F\,T_R$	***synch***	***synch***
T_R		$T_R \to \varepsilon$	$T_R \to *\,F\,T_R$		$T_R \to \varepsilon$	$T_R \to \varepsilon$
F	$F \to$ **id**	***synch***	***synch***	$F \to$ **(** E **)**	***synch***	***synch***

synch: the driver pops current nonterminal *A* and skips input till synch token or skips input until one of FIRST(*A*) is found

Phrase-Level Recovery

Change input stream by inserting missing tokens

For example: **id id** is changed into **id * id**

Pro:	Can be automated
Cons:	Recovery not always intuitive

Can then continue here

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E_R$	***synch***	***synch***
E_R		$E_R \rightarrow +\ T\ E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\ T_R$	***synch***		$T \rightarrow F\ T_R$	***synch***	***synch***
T_R	***insert ****	$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\ F\ T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**	***synch***	***synch***	$F \rightarrow$ **(** E **)**	***synch***	***synch***

insert *: driver inserts missing ***** and retries the production

11

Error Productions

$E \rightarrow T\ E_R$
$E_R \rightarrow +\ T\ E_R \mid \varepsilon$
$T \rightarrow F\ T_R$
$TR \rightarrow *\ F\ T_R \mid \varepsilon$
$F \rightarrow$ **(** E **)** | **id**

Add "*error production*":

$T_R \rightarrow F\ T_R$

to ignore missing *****, e.g.: **id id**

Pro:	Powerful recovery method
Cons:	Cannot be automated

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E_R$	***synch***	***synch***
E_R		$E_R \rightarrow +\ T\ E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\ T_R$	***synch***		$T \rightarrow F\ T_R$	***synch***	***synch***
T_R	$T_R \rightarrow F\ T_R$	$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\ F\ T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**	***synch***	***synch***	$F \rightarrow$ **(** E **)**	***synch***	***synch***

12

LL(*k*) parsing

also known as the lookahead

- Process input *k* symbols at a time.
- Initially, 'current' non-terminal is start symbol.
- Algorithm
  - Loop until no more input
    - Given next k input tokens and 'current' non-terminal T, choose a rule R (T → …)
    - For each element X in rule R from left to right,

      if X is a non-terminal, we will need to 'expand' X

      else if symbol X is a terminal, see if next input symbol matches X; if so, update from the input
- Typically, we consider **LL(1)**

Two Approaches

- Recursive Descent parsing
  - Code tailored to the grammar
- Table Driven – predictive parsing
  - Table tailored to the grammar
  - General Algorithm

Both algorithms driven by the tokens coming from the lexer.

Writing a Recursive Descent Parser

- Generate a procedure for each non-terminal.

  Use next token from yylex() (lookahead) to choose (PREDICT) which production to 'mimic'.
  - for non-terminal X, call procedure X()
  - for terminals X, call 'match(X)'

Ex: B → b C D

```
B() {
     if (lookahead == 'b')
       { match('b'); C(); D(); }
          else …
    }
```


Writing a Recursive Descent Parser

Also need the following:

```
match(symbol)
{
    if (symbol == lookahead)
        lookahead = yylex()
    else error()
}
main()
{
    lookahead = yylex();
    S();  /* S is the start symbol */
    if (lookahead == EOF) then accept
    else reject
}
error()
{ …
}
```

Back to grammar

```
S() {
   if (lookahead == a ) { match(a);B(); }
  else if (lookahead == b) { match(b); C(); }
  else error("expecting a or b");
}

B() {
  if (lookahead == b)
     {match(b); match(b); C();}
  else error();
}

C() {
  if (lookahead == c)
     { match(c) ; match(c) ;}
  else error();
}
```

S → a B
S → b C

B → b b C

C → c c

Parsing abbcc

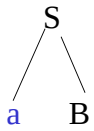
S

Remaining input: abbcc

Call S() from main()

```
S() {
   if (lookahead == a ) { match(a);B(); }        S → a B
  else if (lookahead == b) { match(b); C(); }    S → b C
  else error("expecting a or b");
}
```

