CMPUT 415 - Project Checkpoint 1

Program Documentation

Mike Bujold

Dan Chui

James Osgood

Paul Vandermeer

November 1, 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.

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

We created myerror.c to catch all the errors. 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.

The general syntactic error strategy we used was to place error symbols close to termnial symbols and near synchronizing symbols. By letting small statements and expressions, such as terms, assignments, and types, be reduced to errors, we can usually recover quickly from minor, localized errors. The placement of

error symbols by important separators, such as ';', 'end', ']', and ')', in both recursive and nonrecursive contexts, ensures general recoverability from larger, more disruptive errors.

Additionally, large grammar constructs, such as declaration parts and the entire program, can protect against the parser being unable to recover for strange errors. To better handle error reporting in statements, we let conditionals reduce to errors, thus making 'else', 'then', and 'do' significant symbols. As a stopgap, we implemented a loop counter that would kill the program if it became trapped in a seemingly infinite error reporting loop. The large amount of error symbols in the program has resulted in, what we think, a fairly comprehensive syntactic error reporter, although their are still some instances of open brackets and statements eating up a lot of the input; this number of error symbols have also cause fourteen shift/reduce conflicts.

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.

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.