CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

EXERCISE:
fill in the parse table

(see next slide)

Parse-table M

NON TERMINALS	id	+	*	(	)	$
E						
E'						
T						
T'						
F						

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

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

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

For each production A -> α of G:

- For each terminal a ∈ FIRST(α), add A -> α to M[A,a]
- If ε ∈ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
- If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A -> α to M[A,$]

Parse-table M

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'		
E'						
T						
T'						
F						

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

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

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

For each production A -> α of G:

- For each terminal a ∈ FIRST(α), add A -> α to M[A,a]
- If ε ∈ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
- If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A -> α to M[A,$]

Parse-table M

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'		
E'		E' -> + T E'				
T						
T'						
F						

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

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

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

For each production A -> α of G:

- For each terminal a ∈ FIRST(α), add A -> α to M[A,a]
- If ε ∈ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
- If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A -> α to M[A,$]

Parse-table M

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'		
E'		E' -> + T E'			E' -> ε	E' -> ε
T						
T'						
F						

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

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

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

For each production A -> α of G:

- For each terminal a ∈ FIRST(α), add A -> α to M[A,a]
- If ε ∈ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
- If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A -> α to M[A,$]

Parse-table M

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'		
E'		E' -> + T E'			E' -> ε	E' -> ε
T	T -> F T'			T -> F T'		
T'						
F						

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

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

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

For each production A -> α of G:

- For each terminal a ∈ FIRST(α), add A -> α to M[A,a]
- If ε ∈ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
- If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A -> α to M[A,$]

Parse-table M

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'		
E'		E' -> + T E'			E' -> ε	E' -> ε
T	T -> F T'			T -> F T'		
T'			T' -> * F T			
F						

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

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

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

For each production A -> α of G:

- For each terminal a ∈ FIRST(α), add A -> α to M[A,a]
- If ε ∈ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
- If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A -> α to M[A,$]

Parse-table M

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'		
E'		E' -> + T E'			E' -> ε	E' -> ε
T	T -> F T'			T -> F T'		
T'		T' -> ε	T' -> * F T		T' -> ε	T' -> ε
F						

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

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

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

For each production A -> α of G:

- For each terminal a ∈ FIRST(α), add A -> α to M[A,a]
- If ε ∈ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
- If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A -> α to M[A,$]

Parse-table M

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'		
E'		E' -> + T E'			E' -> ε	E' -> ε
T	T -> F T'			T -> F T'		
T'		T' -> ε	T' -> * F T		T' -> ε	T' -> ε
F				F -> (E)		

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

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

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

For each production A -> α of G:

- For each terminal a ∈ FIRST(α), add A -> α to M[A,a]
- If ε ∈ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
- If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A -> α to M[A,$]

Parse-table M

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'		
E'		E' -> + T E'			E' -> ε	E' -> ε
T	T -> F T'			T -> F T'		
T'		T' -> ε	T' -> * F T		T' -> ε	T' -> ε
F	F -> id			F -> (E)		

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

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

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

For each production A -> α of G:

- For each terminal a ∈ FIRST(α), add A -> α to M[A,a]
- If ε ∈ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
- If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A -> α to M[A,$]

Parse-table M

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'		
E'		E' -> + T E'			E' -> ε	E' -> ε
T	T -> F T'			T -> F T'		
T'		T' -> ε	T' -> * F T		T' -> ε	T' -> ε
F	F -> id			F -> (E)		

