CMPUT 415 – Team YACC – Compiler Documentation

Sarah Anderson
Marko Babic
Guanqui Huang
Aaron Krebs
Shihao Xu

Error and ErrorLL Modules

The Error module is used to model an error object within the code – that is, an Error struct carries with it all information associated with a particular error found during compilation (or at least that information which is relevant during syntax analysis). The Error module also implements the functionality required by other sources which need to record and retrieves the errors (e.g., creating from its members a string for printing in the program listing, printing itself to stderr, etc.).

The ErrorLL module is a simple linked list implementation such that every node is an Error struct. This list is used to keep track of the errors which occur in the program for their later printing into the program listing.

Note that the Error.h file defines the MACRO *MAXERRORS* – this is used to set a limit on the number of errors we plan to report for a given compilation. This maximum is set in place to insure that in the unlikely case that the compiler is given a particular input sequence such that it exposes a loop in the grammar and parsing continues indefinitely we do not continue to consume memory which will never be put to use. Moreover, no human user is going to usefully parse hundreds of errors at once and it is likely the case that if we are reporting a large number of errors while parsing a particular source file then it is far more likely that our compiler is failing in its analysis than it is the case that all these errors actually have occurred and all need to be reported.

ProgList.c and the Program Listing

The function *printProgramListing()* in ProgList.c is run upon completion of the parsing of the input *.pal* file (whether it was successful or not). Using the linked list of Error structs generated during parsing, *printProgramListing()* prints the program listing inlining the errors as appropriate.

To print the errors, we have chosen to seek through the input file from the beginning and print errors at the appropriate lines. This requires a second reading of the input file, it eliminates having to store the entire input file in memory. Not having to store the file in memory allows us to avoid issues with variable (and very long) line lengths. Further, it avoids using the memory to store the file if the -n argument is given.

Unit Testing of C Modules

The correct functioning of every module was assured by creating unit tests for each function in the module. Any time a source file was changed, the unit tests were updated and the test suite run again to affirm their continued correctness. Unit tests are run using the MinUnit framework (see http://www.jera.com/techinfo/jtns/jtn002.html), with some alterations.

Lexical Rules and Tokens

When the parser has not reached the end of file, it tries to match the string that it just parsed from input file with each lexical rule in top-down order,it will return the token to the grammar if the string matches the pattern of any tokens.

The list of token including types names, operators and literals are pre-defined in the definition.tokens file, which is placed into parser.y and testTokenparser.y (the parser and the test parser for lexer testing, respectively) at build time (to avoid duplicate code).

The lexical rules also use Flex states (INITIAL, INSIDE_BLOCK_COMMENT,

INSIDE_STRING_WITH_NEWLINE) to allow finding comment starts and ends more easily.

Grammar Construction and Correctness

The grammar was taken from the given Pal language specification. From there, we removed shift/reduce and reduce/reduce errors primarily by removing redundancies in the productions. Once the obvious redundancies were eliminated we refactored the productions by looking through Bison's output file, and determining which states were causing conflicts and which action should be taken at each state. The biggest problem was to determine how to prevent anonymous scalar types from being declared in array type declarations. This was solved by refactoring scalar_type into a different non-terminal, and moving the other scalars (that were below scalar_type) into simple_type. These will all be removed and replaced with an ID terminal later, so their type does not matter for syntax analysis.

The correctness of the grammar is ensured by extensive integration testing. Out test suite has both correct pal and incorrect pal, and after each grammar change we made sure there were no regressions. This was particularly important when we introduced error tokens into the grammar for error recovery. When we had doubts about the correct behaviour of the grammar, we referred to the Pascal ISO specification.

Error Reporting

Lexical and syntactical errors are reported immediately to the user as they are found by the printing of a message to *stdout* through the use of the standard YACC/Bison default error reporting function *yyerror*(). We chose to print to *stdout* rather than *stderr* since we feel *stderr* should be reserved for errors in the executable, which errors in the input file are the desired regular output for the program. This output can be disabled with the *-q* argument, which is documented in the man page.

For syntax errors caught by Bison during parsing, this message consists of the line and column number on/at which the error occurred (the line and column number are tracked and updated during the parse

by actions in the *lex* file *tokens*), the “verbose” message generated by Bison detailing the cause of the error and an appended token which indicates the token Bison was examining and trying to match to the grammar when the error occurred. All errors which Bison detects are reported provided we have seen less than the maximum number of errors defined in *Error.h*. This strategy was chosen for its simplicity and its surprising effectiveness at delivering a useful error message to the user. Although the reported message and tokens may not always indicate the true, exact reason for a syntax error, it often provides enough information such that in conjunction with the reported line number and column number it is relatively easy to deduce the piece of code which is causing the error. At any rate, these error messages are much more useful than just “Syntax error line n”.

Certain lexical errors are also reported with explicit calls to *yyerror*() in *tokens*.*l*. Namely, if an ending block comment delimiter '*}*' is found without a matching '*{*' delimter, or if a string literal spans multiple lines, or if a token not matchable to any rule in the lexical analyzer appears in the input, then an error is reported. These errors are reported from the lexical analyzer as return specific tokens for multiline strings, or comment delimiters seemed unnecessary. We do, however, return a token for unrecognized tokens as they have use in the grammar (e.g., if the user input a variable named *my_var*, then the underscore is reported to be an illegal character but we return an identifier with value == *myvar* – this was done as the assumption that *my_var* was meant to be a single identifier is a safe one and allows the parser to continue as if this is what the user intended).

Error Recovery

Our approach to error recovery was very much one of trial and error. Namely, we first established a suite of test cases of valid input which we knew our grammar to correctly parse. Then, different pieces of incorrect input were tried until one was found to significantly deteriorate the quality of the compiler's error reporting ability (e.g., if a particular incorrect input caused the compiler to reject tokens until it had reach ended of file it was considered to be an unacceptable response). Once such an

input had been discovered, we made changes to the grammar one rule at a time using tools such as the output generated by Bison when having set the YYDEBUG flag and Bison's ability to generate a human-readable version of its state-space and logic. After each change we re-ran our suite of tests to insure that all previously correctly parsed input remained so and that no shift/reduce or reduce/reduce errors had been introduced into the grammar.

The strategy to improve the compiler's performance against a particular input was very myopic: find a “bad” input, trace back through YYDEBUG output to determine which tokens were being matched at which rules and why the parser was behaving poorly on the particular input, add new rules and error productions to the grammar based on the prior analysis such that the new rules produced no ambiguities in the grammar (i.e., shift/reduce, reduce/reduce errors) and did not results in previously correctly parsable inputs to report errors until the parser was able to recover the incorrect input in an “acceptable” number of tokens (e.g., at the next semi-colon or keyword).

This approach results in somewhat strange, specialized rules appearing the grammar which serve no purpose other than to help the parser recover from a particular class of errors.

Take for example the production '*plist_finvok error R_paren'* in the *func_invok* rule in the grammar. This rule exists entirely to help the parser resynchronize at the closing parenthesis if it enters the error state inside the parenteheses of a function invocation (e.g., the user attempts to use to the keyword var followed by an id, a colon, and a type as an argument to a function at invocation)

Attempting to rework the grammar in order to allow for better error recovery was far and away the most time consuming endeavour thusfar in the project. Difficulties were encountered in attempting to retrace the sequence of shifts and tokens which results in the parser entering the error state and finding out when and why it finally recovered was a painfully meticulous process. Further to that, once it was understood how the parser was entering the error state and consuming tokens it was not always obvious where to introduce error productions and additional rules such that the parser would resynchronize at what we knew/thought to be the most convenient and/or logical place.