Parsing

15-411/15-611 Compiler Design

Ben L. Titzer and Seth Copen Goldstein

Feb 11, 2025

Today – Parsing

Parsing

- Top-down parsers
  - FIRST, FOLLOW, and NULLABLE
- Bottom-up parsers
  - handle pruning
  - parsing method
  - constructing state machine
  - LR0
  - SLR
  - LR(k) & LALR
  - Handling Ambiguity

Languages

- Regular languages
  - Equivalent in power to NFAs and DFAs
  - Can be described by regular expressions
  - Do not handle recursion
- Context-free languages
  - Equivalent in power to PDAs
  - Can be described by *context-free grammars*
  - Handle recursion, necessary for real PLs!

Context-Free Grammar

- A context-free grammar, G, is described by:
  - Σ, a set of terminals …
  - A, a set of non-terminals (NT).
  - S, S ∈ A, the start symbol
  - P, set of productions (aka rules)
    - a production, p, has the form: $A \rightarrow \alpha$
  - E.g.

S := E

S := **`print`** E

E := E + T

E := T

T := F

terminals

non-terminals

Context-Free Grammar

- A context-free grammar, G, is described by:
  - Σ, a set of terminals …
  - A, a set of non-terminals (NT).
  - S, S ∈ A, the start symbol
  - P, set of productions (aka rules)
    - a production, p, has the form: $A \rightarrow \alpha$
  - E.g.

S := E
S := **`print`** E
E := E + T
E := T
T := F

multiple alternatives

What makes a grammar CF?

- Only one NT on left-hand side => context-free
- What makes a grammar context-sensitive?
- $\alpha A \beta \rightarrow \alpha \gamma \beta$ where
  - α or β may be empty,
  - but γ is not-empty
- Are context-sensitive grammars useful for compiler writers?

Simple Grammar of Expressions

S := Exp

Exp := Exp + Exp

Exp := Exp - Exp

Exp := Exp * Exp

Exp := Exp / Exp

Exp := **`id`**

Exp := **`int`**

Describes a language of expressions. e.g.: 2+3*x

Derivation

- A *derivation* is a chosen sequence of productions (expansions)
  - S → Exp → Exp + Exp → id + Exp → id + int
- A successful sequence of expansions that match the input constitute a *parse*
  - Connecting the expansions in each successive step produces a *parse tree*
  - Parse tree is a form of abstract syntax tree
  - Building a *correct AST* is the whole point

Leftmost Derivations

- Leftmost derivation: leftmost NT always chosen

input: 2+3*x

1	S	:= Exp
2	Exp	:= Exp + Exp
3	Exp	:= Exp - Exp
4	Exp	:= Exp * Exp
5	Exp	:= Exp / Exp
6	Exp	:= **id**
7	Exp	:= **int**

S

by 1 ⇒ Exp

by 4 ⇒ Exp * Exp

by 2 ⇒ Exp + Exp * Exp

by 7 ⇒ $\texttt{int}_2$ + Exp * Exp

by 7 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * Exp

by 6 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * $\texttt{id}_x$

Rightmost Derivations

- Rightmost derivation: rightmost NT always chosen

input: 2+3*x

1 S := Exp

2 Exp := Exp + Exp

3 Exp := Exp - Exp

4 Exp := Exp * Exp

5 Exp := Exp / Exp

6 Exp := **id**

7 Exp := **int**

S

by 1 ⇒ Exp

by 4 ⇒ Exp * Exp

by 6 ⇒ Exp * $\mathbf{id}_x$

by 2 ⇒ Exp + Exp * $\mathbf{id}_x$

by 7 ⇒ Exp + $\mathbf{int}_3$ * $\mathbf{id}_x$

by 7 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$

Parse Trees

- symbols in RHS are children of NT being rewritten

input: 2+3*x

S

by 1 ⇒ Exp

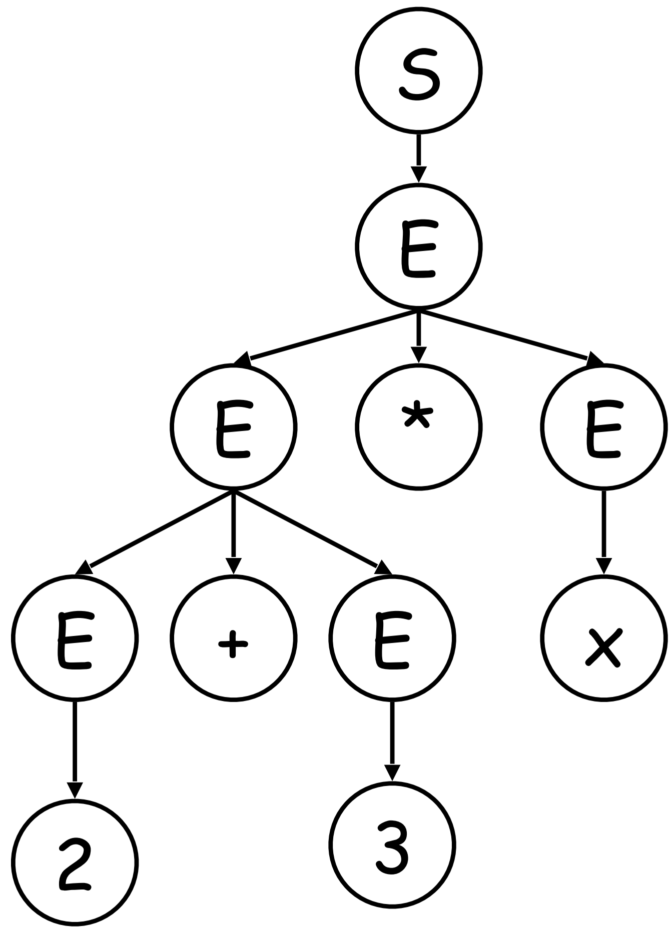
by 4 ⇒ Exp * Exp

by 2 ⇒ Exp + Exp * Exp

by 7 ⇒ $\texttt{int}_2$ + Exp * Exp

by 7 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * Exp

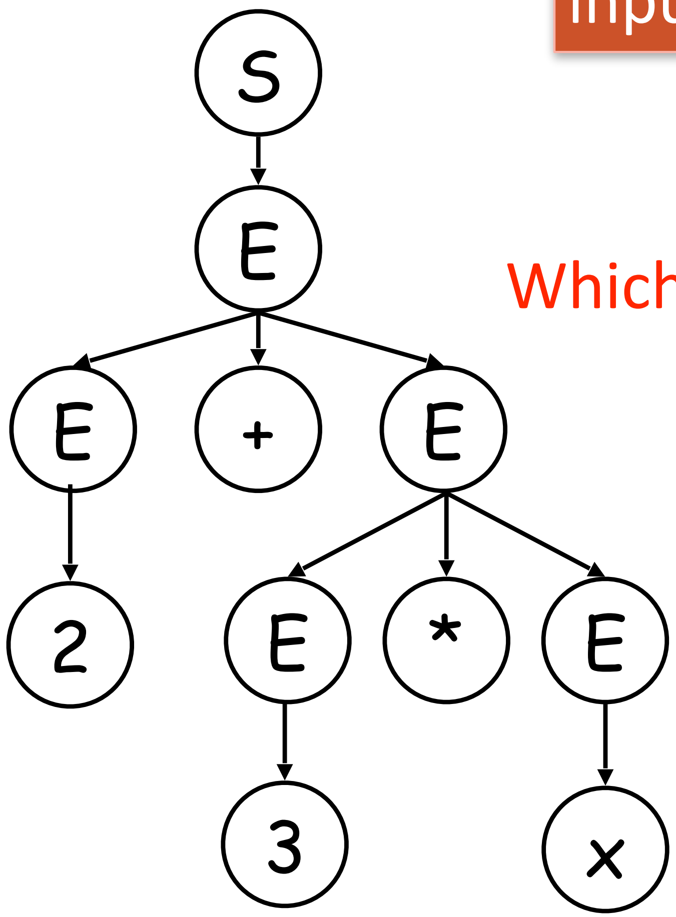
by 6 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * $\texttt{id}_x$



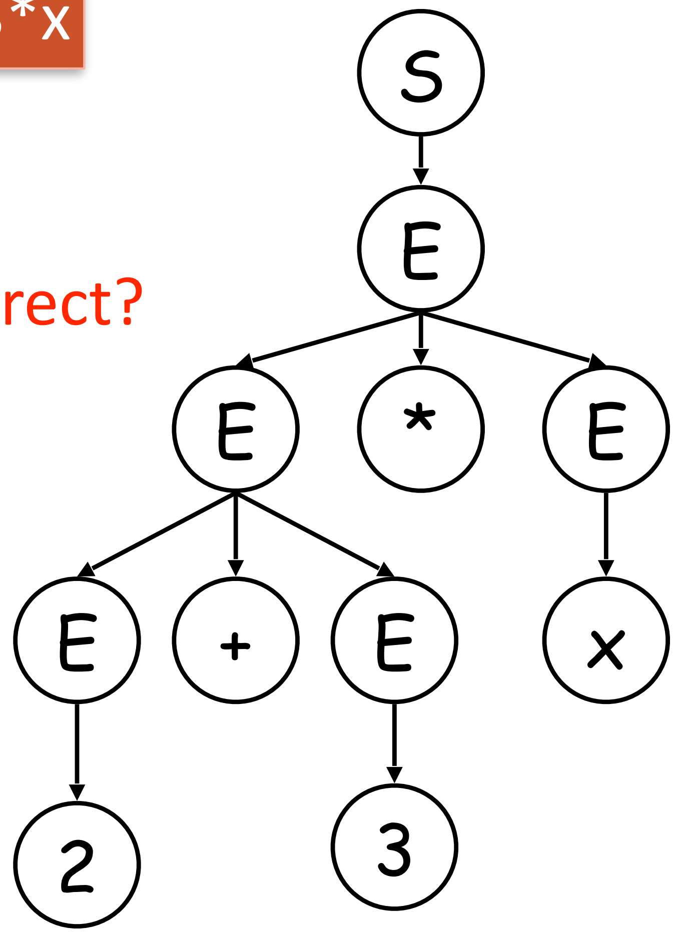
Ambiguity in Grammars

- Some grammars have more than one way to parse a given input, i.e. are ambiguous

input: 2+3*x



Which is correct?



Resolving Ambiguity

- Ambiguity is a problem with the grammar
- One possible fix: Add precedence with more non-terminals
- In this example, one for each level of precedence:
  - (+, -) exp
  - (*, /) term
  - (**id**, **int**) factor
  - Make sure parse derives sentences that respect the precedence
  - Make sure that extra levels of precedence can be bypassed, i.e., “x” is still legal

A Better Exp Grammar

1	S	:= Exp
2	Exp	:= Exp + Term
3	Exp	:= Exp - Term
4	Exp	:= Term
5	Term	:= Term * Factor
6	Term	:= Term / Factor
7	Term	:= Factor
8	Factor	:= **id**
9	Factor	:= **int**

S

by 1 ⇒ Exp

by 2 ⇒ Exp + Term

by 4 ⇒ Term + Term

by 7 ⇒ Factor + Term

by 9 ⇒ $\texttt{int}_2$ + Term

by 5 ⇒ $\texttt{int}_2$ + Term * Factor

by 7 ⇒ $\texttt{int}_2$ + Factor * Factor

by 9 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * Factor

by 8 ⇒ $\texttt{int}_2$ + $\texttt{int}_3$ * $\texttt{id}$

What is the parse tree?

