

Project 4 Programming Languages and Systems Concepts ACSI 511

Fall 2002

William A. Maniatty
Department of Computer Science
University at Albany
Albany, NY

November 11, 2002

1 Administrative Information

This project is due (tentatively) due Monday, December 8, 2002 **at the start of class**. Each student is expected to do their own work and their own debugging.

You should use your account on eve.albany.edu (or the AC Unix machines) to do this project. Project specific queries should be first be directed to Zhaohuei Yang, who will keep a FAQ. To save time for yourself and the TA, please check the faq before asking a question.

Questions or problem reports about eve should should be directed to helpdesk@albany.edu.

2 Project Overview

We are going to explore how to use automation to develop the front end to a compiler or interpreter. Last project, we did lexical analysis (scanning) and syntactic analysis (parsing) in this project for “Blaise” a simplified version of the Nicklaus Wirth’s Pascal programming language. You will be expected to extend the software you develop, if you don’t complete this assignment, you may have difficulty passing the course. We will extend the first project (I suggest using the automated parser that we generated). During this phase we will do intermediate code generation (semantic analysis) and type checking. We will handle type checking at this stage, so you might need to extend the data structure as needed for later phases.

For this project we will make the code actually run. You may try one of the following:

1. Add memory management to your quads (for subroutine invocation and controlling data layout) and implement an interpreter for your quads (for full credit).
2. Generate C source code, which can be compiled and run from your quads. The C source code should compile and run correctly, limited optimization is expected (constant folding, combining of adjacent labels with No Op instructions).
3. Generate Assembly code. You can pick from SPIM (the famous MIPS simulator by Larus), SPARC assembly code or Intel 80x86 assembly code. Correct implementations get 30% extra credit. If constant folding, dead code elimination and merging of adjacent labels are done, an extra 20% will be added for a total of 50% extra credit.

2.1 The Grammar

Recall, our grammar was based on the Pascal programming language, since it is relatively easy to parse using a LL(1) grammars. The presented grammar, however, is LALR(1) (as it has left recursion). The grammar is given below (in BNF Format), with terminals delimited by double quotes "

```
program ::= "program" "id" "(" identifier_list ")" ";"
        declarations
        subprogram_declarations
        compound_statement
        ". ."

identifier_list ::= "id" | identifier_list "," "id"
declarations ::= declarations "var" identifier_list ":" type ";" | epsilon
type = standard_type | "array" "[" "num" "." "num" "]" "of" standard_type
standard_type ::= "integer" | "real" | "boolean"
subprogram_declarations = subprogram_declarations subprogram_declaration ";" |
    epsilon
subprogram_declaration ::= subprogram_head declarations compound_statement
subprogram_head ::= "function" "id" arguments ":" standard_type ";" |
    "procedure" "id" arguments ";"
arguments ::= "(" parameter_list ")" | epsilon
parameter_list ::= identifier_list ":" type |
    parameter_list ";" identifier_list ":" type
compound_statement ::= "begin" optional_statements "end"
optional_statements ::= statement_list | epsilon
statement_list ::= statement | statement_list ";" statement
statement ::= variable "assignop" expression |
    procedure_statement |
    compound_statement |
    "if" expression "then" statement "else" statement |
    "while" expression "do" statement
variable ::= "id" | "id" "[" expression "]"
procedure_statement ::= "id" | id "(" expression_list ")"
expression_list ::= expression | expression_list "," expression
expression ::= simple_expression |
    simple_expression "relop" simple_expression
simple_expression ::= term | sign term | simple_expression addop term
term = factor | term "multop" factor .
factor = "id" | "id" "[" expression "]" | "id" "(" expression_list ")" |
    "not" factor | [sign] "number"
sign ::= "+" | "-"
```

2.2 Code Generation

During this phase, if you are not generating assembly code, you should not worry about putting variables in registers. For assembly code generation, you can use a simple scheme of “spilling” registers (select a victim register and copy its contents back to memory). Generating assembly code is likely to be more challenging than the other options (hence the extra credit). If you are generating SPARC assembly code, I suggest that you do *not* attempt to exploit register windows in order to complete the assignment. See Chapters 5 and 9 from Scott’s book for more details. Be careful to avoid emitting extraneous data into your generated code (like debugging output) as that may keep it from compiling. If you add data, make sure that it is in the format of a comment.

2.3 Software Design And Output

Your parser should accept valid Blaise programs and emit executable code. If you have an interpreter, you can either make it a stand alone program or integrate it in as a post compilation function. The interpreter may read in the commands in a binary format (to avoid having to parse the A program that gives informative error messages for incorrectly formatted input will be preferred. You will need to generate test cases to verify your program's behavior. For the predictive parser phase, recovery is not as strong a requirement (correct parsing of well formed input is more critical), however, phrase level recovery is desirable.

3 Electronic Deliverables

Electronic submission should be done using the submission script described on the FAQ page. Any binaries or scripts you create should compile and run on eve.albany.edu. The following items should be submitted electronically:

1. A file named `README` which contains:
 - (a) A list of files submitted
 - (b) How to compile any executables you might have
 - (c) The names and a brief description of all your test cases.
 - (d) Any known bugs (it is better to tell your users the bugs than for the users to discover them).
 - (e) How to run your programs (any special instructions or bugs)
2. The Source code for your parser with symbol table, intermediate code generation and type checking support, it should either be:
 - The source code for your LALR(1) shift/reduce parser in bison file named `parser.y` and The source code for your scanner in a flex file named `scanner.l`, or
 - The source code for your LL(1) predictive parser named `rdparse.c`.
3. Any other programs needed (you may wish to make a separate module for the symbol table code).
4. A few test cases to show that your program works. It might be a good idea to try a recursive function, call by value and call by reference as well as access to values in containing.
5. The makefile for your parser
6. Any other files needed to build and run your parser