



COMP 412
FALL 2009

Bottom-up Parsing, Part I

Comp 412

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Parsing Techniques



Top-down parsers (LL(1), recursive descent)

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" \Rightarrow may need to backtrack
- Some grammars are backtrack-free *(predictive parsing)*

Bottom-up parsers (LR(1), operator precedence)

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Bottom-up Parsing

(definitions)



The point of parsing is to construct a derivation

A derivation consists of a series of rewrite steps

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

- Each γ_i is a sentential form
 - If γ contains only terminal symbols, γ is a **sentence** in $L(G)$
 - If γ contains 1 or more non-terminals, γ is a **sentential form**
- To get γ_i from γ_{i-1} , expand some NT $A \in \gamma_{i-1}$ by using $A \rightarrow \beta$
 - Replace the occurrence of $A \in \gamma_{i-1}$ with β to get γ_i
 - In a leftmost derivation, it would be the first NT $A \in \gamma_{i-1}$

A **left-sentential form** occurs in a leftmost derivation

A **right-sentential form** occurs in a rightmost derivation

Bottom-up parsers build a rightmost derivation in reverse

Bottom-up Parsing

(definitions)



A bottom-up parser builds a derivation by working from the input sentence back toward the start symbol S

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence}$$

← bottom-up

To reduce γ_i to γ_{i-1} match some *rhs* β against γ_i then replace β with its corresponding *lhs*, A . (*assuming the production $A \rightarrow \beta$*)

In terms of the parse tree, it works from leaves to root

- Nodes with no parent in a partial tree form its *upper fringe*
- Since each replacement of β with A shrinks the upper fringe, we call it a *reduction*.
- Rightmost derivation in reverse processes words *left to right*

The parse tree need not be built, it can be simulated

$$|\text{parse tree nodes}| = |\text{terminal symbols}| + |\text{reductions}|$$



Finding Reductions

Consider the simple grammar

0	Goal	→	<u>a</u> A B <u>e</u>
1	A	→	A <u>b</u> <u>c</u>
2			<u>b</u>
3	B	→	<u>d</u>

<i>Sentential Form</i>	<i>Next Reduction</i>	
	<i>Prod'n</i>	<i>Pos'n</i>
<u>ab</u> bcde	2	2
<u>a</u> A <u>bc</u> de	1	4
<u>a</u> A <u>d</u> e	3	3
<u>a</u> A B <u>e</u>	0	4
Goal	—	—

And the input string abbcde

The trick is scanning the input and finding the next reduction
The mechanism for doing this must be efficient

While the process of finding the next reduction appears to be almost oracular, it can be automated in an efficient way for a large class of grammars

Finding Reductions

(Handles)



The parser must find a substring β of the tree's frontier that matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation $(\Rightarrow \beta \rightarrow A \text{ is in RRD})$

Informally, we call this substring β a *handle*

Formally,

A *handle* of a right-sentential form γ is a pair $\langle A \rightarrow \beta, k \rangle$ where $A \rightarrow \beta \in P$ and k is the position in γ of β 's rightmost symbol.

If $\langle A \rightarrow \beta, k \rangle$ is a handle, then replacing β at k with A produces the right sentential form from which γ is derived in the rightmost derivation.

Because γ is a right-sentential form, the substring to the right of a handle contains *only terminal symbols*

\Rightarrow the parser doesn't need to scan past the handle *(very far)*

Example

derivation

(a very busy slide)



0	Goal	→	Expr
1	Expr	→	Expr + Term
2			Expr - Term
3			Term
4	Term	→	Term * Factor
5			Term / Factor
6			Factor
7	Factor	→	<u>number</u>
8			<u>id</u>
9			(Expr)

Prod'n	Sentential Form	Handle
—	Goal	—
0	Expr	0,1
2	Expr - Term	2,3
4	Expr - Term * Factor	4,5
8	Expr - Term * <id,y>	8,5
6	Expr - Factor * <id,y>	6,3
7	Expr - <num,2> * <id,y>	7,3
3	Term - <num,2> * <id,y>	3,1
6	Factor - <num,2> * <id,y>	6,1
8	<id,x> - <num,2> * <id,y>	8,1

A simple right-recursive form of the classic expression grammar

Handles for rightmost derivation of $x = 2 * y$

Bottom-up Parsing

(Abstract View)



A bottom-up parser repeatedly finds a handle in the current right-sentential form and replaces it with the NT that corresponds to the *rhs* in the handle.

