RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers
Syntax Analysis
Bottom-up Parsing

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Parsing
(Syntax Analysis)

EAC Chapters 3.4

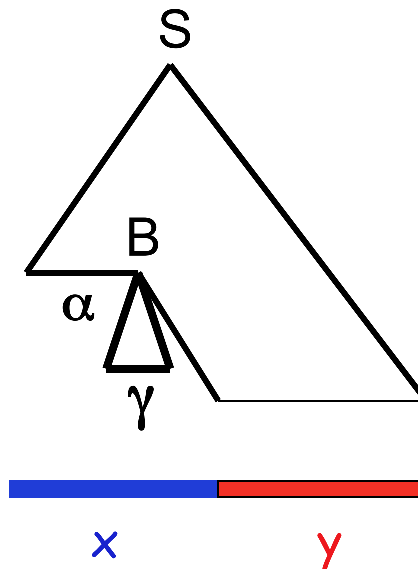
Parsing Techniques: Bottom-up parsers

LR(1), operator precedence

- L ← input: read left-to-right
- R ← construct rightmost derivation (backwards)
- (1) ← 1 input symbol lookahead

rule $B ::= \gamma$

$$S \Rightarrow^*_{rm} \alpha B y \Rightarrow_{rm} \alpha \gamma y \Rightarrow^*_{rm} x\, y$$



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 !

An Important Lesson about Handles

To be a handle, a substring of a sentential form γ must have two properties:

- It must match the right hand side β of some rule $A \rightarrow \beta$
- 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
  - There could be shift or reduce conflicts

Back to x - 2 * y

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

Back to 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. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to 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. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to 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. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to 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. Shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce

