Programming Exercise 3: Parser Generators

1 Introduction

The purpose of a parser generator is to create a parser from a language specification. A language specification is much easier to create and maintain than a hand-written parser, which is the main reason for using parser generators.

There are a wide variety of parser generators available. Most available tools generate LALR parsers, but there are general LR(k) and LL(k) parser generators available as well. In this exercise you will use `bison`, an LALR(1) parser generator.

2 Using `bison`

In order to use `bison` you will have to have the `prog/gnu` module loaded. At a Unix prompt, type `module list`, and see if `prog/gnu` is listed. If it's not listed, type `module add prog/gnu` and `module initadd prog/gnu` to load the module in the current shell and the next time you log in.

Full documentation for `bison` is available as an info document. To view the documentation, start `emacs`, type `C-h i`, and select the entry entitled `bison`. The Solaris AnswerBook contains documentation for `yacc`, which is very similar, but not identical, to `bison`.

3 The Generated Parser

When you compile a parser specification with `bison`, a function named `yyparse` is created. This function in turn calls `yylex` to retrieve tokens from the input, and `yyerror` to report errors.

You can provide `yylex` by updating the rules in `scanner.l` to match the ones you wrote in the previous exercise. A version of `yyerror` is already supplied in `parser.y`.

4 Requirements

You are to write the specifications for expressions, conditions and function definitions. Make sure that both children of an operator node have the same type. You may need to insert `IntegerToReal` nodes in some cases, to convert integers to floating-point numbers.

You also need to augment the grammar with error productions. After an error occurs, parsing of statements should be resumed after the next semicolon. You may insert other error productions if you want to.

Hand in the following:

- A listing of `parser.y` with your changes clearly marked.
- Listings of any other files you have modified, with your changes clearly marked.
- Answers to the questions in the next section.

5 Questions

Question 1 Construct the canonical set of LR(0) items for the following grammar. Can the grammar be used to construct an SLR parser? If not, explain why and construct a new grammar that accepts the same language and can be used to construct an SLR parser.

```
block   :   BEGIN decs ';' stmts END
        ;
decs    :   DEC
        |   DEC ';' decs
        ;
stmts   :   STMT
        |   STMT ';' stmts
        ;
```

Question 2 Show how an LR parser parses the string `1 + (2 - 3)` using the grammar below. Assume that `NUMBER` is the token returned for all numeric constants. Demonstrate each step in the parsing process.

```
expr    :   expr '+' term
        |   expr '-' term
        |   term
        ;
term    :   term '*' factor
        |   term '/' factor
        |   factor
        ;
factor  :   NUMBER
        |   '(' expr ')'
        ;
```

Question 3 What is the difference between an LR(0) and an LR(1) parser. Make up an example grammar and input to demonstrate the difference in operation.

6 Extra Credit Work: LR Parser Generator

Write a program that can read a grammar from a file and construct parsing tables for that grammar. Your program must construct tables sufficient for parsing using SLR(1), LALR(1) or LR(1). It must be capable of printing the sets of LR items, FIRST and FOLLOW sets, lookahead sets and any other information needed in the parsing process.

Hand in your program code and some grammar examples.