


CSE 582 – Compilers

LR Parsing
Hal Perkins
Autumn 2002


10/10/2002 © 2002 Hal Perkins & UW CSE D-1



Agenda

- n LR Parsing
- n Table-driven Parsers
- n Parser States
- n Shift-Reduce and Reduce-Reduce conflicts

10/10/2002 © 2002 Hal Perkins & UW CSE D-2



LR(1) Parsing

- n We'll look at LR(1) parsers
 - n Left to right scan, Rightmost derivation, 1 symbol lookahead
 - n Most practical programming languages have an LR(1) grammar
 - n LALR(1), SLR(1), etc. – subsets of LR(1)

10/10/2002 © 2002 Hal Perkins & UW CSE D-3

Bottom-Up Parsing

- n Idea: Read the input left to right
- n Whenever we've matched the right hand side of a production, reduce it to the appropriate non-terminal and add that non-terminal to the parse tree
- n The upper edge of this partial parse tree is known as the *frontier*

10/10/2002 © 2002 Hal Perkins & UW CSE D-4

Example

- n Grammar
- n Bottom-up Parse

$S ::= aABe$
 $A ::= Abc \mid b$
 $B ::= d$


a b b c d e

10/10/2002 © 2002 Hal Perkins & UW CSE D-5

Details

- n The bottom-up parser reconstructs a reverse rightmost derivation
- n Given the rightmost derivation
 - $S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w$
 - the parser will first discover $\beta_{n-1} \Rightarrow \beta_n$, then $\beta_{n-2} \Rightarrow \beta_{n-1}$, etc.
- n Parsing terminates when
 - n β_1 reduced to S (success), or
 - n No match can be found (syntax error)


10/10/2002 © 2002 Hal Perkins & UW CSE D-6



How Do We Automate This?

- n Key: given what we've already seen and the next input symbol, decide what to do.
- n Choices:
 - n Perform a reduction
 - n Look ahead further
- n Can reduce $A \Rightarrow \beta$ if both of these hold:
 - n $A \Rightarrow \beta$ is a valid production
 - n $A \Rightarrow \beta$ is a step in *this* rightmost derivation
- n This is known as a *shift-reduce* parser

10/10/2002 © 2002 Hal Perkins & UW CSE D-7




Sentential Forms

- n If $S \Rightarrow^* \alpha$, the string α is called a *sentential form* of the of the grammar
- n In the derivation

$$S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \dots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w$$
 each of the β_i are sentential forms
- n A sentential form in a rightmost derivation is called a *right-sentential form* (similarly for leftmost and left-sentential)


10/10/2002 © 2002 Hal Perkins & UW CSE D-8



Handles

- n Informally, a substring of the tree frontier that matches the right side of a production
 - n Even if $A ::= \beta$ is a production, β is a handle only if it matches the frontier at a point where $A ::= \beta$ was used in the derivation
 - n β may appear in many other places in the frontier without being a handle for that production


10/10/2002 © 2002 Hal Perkins & UW CSE D-9



Handles (cont.)

- ▣ Formally, a *handle* of a right-sentential form γ is a production $A ::= \beta$ and a position in γ where β may be replaced by A to produce the previous right-sentential form in the rightmost derivation of γ


10/10/2002 © 2002 Hal Perkins & UW CSE D-10



Handle Examples

- ▣ In the derivation
 - $S \Rightarrow aAbc \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abcde$
 - ▣ $abcde$ is a right sentential form whose handle is $A ::= b$ at position 2
 - ▣ $aAbcde$ is a right sentential form whose handle is $A ::= Abc$ at position 4
 - ▣ Note: some books take the left of the match as the position (e.g., Dragon Book)

10/10/2002 © 2002 Hal Perkins & UW CSE D-11



Implementing Shift-Reduce Parsers

- ▣ Key Data structures
 - ▣ A stack holding the frontier of the tree
 - ▣ A string with the remaining input

