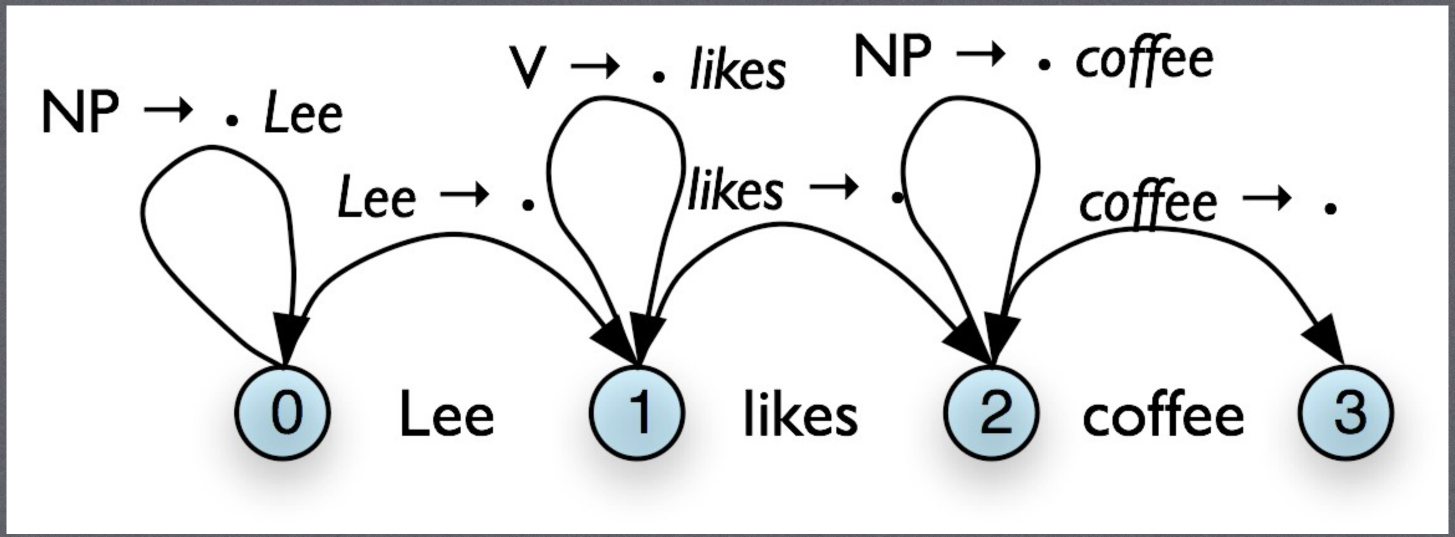


Compiling Techniques

Lecture 7: Bottom-Up Parsing

Christophe Dubach

Overview

- Bottom-Up Parsing
- Finding Reductions
- Handle Pruning
- Shift-Reduce Parsers

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" ⇒ may need to backtrack
  - Some grammars are backtrack-free (LL(1), 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

- The point of parsing is to construct a derivation
- A derivation consists of a series of rewrite steps
  - $S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow$ 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 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 Parsing

- A bottom-up parser builds a derivation by working from the input sentence back toward the start symbol S
  - $S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow$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, this is working 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*.

Finding Reductions

- Consider the simple grammar
- And the input string <u>abbcde</u>

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

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

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

Finding Reductions

- The parser must find a substring β of the tree's frontier that matches some production A → β that occurs as one step in the rightmost derivation
  - Informally, we call this substring β a *handle*
- Formally,
  - A handle of a right-sentential form γ is a pair <A→β,k> where A→β ∈ P and k is the position in γ of β's rightmost symbol.
  - If <A→β,k> 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
- ⇒ the parser doesn't need to scan past the handle (very far)

Finding Reductions

- Critical Insight: If G is unambiguous, then every right-sentential form has a *unique* handle.
- If we can find those handles, we can build a derivation !

