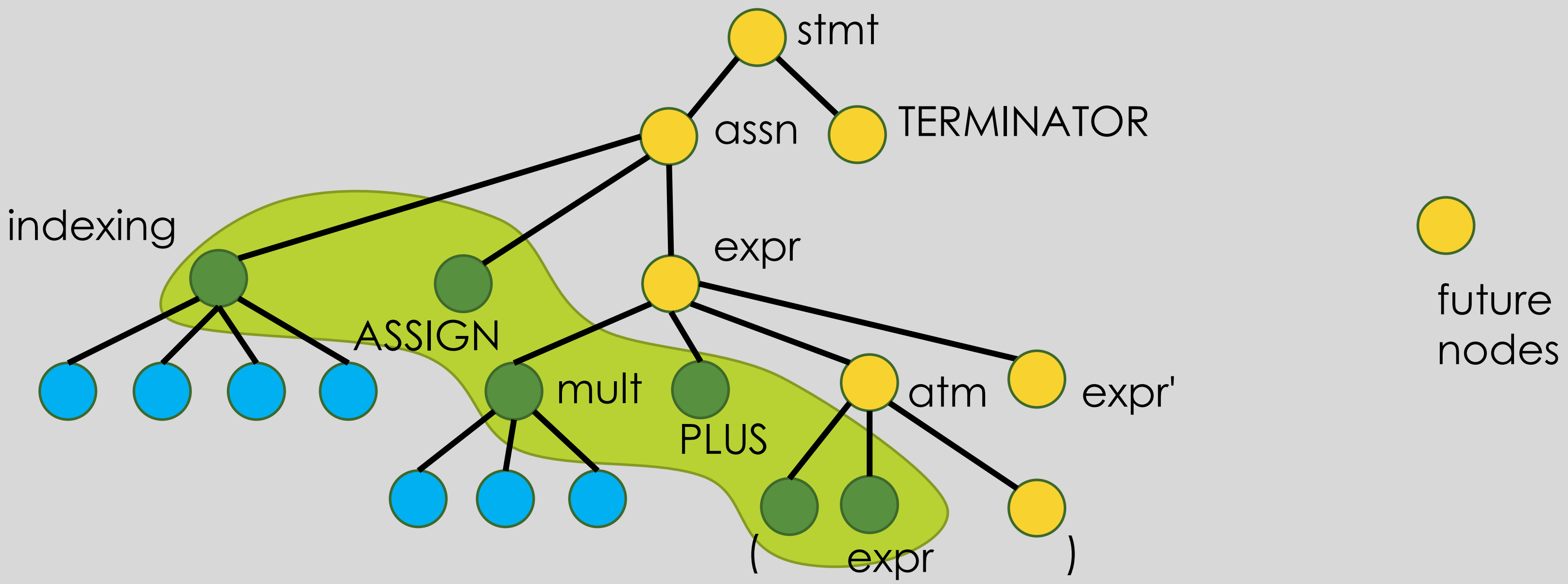
BOTTOM-UP PARSING

CMPT 379 Lecture 16

Lecture Overview

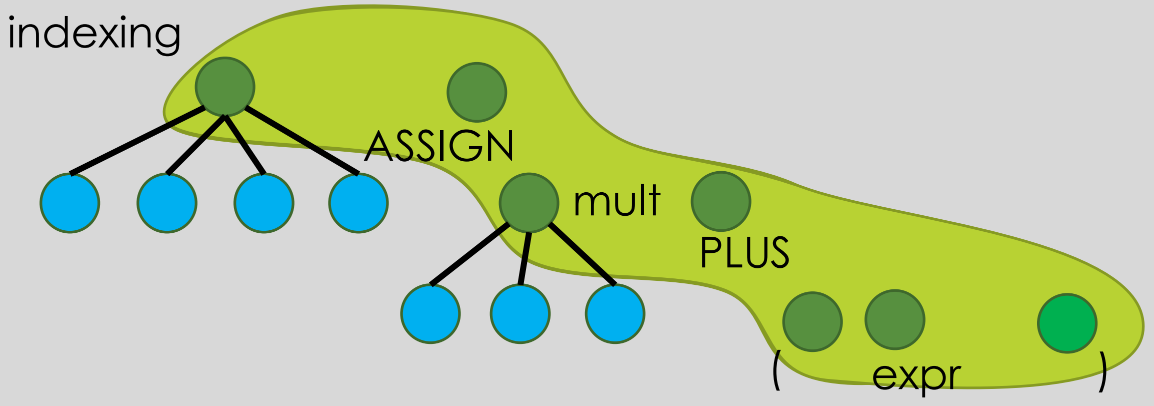
- Shift and Reduce
- LR Parsing
- LR(0) Items and Automaton
- Parsing Conflicts