Parsing a Grammar

- Top-Down
  - start at root of parse-tree
  - pick a production and expand to match input
  - may require backtracking
  - if no backtracking required, predictive
- Bottom-up
  - start at leaves of tree
  - recognize valid prefixes of productions
  - consume input and change state to match
  - use stack to track state

Top-down Parsers

- Starts at root of parse tree and recursively expands children that match the input
- In general case, may require backtracking
- Such a parser uses *recursive descent*
- Easy: one function per nonterminal
- When a grammar does not require backtracking a predictive parser can be built.

Top-Down parsing

- Start with root of tree, i.e., S
- Repeat until entire input matched:
  - pick a non-terminal, A, and pick a production $A \rightarrow \gamma$ that can match input, and expand tree
  - if no such rule applies, backtrack
- Key is obviously selecting the right production

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

	S	| $\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 1 ⇒	E	| $\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 9 ⇒	$\texttt{int}_2$ - T	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$
by 5 ⇒	$\texttt{int}_2$ - T * F	$\texttt{int}_2$ - $\texttt{int}_3$ * $\texttt{id}_x$

input: 2+3*x

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

input: 2+3*x

	S	| int_2 - int_3 * id_x
by 1 ⇒	E	| int_2 - int_3 * id_x
by 2 ⇒	E + T	| int_2 - int_3 * id_x
by 4 ⇒	T + T	| int_2 - int_3 * id_x
by 7 ⇒	F + T	int_2 | - int_3 * id_x

Must backtrack here!

by 9 ⇒	int_2 - T	int_2 - int_3 * id_x
by 5 ⇒	int_2 - T * F	int_2 - int_3 * id_x

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

input: 2+3*x

	S	int_2 - int_3 * id_x
by 1 ⇒	E	int_2 - int_3 * id_x
by 2 ⇒	E + T	int_2 - int_3 * id_x
by 4 ⇒	T + T	int_2 - int_3 * id_x
by 7 ⇒	F + T	int_2 - int_3 * id_x
by 9 ⇒	int_2 + T	int_2 - int_3 * id_x
by 3 ⇒	E - T	int_2 - int_3 * id_x
by 4 ⇒	T - T	int_2 - int_3 * id_x
by 7 ⇒	F - T	int_2 - int_3 * id_x
by 9 ⇒	int_2 - T	int_2 - int_3 * id_x
by 5 ⇒	int_2 - T * F	int_2 - int_3 * id_x

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

	S	$\mid$ int_2 - int_3 * id_x
by 1 ⇒	E	$\mid$ int_2 - int_3 * id_x
by 2 ⇒	E + T	$\mid$ int_2 - int_3 * id_x
by 4 ⇒	T + T	$\mid$ int_2 - int_3 * id_x
by 7 ⇒	F + T	int_2 $\mid$ - int_3 * id_x
by 9 ⇒	int_2 + T	$\mid$ int_2 - int_3 * id_x
by 3 ⇒	E - T	$\mid$ int_2 - int_3 * id_x
by 4 ⇒	T - T	$\mid$ int_2 $\mid$ - int_3 * id_x
by 7 ⇒	F - T	int_2 - $\mid$ int_3 * id_x
by 9 ⇒	int_2 - T	int_2 - int_3 * id_x
by 5 ⇒	int_2 - T * F	int_2 - int_3 * id_x

What kind of derivation is this parsing?

Top-down for Exp Grammar

1	S := E
2	E := E + T
3	E := E - T
4	E := T
5	T := T * F
6	T := T / F
7	T := F
8	F := id
9	F := int

	S		int_2 - int_3 * id_x
by 1 ⇒	E		int_2 - int_3 * id_x
by 2 ⇒	E + T		int_2 - int_3 * id_x
by 2 ⇒	E + E + T		int_2 - int_3 * id_x
by 2 ⇒	E + E + E + T	int_2 - int_3 * id_x

Will not terminate! Why?

grammar is left-recursive

What should we do about it?

Eliminate left-recursion

input: 2+3*x

Eliminating Left-Recursion

- Given 2 productions:

  $A := A\ \alpha$

  $A := \beta$

  Where neither α nor β start with A

  (e.g., For example, E := E + T | T)
  ($+\ T$ is α, T is β)

- Make it right-recursive:

  $A := \beta\ R$

  $R := \alpha\ R$

  $\quad |$

  R is right recursive

- Extends to general case.

Rewriting Exp Grammar

1	S	:= E
2	E	:= E + T
3	E	:= E - T
4	E	:= T
5	T	:= T * F
6	T	:= T / F
7	T	:= F
8	F	:= **`id`**
9	F	:= **`int`**

1	S	:= E
2′	E′	:= + T E′
3′	E′	:= - T E′
4′	E′	:=
5′	T′	:= * F T′
6′	T′	:= / F T′
7′	T′	:=
8	F	:= **`id`**
9	F	:= **`int`**

2	E	:= T E′
5	T	:= F T′

Try again

input: 2+3*x

1	S := E
2	E := T E'
2'	E' := + T E'
3'	E' := - T E'
4'	E' :=
5	T := F T'
5'	T' := * F T'
6'	T' := / F T'
7'	T' :=
8	F := **id**
9	

	S	•int$_2$ - int$_3$ * id$_x$
by 1 ⇒	E	•int$_2$ - int$_3$ * id$_x$
by 2 ⇒	T E'	•int$_2$ - int$_3$ * id$_x$
by 5 ⇒	F T' E'	•int$_2$ - int$_3$ * id$_x$
by 9 ⇒	2 T' E'	int$_2$ •- int$_3$ * id$_x$
by 7' ⇒	2 E'	int$_2$ •- int$_3$ * id$_x$
by 3' ⇒	2 - T E'	int$_2$ - •int$_3$ * id$_x$
by 5 ⇒	2 - F T' E'	int$_2$ - •int$_3$ * id$_x$
by 9 ⇒	2 - 3 T' E'	int$_2$ - int$_3$ •* id$_x$
by 5' ⇒	2 - 3 * F T' E'	int$_2$ - int$_3$ * •id$_x$
		t$_3$ * id$_x$•
		t$_3$ * id$_x$•
		t$_3$ * id$_x$•

Unlike previous time we tried this, it appears that only one production applies at a time. I.e., no backtracking needed. Why?

Lookahead

- How to pick right production?
- Lookahead in input stream for guidance
- General case: arbitrary lookahead required
- Luckily, many context-free grammars can be parsed with limited lookahead
- If we have $A \rightarrow \alpha \mid \beta$, then we want to correctly choose either $A \rightarrow \alpha$ or $A \rightarrow \beta$
- define $\mathrm{FIRST}(\alpha)$ as the set of tokens that can be first symbol of α, i.e.,

  $a \in \mathrm{FIRST}(\alpha)$ iff $\alpha \rightarrow^* a\gamma$ for some γ

Lookahead

- How to pick right production?
- If we have $A \rightarrow \alpha \mid \beta$, then we want to correctly choose either $A \rightarrow \alpha$ or $A \rightarrow \beta$
- define FIRST(α) as the set of tokens that can be first symbol of α, i.e.,

  $a \in \text{FIRST}(\alpha)$ iff $\alpha \rightarrow^* a\gamma$ for some γ
- If $A \rightarrow \alpha \mid \beta$ we want:

  $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$
- If that is always true, we can build a predictive parser.

Computing FIRST(α)

- Given X := A B C, FIRST(X) = FIRST(A B C)
- Can we ignore B or C?
- Consider:

```
A := a
   |
B := b
   | A
C := c
```

Computing FIRST(α)

- Given X := A B C, FIRST(X) = FIRST(A B C)
- Can we ignore B or C?
- Consider:

  A := a
  A :=
  B := b
  B := A
  C := c

- FIRST(X) must also include FIRST(C)
- IOW:
  - Must keep track of NTs that are nullable
  - For nullable NTs, determine FOLLOWS(NT)

nullable(A)

- nullable(A) is
  - true if A can derive the empty string
  - false otherwise
- For example:

B := X Y b

X := x

| Y Y

Y :=

In this case, nullable(X) = nullable(Y) = true

nullable(B) = false

FOLLOW(A)

- FOLLOW(A) is the set of terminals that can immediately follow A in a sentential form.
- I.e.,
  a $\in$ FOLLOW(A) iff S $\Rightarrow^*$ αAaβ for some α and β

Building a Predictive Parser

- We want to know for each non-terminal which production to choose based on the next input character.
- Build a table with rows labeled by non-terminals, A, and columns labeled by terminals, a. We will put the production, $A := \alpha$, in (A, a) iff
  - FIRST(α) contains a or
  - nullable(α) and FOLLOW(A) contains a

The table for the robot

S := B S F
  |
B := b
F := f

	FIRST	FOLLOW	nullable
S	b	$	yes
B	b	b,f	no
F	f	f,$	no

	b	f	$
S			
B			
F			

The table for the robot

S := B S F
 |
B := b
F := f

FIRST(BSF) = b

	FIRST	FOLLOW	nullable
S	b	$	yes
B	b	b,f	no
F	f	f,$	no

nullable(ε)=true
and
FOLLOW(S) = $

	b	f	$
S	S:=BSF		S:=
B	B:=b		
F		F:=f	

Table for exp grammar

1	S := E
2	E := T E′
2′	E′ := + T E′
3′	E′ := - T E′
4′	E′ :=
5	T := F T′
5′	T′ := * F T′
6′	T′ := / F T′
7′	T′ :=
8	F := `id`
9	F := `int`

	FIRST	FOLLOW	nullable
S	id, int	$	
E	id, int	$	
E′	+, -	$	yes
T	id, int	+,-,$	
T′	/, *	+,-,$	yes
F	id, int	/, *,$	

	+	-	*	/	id	int	$
S							
E							
E′							
T							
T′							
F							

Table for exp grammar

1	S := E	
2	E := T E′	
2′	E′ := + T E′	
3′	E′ := - T E′	
4′	E′ :=	
5	T := F T′	
5′	T′ := * F T′	
6′	T′ := / F T′	
7′	T′ :=	
8	F := `id`	
9	F := `int`	

	FIRST	FOLLOW	nullable
S	id, int	$	
E	id, int	$	
E′	+, -	$	yes
T	id, int	+,-,$	
T′	/, *	+,-,$	yes
F	id, int	/, *,$	

	+	-	*	/	id	int	$
S					:=E	:=E	
E					:=TE′	:=TE′	
E′	:=+TE′	:=-TE′					:=
T					:=FT′	:=FT′	
T′	:=	:=	:=*FT′	:=/FT′			:=
F					:=id	:=int	

Using the Table

- Each row in the table becomes a function
- For each input token with an entry:
  Create a series of invocations that implement the production, where
  - a non-terminal is eaten
  - a terminal becomes a recursive call
- For the blank cells implement errors

Example function

	+	-	*	/	id	int	$
S					:=E	:=E	
E					:=TE'	:=TE'	
E'	:=+TE'	:=-TE'			:=TE'	:=TE'	:=
T							
T'	:=	:=	:=*FT'				
F							

How to handle errors?

```
Eprime() {
    switch (token) {
    case PLUS:     eat(PLUS); T(); Eprime(); break;
    case MINUS:    eat(MINUS); T(); Eprime(); break;
    case ID:       T(); Eprime();
    case INT:      T(); Eprime();
    default:       error();
}
```

Left-Factoring

- Predictive parsers need to make a choice based on the next terminal.
- Consider:

S := **`if E then S else S`**
  | **`if E then S`**

- When looking at **`if`**, can't decide
- so left-factor the grammar

S := **`if E then S X`**
X := **`else S`**
  |

Top-Down Parsing

- Can be constructed by hand
- LL(k) grammars can be parsed
  - Left-to-right
  - Leftmost-derivation
  - with k symbols lookahead
- Often requires
  - left-factoring
  - Elimination of left-recursion

Bottom-up parsers

- What is the inherent restriction of top-down parsing, e.g., with LL(k) grammars?