Example

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

The expression grammar

Prod'n.	Sentential Form	Handle
—	*Goal*	—
1	*Expr*	1,1
3	*Expr* - *Term*	3,3
5	*Expr* - *Term* * *Factor*	5,5
9	*Expr* - *Term* * <id,y>	9,5
7	*Expr* - *Factor* * <id,y>	7,3
8	*Expr* - <num,2> * <id,y>	8,3
4	*Term* - <num,2> * <id,y>	4,1
7	*Factor* - <num,2> * <id,y>	7,1
9	<id,x> - <num,2> * <id,y>	9,1

Handles for rightmost derivation of x - 2 * y

Handle-pruning

- The process of discovering a handle & reducing it to the appropriate left-hand side is called *handle pruning*
- Handle pruning forms the basis for a bottom-up parsing method
- 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 simple algorithm
  - for $i \leftarrow n$ to 1 by –1
    - Find the handle $<A_i \rightarrow \beta_i , k_i >$ in γ_i
    - Replace β_i with A_i to generate γ_{i-1}
- This takes 2n steps

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→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
```

How do errors show up?

- failure to find a handle
- hitting EOF & needing to shift (final else clause)

Either generates an error

Example: x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	none	shift
$ id	– num * id		

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 the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Example: x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	none	shift
$ id	– num * id	9,1	red. 9
$ Factor	– num * id	7,1	red. 7
$ Term	– num * id	4,1	red. 4
$ Expr	– num * id		

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 the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Example: x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	none	shift
$ id	– num * id	9,1	red. 9
$ Factor	– num * id	7,1	red. 7
$ Term	– num * id	4,1	red. 4
$ Expr	– num * id	none	shift
$ Expr –	num * id	none	shift
$ Expr – num	* id		

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 the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Example: x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	none	shift
$ id	– num * id	9,1	red. 9
$ Factor	– num * id	7,1	red. 7
$ Term	– num * id	4,1	red. 4
$ Expr	– num * id	none	shift
$ Expr –	num * id	none	shift
$ Expr – num	* id	8,3	red. 8
$ Expr – Factor	* id	7,3	red. 7
$ Expr – Term	* id		

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 the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Example: x - 2 * y

Stack	Input	Handle	Action
$	id – num * id	none	shift
