CMPUT 415 - Project Checkpoint 2
Program Documentation

Matthew Low
Anthony Galati
Stevan Clement
Mike Bujold

9 November 2011

The man page is available with the command *less ./documentation/pal.1*

1 Handling of Lexical Units

The lex program works in the usual way to match tokens, the only interesting part being how it handles comments that are allowed to spread across multiple lines. To do this, the lexer switches into a 'comment' state where it grabs everything until it sees the end of the comment. This state change was deemed necessary for the line counting to continue while observing comments.

So far we are not doing anything with strings between single quotes, just acknowledging them. Garbage characters are collected silently and lex calls an error function which then notifies the user.

2 Syntax Error Reporting Strategies

Syntactic errors were detected by placing the 'error' token in reasonable positions. The yyerror function then uses the information we store holding the line and column position to provide a reasonable message to the user.

Since line endings have little meaning in the PAL language, we don't use the yerrok macro, as there is no way to match an error to a whole line (to do so may mean an error matches the whole program). As such, if we encounter an error, we don't want to output a bunch of other errors that are immediately caused by the first.

3 Resolving Conflicts in the Bison Grammar

The conflicts in the bison grammar were mainly the result of ambiguities created by multiple similar paths to the grammar's atoms. The most serious example of this was how both "type" and "expr", two similar rules, reduced along separate paths to simple_type. We were able to remove this ambiguity from the grammar by removing the rule for reducing to simple_type from those available to expr, and instead replacing it with the applicable atoms.

Another issue with the grammar was the possibility of declaring an array with type anonymous enum. We replaced this rule with two new rules, one of which allows for instantiation of strings and one which allows only previously declared variables as a type.

4 Symbol Table

The symbol table entries follow the course notes in their design. The symbol table itself is an array of linked lists of symbol table entries. The array is size 17 to correspond to the display register values in ASC and allows for a base level of predefined values which we have hardcoded into the compiler.

The symbol table includes a few basic functions as an interface to it. PushLevel and PopLevel are included to change the scope of the program - these are called in Bison when we begin to parse inside a new procedure or function. Various functions have been created to add items to the symbol table - one for each of the main types (var, const, type, func, proc). We have included a symbol table printing function that is current a work in progress.

The symbol table consists of sym_rec structures, which are at the main level of the symbol table. These structures have a class number (eg. OC_TYPE) which defines a field in a union containing more class specific information. The most interesting of these is the struct type_desc including the types of allowed types. This field is important to comparision later on, as we are able to pull out of it the address of a pointer that was created when a type as declared to compare it with other types. Every variable and constant is linked to a type, so through many levels of redirection we are able to get all of the information we require about an items in the symbol table.

5 Semantic Error Checking and Symbol Table Usage

All semantic error checking is handled inside of the Bison grammar - it has helper functions, but the Flex code simply returns tokens. Because of this, we were able to remove INT, REAL, BOOL, and CHAR from the grammar file, allowing Flex to pass Bison just the ID token with the identifier name that it parsed. All integer constants are handled with INT_CONST and real constants with REAL. In the grammar rules for declaration, most variables are handled by recieving an ID token and doing a local lookup with the string that was passed with it.

Populating the symbol table is straight forward for most classes. During function or procedure parsing the expected parameters are added to the next level of the symbol table as well as copied over to a linked list of expected parameters attached to the entry for the function or procedure at the current level. Enumerated types also have a different handling. As the grammar parses them they are collected in a linked list and sent back through the grammar using $$ until a type needs to be defined or a variable needs

type checking null passing

6 References Cited

Only the class notes, class discussions, Bison/Flex manuals, and a few helpful O'Reilly guides were used in completing this checkpoint.