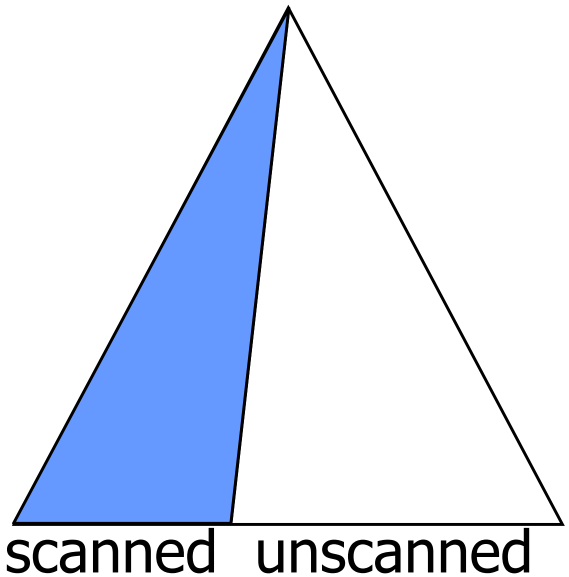
Bottom-up parsers

- What is the inherent restriction of top-down parsing, e.g., with LL(k) grammars?
- Bottom-up parsers use the entire right-hand side of the production
- LR(k):
  - Left-to-right parse,
  - Rightmost derivation (in reverse),
  - k look ahead tokens

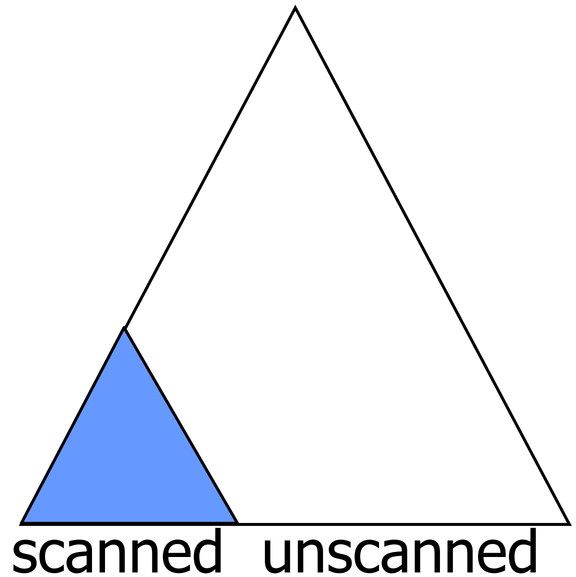
Top-down vs. Bottom-up

LL(k), recursive descent

LR(k), shift-reduce



Top-down



Bottom-up

Example - Top-down

S := X
X := X a
  | b

Is this grammar LL(k)?

How can we make it LL(k)?

S := X
X := b R
R := a R
  |

What about a bottom up parse?

Example - Bottom-up

S := X

X := X a

| b

right-most derivation:

$S \Rightarrow X \Rightarrow Xa \Rightarrow Xaa \Rightarrow baa$

Left-to-Right, Rightmost in reverse

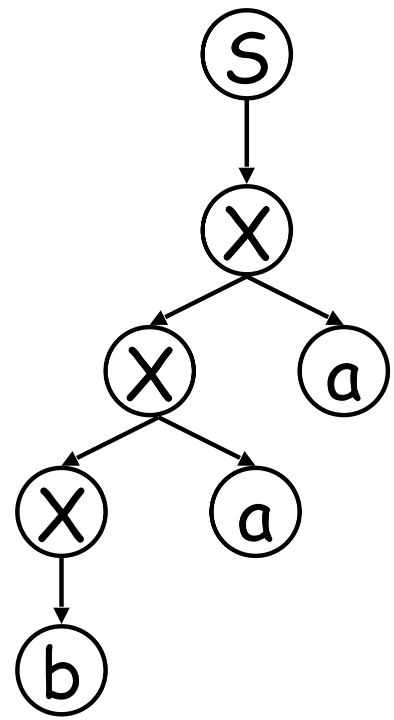
baa

Xaa

Xa

X

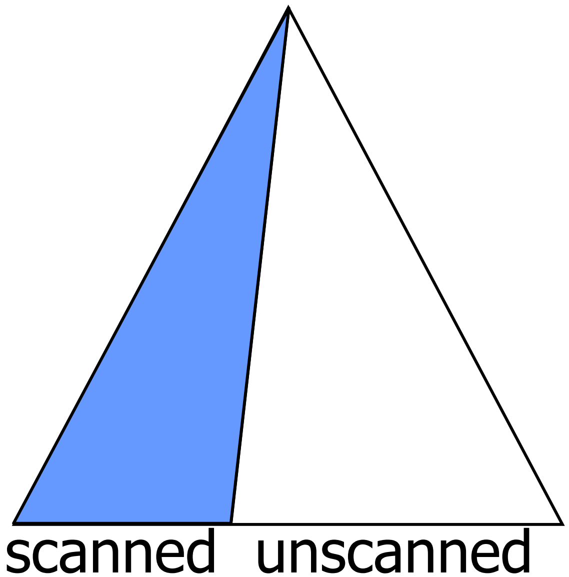
S



LR parser gets to look at an entire right hand side.

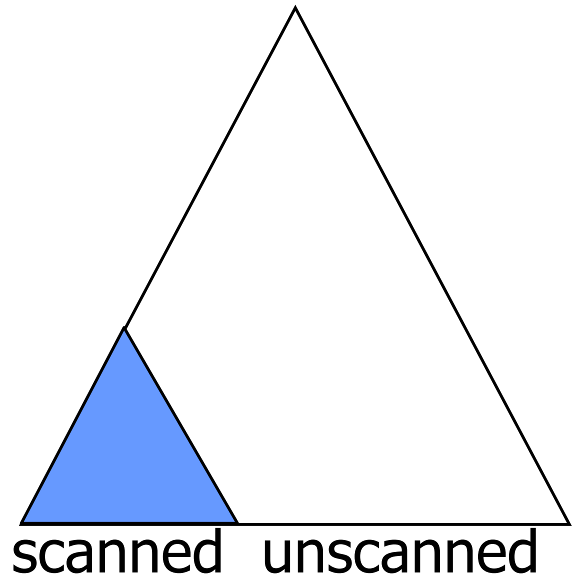
Top-down vs. Bottom-up

LL(k), recursive descent



Top-down

LR(k), shift-reduce



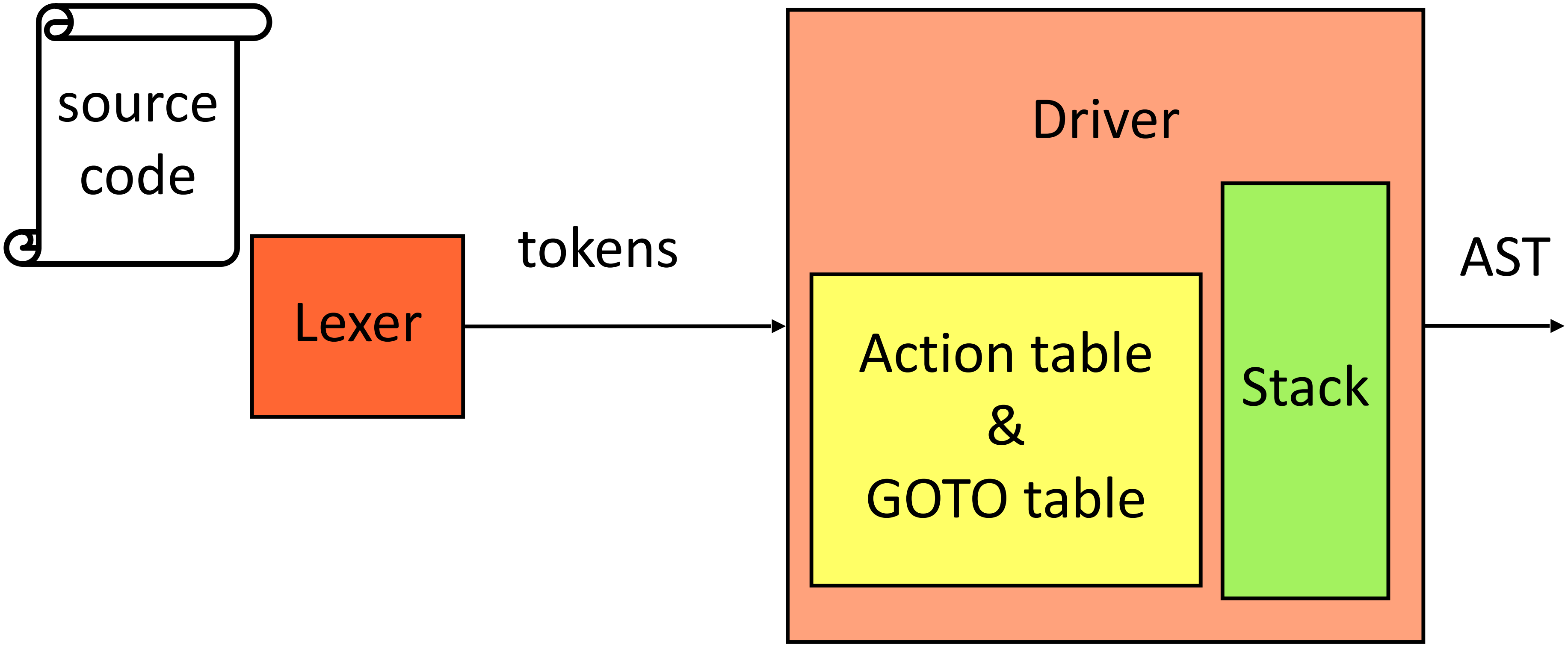
Bottom-up

A Shift-Reduce Parser

- Implement as a FSM with a stack
- Stack holds sequences of symbols
- Input stream holds remaining source
- Four actions:
  - shift: push token from input stream onto stack
  - reduce: right-end of a handle (β of $A \rightarrow \beta$) is at top of stack, pop handle (β), push A
  - accept: success
  - error: syntax error discovered