$ id	– num * id	9,1	red. 9
$ Factor	– num * id	7,1	red. 7
$ Term	– num * id	4,1	red. 4
$ Expr	– num * id	none	shift
$ Expr –	num * id	none	shift
$ Expr – num	* id	8,3	red. 8
$ Expr – Factor	* id	7,3	red. 7
$ Expr – Term	* id	none	shift
$ Expr – Term *	id	none	shift
$ Expr – Term * id			

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 the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Example: x - 2 * y

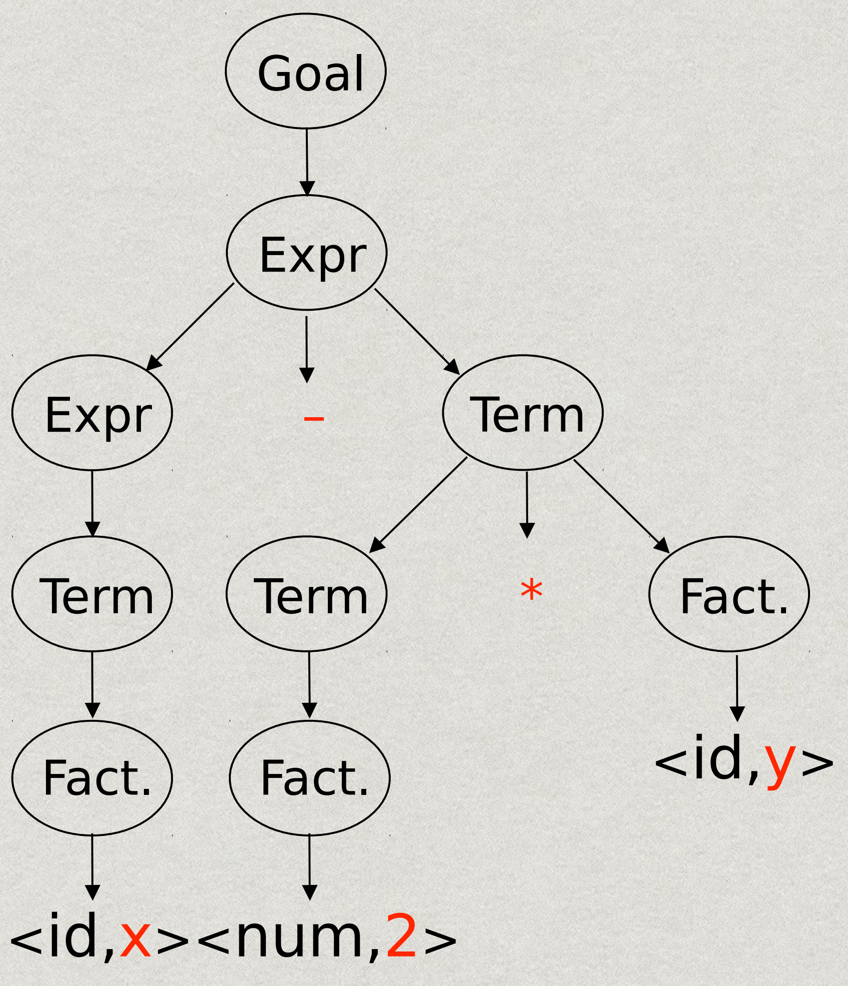
Stack	Input	Handle	Action
$	id – num * id	none	shift
$ id	– num * id	9,1	red. 9
$ Factor	– num * id	7,1	red. 7
$ Term	– num * id	4,1	red. 4
$ Expr	– num * id	none	shift
$ Expr –	num * id	none	shift
$ Expr – num	* id	8,3	red. 8
$ Expr – Factor	* id	7,3	red. 7
$ Expr – Term	* id	none	shift
$ Expr – Term *	id	none	shift
$ Expr – Term * id		9,5	red. 9
$ Expr – Term * Factor		5,5	red. 5
$ Expr – Term		3,3	red. 3
$ Expr		1,1	red. 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 the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Example: x - 2 * y

Stack	Input	Action
$	id – num * id	shift
$ id	– num * id	red. 9
$ Factor	– num * id	red. 7
$ Term	– num * id	red. 4
$ Expr	– num * id	shift
$ Expr –	num * id	shift
$ Expr – num	* id	red. 8
$ Expr – Factor	* id	red. 7
$ Expr – Term	* id	shift
$ Expr – Term *	id	shift
$ Expr – Term * id		red. 9
$ Expr – Term * Factor		red. 5
$ Expr – Term		red. 3
$ Expr		red. 1
$ Goal		accept



Shift-Reduce Parsing

- *Shift reduce parsers are easily built and easily understood*
- A shift-reduce parser has just *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
- If handle-finding requires state, put it in the stack ⇒ 2x work

> Handle finding is key
> - handle is on stack
> - finite set of handles
>
> ⇒ use a DFA !

Finding Handles

- Critical Question: How can we know when we have found a handle without generating lots of different derivations?
  - Answer: we use look ahead in the grammar along with tables produced as the result of analysing the grammar.
  - LR(1) parsers build a DFA that runs over the stack & finds them

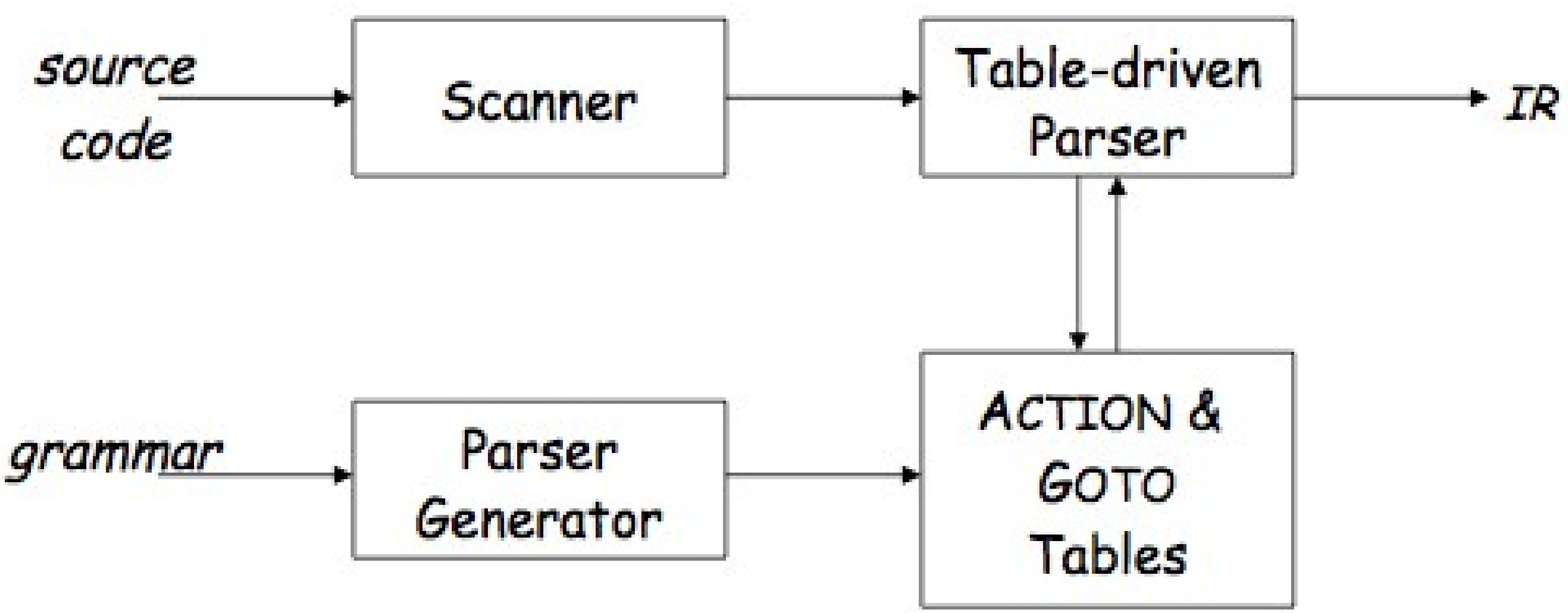
LR(1) Parsers

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition
- LR(1) parsers recognise languages that have an LR(1) grammar
- Informal definition:
  - A grammar is LR(1) if, given a rightmost derivation $S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow$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) Skeleton Parser