To construct a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$$

Apply the following simple algorithm

for $i \leftarrow n$ to 1 by -1

Find the handle $\langle A_i \rightarrow \beta_i, k_i \rangle$ in γ_i

Replace β_i with A_i to generate γ_{i-1}

of course, n is unknown until the derivation is built

This takes $2n$ steps

If we think about this process in terms of the syntax tree, we are locating a handle on the upper frontier of the partially complete parse tree and pruning it - a *handle pruning parser*



More on Handles

Bottom-up reduce parsers find a reverse rightmost derivation

- Process input left to right
 - Upper fringe of partially completed parse tree is $NT^* T^*$
 - The handle always appears with its right end at the junction between NT^* and T^* (*the hot spot for LR parsing*)
 - We can keep the prefix of the upper fringe of the partially completed parse tree on a stack
 - The stack makes the position information irrelevant

More on Handles



Bottom-up parsers find a reverse rightmost derivation

- Process input left to right
 - Upper fringe of partially completed parse tree is $NT^* T^*$
 - The handle always appears with its right end at the junction between NT^* and T^* (*the hot spot for LR parsing*)
 - We can keep the prefix of the upper fringe of the partially completed parse tree on a stack
 - The stack makes the position information irrelevant
- Handles appear at the top of the stack
- All the information for the decision is at the hot spot
 - The next word in the input stream
 - The rightmost NT on the fringe & its immediate left neighbors
 - In an LR parser, additional information in the form of a “state”



Handles Are Unique

Theorem:

*If G is unambiguous, then every right-sentential form has a **unique** handle.*

Sketch of Proof:

- 1 G is unambiguous \Rightarrow rightmost derivation is unique
- 2 \Rightarrow a unique production $A \rightarrow \beta$ applied to derive γ_i from γ_{i-1}
- 3 \Rightarrow a unique position k at which $A \rightarrow \beta$ is applied
- 4 \Rightarrow a unique handle $\langle A \rightarrow \beta, k \rangle$

This all follows from the definitions

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

The handle always appears with its right end at the stack top
 \rightarrow How many right-hand sides must the parser consider?



Shift-reduce Parsing

To implement a bottom-up, handle-pruning parser, we adopt the shift-reduce paradigm

A **shift-reduce parser** is a stack automaton with four actions

- **Shift** — next word is shifted onto the stack
- **Reduce** — right end of handle is at top of stack
Locate left end of handle within the stack
Pop handle off stack & push appropriate *lhs*
- **Accept** — stop parsing & report success
- **Error** — call an error reporting/recovery routine

Accept & Error are simple

Shift is just a push and a call to the scanner

Reduce takes $|rhs|$ pops & 1 push

*But how does the parser know when to shift and when to reduce?
It shifts until it has a handle at the top of the stack.*



Bottom-up Parser

A simple *shift-reduce parser*:

```
push INVALID
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
  if the top of the stack is a handle  $A \rightarrow \beta$ 
    then // reduce  $\beta$  to  $A$ 
      pop  $|\beta|$  symbols off the stack
      push  $A$  onto the stack
  else if (token  $\neq$  EOF)
    then // shift
      push token
      token ← next_token( )
  else // need to shift, but out of input
    report an error
```

What happens on an error?

- It fails to find a handle
- Thus, it keeps shifting
- Eventually, it consumes all input

This parser reads all input before reporting an error, not a desirable property.

Error localization is an issue in the handle-finding process that affects the practicality of shift-reduce parsers...

