Bottom-up Parsing

Recap of Top-down Parsing

- Top-down parsers build syntax tree from root to leaves
- Left-recursion causes non-termination in top-down parsers
  - Transformation to eliminate left recursion
  - Transformation to eliminate common prefixes in right recursion
- FIRST, FIRST$^+$, & FOLLOW sets + LL(1) condition
  - LL(1) uses left-to-right scan of the input, leftmost derivation of the sentence, and 1 word lookahead
  - LL(1) condition means grammar works for predictive parsing
- Given an LL(1) grammar, we can
  - Build a recursive descent parser
  - Build a table-driven LL(1) parser
- LL(1) parser doesn't explicitly build the parse tree
  - Keeps lower fringe of partially complete tree on the stack

Parsing Techniques

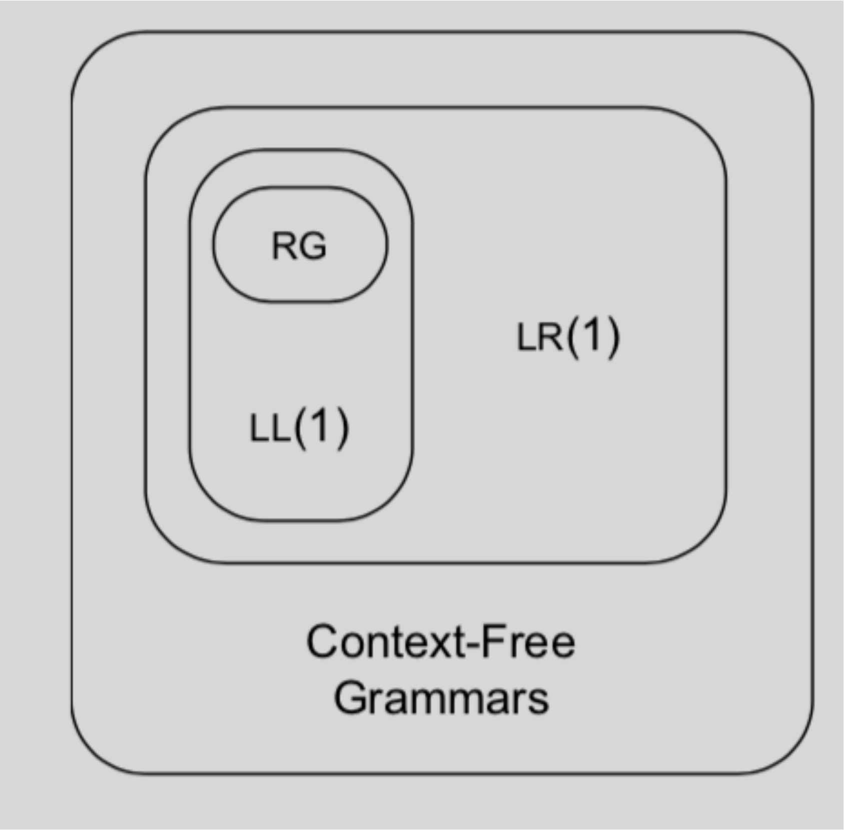
Top-down parsers (LL(1), recursive descent)

- 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
- Bottom-up parsers handle a large class of grammars

Bottom-up parser handle a larger class of grammars



Bottom-up Parsing (recap of definitions)

The point of parsing is to construct a derivation

A derivation consists of a series of rewrite steps

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

- 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

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

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

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 $A \rightarrow \beta$)

