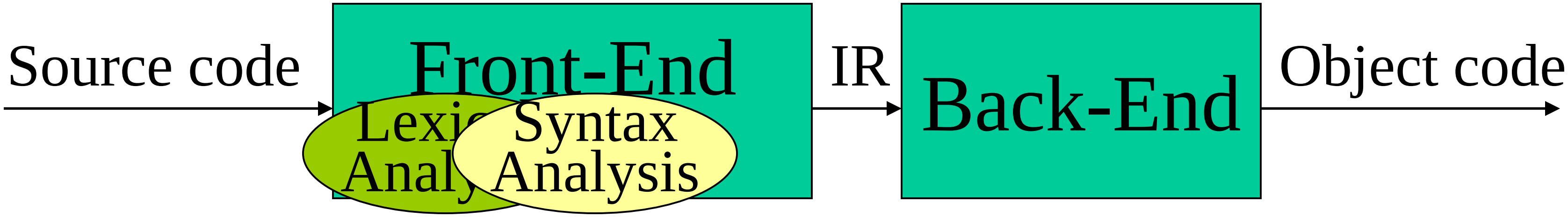
Lecture 9: Bottom-Up Parsing



(from last lecture) Top-Down Parsing:

- Start at the root of the tree and grow towards leaves.
- Pick a production and try to match the input.
- We may need to backtrack if a bad choice is made.
- Some grammars are backtrack-free (predictive parsing).

Today's lecture:

Bottom-Up parsing

Bottom-Up Parsing: What is it all about?

Goal: Given a grammar, G, construct a parse tree for a string (i.e., sentence) by starting at the leaves and working to the root (i.e., by working from the input sentence back toward the start symbol S).

Recall: the point of parsing is to construct a derivation:

$$S \Rightarrow \delta_0 \Rightarrow \delta_1 \Rightarrow \delta_2 \Rightarrow \ldots \Rightarrow \delta_{n-1} \Rightarrow sentence$$

To derive δ_{i-1} from δ_i, we match some *rhs b* in δ_i, then replace *b* with its corresponding *lhs*, *A*. This is called a **reduction** (it assumes $A \rightarrow b$).

The **parse tree** is the result of the **tokens** and the **reductions**.

Example: Consider the grammar below and the input string **`abbcde`**.

```
1.  Goal→aABe
2.      A→Abc
3.        |b
4.      B→d
```

Sentential Form	Production	Position
abbcde	3	2
a A bcde	2	4
a A de	4	3
a A B e	1	4
Goal	-	-

Input string: abcde

1. Goal→ABB
2. A→Abc
3. |b
4. |a
5. B→d
6. |e

Sentential Form	Production	Position

Finding Reductions

- What are we trying to find?
  - A substring *b* that matches the right-side of a production that occurs as one step in the rightmost derivation. Informally, this substring is called a **handle**.
- Formally, a handle of a right-sentential form δ is a pair $<A \rightarrow b,k>$ where $A \rightarrow b \in P$ and k is the position in δ of *b*'s rightmost symbol.

(right-sentential form: a sentential form that occurs in some rightmost derivation).

  - Because δ is a right-sentential form, the substring to the right of a handle contains only terminal symbols. Therefore, the parser doesn't need to scan past the handle.
  - If a grammar is unambiguous, then every right-sentential form has a unique handle (sketch of proof by definition: if unambiguous then rightmost derivation is unique; then there is unique production at each step to produce a sentential form; then there is a unique position at which the rule is applied; hence, unique handle).

  If we can find those handles, we can build a derivation!

Motivating Example

Given the grammar of the left-hand side below, find a rightmost derivation for *x – 2*y* (starting from Goal there is only one, the grammar is not ambiguous!). In each step, identify the handle.

1. *Goal* → *Expr*
2. *Expr* → *Expr + Term*
3. | *Expr – Term*
4. | *Term*
5. *Term* → *Term * Factor*
6. | *Term / Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Production	**Sentential Form**	**Handle**
-	*Goal*	-
1	*Expr*	1,1
3	*Expr – Term*	3,3

<u>Problem</u>: given the sentence *x – 2*y*, find the handles!

