CS4121 PascalJunior (PJ) Expression Interpreter Project

Due Date: Wednesday, January 29, 2025 at *11:59pm*

Purpose

The purpose of this project is to gain experience in giving meaning to a programming language by interpreting a subset of PascalJunior (PJ), which is instead a subset of Pascal with some minor changes. Specifically, you will interpret integer and string write operations, integer read operations, integer arithmetic and logical expressions and assignment statements. Essentially, this project interprets PJ using C/C++.

Project Summary

In this project, you will add actions to a parser and a scanner provided by me. You must add code to do the following:

1. Design a data structure to store integer variables declared in a PJ program and to track variable values (You can also use the provided symbol table structure for this task. See the "Additional Notes" section.)
2. Pass attributes of tokens from the scanner to the parser when needed.
3. Interpret string write statements with or without a formatting constraint.
4. Interpret integer write statements with or without a formatting constraint.
5. Interpret integer read statements.
6. Interpret integer expressions.
7. Interpret logic expressions.
8. Interpret boolean values as 0s or 1s.
9. Interpret assignment statements which assigns values to integer or boolean variables.

Requirements

Write all of your code in C or C++ . It will be tested on a Rekhi CS lab machine and MUST work there. You will receive no special consideration for programs which "work" elsewhere, but not on a CS machine. You may also use the remote Linux servers provided by IT (guardian and colossus).

Input. The file `PJProject1.tgz` contains the parser needed to begin this project. You will need to modify the actions in the project files `parser/PJParser.y` and `parser/PJScanner.l` to do this project. Currently, the actions in the parser just emit the rules that are reduced. (After you have a good understanding of this project, you can comment out line 19 of `parser/PJParser.y`. Then the rules will not be printed.) You also need to set up certain token attributes in the scanner. Sample input for this project is provided in the project directory `input`. You should **NOT** add, delete or change any syntax/lexical rules.

To run your interpreter, use the command

```
pjc <file>.pas
```

Type in your input if the program contains read statements.

Submission. Your code should be well-documented. You will submit all of your files, by tarring up your project directory using the command

```
tar -czf PJProject1.tgz PJProject1
```

Submit the file `PJProject1.tgz` via the CS4121 Canvas page. Make sure you do a 'make clean' of your directory before executing the tar command. This will remove all of the '.o' files and make your tar file much smaller.

I strongly recommend that you check your submission by downloading the submitted `PJProject1.tgz`, copying it to a temporary directory, and then using the following command to extract the files:

```
tar -xvf PJProject1.tgz
```

You can then compile and test your code one more time there.

Data Structures and Documentation I have provided several C data structures for those who will be programming in C. There are doubly linked list, symbol table and string manipulation routines in the directory `PJProject1/util`. The HTML Doxygen documentation for the provided code is in `PJProject1/Documentation/html/index.html`. You may ask me any questions regarding these routines. You will not likely need any of these structures now, but you may want to familiarize yourself with them. For those coding in C++, you may use STL.

Makefile Structure The Makefiles for the project are set up to automatically generate make dependences. In a particular directory (*e.g.*, `parser`), you may add new files for compilation by adding the source file name to the `SRCS` variable declaration on the first line of that directory's `Makefile`. For example, to add the file `newfile.c` to be compiled in the `parser` directory, change the first line of `parser/Makefile` from

```
SRCS = PJScanner.c PJParser.c
```

to

```
SRCS = PJScanner.c PJParser.c newfile.c
```

Nothing else needs to be done. Do not add source files to the root directory `PJProject1` as the make files assume there are no source files in that directory.

If you would like to add your own subdirectory (*e.g.*, `newdir`) to `PJProject1`, then change the line

```
DIRS = parser util
```

in `PJProject1/Makefile` to

```
DIRS = parser util newdir
```

and the line

```
LIBS = parser/libparser-g.a util/libutil-g.a
```

in `PJProject1/Makefile` to

```
LIBS = parser/libparser-g.a util/libutil-g.a newdir/libnewdir-g.a
```

Then, copy `util/Makefile` to `newdir/Makefile`. Finally, change the `SRCS` declaration in `newdir/Makefile` to contain only the source files in that directory and change the line

```
ARCHIVE = libutil$(ENV).a
```

to

```
ARCHIVE = libnewdir$(ENV).a
```

An Example

Given the following PJ program (7.addsub.pas):

```
PROGRAM exprAddSub;

VAR i,j,k,l : LONGINT;

BEGIN
    i := 1; k := 3; l := 4;
    j := i + l + k;
    WRITE('j = 1+3+4 =':15);
    WRITELN(j:4);
    l := j - 8 - k;
    WRITE('j - 8 - 3 = ':15);
    WRITELN(l:4)
END.
```

your expected output should be

```
j = 1+3+4 =   8
j - 8 - 3 =  -3
```

Additional Notes (Please read!)

The lexeme of `STRING` token as it is includes the leading and ending single quotes. The actual content of the string should not include the quotes.

PJ function `WRITE` will print an integer value or a string depending on the parameter type. `WRITELN` will append newline to the output. Formatted output `WRITE(x:5)` performs the same as C `printf("%5d", x)`. See `https://wiki.freepascal.org/Basic_Pascal_Tutorial/Chapter_2/Formatting_output` for more explanation of Pascal formatting output.

I have also provided you two functions getValue() and setValue() in PJParse.y. These two functions are useful to handle variable load and assignment, respectively. The two functions utilize a symbol table structure which is essentially a map from a string to an integer. You can also design and use your own structures to handle variables.

Interpret `TRUE` as an integer of value 1 and `FALSE` 0. All logical expressions should generate either 1 or 0.

All PJ programs can be compiled by `fpc` available in the lab machines. You can run the compiled executable to check the expected output of a PJ program. The only difference is that the course project outputs 1 or 0 for boolean values, while the `fpc`-generated executable outputs `true` or `false`.