Bottom-up Parsing

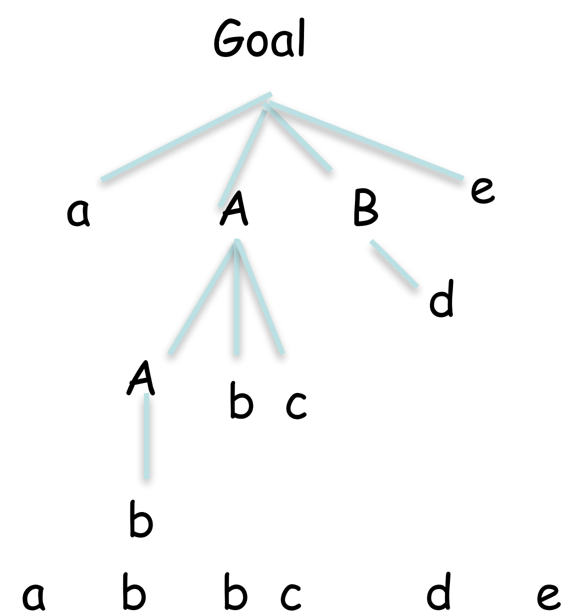
In terms of the parse tree, it works from leaves to root

- Nodes with no parent in a partial tree form its upper fringe (border)

Consider the grammar

The input string abbcde

0	Goal	→	a A B e
1	A	→	A b c
2		|	b
3	B	→	d



- Since each replacement of β with A shrinks the upper fringe, we call it a reduction.

(remember we are constructing a rightmost derivation)

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

0	Goal	→	a A B e
1	A	→	A b c
2		|	b
3	B	→	d

The input string abbcde

Sentential Form	Reduction Prod'n	Reduction Pos'n
abbcde	2	2
a A bcde	1	4
a A de	3	3
a A B e	0	4
Goal	—	—

The trick is scanning the input and finding the next reduction

The mechanism for doing this must be efficient

"Position" specifies where the right end of β occurs in the current sentential form.

Leftmost reductions for rightmost derivations

0	Goal	→	a A B e
1	A	→	A b c
2		|	b
3	B	→	d

Rightmost derivation
Goal
a A B e
a A de
a A bcde
abbcde

To reconstruct a Rightmost derivation bottom up we have to look for the leftmost substring that matches a right handside of a derivation!

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. We call this substring β an handle

An 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.

For this string is b not d !!

handles	A->β	k
abbcde	2	2
a A bcde	1	4
a A de	3	3
a A B e	0	4
Goal	—	—

A property of handles

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

handles	A->β	k
abbcde	2	2
a A bcde	1	4
a A de	3	3
a A B e	0	4
Goal	—	—

Example

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

Bottom up parsers handle either left-recursive or right-recursive grammars.

A simple left-recursive form of the classic expression grammar

Example

A simple left-recursive form of the classic expression grammar

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

derivation

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

Rightmost derivation of x - 2 * y

Example

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

Rightmost derivation of x - 2 * y

0	Goal	→	Expr
1	Expr	→	Expr + Term
2		|	Expr - Term
3		|	Term
4	Term	→	Term * Fact
5		|	Term / Facto
6		|	Factor
7	Factor	→	number
8		|	id
9		|	(Expr)

parse

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

Handles for rightmost derivation of x - 2 * y

Bottom-up Parsing (Abstract View)

A bottom-up parser repeatedly finds a handle $A \rightarrow \beta$ in the current right-sentential form and replaces β with A.

To construct a rightmost derivation

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

Apply the following conceptual 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

More on Handles

Bottom-up reduce parsers find a rightmost derivation in reverse order

- Rightmost derivation ⇒ rightmost NT expanded at each step in the derivation
- Processed in reverse ⇒ parser proceeds left to right

These statements are somewhat counter-intuitive

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 $\langle A \rightarrow \beta, \boldsymbol{k} \rangle$

This all follows from the definitions

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

Shift-reduce Parsing

To implement a bottom-up 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 (push)
- 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

Reduce consists in |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.

It uses a stack where we memorize terminal and nonterminal

Bottom-up Parser

What happens on an error?

