CS143 Handout 17
Summer 2012 July 11th, 2011

Written Set 2: Parsing

This written assignment will give you a chance to play around with the various parsing algorithms we've talked about in class. Once you've completed this written assignment, you should have a much deeper understanding of how the different parsing algorithms work and will be able to reason about their strengths and weaknesses.

For the purposes of this handout, you can assume that all context-free grammars are "clean" grammars. That means that

1. Every nonterminal can eventually derive a string of terminals. In other words, we won't be dealing with CFGs like S → A, where the nonterminal A can't be further expanded, or S → Sa, where S can never be fully expanded.

2. Every nonterminal is reachable from the start symbol. That is, grammars like

    S → a | bS
    X → c

    are not allowed.

Due Wednesday, July 18th at 5:00PM

Submission Instructions

There are two ways to submit this assignment:

1. Submit a physical copy of your answers in the filing cabinet in the open space near the handout hangout in the Gates building. If you haven't been there before, it's right inside the entrance labeled "Stanford Engineering Venture Fund Laboratories."
2. Send an email with an electronic copy of your answers to the staff list at cs143-sum1112-staff@lists.stanford.edu. Please include the string [WA2] somewhere in the subject line so that it's easier for us to find your submission.

This assignment is worth 10% of your grade in this course, and all problems are weighted evenly. Good luck!

Warm-up exercises. We won't be grading either of the next two problems; they're here purely for you to get a chance to play around with the material before diving into the trickier questions in this problem set. While they aren't mandatory, I **strongly** suggest that you play around with them to make sure that you understand how the different parsing algorithms work.

Warm-up exercise 1: LL parsing

In class, we explored a simple programming language with the following structure:

Program	→	Statement
Statement	→	**if** Expression **then** Block
	→	**while** Expression **do** Block
	→	Expression**;**
Expression	→	Term **=> identifier**
	→	**isZero?** Term
	→	**not** Expression
	→	**++ identifier**
	→	**−− identifier**
Term	→	**identifier**
	→	**constant**
Block	→	Statement
	→	{ Statements }
Statements	→	Statement Statements
	→	ε

Trace through how an LL(1) parser for this grammar would parse the program

```
while isZero? identifier do {
        if not isZero? identifer then
                constant => identifier;
}
```

When tracing through the parser steps, you don't need to explicitly construct FIRST and FOLLOW sets. Try reasoning your way through how the parser would operate to see if you understand why all of the given productions would be applied at each point.

Warm-up exercise 2: LR parsing (courtesy of Julie Zelenski).

Before jumping into the mechanics of creating and analyzing LR parsers, you might want to start with an exercise to verify you have an intuitive understanding of how LR parsing operates. Consider the following grammar for a simplified C-like function prototype. The terminals are `{T_Ident T_Int T_Double T_Char ( ) ; , }`. These tokens will be recognized by the scanner and passed to the parser.

```
Proto      →   Type T_Ident ( ParamList ) ;
Type       →   T_Int | T_Double | T_Char
ParamList  →   ParamList , Param | Param
Param      →   Type T_Ident
```

i. Trace through the sequence of shift and reduce actions by an LR parser on the input: `double Binky(int winky, char dinky);` Rather than building the configurating sets and LR table and mechanically following the algorithm, try to use your understanding of the shift-reduce technique to guide you.
ii. Why does the parser not attempt a reduction to `Param` after pushing the first sequence of type and identifier onto the parse stack? Doesn't the sequence on top of the stack match the right side? What other requirements must be met before a reduction is performed?
iii. Verify that the grammar is LR(0), and thus can be parsed by the weakest of the LR parsers.

The remainder of these problems will be graded and should be completed and turned in.

Problem 1: LL(1)

Suppose that we want to describe Java-style class declarations like these using a grammar:

```
class Car extends Vehicle
public class JavaIsCrazy implements Factory, Builder, Listener
public final class President extends Person implements Official
```

One such grammar for this is

(1) C → P F **`class`** **`identifier`** X Y
(2) P → **`public`**
(3) P → ε
(4) F → **`final`**
(5) F → ε
(6) X → **`extends`** **`identifier`**
(7) X → ε
(8) Y → **`implements`** I
(9) Y → ε
(10) I → **`identifier`** J
(11) J → **`,`** I *(note the comma before the I)*
(12) J → ε

Your job is to construct an LL(1) parser table for this grammar. For reference, the terminals in this grammar are

```
public final class identifier extends implements , $
```

Where **`$`** is the end-of-input marker, and the nonterminals are C P F X Y I J.

i. Compute the FIRST sets for each of the nonterminals in this grammar. Show your result.
ii. Compute the FOLLOW sets for each of the nonterminals in this grammar. Show your result.
iii. Using your results from (i) and (ii), construct the LL(1) parser table for this grammar. When indicating which productions should be performed, please use our numbering system from above. Show your result.