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

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

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

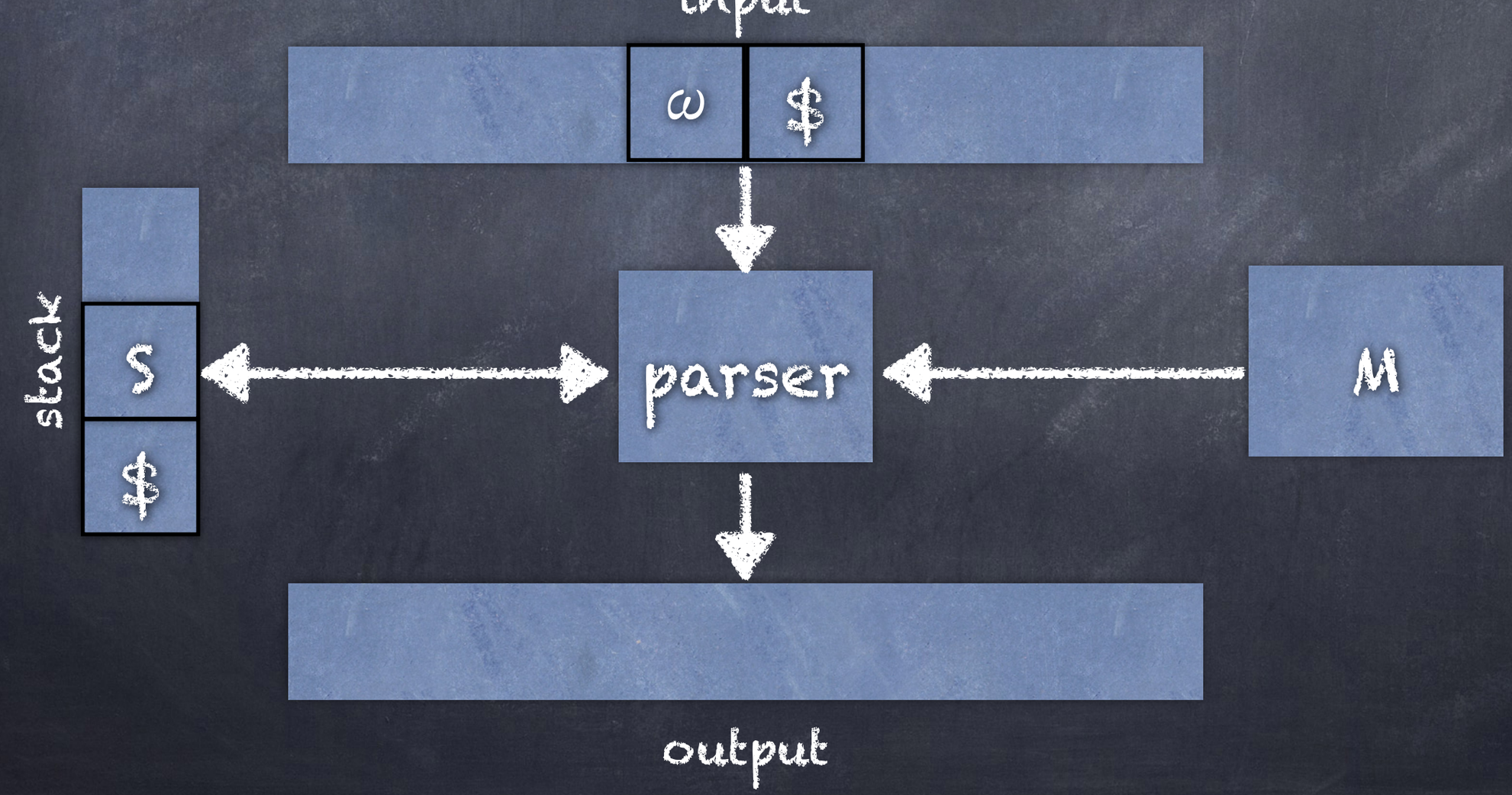
For each production A -> α of G:

- For each terminal a ∈ FIRST(α), add A -> α to M[A,a]
- If ε ∈ FIRST(α), then for each terminal b in FOLLOW(A), add A -> α to M[A,b]
- If ε ∈ FIRST(α) and $ ∈ FOLLOW(A), add A -> α to M[A,$]

Algorithm 4.34 [p. 226]

INPUT: A string ω and a parsing table M for a grammar $G=(N,T,P,S)$.

