Bottom up Parsing

- Bottom up parsing trys to transform the input string into the start symbol.
- Moves through a sequence of sentential forms (sequence of Nonterminal or terminals). Trys to identify some _substring_ of the sentential form that is the rhs of some production.
- E -> E + E | E * E | x
  - _x_ + x * x
  - E + _x_ * x
  - _E + E_ * x
  - E * _x_
  - _E * E_
  - E

> The substring (shown in color and italics) for each step) may contain both terminal and non-terminal symbols. This string is the rhs of some production, and is often called a handle.

Bottom Up Parsing

Implemented by Shift-Reduce parsing

- data structures: input-string and stack.
- look at symbols on top of stack, and the input-string and decide:
  - shift (move first input to stack)
  - reduce (replace top n symbols on stack by a non-terminal)
  - accept (declare victory)
  - error (be gracious in defeat)

Example Bottom up Parse

Consider the grammar: (note: left recursion is NOT a problem, but the grammar is still layered to prevent ambiguity)

```
1. E ::= E + T
2. E ::= T
3. T ::= T * F
4. T ::= F
5. F ::= ( E )
6. F ::= id
```

stack	*Input*	*Action*
	x + y	shift
x	+ y	reduce 6
F	+ y	reduce 4
T	+ y	reduce 2
E	+ y	shift
E +	y	shift
E + y		reduce 6
E + F		reduce 4
E + T		reduce 1
E		accept

The concatenation of the stack and the input is a sentential form. The input is all terminal symbols, the stack is a combination of terminal and non-terminal symbols

LR(k)

- Grammars which can decide whether to shift or reduce by looking at only k symbols of the input are called LR(k).
  - Note the symbols on the stack don't count when calculating k

- L is for a Left-to-Right scan of the input

- R is for the Reverse of a Rightmost derivation

Problems (ambiguous grammars)

1) shift reduce conflicts:

stack	*Input*	*Action*
x + y	+ z	?

stack	*Input*	*Action*
if x t if y t s2	e s3	?

2) reduce reduce conflicts:

suppose both procedure call and array reference have similar syntax:

- x(2) := 6
- f(x)

stack	*Input*	*Action*
id (id	) id	?

Should id reduce to a parameter or an expression. Depends on whether the bottom most id is an array or a procedure.

Using ambiguity to your advantage

- Shift-Reduce and Reduce-Reduce errors are caused by ambiguous grammars.
- We can use resolution mechanisms to our advantage. Use an ambiguous grammar (smaller more concise, more natural parse trees) but resolve ambiguity using rules.
- Operator Precedence
  - Every operator is given a precedence
  - Precedence of the operator closest to the top of the stack and the precedence of operator next on the input decide shift or reduce.
  - Sometimes the precedence is the same. Need more information: Associativity information.

Example Precedence Parser

	+	*	(	)	id	$
+	:>	<:	<:	:>	<:	:>
*	:>	:>	<:	<:	<:	:>
(	<:	<:	<:	=	<:	
)	:>	:>		:>		:>
id	:>	:>		:>		:>
$	<:	<:	<:		accept <:	

input : x * x + y

stack	*Input*	*Action*
$ E * E	+ y $	reduce!

