

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

Shift Reduce Parsing

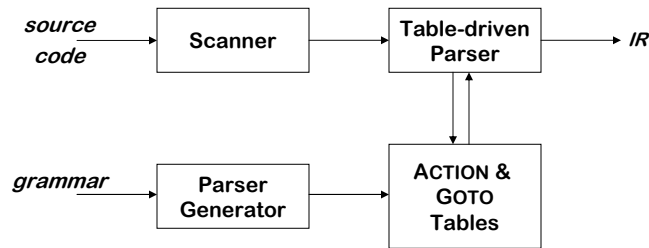
- a **right-sentential form** is any string that may occur in a legal rightmost derivation
- a **viable prefix** of a right-sentential form is any prefix that does not continue past the right end of its rightmost handle

Shift-reduce parsers

- **operator precedence** - define precedence between operands to guide reductions
- **LR(1)** --- construct DFA for recognizing viable prefix, storing lookahead information in DFA
- **SLR(1)** --- LR(0) + FOLLOW
 - > construct DFA for recognizing viable prefix, use *FOLLOW* to guide reductions
- **LALR(1)** --- construct DFA for recognizing viable prefix, propagating lookahead information in DFA

LR(1) Parsers

A table-driven LR(1) parser looks like



Tables *can* be built by hand
It is a perfect task to automate

CMSC430 Spring 2007

3

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

CMSC430 Spring 2007

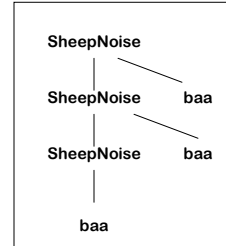
4

LR(1) Parsers (parse tables)

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

The grammar

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



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	<i>SheepNoise</i>
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 <u>baa</u> s ₂	EOF	reduce 3
\$ s ₀ <i>SN</i> s ₁	EOF	accept

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

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

GOTO	
State	<i>SheepNoise</i>
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	accept

1	Goal	→	Sheep Noise
2	Sheep Noise	→	Sheep Noise 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	0
2	0
3	0

CMSC430 Spring 2007

7

LR(0) items

- An LR(0) item is a string α , where α is a production from G with a \bullet at some position in the rhs
 - > The \bullet indicates how much of an item we have seen at a given state in the parse.
- $[A ::= \bullet X Y Z]$ indicates that the parser is looking for a string that can be derived from $X Y Z$
- $[A ::= X Y \bullet Z]$ indicates that the parser has seen a string derived from $X Y$ and is looking for one derivable from Z

LR(0) Items (no lookahead)

- The production $A ::= X Y Z$ generates 4 LR(0) items.
 - > $[A ::= \bullet X Y Z]$
 - > $[A ::= X \bullet Y Z]$
 - > $[A ::= X Y \bullet Z]$
 - > $[A ::= X Y Z \bullet]$

CMSC430 Spring 2007

8

LR(0) machine

Definitions

- *closure* of $[A ::= \alpha \bullet B \beta]$ contains itself and any items of form $[B ::= \bullet \omega]$, repeat for new items.
- *goto(X)* of $[A ::= \alpha \bullet X \beta]$ contains the closure of $[A ::= \alpha X \bullet \beta]$

LR(0) DFA construction

1. begin with closure of start symbol $[S ::= \bullet \alpha]$
2. for each state, calculate *goto(X)* for all grammar symbols X , generating states
3. repeat step 2 for all newly generated states

Properties

- states in the DFA are sets of LR(0) items
- states represent viable prefixes of productions
- to recognize viable prefixes of language, save state of current production on stack when reducing new nonterminal

CMSC430 Spring 2007

9

LR(0) items

The Grammar

P1	E	::=	T + E
P2			T
P3	T	::=	id

The Augmented Grammar

P0	S'	::=	E
P1	E	::=	T + E
P2			T
P3	T	::=	id

S0	[S' ::= • E]	goto(S1)
	[E ::= • T + E]	goto(S2)
	[E ::= • T]	goto(S2)
	[T ::= • id]	goto(S3)
S1	[S' ::= E •]	Reduce (and accept)
S2	[E ::= T • + E]	goto(S4)
	[E ::= T •]	Reduce
S3	[T ::= id •]	Reduce
S4	[E ::= T + • E]	goto(S5)
	[E ::= • T + E]	goto(S2)
	[E ::= • T]	goto(S2)
	[T ::= • id]	goto(S3)
S5	[E ::= T + E •]	Reduce

Oops - Shift-reduce Conflict.
What do you do here?

