Parsing VI
The LR(1) Table Construction

N.B.: This lecture uses a left-recursive version of the SheepNoise grammar. The book uses a right-recursive version.

The derivations (& the tables) are different.

Copyright 2003, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.

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 $A \rightarrow \beta$ with a • at some position in the rhs
- δ is a lookahead string of length $\leq$ k (words or **EOF**)

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

[$A \rightarrow \bullet\beta\gamma$,<u>a</u>] 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

[$A \rightarrow \beta\bullet\gamma$,<u>a</u>] means that the input sees so far is consistent with the use of $A \rightarrow \beta\gamma$ at this point in the parse, <u>and</u> that the parser has already recognized β.

[$A \rightarrow \beta\gamma\bullet$,<u>a</u>] means that the parser has seen $\beta\gamma$, <u>and</u> that a lookahead symbol of <u>a</u> is consistent with reducing to A.

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
      - [S′ →•S,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**

The SheepNoise Grammar (revisited)

We will use this grammar extensively in today's lecture

3. Goal → SheepNoise
4. SheepNoise → SheepNoise baa
5. | baa

Computing FIRST Sets

Define FIRST as

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

Note: if $\alpha = X\beta$, FIRST(α) = FIRST(X)

To compute FIRST

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

⇒ Algorithm halts

Computation of FOLLOW uses FIRST, so build FIRST sets before FOLLOW sets

Computing FIRST Sets

```
for each x ∈ T, FIRST(x) ← { x }
for each A ∈ NT, FIRST(A) ← Ø

while (FIRST sets are still changing)
  for each p ∈ P, of the form A→β,
    if β is ε then
      FIRST(A) ← FIRST(A) ∪ { ε }
    else if β is B1B2…Bk then begin
      FIRST(A) ← FIRST(A) ∪ ( FIRST(B1) - { ε } )
       for i ← 1 to k-1 by 1 while ε ∈ FIRST(Bi )
         FIRST(A) ← FIRST(A) ∪ ( FIRST(Bi +1) - { ε } )
        if i = k-1 and ε ∈ FIRST(Bk )
          then FIRST(A) ← FIRST(A) ∪ { ε }
        end
```

For SheepNoise:

FIRST(Goal) = { baa }
FIRST(SN) = { baa }
FIRST(baa) = { baa }

Computing FOLLOW Sets

FOLLOW(S) $\leftarrow$ {EOF }

for each A $\in$ NT, FOLLOW(A) $\leftarrow$ Ø

while (FOLLOW sets are still changing)

  for each p $\in$ P, of the form $A \rightarrow \beta_1\beta_2 \ldots \beta_k$

    FOLLOW(β_k) $\leftarrow$ FOLLOW(β_k) $\cup$ FOLLOW(A)

    TRAILER $\leftarrow$ FOLLOW(A)

    for i $\leftarrow$ k down to 2

      if $\varepsilon \in$ FIRST(β_i) then

        FOLLOW(β_{i-1}) $\leftarrow$ FOLLOW(β_{i-1}) $\cup$ { FIRST(β_i) – { ε } }

        $\cup$ TRAILER

      else

        FOLLOW(β_{i-1}) $\leftarrow$ FOLLOW(β_{i-1}) $\cup$ FIRST(β_i)

        TRAILER $\leftarrow$ Ø

For SheepNoise:

FOLLOW(Goal) = { EOF }

FOLLOW(SN) = { baa, EOF }

Computing Closures

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

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

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

So, S0 is

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

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

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 ← Ø
    ∀ items [A→β•Xδ,a] ∈ s
      new ← new ∪
[A→βX•δ,a]
    return closure(new)
```

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

Goto() moves forward

Example from SheepNoise

S_0 is { [Goal→ • SheepNoise,EOF], [SheepNoise→ • SheepNoise baa,EOF],

[SheepNoise→ • baa,EOF], [SheepNoise→ • SheepNoise baa,baa],

[SheepNoise→ • baa,baa] }

Goto(S_0 , baa)

- Loop produces

Item	From
[SheepNoise→baa•, EOF]	Item 3 in s_0
[SheepNoise→baa•, baa]	Item 5 in s_0

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

In the construction, this produces s2

{ [SheepNoise→baa •, {EOF,baa}]}

New, but obvious, notation for two distinct items

[SheepNoise→baa •, EOF] &
[SheepNoise→baa •, baa]

Example from SheepNoise

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

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] }

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

Building the Canonical Collection

Start from s_0 = closure([S'→S,<u>EOF</u>])

Repeatedly construct new states, until all are found

The algorithm

$s_0 \leftarrow$ closure ([S'→S,<u>EOF</u>])
$S \leftarrow \{ s_0 \}$
$k \leftarrow 1$

while (S is still changing)
  $\forall s_j \in S$ and $\forall x \in (T \cup NT)$
    $s_k \leftarrow$ goto(s_j,x)
    record $s_j \rightarrow s_k$ on x
  if $s_k \notin S$ then
    $S \leftarrow S \cup s_k$
    $k \leftarrow k + 1$

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

Worklist version is faster

Example from SheepNoise

Starts with S_0

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

[SheepNoise→• baa, EOF], [SheepNoise→ • SheepNoise baa, baa],

[SheepNoise→ • baa, baa] }

Example from SheepNoise

Starts with S_0

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] }

Example from SheepNoise

Starts with S_0

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] }

Example from SheepNoise

Starts with S_0

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

[SheepNoise→• baa, EOF], [SheepNoise→ • Shee

[SheepNoise→ • baa, baa] }

Nothing more to compute, since • is at the end of every item in S_3 .

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] }

Example (grammar & sets)

Simplified, right recursive expression grammar

Goal → Expr
Expr → Term - Expr
Expr → Term
Term → Factor * Term
Term → Factor
Factor → ident

Symbol	FIRST
Goal	{ ident }
Expr	{ ident }
Term	{ ident }
Factor	{ ident }
-	{ - }
*	{ * }
ident	{ ident }

Example (building the collection)

Initialization Step

$s_0 \leftarrow$ closure({ [Goal → •Expr , EOF] })

{ [Goal → • Expr , EOF], [Expr → • Term - Expr , EOF],
[Expr → • Term , EOF], [Term → • Factor * Term , EOF],
[Term → • Factor * Term , -], [Term → • Factor , EOF],
[Term → • Factor , -], [Factor → • ident , EOF],
[Factor → • ident , -], [Factor → • 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 → • Expr , EOF], [Expr → • Term - Expr , EOF],
[Expr → • Term , EOF], [Term → • Factor * Term , EOF],
[Term → • Factor * Term , -], [Term → • Factor , EOF],
[Term → • Factor , -], [Factor → • <u>ident</u> , EOF],
[Factor → • <u>ident</u> , -], [Factor→ • <u>ident</u>, *] }

S_1 : { [Goal → Expr •, EOF] }

S_2 : { [Expr → Term • - Expr , EOF], [Expr → Term •, EOF] }

S_3 : { [Term → Factor • * Term , EOF],[Term → Factor • * Term , -],
[Term → Factor •, EOF], [Term → Factor •, -] }

S_4 : { [Factor → <u>ident</u> •, EOF],[Factor → <u>ident</u> •, -], [Factor → <u>ident</u> •, *] }

S_5 : { [Expr → Term - • Expr , EOF], [Expr → • Term - Expr , EOF],
[Expr → • Term , EOF], [Term → • Factor * Term , -],