```
stack.push(INVALID); stack.push(s0);
not_found = true;
token = scanner.next_token();
do while (not_found) {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {
        stack.popnum(2*|β|); // pop 2*|β| symbols
        s = stack.top();
        stack.push(A);
        stack.push(GOTO[s,A]);
    }
    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 not_found = false;
    else report a syntax error and recover;
}
report success;
```

The skeleton parser

- uses ACTION & GOTO tables
- does |*words*| shifts
- does |derivation| reductions
- does 1 accept
- detects errors by failure of 3 other cases

LR(1) 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
```

The tables

ACTION		
State	**EOF**	**baa**
s0	-	shift s2
s1	accept	shift s3
s2	reduce 3	reduce 3
s3	reduce 2	reduce 2

GOTO	
State	**SN**
s0	s1
s1	
s2	
s3	

LR(1) 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

The tables

ACTION

State	**EOF**	**baa**
s0	-	shift s2
s1	accept	shift s3
s2	reduce 3	reduce 3
s3	reduce 2	reduce 2

GOTO

State	**SN**
s0	s1
s1	
s2	
s3	

Example: "baa"

STACK	INPUT	ACTION
s0	baa EOF	shift s2
s0 baa s2	EOF	reduce 3
s0 SN s1	EOF	accept

LR(1) 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

The tables

ACTION

State	**EOF**	**baa**
s0	-	shift s2
s1	accept	shift s3
s2	reduce 3	reduce 3
s3	reduce 2	reduce 2