A basic bottom-up parser

- The process of discovering a handle is called handle pruning.
- To construct a rightmost derivation, apply the simple algorithm:

  for $i=n$ to 1, step -1

  find the handle $<A \rightarrow b,k>_i$ in δ_i

  replace b with A to generate δ_{i-1}

  (needs 2n steps, where n is the length of the derivation)

- One implementation is based on using a stack to hold grammar symbols and an input buffer to hold the string to be parsed. Four operations apply:
  - **shift**: next input is shifted (pushed) onto the top of the stack
  - **reduce**: right-end of the handle is on the top of the stack; locate left-end of the handle within the stack; pop handle off stack and push appropriate non-terminal left-hand-side symbol.
  - **accept**: terminate parsing and signal success.
  - **error**: call an error recovery routine.

Implementing a shift-reduce parser

```
push $ onto the stack
token = next_token()
repeat
    if the top of the stack is a handle A→b
        then /* reduce b to A */
                pop the symbols of b off the stack
                push A onto the stack
        elseif (token != eof) /* eof: end-of-file = end-of-input */
                then /* shift */
                        push token
                        token=next_token()
                else /* error */
                        call error_handling()
until (top_of_stack == Goal && token==eof)
```

Errors show up: a) when we fail to find a handle, or b) when we hit EOF and we need to shift. The parser needs to recognise syntax errors.

Example: x–2*y

Stack	Input	Handle	Action
$	id – num * id	None	Shift
$ id	– num * id	9,1	Reduce 9
$ Factor	– num * id	7,1	Reduce 7
$ Term	– num * id	4,1	Reduce 4
$ Expr	– num * id	None	Shift !!
$ Expr –	num * id	None	Shift
$ Expr – num	* id	8,3	Reduce 8
$ Expr – Factor	* id	7,3	Reduce 7
$ Expr – Term	* id	None	Shift !!
$ Expr – Term *	id	None	Shift
$ Expr – Term * id		9,5	Reduce 9
$ Expr – Term * Factor		5,5	Reduce 5
$ Expr – Term		3,3	Reduce 3
$ Expr		1,1	Reduce 1
$ Goal		none	Accept

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

– 1. Shift until top of stack is the right end of the handle

– 2. Find the left end of the handle and reduce

(5 shifts, 9 reduces, 1 accept)

Example: x/4+2*y

Stack	Input	Handle	Action
$			

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

What can go wrong?

(think about the steps with an exclamation mark in the previous slide)

- **Shift/reduce conflicts**: the parser cannot decide whether to shift or to reduce.

  Example: the dangling-else grammar; usually due to ambiguous grammars.

  Solution: a) modify the grammar; b) resolve in favour of a shift.

- **Reduce/reduce conflicts**: the parser cannot decide which of several reductions to make.

  Example: **`id(id,id)`**; reduction is dependent on whether the first **`id`** refers to array or function.

  May be difficult to tackle.

Key to efficient bottom-up parsing: the handle-finding mechanism.

LR(1) grammars

(a beautiful example of applying theory to solve a complex problem in practice)

A grammar is LR(1) if, given a rightmost derivation, we can (I) isolate the handle of each right-sentential form, and (II) determine the production by which to reduce, by scanning the sentential form from left-to-right, going at most 1 symbol beyond the right-end of the handle.

- LR(1) grammars are widely used to construct (automatically) efficient and flexible parsers:
  - Virtually all context-free programming language constructs can be expressed in an LR(1) form.
  - LR grammars are the most general grammars parsable by a non-backtracking, shift-reduce parser (deterministic CFGs).
  - Parsers can be implemented in time proportional to tokens+reductions.
  - LR parsers detect an error as soon as possible in a left-to-right scan of the input.

L stands for left-to-right scanning of the input; R for constructing a rightmost derivation in reverse; 1 for the number of input symbols for lookahead.

