CSC 448: Compiler Design: 2015 Spring, Assignment #3

Last modified 2015 May 15

Student: Yuancheng Zhang

Purpose:

To go over theory the theory of LL(1) and LR(1) parsing.

Computing:

Please ssh into ctilinux1.cstcis.cti.depaul.edu, or use your own Unix machine.

Assignment:

1. **Improving `llParserMaker.cpp` (40 points)**
   (empty, see the cpp file)

2. **Questions from the book, chapter 5, pages 173-178 (30 points)**

Question 9

9. Recall that an LL(k) grammar allows *k* tokens of lookahead. Construct an LL(2) parser for the following grammar:

```
1  Stmt   → id ;
2          | id ( IdList ) ;
3  IdList → id
4          | id , IdList
```

LL(1)

First Set:

terminals:

First(id) = { id }

First(;) = { ; }

First(( ) = { (}

First( )) = {) }

non-terminals:

First(Stmt) = { id }

First(IdList) = { id }

Follow Set:

Follow(Stmt) = { $ }

Follow(IdList) = {) }

LL(1) Parsing Table

	id	;	(	)	$
Stmt	Stmt->id; Stmt->id(IdList);				
IdList	IdList->id IdList->Id,IdList				

Therefore, this grammer is not LL(1).

LL(2)

$First_2$(Stmt) -> { id; id(}

$First_2$(IdList) -> { id id, }

Follow(Stmt) -> { $ }

Follow(IdList) -> {) }

LL(2) Parsing Tabled

	id$	id;	id(	id,	*others*
Stmt	*error*	Stmt->id;	Stmt->id(IdList);	*error*	*error*
IdList	IdList->id	*error*	*error*	IdList->Id,IdList	*error*

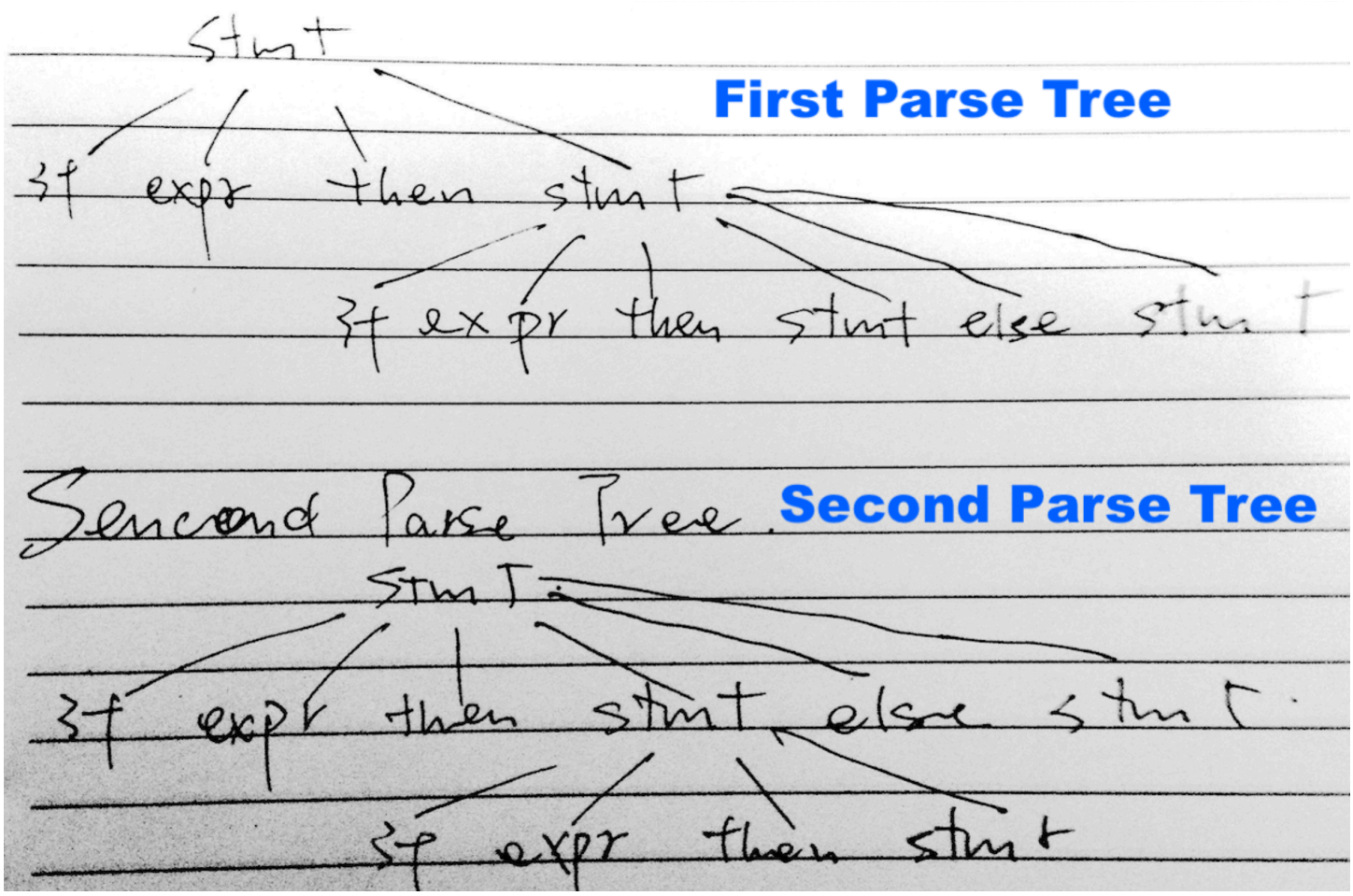
Therefore, this grammar is LL(2).