Bottom-up Parsing



Stack contains roots of discovered subtrees

Shifting in Bottom-Up

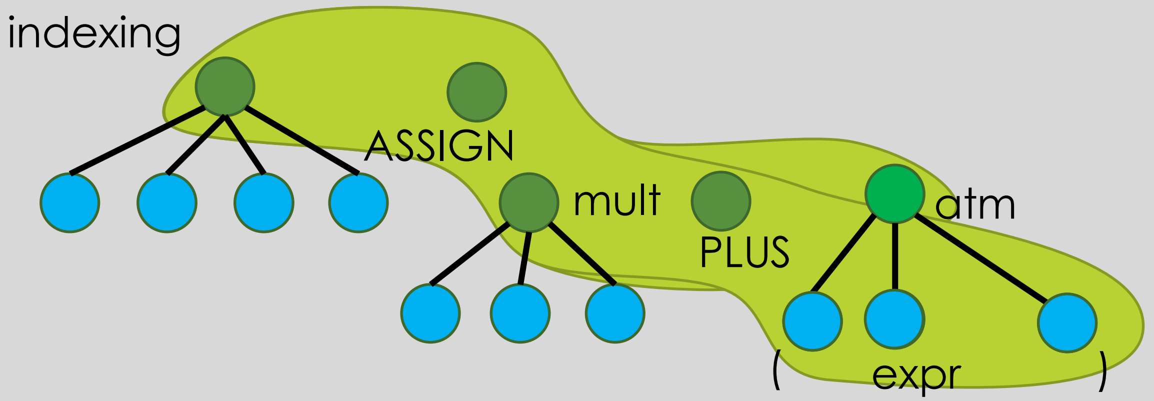


Input: **)** ;