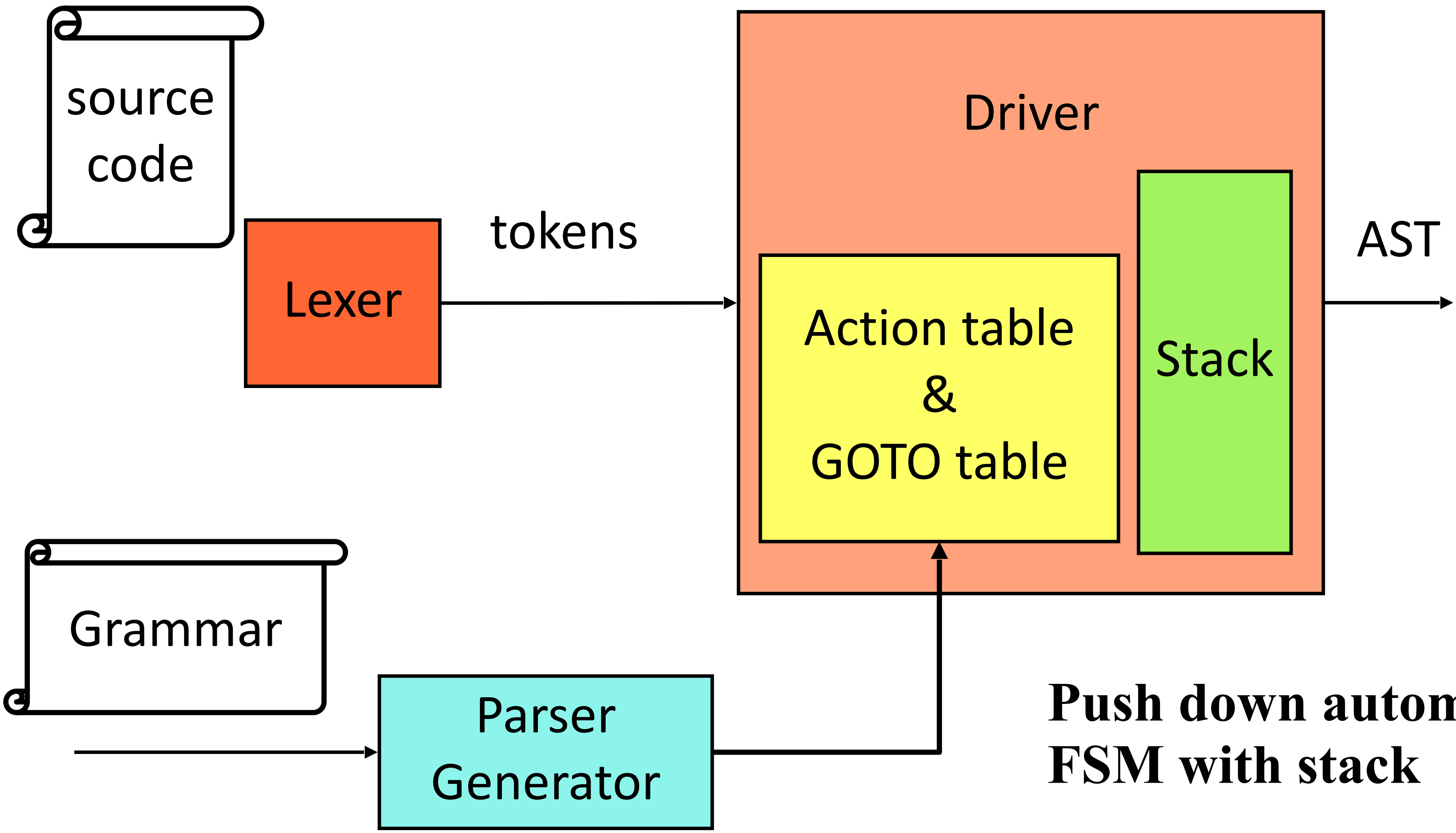
Key is recognizing handles efficiently

Table-driven LR(k) parsers



Push down automata:
FSM with stack

Table-driven LR(k) parsers



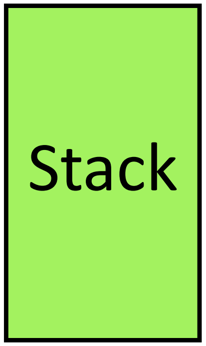
**Push down automata:
FSM with stack**

Parser Loop

Driver

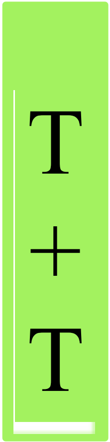
- Same code regardless of grammar
  - only tables change
- (Very) General Algorithm:
  - Based on table contents, top of stack, and current input token either
    - **shift**: push token onto stack and read next token
    - **reduce**: replace part of stack with the correct rule (NT) that derived it
    - **accept**: successfully parsed entire input
    - **error**: input not in language

Stack



- Represents the input parsed so far
- Contents?
  - Symbols: terminals (and non-terminals)
  - Must also store previously seen *states*
    - the context of the current position
  - In fact, nonterminals unnecessary
    - include for readability

x + y• + z



Parser Tables

Action table
&
GOTO table

Action table

- given state *s* and **terminal** *a* tells parser loop what action (shift, reduce, accept, reject) to perform

Goto table

- used when performing reduction; given a state *s* and **nonterminal** *X* says what state to transition to

Parser Tables

Action table
&
GOTO table

s*N* push state *N* onto stack

r*R* reduce by rule *R*

g*N* goto state *N*

a accept

error

state	action: *ident*	action: +	action: $	goto: E	goto: T
0	**s3**			**g1**	**g2**
1			**a**		
2		**s4**	**r2**		
3		**r3**	**r3**		
4	**s3**			**g5**	**g2**
5			**r1**		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Parser Loop Revisited

Driver

```
while(true)
   s = state on top of stack
   a = current input token
   if(action[s][a] == sN)                        shift
      push N
      read next input token
   else if(action[s][a] == rR)                   reduce
      pop rhs of rule R from stack
      X = lhs of rule R
      N = state on top of stack
      push goto[N][X]
   else if(action[s][a] == a)                    accept
      return success
   else                                    error
      return failure
```

Example

state	action			goto	
	ident	+	$	E	T
0	**s3**			**g1**	**g2**
1			**a**		
2		**s4**	**r2**		
3		**r3**	**r3**		
4	**s3**			**g5**	**g2**
5			**r1**		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **x**

State on top of the stack = **0**

(0,S)

Stack

x + **y$**

Example

state	action			goto	
	ident	+	$	E	T
0	**s3**			**g1**	**g2**
1			**a**		
2		**s4**	**r2**		
3		**r3**	**r3**		
4	**s3**			**g5**	**g2**
5			**r1**		

0 S ⟶ E$

1 E ⟶ T + E

2 E ⟶ T

3 T ⟶ *identifier*

Current input token = **+**

State on top of the stack = **3**

(3,x)

(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **3**

(3,x)
(0,S)

x + y$

Example

	action			goto	
state	*ident*	+	$	E	T
0	**s3**			**g1**	**g2**
1			**a**		
2		**s4**	**r2**		
3		**r3**	**r3**		
4	**s3**			**g5**	**g2**
5			**r1**		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **3**

(3,x)

(0,S)

x + y$

Example

	action			goto	
state	*ident*	+	$	E	T
0	**s3**			**g1**	**g2**
1			**a**		
2		**s4**	**r2**		
3		**r3**	**r3**		
4	**s3**			**g5**	**g2**
5			**r1**		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Current input token = **+**
State on top of the stack = **0**

(3,x)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **2**

(2,T)

(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **+**

State on top of the stack = **2**

(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Current input token = **y**
State on top of the stack = **4**

(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **y**

State on top of the stack = **4**

(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **3**

(3,y)
(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	**s3**			**g1**	**g2**
1			**a**		
2		**s4**	**r2**		
3		**r3**	**r3**		
4	**s3**			**g5**	**g2**
5			**r1**		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **3**

(?,T)

(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **2**

(2,T)
(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow$ *identifier*

Current input token = **$**

State on top of the stack = **2**

(2,T)
(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S ⟶ E$

1 E ⟶ T + E

2 E ⟶ T

3 T ⟶ *identifier*

Current input token = **$**

State on top of the stack = **2**

(?,E)

(4,+)

(2,T)

(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow \textit{identifier}$

Current input token = $

State on top of the stack = 5

(5,E)
(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **5**

(5,E)
(4,+)
(2,T)
(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	**s3**			**g1**	**g2**
1			**a**		
2		**s4**	**r2**		
3		**r3**	**r3**		
4	**s3**			**g5**	**g2**
5			**r1**		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **5**

x + y$

(5,E)

(4,+)

(2,T)

(0,S)

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

Current input token = **$**

State on top of the stack = **1**

(1,E)

(0,S)

x + y$

Example

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

Accept!

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Current input token = **$**
State on top of the stack = **1**

(1,E)
(0,S)

x + y$

A Rightmost Derivation

1 S := Exp
2 Exp := Exp + Term
3 Exp := Exp - Term
4 Exp := Term
5 Term := Term * Factor
6 Term := Term / Factor
7 Term := Factor
8 Factor := **id**
9 Factor := **int**

input: 2+3*x

S
by 1 ⇒ Exp
by 2 ⇒ Exp + Term
by 5 ⇒ Exp + Term * Factor
by 8 ⇒ Exp + Term * $\mathbf{id}_x$
by 7 ⇒ Exp + Factor * $\mathbf{id}_x$
by 9 ⇒ Exp + $\mathbf{int}_3$ * $\mathbf{id}_x$
by 4 ⇒ Term + $\mathbf{int}_3$ * $\mathbf{id}_x$
by 7 ⇒ Factor + $\mathbf{int}_3$ * $\mathbf{id}_x$
by 9 ⇒ $\mathbf{int}_2$ + $\mathbf{int}_3$ * $\mathbf{id}_x$

A Rightmost Derivation In Reverse

int$_2$ + **int**$_3$ * **id**$_x$

Factor + **int**$_3$ * **id**$_x$

Term

Lets keep track of where we are in the input.