GOTO

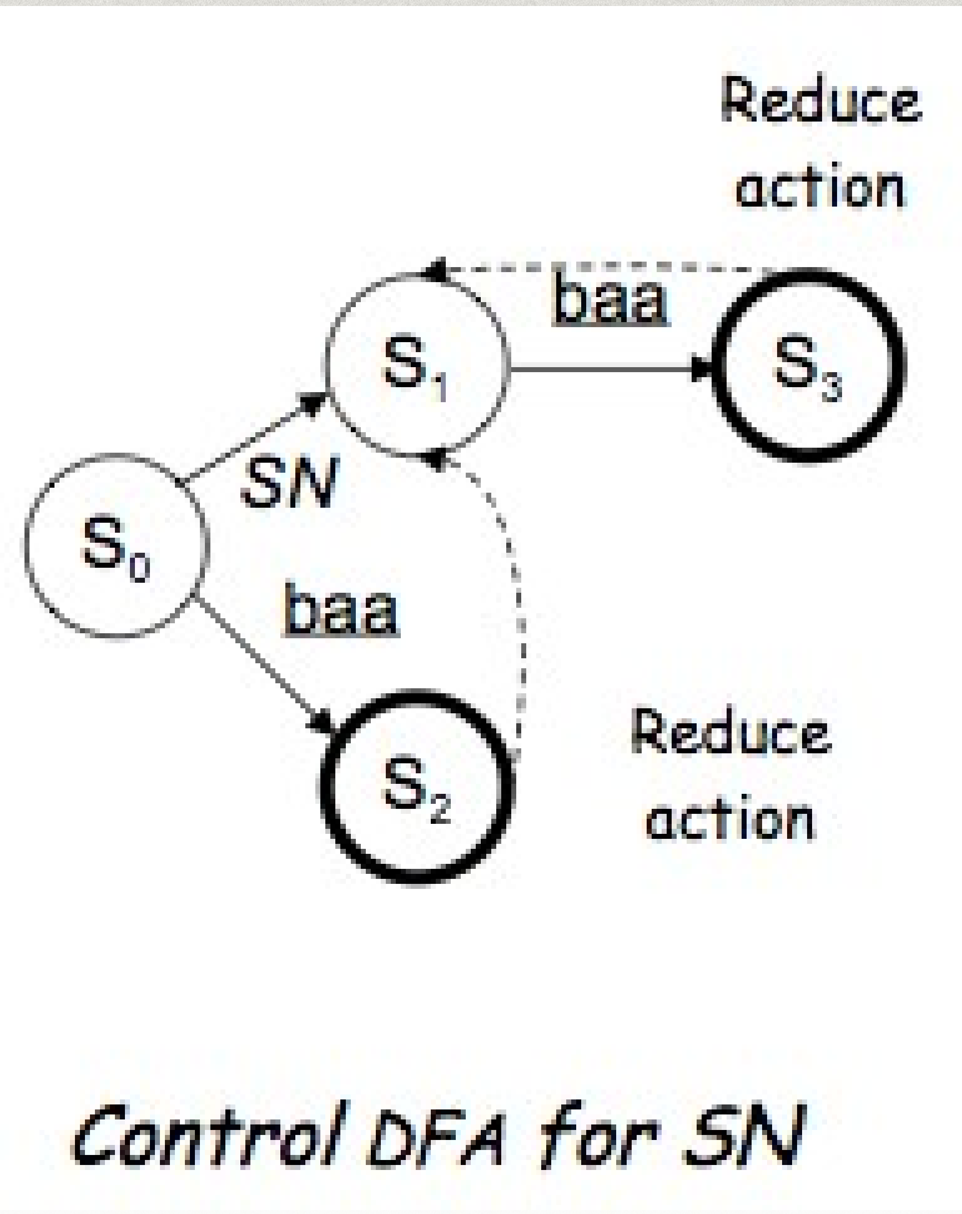
State	**SN**
s0	s1
s1	
s2	
s3	

Example: "baa baa"

STACK	INPUT	ACTION
s0	baa baa EOF	shift s2
s0 baa s2	baa EOF	reduce 3
s0 SN s1	baa EOF	shift s3
s0 SN s1 baa s3	EOF	reduce 2
s0 SN s1	EOF	accept

LR(1) Parse Tables

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



Example: "baa baa"

STACK	INPUT	ACTION
s0	baa baa EOF	shift s2
s0 baa s2	baa EOF	reduce 3
s0 SN s1	baa EOF	shift s3
s0 SN s1 baa s3	EOF	reduce 2
s0 SN s1	EOF	accept

ACTION			GOTO	
State	**EOF**	**baa**	**State**	**SN**
s0	-	shift s2	s0	s1
s1	accept	shift s3	s1	
s2	reduce 3	reduce 3	s2	
s3	reduce 2	reduce 2	s3	

Parse Tables

- The process of creating the parse tables can be automated
- More details in the book (EaC)

Beyond Syntax

```
fie(a,b,c,d)
   int a, b, c, d;
{ ... }

fee() {
   int f[3],g[0],
      h, i, j, k;
  char *p;
   fie(h,i,"ab",j, k);
   k = f * i + j;
   h = g[17];
   printf("<%s,%s>.\n",
      p,q);
   p = 10;
}
```

What is wrong with this program?

(let me count the ways ...)

- declared g[0], used g[17]
- wrong number of args to fie()
- "ab" is not an <u>int</u>
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are "deeper than syntax"

Preview

- Context-Sensitive Analysis