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

   $S \rightarrow A$

   where the nonterminal A can't be further expanded, or

   $S \rightarrow Sa$

   where S can never be fully expanded.

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

   $S \rightarrow a \mid bS$
   $X \rightarrow c$

   are not allowed.

Due Monday, July 18th at 5:00PM

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. Use the example on page 2 of the LR(0) handout as a model.

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 Zergling extends Unit implements Rush
```

One such grammar for this is

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

Your job is to construct an LL(1) parser table for this grammar. We've already added the new start state to the top of the grammar, so there's no need to introduce your own.

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

S 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(*k*)

In class, we talked about LL(1) parsing, in which we scan the tokens from left-to-right, tracing out a leftmost derivation, at each point using one token of lookahead to guide our search. If you'll recall, our LL(1) parser uses knowledge of the current nonterminal and our one token of lookahead to determine which production of that nonterminal should be performed. We say that a grammar is LL(1) if given any nonterminal A, there was a unique production to be performed based on the next token of lookahead.

This definition can be generalized to **LL(*k*)**, in which we have *k* tokens of lookahead. A grammar is LL(*k*) if given the current nonterminal and knowledge of the next *k* tokens of lookahead, it is possible to uniquely determine which production should be performed.

i. Give an example of a grammar that is LL(2) but not LL(1). You must explain why your grammar meets these criteria, but you don't need to formally prove it.
ii. Prove that any LL(1) grammar is LL(2).

Problem 3: LL(0)

An **LL(0) parser** is a parser similar to an LL(1) parser, except that there are **zero** tokens of lookahead to determine which production to use.

Describe the set of grammars that can be parsed with an LL(0) parser. As a hint, LL(0) is an extremely weak method, and is never used in compilers.

Problem 4: Left Factoring

Recall that a set of productions for a nonterminal A can be *left factored* if they share a common, non-empty prefix. For example, the productions

$A \rightarrow E + T$
$A \rightarrow E$

are left-factorable because the productions can be written as

$A \rightarrow EY$
$Y \rightarrow + T$
$Y \rightarrow \varepsilon$

In many (but not all) cases, grammars that are left-factorable are not LL(1), but their factored equivalents are LL(1). This question asks you to consider why this is.

i. Explain, at a high level, why many grammars that are left-factorable are not LL(1).
ii. Give an example of a grammar that is left-factorable but is still LL(1). Make sure to explain why your grammar is LL(1).
iii. Give an example of a grammar that is left-factorable that is LL(2) but not LL(1). Make sure to explain why your grammar meets these criteria.

Problem 5: SLR(1) Parsing

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

(1) $S \rightarrow P$
(2) $P \rightarrow$ **(**P**)** P
(3) $P \rightarrow \varepsilon$

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 $P \to \varepsilon$, there is only one LR(0) item, which is $P \to \cdot$
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 $P \to \cdot$ 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 6: The Power of LR Parsers

In class we discussed the relative power of the various bottom-up parsing algorithms. In this question, you will formally establish this.

i. Prove that any LR(0) grammar is SLR(1).
ii. Prove that any SLR(1) grammar is LALR(1).
iii. Prove that any LALR(1) grammar is LR(1).

Hint: For part (i), recall that a grammar is SLR(1) if it is possible to construct action and goto tables using the SLR(1) algorithm without encountering any conflicts. Think about what types of conflicts could occur in this table, then show that if the grammar in question is LR(0) that these conflicts could not occur. Use similar logic in parts (ii) and (iii).

Problem 7: Manual Conflict Resolution

In class, we talked about a context-free grammar 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)·	

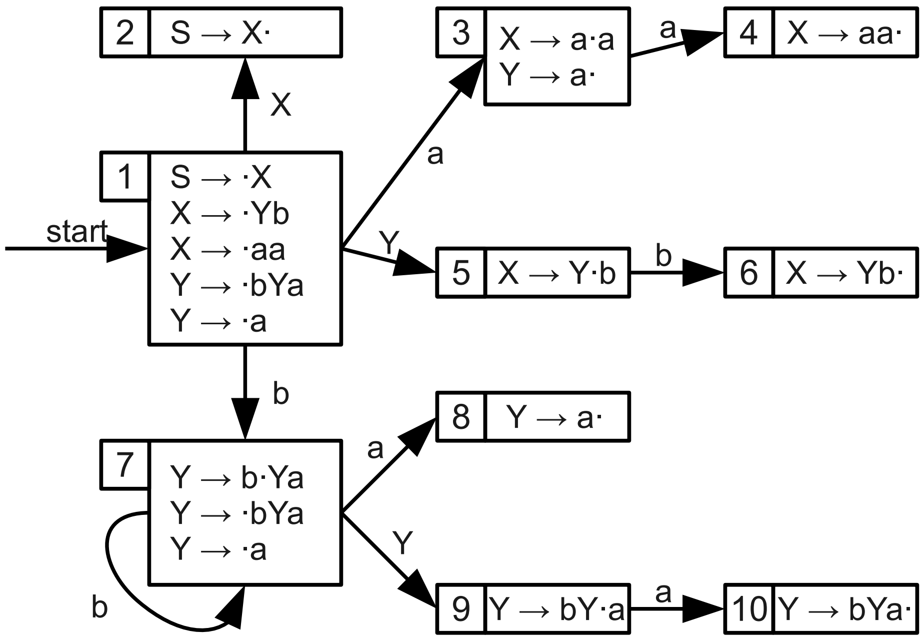
i. Construct an SLR(1) parse table for this grammar. The grammar is **not** SLR(1) and so there will be conflicts in your table. When this happens, list all actions that should be taken in a given state. Please use our numbering for the different reductions and the LR(0) configurating sets.
ii. LR(1) 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?
iii. 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 disjunction ("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, update the SLR(1) parser table you created in part (i) to resolve all of the conflicts in this grammar.

Problem 8: 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?

Submission Instructions

The submission instructions are the same as for last week. You may either email your solution to cs143-sum1011-staff@lists.stanford.edu, or drop off a hard copy in the filing cabinet in the Gates open space under the entrance marked "Stanford Engineering Venture Fund Laboratories."

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

Good luck!