CMSC430 Spring 2007

10

LR(0) acceptance

- A grammar is LR(0) if the associated item lists contain no shift reduce conflicts (or reduce-reduce conflicts)
 - Using the tables for this grammar, try to parse $a+b$.
 - > For what states do you shift?
 - > For what states do you reduce?
 - > What happens?
 - But there are LR(0) grammars:
 - > Consider earlier grammar:
 Goal ::= SheepNoise
 SheepNoise ::= SheepNoise baa
 | baa
 - Build LR(0) item lists
- | | | |
|----|------------------------------------|-----------|
| S0 | [Goal ::= • SheepNoise] | goto (S1) |
| | [SheepNoise ::= • SheepNoise baa] | goto (S2) |
| | [SheepNoise ::= • baa] | goto(S3) |
| S1 | [Goal ::= SheepNoise •] | Accept |
| s2 | [SheepNoise ::= SheepNoise • baa] | goto(S4) |
| S3 | [SheepNoise ::= baa •] | Reduce |
| S4 | [SheepNoise ::= SheepNoise baa •] | Reduce |
- No conflicts in table, so grammar is LR(0) and any string can be parsed.

CMSC430 Spring 2007

11

SLR(1)

- Perhaps all is not lost. Consider LR(0) conflict previously in parsing $a+b$. If at the point of conflict we can then look one symbol ahead, perhaps we can resolve the problem.
- That is, state S2 was:

S2	[E ::= T • + E]	goto(S4)
	[E ::= T •]	Reduce
- For what inputs do we shift? For what inputs do we reduce?
 - > If you look at grammar you should see that shift is the desired action only when the next symbol is the + symbol. So a one character lookahead may be all that is needed to resolve the problem.
- Define an *inadequate state* as a state containing LR(0) items, which have either a shift-reduce or a shift-shift conflict.
- A grammar is **SLR(1)** if for each inadequate state S:
 - > If $[X ::= \alpha \cdot \beta]$ and $[Y ::= \omega \cdot]$ are in S then $\text{First}(\beta) \cap \text{Follow}(Y) = \emptyset$, and
 - > If $[X ::= \alpha \cdot]$ and $[Y ::= \omega \cdot]$ are in S then $\text{Follow}(X) \cap \text{Follow}(Y) = \emptyset$

CMSC430 Spring 2007

12

Redo Expression grammar – Grammar now SLR(1)

- The Grammar

P1 E ::= T + E
 P2 | T
 P3 T ::= id

- The Augmented Grammar

P0 S' ::= E
 P1 E ::= T + E
 P2 | T
 P3 T ::= id

S0
 [S' ::= • E] goto(S1)
 [E ::= • T + E] goto(S2)
 [E ::= • T] goto(S2)
 [T ::= • id] goto(S3)

S1
 [S' ::= E •] Reduce (and accept)

S2
 [E ::= T • + E] First(+) = +, goto(S4)
 [E ::= T •] Follow(E) = eof, Reduce

S3
 [T ::= id •] Reduce

S4
 [E ::= T + • E] goto(S5)
 [E ::= • T + E] goto(S2)
 [E ::= • T] goto(S2)
 [T ::= • id] goto(S3)

S5
 [E ::= T + E •] Reduce

CMSC430 Spring 2007

13

Computing FIRST Sets

Define FIRST as

- If $\alpha \Rightarrow^* \underline{a}\beta$, $\underline{a} \in T$, $\beta \in (T \cup NT)^*$, then $\underline{a} \in \text{FIRST}(\alpha)$
- If $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$

Note: if $\alpha = X\beta$, $\text{FIRST}(\alpha) = \text{FIRST}(X)$

To compute FIRST

- Use a fixed-point method
- $\text{FIRST}(A) \in 2^{(T \cup \epsilon)}$
- Loop is monotonic

⇒ Algorithm halts

For *SheepNoise*:

$\text{FIRST}(\textit{Goal}) = \{ \underline{\textit{baa}} \}$
 $\text{FIRST}(\textit{SM}) = \{ \underline{\textit{baa}} \}$
 $\text{FIRST}(\underline{\textit{baa}}) = \{ \underline{\textit{baa}} \}$

```

for each  $x \in T$ ,  $\text{FIRST}(x) \leftarrow \{x\}$ 
for each  $A \in NT$ ,  $\text{FIRST}(A) \leftarrow \emptyset$ 
while (FIRST sets are still changing)
  for each  $p \in P$ , of the form  $A \rightarrow \beta$ ,
    if  $\beta$  is  $\epsilon$  then
       $\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \{ \epsilon \}$ 
    else if  $\beta$  is  $B_1 B_2 \dots B_k$  then begin
       $\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup ( \text{FIRST}(B_1) - \{ \epsilon \} )$ 
      for  $i \leftarrow 1$  to  $k-1$  by 1 while  $\epsilon \in \text{FIRST}(B_i)$ 
         $\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup ( \text{FIRST}(B_{i+1}) - \{ \epsilon \} )$ 
      if  $i = k-1$  and  $\epsilon \in \text{FIRST}(B_k)$ 
        then  $\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \{ \epsilon \}$ 
    end
end
    
```

CMSC430 Spring 2007

14

LR(1) items

We can get more powerful parsers by keeping track of lookahead information in the states of the *LR* parser.

- An *LR(k)* item is a pair $[\alpha, \beta]$ where
 - > α is a production from G with a \bullet at some position in the *rhs*
 - > β is a lookahead string containing k symbols (terminals or *eof*)

LR(1) items

- example: $[A ::= X \bullet Y Z, a]$
- several *LR(1)* items may have the same *core* {i.e., *LR(0) item lists*}
 - > $[A ::= X \bullet Y Z, a]$
 - > $[A ::= X \bullet Y Z, b]$
- we represent this as $[A ::= X \bullet Y Z, ab]$

LR(1) Lookahead

What's the point of all these lookahead symbols?

- carry them along to allow us to choose correct reduction when there is any choice
- lookaheads are bookkeeping, unless item has \bullet at right end.
 - > in $[A ::= X \bullet Y Z, a]$, a has no direct use
 - > in $[A ::= X Y Z \bullet, a]$, a is useful
- allows use of (non-invertible) grammars where productions have the same *rhs*

The point

- For $[A ::= \alpha \bullet, a]$ and $[B ::= \alpha \bullet, b]$,
- we can decide between reducing to A and to B by looking at limited right context!

LR(1) machine

Definitions

- **closure** of $[A ::= \alpha \cdot B \beta, a]$ contains itself and any items of form $[B ::= \cdot \omega, \text{First}(\beta a)]$, repeat for new items.
- **goto(X)** of $[A ::= \alpha \cdot X \beta, a]$ contains the closure of $[A ::= \alpha X \cdot \beta, a]$.

LR(1) DFA construction

- begin with closure of start symbol $[S ::= \cdot \alpha, \text{eof}]$
- for each state, calculate $\text{goto}(X)$ for all grammar symbols X , generating states
- repeat step 2 for all newly generated states

Properties

- $[A ::= X \cdot YZ, \alpha] \rightarrow$ have recognized X & YZ would be valid
- $[A ::= X \cdot YZ, \alpha] \rightarrow [Y ::= \cdot \beta, \omega]$ & $[Y ::= \cdot \delta, \gamma]$ are also valid, where $\omega, \gamma \in \text{FIRST}(Z \alpha)$ recognizing Y takes parser to $[A ::= XY \cdot Z, \alpha]$

Example: LR(1) states

S0	$[S' ::= \cdot E, \$]$ $[E ::= \cdot T + E, \$]$ $[E ::= \cdot T, \$]$ $[T ::= \cdot \text{id}, +]$ $[T ::= \cdot \text{id}, \$]$	goto(S1) goto(S2) FIRST($\epsilon \$$) = \$ goto(S2) FIRST($\epsilon \$$) = \$ goto(S3) FIRST($+ E \$$) = + goto(S3) FIRST($\epsilon \$$) = \$
S1	$[S' ::= E \cdot, \$]$	Reduce
S2	$[E ::= T \cdot + E, \$]$ $[E ::= T \cdot, \$]$	If next=+ goto(S4) if next=\$ Reduce
S3	$[T ::= \text{id} \cdot, +]$ $[T ::= \text{id} \cdot, \$]$	Reduce Reduce
S4	$[E ::= T + \cdot E, \$]$ $[E ::= \cdot T + E, \$]$ $[E ::= \cdot T, \$]$ $[T ::= \cdot \text{id}, +]$ $[T ::= \cdot \text{id}, \$]$	goto(S5) goto(S2) FIRST($\epsilon \$$) = \$ goto(S2) FIRST($\epsilon \$$) = \$ goto(S3) FIRST($+ E \$$) = + goto(S3) FIRST($\epsilon \$$) = \$
S5	$[E ::= T + E \cdot, \$]$	Reduce