[Term → • Factor , -], [Term → • Factor * Term , EOF],
[Term → • Factor , EOF], [Factor → • <u>ident</u> , *],
[Factor → • <u>ident</u> , -], [Factor → • <u>ident</u> , EOF] }

Example (Summary)

S_6 : { [Term → Factor * • Term , EOF], [Term → Factor * • Term , -],
[Term → • Factor * Term , EOF], [Term → • Factor * Term , -],
[Term → • Factor , EOF], [Term → • Factor , -],
[Factor → • ident , EOF], [Factor → • ident , -], [Factor → • ident ,
*] }

S_7: { [Expr → Term - Expr •, EOF] }

S_8 : { [Term → Factor * Term •, EOF], [Term → Factor * Term •, -] }

Example (Summary)

The Goto Relationship (from the construction)

State	Expr	Term	Factor	-	*	Ident
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 $[A \rightarrow \beta \bullet \underline{a}d,\underline{b}]$ and goto$(s_x,\underline{a}) = s_k$, $\underline{a} \in T$
    - then ACTION[x,$\underline{a}$] $\leftarrow$ "shift k"
  - else if i is $[S' \rightarrow S \bullet, EOF]$
    - then ACTION[x ,$\underline{a}$] $\leftarrow$ "accept"
  - else if i is $[A \rightarrow \beta \bullet, \underline{a}]$
    - then ACTION[x,$\underline{a}$] $\leftarrow$ "reduce $A \rightarrow \beta$"
- $\forall$ $n \in NT$
  - if 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 $[A \rightarrow \beta \bullet X \delta, \underline{a}]$

Example (Filling in the tables)

The algorithm produces the following table

	ACTION				GOTO		
	Ident	-	*	EOF	Expr	Term	Factor
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			

Plugs into the skeleton LR(1) parser

What can go wrong?

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

- First item generates "shift", second generates "reduce"
- Both define ACTION[s,$\underline{a}$] — 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 is set s contains $[A \rightarrow \gamma \cdot, \underline{a}]$ and $[B \rightarrow \gamma \cdot, \underline{a}]$?

- Each generates "reduce", but with a different production
- Both define ACTION[s,$\underline{a}$] — cannot do both reductions
- This fundamental ambiguity is called a reduce/reduce error
- Modify the grammar to eliminate it (PL/I'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) $\Rightarrow$ 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

LL(k) $\Rightarrow$ Parser must select the reduction based on

1. The complete left context
2. The next k terminals

Thus, LR(k) examines more context

"… 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

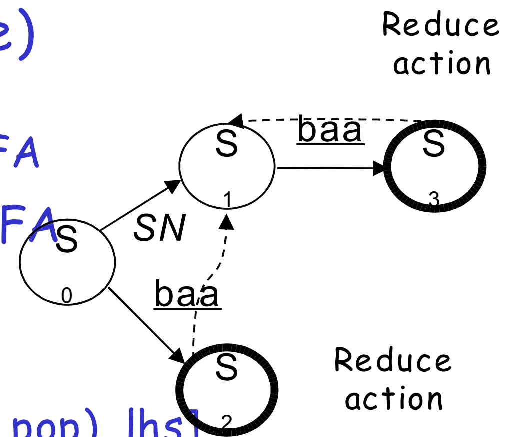
	Advantages	Disadvantages
Top-down recursive descent	Fast Good locality Simplicity Good error detection	Hand-coded High maintenance Right associativity
LR(1)	Fast Deterministic langs. Automatable Left associativity	Large working sets Poor error messages Large table sizes

Extra Slides Start Here

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

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)

Terminal or non-terminal

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

LR(1) Items

The production $A \rightarrow \beta$, where $\beta = B_1B_1B_1$ with lookahead $\underline{a}$, can give rise to 4 items

$[A \rightarrow \cdot B_1B_2B_3, \underline{a}]$, $[A \rightarrow B_1 \cdot B_2B_3, \underline{a}]$, $[A \rightarrow B_1B_2 \cdot B_3, \underline{a}]$, & $[A \rightarrow B_1B_2B_3 \cdot, \underline{a}]$

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 $\underline{a}$ implies a reduction by $A \rightarrow \beta$
  - For $\{ [A \rightarrow \beta \cdot, \underline{a}], [B \rightarrow \gamma \cdot \delta, \underline{b}] \}$, $\underline{a} \Rightarrow$ ***reduce*** to A; FIRST(δ) $\Rightarrow$ ***shift***

$\Rightarrow$ Limited right context is enough to pick the actions

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 *

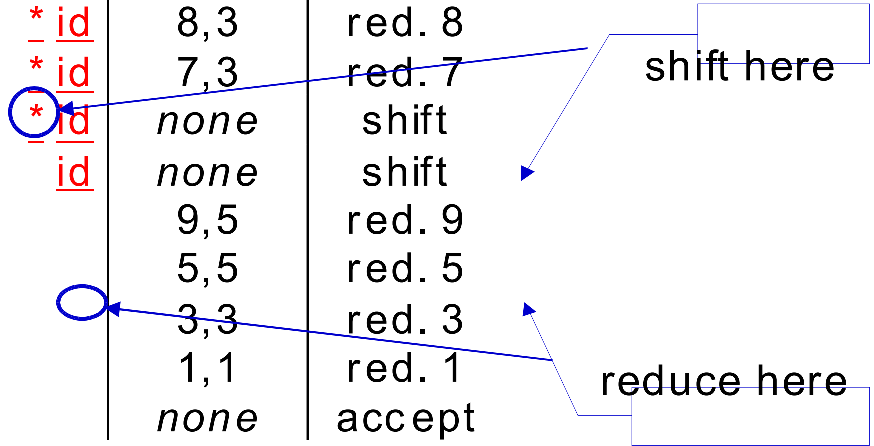
How did it choose to expand Term rather than reduce to Expr?

- 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

Bac

Remember this slide from last lecture?

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