10/10/2002 © 2002 Hal Perkins & UW CSE D-12

Shift-Reduce Parser Operations

- n *Reduce* – if the top of the stack is the right side of a handle $A::=\beta$, pop the right side β and push the left side A .
- n *Shift* – push the next input symbol onto the stack
- n *Accept* – announce success
- n *Error* – syntax error discovered

10/10/2002 © 2002 Hal Perkins & UW CSE D-13

Shift-Reduce Example

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

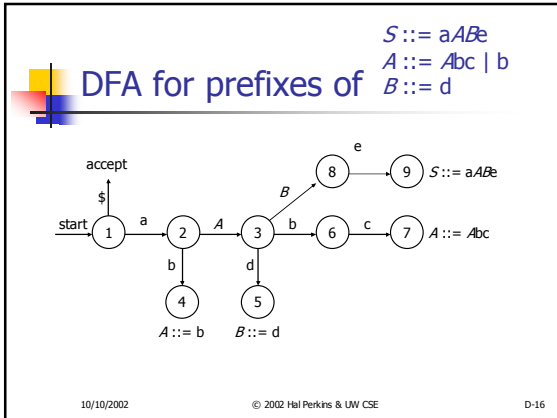
Stack	Input	Action
\$	abcde\$	<i>shift</i>

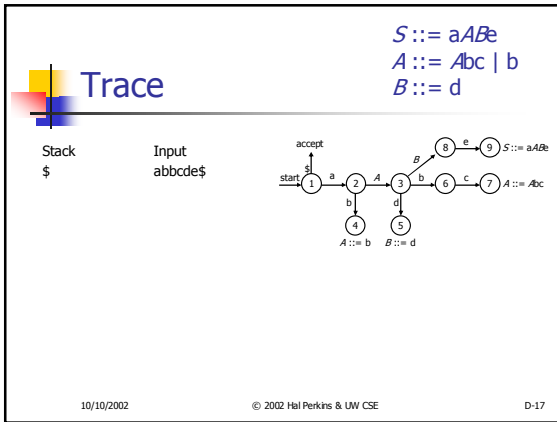
10/10/2002 © 2002 Hal Perkins & UW CSE D-14

How Do We Automate This?


- n Def. *Viable prefix* – a prefix of a right-sentential form that can appear on the stack of the shift-reduce parser
 - n Equivalent: a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form
- n Idea: Construct a DFA to recognize viable prefixes given the stack and remaining input
 - n Perform reductions when we recognize them

10/10/2002 © 2002 Hal Perkins & UW CSE D-15






- Observations**
- n Way too much backtracking
 - n We want the parser to run in time proportional to the length of the input
 - n Where the heck did this DFA come from anyway?
 - n From the underlying grammar
 - n We'll defer construction details for now
- 10/10/2002 © 2002 Hal Perkins & UW CSE D-18



Avoiding DFA Rescanning

- Observation: after a reduction, the contents of the stack are the same as before except for the new non-terminal on top
 - ∴ Scanning the stack will take us through the same transitions as before until the last one
 - ∴ If we record state numbers on the stack, we can go directly to the appropriate state when we pop the right hand side of a production from the stack

10/10/2002 © 2002 Hal Perkins & UW CSE D-19




Stack

- Change the stack to contain pairs of states and symbols from the grammar

$$s_0 X_1 S_1 X_1 S_1 \dots X_n S_n$$
 - State s_0 represents the accept state
 - (Not always added – depends on particular presentation)
- Observation: in an actual parser, only the state numbers need to be pushed, since they implicitly contain the symbol information, but for explanations, it's clearer to use both.