Back to $x - 2 * y$



Stack	Input	Handle	Action
\$	<u>id</u> - num * <u>id</u>	none	shift
\$ <u>id</u>	- <u>num</u> * <u>id</u>		

0	Goal	→	Expr
1	Expr	→	Expr + Term
2			Expr - Term
3			Term
4	Term	→	Term * Factor
5			Term / Factor
6			Factor
7	Factor	→	<u>number</u>
8			<u>id</u>
9			(Expr)

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce

Back to $x - 2 * y$



Stack	Input	Handle	Action
\$	<u>id</u> - num * <u>id</u>	none	shift
\$ <u>id</u>	- <u>num</u> * <u>id</u>	8,1	reduce 8
\$ <i>Factor</i>	- <u>num</u> * <u>id</u>	6,1	reduce 6
\$ <i>Term</i>	- <u>num</u> * <u>id</u>	3,1	reduce 4
\$ <i>Expr</i>	- <u>num</u> * <u>id</u>		

0	Goal	→	<i>Expr</i>
1	<i>Expr</i>	→	<i>Expr</i> + <i>Term</i>
2			<i>Expr</i> - <i>Term</i>
3			<i>Term</i>
4	<i>Term</i>	→	<i>Term</i> * <i>Factor</i>
5			<i>Term</i> / <i>Factor</i>
6			<i>Factor</i>
7	<i>Factor</i>	→	<u>number</u>
8			<u>id</u>
9			(<i>Expr</i>)

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce

Back to $x - 2 * y$



Stack	Input	Handle	Action
\$	<u>id</u> - num * <u>id</u>	none	shift
\$ <u>id</u>	- <u>num</u> * <u>id</u>	8,1	reduce 8
\$ Factor	- <u>num</u> * <u>id</u>	6,1	reduce 6
\$ Term	- <u>num</u> * <u>id</u>	3,1	reduce 3
\$ Expr	- <u>num</u> * <u>id</u>	none	shift
\$ Expr -	<u>num</u> * <u>id</u>	none	shift
\$ Expr - <u>num</u>	* <u>id</u>		

0	Goal	→	Expr
1	Expr	→	Expr + Term
2			Expr - Term
3			Term
4	Term	→	Term * Factor
5			Term / Factor
6			Factor
7	Factor	→	<u>number</u>
8			<u>id</u>
9			(Expr)

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce



Back to $x - 2 * y$

Stack	Input	Handle	Action
\$	<u>id</u> - <u>num</u> * <u>id</u>	none	shift
\$ <u>id</u>	- <u>num</u> * <u>id</u>	8,1	reduce 8
\$ <i>Factor</i>	- <u>num</u> * <u>id</u>	6,1	reduce 6
\$ <i>Term</i>	- <u>num</u> * <u>id</u>	3,1	reduce 3
\$ <i>Expr</i>	- <u>num</u> * <u>id</u>	none	shift
\$ <i>Expr</i> -	<u>num</u> * <u>id</u>	none	shift
\$ <i>Expr</i> - <u>num</u>	* <u>id</u>	7,3	reduce 7
\$ <i>Expr</i> - <i>Factor</i>	* <u>id</u>	6,3	reduce 6
\$ <i>Expr</i> - <i>Term</i>	* <u>id</u>		

0	Goal	→	<i>Expr</i>
1	<i>Expr</i>	→	<i>Expr</i> + <i>Term</i>
2			<i>Expr</i> - <i>Term</i>
3			<i>Term</i>
4	<i>Term</i>	→	<i>Term</i> * <i>Factor</i>
5			<i>Term</i> / <i>Factor</i>
6			<i>Factor</i>
7	<i>Factor</i>	→	<u>number</u>
8			<u>id</u>
9			(<i>Expr</i>)

