CSE 401 – Compilers

LR Parser Construction
Hal Perkins
Winter 2009

Agenda

- LR(0) state construction
- FIRST, FOLLOW, and nullable
- Variations: SLR, LR(1), LALR

LR State Machine

- Idea: Build a DFA that recognizes handles
  - Language generated by a CFG is generally not regular, but
  - Language of handles for a CFG is regular
    - So a DFA can be used to recognize handles
  - Parser reduces when DFA accepts

Prefixes, Handles, &c (review)

- If *S* is the start symbol of a grammar *G*,
  - If $S \Rightarrow^* \alpha$ then α is a *sentential form* of *G*
  - γ is a *viable prefix* of *G* if there is some derivation $S \Rightarrow^*_{rm} \alpha A w \Rightarrow^*_{rm} \alpha\beta w$ and γ is a prefix of $\alpha\beta$.
  - The occurrence of β in $\alpha\beta w$ is a *handle* of $\alpha\beta w$
- An *item* is a marked production (a . at some position in the right hand side)
  - [*A* ::= . *X Y*]   [*A* ::= *X* . *Y*]   [*A* ::= *X Y* .]

Building the LR(0) States

- Example grammar

  S′ ::= *S* $
  S ::= (*L*)
  S ::= x
  L ::= *S*
  L ::= *L* , *S*

  - We add a production S′ with the original start symbol followed by end of file ($)
  - Question: What language does this grammar generate?

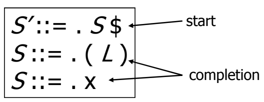
Start of LR Parse

0. *S′* ::= *S* $
1. *S* ::= (*L*)
2. *S* ::= x
3. *L* ::= *S*
4. *L* ::= *L* , *S*

- Initially
  - Stack is empty
  - Input is the right hand side of *S′*, i.e., *S* $
  - Initial configuration is [*S′* ::= . *S* $]
  - But, since position is just before *S*, we are also just before anything that can be derived from *S*

0. $S' ::= S\$$
1. $S ::= (L)$
2. $S ::= x$
3. $L ::= S$
4. $L ::= L , S$

Initial state

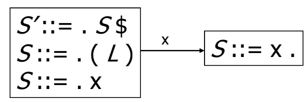


- A state is just a set of items
  - Start: an initial set of items
  - Completion (or closure): additional productions whose left hand side appears to the right of the dot in some item already in the state

1/16/2009 © 2002-09 Hal Perkins & UW CSE E-7

0. $S' ::= S\$$
1. $S ::= (L)$
2. $S ::= x$
3. $L ::= S$
4. $L ::= L , S$

Shift Actions (1)

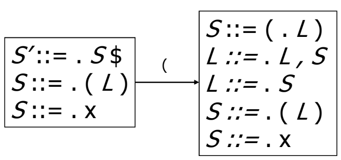


- To shift past the x, add a new state with the appropriate item(s)
  - In this case, a single item; the closure adds nothing
  - This state will lead to a reduction since no further shift is possible

1/16/2009 © 2002-09 Hal Perkins & UW CSE E-8

0. $S' ::= S\$$
1. $S ::= (L)$
2. $S ::= x$
3. $L ::= S$
4. $L ::= L , S$

Shift Actions (2)

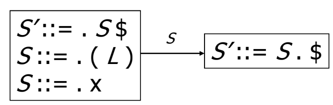


- If we shift past the (, we are at the beginning of *L*
- the closure adds all productions that start with *L*, which requires adding all productions starting with *S*

1/16/2009 © 2002-09 Hal Perkins & UW CSE E-9

0. $S' ::= S\$$
1. $S ::= (L)$
2. $S ::= x$
3. $L ::= S$
4. $L ::= L , S$

Goto Actions



- Once we reduce *S*, we'll pop the rhs from the stack exposing the first state. Add a *goto* transition on *S* for this.