Parsing abbcc



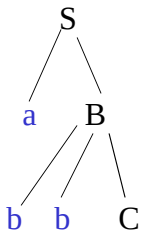
Remaining input: bbcc

Call B() from A():

```
B() {
  if (lookahead == b)
      {match(b); match(b); C();}
  else error();
}
```

B → b b C

Parsing abbcc



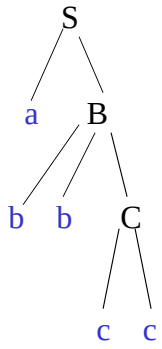
Remaining input: cc

Call C() from B():

```
C() {
  if (lookahead == c)
      { match(c) ; match(c) ;}
  else error();
}
```

C → c c

Parsing abbcc



Remaining input:

How do we find the lookaheads?

- Can compute PREDICT sets from FIRST and FOLLOW for LL(1) parsing:
- PREDICT(A → α)
  - = (FIRST(α) – {ε}) ∪ FOLLOW(A) if ε in FIRST(α)
  - = FIRST(α) if ε not in FIRST(α)

NOTE: ε never in PREDICT sets

For LL(k) grammars, the PREDICT sets for the productions associated with a given non-terminal must be disjoint.

Example

Production	Predict
E → T E'	= FIRST(T) = {(,id}
E' → + T E'	{+}
E' → ε	= FOLLOW(E') = {$,)}
T → F T'	= FIRST(F) = {(,id}
T' → * F T'	{*}
T' → ε	= FOLLOW(T') = {+,$,)}
F → id	{id}
F → (E)	{(}

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

Assume E is the start symbol


```
E() {
  if (lookahead in {(,id } ) { T(); E_prime(); }          E → T E'
  else error("E expecting ( or identifier");
}

E_prime() {
   if (lookahead in {+}) {match(+); T(); E_prime();}     E' → + T E'
   else if (lookahead in {),end_of_file}) return;        E' → e
   else error("E_prime expecting +, ) or end of file");
}

T() {
   if (lookahead in {(,id}) { F(); T_prime(); }          T → F T'
   else error("T expecting ( or identifier");
}
```

```
T_prime() {
  if (lookahead in {*}) {match(*); F(); T_prime();}   T' → * F T'
  else if (lookahead in {+,),end_of_file}) return;          T' → ε
  else error("T_prime expecting *, ) or end of file"); }

F() {
  if (lookahead in {id}) match(id);                              F → id
  else if (lookahead in {(} ) { match( ( ); E(); match ( ) ); }    F → ( E )
  else error("F expecting ( or identifier");}
```


Parsing a + b * c

E

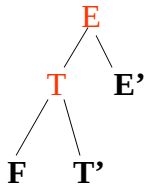
Remaining input: a+b*c

Parsing a + b * c

Remaining input: a+b*c

```
E() {
  if (lookahead in {(,id } ) { T(); E_prime(); }
  else error("E expecting ( or identifier");
}
```

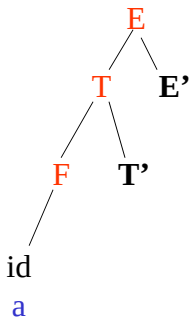

Parsing a + b * c



Remaining input: a+b*c

```
T() {
  if (lookahead in {(,id } ) { F(); T_prime(); }
  else error("T expecting ( or identifier");
}
```

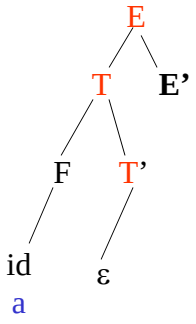
Parsing a + b * c



Remaining input: +b*c

```
F() {
  if (lookahead in {id } )  match(id)
  else if (lookahead in { ( } {
       match( ( ); E(); match( ) ); }
  else error("F expecting ( or identifier");
}
```

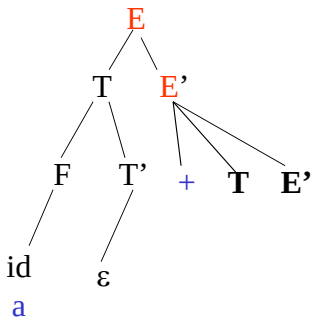

Parsing a + b * c



Remaining input: +b*c

```
T_prime() {
  if (lookahead in {*})   {match(*); F(); T_prime();}'
    else if (lookahead in {+,),end_of_file}) return;
    else error("T_prime expecting *, ) or end of file"); }
}
```