OUTPUT: If $\omega \in \mathcal{L}(G)$, a leftmost derivation of ω, error otherwise



Algorithm 4.34 [p. 226]

Let a be the first symbol of ω
Let X be the top stack symbol
while (X $\neq$ \$) {
  if (X == a) { pop the stack, advance a in ω }
  else if (X is a terminal) { error }
  else if (M[X,a] is blank) { error }
  else if (M[X,a] is $X \to Y_1 Y_2 \dots Y_k$) {
    output $X \to Y_1 Y_2 \dots Y_k$
    pop the stack
    push $Y_k \dots Y_2 Y_1$ onto the stack
  }
  Let X be the top stack symbol
}
Accept if a == X == \$

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε
id + id * id $	E' $	E' -> + T E'

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε
id + id * id $	E' $	E' -> + T E'
id + id * id $	+ T E' $	

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε
id + id * id $	E' $	E' -> + T E'
id + id * id $	+ T E' $	
id + id * id $	T E' $	T -> F T'

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε
id + id * id $	E' $	E' -> + T E'
id + id * id $	+ T E' $	
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε
id + id * id $	E' $	E' -> + T E'
id + id * id $	+ T E' $	
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε
id + id * id $	E' $	E' -> + T E'
id + id * id $	+ T E' $	
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> * F T'

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	+ T E' $	
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> * F T'
id + id * id $	* F T' E' $	

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	+ T E' $	
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> * F T'
id + id * id $	* F T' E' $	
id + id * id $	F T' E' $	F -> id

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	+ T E' $	
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> * F T'
id + id * id $	* F T' E' $	
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	+ T E' $	
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> * F T'
id + id * id $	* F T' E' $	
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	+ T E' $	
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> * F T'
id + id * id $	* F T' E' $	
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε
id + id * id $	E' $	E' -> ε

```
if (X == a) { pop, advance a in ω }
else if (X is a terminal) { error }
else if (M[X,a] is blank) { error }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT

```
while (X ≠ $) {
  if (X == a) { pop the stack, advance a in ω }
  else if (X is a terminal) { error }
  else if (M[X,a] is blank) { error }
  else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop the stack
    push Yk ... Y2 Y1 onto the stack
  }
  Let X be the top stack symbol
}
Accept if a == X == $
```

id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> * F T'
id + id * id $	* F T' E' $	
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε
id + id * id $	E' $	E' -> ε
id + id * id $	$	

Since both a and X are $, accept

INPUT (a)	STACK (X)	OUTPUT
id + id * id $	E $	E -> T E'
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε
id + id * id $	E' $	E' -> + T E'
id + id * id $	+ T E' $	
id + id * id $	T E' $	T -> F T'
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> * F T'
id + id * id $	* F T' E' $	
id + id * id $	F T' E' $	F -> id
id + id * id $	id T' E' $	
id + id * id $	T' E' $	T' -> ε
id + id * id $	E' $	E' -> ε
id + id * id $	$	

Since both a and X are $, accept

Error recovery in predictive parsing

Basic idea: if input is unexpected given the current state of the parse, try to "synchronize" so that parse can continue (even though no machine code generation will occur once an error has been detected)

Strategies for error recovery

Skip input symbols until a synchronizing token appears.

1. synch(A) includes FOLLOW(A)
2. ... <expr> ; FOLLOW(<expr>) is {';'}, but if ';' is missing, then include all keyword-beginning statement starts in synch(A)
3. add contents of FIRST(A) to synch(A)
4. if A -> ε use this "absorb" A
5. if terminal a on the stack cannot be matched, pop it and issue a message "'a' was inserted". (Synch is set to all other tokens.)

synch is added to each blank M[X,a] when a ∈ FOLLOW(X)

FOLLOW(E) = FOLLOW(E') = {) , $ }

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'	synch	synch
E'		E' -> + T E'			E' -> ε	E' -> ε
T	T -> F T'			T -> F T'		
T'		T' -> ε	T' -> * F T		T' -> ε	T' -> ε
F	F -> id			F -> (E)		

synch is added to each blank M[X,a] when a ∈ FOLLOW(X)

FOLLOW(T) = FOLLOW(T') = { + ,) , $ }

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'	synch	synch
E'		E' -> + T E'			E' -> ε	E' -> ε
T	T -> F T'	synch		T -> F T'	synch	synch
T'		T' -> ε	T' -> * F T		T' -> ε	T' -> ε
F	F -> id			F -> (E)		

synch is added to each blank M[X,a] when a ∈ FOLLOW(X)

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

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'	synch	synch
E'		E' -> + T E'			E' -> ε	E' -> ε
T	T -> F T'	synch		T -> F T'	synch	synch
T'		T' -> ε	T' -> * F T		T' -> ε	T' -> ε
F	F -> id	synch	synch	F -> (E)	synch	synch

Modified parse-table M

NON TERMINALS	id	+	*	(	)	$
E	E -> T E'			E -> T E'	synch	synch
E'		E' -> + T E'			E' -> ε	E' -> ε
T	T -> F T'	synch		T -> F T'	synch	synch
T'		T' -> ε	T' -> * F T		T' -> ε	T' -> ε
F	F -> id	synch	synch	F -> (E)	synch	synch

synch is added to each blank M[X,a] when a ∈ FOLLOW(X)

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

Algorithm 4.34 (modified w/error handling)

Let a be the first symbol of ω
Let X be the top stack symbol
while (X ≠ $) {
  if (x == a) { pop the stack, advance a in ω }
  else if (X is a terminal) { pop stack }
  else if (M[X,a] is blank) { advance a in ω }
  else if (M[X,a] is synch) { pop stack }
  else if (M[X,a] is $X \rightarrow Y_1 Y_2 \ldots Y_k$) {
    output $X \rightarrow Y_1 Y_2 \ldots Y_k$
    pop the stack
    push $Y_k \ldots Y_2 Y_1$ onto the stack
  }
  Let X be the top stack symbol
}
Accept if a == X == $

INPUT (a)	STACK (X)	OUTPUT
+ id * + id $	E $	???