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce



Back to $x - 2 * y$

Stack	Input	Handle	Action
\$	<u>id</u> - <u>num</u> * <u>id</u>	none	shift
\$ <u>id</u>	- <u>num</u> * <u>id</u>	8,1	reduce 8
\$ Factor	- <u>num</u> * <u>id</u>	6,1	reduce 6
\$ Term	- <u>num</u> * <u>id</u>	3,1	reduce 3
\$ Expr	- <u>num</u> * <u>id</u>	none	shift
\$ Expr -	<u>num</u> * <u>id</u>	none	shift
\$ Expr - <u>num</u>	* <u>id</u>	7,3	reduce 7
\$ Expr - Factor	* <u>id</u>	6,3	reduce 6
\$ Expr - Term	* <u>id</u>	none	shift
\$ Expr - Term *	<u>id</u>	none	shift
\$ Expr - Term * <u>id</u>			

0	Goal	→	Expr
1	Expr	→	Expr + Term
2			Expr - Term
3			Term
4	Term	→	Term * Factor
5			Term / Factor
6			Factor
7	Factor	→	<u>number</u>
8			<u>id</u>
9			(Expr)

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce



Back to x - 2 * y

Stack	Input	Handle	Action
\$	<u>id</u> - num * <u>id</u>	none	shift
\$ <u>id</u>	- <u>num</u> * <u>id</u>	8,1	reduce 8
\$ Factor	- <u>num</u> * <u>id</u>	6,1	reduce 6
\$ Term	- <u>num</u> * <u>id</u>	3,1	reduce 3
\$ Expr	- <u>num</u> * <u>id</u>	none	shift
\$ Expr -	<u>num</u> * <u>id</u>	none	shift
\$ Expr - <u>num</u>	* <u>id</u>	7,3	reduce 7
\$ Expr - Factor	* <u>id</u>	6,3	reduce 6
\$ Expr - Term	* <u>id</u>	none	shift
\$ Expr - Term *	<u>id</u>	none	shift
\$ Expr - Term * <u>id</u>		8,5	reduce 8
\$ Expr - Term * Factor		4,5	reduce 4
\$ Expr - Term		2,3	reduce 2
\$ Expr		0,1	reduce 0
\$ Goal		none	accept

0	Goal	→	Expr
1	Expr	→	Expr + Term
2			Expr - Term
3			Term
4	Term	→	Term * Factor
5			Term / Factor
6			Factor
7	Factor	→	<u>number</u>
8			<u>id</u>
9			{ Expr }