A simple shift-reduce parser:

```
push $
token ← next_token( )
repeat until (top of stack = Goal and token = EOF)
    if the top of the stack is a handle A→β
        then      // reduce β to A
            pop |β| symbols off the stack
            push A onto the stack
        else if (token ≠ EOF)
            then // shift
                push token
                token ← next_token( )
        else    // need to shift, but out of input
      report 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…

We will fix this issue later.

Back to x - 2 * y

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

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

Expr is not a handle at this point because it does not occur in this point in a rightmost derivation of
id - num * id

While that statement sounds like oracular mysticism, we will see that the decision can be automated efficiently.

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

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

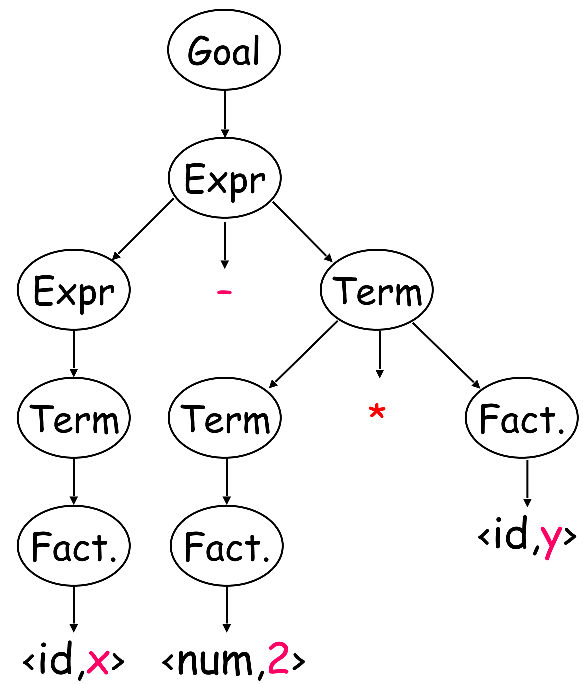
Stack	Input	Handle	Action
$	id - num * id	none	shift
$ id	- num * id	8,1	reduce 8
$ Factor	- num * id	6,1	reduce 6
$ Term	- num * id	3,1	reduce 3
$ Expr	- num * id	none	shift
$ Expr -	num * id	none	shift
$ Expr - num	* id	7,3	reduce 7
$ Expr - Factor	* id	6,3	reduce 6
$ Expr - Term	* id	none	shift
$ Expr - Term *	id	none	shift
$ Expr - Term * id		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	→	number
8		|	id
9		|	(Expr)

5 shifts + 9 reduces + 1 accept

Parse tree for x - 2 * y

Stack	Input	Action
$	id - num * id	shift
$ id	- num * id	reduce 8
$ Factor	- num * id	reduce 6
$ Term	- num * id	reduce 3
$ Expr	- num * id	shift
$ Expr -	num * id	shift
$ Expr - num	* id	reduce 7
$ Expr - Factor	* id	reduce 6
$ Expr - Term	* id	shift
$ Expr - Term *	id	shift
$ Expr - Term * id		reduce 8