Question 10

10. Show the two distinct parse trees that can be constructed for

if expr then if expr then other else other

using the grammar given in Figure 5.17. For each parse tree, explain the correspondence of then and else.



Question 11

11. In Section 5.7, it is established that LL(1) parsers operate in linear time. That is, when parsing an input, the parser requires *on average* only a constant-bounded amount of time per input token.

    Is it ever the case that an LL(1) parser requires more than a constant-bounded amount of time to accept some particular symbol? In other words, can we bound by a constant the time interval between successive calls to the scanner to obtain the next token?

No, we can't. When the parser accepts a token, we don't know how many times the stack will pop or push. If each stack action costs a constant time, we don't know how long it takes before obtain the next token.

3. **Questions from the book, chapter 6, pages 224-233 (30 points)**

Question 12

12. Given the claim of Exercise 11, explain why the following statement is true or false:

There is no LR(k) grammar for the language

$$\{0^n1^n\mathsf{a}\} \cup \{0^n1^{2n}\mathsf{b}\}.$$

```
1  Start  → Single  a
2           | Double  b
3  Single → 0  Single  1
4           | 0  1
5  Double → 0  Double  1  1
6           | 0  1  1
```

It's true. This language is inherently nondeterministic. Only deterministic context-free grammar can be parsed by LR parser. There is no LR(k) grammar for this.

Question 13

13. Discuss why is it not possible during LR(0) construction to obtain a shift/reduce conflict on a nonterminal.

If we shift a nontermianl X onto the top of the stack, it means that X is a handle, which is a reduction that also allows further reductions back to the start symbol. X must be reduced definitely. Therefore, there is no shift/reduce conflict on a nonterminal.

Question 40

40. Recall the **dangling else** problem introduced in Chapter 5. A grammar for a simplified language that allows conditional statements follows:

```
1  Start → Stmt $
2  Stmt  → if e then Stmt else Stmt
3        |  if e then Stmt
4        |  other
```

Explain why the grammar is or is not LALR(1).

This grammar is not not LALR(1).
This grammar is ambiguous. Any ambiguous grammar can not be LR.
If **if e then Stmt** is on the stack and **else Stmt** is unprocessed input stream, we don't know whether if e or Stmt is the handle. There is a shift/reduce conflict.