CSE 401 – Compilers

Lecture 8: LR Parser Construction

Michael Ringenburg

Winter 2013

Reminders/ Announcements

- Project part 2 is due Monday.
- Next week:
  - We'll assign project part 2 (due 2 weeks later) – we should get through the necessary material by Wednesday, and you'll review it in Sections on Thursday.
  - We'll also assign homework 2 (due 1 week later).
- Changed the schedule on the web slightly, in order to make sure we get through everything you need for project part 2.

Agenda

- Finish describing shift-reduce and reduce-reduce conflicts (from last lecture).
- Building LR parser DFAs
  - LR(0) state construction
  - Adding FIRST, FOLLOW, and nullable (SLR parsing)
  - Briefly: LR(1), LALR, and the hierarchy of parsers/ grammars.

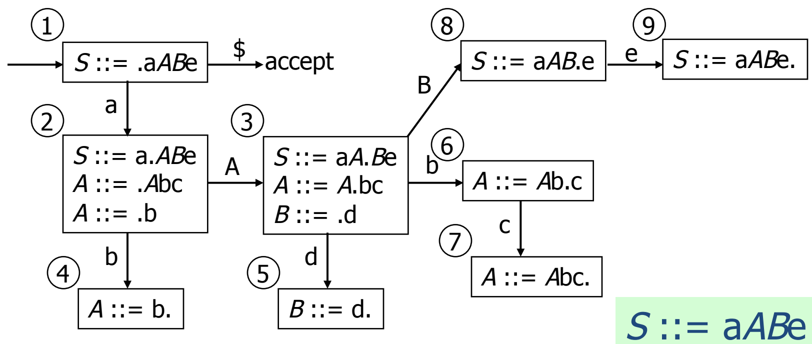
Quick Review

- An *item* is a marked production (a . at some position on the right hand side)
  - [S ::= . a *A* *B* e] [S ::= a . *A* *B* e] [S ::= a *A* . *B* e] [S ::= a *A* *B* . e] [S ::= a *A* *B* e .]
  - [*A* ::= . *A* b c] [*A* ::= *A* . b c] [*A* ::= *A* b c .]
  - [A ::= . b] [A ::= b .]
  - [B ::= . d] [B ::= d .]

S ::= a*AB*e
A ::= *A*bc | b
B ::= d

- A parser DFA state corresponds to a set of items, where each item corresponds to a handle that we might be scanning in that state, as well as how much of the handle we have already read.

Review: DFA States & Items



S ::= aABe
A ::= Abc | b
B ::= d

Items & Shift/Reduce

- What do we do if the dot is at the end of an item?
  - We've seen the entire handle, so …
  - Reduce by the production!
- What if the dot is not at the end of the item?
  - We need to read more input to find the rest of the handle, so …
  - Shift!

Problems with Grammars

- Grammars can cause problems when constructing a LR parser
  - Recall that states may (and often do) correspond to multiple items
  - What if one item in a state indicates we should shift (part way through), and another indicates we should reduce (end)?
    - **Shift-reduce conflict**
  - What if we are at the end of two different items in then state, indicating two *different* reductions?
    - **Reduce-reduce conflict**

Shift-Reduce Conflicts

- Situation: both a shift and a reduce are possible at a given point in the parse (equivalently: in a particular state of the DFA)
- Classic example: if-else statement (condition omitted to save space)

  S ::= ifthen *S* | ifthen *S* else *S*

Parser States

- State 3 has a shift-reduce conflict
  - Can shift past else into state 4 (s4)
  - Can reduce (r1)
    S ::= ifthen *S*

(Note: some items omitted in states 2-4 to save space)

1. *S* ::= ifthen *S*
2. *S* ::= ifthen *S* else *S*

1 | *S* ::= . ifthen *S*
  | *S* ::= . ifthen *S* else *S*

ifthen ↓

2 | *S* ::= ifthen . *S*
  | *S* ::= ifthen . *S* else *S*

S ↓

3 | *S* ::= ifthen *S* .
  | *S* ::= ifthen *S* . else *S*

else ↓

4 | *S* ::= ifthen *S* else . *S*

Solving Shift-Reduce Conflicts

- Fix the grammar (like we saw before)
  - Done in Java reference grammar, others
- Use a parser generator with a "longest match" rule – i.e., if there is a conflict, choose to shift instead of reduce
  - Does exactly what we want for if-else case
  - Guideline: a few shift-reduce conflicts are fine, but be sure they do what you want

Reduce-Reduce Conflicts

- Situation: two different reductions are possible in a given state
- Contrived example
  1. *S* ::= *A*
  2. *S* ::= *B*
  3. *A* ::= x
  4. *B* ::= x
- What happens when you try to parse x?
  - Which reduction do you use initially? r3 or r4?