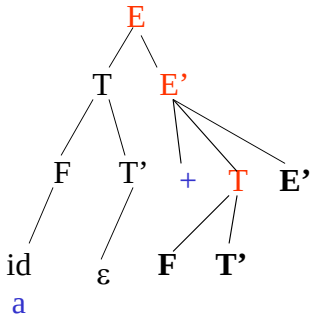
Parsing a + b * c



Remaining input: b*c

```
E_prime() {
 if (lookahead in {+})
      {match(+); T(); E_prime();}'
 else if (lookahead in {),end_of_file})
      return;
 else
 error("E_prime expecting *, ) or end of file"); }
}
```

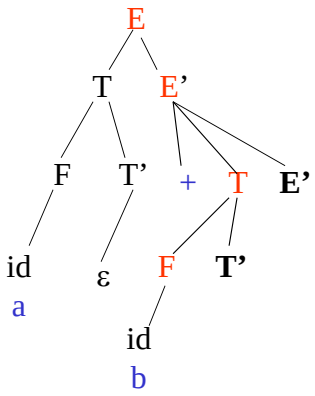

Parsing a + b * c



Remaining input: b*c

```
T() {
 if (lookahead in {(,id } ) { F(); T_prime(); }
 else error("T expecting ( or identifier");
}
```

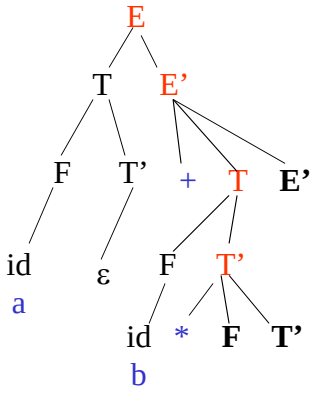
Parsing a + b * c



Remaining input: *c

```
F() {
 if (lookahead in {id } )  match(id)
 else if (lookahead in { ( } {
     match( ( ); E(); match( ) ); }
 else error("F expecting ( or identifier");
}
```

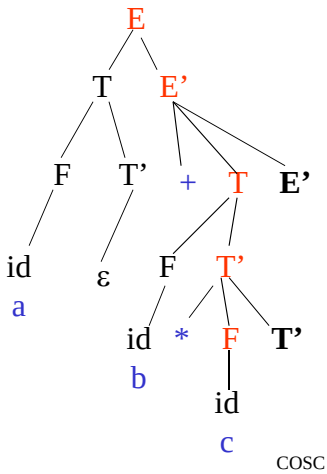

Parsing a + b * c



Remaining input: c

```
T_prime() {
 if (lookahead in {*})
     {match(*); F(); T_prime();}'
 else if (lookahead in {+,),end_of_file})
     return;
 else
 error("T_prime expecting *, ) or end of file"); }
}
```

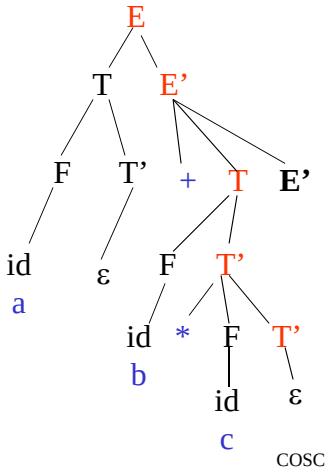
Parsing a + b * c



Remaining input:

```
F() {
 if (lookahead in {id } )  match(id)
 else if (lookahead in { ( } {
      match( ( ); E(); match( ) ); }
 else error("F expecting ( or identifier");
}
```

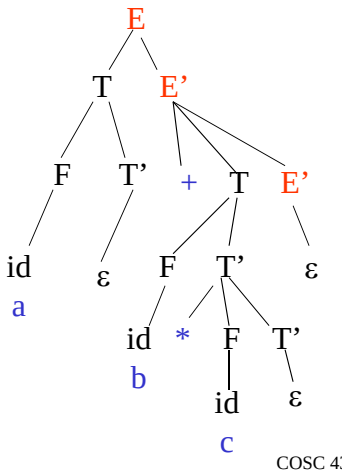

Parsing a + b * c



Remaining input:

```
T_prime() {
 if (lookahead in {*})
      {match(*); F(); T_prime();}'
 else if (lookahead in {+,),end_of_file})
      return;
 else
 error("T_prime expecting *, ) or end of file"); }
}
```

