Bottom-Up Parsing

LR Parsing

Also called “Shift-Reduce Parsing”

Find a rightmost derivation
Finds it in reverse order

LR Grammars

Can be parsed with an LR Parser

LR Languages

Can be described with LR Grammar
Can be parsed with an LR Parser

Syntax Analysis - Part 2

Regular Languages
$\subset$
LL Languages
$\subset$
LR Languages
$\subset$
Unambiguous Languages
$\subset$
All Context-Free Languages
$\subset$
All Languages

Proper Subset

LR Parsing Techniques:

LR Parsing

Most General Approach

SLR

Simpler algorithm, but not as general

LALR

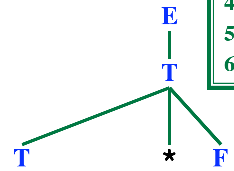
More complex, but saves space

© Harry H. Porter, 2005

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

A Rightmost Derivation

Rules Used:	*Right-Sentential Forms:*
	E
E → T	T
T → T * F	T * F

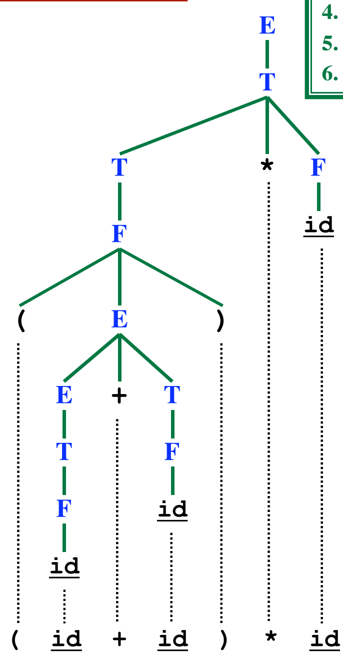


Syntax Analysis - Part 2

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

A Rightmost Derivation

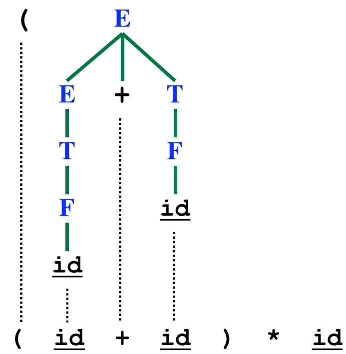
Rules Used:	*Right-Sentential Forms:*
	E
E → T	T
T → T * F	T * F
F → id	T * id
T → F	F * id
F → (E)	(E) * id
E → E + T	(E + T) * id
T → F	(E + F) * id
F → id	(E + id) * id
E → T	(T + id) * id
T → F	(F + id) * id
F → id	(id + id) * id



1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

A Rightmost Derivation in Reverse

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id

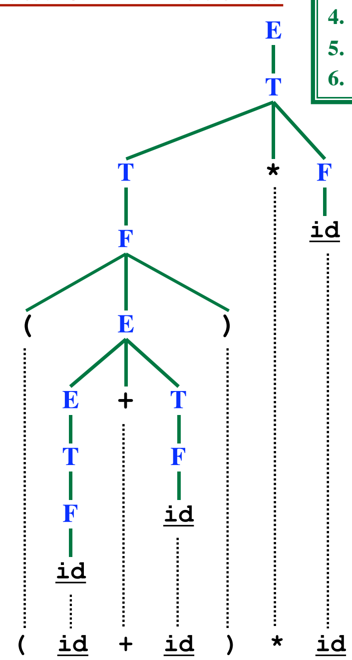


Syntax Analysis - Part 2

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

A Rightmost Derivation in Reverse

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id
F → (E)	F * id
T → F	T * id
F → id	T * F
T → T * F	T
E → T	E



1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

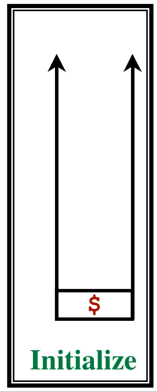
An LR Parse

Rules Used:

Right-Sentential Forms:

(id + id) * id

(



1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

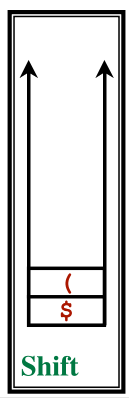
Rules Used:

Right-Sentential Forms:

(id + id) * id

(

(

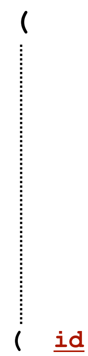


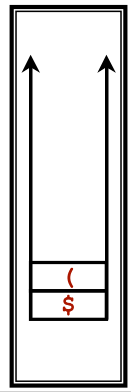
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used: *Right-Sentential Forms:*

(id + id) * id



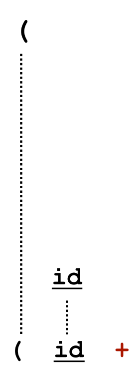


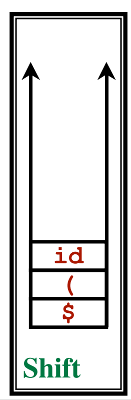
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used: *Right-Sentential Forms:*

(id + id) * id





1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

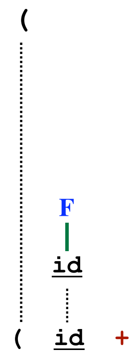
Rules Used:

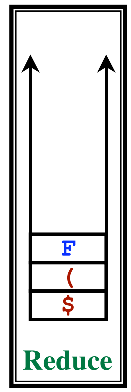
F → id

Right-Sentential Forms:

(id + id) * id

(F + id) * id





1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used:

F → id

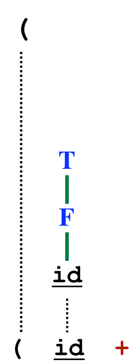
T → F

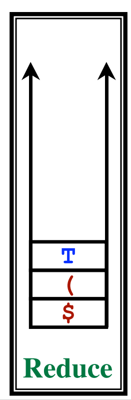
Right-Sentential Forms:

(id + id) * id

(F + id) * id

(T + id) * id

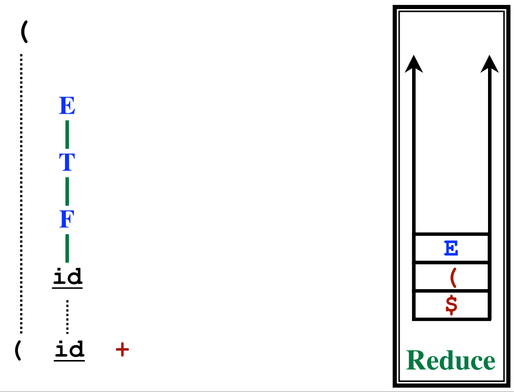




1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id

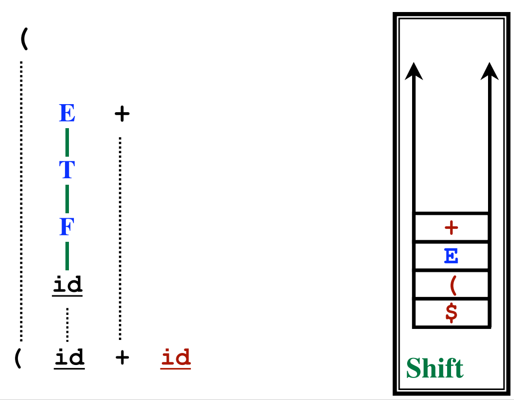


Syntax Analysis - Part 2

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

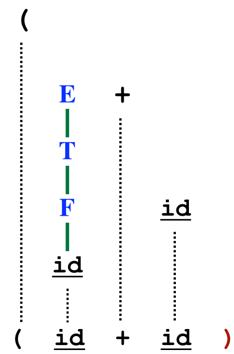
Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id