SHIFT

Stack contains roots of discovered subtrees

Reducing in Bottom-up

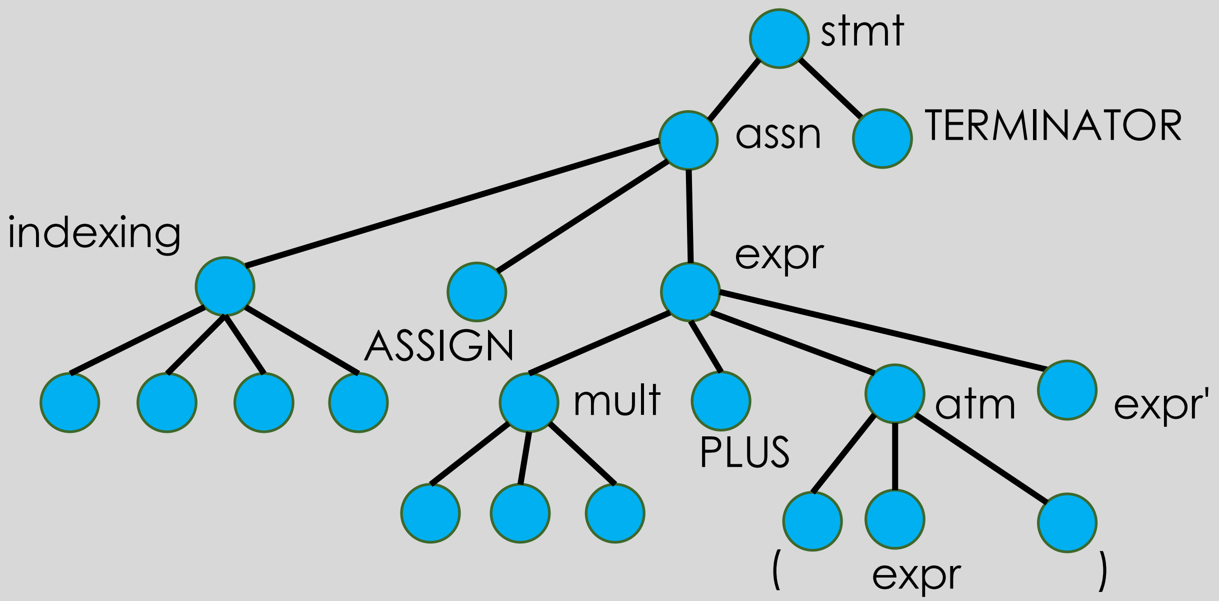


Input: ;

REDUCE
atm → (expr)

Stack contains roots of discovered subtrees

Bottom-up Parsing



Bottom-up Parsing

Bottom-up parsing starts with the input and works towards a parse tree that is rooted at the start symbol. The derivation

$$S \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \gamma_3 \rightarrow \gamma_4 \cdots \gamma_{n-1} \rightarrow \gamma_n$$

does the opposite. It starts at the start symbol and derives the input. Thus bottom-up parsing discovers the derivation backwards.

A handle is a pair of a production and a stack location, such as $\langle A \rightarrow \beta, k \rangle$. If bottom-up parsing finds a handle, then it will reduce by removing β from the stack at k and replacing it with A. Finding handles efficiently is the key to bottom-up parsing.

LR( ) Parsing Algorithm

```
create stack T
T.push($)                 // sentinel
T.push(start state)
token = NextToken()

loop forever:
    state = T.top()
    switch Action[state, token] {

    case "reduce A → β":
        pop 2|β| elements from T
        state = T.top()
        T.push(A)         // with children
                          // that were popped
        T.push( Goto[state, A] )
        break

    case "shift s_i":
        T.push(token)
        T.push(s_i)
        token = NextToken()
        break

    case "accept":
        break out of loop

    default:
        Fail()

    }
```

Parentheses Grammar

1. Goal $\rightarrow$ List
2. List $\rightarrow$ List Pair
3. List $\rightarrow$ Pair
4. Pair $\rightarrow$ **(** Pair **)**
5. Pair $\rightarrow$ **( )**

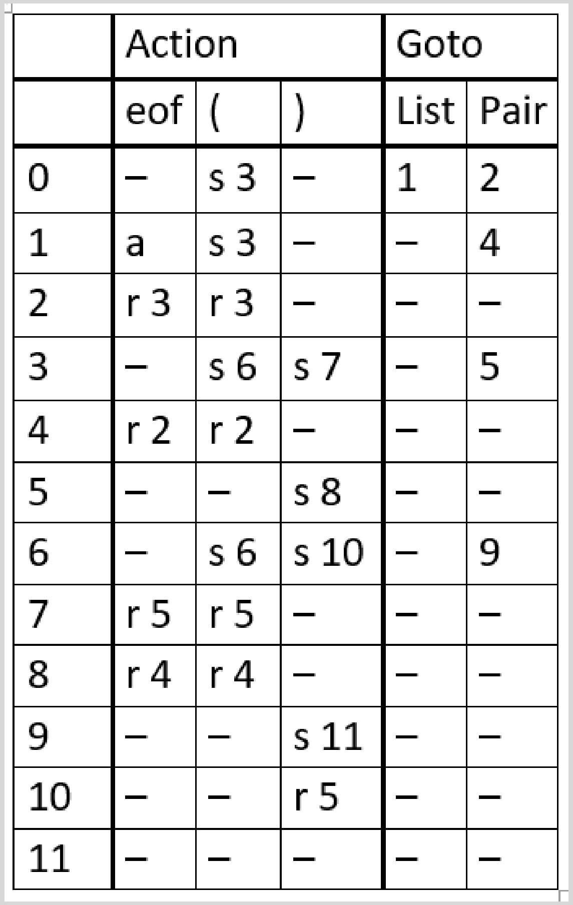
not quite all strings of balanced parentheses.