Problem 2: LL(1) Conflicts

Consider the following grammar, which describes lists of nouns:

```
S → noun | noun and noun | M, noun, and noun
M → M, noun | noun
```

For example, this grammar can generate (among others) the strings

```
noun
noun and noun
noun, noun, and noun
noun, noun, noun, and noun
```

For reference, the terminals in this grammar are

`noun    ,    and    $`

And the nonterminals are S and M.

i. This grammar is not LL(1). Identify the conflicts in the grammar that make it not LL(1) and explain each.
ii. Rewrite the grammar so that it is LL(1). To prove that your grammar is LL(1), construct an LL(1) parsing table for it. You do not need to explicitly show the FIRST or FOLLOW sets, thought it might be useful to compute them.

Problem 3: LL(1) and LR(0)

As mentioned in lecture, LR(1) grammars encompass all LL(1) grammars, plus many others. However, LL(1) and LR(0) grammars are incomparable – neither is a subset of the other.

i. Give an example of a grammar that is LL(1) but not LR(0), and explain why.
ii. Give an example of a grammar that is LR(0) but not LL(1), and explain why.

Problem 4: SLR(1) Parsing

Below is a context-free grammar for strings of balanced parentheses:

(1) S → P
(2) P → **(**P**)** P
(3) P → ε

For example, this grammar can generate the strings

```
()
(()())()
((()))()(()())
```

In this question, you will construct an SLR(1) parser for this grammar. For reference, the terminal symbols are

```
(    )    $
```

where **$** is the end-of-input marker, and the two nonterminals are S and P. S is the special "start" nonterminal, so you don't need to add this yourself.

i. Construct the LR(0) configurating sets for this grammar. Show your result. As a hint, there are six total configurating sets. Note that when dealing with the production P → ε, there is only one LR(0) item, which is P → ·
ii. Compute the FOLLOW sets for each nonterminal in the grammar. Show your result.
iii. Using your results from (i) and (ii), construct an SLR(1) parsing table for this grammar. Show your result. Note that the LR(0) item P → · is a reduce item.
iv. Identify at least one entry in the parsing table that would be a shift/reduce conflict in an LR(0) parser, or explain why one does not exist.
v. Identify at least one entry in the parsing table that would be a reduce/reduce conflict in an LR(0) parser, or explain why one does not exist.

Problem 5: Manual Conflict Resolution

In class, we talked about a CFGs for regular expressions. One such CFG is as follows:

1. S → R
2. R → RR
3. R → R | R *(note that | is a terminal symbol in the grammar)*
4. R → R*
5. R → ε *(note that ε is a terminal symbol in the grammar)*
6. R → a
7. R → (R)

Here are the LR(0) configurating sets for this grammar:

(1) S → ·R R → ·RR R → ·R | R R → ·R* R → ·(R) R → ·ε R → ·a	(2) S → R· R → R·R R → R· | R R → R·* R → ·RR R → ·R | R R → ·R* R → ·(R) R → ·ε R → ·a	(3) R → RR· R → R·R R → R· | R R → R·* R → ·RR R → ·R | R R → ·R* R → ·(R) R → ·a R → ·ε	(4) R → (·R) R → ·RR R → ·R|R R → ·R* R → ·(R) R → ·a R → ·ε	(5) R → (R·) R → R·R R → R·|R R → R·* R → ·RR R → ·R|R R → ·R* R → ·(R) R → ·a R → ·ε	(6) R → R |· R R → ·RR R → ·R | R R → ·R* R → ·ε R → ·a R → ·(R)
(7) R → R | R· R → R·R R → R· | R R → R·* R → ·RR R → ·R|R R → ·R* R → ·ε R → ·a R → ·(R)	(8) R → R*·	(9) R → ε·	(10) R → a·	(11) R → (R)·	

Here is an SLR(1) parsing table for the above configurating sets. This grammar is not SLR(1), so there are several conflicts. In those cases, all actions are listed.