Back to 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

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 & reduce

Back to 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

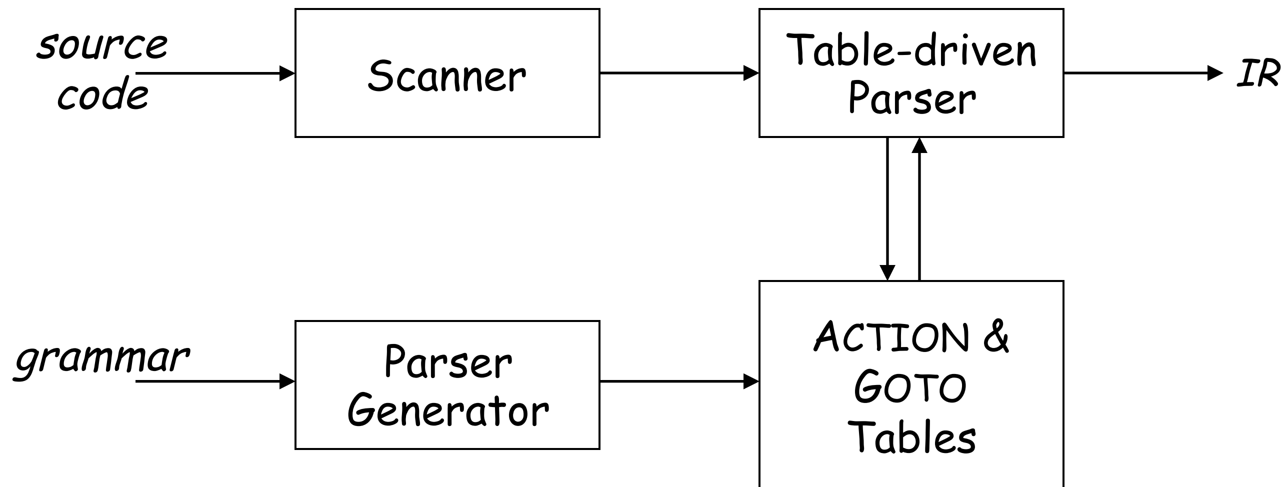
shift here

reduce here

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

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;
```

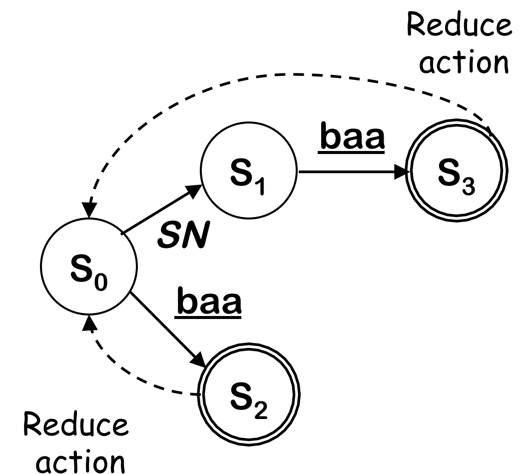
The skeleton parser

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

LR(1) Parsers

How does this LR(1) stuff work?

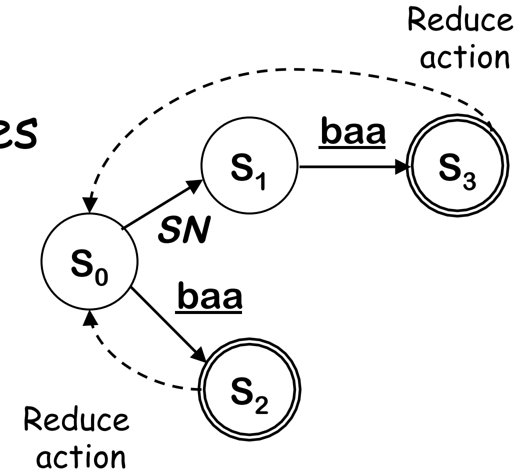
- Unambiguous grammar ⇒ unique rightmost derivation
- Keep upper fringe on a stack
  - → All active handles include top of stack (TOS)
  - → Shift inputs until TOS is right end of a handle
- Language of handles is regular (finite)
  - → Build a handle-recognizing DFA
  - → ACTION & GOTO tables encode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



Control DFA for SN

LR(1) Parsers (parse tables)

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



Control DFA for SN

The grammar

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

The tables

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

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

Building LR(1) Parsers

How do we generate the ACTION and GOTO tables?

- Use the grammar to build a model of the DFA
- Use the model to build ACTION & GOTO tables
- If construction succeeds, the grammar is LR(1)

The Big Picture

- Model the state of the parser
- Use two functions *goto(s, X)* and *closure(s)*
  - → *goto()* is analogous to *move()* in the subset construction
  - → *closure()* adds information to round out a state
- Build up the states and transition functions of the DFA
- Use this information to fill in the ACTION and GOTO tables

Terminal or non-terminal

LR(*k*) items

The LR(1) table construction algorithm uses LR(1) items to represent valid configurations of an LR(1) parser

An LR(*k*) item is a pair [*P*, δ], where

P is a production *A*→β with a • at some position in the *rhs*

δ is a lookahead string of length ≤ *k* (words or EOF)

The • in an item indicates the position of the top of the stack

LR(1):

[*A*→•βγ,a] means that the input seen so far is consistent with the use of *A* →βγ immediately after the symbol on top of the stack

[*A* →β•γ,a] means that the input seen so far is consistent with the use of *A* →βγ at this point in the parse, ***and*** that the parser has already recognized β.

[*A* →βγ•,a] means that the parser has seen βγ, ***and*** that a lookahead symbol of a is consistent with reducing to *A*.

LR(1) Items

The production $A \rightarrow \beta$, where $\beta = B_1 B_1 B_1$ with lookahead a, can give rise to 4 items

$[A \rightarrow \cdot B_1 B_2 B_3, \underline{a}]$, $[A \rightarrow B_1 \cdot B_2 B_3, \underline{a}]$, $[A \rightarrow B_1 B_2 \cdot B_3, \underline{a}]$, & $[A \rightarrow B_1 B_2 B_3 \cdot, \underline{a}]$

The set of LR(1) items for a grammar is finite

What's the point of all these lookahead symbols?

- Carry them along to choose the correct reduction, *if there is a choice*
- Lookaheads are bookkeeping, unless item has • at right end
  - → Has no direct use in $[A \rightarrow \beta \cdot \gamma, \underline{a}]$
  - → In $[A \rightarrow \beta \cdot, \underline{a}]$, a lookahead of a implies a reduction by $A \rightarrow \beta$
  - → For $\{ [A \rightarrow \beta \cdot, \underline{a}], [B \rightarrow \gamma \cdot \delta, \underline{b}] \}$, a ⇒ ***reduce*** to *A*; FIRST(δ) ⇒ ***shift***

⇒ Limited right context is enough to pick the actions (unique, i.e., deterministic choice)

LR(1) Table Construction

High-level overview

1 Build the canonical collection of sets of LR(1) Items, *I*

a Begin in an appropriate state, s_0

- Assume: $S' \rightarrow S$, and S' is unique start symbol that does not occur on any RHS of a production (extended CFG - ECFG)
- $[S' \rightarrow \cdot S, \underline{EOF}]$, along with any equivalent items
- Derive equivalent items as *closure*(s_0)

b Repeatedly compute, for each s_k, and each X, *goto*(s_k,X)

- If the set is not already in the collection, add it
- Record all the transitions created by *goto*( )

This eventually reaches a fixed point

2 Fill in the table from the collection of sets of LR(1) items

The canonical collection completely encodes the transition diagram for the handle-finding ***DFA***

Computing Closures

Closure(s) adds all the items implied by items already in *s*

- Any item $[A \rightarrow \beta \bullet B\delta, \underline{a}]$ implies $[B \rightarrow \bullet \tau, x]$ for each production with *B* on the *lhs*, and each $x \in \text{FIRST}(\delta\underline{a})$

The algorithm

