CMPUT 415 - Project Checkpoint 2

Program Documentation

Mike Bujold

Dan Chui

James Osgood

November 9, 2012

The man page is available by typing *make man*

Introduction

Our goals for this checkpoint are threefold:

1. Implement fully functional and correct lexical analysis.
2. Implement correct syntactical analysis, including adequate and meaningful error-detection and recovery.
3. Symbol table and semantic analysis.

It is our ongoing intention to report on any errors in the program in a way that is meaningful to the user so that corrective action may be taken. The compiler 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.

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 stop-gap, 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 twelve shift/reduce conflicts.

Symbol Table

We used the glib-2.0 library to implement data structures within our compiler, namely the symbol table.
Our symbol table is a stack; for each new scope, we push a new level onto the stack. Lookup then looks at the top of the stack for localLookup, and begins its lookup from the top of the stack to the bottom for globalLookup. Each level of the symbol table stack is a hash table, where the symbol is retrieved using its identifier.

Symbols

Symbols are first divided into classes using an `object_class` enumerated type which determines what object class the symbol belongs to (eg. var, const, function, etc.). Objects can then have a `type_class` enumerated type which indicates what type of variable can be expected.

Symbols are stored in a `symbol_rec` struct, which is composed of a symbol `name` field, an `object_class` field, and a `oc_descriptions` union, which contains further parameters based on which object class the record contains. The descriptions for each object class contain data that pertains to each class. For instance, the description for a `OC_CONST` object would contain the value of the constant, the description for a `OC_FUNCTION` object would contain a pointer to the list of parameters and the return type for that object.

Type Classes are a special object case that have a further structure with data pertinent to that variable. For instance, type class `TC_SCALAR` has two extra fields in it's description, one for length and another to hold a pointer to the list of elements. Type symbols are special cases of symbols; they have a type_attr description, which defines the type itself, and this type symbol is pointed to by other symbols, showing what type they are.

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.