Exp + **int**$_3$ * **id**$_x$

Exp + Factor * **id**$_x$

Exp + Term * **id**$_x$

Exp + Term * Factor

Exp + Term

Exp

S

A Rightmost Derivation In Reverse

$\texttt{int}_2$ + $\texttt{int}_3$ * $\texttt{id}_x$

Factor + $\texttt{int}_3$ * $\texttt{id}_x$

Term + $\texttt{int}_3$ * $\texttt{id}_x$

Exp + $\texttt{int}_3$ * $\texttt{id}_x$

Exp + Factor * $\texttt{id}_x$

Exp + Term * $\texttt{id}_x$

Exp + Term * Factor

Exp + Term

Exp

S

$\texttt{int}_2$ ● + $\texttt{int}_3$ * $\texttt{id}_x$

Factor ● + $\texttt{int}_3$ * $\texttt{id}_x$

Term ● + $\texttt{int}_3$ * $\texttt{id}_x$

Exp + $\texttt{int}_3$ ● * $\texttt{id}_x$

Exp + Factor ● * $\texttt{id}_x$

Exp + Term * $\texttt{id}_x$ ●

Exp + Term * Factor ●

Exp + Term ●

Exp ●

S ●

A Rightmost Derivation In Reverse

int$_2$ + int$_3$ * id$_x$

Factor + int$_3$ * id$_x$

Term + int$_3$ * id$_x$

Exp + int$_3$ * id$_x$

Exp + Factor * id$_x$

Exp + Term *

Exp + Term * Factor

Exp + Term

Exp

S

int$_2$ • + int$_3$ * id$_x$

Factor • + int$_3$ * id$_x$

Term • + int$_3$ * id$_x$

Exp + int$_3$ • * id$_x$

Exp + Factor • * id$_x$

* id$_x$ •

* Factor •

Exp + Term •

Exp •

S •

Lets format this differently,
<prefix of sentential form> input

A Rightmost Derivation In Reverse

	$\texttt{int}_2$ + $\texttt{int}_3$ * $\texttt{id}_x$ \$
$\texttt{int}_2$	+ $\texttt{int}_3$ * $\texttt{id}_x$ \$
Factor	+ $\texttt{int}_3$ * $\texttt{id}_x$ \$
Term	+ $\texttt{int}_3$ * $\texttt{id}_x$ \$
Exp	+ $\texttt{int}_3$ * $\texttt{id}_x$ \$
Exp +	$\texttt{int}_3$ * $\texttt{id}_x$ \$
Exp + $\texttt{int}_3$	* $\texttt{id}_x$ \$
Exp + Factor	* $\texttt{id}_x$ \$
Exp + Term	* $\texttt{id}_x$ \$
Exp + Term *	$\texttt{id}_x$ \$
Exp + Term * $\texttt{id}_x$	\$
Exp + Term * Factor	\$
Exp + Term	\$
Exp	\$
S	\$

A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x$ \$
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$
Term	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$
Exp +	$\mathtt{int}_3 * \mathtt{id}_x$ \$
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x$ \$
Exp + Factor	$* \mathtt{id}_x$ \$
Exp + Term	$* \mathtt{id}_x$ \$
Exp + Term *	$\mathtt{id}_x$ \$
Exp + Term * $\mathtt{id}_x$	\$
S	\$

LR-Parser either:

1. shifts a terminal or
2. reduces by a production.

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	
Factor	+ **int**$_3$ * **id**$_x$ $	
Term	+ **int**$_3$ * **id**$_x$ $	
Exp	+ **int**$_3$ * **id**$_x$ $	
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor		
Term		
Exp		
Exp +		
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

When we reduce by a production: A → β,
β is "popped" off the end of the parsed input.

E.g., here β is 'int' and production is F → int

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$ \$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x$ \$	
Exp	$+ \texttt{int}_3 * \texttt{id}_x$ \$	
Exp +	$\texttt{int}_3 * \texttt{id}_x$ \$	
Exp + $\texttt{int}_3$	$* \texttt{id}_x$ \$	
Exp + Factor	$* \texttt{id}_x$ \$	
Exp + Term	$* \texttt{id}_x$ \$	
Exp + Term *	$\texttt{id}_x$ \$	
Exp + Term * $\texttt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by E → T
Exp	+ **int**$_3$ * **id**$_x$ $	
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x \ \$$	shift 2
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	reduce by F → int
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	reduce by T → F
Term	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	reduce by E → T
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x \ \$$	shift +
Exp +	$\mathtt{int}_3 * \mathtt{id}_x \ \$$	
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x \ \$$	
Exp + Factor	$* \mathtt{id}_x \ \$$	
Exp + Term	$* \mathtt{id}_x \ \$$	
Exp + Term *	$\mathtt{id}_x \ \$$	
Exp + Term * $\mathtt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$ \$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by E → T
Exp	$+ \texttt{int}_3 * \texttt{id}_x$ \$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x$ \$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x$ \$	
Exp + Factor	$* \texttt{id}_x$ \$	
Exp + Term	$* \texttt{id}_x$ \$	
Exp + Term *	$\texttt{id}_x$ \$	
Exp + Term * $\texttt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	

A Rightmost Derivation In Reverse

	`int`$_2$ + `int`$_3$ * `id`$_x$ $	shift 2
`int`$_2$	+ `int`$_3$ * `id`$_x$ $	reduce by F → int
Factor	+ `int`$_3$ * `id`$_x$ $	reduce by T → F
Term	+ `int`$_3$ * `id`$_x$ $	reduce by E → T
Exp	+ `int`$_3$ * `id`$_x$ $	shift +
Exp +	`int`$_3$ * `id`$_x$ $	shift 3
Exp + `int`$_3$	* `id`$_x$ $	reduce by F → int
Exp + Factor	* `id`$_x$ $	
Exp + Term	* `id`$_x$ $	
Exp + Term *	`id`$_x$ $	
Exp + Term * `id`$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$ \$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by E → T
Exp	$+ \texttt{int}_3 * \texttt{id}_x$ \$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x$ \$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x$ \$	reduce by F → int
Exp + Factor	$* \texttt{id}_x$ \$	reduce by T → F
Exp + Term	$* \texttt{id}_x$ \$	
Exp + Term *	$\texttt{id}_x$ \$	
Exp + Term * $\texttt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by E → T
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	reduce by T → F
Exp + Term	* **id**$_x$ $	shift *
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2$ + $\texttt{int}_3$ * $\texttt{id}_x$ $	shift 2
$\texttt{int}_2$	+ $\texttt{int}_3$ * $\texttt{id}_x$ $	reduce by F → int
Factor	+ $\texttt{int}_3$ * $\texttt{id}_x$ $	reduce by T → F
Term	+ $\texttt{int}_3$ * $\texttt{id}_x$ $	reduce by E → T
Exp	+ $\texttt{int}_3$ * $\texttt{id}_x$ $	shift +
Exp +	$\texttt{int}_3$ * $\texttt{id}_x$ $	shift 3
Exp + $\texttt{int}_3$	* $\texttt{id}_x$ $	reduce by F → int
Exp + Factor	* $\texttt{id}_x$ $	reduce by T → F
Exp + Term	* $\texttt{id}_x$ $	shift *
Exp + Term *	$\texttt{id}_x$ $	shift x
Exp + Term * $\texttt{id}_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	`int`$_2$ + `int`$_3$ * `id`$_x$ \$	shift 2
`int`$_2$	+ `int`$_3$ * `id`$_x$ \$	reduce by F → int
Factor	+ `int`$_3$ * `id`$_x$ \$	reduce by T → F
Term	+ `int`$_3$ * `id`$_x$ \$	reduce by E → T
Exp	+ `int`$_3$ * `id`$_x$ \$	shift +
Exp +	`int`$_3$ * `id`$_x$ \$	shift 3
Exp + `int`$_3$	* `id`$_x$ \$	reduce by F → int
Exp + Factor	* `id`$_x$ \$	reduce by T → F
Exp + Term	* `id`$_x$ \$	shift *
Exp + Term *	`id`$_x$ \$	shift x
Exp + Term * `id`$_x$	\$	reduce by F → id
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ \$	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ \$	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ \$	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ \$	reduce by E → T
Exp	+ **int**$_3$ * **id**$_x$ \$	shift +
Exp +	**int**$_3$ * **id**$_x$ \$	shift 3
Exp + **int**$_3$	* **id**$_x$ \$	reduce by F → int
Exp + Factor	* **id**$_x$ \$	reduce by T → F
Exp + Term	* **id**$_x$ \$	shift *
Exp + Term *	**id**$_x$ \$	shift x
Exp + Term * **id**$_x$	\$	reduce by F → id
Exp + Term * Factor	\$	reduce by T → T * F
Exp + Term	\$	
Exp	\$	
S	\$	

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by E → T
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	reduce by T → F
Exp + Term	* **id**$_x$ $	shift *
Exp + Term *	**id**$_x$ $	shift x
Exp + Term * **id**$_x$	$	reduce by F → id
Exp + Term * Factor	$	reduce by T → T * F
Exp + Term	$	reduce by E → E + T
Exp	$	
S	$	

A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$ \$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by E → T
Exp	$+ \texttt{int}_3 * \texttt{id}_x$ \$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x$ \$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x$ \$	reduce by F → int
Exp + Factor	$* \texttt{id}_x$ \$	reduce by T → F
Exp + Term	$* \texttt{id}_x$ \$	shift *
Exp + Term *	$\texttt{id}_x$ \$	shift x
Exp + Term * $\texttt{id}_x$	\$	reduce by F → id
Exp + Term * Factor	\$	reduce by T → T * F
Exp + Term	\$	reduce by E → E + T
Exp	\$	reduce by S → E
S	\$	

A Rightmost Derivation In Reverse

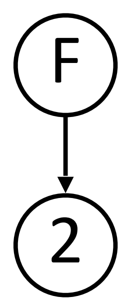
	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$ \$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by E → T
Exp	$+ \texttt{int}_3 * \texttt{id}_x$ \$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x$ \$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x$ \$	reduce by F → int
Exp + Factor	$* \texttt{id}_x$ \$	reduce by T → F
Exp + Term	$* \texttt{id}_x$ \$	shift *
Exp + Term *	$\texttt{id}_x$ \$	shift x
Exp + Term * $\texttt{id}_x$	\$	reduce by F → id
Exp + Term * Factor	\$	reduce by T → T * F
Exp + Term	\$	reduce by E → E + T
Exp	\$	reduce by S → E
S	\$	accept!

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	
Factor	+ **int**$_3$ * **id**$_x$ $	
Term	+ **int**$_3$ * **id**$_x$ $	
Exp	+ **int**$_3$ * **id**$_x$ $	
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	2

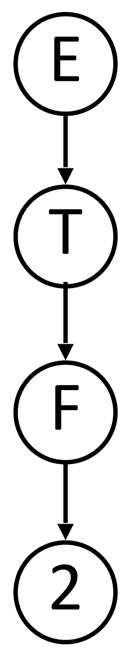
A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	
Term	+ **int**$_3$ * **id**$_x$ $	
Exp	+ **int**$_3$ * **id**$_x$ $	
Exp +	**int**$_3$ * **id**$_x$ $	
Exp + **int**$_3$	* **id**$_x$ $	
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	



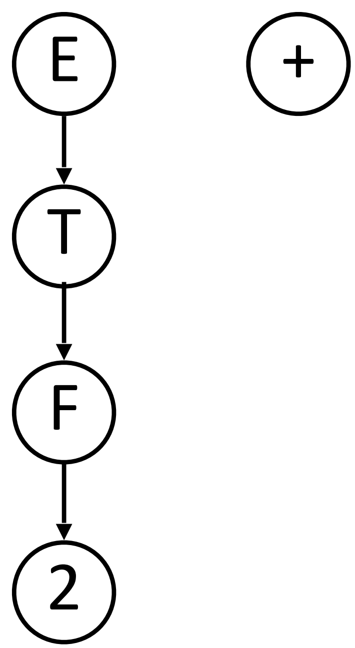
A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x \ \$$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	reduce by E → T
Exp	$+ \texttt{int}_3 * \texttt{id}_x \ \$$	
Exp +	$\texttt{int}_3 * \texttt{id}_x \ \$$	
Exp + $\texttt{int}_3$	$* \texttt{id}_x \ \$$	
Exp + Factor	$* \texttt{id}_x \ \$$	
Exp + Term	$* \texttt{id}_x \ \$$	
Exp + Term *	$\texttt{id}_x \ \$$	
Exp + Term * $\texttt{id}_x$	$\$$	
Exp + Term * Factor	$\$$	
Exp + Term	$\$$	
Exp	$\$$	
S	$\$$	



