Checkpoint 1 Documentation

Damon Barrett - 0834104

March 6, 2017

1 Design and Implementation

The design of the $C-$ compiler was carried out step-wise. The scanner was completed and tested first, before the same was done for the parser. Error recovery was added last, and does not cover all cases. The compiler is implemented in pure C.

1.1 Scanner

The scanner uses a record type to send tokens pulled from the input stream to the parser. The record type was chosen for a few reasons. First, the record type allows easy packaging of multiple data points. The record type also allows for easy extensibility if we ever wish to return more information to the parser.

Currently, the record type holds three tokens of information, the enumerated token type, the token value, and the line on which the token was found. The token type is obviously used in the grammar rules of the parse, and the token value and line number are used in providing rich error messages to the user.

The scanner is implemented using flex.

1.2 Parser

The parser for the $C-$ compiler is implemented using Yacc. The parser is responsible for translating the token stream from the scanner into the abstract syntax tree. The parser generates the abstract syntax tree from the bottom-up.

The generated abstract syntax tree is composed of ASTNode type objects. This record type holds a type (IfStatement, Expression, etc...), a value, and up

to 15 children. Not every node uses all parts of the ASTNode record. When the abstract syntax tree is printed, the node type is used to determine the print formatting of the node contents.

1.2.1 Error Recovery

The error management for the parser is managed by an ErrorList internal to the Yacc file. The ErrorList contains a list of errors generated by the parser stemming from the current file. The ErrorList has a new Error added to it each time an error is encountered. Errors can be added by the yyerror() function, or by various error recovery actions embedded in the actions of the parser.

Upon completion or termination of the parse, the complete error list is printed. The error list is always in sorted order by line number, so printing the errors in chronological order is as simple as printing the list.

1.3 Testing

Testing took place after the creation of the scanner to ensure that the proper tokens were being returned. The next major round of testing occurred after the implementation of the parser. Correct *.cm* source files were fed to the parser to ensure the syntax tree being produced was correct. The last testing was done alongside implementations of specific error recovery mechanisms. A mechanism would be implemented, then subjected to a variety of conditions to judge is capability.

All parser and scanner tests were carried out using a test set of five files

containing all lexical constructs, and tests for error recovery mechanisms were created as needed.

2 Reflection

2.1 Scanner

The scanner for this assignment was trivial to implement after completing the warm-up assignment. The scanner portion of the assignment did not give me any problems, and gave a solid base from which to start.

2.2 Parser

This was my first time using a parser tool in such a thorough manner, and overall it was great. Using a parser tool is definitely much easier than parsing by hand, even when considering the slight learning curve required for the tools syntax. The ability to add actions, and reference specific components of rules made parser development relatively straight forward.

2.3 Error Recovery

Error recovery was implemented once the parser was stable. This portion of the assignment proved the biggest challenge for me. Writing rules to try and recover from seemingly common errors often fixed the specific error I was working on, but caused a conflict on some other file that caused the error messages printed to be horrendously inaccurate. Eventually, some progress was made on some

cases, however the error recovery implementation is not as complete as I hoped.

3 Future Improvements

Overall, I feel that the scanner portion of my compiler needs no improvement The parsing component of the parser is also at the same stage of viability, and seems to do an excellent job translating code to abstract syntax trees.

The major improvements in the compiler are required in the error recovery portion. The error recovery sections need to be expanded to cover more cases, in order to provide the most detailed feedback possible to the programmer. Recognizing more common error scenarios is possibly the best way to go about this.