```
if (x == a) { pop, advance a in ω }
else if (X is a terminal) { pop }
else if (M[X,a] is blank) { advance a in ω }
else if (M[X,a] is synch) { pop }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
+ id * + id $	E $	error, skip +

```
if (x == a) { pop, advance a in ω }
else if (X is a terminal) { pop }
else if (M[X,a] is blank) { advance a in ω }
else if (M[X,a] is synch) { pop }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
+ id * + id $	E $	error, skip +
+ id * + id $	E $	E -> T E'
+ id * + id $	T E' $	T -> F T'
+ id * + id $	F T' E' $	F -> id
+ id * + id $	id T' E' $	
+ id * + id $	T' E' $	T' -> * F T'
+ id * + id $	* F T' E' $	
+ id * + id $	F T' E' $	???

```
if (x == a) { pop, advance a in ω }
else if (X is a terminal) { pop }
else if (M[X,a] is blank) { advance a in ω }
else if (M[X,a] is synch) { pop }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

INPUT (a)	STACK (X)	OUTPUT
+ id * + id $	E $	error, skip +
+ id * + id $	E $	E -> T E'
+ id * + id $	T E' $	T -> F T'
+ id * + id $	F T' E' $	F -> id
+ id * + id $	id T' E' $	
+ id * + id $	T' E' $	T' -> * F T'
+ id * + id $	* F T' E' $	
+ id * + id $	F T' E' $	error, M[F,+] = synch, pop F

```
if (x == a) { pop, advance a in ω }
else if (X is a terminal) { pop }
else if (M[X,a] is blank) { advance a in ω }
else if (M[X,a] is synch) { pop }
else if (M[X,a] is X -> Y1 Y2 ... Yk) {
    output X -> Y1 Y2 ... Yk
    pop
    push Yk ... Y2 Y1
}
Let X be the top stack symbol
```