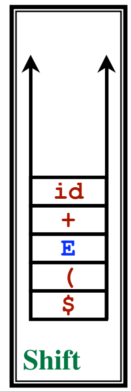


1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id

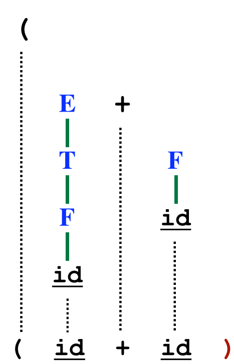


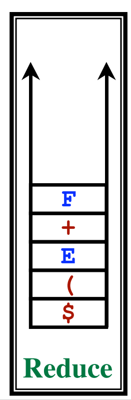


1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id

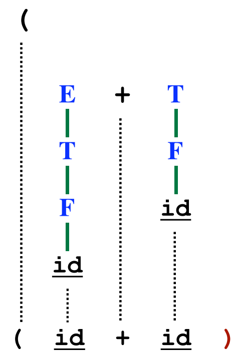


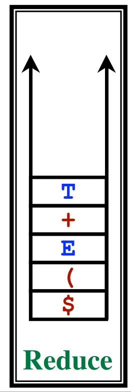


1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id

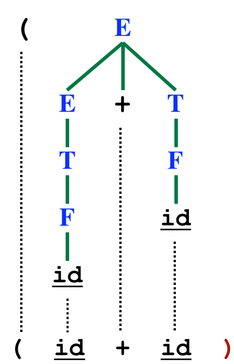


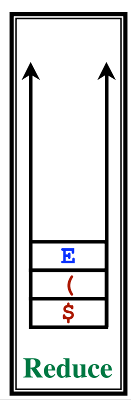


1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id

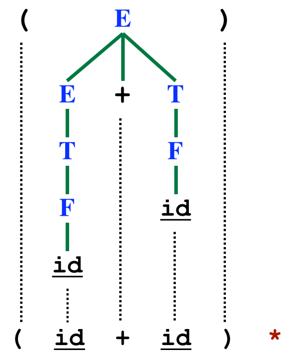


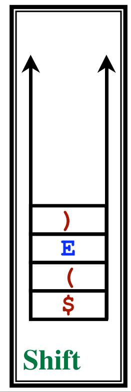


An LR Parse

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id



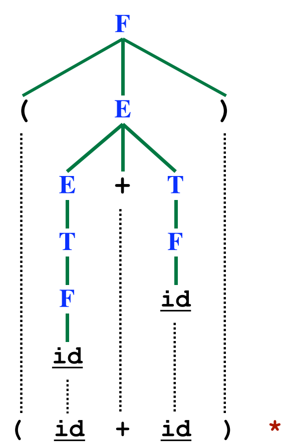


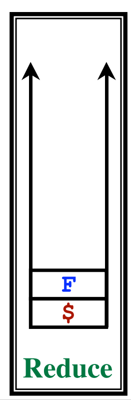
Syntax Analysis - Part 2

An LR Parse

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id
F → (E)	F * id

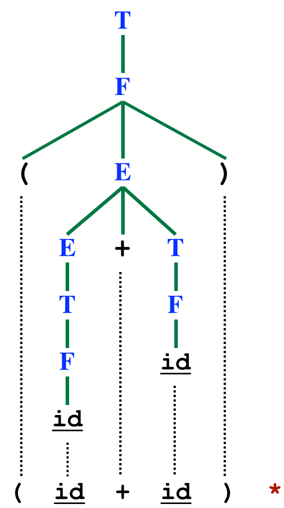


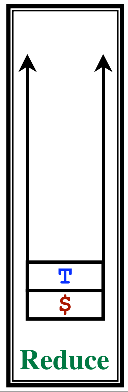


1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id
F → (E)	F * id
T → F	T * id