Parser States for

1. *S* ::= *A*
2. *S* ::= *B*
3. *A* ::= x
4. *B* ::= x

1 | *S* ::= .*A*
S ::= .*B*

Parser States for

1. $S ::= A$
2. $S ::= B$
3. $A ::= \text{x}$
4. $B ::= \text{x}$

1

$S ::= .A$
$S ::= .B$
$A ::= .\text{x}$
$B ::= .\text{x}$

Parser States for

1. $S ::= A$
2. $S ::= B$
3. $A ::= \text{x}$
4. $B ::= \text{x}$

1

$S ::= .A$
$S ::= .B$
$A ::= .\text{x}$
$B ::= .\text{x}$

$\xrightarrow{\text{x}}$

2

$A ::= \text{x}.$
$B ::= \text{x}.$

- State 2 has a reduce-reduce conflict (r3, r4)

Handling Reduce-Reduce Conflicts

- These normally indicate a problem with the grammar – can't be parsed by this type of parser.
- How to fix?
  - Use a different kind of parser generator that takes lookahead information into account when constructing the states
    - SLR, LALR, LR(1)
    - Most practical tools use this information
    - However, reduce-reduce conflicts are still possible – these will only eliminate some.
  - Fix the grammar

Another (more realistic) Reduce-Reduce Conflict

- Suppose the grammar separates arithmetic and boolean expressions, so you can't use a boolean typed identifier in an arithmetic expression (and vice versa):

  expr ::= *aexp* | *bexp*

  aexp ::= *aexp* * *aident* | *aident*

  bexp ::= *bexp* && *bident* | *bident*

  aident ::= *id*

  bident ::= *id*

- This will create a reduce-reduce conflict

Covering Grammars

- A solution is to merge *aident* and *bident* into a single non-terminal (or use *id* in place of *aident* and *bident* everywhere they appear)
- This is a *covering grammar*
  - Includes some programs that are not generated by the original grammar (allows booleans in arithmetic, and vice versa).
  - Use the type checker or other static semantic analysis to weed out illegal programs later

Agenda

- Finish describing shift-reduce and reduce-reduce conflicts (from last lecture).
- Building LR parser DFAs
  - LR(0) state construction
  - Adding FIRST, FOLLOW, and nullable (SLR parsing)
  - Briefly: LR(1), LALR, and the hierarchy of parsers/ grammars.

LR State Machine

- Our LR parsing algorithm requires a DFA that recognizes viable prefixes/handles.
  - We constructed one by hand for our sample language.
- How do we do it in general?
  - Real answer: You don't, you use a tool! ☺ But we should still understand the process.
  - Recall that the language generated by a CFG is generally not regular, but
  - Language of handles and viable prefixes is regular

Building the LR(0) States

- Example grammar

  S ::= (*L*)
  S ::= x
  L ::= *S*
  L ::= *L* , *S*
  - Question: What language does this grammar generate?

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 ($). If we get to the end of this item [S' ::= S$.], we accept rather than reduce.
  – Question: What language does this modified grammar generate?

Start of LR Parse

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

- At the beginning of the parse:
  – 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*

Initial state

$S' ::= .\ S\ \$$
$S ::= .\ (\ L\ )$
$S ::= .\ \mathrm{x}$

start

completion

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

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

Shift Actions (1)

$$\begin{array}{|l|}\hline S' ::= .\ S\ \$ \\ S ::= .\ (\ L\ ) \\ S ::= .\ \mathrm{x} \\ \hline \end{array} \xrightarrow{\mathrm{x}} \boxed{S ::= \mathrm{x}\ .}$$

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

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

Shift Actions (2)

$S' ::= .\ S\ \$$
$S ::= .\ (\ L\ )$
$S ::= .\ \mathrm{x}$

$\xrightarrow{(}$

$S ::= (\ .\ L\ )$
$L ::= .\ L\ ,\ S$
$L ::= .\ S$
$S ::= .\ (\ L\ )$
$S ::= .\ \mathrm{x}$

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

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

Reduce Actions

$S' ::= .\ S\ \$$
$S ::= .\ (\ L\ )$
$S ::= .\ \mathrm{x}$

$\xrightarrow{S}$

$S' ::= S\ .\ \$$

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

- If we reduce to *S*, and popping the rhs exposes the first state, we can consume an S in the first item. Add a *goto* transition on *S* for this.