LR Parsing: Background

- Read tokens from an input buffer (same as with shift-reduce parsers)
- Add an extra state information after each symbol in the stack. The state summarises the information contained in the stack below it. The stack would look like:

  $\$\ S_0\ Expr\ S_1\ -\ S_2\ num\ S_3$

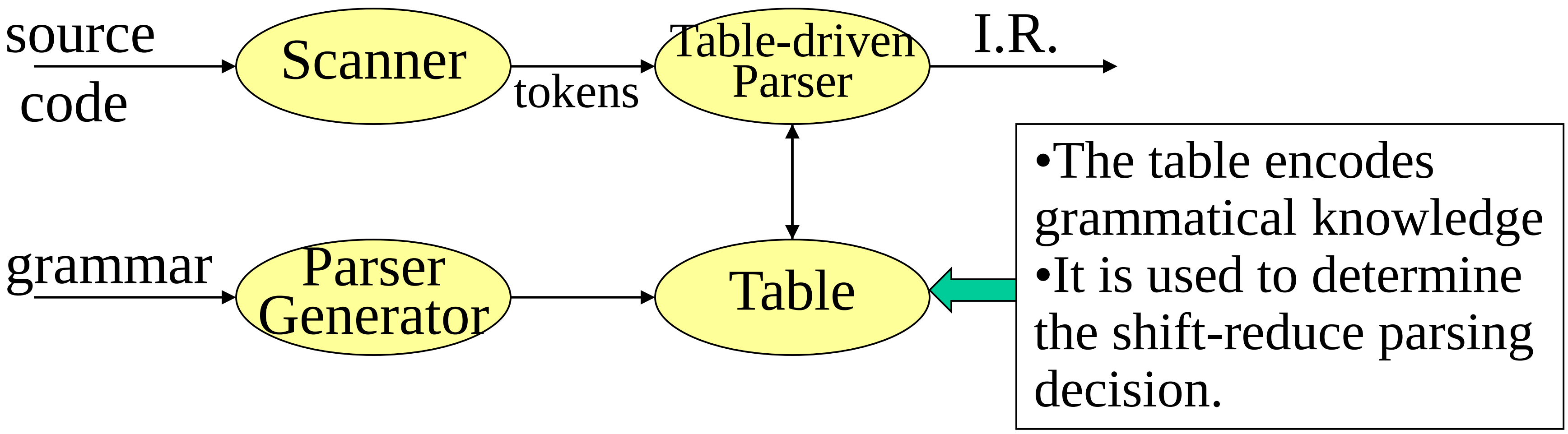
- Use a table that consists of two parts:
  – **action**[state_on_top_of_stack, input_symbol]: returns one of: shift s (push a symbol and a state); reduce by a rule; accept; error.
  – **goto**[state_on_top_of_stack,non_terminal_symbol]: returns a new state to push onto the stack after a reduction.

Skeleton code for an LR Parser

```
Push $ onto the stack
push s0
token=next_token()
repeat
  s=top_of_the_stack  /* not pop! */
  if ACTION[s,token]==‘reduce A→b’
     then  pop 2*(symbols_of_b) off the stack
           s=top_of_the_stack  /* not pop! */
           push A; push GOTO[s,A]
     elseif ACTION[s,token]==‘shift sx’
                then push token; push sx
                     token=next_token()
                elseif ACTION[s,token]==‘accept’
                     then break
                     else report_error
end repeat
report_success
```

The Big Picture: Prelude to what follows

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context for handle recognition.
- They can be built by hand; perfect to automate too!
- <u>Summary</u>: Bottom-up parsing is more powerful!