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

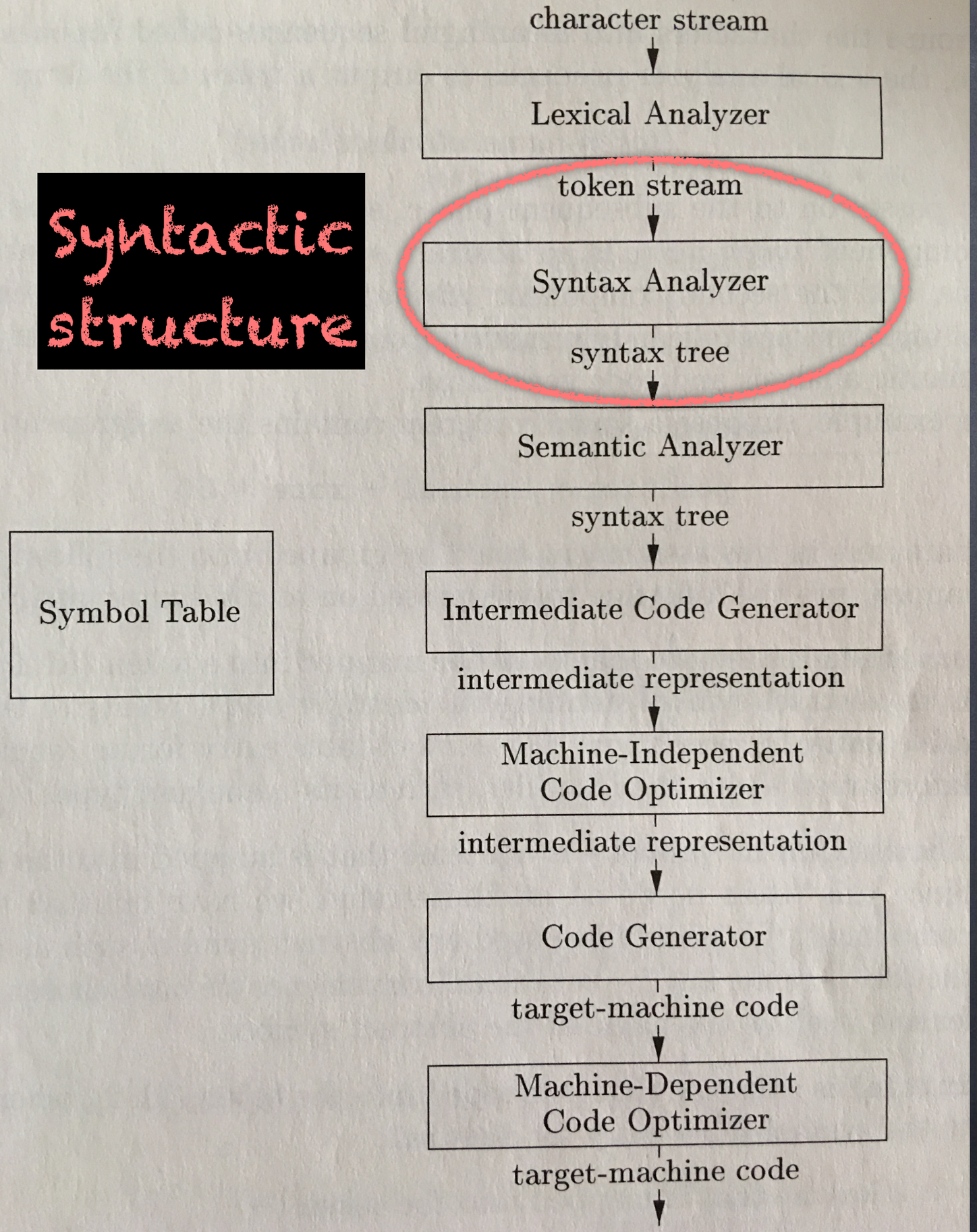
INPUT (a)	STACK (X)	OUTPUT
+ id * + id $	E $	error, skip +
+ id * + id $	E $	E -> T E'
+ id * + id $	T E' $	T -> F T'
+ id * + id $	F T' E' $	F -> id
+ id * + id $	id T' E' $	
+ id * + id $	T' E' $	T' -> * F T'
+ id * + id $	* F T' E' $	
+ id * + id $	F T' E' $	error, M[F,+] = synch, pop F
+ id * + id $	T' E' $	T' -> ε
+ id * + id $	E' $	E' -> + T E'
+ id * + id $	+ T E' $	
+ id * + id $	T E' $	T -> F T'
+ id * + id $	F T' E' $	F -> id
+ id * + id $	id T' E' $	
+ id * + id $	T' E' $	T' -> ε
+ id * + id $	E' $	E' -> ε
+ id * + id $	$	

Since there were errors, do not accept.