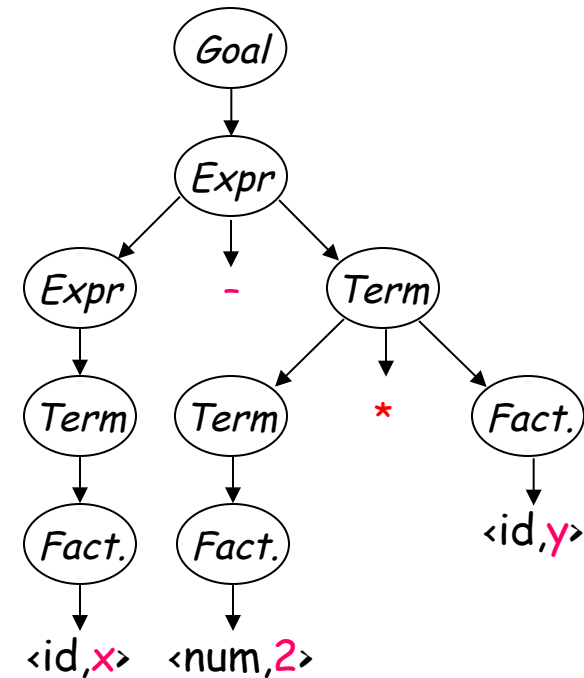
5 shifts +
9 reduces +
1 accept

1. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle and reduce



Back to x - 2 * y

Stack	Input	Action
\$	<u>id</u> - <u>num</u> * <u>id</u>	shift
\$ <u>id</u>	- <u>num</u> * <u>id</u>	reduce 8
\$ <i>Factor</i>	- <u>num</u> * <u>id</u>	reduce 6
\$ <i>Term</i>	- <u>num</u> * <u>id</u>	reduce 3
\$ <i>Expr</i>	- <u>num</u> * <u>id</u>	shift
\$ <i>Expr</i> -	<u>num</u> * <u>id</u>	shift
\$ <i>Expr</i> - <u>num</u>	* <u>id</u>	reduce 7
\$ <i>Expr</i> - <i>Factor</i>	* <u>id</u>	reduce 6
\$ <i>Expr</i> - <i>Term</i>	* <u>id</u>	shift
\$ <i>Expr</i> - <i>Term</i> *	<u>id</u>	shift
\$ <i>Expr</i> - <i>Term</i> * <u>id</u>		reduce 8
\$ <i>Expr</i> - <i>Term</i> * <i>Factor</i>		reduce 4
\$ <i>Expr</i> - <i>Term</i>		reduce 2
\$ <i>Expr</i>		reduce 0
\$ <i>Goal</i>		accept



Corresponding Parse Tree

An Important Lesson about

The additional left context is precisely the reason that LR(1) grammars express a superset of the languages that can be expressed as LL(1) grammars

- A handle must be a substring of α
 - It must match the right hand side of a production
 - There must be some rightmost derivation from the goal symbol that produces the sentential form γ with $A \rightarrow \beta$ as the last production applied
- Simply looking for right hand sides that match strings is not good enough
- **Critical Question:** How can we know when we have found a handle without generating lots of different derivations?
 - **Answer:** We use left context, encoded in the sentential form, left context encoded in a "parser state", and a lookahead at the next word in the input. (Formally, 1 word beyond the handle.)
 - Parser states are derived by reachability analysis on grammar
 - We build all of this knowledge into a handle-recognizing DFA



LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- LR(1) parsers recognize languages that have an LR(1) grammar

Informal definition:

A grammar is LR(1) if, given a rightmost derivation

$$S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \textit{sentence}$$

We can

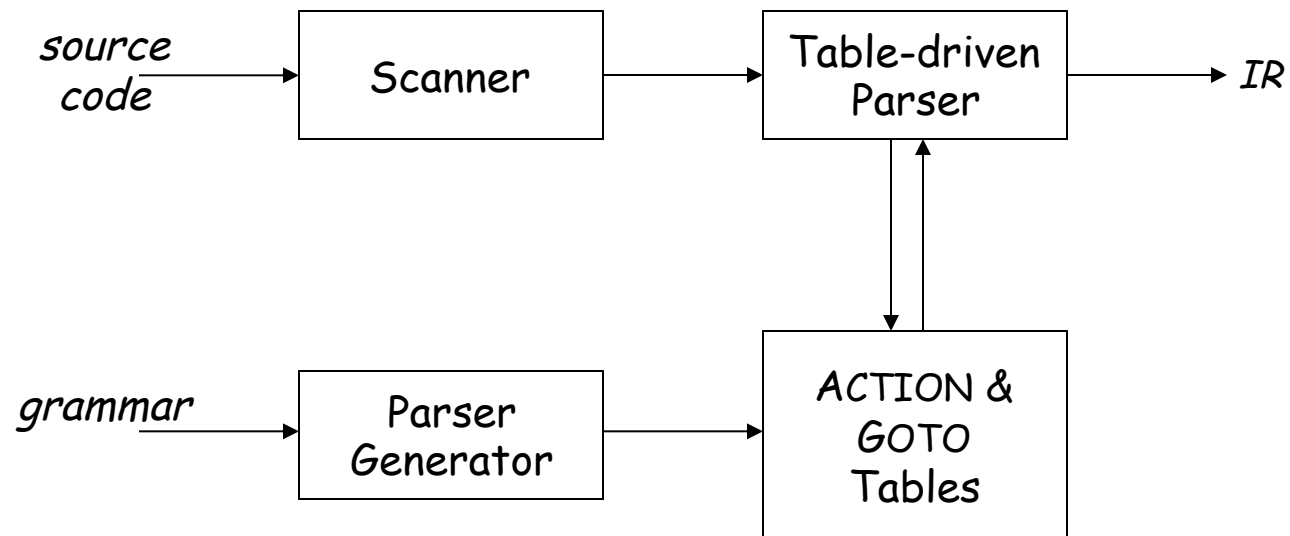
1. *isolate the handle of each right-sentential form γ_i , and*
2. *determine the production by which to reduce,*