LALR(1) parsers

Problem

- LR(1) parsers are powerful, but have many more states than LR(0) (approximately x 10 for Pascal)
- larger state tables longer to construct, run

LALR(1) parsers

- define the *core* of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.
 - > example of two sets of LR(1) items with same core:
[A ::= $\alpha \cdot \beta$, a], [A ::= $\alpha \cdot \beta$, b], and [A ::= $\alpha \cdot \beta$, c], [A ::= $\alpha \cdot \beta$, d]
- if two sets of LR(1) items, I1 and I2 have the same core, we can merge the states that represent them in the ACTION and GOTO tables
- almost as powerful as LR(1), same size as LR(0)

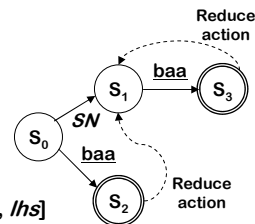
CMSC430 Spring 2007

19

LR(1) Parsers – Formal definitions – Repeat of previous slides

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[state at TOS (after pop), *lhs*]
 - > For *SN*, this takes the DFA to s_1



Control DFA for SN

CMSC430 Spring 2007

20

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

An LR(k) item is a pair $[P, \delta]$, where

P is a production $A \rightarrow \beta$ with a \bullet at some position in the *rhs*

δ is a lookahead string of length $\leq k$ (words or EOF)

The \bullet in an item indicates the position of the top of the stack

$[A \rightarrow \bullet \beta \gamma, \underline{a}]$ means that the input seen so far is consistent with the use of $A \rightarrow \beta \gamma$ immediately after the symbol on top of the stack

$[A \rightarrow \beta \bullet \gamma, \underline{a}]$ means that the input seen so far is consistent with the use of $A \rightarrow \beta \gamma$ at this point in the parse, *and* that the parser has already recognized β .

$[A \rightarrow \beta \gamma \bullet, \underline{a}]$ means that the parser has seen $\beta \gamma$, *and* that a lookahead symbol of \underline{a} is consistent with reducing to A .

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

LR(1) Items

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

$[A \rightarrow \bullet B_1 B_1 B_1, \underline{a}]$, $[A \rightarrow B_1 \bullet B_1 B_1, \underline{a}]$, $[A \rightarrow B_1 B_1 \bullet B_1, \underline{a}]$, & $[A \rightarrow B_1 B_1 B_1 \bullet, \underline{a}]$

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 correct reduction *(if a choice occurs)*
 - Lookaheads are bookkeeping, unless item has \bullet at right end
 - > Has no direct use in $[A \rightarrow \beta \bullet \gamma, \underline{a}]$
 - > In $[A \rightarrow \beta \bullet, \underline{a}]$, a lookahead of \underline{a} implies a reduction by $A \rightarrow \beta$
 - > For $\{ [A \rightarrow \beta \bullet, \underline{a}], [B \rightarrow \gamma \bullet \delta, \underline{b}] \}$, $\underline{a} \Rightarrow$ reduce to A ; $\text{FIRST}(\delta) \Rightarrow$ shift
- \Rightarrow Limited right context is enough to pick the actions

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
 - $\rightarrow [S' \rightarrow \bullet S, \text{EOF}]$, along with any equivalent items
 - \rightarrow Derive equivalent items as $\text{closure}(i_0)$
- b Repeatedly compute, for each $s_{i'}$ and each X , $\text{goto}(s_{i'}, X)$
 - \rightarrow If the set is not already in the collection, add it
 - \rightarrow Record all the transitions created by $\text{goto}()$

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

Back to Finding Handles

Revisiting an issue from last class

Parser in a state where the stack (the fringe) was

$Expr = Term$

With lookahead of $*$

How did it choose to expand $Term$ rather than reduce to $Expr$?

- *Lookahead* symbol is the key
- With lookahead of $+$ or $=$, parser should reduce to $Expr$
- With lookahead of $*$ or $/$, parser should shift
- Parser uses lookahead to decide
- All this context from the grammar is encoded in the handle recognizing mechanism

Remember this slide?