Phases of a compiler

Figure 1.6, page 5 of text

Syntactic structure



Bottom-up parsing

Top-down predictive parsing gave us a quick overview of issues related to parsing.

With this context we can more easily describe bottom-up parsing.

Example grammar

E -> E + T

E -> T

T -> T * F

T -> F

F -> (E)

F -> id

Same expression grammar we used for top-down presentation.

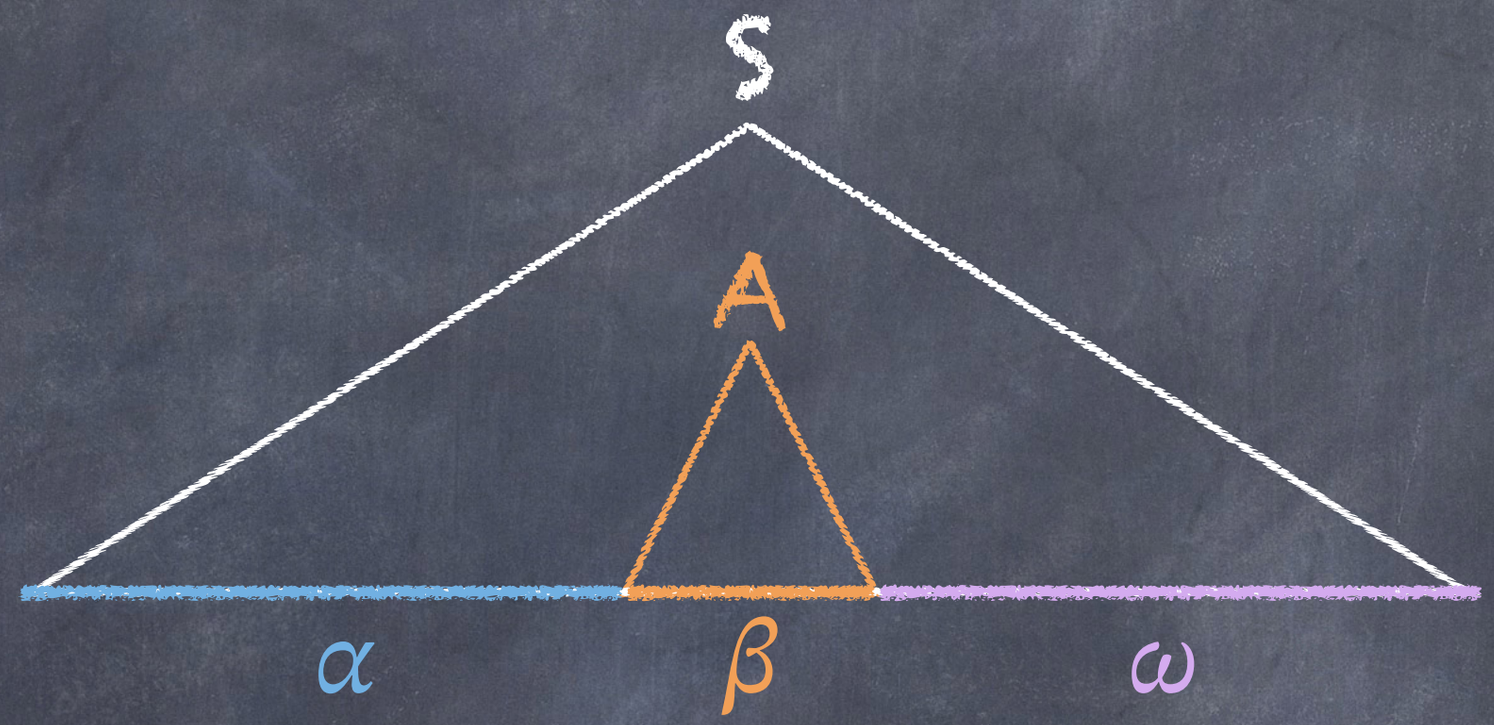
Terminology

- If $S \Rightarrow^*_{lm} \alpha$ then we call α a **left-sentential form** of the grammar (lm means leftmost)
- If $S \Rightarrow^*_{rm} \alpha$ then we call α a **right-sentential form** of the grammar (rm means rightmost)

handle

- "Informally, a 'handle' is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation." [p. 235]
- "Formally, if $S \Rightarrow^*_{rm} \alpha A \omega \Rightarrow_{rm} \alpha\beta\omega$, then the production A -> β in the position following α is a handle of $\alpha\beta\omega$" [p. 235]
- "Alternatively, a handle of a right-sentential form γ is a production A -> β and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ." [p. 235]

As a picture



"A handle A -> β in the parse tree for αβω" Fig 4.27 [p. 236]

A rightmost derivation of the string

id * id

Rightmost derivation	Production
E	
⇒ T	E -> T
⇒ T * F	T -> T * F
⇒ T * id	F -> id
⇒ F * id	T -> F
⇒ id * id	F -> id

Recall grammar

E -> E + T
E -> T
T -> T * F
T -> F
F -> (E)
F -> id

[p.235]

A bottom-up parse: what we're aiming for!

Table is reverse of that on previous slide.

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

id * id has handle id

(or more formally F -> id is a handle)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

F * id has handle F

(or more formally T -> F is a handle)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

T * id has handle id

(or more formally F -> id is a handle after T *)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

T * F has handle T * F

(or more formally T -> T * F is a handle)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

T has handle T

(or more formally E -> T is a handle)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