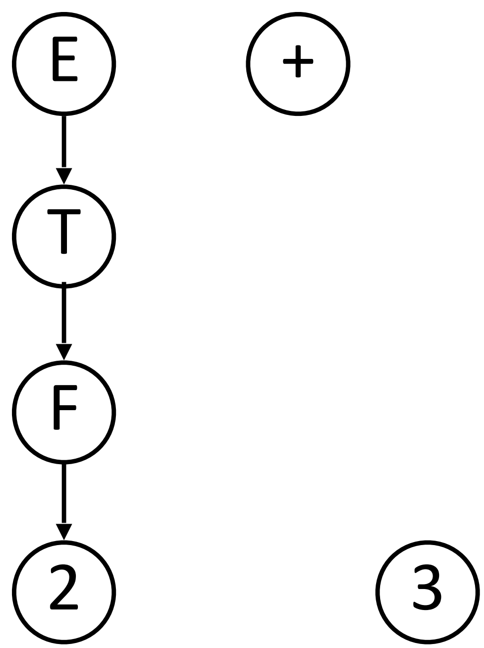
A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ \$	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ \$	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ \$	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ \$	reduce by E → T
Exp	+ **int**$_3$ * **id**$_x$ \$	shift +
Exp +	**int**$_3$ * **id**$_x$ \$	
Exp + **int**$_3$	* **id**$_x$ \$	
Exp + Factor	* **id**$_x$ \$	
Exp + Term	* **id**$_x$ \$	
Exp + Term *	**id**$_x$ \$	
Exp + Term * **id**$_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	



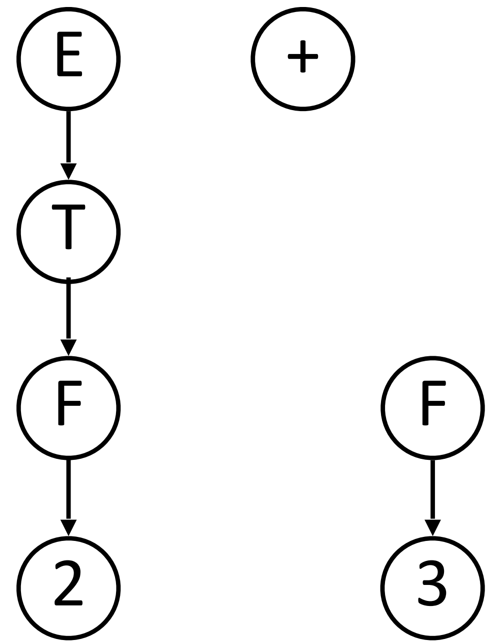
A Rightmost Derivation In Reverse

	$\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x$ \$	shift 2
$\mathtt{int}_2$	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	reduce by F → int
Factor	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	reduce by T → F
Term	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	reduce by E → T
Exp	$+ \mathtt{int}_3 * \mathtt{id}_x$ \$	shift +
Exp +	$\mathtt{int}_3 * \mathtt{id}_x$ \$	shift 3
Exp + $\mathtt{int}_3$	$* \mathtt{id}_x$ \$	
Exp + Factor	$* \mathtt{id}_x$ \$	
Exp + Term	$* \mathtt{id}_x$ \$	
Exp + Term *	$\mathtt{id}_x$ \$	
Exp + Term * $\mathtt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	



A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$ \$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by E → T
Exp	$+ \texttt{int}_3 * \texttt{id}_x$ \$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x$ \$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x$ \$	reduce by F → int
Exp + Factor	$* \texttt{id}_x$ \$	
Exp + Term	$* \texttt{id}_x$ \$	
Exp + Term *	$\texttt{id}_x$ \$	
Exp + Term * $\texttt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	



Handles

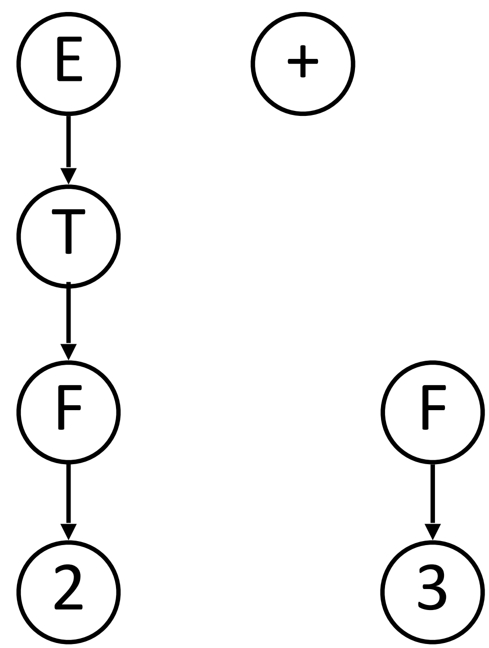
- LR parsing is handle pruning
- LR parsing finds a rightmost derivation (in reverse)
- A handle in γ, a right-hand sentential form, is
  - a position in γ matching β
  - a production $A \rightarrow \beta$

$$S \rightarrow^* \alpha A w \rightarrow \alpha\beta w$$

- if a grammar is unambiguous, then every γ has exactly 1 handle

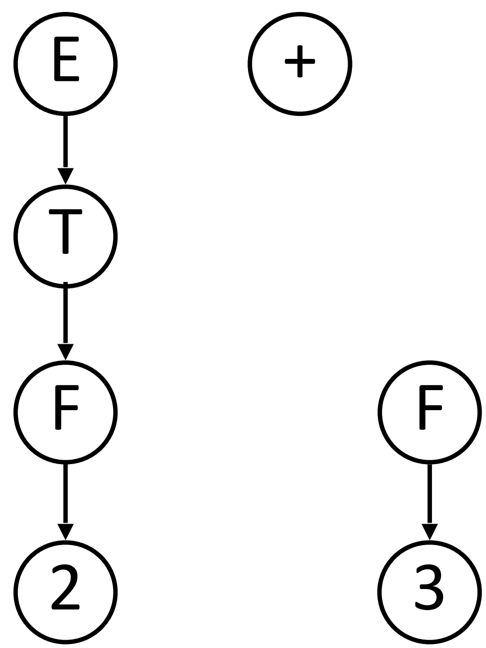
A Rightmost Derivation In Reverse

	$\texttt{int}_2 + \texttt{int}_3 * \texttt{id}_x$ \$	shift 2
$\texttt{int}_2$	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by F → int
Factor	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by T → F
Term	$+ \texttt{int}_3 * \texttt{id}_x$ \$	reduce by E → T
Exp	$+ \texttt{int}_3 * \texttt{id}_x$ \$	shift +
Exp +	$\texttt{int}_3 * \texttt{id}_x$ \$	shift 3
Exp + $\texttt{int}_3$	$* \texttt{id}_x$ \$	reduce by F → int
Exp + Factor	$* \texttt{id}_x$ \$	
Exp + Term	$* \texttt{id}_x$ \$	
Exp + Term *	$\texttt{id}_x$ \$	
Exp + Term * $\texttt{id}_x$	\$	
Exp + Term * Factor	\$	
Exp + Term	\$	
Exp	\$	
S	\$	



A Rightmost Derivation In Reverse

Where is next handle?		
	int$_2$ + **int**$_3$ * **id**$_x$ $	shift 2
int$_2$	+ **int**$_3$ * **id**$_x$ $	reduce by F → int
Factor	+ **int**$_3$ * **id**$_x$ $	reduce by T → F
Term	+ **int**$_3$ * **id**$_x$ $	reduce by E → T
Exp	+ **int**$_3$ * **id**$_x$ $	shift +
Exp +	**int**$_3$ * **id**$_x$ $	shift 3
Exp + **int**$_3$	* **id**$_x$ $	reduce by F → int
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	

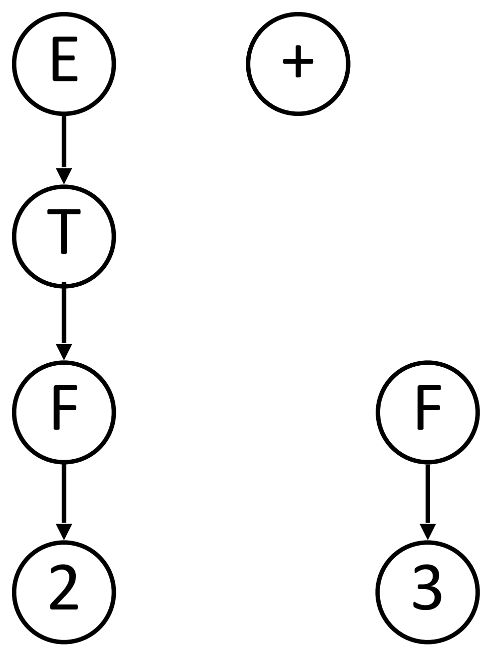


A Rightmos… …everse

1	S	:= Exp
2	Exp	:= Exp + Term
3	Exp	:= Exp - Term
4	Exp	:= Term
5	Term	:= Term * Factor
6	Term	:= Term / Factor
7	Term	:= Factor
8	Factor	:= **id**
9	Factor	:= **int**

Where is next handle?

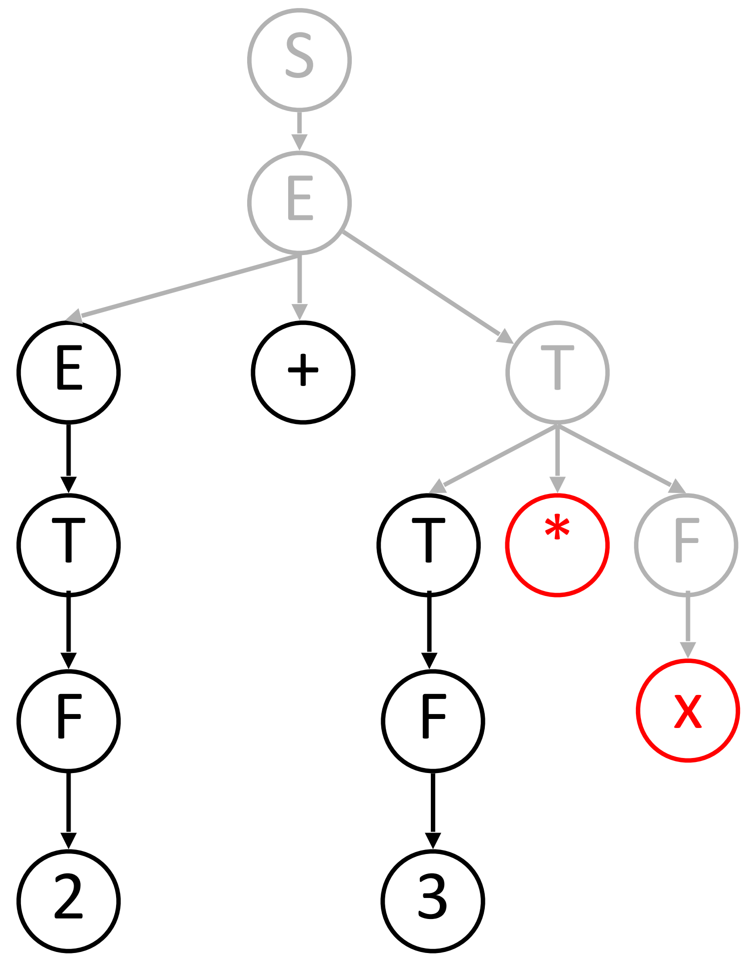
int$_2$		F → int
Factor		T → F
Term		E → T
Exp		
Exp +		
Exp + **int**$_3$		F → int
Exp + Factor	* **id**$_x$ $	
Exp + Term	* **id**$_x$ $	
Exp + Term *	**id**$_x$ $	
Exp + Term * **id**$_x$	$	
Exp + Term * Factor	$	
Exp + Term	$	
Exp	$	
S	$	