topmost terminal

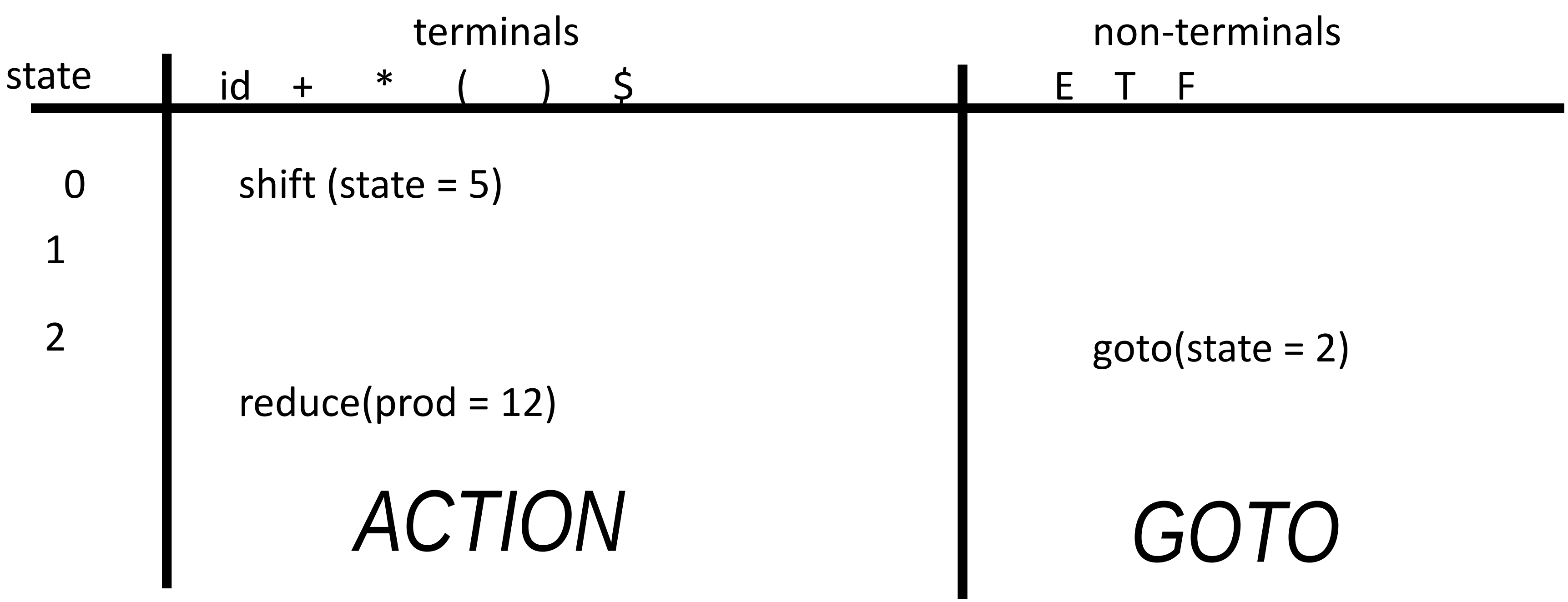
next input

Precedence parsers

- Precedence parsers have limitations
- No production can have two consecutive non-terminals
- Parse only a small subset of the Context Free Grammars
- Need a more robust version of shift- reduce parsing.
- LR - parsers
  - State based - finite state automatons (w / stack)
  - Accept the widest range of grammars
  - Easily constructed (by a machine)
  - Can be modified to accept ambiguous grammars by using precedence and associativity information.

LR Parsers

- Table Driven Parsers
- Table is indexed by *state* and *symbols* (both term and non-term)
- Table has two components.
  - ACTION part
  - GOTO part



LR Table encodes FSA

E -> E + T | T

T -> T * F | F

F -> (E) | id

transition on terminal is a shift in action table, on nonterminal is a goto entry