	a	ε	(	)	|	*	$	R
(1)	s10	s9	s4					s2
(2)	s10	s9	s4		s6	s8	acc	s3
(3)	r2/s10	r2/s9	r2/s4	r2	r2/s6	r2/s8	r2	s3
(4)	s10	s9	s4					s5
(5)	s10	s9	s4	s11	s6	s8		s3
(6)	s10	s9	s4					s7
(7)	r3/s10	r3/s9	r3/s4	r3	r3/s6	r3/s8	r3	s3
(8)	r4	r4	r4	r4	r4	r4	r4	
(9)	r5	r5	r5	r5	r5	r5	r5	
(10)	r6	r6	r6	r6	r6	r6	r6	
(11)	r7	r7	r7	r7	r7	r7	r7	

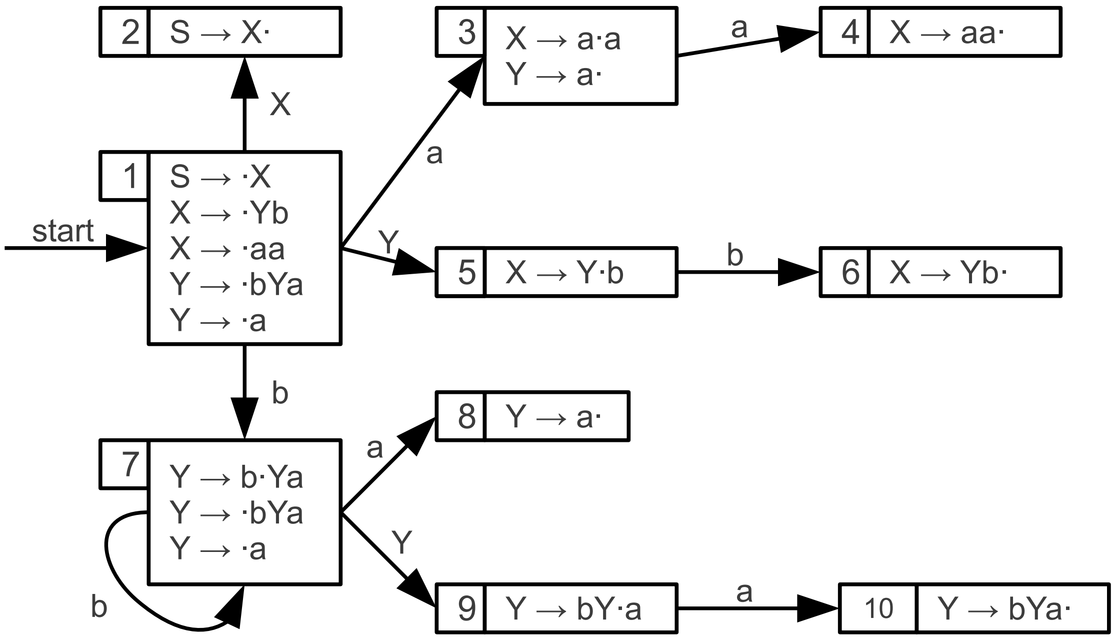
i. Let's suppose that we want to resolve the conflicts in this grammar by using our knowledge of the precedence rules for regular expressions. In particular, we know that union ("or") has lowest precedence and is left-associative, concatenation has middle precedence and is left-associative, and Kleene closure ("star") has highest precedence and is left-associative. Given these rules, explain how you would update the above SLR(1) parser table to resolve all of the conflicts in this grammar.
ii. LR(1) is a much stronger parsing algorithm than SLR(1). Would using an LR(1) parser instead of the SLR(1) parser resolve the ambiguities? Why or why not?

Problem 6: LALR(1)-by-SLR(1)

Here is a (very contrived!) grammar that is known not to be SLR(1):

$S \rightarrow X$
$X \rightarrow Yb \mid aa$
$Y \rightarrow a \mid bYa$

Here is the associated LR(0) automaton for this grammar:



In this question, you'll get to see how the LALR(1)-by-SLR(1) algorithm works in practice.

i. Why is this grammar not SLR(1)?
ii. Using the LR(0) automaton, construct the augmented grammar that you will use in the algorithm. Show your result.
iii. Compute the FOLLOW sets for every nonterminal in the grammar. Show your result.
iv. Using these FOLLOW sets and the LR(0) automaton, construct the LALR(1) lookaheads for each reduce item in the automaton. Show your results. Note that there are a total of 6 reduce items in the automaton.
v. Is this grammar LALR(1)? Why or why not?
vi. Is this grammar LR(1)? Why or why not?