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

Computing Closures

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

- Any item $[A \rightarrow \beta \bullet B \delta, a]$ implies $[B \rightarrow \bullet \tau, x]$ for each production with *B* on the *lhs*, and each $x \in \text{FIRST}(\delta a)$
- Since $\beta B \delta$ is valid, any way to derive $\beta B \delta$ is valid, too

The algorithm

```

Closure(s)
while (s is still changing)
  ∀ items  $[A \rightarrow \beta \bullet B \delta, a] \in s$ 
  ∀ productions  $B \rightarrow \tau \in P$ 
  ∀  $b \in \text{FIRST}(\delta a)$  //  $\delta$  might be  $\epsilon$ 
  if  $[B \rightarrow \bullet \tau, b] \notin s$ 
  then add  $[B \rightarrow \bullet \tau, b]$  to s
    
```

- Classic fixed-point algorithm
- Halts because $s \subset \text{ITEMS}$
- Worklist version is faster

Closure "fills out" a state

Example From SheepNoise

Initial step builds the item $[Goal \rightarrow \bullet SheepNoise, EOF]$
and takes its *closure()*

Closure($[Goal \rightarrow \bullet SheepNoise, EOF]$)

Item	From
$[Goal \rightarrow \bullet SheepNoise, EOF]$	Original item
$[SheepNoise \rightarrow \bullet SheepNoise \underline{baa}, EOF]$	1, δa is EOF
$[SheepNoise \rightarrow \bullet \underline{baa}, EOF]$	1, δa is EOF
$[SheepNoise \rightarrow \bullet SheepNoise \underline{baa}, \underline{baa}]$	2, δa is <u>baa</u> EOF
$[SheepNoise \rightarrow \bullet \underline{baa}, \underline{baa}]$	2, δa is <u>baa</u> EOF

So, S_0 is

{ $[Goal \rightarrow \bullet SheepNoise, EOF]$, $[SheepNoise \rightarrow \bullet SheepNoise \underline{baa}, EOF]$,
 $[SheepNoise \rightarrow \bullet \underline{baa}, EOF]$, $[SheepNoise \rightarrow \bullet SheepNoise \underline{baa}, \underline{baa}]$,
 $[SheepNoise \rightarrow \bullet \underline{baa}, \underline{baa}]$ }

Computing Gotos

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

- $Goto(\{ [A \rightarrow \beta \cdot X \delta, \underline{a}] \}, X)$ produces $[A \rightarrow \beta X \delta, \underline{a}]$ (obviously)
- It also includes $closure([A \rightarrow \beta X \delta, \underline{a}])$ to fill out the state

The algorithm

```
Goto(s, X)
new ← ∅
∀ items [A → β · X δ, a] ∈ s
  new ← new ∪ [A → β X δ, a]
return closure(new)
```

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

$Goto()$ advances the parse

Example from SheepNoise

S_0 is $\{ [Goal \rightarrow \cdot SheepNoise, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, EOF], [SheepNoise \rightarrow \cdot \underline{baa}, EOF], [SheepNoise \rightarrow \cdot SheepNoise \underline{baa}, \underline{baa}], [SheepNoise \rightarrow \cdot \underline{baa}, \underline{baa}] \}$

$Goto(S_0, \underline{baa})$

- Loop produces

Item	From
$[SheepNoise \rightarrow \underline{baa} \cdot, EOF]$	Item 3 in s_0
$[SheepNoise \rightarrow \underline{baa} \cdot, \underline{baa}]$	Item 5 in s_0

- Closure adds nothing since \cdot is at end of rhs in each item

In the construction, this produces s_2
 $\{ [SheepNoise \rightarrow \underline{baa} \cdot, \{EOF, \underline{baa}\}] \}$

New, but obvious, notation for two distinct items
 $[SheepNoise \rightarrow \underline{baa} \cdot, EOF]$ &
 $[SheepNoise \rightarrow \underline{baa} \cdot, \underline{baa}]$

Example from SheepNoise

$$S_0 : \{ [Goal \rightarrow \bullet SheepNoise, EOF], [SheepNoise \rightarrow \bullet SheepNoise \text{ baa}, EOF], \\ [SheepNoise \rightarrow \bullet \text{baa}, EOF], [SheepNoise \rightarrow \bullet SheepNoise \text{ baa}, \text{baa}], \\ [SheepNoise \rightarrow \bullet \text{baa}, \text{baa}] \}$$

$$S_1 = Goto(S_0, SheepNoise) = \\ \{ [Goal \rightarrow SheepNoise \bullet, EOF], [SheepNoise \rightarrow SheepNoise \bullet \text{baa}, EOF], \\ [SheepNoise \rightarrow SheepNoise \bullet \text{baa}, \text{baa}] \}$$

$$S_2 = Goto(S_0, \text{baa}) = \{ [SheepNoise \rightarrow \text{baa} \bullet, EOF], [SheepNoise \rightarrow \text{baa} \bullet, \text{baa}] \}$$

$$S_3 = Goto(S_1, \text{baa}) = \{ [SheepNoise \rightarrow SheepNoise \text{ baa} \bullet, EOF], \\ [SheepNoise \rightarrow SheepNoise \text{ baa} \bullet, \text{baa}] \}$$