1. Goal $\rightarrow$ List
2. List $\rightarrow$ List Pair
3. List $\rightarrow$ Pair
4. Pair $\rightarrow$ **(** List **)**
5. Pair $\rightarrow$ **( )**

all nonempty strings of balanced parentheses.

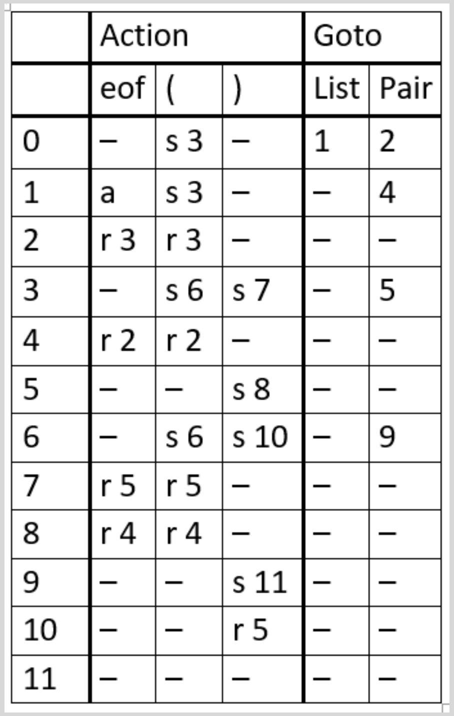
Parentheses Grammar

1. Goal → List
2. List → List Pair
3. List → Pair
4. Pair → **(** Pair **)**
5. Pair → **( )**



	Action			Goto	
	eof	(	)	List	Pair
0	–	s 3	–	1	2
1	a	s 3	–	–	4
2	r 3	r 3	–	–	–
3	–	s 6	s 7	–	5
4	r 2	r 2	–	–	–
5	–	–	s 8	–	–
6	–	s 6	s 10	–	9
7	r 5	r 5	–	–	–
8	r 4	r 4	–	–	–
9	–	–	s 11	–	–
10	–	–	r 5	–	–
11	–	–	–	–	–

Parentheses Grammar Example



	Action			Goto	
	eof	(	)	List	Pair
0	–	s 3	–	1	2
1	a	s 3	–	–	4
2	r 3	r 3	–	–	–
3	–	s 6	s 7	–	5
4	r 2	r 2	–	–	–
5	–	–	s 8	–	–
6	–	s 6	s 10	–	9
7	r 5	r 5	–	–	–
8	r 4	r 4	–	–	–
9	–	–	s 11	–	–
10	–	–	r 5	–	–
11	–	–	–	–	–

Input: (( )) ( )

stack	state	token	action
$ 0	0	(	s 3
$ 0 (3	3	(	s 6
$ 0 (3 (6	6	)	s 10
$ 0 (3 (6) 10	10	)	r 5
$ 0 (3 Pair 5	5	)	s 8
$ 0 (3 Pair 5) 8	8	(	r 4
$ 0 Pair 2	2	(	r 3
$ 0 List 1	1	(	s 3
$ 0 List 1 (3	3	)	s 7
$ 0 List 1 (3) 7	7	eof	r 5
$ 0 List 1 Pair 4	4	eof	r 2
$ 0 List 1	1	eof	accept

LR(0) Parsing and LR(0) Items

We will examine LR(0) parsing, which is LR parsing with no lookahead.

To construct Action and Goto tables, we will use LR(0) items; an LR(0) item is a production with a dot somewhere on the right-hand side. Here are the LR(0) items for the parenthesis grammar.