What happens if we reduce something that's not a handle?

T * id has handle id

(or more formally F -> id is a handle after T *)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id

Consider this point in the previous table.

We identified F -> id as a handle.

figure 4.26 [p.235]

Example - figure 4.26 [p.235]

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	T	E -> T

What if ...

... we made a different choice?

Example - figure 4.26 [p.235]

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	T	E -> T
E * id	id	F -> id
E * F	F	T -> F
E * T	T	E -> T
E * E	*FAIL*	

T * id could be reduced to E * id using production E -> T, but E -> T is not a handle since that reduction does not represent "one step along the reverse of a rightmost derivation."

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

Basic idea

If we know what the handle is for each right sentential form, we can run the rightmost derivation in reverse!

Handle pruning [p 235]

- "A rightmost derivation in reverse can be obtained by 'handle pruning' "
- If $\omega \in \mathcal{L}(G)$:

Rightmost derivation →

$$S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \cdots \Rightarrow_{rm} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = \omega$$

← Handle pruning

Big question

How do we figure out the handles?

Big question

How do we figure out the handles?

We'll answer this in a bit, but first let's consider how a parse will proceed in a bit more detail.

Shift-reduce parsing

STACK [Bottom...Top]	INPUT
$	ω $
	
$ S	$

[modified from fig 4.28, p 237]

Revisit example, with input: id * id $

Stack	Lookahead	Handle	Action
$	id * id $		Shift
$ id	* id $	id	Reduce F -> id
$ F	* id $	F	Reduce T -> F
$ T	* id $		Shift
$ T *	id $		Shift
$ T * id	$	id	Reduce F -> id
$ T * F	$	T * F	Reduce T -> T * F
$ T	$	T	Reduce E -> T
$ E	$		Accept

Observations [p 235]

- ω, the string after the handle, must be $\in T^*$
- We say "a handle" rather than "the handle" since the grammar may be ambiguous and may therefore allow more than one rightmost derivation of $\alpha\beta\omega$.
- If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.