CMSC430 Spring 2007

31

Building the Canonical Collection

Start from $s_0 = closure([S' \rightarrow S, EOF])$

Repeatedly construct new states, until all are found

The algorithm

```

s0 ← closure([S' → S, EOF])
S ← { s0 }
k ← 1
while ( S is still changing )
  ∀ sj ∈ S and ∀ x ∈ ( T ∪ NT )
    sk ← goto(sj, x)
    record sj → sk on x
    if sk ∉ S then
      S ← S ∪ sk
      k ← k + 1
  
```

- Fixed-point computation
- Loop adds to S
- $S \subseteq 2^{ITEMS}$, so S is finite
- Worklist version is faster

CMSC430 Spring 2007

32

Example (grammar & sets)

Simplified, right recursive expression grammar

$Goal \rightarrow Expr$
 $Expr \rightarrow Term - Expr$
 $Expr \rightarrow Term$
 $Term \rightarrow Factor * Term$
 $Term \rightarrow Factor$
 $Factor \rightarrow \underline{ident}$

Symbol	FIRST
Goal	{ <u>ident</u> }
Expr	{ <u>ident</u> }
Term	{ <u>ident</u> }
Factor	{ <u>ident</u> }
-	{ - }
*	{ * }
<u>ident</u>	{ <u>ident</u> }

Example (building the collection)

Initialization Step

$s_0 \leftarrow \text{closure}(\{ [Goal \rightarrow \bullet Expr, EOF] \})$
 $\{ [Goal \rightarrow \bullet Expr, EOF], [Expr \rightarrow \bullet Term - Expr, EOF], [Expr \rightarrow \bullet Term, EOF],$
 $[Term \rightarrow \bullet Factor * Term, EOF], [Term \rightarrow \bullet Factor * Term, -],$
 $[Term \rightarrow \bullet Factor, EOF], [Term \rightarrow \bullet Factor, -],$
 $[Factor \rightarrow \bullet \underline{ident}, EOF], [Factor \rightarrow \bullet \underline{ident}, -], [Factor \rightarrow \bullet \underline{ident}, *] \}$
 $S \leftarrow \{s_0\}$

Example (building the collection)

Iteration 1

$s_1 \leftarrow goto(s_0, Expr)$
 $s_2 \leftarrow goto(s_0, Term)$
 $s_3 \leftarrow goto(s_0, Factor)$
 $s_4 \leftarrow goto(s_0, \underline{ident})$

Iteration 2

$s_5 \leftarrow goto(s_2, -)$
 $s_6 \leftarrow goto(s_3, *)$

Iteration 3

$s_7 \leftarrow goto(s_5, Expr)$
 $s_8 \leftarrow goto(s_6, Term)$

Example

(Summary)

$S_0 : \{ [Goal \rightarrow \bullet Expr, EOF], [Expr \rightarrow \bullet Term - Expr, EOF], [Expr \rightarrow \bullet Term, EOF], [Term \rightarrow \bullet Factor * Term, EOF], [Term \rightarrow \bullet Factor * Term, -], [Term \rightarrow \bullet Factor, EOF], [Term \rightarrow \bullet Factor, -], [Factor \rightarrow \bullet \underline{ident}, EOF], [Factor \rightarrow \bullet \underline{ident}, -], [Factor \rightarrow \bullet \underline{ident}, *] \}$

$S_1 : \{ [Goal \rightarrow Expr \bullet, EOF] \}$

$S_2 : \{ [Expr \rightarrow Term \bullet - Expr, EOF], [Expr \rightarrow Term \bullet, EOF] \}$

$S_3 : \{ [Term \rightarrow Factor \bullet * Term, EOF], [Term \rightarrow Factor \bullet * Term, -], [Term \rightarrow Factor \bullet, EOF], [Term \rightarrow Factor \bullet, -] \}$