A Rightmost Derivation In Reverse

Where is next handle? E+F*x and T→ F

int$_2$	+ **int**$_3$ * **id**$_x$ $
Factor	+ **int**$_3$ * **id**$_x$ $
Term	+ **int**$_3$ * **id**$_x$ $
Exp	+ **int**$_3$ * **id**$_x$ $
Exp +	**int**$_3$ * **id**$_x$ $
Exp + **int**$_3$	* **id**$_x$ $
Exp + Factor	* **id**$_x$ $
Exp + Term	* **id**$_x$ $
Exp + Term *	**id**$_x$ $
Exp + Term * **id**$_x$	$
Exp + Term * Factor	$
Exp + Term	$
Exp	$
S	$



Handle Pruning

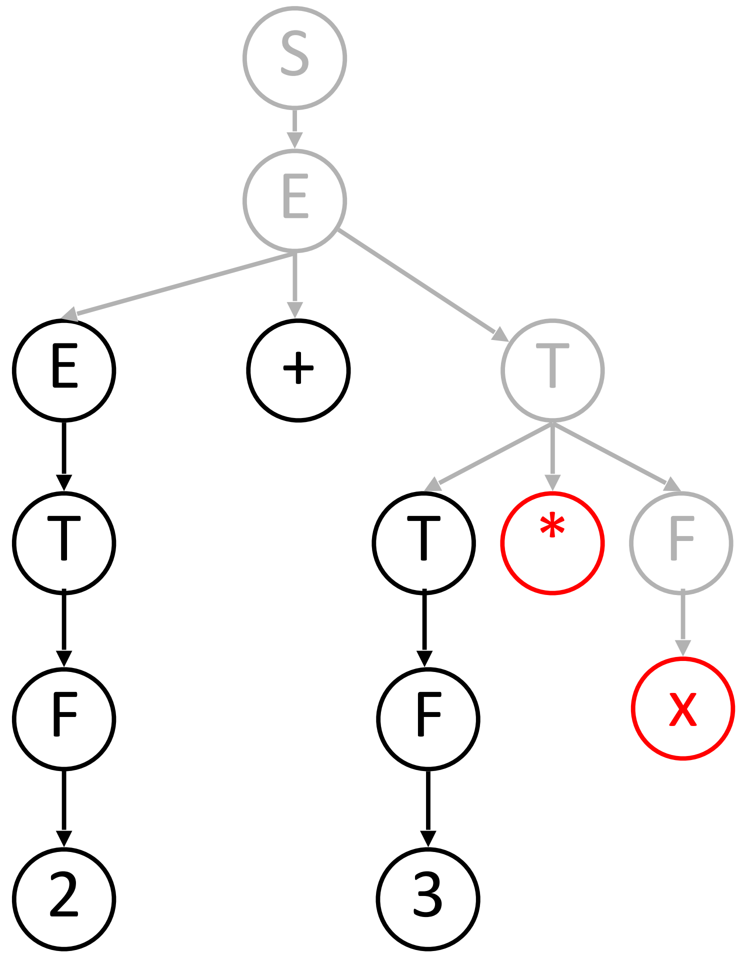
- LR parsing consists of
  - shifting until there is a handle on the top of the stack
  - reducing handle
- Key is handle is always on top of stack, i.e., if β is a handle with $A \rightarrow \beta$, then β can be found on top of stack.

A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ \$
int$_2$	+ **int**$_3$ * **id**$_x$ \$
Factor	+ **int**$_3$ * **id**$_x$ \$
Term	+ **int**$_3$ * **id**$_x$ \$
Exp	+ **int**$_3$ * **id**$_x$ \$
Exp +	**int**$_3$ * **id**$_x$ \$
Exp + **int**$_3$	* **id**$_x$ \$
Exp + Factor	* **id**$_x$ \$
Exp + Term	* **id**$_x$ \$

Exp + Term *	**id**$_x$ \$
Exp + Term * **id**$_x$	\$
Exp + Term * Factor	\$
Exp + Term	\$
Exp	\$
S	\$

top of stack does not have a handle, so must shift.



A Rightmost Derivation In Reverse

	int$_2$ + **int**$_3$ * **id**$_x$ $
int$_2$	+ **int**$_3$ * **id**$_x$ $
Factor	+ **int**$_3$ * **id**$_x$ $
Term	+ **int**$_3$ * **id**$_x$ $
Exp	+ **int**$_3$ * **id**$_x$ $
Exp +	**int**$_3$ * **id**$_x$ $
Exp + **int**$_3$	* **id**$_x$ $
Exp + Factor	* **id**$_x$ $
Exp + Term	* **id**$_x$ $
Exp + Term *	**id**$_x$ $
Exp + Term * **id**$_x$	$
Exp + Term * Factor	$
Exp + Term	$
Exp	$
S	$

Now, x is a handle.

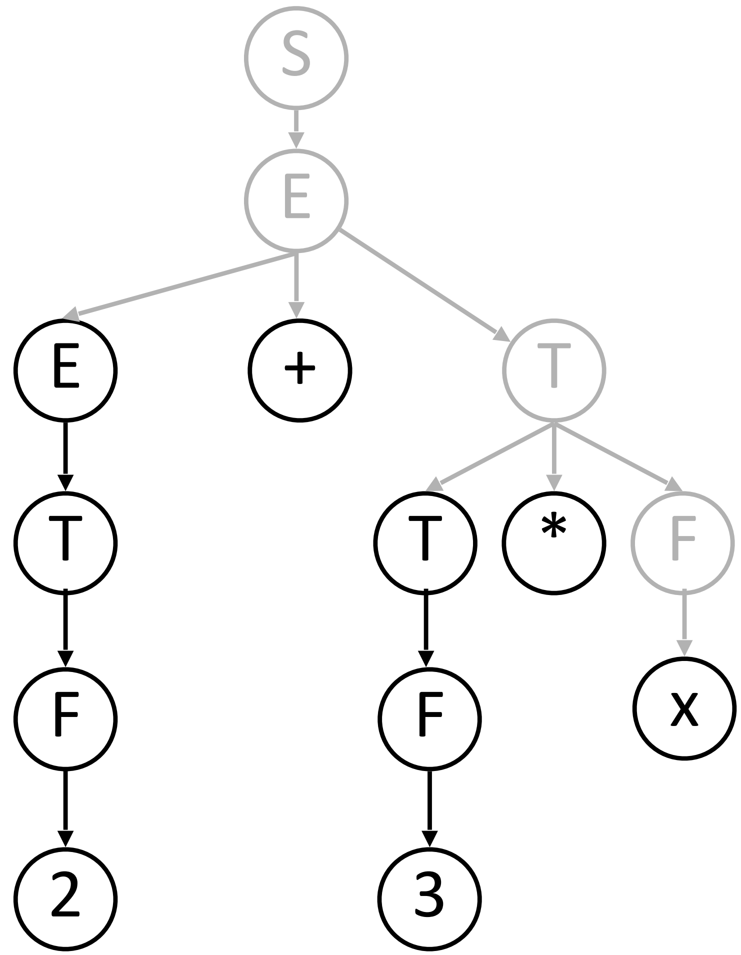
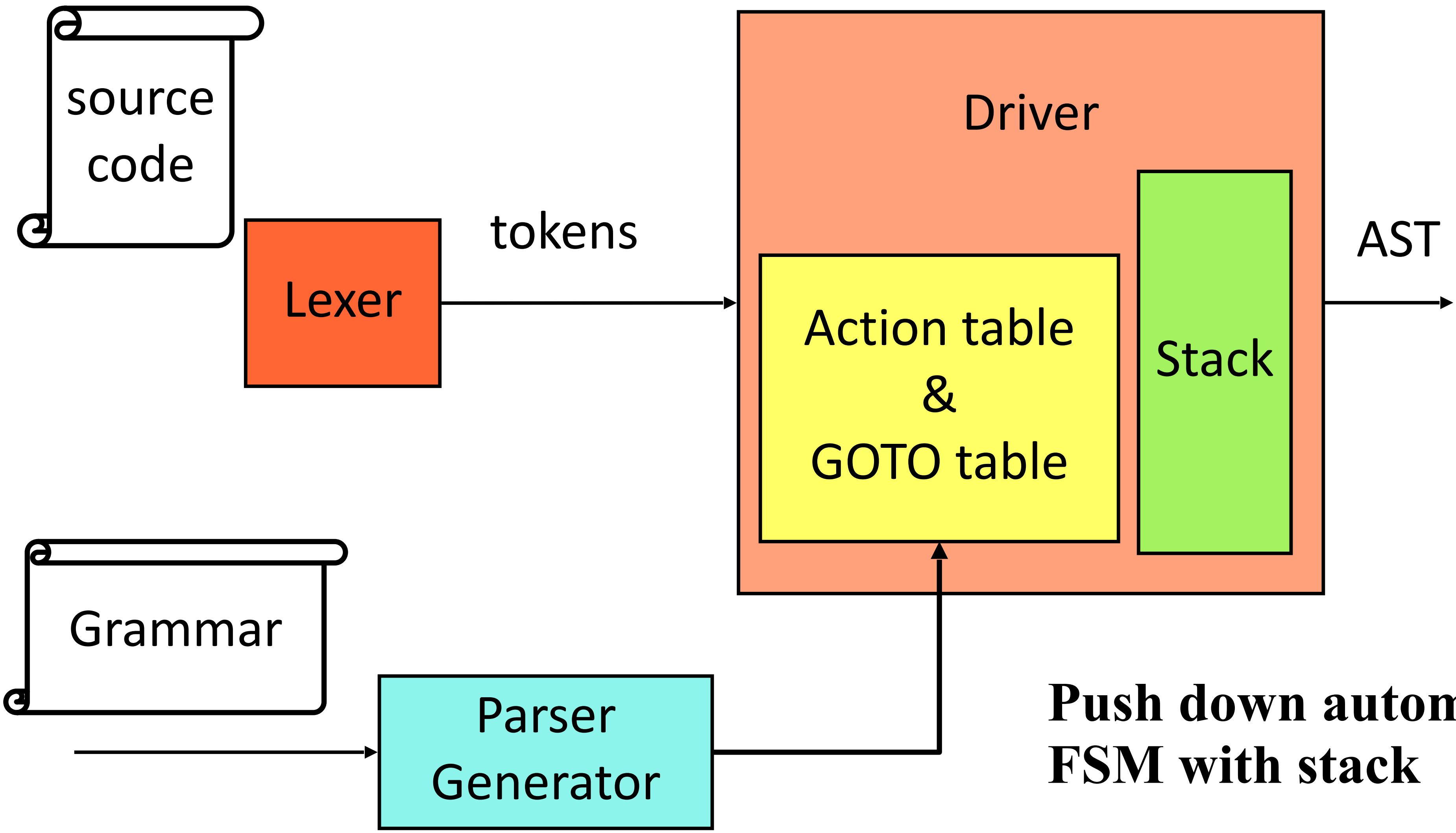


Table-driven LR(k) parsers



Push down automata:
FSM with stack

The parser generator

Parser Generator

- Finds handles
- Creates the action and GOTO tables.
- Creates the states
  - Each state indicates how much of a handle we have seen
  - each state is a set of *items*

Items

- Items are used to identify handles.
- LR(k) items have the form:

  [production-with-dot, lookahead]

- For example, $A \rightarrow a\ X\ b$ has 4 LR(0) items
  - [$A \rightarrow \bullet a\ X\ b$]
  - [$A \rightarrow a \bullet X\ b$]
  - [$A \rightarrow a\ X \bullet b$]
  - [$A \rightarrow a\ X\ b \bullet$]

The ● indicates how much of the handle we have recognized.