1/16/2009 © 2002-09 Hal Perkins & UW CSE E-10

Basic Operations

- *Closure* (*S*)
  - Adds all items implied by items already in *S*
- *Goto* (*I, X*)
  - *I* is a set of items
  - *X* is a grammar symbol (terminal or non-terminal)
  - *Goto* moves the dot past the symbol *X* in all appropriate items in set *I*

1/16/2009 © 2002-09 Hal Perkins & UW CSE E-11

Closure Algorithm

- *Closure* (*S*) =

  repeat
    for any item [A ::= α . $X\beta$] in *S*
      for all productions *X* ::= γ
        add [*X* ::= . γ] to *S*
  until *S* does not change
  return *S*

1/16/2009 © 2002-09 Hal Perkins & UW CSE E-12

Goto Algorithm

- *Goto* (*I*, *X*) =

  set *new* to the empty set
  for each item [A ::= α . *X* β] in *I*
  add [A ::= α *X* . β] to *new*
  return *Closure* (*new*)

  - This may create a new state, or may return an existing one

LR(0) Construction

- First, augment the grammar with an extra start production *S'* ::= *S* $
- Let *T* be the set of states
- Let *E* be the set of edges
- Initialize *T* to *Closure* ([*S'* ::= . *S* $])
- Initialize *E* to empty

LR(0) Construction Algorithm

```
repeat
  for each state I in T
    for each item [A ::= α . X β] in I
      Let new be Goto( I, X )
      Add new to T if not present
      Add I -X-> new to E if not present
until E and T do not change in this iteration
```

- Footnote: For symbol $, we don't compute *goto* (*I*, $); instead, we make this an *accept* action.

LR(0) Reduce Actions

- Algorithm:

  Initialize *R* to empty
  for each state *I* in *T*
  for each item [*A* ::= α .] in *I*
  add (*I*, *A* ::= α) to *R*

Building the Parse Tables (1)

- For each edge $I \xrightarrow{X} J$
  - if X is a terminal, put s*j* in column X, row *I* of the action table (shift to state *j*)
  - If X is a non-terminal, put g*j* in column X, row *I* of the goto table

Building the Parse Tables (2)

- For each state *I* containing an item [*S'* ::= *S* . $], put *accept* in column $ of row *I*
- Finally, for any state containing [*A* ::= γ .] put action r*n* in every column of row *I* in the table, where *n* is the production number

Example: States for

0. $S' ::= S\$$
1. $S ::= (L)$
2. $S ::= x$
3. $L ::= S$
4. $L ::= L , S$

Example: Tables for

0. $S' ::= S\$$
1. $S ::= (L)$
2. $S ::= x$
3. $L ::= S$
4. $L ::= L , S$

Where Do We Stand?

- We have built the LR(0) state machine and parser tables
  - No lookahead yet
  - Different variations of LR parsers add lookahead information, but basic idea of states, closures, and edges remains the same

A Grammar that is not LR(0)

- Build the state machine and parse tables for a simple expression grammar

  $S ::= E\$$

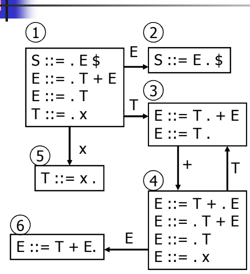
  $E ::= T + E$

  $E ::= T$

  $T ::= x$

LR(0) Parser for

0. $S ::= E\$$
1. $E ::= T + E$
2. $E ::= T$
3. $T ::= x$



	x	+	$	E	T
1	s5			g2	G3
2			acc		
3	r2	s4,r2	r2		
4	s5			g6	G3
5	r3	r3	r3		
6	r1	r1	r1		

- State 3 is has two possible actions on +
  - shift 4, or reduce 2
- ∴ Grammar is not LR(0)

SLR Parsers

- Idea: Use information about what can follow a non-terminal to decide if we should perform a reduction
- Easiest form is SLR – Simple LR
- So we need to be able to compute FOLLOW(A) – the set of symbols that can follow A in any possible derivation
  - But to do this, we need to compute FIRST(γ) for strings γ that can follow A

Calculating FIRST(γ)

- Sounds easy… If $\gamma = X\,Y\,Z$, then FIRST(γ) is FIRST(X), right?
  - But what if we have the rule X ::= ε?
  - In that case, FIRST(γ) includes anything that can follow an X – i.e. FOLLOW(X)