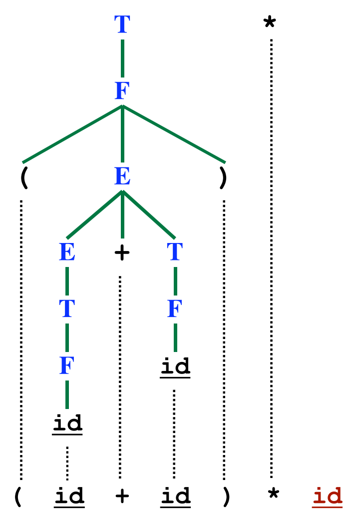


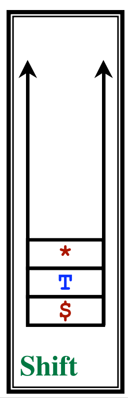


1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id
F → (E)	F * id
T → F	T * id

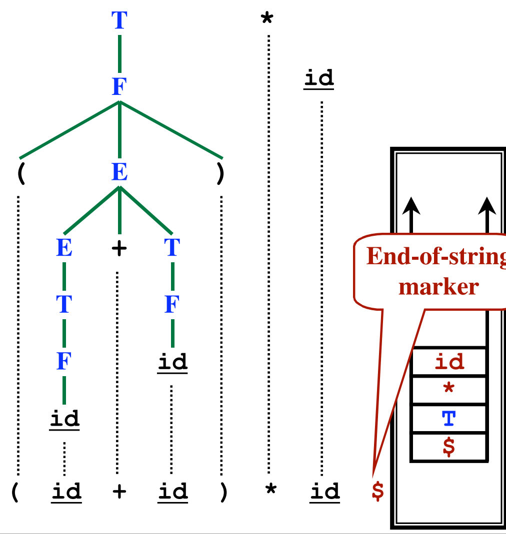




1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

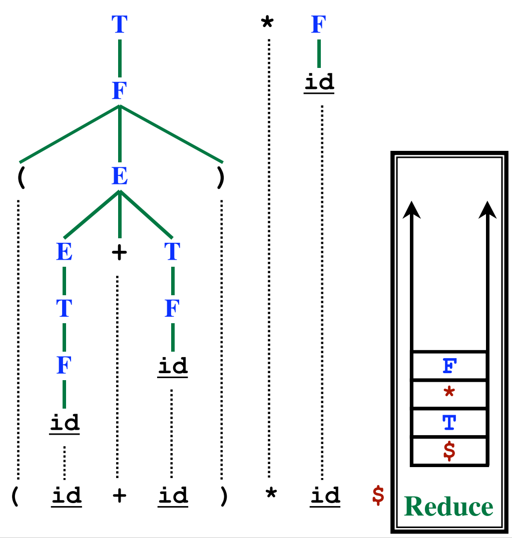
Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id
F → (E)	F * id
T → F	T * id



1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

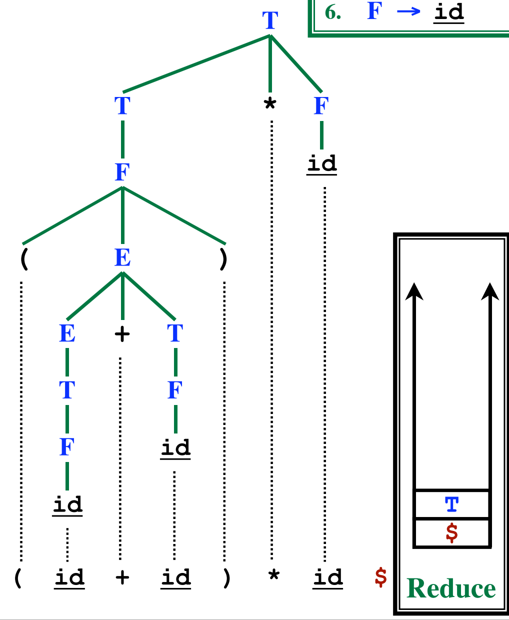
Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id
F → (E)	F * id
T → F	T * id
F → id	T * F