by scanning γ_i from *left-to-right*, going at most 1 symbol beyond the right end of the handle of γ_i



LR(1) Parsers

A table-driven LR(1) parser looks like



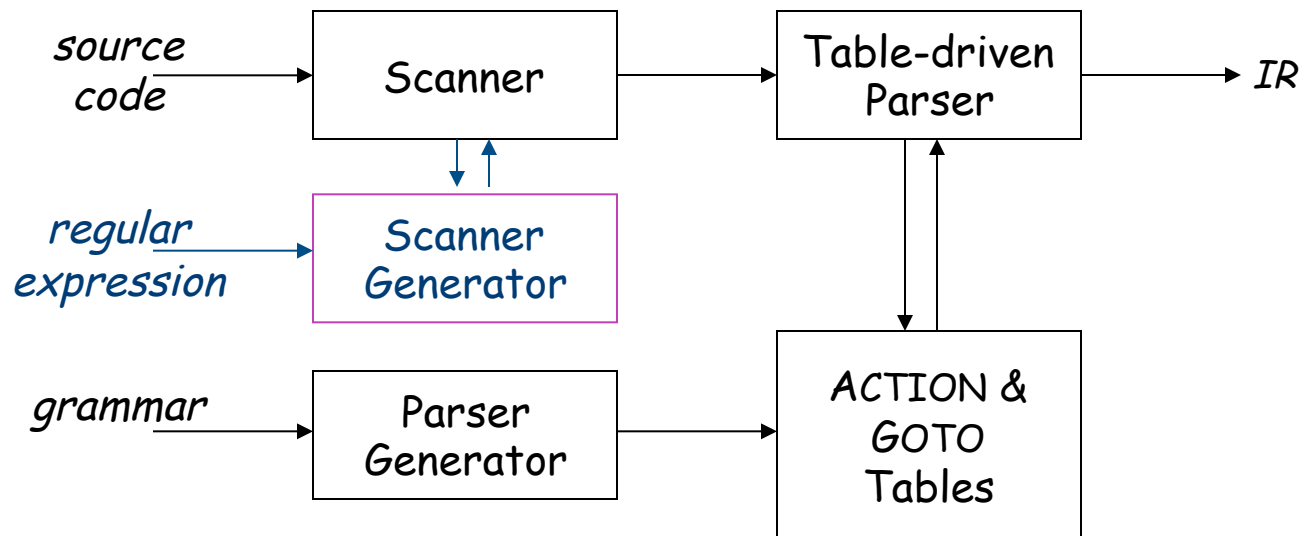
Tables can be built by hand

However, this is a perfect task to automate



LR(1) Parsers

A table-driven LR(1) parser looks like



Tables can be built by hand

However, this is a perfect task to automate

Just like automating construction of scanners ...

Except that compiler writers use parser generators ...



LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push( $s_0$ ); // initial state
token = scanner.next_token();
loop forever {
  s = stack.top();
  if ( ACTION[s,token] == "reduce  $A \rightarrow \beta$ " ) then {
    stack.popnum( $2 * |\beta|$ ); // pop  $2 * |\beta|$  symbols
    s = stack.top();
    stack.push(A); // push A
    stack.push(GOTO[s,A]); // push next state
  }
  else if ( ACTION[s,token] == "shift  $s_i$ " ) then {
    stack.push(token); stack.push( $s_i$ );
    token ← scanner.next_token();
  }
  else if ( ACTION[s,token] == "accept"
            & token == EOF )
    then break;
  else throw a syntax error;
}
report success;
```

The skeleton parser

- relies on a stack & a scanner
- uses two tables, called ACTION & GOTO
ACTION: state x word \rightarrow state
GOTO: state x NT \rightarrow state
- shifts |words| times
- reduces |derivation| times
- accepts at most once
- detects errors by failure of the other three cases
- follows basic scheme for shift-reduce parsing from last lecture

LR(1) Parsers

(parse tables)



To make a parser for $L(G)$, need a set of tables

The grammar

- 1 *Goal* → *SheepNoise*
- 2 *SheepNoise* → *SheepNoise* baa
- 3 | baa

Remember, this is the left-recursive *SheepNoise*; EaC shows the right-recursive version.

The tables

ACTION Table		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

GOTO Table	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



Example Parse 1

The string baa

Stack	Input	Action
\$ s_0	<u>baa</u> EOF	

- 1 *Goal* → *SheepNoise*
- 2 *SheepNoise* → *SheepNoise* baa
- 3 | baa

State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



Example Parse 1

The string baa

Stack	Input	Action
\$ s_0	<u>baa</u> EOF	<i>shift 2</i>
\$ s_0 <u>baa</u> s_2	EOF	

- 1 *Goal* → *SheepNoise*
- 2 *SheepNoise* → *SheepNoise* baa
- 3 | baa

ACTION Table		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

GOTO Table	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



Example Parse 1

The string baa

Stack	Input	Action
\$ s_0	<u>baa</u> EOF	<i>shift 2</i>
\$ s_0 <u>baa</u> s_2	EOF	<i>reduce 3</i>
\$ s_0 SN s_1	EOF	

1	<i>Goal</i>	→	<i>SheepNoise</i>
2	<i>SheepNoise</i>	→	<i>SheepNoise</i> <u>baa</u>
3			<u>baa</u>

ACTION Table		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

GOTO Table	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



Example Parse 1

The string baa

Stack	Input	Action
\$ s_0	<u>baa</u> EOF	<i>shift 2</i>
\$ s_0 <u>baa</u> s_2	EOF	<i>reduce 3</i>
\$ s_0 SN s_1	EOF	<i>accept</i>

- 1 *Goal* → *SheepNoise*
- 2 *SheepNoise* → *SheepNoise* baa
- 3 | baa

Notice that we never cleared the stack — the table construction moved *accept* earlier by one action

ACTION Table		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

GOTO Table	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



Example Parse 2

The string baa baa

Stack	Input	Action
\$ s_0	<u>baa</u> <u>baa</u> EOF	1 <i>Goal</i> → <i>SheepNoise</i>
		2 <i>SheepNoise</i> → <i>SheepNoise</i> <u>baa</u>
		3 <u>baa</u>

ACTION Table		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

GOTO Table	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