10/10/2002 © 2002 Hal Perkins & UW CSE D-20



Encoding the DFA in a Table

- A shift-reduce parser's DFA can be encoded in two tables
 - One row for each state
 - *action* table encodes what to do given the current state and the next input symbol
 - *goto* table encodes the transitions to take after a reduction


10/10/2002 © 2002 Hal Perkins & UW CSE D-21



Actions (1)

- Given the current state and input symbol, the main possible actions are
 - si – shift the input symbol and state i onto the stack (i.e., shift and move to state i)
 - rj – reduce using grammar production j
 - The production number tells us how many <symbol, state> pairs to pop off the stack


10/10/2002 © 2002 Hal Perkins & UW CSE D-22



Actions (2)

- Other possible *action* table entries
 - *accept*
 - blank – no transition – syntax error
 - A LR parser will detect an error as soon as possible on a left-to-right scan
 - A real compiler needs to produce an error message, recover, and continue parsing when this happens

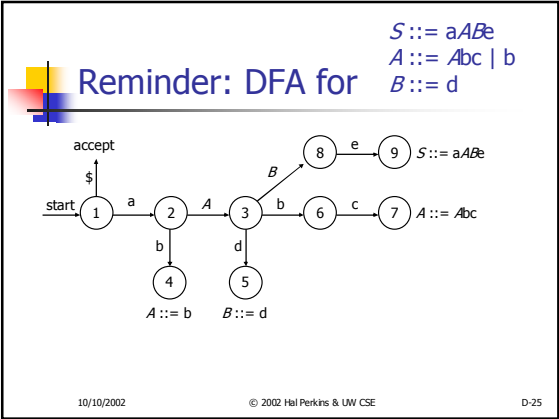
10/10/2002 © 2002 Hal Perkins & UW CSE D-23



Goto

- When a reduction is done, <symbol, state> pairs are popped from the stack revealing a state *uncovered_s* on the top of the stack
- $\text{goto}[\textit{uncovered_s}, A]$ is the new state to push on the stack when reducing production $A ::= \beta$ (after popping β)

10/10/2002 © 2002 Hal Perkins & UW CSE D-24



$S ::= aAB\epsilon$
 $A ::= Abc$
 $A ::= b$
 $B ::= d$

LR Parse Table for

State	action					goto		
	a	b	c	d	e	\$	A	B
1	s2					acc		
2		s4					g3	
3		s6		s5				g8
4	r3	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4	r4		
6			s7					
7	r2	r2	r2	r2	r2	r2		
8					s9			
9	r1	r1	r1	r1	r1	r1		

10/10/2002 © 2002 Hal Perkins & UW CSE D-26

LR Parsing Algorithm (1)

```

word = scanner.getToken();
while (true) {
  s = top of stack;
  if (action[s, word] = s') {
    push word; push / (state);
    word = scanner.getToken();
  } else if (action[s, word] = rj) {
    pop 2 * length of right side of
    production j (2*|βj|);
    uncovered_s = top of stack;
    push left side A of production j;
    push state goto[uncovered_s, A];
  }
  } else if (action[s, word] = accept ) {
    return;
  } else {
    // no entry in action table
    report syntax error;
    halt or attempt recovery;
  }
}
  
```

10/10/2002 © 2002 Hal Perkins & UW CSE D-27

Example

Stack: \$
Input: abbcd\$

$S ::= aAB\epsilon$
 $A ::= Abc$
 $A ::= b$
 $B ::= d$

S	action						goto	
	a	b	c	d	e	\$	A	B
1	s2					ac		
2		s4					g3	
3		s6		s5				g8
4	r3	r3	r3	r3	r3	r3		
5	r4	r4	r4	r4	r4	r4		
6				s7				
7	r2	r2	r2	r2	r2	r2		
8					s9			
9	r1	r1	r1	r1	r1	r1		

10/10/2002 © 2002 Hal Perkins & UW CSE D-28

LR States

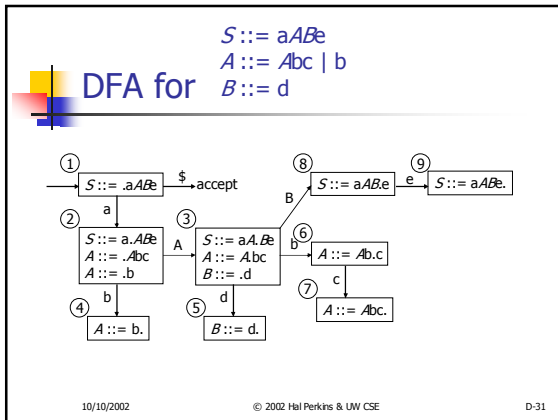
- Idea is that each state encodes
 - The set of all possible productions that we could be looking at, given the current state of the parse, and
 - Where we are in the right hand side of each of those productions