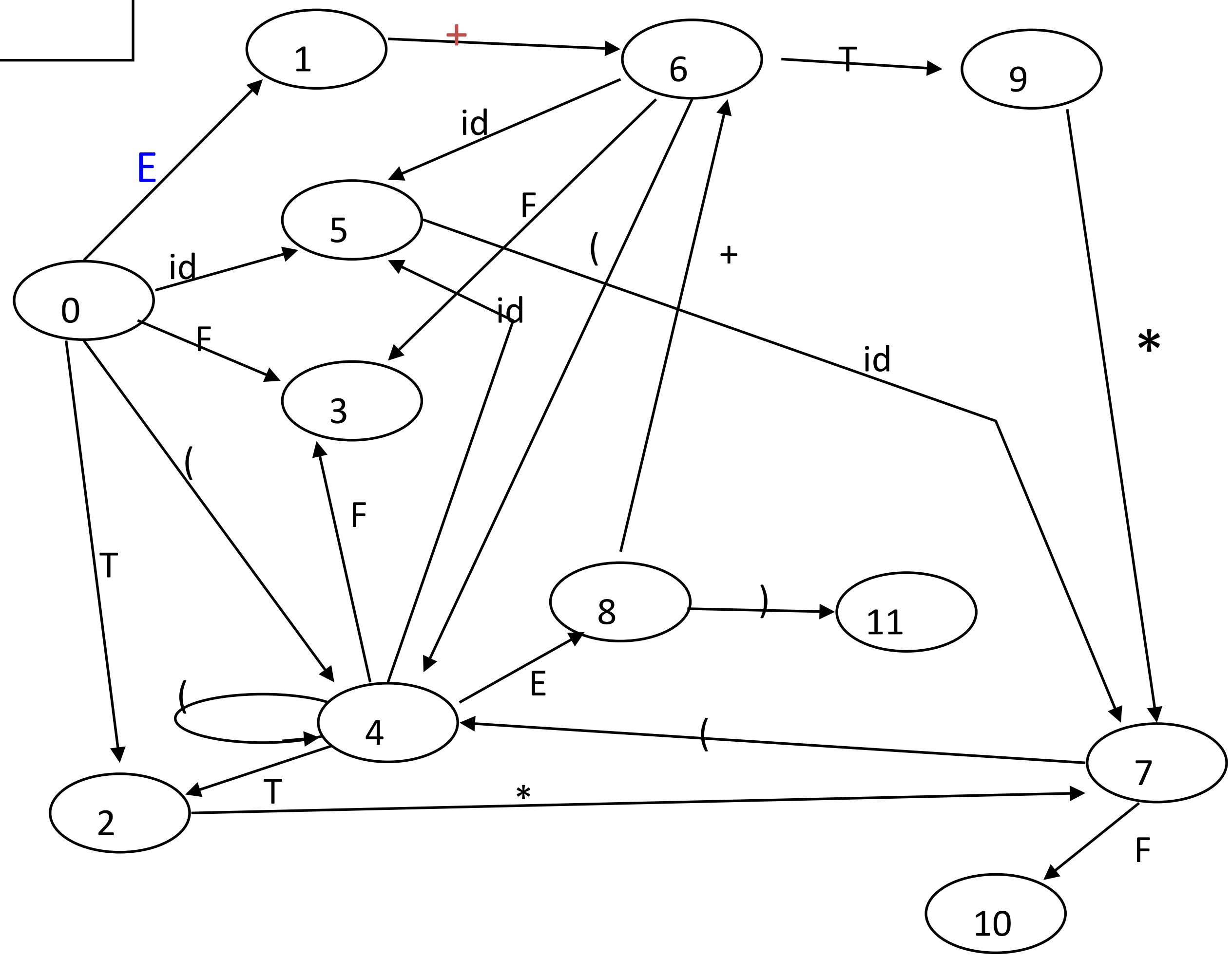


Table vs FSA

- The Table encodes the FSA

- The action part encodes
  - Transitions on terminal symbols (shift)
  - Finding the end of a production (reduce)

- The goto part encodes
  - Tracing backwards the symbols on the RHS
  - Transition on non-terminal, the LHS

- Tables can be quite compact

LR Table

	terminals						non-terminals		
state	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Reduce Action

- If the top of the stack is the rhs for some production n
- And the current action is “reduce n”
- We pop the rhs, then look at the state on the top of the stack, and index the goto-table with this state and the LHS non-terminal.
- Then push the lhs onto the stack in the new s found in the goto-table.

(?,0)(id,5)   * id + id $

Where: Action(5,*) = reduce 6

Production 6 is: F ::= id

And: GOTO(0,F) = 3

(?,0)(F,3)   * id + id $

Example Parse

Stack	*Input*
(?,0)	id * id + id $
(?,0)(id,5)	* id + id $
(?,0)(F,3)	* id + id $
(?,0)(T,2)	* id + id $
(?,0)(T,2)(*,7)	id + id $
(?,0)(T,2)(*,7)(id,5)	+ id $
(?,0)(T,2)(*,7)(F,10)	+ id $
(?,0)(T,2)	+ id $
(?,0)(E,1)	+ id $
(?,0)(E,1)(+,6)	id $
(?,0)(E,1)(+,6)(id,5)	$
(?,0)(E,1)(+,6)(F,3)	$
(?,0)(E,1)(+,6)(T,9)	$
(?,0)(E,1)	$

1) E -> E + T
2) E -> T
3) T -> T * F
4) T -> F
5) F -> (E)
6) F -> id

Review

- Bottom up parsing transforms the input into the start symbol.
- Bottom up parsing looks for the rhs of some production in the partially transformed intermediate result
- Bottom up parsing is OK with left recursive grammars
- Ambiguity can be used to your advantage in bottom up partsing.
- The LR(k) languages = LR(1) languages = CFL