Wellesley College ◊ CS301 Compiler Design ◊ November 11, 2000
Handout #11

PROBLEM SET 6

Due on Friday, November 17

READING

- Appel, Chapters 3 and 4 (these will help a lot with all the problems).
- The ML-Yacc User's Manual (accessible from the CS301 home page).

OVERVIEW

The purpose of this problem set is to give you some experience with theoretical and practical aspects of LL and LR parsing. The assignment has three pencil and paper problems worth 60 points, and a programming assignment worth 90 points.

COLLABORATION DETAILS

On this assignment, you should work in pairs such that you are working with someone you have not yet worked with this semester in CS301. You will be working with the same partner on both PS6 and PS7.

Note that you should not simply split up Problems 1, 2, and 3 between members of a team. Ideally, such problems should be done together. Even if one person "takes the lead" on a problem, the other person should check all the details of the first person and "sign off" on a problem before it is turned in.

SUBMISSION DETAILS

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 5pm on the due date. The hardcopy packet should include the following:

- Your problem set header sheet.
- Your pencil and paper answers to Problems 1, 2, and 3.
- Your final version of the files `Bobcat.lex` and `Bobcat.yacc` from Problem 4.

Your softcopy submission should consist of your local versions of the `bobcat/myparser` directory described below.

PROBLEM 1 [20]: NULLABLE, FIRST, AND FOLLOW

Exercise 3.6 in Appel.

PROBLEM 2 [20]: SLR vs. LALR(1)

Exercise 3.13 in Appel. Do the exercise in three steps:

1. Show that the grammar is not SLR by developing a parsing table from the a state diagram using SLR items and showing that the parsing table has conflicts.
2. Show that the language is LR(1) by doing the same exercise based on a state diagram using LR(1) items, and showing the resulting table has no conflicts.
3. Based on the description on p. 64 of Appel, show that the LR(1) parsing table can be compressed into an LALR(1) parsing table without introducing any conflicts.

PROBLEM 3 [20]: LL(1) vs. LALR(1)

Exercise 3.14 in Appel.

PROBLEM 4 [90]: AN ML-YACC GENERATED PARSER FOR BOBCAT

In this problem, you will develop a LALR(1) parser for Bobcat using ML-Lex and ML-Yacc. In particular, the goal of this problem is develop a Bobcat.lex file specifying a `lexer` for Bobcat and a `Bobcat.yacc` file specifying a parser for Bobcat. Your parser should return an abstract syntax tree for every syntactically valid Bobcat program. The syntax of Bobcat is fully specified in Section 2 of the Bobcat Reference Manual (Handout #10).

To begin this problem, you should familiarize yourself with ML-Yacc by studying Section 3.4 and Chapter 4 of Appel. You may also want to study or experiment with the sample ML-Yacc parsers we studied during tutorial. These can be found in the cs301 CVS hierarchy in the `simp` directory. (Perform `cvs update -d` in your local cs301 directory to grab these.) You are also encouraged to at least skim the on-line ML-Yacc user manual. You will probably consult this several times while working on this problem, and it's nice to have a sense for its contents before you begin.

Begin the coding part of your problem by installing the directory `bobcat/myparser` in your local cs301 directory hierarchy. You can do this by executing the following in a Unix shell:

```
cd your-local-bobcat-directory
cvs update -d
```

The `bobcat/myparser` has contents similar to the `bobcat/parser` directory you used for PS5. It is a separate directory to avoid confusion between the parser you are developing and the parser with which you were provided for PS5. In addition to several `.cm` files, the `myparser` directory contains the following:

- `Bobcat.lex`: This is a skeleton lexer specification that you should flesh out as part of this problem.

- `BobcatLexDefs.sml`: This file contains many handy functions that can considerably simplify your lexer definition. You should study this file before fleshing out `Bobcat.lex`. The `Bobcat.lex` skeleton file is configured in such a way that all functions in `BobcatLexDefs.sml` are imported into it and can be used without qualification.

- `Bobcat.yacc`: This is a skeleton parser specification that you should flesh out as part of this problem. The set of terminals has already been specified for you. What you need to specify are: (1) the set of nonterminals; (2) the productions of the grammar; and (3) precedence and associativity for terminals (and possibly productions).

- `ParserTest.sml`: This file contains code for testing your parser. You may want to edit the list of sample files that are used to test the parser.

- `Parser.sml`: This file contains code that glues together the structures automatically generated by ML-Lex and ML-Yacc. You need not study this file. But you may need to use two functions exported by the `Parser` structure in this file:

  `parseString: string -> AST.prog`
  Returns the abstract syntax tree for the Bobcat program specified by the literal string argument.

  `parseFile: string -> AST.prog`
  Returns the abstract syntax tree for the Bobcat program that is the contents of the file whose name is the given string.

It is recommended that you approach this problem in the following stages:

Stage 1: The Lexer

Flesh out the lexer specification in `Bobcat.lex`. Your specification can be very short if you make full use the auxiliary functions in `BobcatLexDefs.sml`.

In your specification, you will be using the token datatype automatically generated by ML-Yacc from the specification of terminals. Every token has components that indicate the leftmost and rightmost positions of the characters from which it was constructed. In this implementation, each position is a pair of integers: the first represents the line number, and the second represents a character on that line. We will refer to a pair of such positions as the *extent* of the token.