1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

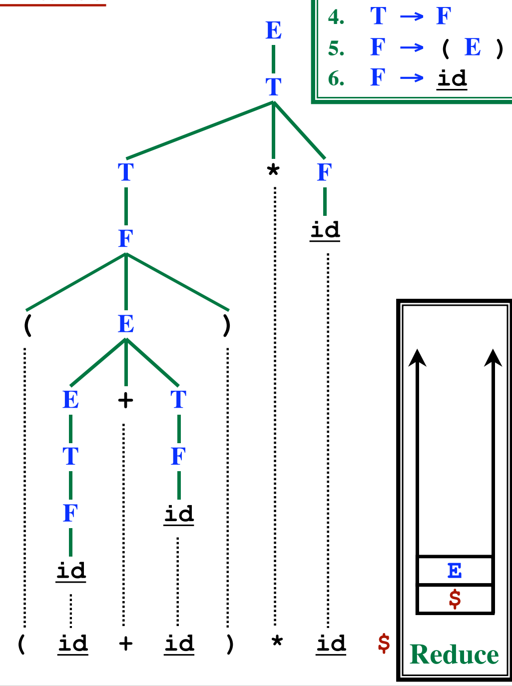
Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id
F → (E)	F * id
T → F	T * id
F → id	T * F
T → T * F	T



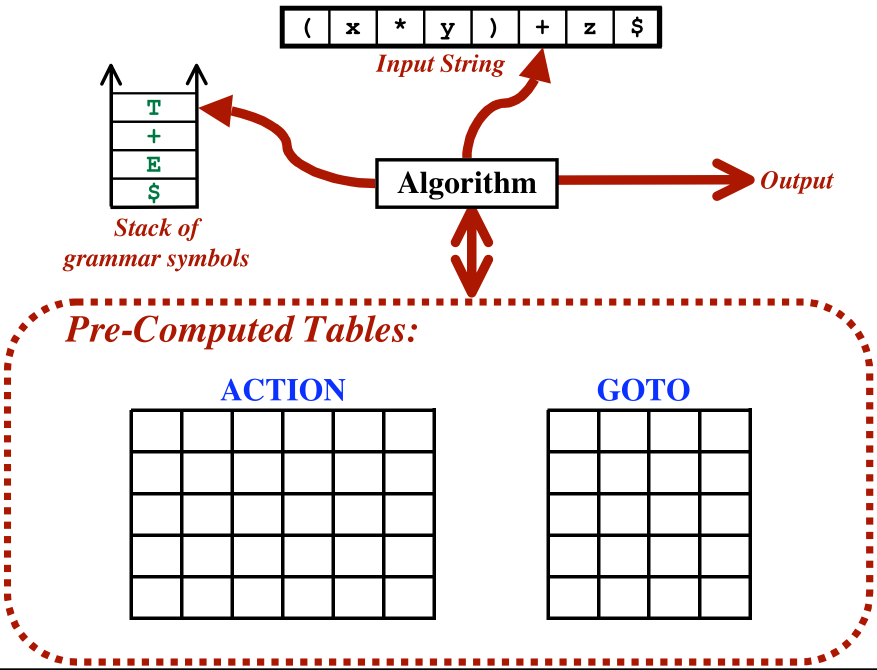
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

An LR Parse

Rules Used:	*Right-Sentential Forms:*
	(id + id) * id
F → id	(F + id) * id
T → F	(T + id) * id
E → T	(E + id) * id
F → id	(E + F) * id
T → F	(E + T) * id
E → E + T	(E) * id
F → (E)	F * id
T → F	T * id
F → id	T * F
T → T * F	T
E → T	E



The LR Parsing Algorithm



Handles

Definition: "**Handle**"

Given a right-sentential form γ,
A handle is
- A position in γ
- A rule $A \rightarrow \beta$

Such that if you do a reduction by $A \rightarrow \beta$ at that point,
it is a valid step in a rightmost derivation.

In other words...
let
$$\gamma = \alpha\beta w$$
then
$$S \Rightarrow_{RM}{}^{*} \alpha A w \Rightarrow_{RM} \alpha\beta w$$

Handles: Example

1. S → `f` A B `e`
2. A → A `g` `c`
3. A → `g`
4. B → `d`

A rightmost derivation, in reverse:

Input String:
  `f g g c d e`
Reduce by A → `g`
  `f` A `g c d e`
Reduce by A → A `g c`
  `f` A `d e`
Reduce by B → `d`
  `f` A B `e`
Reduce by S → `f` A B `e`
  S

Success! The handles are in red!

