CSE431 – Translation of Computer Languages

- Bottom-Up Parsers

Doug Shook

Review

- We've seen two ways to construct top-down parsers
- What does a parser do?
- What kind of parse do top-down parsers generate?
- What are some of the problems with top-down parsers?

Bottom-Up Parsers

- Also known as LR(k) Parsers
  - What do we mean by LR(k)?
- Works from the bottom of the parse tree to the top
  - Terminals to start nonterminal
  - Replaces a rules right hand side with its left hand side
- Also known as *shift-reduce*
  - Shifts symbols onto the stack
  - Reduce symbols to nonterminals

Bottom-Up Parse

- Let's go through the process with the following example:

Start -> S$
S -> AC
A -> aBCd
| BQ
| λ
B -> bB
| λ
C -> c
| λ
Q -> q
| λ

"abbdc$"

Knitting

- The LR Parse process is said to resemble knitting
- Consider two needles:
  - Stack
  - Input stream
- A *shift* consists of moving symbols from input to stack
- A *reduction* consists of reversing a production

LR Parsing

- The actions we're expected to perform come from a table
  - This table is used to track *state* of the stack
- The next state as well as the action to be performed are given from the table
  - Current State
  - Next symbol from input

LR Parsing

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	3	2	8		8	8	accept	4	1	5		
1			11			4					14	
2		2	8	8	8	8				13		
3		2	8	8						9		
4						8						
5			10		7	10						6
6			6			6						
7			9			9						
8						1						
9			11	4							10	
10				12								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

4470485 2014/02/14 128.252.20.193

LR Grammars

- Grammars are LR(K) iff we can construct a parse table
  - Must be deterministic
- All LL grammars are or can be rewritten as LR
  - What about left recursion?
  - Common prefix?
- Are there any grammars that cannot be LR?

Exercises

- Write pseudocode for the parsing process
  - Solution in text

- Show the parse process for the following:

```
Start -> E $
E     -> plus E E
       | num
```

"plus num num $"

State	num	plus	$	Start	E
0	2	1		accept	3
1	2	1			5
2	reduce 3				
3			4		
4	reduce 1				
5	2	1			6
6	reduce 2				

Table Construction

- How do we construct the table?
  - That's the tricky part!
  - Multiple methods, some work better than others
    - Parse process is the same, regardless of the table
  - The *k* in LR(*k*) refers to lookahead used during *table construction*
    - LR(0) still uses lookahead during the parse

LR(0) Table Construction

- To make a reduction for E -> plus E E what needs to be true?
- To keep track of our progress, we'll use LR(0) items
  - Use • as a bookmark
- Examples:
  - • plus E E
  - plus • E E
  - plus E • E
  - plus E E •
- What would E -> λ look like?

LR(0) Table Construction

- The state machine for an LR parse table is referred to as a Characteristic Finite State Machine

- Each state is comprised of items
  - One kernel item
  - Closure of the kernel item

- Given Start -> • E $
  - Closure contains E -> • plus E E and E -> • num

  - If closure productions also have non-terminals next to the bookmark then must add closure of those items, too

LR(0) Table Construction

```
Start with item corresponding to beginning of
start state, add to worklist
while worklist is not empty
    fetch a state from list
    form state closure
    add transitions to new states based on
    initial symbols
    if bookmark is at the end of an item, mark
    accepting
```

LR(0) Table Construction

- Once the CFSM is constructed, we can generate the parse table
  - Each state becomes a row
  - Each symbol becomes a column
  - Transitions become?
  - Accepting states become?

Exercises

- Construct the CFSM and parse table for the following grammar:

```
S -> E $
E -> E + T
   | T
T -> a
   | ( E )
```

- Use your table to parse “a + ((a+a))$”
  - Then try “a + (a $”

Table Construction

- Let's try another (similar) example:

```
S -> E $
E -> E + T
   | T
T -> T * a
   | a
```

Conflicts

- What if there are two actions for a given state?
  - We want this to be deterministic....
- Shift-Reduce
  - Shift and reduce rule in the same state
- Reduce-Reduce
  - Two or more reduce rules in the same state
- Shift-Shift isn't possible
  - Why?
- We need something more powerful to resolve these

Conflicts

- How do conflicts occur?
- Two reasons:
  - One you should already know
  - Table building method is not powerful enough
    - Try using more lookahead
    - Try using a more powerful method
- It is impossible to tell what the reason of a conflict is based only on the conflict itself

Detecting Ambiguity

- Cannot be done systematically, but try this:

- Form CFSM, find inadequate states
  - Find a path to the inadequate states
    - What will this yield?
  - Attempt to find a single string which parses correctly under both actions
    - This is undecidable!

- Now you have two parse trees for the same string, and know which productions should be examined

Exercises

- The following grammar is ambiguous:

```
S -> E $
E -> E plus E
   |  num
```

- Try to generate the CFSM for this grammar and identify the inadequate states
  - Generate a string that proves ambiguity
  - See if you can fix it

Another Example

- Examine the following grammar and construct the CFSM:

```
S     -> Exprs $
Exprs -> E a
       | F b
E     -> E plus num
       | num
F     -> F plus num
       | num
```

- Are there any conflicts? If so, what is causing them?
- What should the parse for “num plus plus num a $” look like?

Another Example

- Would using more lookahead help us?

- Is this grammar LR(k)?