$ Expr - Term * Factor		reduce 4
$ Expr - Term		reduce 2
$ Expr		reduce 0
$ Goal		accept



Corresponding Parse Tree

An Important Lesson about Handles

An handle must be a substring of a sentential form γ such that :

- It must match the right hand side β of some rule $A \rightarrow \beta$; and
- 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 an handle without generating lots of different derivations?

Answer: We use left context encoded in a "parser state" and a lookahead at the next word in the input. (Formally, 1 word beyond the handle.)

LR(1) Parsers

- LR(1) parsers use states to encode information on the left context and also use 1 word beyond the handle.

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

- Such information is encoded in a GOTO and ACTION tables

The actions are driven by the state and the lookhaed

LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- The class of grammars that these parsers recognize is called the set of LR(1) grammars

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

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

We can

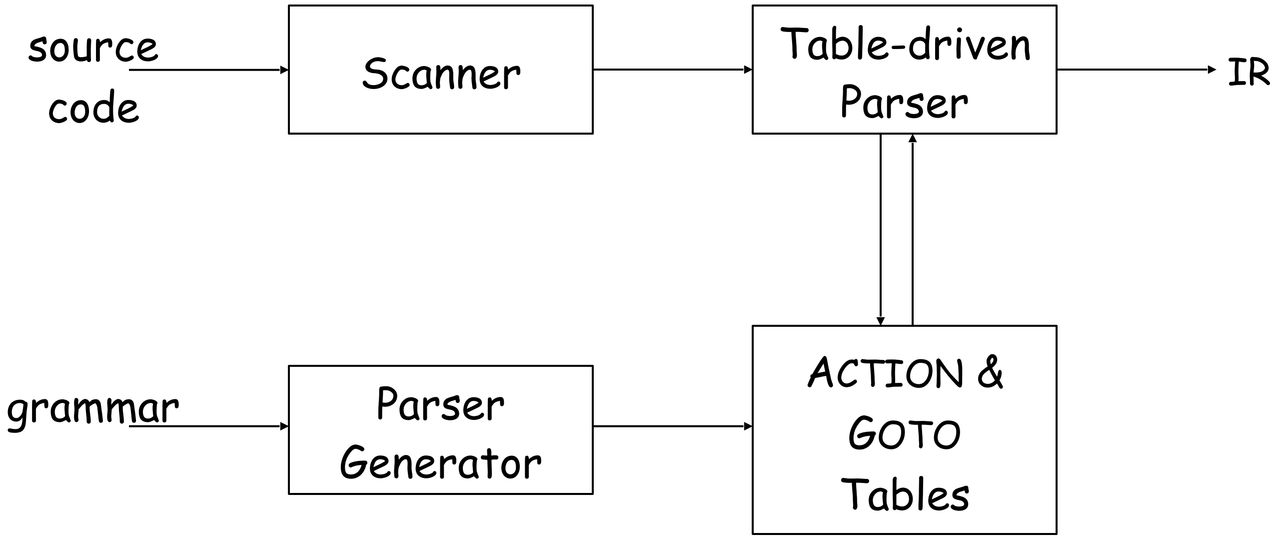
1. isolate the handle of each right-sentential form γ_i, and
2. determine the production by which to reduce,

going at most 1 symbol beyond the right end of the handle of γ_i

LR(1) means left-to-right scan of the input, rightmost derivation (in reverse), and 1 word of lookahead.

LR(1) Parsers

A table-driven LR(1) parser looks like

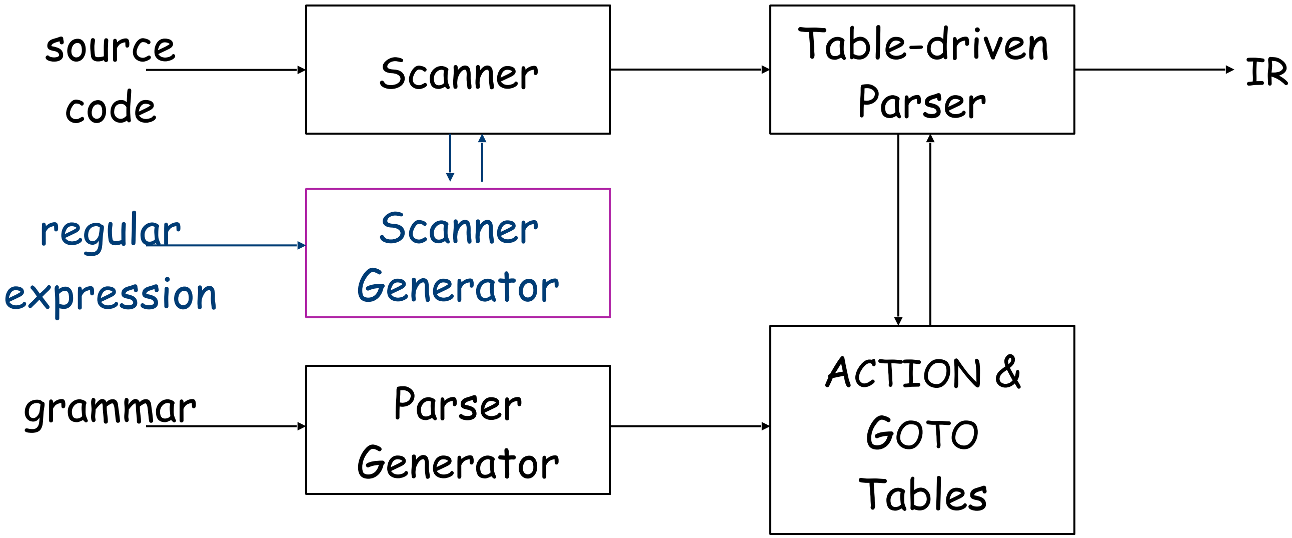


Tables can be built by hand

However, this is a perfect task to automate

LR(1) Parsers

A table-driven LR(1) parser looks like



Tables <u>can</u> be built by hand

However, this is a perfect task to automate

Just like automating construction of scanners …

It uses a stack where we memorize pairs of the form (T U NT, state)

LR(1) Skeleton Parser