FIRST, FOLLOW, and nullable

- nullable(X) is true if X can derive the empty string
- Given a string γ of terminals and non-terminals, FIRST(γ) is the set of terminals that can begin strings derived from γ.
- FOLLOW(X) is the set of terminals that can immediately follow X in some derivation
- All three of these are computed together

Computing FIRST, FOLLOW, and nullable (1)

- Initialization

  set FIRST and FOLLOW to be empty sets
  set nullable to false for all non-terminals
  set FIRST[a] to a for all terminal symbols a

Computing FIRST, FOLLOW, and nullable (2)

```
repeat
  for each production X := Y1 Y2 … Yk
     if Y1 … Yk are all nullable (or if k = 0)
        set nullable[X] = true
     for each i from 1 to k and each j from i+1 to k
        if Y1 … Yi-1 are all nullable (or if i = 1)
           add FIRST[Yi] to FIRST[X]
        if Yi+1 … Yk are all nullable (or if i = k)
           add FOLLOW[X] to FOLLOW[Yi]
        if Yi+1 … Yj-1 are all nullable (or if i+1=j)
           add FIRST[Yj] to FOLLOW[Yi]
Until FIRST, FOLLOW, and nullable do not change
```


Example

- Grammar

  Z ::= d
  Z ::= $X\,Y\,Z$
  Y ::= ε
  Y ::= c
  X ::= Y
  X ::= a

	nullable	FIRST	FOLLOW
X			
Y			
Z			

SLR Construction

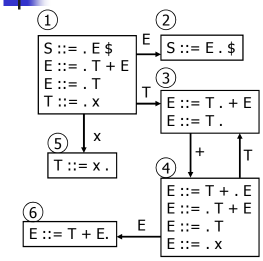
- This is identical to LR(0) – states, etc., except for the calculation of reduce actions
- Algorithm:

  Initialize R to empty
  for each state I in T
    for each item [A ::= α .] in I
      for each terminal a in FOLLOW(A)
        add (I, a, A ::= α) to R

  - i.e., reduce α to A in state I only on lookahead a

SLR Parser for

0. S ::= E $
1. E ::= T + E
2. E ::= T
3. T ::= x



	x	+	$	E	T
1	s5			g2	g3
2			acc		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

On To LR(1)

- Many practical grammars are SLR
- LR(1) is more powerful yet
- Similar construction, but notion of an item is more complex, incorporating lookahead information

LR(1) Items

- An LR(1) item [*A* ::= α . β, a] is
  - A grammar production (*A* ::= αβ)
  - A right hand side position (the dot)
  - A lookahead symbol (a)
- Idea: This item indicates that α is the top of the stack and the next input is derivable from βa.
- Full construction: see the book

LR(1) Tradeoffs

- LR(1)
  - Pro: extremely precise; largest set of grammars
  - Con: potentially very large parse tables with many states

LALR(1)

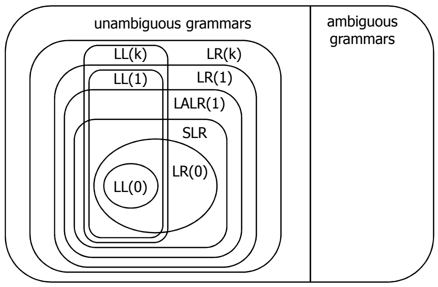
- Variation of LR(1), but merge any two states that differ only in lookahead
  - Example: these two would be merged

    [*A* ::= x . , a]
    [*A* ::= x . , b]

LALR(1) vs LR(1)

- LALR(1) tables can have many fewer states than LR(1)
- LALR(1) may have reduce conflicts where LR(1) would not (but in practice this doesn't happen often)

Language Heirarchies



1/16/2009 © 2002-09 Hal Perkins & UW CSE E-37

Coming Attractions

- LL(k) Parsing – Top-Down
- Recursive Descent Parsers
  - What you can do if you need a parser in a hurry
- But first, the next part of the project: parsing and AST generation

1/16/2009 © 2002-09 Hal Perkins & UW CSE E-38