$S_4 : \{ [Factor \rightarrow \underline{ident} \bullet, EOF], [Factor \rightarrow \underline{ident} \bullet, -], [Factor \rightarrow \underline{ident} \bullet, *] \}$

$S_5 : \{ [Expr \rightarrow Term - \bullet Expr, EOF], [Expr \rightarrow \bullet Term - Expr, EOF], [Expr \rightarrow \bullet Term, EOF], [Term \rightarrow \bullet Factor * Term, -], [Term \rightarrow \bullet Factor, -], [Term \rightarrow \bullet Factor * Term, EOF], [Term \rightarrow \bullet Factor, EOF], [Factor \rightarrow \bullet \underline{ident}, *], [Factor \rightarrow \bullet \underline{ident}, -], [Factor \rightarrow \bullet \underline{ident}, EOF] \}$

Example

(Summary)

$S_6 : \{ [Term \rightarrow Factor * Term, EOF], [Term \rightarrow Factor * Term, -],$
 $[Term \rightarrow \bullet Factor * Term, EOF], [Term \rightarrow \bullet Factor * Term, -],$
 $[Term \rightarrow \bullet Factor, EOF], [Term \rightarrow \bullet Factor, -],$
 $[Factor \rightarrow \bullet \underline{ident}, EOF], [Factor \rightarrow \bullet \underline{ident}, -], [Factor \rightarrow \bullet \underline{ident}, *] \}$

$S_7 : \{ [Expr \rightarrow Term - Expr \bullet, EOF] \}$

$S_8 : \{ [Term \rightarrow Factor * Term \bullet, EOF], [Term \rightarrow Factor * Term \bullet, -] \}$

Example

(Summary)

The Goto Relationship (from the construction)

State	Expr	Term	Factor	-	*	<u>ident</u>
0	1	2	3			4
1						
2				5		
3					6	
4						
5	7	2	3			4
6		8	3			4
7						
8						

Filling in the ACTION and GOTO Tables

The algorithm

```

 $\forall$  set  $s_x \in S$ 
 $\forall$  item  $i \in s_x$ 
  if  $i$  is  $[A \rightarrow \beta \cdot a \delta, b]$  and  $\text{goto}(s_x, a) = s_k, a \in T$ 
    then ACTION[x,a]  $\leftarrow$  "shift k"
  else if  $i$  is  $[S' \rightarrow S \cdot, \text{EOF}]$ 
    then ACTION[x,a]  $\leftarrow$  "accept"
  else if  $i$  is  $[A \rightarrow \beta \cdot, a]$ 
    then ACTION[x,a]  $\leftarrow$  "reduce  $A \rightarrow \beta$ "
 $\forall n \in NT$ 
  if  $\text{goto}(s_x, n) = s_k$ 
    then GOTO[x,n]  $\leftarrow$  k
  
```

x is the state number

Many items generate no table entry

> Closure() instantiates FIRST(X) directly for $[A \rightarrow \beta \cdot X \delta, a]$

Example

(Filling in the tables)

The algorithm produces the following table

	ACTION				GOTO		
	<u>Ident</u>	-	*	EOF	<i>Expr</i>	<i>Term</i>	<i>Factor</i>
0	s 4				1	2	3
1				acc			
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			

END OF DUPLICATE DEFINITION OF LR(1)

What can go wrong?

What if set s contains $[A \rightarrow \beta \cdot a \gamma, b]$ and $[B \rightarrow \beta \cdot, a]$?

- First item generates “shift”, second generates “reduce”
- Both define ACTION[s,a] — cannot do both actions
- This is a fundamental ambiguity, called a *shift/reduce error*
- Modify the grammar to eliminate it *(if-then-else)*
- Shifting will often resolve it correctly

What if set s contains $[A \rightarrow \gamma^*, a]$ and $[B \rightarrow \gamma^*, a]$?

- Each generates “reduce”, but with a different production
- Both define ACTION[s,a] — cannot do both reductions
- This is a fundamental ambiguity, called a *reduce/reduce conflict*
- Modify the grammar to eliminate it *(PLI's overloading of (...))*

In either case, the grammar is not LR(1)

Shrinking the Tables

Three options:

- Combine terminals such as number & identifier, + & -, * & /
 - > Directly removes a column, may remove a row
 - > For expression grammar, 198 (vs. 384) table entries
- Combine rows or columns
 - > Implement identical rows once & remap states
 - > Requires extra indirection on each lookup
 - > Use separate mapping for ACTION & for GOTO
- Use another construction algorithm
 - > Both LALR(1) and SLR(1) produce smaller tables
 - > Implementations are readily available