```
Closure( s )
  while ( s is still changing )
    ∀ items [A → β•Bδ,a] ∈ s
      ∀ productions  B → τ ∈ P
        ∀ b  ∈ FIRST(δa) // δ might be ε
          if  [B → •τ,b] ∉ s
            then add [B→ •τ,b] to s
```

- Classic fixed-point method
- Halts because $s \subset$ ITEMS
- Worklist version is faster

Closure "fills out" a state

Computing Gotos

Goto(s,x) computes the state that the parser would reach if it recognized an *x* while in state *s*

- *Goto*({ [$A \rightarrow \beta \bullet X \delta, \underline{a}$] }, *X*) produces [$A \rightarrow \beta X \bullet \delta, \underline{a}$] *(easy part)*
- Should also includes *closure*([$A \rightarrow \beta X \bullet \delta, \underline{a}$]) *(fill out the state)*

The algorithm

Goto(*s*, *X*)
  new $\leftarrow \emptyset$
  $\forall$ *items* [$A \rightarrow \beta \cdot X \delta, \underline{a}$] $\in s$
    new $\leftarrow$ *new* $\cup$ [$A \rightarrow \beta X \cdot \delta, \underline{a}$]
  return closure(new)

- Not a fixed-point method!
- Straightforward computation
- Uses *closure*( )

Goto() moves forward

Building the Canonical Collection

Start from s_0 = *closure*([*S'*→*S*,EOF])

Repeatedly construct new states, until all are found

The algorithm