Example Parse 2

The string baa baa

Stack	Input	Action
\$ s_0	<u>baa</u> <u>baa</u> EOF	<i>shift 2</i>
\$ s_0 <u>baa</u> s_2	<u>baa</u> EOF	

1	<i>Goal</i>	→	<i>SheepNoise</i>
2	<i>SheepNoise</i>	→	<i>SheepNoise</i> <u>baa</u>
3			<u>baa</u>

ACTION Table		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

GOTO Table	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



Example Parse 2

The string baa baa

Stack	Input	Action	
\$ s_0	<u>baa</u> <u>baa</u> EOF	<i>shift 2</i>	1 <i>Goal</i> → <i>SheepNoise</i>
\$ s_0 <u>baa</u> s_2	<u>baa</u> EOF	<i>reduce 3</i>	2 <i>SheepNoise</i> → <i>SheepNoise</i> <u>baa</u>
\$ s_0 SN s_1	<u>baa</u> EOF		3 <u>baa</u>

Last example, we faced EOF and we accepted. With baa, we shift ...

ACTION Table		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

GOTO Table	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



Example Parse 2

The string baa baa

Stack	Input	Action
\$ s_0	<u>baa</u> <u>baa</u> EOF	<i>shift 2</i>
\$ s_0 <u>baa</u> s_2	<u>baa</u> EOF	<i>reduce 3</i>
\$ s_0 SN s_1	<u>baa</u> EOF	<i>shift 3</i>
\$ s_0 SN s_1 <u>baa</u> s_3	EOF	

1	<i>Goal</i>	→	<i>SheepNoise</i>
2	<i>SheepNoise</i>	→	<i>SheepNoise</i> <u>baa</u>
3			<u>baa</u>

ACTION Table		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

GOTO Table	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



Example Parse 2

The string baa baa

Stack	Input	Action	
\$ s ₀	<u>baa</u> <u>baa</u> EOF	<i>shift 2</i>	1 <i>Goal</i> → <i>SheepNoise</i>
\$ s ₀ <u>baa</u> s ₂	<u>baa</u> EOF	<i>reduce 3</i>	2 <i>SheepNoise</i> → <i>SheepNoise</i> <u>baa</u>
\$ s ₀ SN s ₁	<u>baa</u> EOF	<i>shift 3</i>	3 <u>baa</u>
\$ s ₀ SN s ₁ <u>baa</u> s ₃	EOF	<i>reduce 2</i>	
\$ s ₀ SN s ₁	EOF		

Now, we accept

ACTION Table		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

GOTO Table	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



Example Parse 2

The string baa baa

Stack	Input	Action
\$ s_0	<u>baa</u> <u>baa</u> EOF	<i>shift 2</i>
\$ s_0 <u>baa</u> s_2	<u>baa</u> EOF	<i>reduce 3</i>
\$ s_0 SN s_1	<u>baa</u> EOF	<i>shift 3</i>
\$ s_0 SN s_1 <u>baa</u> s_3	EOF	<i>reduce 2</i>
\$ s_0 SN s_1	EOF	<i>accept</i>

1	<i>Goal</i>	→	<i>SheepNoise</i>
2	<i>SheepNoise</i>	→	<i>SheepNoise</i> <u>baa</u>
3			<u>baa</u>

ACTION Table		
State	EOF	<u>baa</u>
0	—	<i>shift 2</i>
1	<i>accept</i>	<i>shift 3</i>
2	<i>reduce 3</i>	<i>reduce 3</i>
3	<i>reduce 2</i>	<i>reduce 2</i>

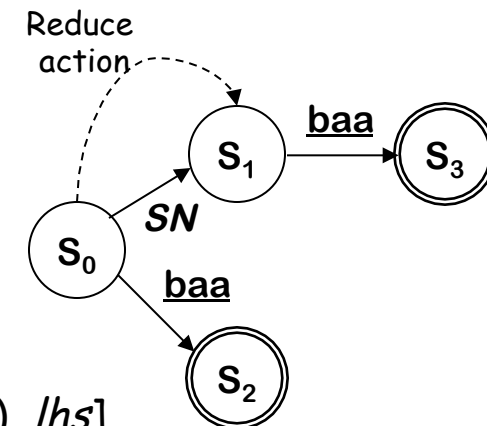
GOTO Table	
State	<i>SheepNoise</i>
0	1
1	0
2	0
3	0



LR(1) Parsers

How does this LR(1) stuff work?

- Unambiguous grammar \Rightarrow unique rightmost derivation
- Keep upper fringe on a stack
 - All active handles include top of stack (TOS)
 - Shift inputs until TOS is right end of a handle
- Language of handles is regular (finite)
 - Build a handle-recognizing DFA
 - ACTION & GOTO tables encode the DFA
- To match subterm, invoke subterm DFA & leave old DFA's state on stack
- Final state in DFA \Rightarrow a *reduce* action
 - New state is $GOTO[\text{state at TOS (after pop)}, lhs]$
 - For SN , this takes the DFA to s_1



Control DFA for SN