Non-value-bearing tokens are created by invoking the name of the terminal as a constructor on two position values that indicate the leftmost and rightmost positions of the token. Value bearing tokens are created by by invoking the name of the terminal as a constructor on three arguments: the value followed by the leftmost and rightmost position values. Here are some examples of constructor invocations that create tokens:

`ELSE((3,4),(3,7))` creates an ELSE token covering characters 4 through 7 on line 3.

`INTLIT(301, (23,17), (23,19))` creates an integer literal token with value 301 constructed from characters 17 through 19 on line 23.

Most tokens are on a single line. The one exception is string literal tokens, which may span several lines.

The `BobcatLexDefs` file contains several functions that construct appropriate tokens using the `yytext` and `yypos` information available in the lexer. You should be able to create all Bobcat tokens by calling one of these functions. You may wish to study the solutions to Problem Set 2 (in preparation) for an example of a lexer specification based on similar auxiliary definitions. I will try to post the solution code for Problem Sets 1 – 4 to the cvs-controlled directory `cs301/kittty-solutions` in the near future.

Stage 2: The Core Parser

Flesh out the parser specification `Bobcat.yacc`, assuming that all programs have the form

`let` *{GlobalDecl}* `in` *{Exp[;]}* `end`

(The syntactic sugar for programs presented in Section 2.3 of the Bobcat Reference Manual is tricky to implement, and should be left for later.)

To accomplish this task, you will need to extend the set of non-terminals in Bobcat.yacc,

implement productions for all Bobcat expressions and declarations, and add associativity and precedence declarations. It helps to do this part in small chunks. That is, pick one or two expressions, write productions for them, test the resulting parser, and continue with more expressions.

You should process your specification by invoking ML-Yacc as follows within the Unix shell

```
ml-yacc Bobcat.yacc
```

This generates a number of files that are used by other code within the `myparser` directory. You should perform the following in the SML interpreter after compiling the specification:

```
CM.make'("Parser.cm");
```

Keep in mind that any time you change the `Bobcat.yacc` file, you must repeat both of the above steps. After you have "made" the parser in SML, you can test it manually using `Parser.parseString` and `Parser.parseFile`, or automatically via `ParserTest.test()`. The former are helpful early in the game, when you are still implementing many of the features. The latter is helpful later in the game, after you have implemented most of the features.

It will be common for you to encounter shift/reduce and reduce/reduce conflicts reported by ML-Yacc. Study the `Bobcat.yacc.desc` file to determine the source of the conflicts. You should remove all such conflicts. That is, you should not rely on the "default" way that ML-Yacc handles the conflicts, even if it does the "right thing" in the cases you care about.

For this stage, it is possible to use associativity and precedence declarations to resolve all conflicts, though sometimes you need to use them in a tricky way. However, you are welcome to explore the alternative approach of rewriting grammar rules rather than relying on these declarations.

Stage 3: Program Desugarings

Extend your parser with the two desugarings for top-level programs discussed on Section 2.3 of the Bobcat Reference Manual. It turns out that these desugarings are rather challenging to implement because they introduce some potential ambiguities that do not arise otherwise. In particular, a single *Exp* can be considered as a program. If it is a let expression, should the let be viewed as the first keyword of a program or the first keyword of an expression? It turns out that it is always safe to assume that the first `let` is part of the program syntax and not the expression syntax. You should embed this assumption into your grammar.

This part is very tricky. As far as I can tell, it requires rewrites to the grammar and cannot be accomplished just with associativity and precedence declarations. You are likely to find that rewriting grammar rules removes some conflicts but introduces others. Studying the `Bobcat.yacc.desc` files is very important in this stage for figuring out what's going on.

It has been my experience with ML-Yacc that elegant specifications using lots of abstraction are difficult to come by. In particular, introducing abstractions can often introduce new ambiguities. In some cases, it seems to be a good idea to replicate big chunks of productions in order to avoid ambiguities.

EXTRA CREDIT [50]: A BOBCAT EVALUATOR

Implement an evaluator for Bobcat programs. The evaluator will be similar to the Kitty evaluator from Problem Set 1, but needs to correctly implement functions and those aspects of Bobcat that differ from Kitty. Here are some things to keep in mind:

4. In Kitty, an expression can evaluate either to an integer or to no value. In Bobcat, an expression can evaluate to an integer, character, or boolean value, or to no value.

5. You can simplify the evaluator by only evaluating expressions that are type correct. In particular, for a type correct expression, you can maintain a single global variable environment and a single global function environment in addition to a local variable environment.

6. You will need to implement Bobcat's standard library functions. There are many ways to do this. One approach is to maintain a table of implementations for all standard library functions. Another approach is to maintain such a table only for the primitive library functions, and to interpret calls to the non-primitive functions based on their definitions in the "standard prelude".

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS301 Problem Set 6
Due Friday, November 17, 2000

Names of Team Members:

Date & Time Submitted:

Soft Copy Directory:

Collaborators (*any teams collaborated with in the process of doing the problem set)*:

*In the **Time** column, please estimate the total time each team member spent on the parts of this problem set. (Note that spending less time than your partner does not necessarily imply that you contributed less.) Please try to be as accurate as possible; this information will help me to design future problem sets. I will fill out the **Score** column when grading your problem set.*

Part	**Time For** _____ **(Member #1)**	**Time For** _____ **(Member #2)**	**Score**
General Reading			
Problem 1 [20]			
Problem 2 [20]			
Problem 3 [20]			
Problem 4 [90]			
Extra Credit [50]			
Total			