```
stack.push($);
stack.push(s0);                    // initial state
token = scanner.next_token();
loop forever {
    s = stack.top(); // reads the top of the stack
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|);       // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);             // push A
        stack.push(GOTO[s,A]);     // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
            stack.push(token); stack.push(si);
            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 → action

GOTO: state x NT → state

- detects errors by failure of the other three cases

To make a parser for L(G), need the ACTION and GOTO tables

The grammar

1	Goal	$\rightarrow$	SheepNoise
2	SheepNoise	$\rightarrow$	SheepNoise baa
3		|	baa

For now assume we have the tables

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0

Example Parse 1

The string baa

Stack	Input	Action
$ s_0	baa EOF	

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

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0

Example Parse 1

The string baa

Stack	Input	Action
$ s_0	baa EOF	shift 2
$ s_0 baa s_2	EOF	

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

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0

Example Parse 1

The string baa

Stack	Input	Action
$ s_0	baa EOF	shift 2
$ s_0 baa s_2	EOF	reduce 3
$ s_0 SN s_1	EOF	

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

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0

Example Parse 1

The string baa

Stack	Input	Action
$ s_0	baa EOF	shift 2
$ s_0 baa s_2	EOF	reduce 3
$ s_0 SN s_1	EOF	accept

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

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0

Example Parse 2

The string baa baa

Stack	Input	Action
$ s_0	baa baa EOF	

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

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0

Example Parse 2

The string baa baa

Stack	Input	Action
$ s_0	baa baa EOF	shift 2
$ s_0 baa s_2	baa EOF	

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

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0

Example Parse 2

The string baa baa

Stack	Input	Action
$ s_0	baa baa EOF	shift 2
$ s_0 baa s_2	baa EOF	reduce 3
$ s_0 SN s_1	baa EOF	

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

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

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0

Example Parse 2

The string baa baa

Stack	Input	Action
$ s_0	baa baa EOF	shift 2
$ s_0 baa s_2	baa EOF	reduce 3
$ s_0 SN s_1	baa EOF	shift 3
$ s_0 SN s_1 baa s_3	EOF	

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

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0

Example Parse 2

The string baa baa

Stack	Input	Action
$ s_0	baa baa EOF	shift 2
$ s_0 baa s_2	baa EOF	reduce 3
$ s_0 SN s_1	baa EOF	shift 3
$ s_0 SN s_1 baa s_3	EOF	reduce 2
$ s_0 SN s_1	EOF	

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

Now, we accept

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0

Example Parse 2

The string <u>baa</u> <u>baa</u>

Stack	Input	Action
$ s_0	baa baa EOF	shift 2
$ s_0 baa s_2	baa EOF	reduce 3
$ s_0 SN s_1	baa EOF	shift 3
$ s_0 SN s_1 baa s_3	EOF	reduce 2
$ s_0 SN s_1	EOF	accept

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

ACTION Table		
State	EOF	baa
0	—	shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

GOTO Table	
State	SheepNoise
0	1
1	0
2	0
3	0