Example

Consider the following grammar and tables:

1. *Goal* $\rightarrow$ *CatNoise*
2. *CatNoise* $\rightarrow$ *CatNoise miau*
3. | *miau*

STATE	ACTION		GOTO
	eof	miau	CatNoise
0	-	Shift 2	1
1	accept	Shift 3	
2	Reduce 3	Reduce 3	
3	Reduce 2	Reduce 2	

<u>Example 1</u>: (input string miau)

Stack	**Input**	**Action**
$ s0	miau eof	Shift 2
$ s0 miau s2	eof	Reduce 3
$ s0 CatNoise s1	eof	Accept

<u>Example 2</u>: (input string miau miau)

Stack	**Input**	**Action**
$ s0	miau miau eof	Shift 2
$ s0 miau s2	miau eof	Reduce 3
$ s0 CatNoise s1	miau eof	Shift 3
$ s0 CatNoise s1 miau s3	eof	Reduce 2
$ s0 CatNoise s1	eof	accept

Note that there cannot be a syntax error with CatNoise, because it has only 1 terminal symbol. “miau woof” is a lexical problem, not a syntax error!

eof is a convention for end-of-file (=end of input)

Example: the expression grammar

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

STATE	ACTION							GOTO		
	eof	+	–	*	/	num	id	Expr	Term	Factor
0						S 4	S 5	1	2	3
1	Acc	S 6	S 7							
2	R 4	R 4	R 4	S 8	S 9					
3	R 7	R 7	R 7	R 7	R 7					
4	R 8	R 8	R 8	R 8	R 8					
5	R 9	R 9	R 9	R 9	R 9					
6						S 4	S 5		10	3
7						S 4	S 5		11	3
8						S 4	S 5			12
9						S 4	S 5			13
10	R 2	R 2	R 2	S 8	S 9					
11	R 3	R 3	R 3	S 8	S 9					
12	R 5	R 5	R 5	R 5	R 5					
13	R 6	R 6	R 6	R 6	R 6					

Parse: a) X+2*Y
b)X/4 – Y*5

STATE	ACTION							GOTO		
	eof	+	–	*	/	num	id	Expr	Term	Factor
0						S 4	S 5	1	2	3
1	Acc	S 6	S 7							
2	R 4	R 4	R 4	S 8	S 9					
3	R 7	R 7	R 7	R 7	R 7					
4	R 8	R 8	R 8	R 8	R 8					
5	R 9	R 9	R 9	R 9	R 9					
6						S 4	S 5		10	3
7						S 4	S 5		11	3
8						S 4	S 5			12
9						S 4	S 5			13
10	R 2	R 2	R 2	S 8	S 9					
11	R 3	R 3	R 3	S 8	S 9					
12	R 5	R 5	R 5	R 5	R 5					
13	R 6	R 6	R 6	R 6	R 6					

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

a) X+2*Y

Stack	Input	Action
$s0	X/4-Y*5	S5
$s0Xs5	/4-Y*5	R9
$s0FactorS3	/4-Y*5	R7
$s0TermS2	/4-Y*5	S9
$S0TermS2/S9	4-Y*5	S4
$S0TermS2/S94S4	-Y*5	R8
$S0TermS2/S9FactorS13	-Y*5	R6
$S0TermS2	-Y*5	R4
$S0ExprS1	-Y*5	S7
$S0ExprS1-S7	Y*5	S5
$S0ExprS1-S7YS5	*5	R9
$S0ExprS1-S7FactorS3	*5	R7
$S0ExprS1-S7TermS2	*5	S8
$S0ExprS1-S7TermS2*S8	5	S4
$S0ExprS1-S7TermS2*S85S4	Eof	R8
$S0ExprS1-S7TermS2*S8FactorS12	Eof	R5
$S0ExprS1-S7TermS11	Eof	R3
$S0ExprS1	Eof	Acc