Handles: Example

1. S → `f` A B `e`
2. A → A `g` `c`
3. A → `g`
4. B → `d`

A rightmost derivation, in reverse:

Same String

This is NOT a handle!

Input String:
  `f g g c d e`
Reduce by A → `g`
  `f` A `g c d e`
Reduce by A → `g`
  `f` A A `c d e`

Now we are stuck!
No way to continue reducing!

Must be careful in deciding when to reduce, or else we may get stuck!

Shift-Reduce Parsing

Goal:
Find handles and perform reductions.

Is there a handle on the top of the stack?
Yes: Do a reduction
No: Shift another input symbol onto the stack

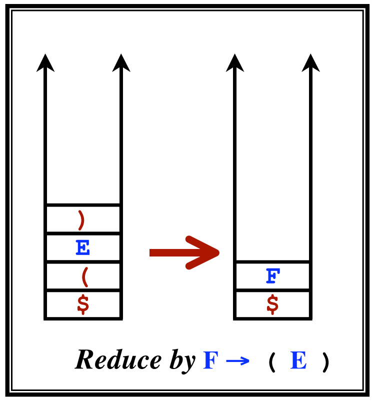
Possible Actions:

Shift
Push current input symbol onto stack
Advance input to next symbol

Reduce
A handle is on the top of the stack
Pop the handle
Push the lefthand side of the rule

Accept
Report success and terminate

Error
Report error and terminate



1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Notation for a Shift-Reduce Execution

Time

STACK	INPUT	ACTION
$	(id+id)*id$	
$(	id+id)*id$	Shift
$(id	+id)*id$	Shift
$(F	+id)*id$	Reduce by F → id
$(T	+id)*id$	Reduce by T → F
$(E	+id)*id$	Reduce by E → T
$(E+	)*id$	Shift
$(E+id	)*id$	Shift
$(E+F	)*id$	Reduce by F → id
$(E+T	)*id$	Reduce by T → F
$(E	)*id$	Reduce by E → E + T
$(E)	*id$	Shift
$F	*id$	Reduce by F → (E)
$T	*id$	Reduce by T → F
$T*	id$	Shift
$T*id	$	Shift
$T*F	$	Reduce by F → id
$T	$	Reduce by T → T * F
$E	$	Reduce by E → T
$E	$	Accept

Shift-Reduce Actions

(Initial Setup)

STACK	INPUT	ACTION
`$`	`...input...$`	

Shift

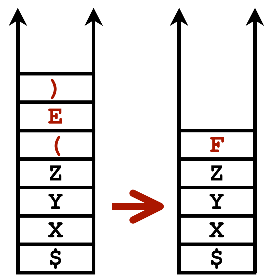
Push current input symbol onto stack
Advance input to next symbol

STACK	INPUT	ACTION
`$XYZ`	`abc...$`	
`$XYZa`	`bc...$`	Shift

Reduce

A handle is on the top of the stack
Pop the handle
Push the lefthand side of the rule

STACK	INPUT	ACTION
`$XYZ(E)`	`...$`	
`$XYZF`	`...$`	Reduce by $F \rightarrow (E)$



Accept

Report success and terminate

STACK	INPUT	ACTION
`$S`	`...$`	Accept

Error

Report error and terminate

How do we know what to do at each step?

Given:

- The stack and the current input symbol
- The tables (ACTION and GOTO)

Should be deterministic!

Reduce-Reduce Conflict

Can reduce by 2 different rules... Which to use???

Shift-Reduce Conflict

Can either shift or reduce... Which to do???

