CMPUT 415 - Project Checkpoint 1

Program Documentation

Mike Bujold

Dan Chui

James Osgood

Paul Vandermeer

October 5, 2012

The man page is available by typing *make man*

Introduction

The goals for this checkpoint are to have a functional compiler 'shell'. That is, it must be able to parse a program listing and report on any errors in the program in a way that is meaningful to the user so that corrective action may be taken. It must also recover from each error in such a manner that further reporting of errors is possible, up until the end of the program listing.
Currently, because of the limited functionality of the compiler, only the '-n' flag is operational when running *pal*. The '-n' option supresses the program listing from being output.

Handling of Lexical Units

Our lexer parses tokens in the following order: first, all single and multi-line comments are parsed, followed by reserved *pal* keywords, then numbers and variable names. Relational operators and other lexical units are parsed last. Any remaining unparsed tokens are presumed invalid, and the grammar handles reporting the error accordingly.

Syntax Error Reporting & Recovery

The bulk of the error-handling code is in *myerror.c*. The method we used for error handling is when an error is found, it is added to a linked list. As soon as we finish parsing a line, we output that list, clear it, and move onto the next line.
We also used a process of finding errors by creating what we call a "loop error"

catcher. A ”loop error” is an error in the code that bison knows about, but is unable to handle. It doesn’t reduce or shift, it stays and does the same thing over and over again. We catch these by detecting if it occurs 300 times in a single line (in theory, there shouldn’t be more than 300 errors on one line).
To recover from the loop error, we set yyparse to return a value of 1, which means there was an error and returns to the main function, where you do a while loop and detect if yyparse has returned a value greater than 0. After you output all errors caught, we go back to yyparse until it returns 0, that means it has parsed successfully. This is how we recover from a ”loop error”.
We took bison error tokens before semi-colons, ’begin’ tokens, ’end’ tokens and between brackets.
The errors are displayed by showing the program line, followed by all the errors in the line. It displays the line number, the character number in the line and the error that was detected (eg. syntax error). Underneath that line, it displays a ’caret’ underneath the location of the error.

Testing

To facilitate large scale testing and efficient reading of listing files, we created a python script to run the compiler on multiple tests and pipe the resulting output to a testing log.
When outputting program listings with the errors and line numbers appended, they are added to the program listing as comments, so the outputted program listing can be re-compiled an indefinite number of times.

Problems

When printing the program listing, we start printing the line number at the first noncommented line, but we do count the number of lines correctly. If there exists a comment block at the start of the program, it is printed as a single line. Since we do not strip out comments when listing the parsed file, the listing file may not be further parsable (that is, the listing may cause additional errors if run through the compiler). This is because of block comments, in a line where an error occurred, whose closed brackets interrupt the block comment of the error message added to the listing.