10/10/2002 © 2002 Hal Perkins & UW CSE D-29

Items

- An *item* is a production with a dot in the right hand side
- Example: Items for production $A ::= XY$
 - $A ::= .XY$
 - $A ::= X.Y$
 - $A ::= XY.$
- Idea: The dot represents a position in the production

10/10/2002 © 2002 Hal Perkins & UW CSE D-30



Problems with Grammars

- Grammars can cause to problems when constructing a LR parser
 - Shift-reduce conflicts
 - Reduce-reduce conflicts

10/10/2002 © 2002 Hal Perkins & UW CSE D-32

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

$$S ::= \text{ifthen } S \mid \text{ifthen } S \text{ else } S$$

10/10/2002 © 2002 Hal Perkins & UW CSE D-33

Parser States for

$S ::= \text{ifthen } S$
 $S ::= \text{ifthen } S \text{ else } S$

① $S ::= \text{.ifthen } S$
 $S ::= \text{.ifthen } S \text{ else } S$

ifthen ↓

② $S ::= \text{ifthen .} S$
 $S ::= \text{ifthen .} S \text{ else } S$

S ↓

③ $S ::= \text{ifthen } S.$
 $S ::= \text{ifthen } S. \text{else } S$

else ↓

④ $S ::= \text{ifthen } S \text{ else .} S$

- State 3 has a shift-reduce conflict
 - Can shift past else into state 4 (s4)
 - Can reduce (r1) $S ::= \text{ifthen } S$

(Note: other $S ::= \text{.ifthen}$ items not included in states 2-4 to save space)

10/10/2002 © 2002 Hal Perkins & UW CSE D-34

Solving Shift-Reduce Conflicts

- Fix the grammar
- Use a parse tool 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
 - Moral: a few shift-reduce conflicts are fine, but be sure that they do what you want

10/10/2002 © 2002 Hal Perkins & UW CSE D-35

Reduce-Reduce Conflicts

- Situation: two different reductions are possible in a given state
- Contrived example
 - $S ::= A$
 - $S ::= B$
 - $A ::= x$
 - $B ::= x$

10/10/2002 © 2002 Hal Perkins & UW CSE D-36

$S ::= A$
 $S ::= B$
 $A ::= x$
 $B ::= x$

Parser States for

①

$S ::= \cdot A$
$S ::= \cdot B$
$A ::= \cdot x$
$B ::= \cdot x$

②

$A ::= x \cdot$
$B ::= x \cdot$

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

10/10/2002
© 2002 Hal Perkins & UW CSE
D-37

Handling Reduce-Reduce Conflicts

- These normally indicate a serious problem with the grammar.
- Fixes
 - Use a different kind of parser generator that takes lookahead information into account when constructing the states (LR(1) instead of SLR(1) for example)
 - Most practical tools already use this information
 - Fix the grammar

10/10/2002
© 2002 Hal Perkins & UW CSE
D-38


Another Reduce-Reduce Conflict

- Suppose the grammar separates arithmetic and boolean expressions


```

expr ::= aexp | bexp
aexp ::= aexp * aident | aident
bexp ::= bexp && bident | bident
aident ::= id
bident ::= id
      
```
- This will create a reduce-reduce conflict


10/10/2002
© 2002 Hal Perkins & UW CSE
D-39



Covering Grammars

- n 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)
- n This is a *covering grammar*
 - n Includes some programs that are not generated by the original grammar
 - n Use the type checker or other static semantic analysis to weed out illegal programs later

10/10/2002 © 2002 Hal Perkins & UW CSE D-40



Coming Attractions

- n Constructing LR tables
 - n We'll present a simple version (SLR(0)) in lecture, then talk about extending it to LR(1)
- n LL parsers and recursive descent
- n Continue reading ch. 3

10/10/2002 © 2002 Hal Perkins & UW CSE D-41