What LR(0) Items Mean

- [$X \rightarrow \bullet\ \alpha\ \beta\ \gamma$]
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$
- [$X \rightarrow \alpha \bullet \beta\ \gamma$]
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we have already recognized α
- [$X \rightarrow \alpha\ \beta \bullet \gamma$]
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we have already recognized $\alpha\ \beta$
- [$X \rightarrow \alpha\ \beta\ \gamma\ \bullet$]
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we can reduce to X

Generating the States

- Start with start production.
- In this case, "S ⟶ E$"

S ⟶ •E$

0 S ⟶ E$

1 E ⟶ T + E

2 E ⟶ T

3 T ⟶ *identifier*

- Each state is consistent with what we have already shifted from the input and what is possible to reduce. So, what other items should be in this state?

Completing a state

- For each item in a state, add in all other consistent items.

S → •E$
E → •T + E
E → •T
T → •*identifier*

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

- This is called, taking the closure of the state.

Closure*

```
closure(state)
   repeat
      foreach item A → a•Xb in state
         foreach production X → w
            state.add(X → •w)
   until state does not change
   return state
```

Intuitively:

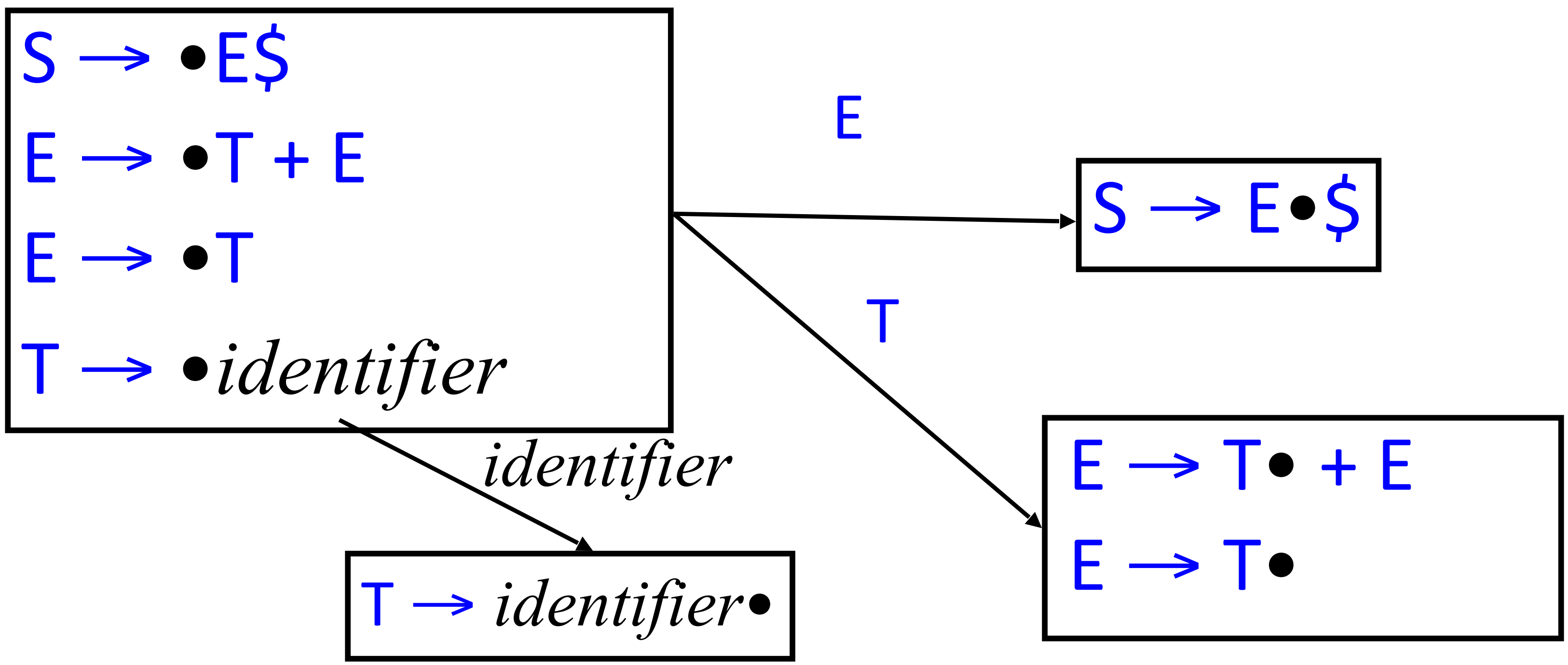
Given a set of items, add all production rules that could produce the nonterminal(s) at the current position in each item

*: for LR(0) items

What about the other states?

- How do we decide what the other states are?
- How do we decide what the transitions between states are?

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*



Next(state, sym)

- Next function determines what state to goto based on current state and symbol being recognized.
- For Non-terminal, this is used to determine the GOTO table.
- For terminal, this is used to determine the shift action.

Constructing states

```
initial_state = closure({start production})
state_set.add(initial_state)
state_queue.push(initial_state)

while(!state_queue.empty())
   s = state_queue.pop()
   foreach item A → a•Xb in s
      n = closure(next(s, X))
      if(!state_set.contains(n))
         state_set.add(n)
         state_queue.push(n)
```

A state is a set of LR(0) items

get "next" state

Closure*

closure({$S \rightarrow \bullet E\$$}) =

$S \rightarrow \bullet E\$$

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow$ *identifier*

*: for LR(0) items

Closure*

closure({$S \rightarrow \bullet E\$$}) =

$S \rightarrow \bullet E\$$

$E \rightarrow \bullet T + E$

$E \rightarrow \bullet T$

$T \rightarrow \bullet \mathit{identifier}$

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow \mathit{identifier}$

*: for LR(0) items

Next

```
next(state, X)
    ret = empty
    foreach item A → a•Xb in state
        ret.add(A → aX•b)
    return ret
```

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow$ *identifier*

initial:

$S \rightarrow \bullet E\$$

$E \rightarrow \bullet T + E$

$E \rightarrow \bullet T$

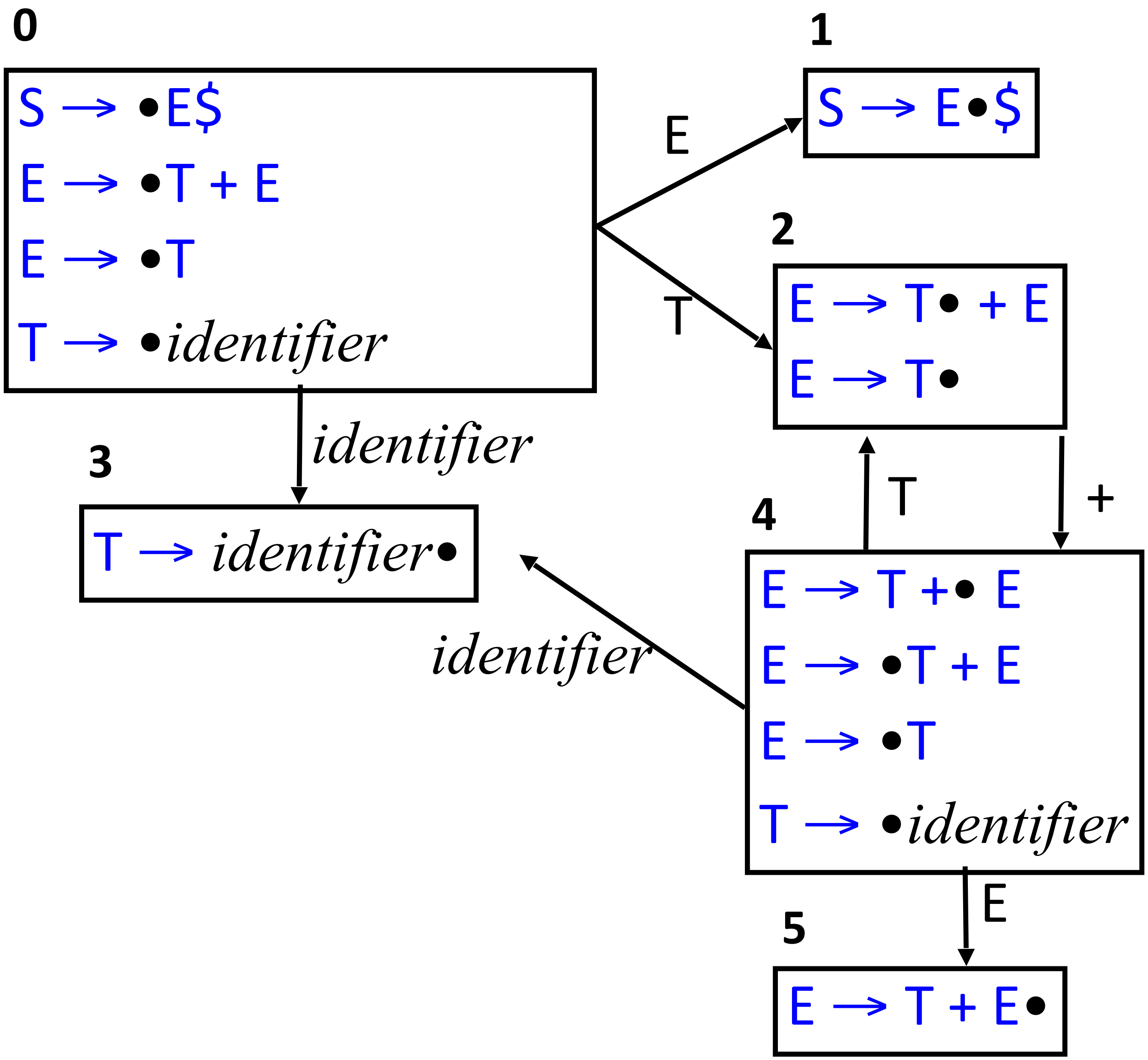
$T \rightarrow \bullet$*identifier*

next(initial, E)

next(initial, T)

next(initial, *identifier*)

Example



0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*