Goal → · List

Goal → List ·

List → · List Pair

List → List · Pair

List → List Pair ·

List → · Pair

List → Pair ·

Pair → · **(** Pair **)**

Pair → **(** · Pair **)**

Pair → **(** Pair · **)**

Pair → **(** Pair **)** ·

Pair → · **( )**

Pair → **(** · **)**

Pair → **( )** ·

LR(0) Items

Intuitively, an LR(0) item $A \rightarrow \beta \cdot \gamma$ indicates that we have just seen a string derivable from β on the input and we hope next to see a string derivable from γ.

We work with sets of items and have need of an operation we call closure. The closure of a set of items I can be computed by:

1. Initially, add every item in I to CLOSURE(I).
2. If $A \rightarrow \alpha \cdot B\beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is a production, add the item $B \rightarrow \cdot \gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

LR(0) Automaton

The LR(0) automaton has sets of items for states, and has transitions on every grammar symbol (terminal or nonterminal).

We construct the LR(0) automaton in much the way we did the subset construction – by starting with the start state and exploring the states we can reach from there.

The start state of the LR(0) automaton is the set of items that is the closure of every S-production with a dot on the far left, where S is the start state.

CLOSURE(Goal → · List) = {Goal → · List, List → · List Pair, List → · Pair, Pair → · **(** Pair **)**, Pair → · **( )** }

LR(0) Automaton

To compute the state after a transition on grammar symbol w, take the items before the transition, eliminate any that do not have a dot before a w in them, and move the dot in those items remaining to the right of that w. Then take the closure.

LR(0) Automaton

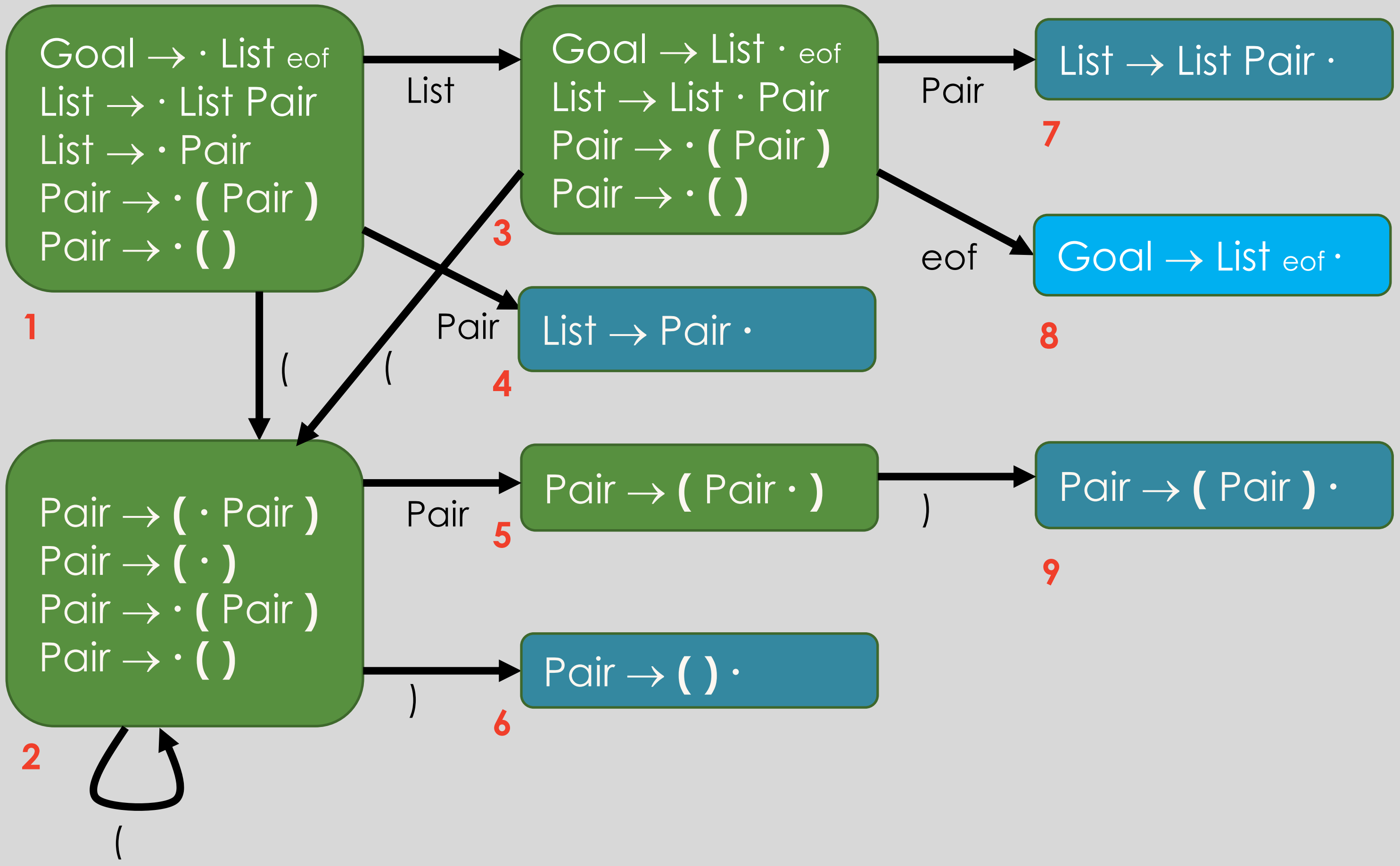
For instance, suppose we have the start state:

{Goal → · List, List → · List Pair, List → · Pair, Pair → · **(** Pair **)**, Pair → · **( )** }

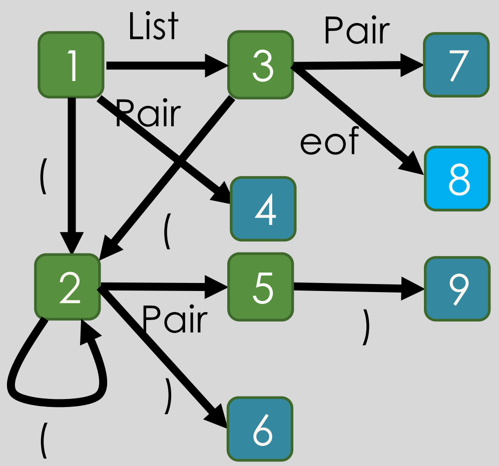
and we wish to compute the transition on the symbol List. We keep only those items that have a List after the dot (the items Goal → · List and List → · List Pair) and move the dot over the List in each of them to get Goal → List · and List → List · Pair. Then we'd take the closure to get the set

{ Goal → List · , List → List · Pair, Pair → · **(** Pair **)**, Pair → · **( )** }

LR(0) Automaton



LR(0) Parsing Tables



	Action			Goto	
	eof	(	)	List	Pair
1	–	s 2	–	3	4
2	–	s 2	s 6	–	5
3	s 8	s 2	–	–	7
4	r 3	r 3	r 3	–	–
5	–	–	s 9	–	–
6	r 5	r 5	r 5	–	–
7	r 2	r 2	r 2	–	–
8	a	a	a	–	–
9	r 4	r 4	r 4	–	–

Parsing Conflicts

If a state has an item with a dot on its far right-hand side, you can reduce by the production in that item.

$$\text{Pair} \rightarrow \textbf{(}\ \text{Pair}\ \textbf{)} \cdot$$

If there is a state with two or more different items with a dot on the far right-hand side, then there is a reduce-reduce conflict. The parser cannot handle this because it doesn't know which production to reduce by.

If there is a state with a transition on a terminal and an item with a dot on its far right-hand side, then there is a shift-reduce conflict. Again the parser cannot know which action to take.

Parsing Conflicts

Most automatic parser generators (like yacc, bison, or ANTLR) take a grammar as input.

Some will left-factor and remove left-recursion for you, and some assume that you have already done this.

Most will report any reduce-reduce or shift-reduce conflicts that they encounter. Many have methods for the grammar writer to specify how to resolve the conflict, for instance, taking one reduction over another or always prefering a shift to a reduce.