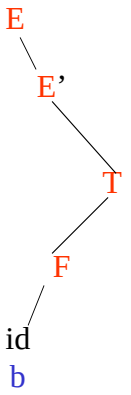
Parsing a + b * c



Remaining input:

```
E_prime() {
  if (lookahead in {+})
       {match(+); T(); E_prime();}'
  else if (lookahead in {),end_of_file})
       return;
  else
  error("E_prime expecting *, ) or end of file"); }
}
```


Stacks in Recursive Descent Parsing



- Runtime stack
- Procedure activations correspond to a path in parse tree from root to some interior node

Two Approaches

- Recursive Descent parsing
  - Code tailored to the grammar
- Table Driven – predictive parsing
  - Table tailored to the grammar
  - General Algorithm

Both algorithms driven by the tokens coming from the lexer.

LL(1) Predictive Parse Tables

An LL(1) Parse table is a mapping T:

N x T → production P or **error**

1. For all productions A → α do

   For each terminal t in Predict(A →α),

   T[A,t] = A → α
2. Every undefined table entry is an error.

Using LL(1) Parse Tables

ALGORITHM

INPUT: token sequence to be parsed, followed by '$' (end of file)

DATA STRUCTURES:

- Parse stack: Initialized by pushing '$' and then pushing the start symbol
- Parse table T

Algorithm: Predictive Parsing

```
push($); push(start_symbol);
lookahead = yylex()
repeat
   X = pop(stack)
   if X is a terminal symbol or $ then
      if X = lookahead then
          lookahead = yylex()
      else error()
   else  /* X is non-terminal */
      if T[X][lookahead] = X → Y1 Y2 …Ym
           push(Ym) … push (Y1)
      else error()
 until X = $ token
```

similar to 'match'

Example

N\T	+	*	(	)	ID	$
E						
E’						
T						
T’						
F						

Production	Predict
1: E → T E’	{(,id}
2: E’ → + T E’	{+}
3: E’ → ε	{$,)}
4: T → F T’	{(,id}
5: T’ → * F T’	{*}
6: T’ → ε	{+,$,)}
7: F → id	{id}
8: F → (E)	{(}

N\T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Production	Predict
1: E → T E’	{(,id}
2: E’ → + T E’	{+}
3: E’ → ε	{$,)}
4: T → F T’	{(,id}
5: T’ → * F T’	{*}
6: T’ → ε	{+,$,)}
7: F → id	{id}
8: F → (E)	{(}

NT/T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Stack	Input	Action
$E	a+b*c$	E → T E’

Assume E is the start symbol

NT/T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’

NT/T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id

NT/T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match

NT/T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε

NT/T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + T E’

NT/T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + T E’
$E’ T +	+b*c$	match

NT/T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + T E’
$E’ T +	+b*c$	match
$E’ T	b*c$	T → F T’

NT/T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + T E’
$E’ T +	+b*c$	match
$E’ T	b*c$	T → F T’
$E’T’F	b*c$	F → id

NT/T	+	*	(	)	ID	$
E			1		1	
E’	2			3		3
T			4		4	
T’	6	5		6		6
F			8		7	

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + T E’
$E’ T +	+b*c$	match
$E’ T	b*c$	T → F T’
$E’T’F	b*c$	F → id
$E’T id	b*c$	match

Parsing a + b * c

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T		T → F T’
$E’T’F		F → id
$E’T’id		match
$E’T’	+b*c$	T’ → ε
$E’		E’ → + T E’
$E’T+		match
$E’T	b*c$	T → F T’

Stack	Input	Action
$E’T’F		F → id
$E’T’id		match
$E’T’	*c$	T’ → * F T’
$E’T’F*		match
$E’T’F	c$	F → id
$E’T’id		match
$E’T’	$	T’ → ε
$E’		E’ → ε
$		accept