LR Parsing Approach:

Build Tables
(Algorithm to follow)
Each table entry will have one action (SHIFT, REDUCE, ACCEPT, or ERROR)
Failure when building the tables?
Some entry has multiple actions!
∴ The grammar is not LR!
LR Grammars are unambiguous
Only one rightmost derivation
∴ There is only one handle at each step

LR Parsing

One Parsing Algorithm
Several Ways to Build the Tables

SLR (or "Simple LR")
- May fail to build a table for some LR grammars
- SLR Grammars ⊂ LR Grammars
- Easiest to understand

LR (or " Canonical LR")
- The general algorithm
- Will work for any LR Grammar

LALR (or " Lookahead LR")
- Will build smaller tables
- May fail for some LR Grammars
- SLR Grammars ⊂ LALR Grammars ⊂ LR Grammars
- Most difficult to understand
- Used in parser generators

LR(1) Parsing

The knowledge of what we've parsed so far is in the stack.
Some knowledge is buried in the stack.
We need a "summary" of what we've learned so far.

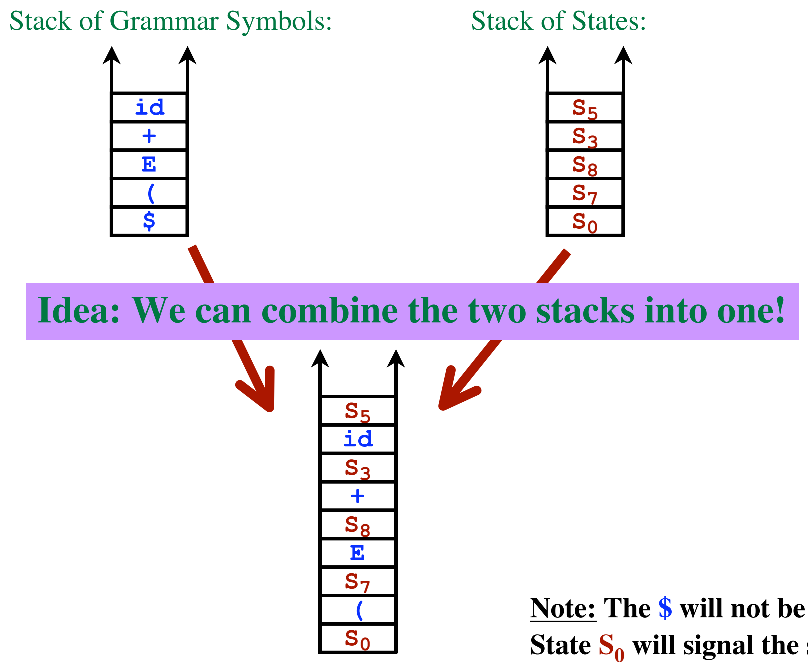
LR Parsing uses a second stack for this information.

Stack 1: Stack of grammar symbols (terminals and nonterminals)
Stack 2: Stack of "states".

States = { S_0, S_1, S_2, S_3, ... , S_N }
Implementation: Just use integers (0, 1, 2, 3, ...)
⇒ Just use a stack of integers

When deciding on an action...
- Consult the Parsing Tables (ACTION, and GOTO)
- Consult the top of the stack of states

The Stack of States



Note: The $ will not be needed.
State S_0 will signal the stack bottom.

Key to Notation

- **S4** = "Shift and push state 4"
- **R5** = "Reduce by rule 5"
- **Acc** = Accept
- **(blank)** = Syntax Error

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

"ACTION" Table (*States*)

	id	+	*	(	)	$
0	S5			S4		
1		S6				Acc
2		R2	S7		R2	R2
3		R4	R4		R4	R4
4	S5			S4		
5		R6	R6		R6	R6
6	S5			S4		
7	S5			S4		
8		S6			S11	
9		R1	S7		R1	R1
10		R3	R3		R3	R3
11		R5	R5		R5	R5

"GOTO" Table

	E	T	F
0	1	2	
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

Example LR Parse: (id+id)*id

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

STACK	INPUT	ACTION
0	(id+id)*id$	

What next?

Example LR Parse: (id+id)*id

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

STACK	INPUT	ACTION
0	(id+id)*id$	
0(4	id+id)*id$	Shift 4

What next?