STATE	ACTION							GOTO		
	eof	+	–	*	/	num	id	Expr	Term	Factor
0						S 4	S 5	1	2	3
1	Acc	S 6	S 7							
2	R 4	R 4	R 4	S 8	S 9					
3	R 7	R 7	R 7	R 7	R 7					
4	R 8	R 8	R 8	R 8	R 8					
5	R 9	R 9	R 9	R 9	R 9					
6						S 4	S 5		10	3
7						S 4	S 5		11	3
8						S 4	S 5			12
9						S 4	S 5			13
10	R 2	R 2	R 2	S 8	S 9					
11	R 3	R 3	R 3	S 8	S 9					
12	R 5	R 5	R 5	R 5	R 5					
13	R 6	R 6	R 6	R 6	R 6					

b)X/4 – Y*5

1. *Goal* → *Expr*
2. *Expr* → *Expr* + *Term*
3. | *Expr* – *Term*
4. | *Term*
5. *Term* → *Term* * *Factor*
6. | *Term* / *Factor*
7. | *Factor*
8. *Factor* → *number*
9. | *id*

Example:

Goal→Expr
Expr→Term-Expr
Expr→Term
*Term→Factor*Term*
Term→Factor
Factor→id

STATE	ACTION				GOTO		
	id	-	*	eof	Expr	Term	Factor
0	S 4				1	2	3
1				Accept			
2		S 5		R 3			
3		R 5	S 6	R 5			
4		R 6	R 6	R 6			
5	S 4				7	2	3
6	S 4					8	3
7				R 2			
8		R 4		R 4			

STATE	ACTION				GOTO		
	id	-	*	eof	Expr	Term	Factor
0	S 4				1	2	3
1				Accept			
2		S 5		R 3			
3		R 5	S 6	R 5			
4		R 6	R 6	R 6			
5	S 4				7	2	3
6	S 4					8	3
7				R 2			
8		R 4		R 4			

$Goal \rightarrow Expr$

$Expr \rightarrow Term\text{-}Expr$

$Expr \rightarrow Term$

$Term \rightarrow Factor*Term$

$Term \rightarrow Factor$

$Factor \rightarrow id$

X – Y *5

STATE	ACTION				GOTO		
	id	-	*	eof	Expr	Term	Factor
0	S 4				1	2	3
1				Accept			
2		S 5		R 3			
3		R 5	S 6	R 5			
4		R 6	R 6	R 6			
5	S 4				7	2	3
6	S 4					8	3
7				R 2			
8		R 4		R 4			

Goal→Expr
Expr→Term-Expr
Expr→Term
*Term→Factor*Term*
Term→Factor
Factor→id

X - Y /5

Example : LR(1) Table Generation

1. $Goal \rightarrow CatNoise$
2. $CatNoise \rightarrow CatNoise\ miau$
3. $\qquad\qquad\qquad\quad /\ miau$

Example : LR(1) Table Generation

Goal→Expr
Expr→Term-Expr
Expr→Term
*Term→Factor*Term*
Term→Factor
Factor→id

STATE	ACTION				GOTO		

1. $Goal \rightarrow Expr$
2. $Expr \rightarrow Expr + Term$
3. $\quad | \ Expr - Term$
4. $\quad | \ Term$
5. $Term \rightarrow Term * Factor$
6. $\quad | \ Term \, / \, Factor$
7. $\quad | \ Factor$
8. $Factor \rightarrow number$
9. $\quad | \ id$

Summary

- **Top-Down Recursive Descent**: Pros: Fast, Good locality, Simple, good error-handling. Cons: Hand-coded, high-maintenance.
- **LR(1):** Pros: Fast, deterministic languages, automatable. Cons: large working sets, poor error messages.
- **What is left to study?**
  - Checking for context-sensitive properties
  - Laying out the abstractions for programs & procedures.
  - Generating code for the target machine.
  - Generating good code for the target machine.

Example: The Table (slide 4 of 4)

Goal→Expr
Expr→Term-Expr
Expr→Term
*Term→Factor*Term*
Term→Factor
Factor→id

STATE	ACTION				GOTO		
	id	-	*	eof	Expr	Term	Factor
0	S 4				1	2	3
1				Accept			
2		S 5		R 3			
3		R 5	S 6	R 5			
4		R 6	R 6	R 6			
5	S 4				7	2	3
6	S 4					8	3
7				R 2			
8		R 4		R 4			