Basic Operations

- *Closure* (*S*)
  - Adds all items "implied by" items already in *S.* If a nonterminal is directly to the right of the dot, add items for the start of its productions (transitively).
- *Transition* (*I, X*) (sometimes called *Goto*, but I find this misleading)
  - *I* is a set of **items** (typically the items for a state)
  - *X* is a grammar symbol (terminal or non-terminal)
  - *Transition* moves the dot past the symbol *X* in all appropriate items in set *I*

Closure Algorithm

- Fixed point algorithm for Closure
- *Closure* (*S*) =

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

Transition Algorithm

- *Transition* (*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 *Transition* (*I*, *X*)
      Add *new* to *T* if not present
      Add $I \xrightarrow{X} new$ to *E* if not present
until *E* and *T* do not change in this iteration

- Footnote: For symbol $ (only appears in items of production S' ::= S $), we don't compute *transition* (*I*, $); instead, we make this an *accept* action.

Example: States for

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

Building the Parse Tables

- 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

- 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* (reduce) in every column of row *I* in the table, where *n* is the *production* number

Example: Tables for

0. $S' ::= S\,\$$
1. $S ::= (L)$
2. $S ::= \mathrm{x}$
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* $

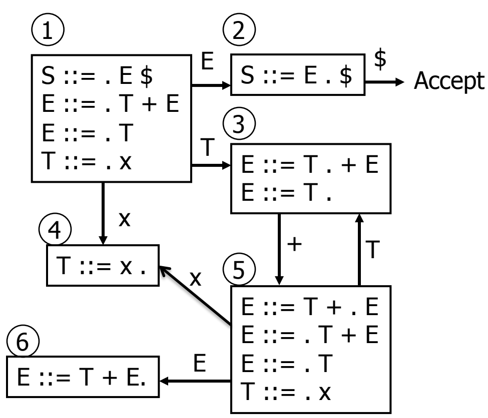
  E ::= *T* + *E*

  E ::= *T*

  T ::= x

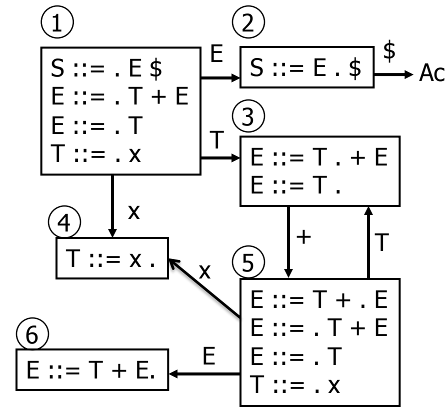
LR(0) Parser for

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



LR(0) Parser for

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

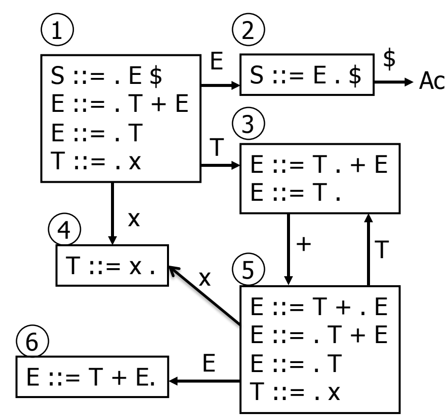


	x	+	$	E	T
1					
2					
3					
4					
5					
6					

- First, add the shift and goto transitions (edges of the DFA).

LR(0) Parser for

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

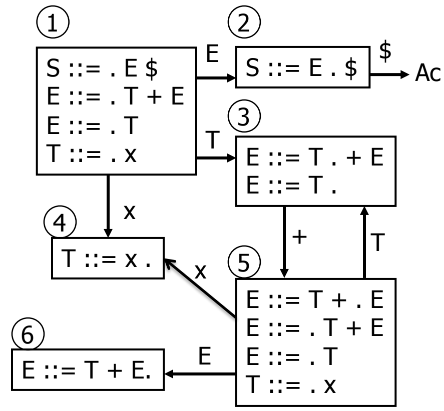


	x	+	$	E	T
1	s4			g2	g3
2					
3		s5			
4					
5	s4			g6	g3
6					

- Then, add the reduce and accept actions.

LR(0) Parser for

0. $S ::= E\,\$$
1. $E ::= T + E$
2. $E ::= T$
3. $T ::= \mathrm{x}$



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

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

Next Time

- How do we use lookahead to solve this issue?
  - We'll show the simplest way, known as SLR (simplified LR) parsing.
  - We'll also briefly describe how lookahead is used in the more complex LALR(k) and LR(k) parsers.
- Start describing how to create a parser with CUP, and use it to build an AST (likely won't finish until Wednesday).
  - This is what you'll do in your project.
  - Plus, how to use the visitor pattern to work with your AST!