```
cc0 ← closure( [S' → •S, EOF])
CC  ← { cc0 }
while ( new sets are still being added to CC)
   for each unmarked set ccj ∈ CC
      mark ccj as processed
      for each x following a •  in an item in ccj
         temp ← goto(ccj, x)
         if temp ∉ CC
            then CC ← CC ∪ { temp }
         record transitions from ccj to temp on x
```

- Fixed-point computation (worklist version)
- Loop adds to *CC*
- $CC \subseteq 2^{ITEMS}$, so *CC* is finite

RUTGERS

The SheepNoise Grammar (*revisited*)

We will use this grammar again:

Goal → SheepNoise
SheepNoise → SheepNoise baa
| baa

Example From SheepNoise

Initial step builds the item [*Goal*→•*SheepNoise*,EOF] and takes its *closure( )*

Closure([*Goal*→•*SheepNoise*,EOF])

Item	*From*
[*Goal*→•*SheepNoise*,EOF]	Original item
[*SheepNoise*→•*SheepNoise* baa,EOF]	1, δa is EOF
[*SheepNoise*→ • baa,EOF]	1, δa is EOF
[*SheepNoise*→•*SheepNoise* baa,baa]	2, δa is baa EOF
[*SheepNoise*→ • baa,baa]	2, δa is baa EOF

NOTE: this is the left-recursive SheepNoise; EaC shows the right-recursive version.

So, cc_0 is

{ [*Goal*→ • *SheepNoise*,EOF], [*SheepNoise*→ • *SheepNoise* baa,EOF],
[*SheepNoise*→• baa,EOF], [*SheepNoise*→ • *SheepNoise* baa,baa],
[*SheepNoise*→ • baa,baa] }

Example from SheepNoise

cc_0 is { [*Goal*→ • *SheepNoise*,EOF], [*SheepNoise*→ • *SheepNoise* baa,EOF],
[*SheepNoise*→ • baa,EOF], [*SheepNoise*→ • *SheepNoise* baa,baa],
[*SheepNoise*→ • baa,baa] }

Goto(cc_0 , baa)

- Loop produces

Item	*From*
[*SheepNoise*→baa•, EOF]	Item 3 in cc_0
[*SheepNoise*→baa•, baa]	Item 5 in cc_0

- Closure adds nothing since • is at end of *rhs* in each item

In the construction, this produces cc_2

{ [*SheepNoise*→baa •, {EOF,baa}]}

New, but *obvious*, notation for two distinct items
[*SheepNoise*→baa •, EOF] &
[*SheepNoise*→baa •, baa]

Example from SheepNoise

Starts with cc_0

cc_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→• baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

Example from SheepNoise

Starts with cc_0

cc_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→• baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

Iteration 1 computes

cc_1 = *Goto*(cc_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

cc_2 = *Goto*(cc_0 , baa) = { [*SheepNoise*→ baa •, EOF],
[*SheepNoise*→ baa •, baa] }

Example from SheepNoise

Starts with cc_0

cc_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→• baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

Iteration 1 computes

cc_1 = *Goto*(cc_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

cc_2 = *Goto*(cc_0 , baa) = { [*SheepNoise*→ baa •, EOF],
[*SheepNoise*→ baa •, baa] }

Iteration 2 computes

cc_3 = *Goto*(cc_1 , baa) = { [*SheepNoise*→ *SheepNoise* baa •, EOF],
[*SheepNoise*→ *SheepNoise* baa •, baa] }

Example from SheepNoise

Starts with cc_0

cc_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF],
[*SheepNoise*→• baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa],
[*SheepNoise*→ • baa, baa] }

Iteration 1 computes

cc_1 = *Goto*(cc_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF],
[*SheepNoise*→ *SheepNoise* • baa, baa] }

cc_2 = *Goto*(cc_0 , baa) = { [*SheepNoise*→ baa •, EOF],
[*SheepNoise*→ baa •, baa] }

Nothing more to compute, since • is at the end of every item in cc_3.

Iteration 2 computes

cc_3 = *Goto*(cc_1 , baa) = { [*SheepNoise*→ *SheepNoise* baa •, EOF],
[*SheepNoise*→ *SheepNoise* baa •, baa] }

Example from SheepNoise

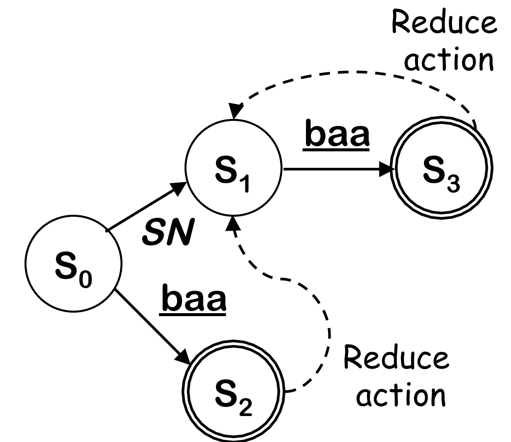
Canonical LR(1) Collection:

S_0 : { [*Goal*→ • *SheepNoise*, EOF], [*SheepNoise*→ • *SheepNoise* baa, EOF], [*SheepNoise*→• baa, EOF], [*SheepNoise*→ • *SheepNoise* baa, baa], [*SheepNoise*→ • baa, baa] }

Iteration 1 computes

S_1 = *Goto*(S_0 , *SheepNoise*) =
{ [*Goal*→ *SheepNoise* •, EOF], [*SheepNoise*→ *SheepNoise* • baa, EOF], [*SheepNoise*→ *SheepNoise* • baa, baa] }

S_2 = *Goto*(S_0 , baa) = { [*SheepNoise*→ baa •, EOF], [*SheepNoise*→ baa •, baa] }



Iteration 2 computes

S_3 = *Goto*(S_1 , baa) = { [*SheepNoise*→ *SheepNoise* baa •, EOF], [*SheepNoise*→ *SheepNoise* baa •, baa] }

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

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

Example Parse 1

The string "baa"

Stack	Input	Action
$ s_0	baa EOF	shift 2

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

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

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

Example Parse 1

The string "baa"

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

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

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

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

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

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

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

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

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

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

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

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

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

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

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

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	accept

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

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

GOTO	
State	*SheepNoise*
0	1
1	-
2	-
3	-

Next class

Wrap-up Syntax Analysis

Start Context-Sensitive Analysis

Read EaC: Chapters 4.1 – 4.3