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Example LR Parse: (id+id)*id

STACK	INPUT	ACTION
0	(id+id)*id$	
0(4	id+id)*id$	Shift 4
0(4id5	+id)*id$	Shift 5
0(4F3	+id)*id$	Reduce by F → id
0(4T2	+id)*id$	Reduce by T → F
0(4E8	+id)*id$	Reduce by E → T
0(4E8+6	)*id$	Shift 6
0(4E8+6id5	)*id$	Shift 5
0(4E8+6F3	)*id$	Reduce by F → id
0(4E8+6T9	)*id$	Reduce by T → F
0(4E8	)*id$	Reduce by E → E + T
0(4E4)11	*id$	Shift
0F3	*id$	Reduce by F → (E)
0T2	*id$	Reduce by T → F
0T2*7	id$	Shift 7
0T2*7id5	$	Shift 5
0T2*7F10	$	Reduce by F → id
0T2	$	Reduce by T → T * F
0E1	$	Reduce by E → T
		Accept

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Output:

6. F → id
4. T → F
2. E → T
6. F → id
4. T → F
1. E → E + T
5. F → (E)
4. T → F
6. F → id
3. T → T * F
2. E → T

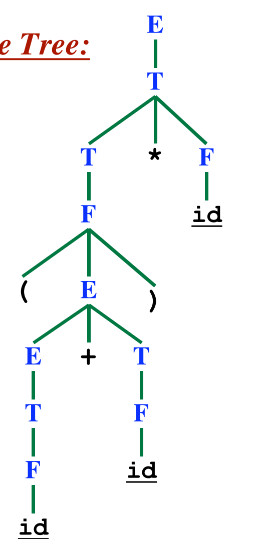
Reversed:

2. E → T
3. T → T * F
6. F → id
4. T → F
5. F → (E)
1. E → E + T
4. T → F
6. F → id
2. E → T
4. T → F
6. F → id

Rightmost Derivation:

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

Parse Tree:



The LR Parsing Algorithm

Input:
- String to parse, w
- Precomputed ACTION and GOTO tables for grammar G

Output:
- Success, if w ∈ L(G) plus a trace of rules used
- Failure, if syntax error

```
push state 0 onto the stack
loop
  s = state on top of stack
  c = next input symbol
  if ACTION[s,c] = "Shift N" then
    push c onto the stack
    advance input
    push state N onto stack
  elseif ACTION[s,c] = "Reduce R"
   then
    let rule R be A → β
    pop 2*|β| items off the stack
    s' = state now on stack top
    push A onto stack
    push GOTO[s',A] onto stack
    print "A → β"
  elseif ACTION[s,c] = "Accept"
   then
    return success
  else
    print "Syntax error"
    return
  endIf
endLoop
```


LR Grammars

"LR(1)"

Scan the input left-to-right

Find a rightmost derivation

Using one symbol of look-ahead

What to do next?
- Look at the stack
- Look at the next input symbol
  - LR(1) Typical
  - LR(k) Look at the next k input symbols

"LR" means LR(k) for some k.

A language is LR if...
- it can be described by an LR Grammar
- it can be parsed by an LR Parser

LR Grammars are never ambiguous

Not Ambiguous?
   Some unambiguous grammars are still not LR!

Most Programming Languages...
   use LR grammars (or can be transformed into equivalent LR grammars)

An Unambiguous Grammar which is *NOT* LR

S → A | B
A → (A)
  → ( )
B → (B))
  → ( ))

Example Strings:
((( )))
((( ))))))

The problem:

Imagine seeing this input:

((((((( )) . . .

The LR Parser must reduce by either

A → ( )

or

B → ( ))

But you cannot decide which rule to use

It may require an arbitrarily long look-ahead

In general, you may need arbitrarily long input before deciding!

Relationship of Language Classes

Regular Languages
$\subset$
LL Languages
$\subset$
LR Languages
$\subset$
Unambiguous Languages
$\subset$
All Context-Free Languages
$\subset$
All Languages