LR(k) versus LL(k) (Top-down Recursive Descent)

Finding Reductions

LR(k) ⇒ Each reduction in the parse is detectable with

- 1 the complete left context,**
- 2 the reducible phrase, itself, and**
- 3 the k terminal symbols to its right**

3 LL(k) ⇒ Parser must select the reduction based on

- 1 The complete left context**
- 2 The next k terminals**

2 Thus, LR(k) examines more context

2 “... in practice, programming languages do not actually seem to fall in the gap between LL(1) languages and deterministic languages” J.J. Horning, “LR Grammars and Analysers”, in *Compiler Construction, An Advanced Course*, Springer-Verlag, 1976

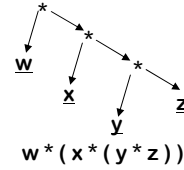
•Summary

	<i>Advantages</i>	<i>Disadvantages</i>
Top-down recursive descent	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

Left Recursion versus Right Recursion

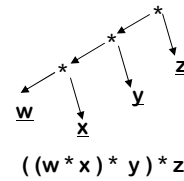
Right recursion

- Required for termination in top-down parsers
- Uses (on average) more stack space
- Produces right-associative operators



Left recursion

- Works fine in bottom-up parsers
- Limits required stack space
- Produces left-associative operators



Rule of thumb

- Left recursion for bottom-up parsers
- Right recursion for top-down parsers

Associativity

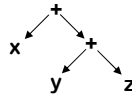
What difference does it make?

- Can change answers in floating-point arithmetic
- Exposes a different set of common subexpressions

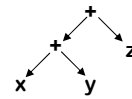
Consider $x+y+z$



Ideal operator



Left association



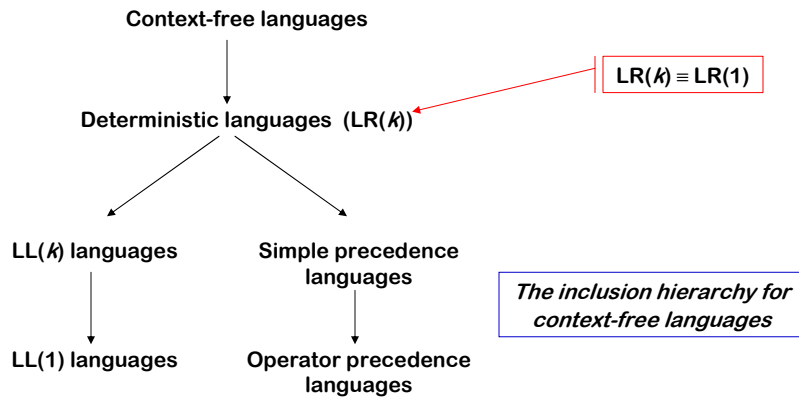
Right association

What if $y+z$ occurs elsewhere? Or $x+y$? or $x+z$?

What if $x = 2$ & $z = 17$? Neither left nor right exposes 19.

Best choice is function of surrounding context

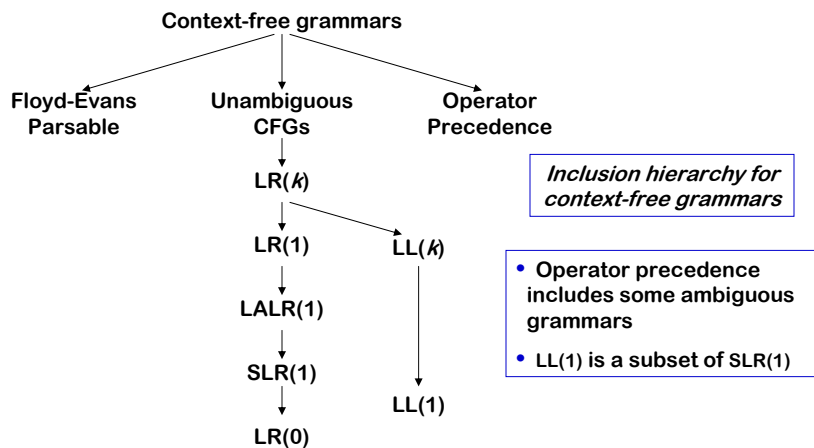
Hierarchy of Context-Free Languages



CMSC430 Spring 2007

47

Hierarchy of Context-Free Grammars



CMSC430 Spring 2007

48