Crafting a Compiler

Nevin Leh
Aaron Newhall
Tim Weghen

April 3, 2016

1 Introduction

The purpose of this project is to create a compiler. This is essentially a translation of a human readable language into a computer readable language. This is being done to explore the realm of computing at the most basic level and to illustrate the process of creating a compiler from start to finish.

2 Background

3 Methods and Discussion

3.1 Scanner

Creating a scanner is the first step of creating a compiler. The scanner can be compared to the dictionary in ordinary language. It simple finds all the legal words and symbols(called tokens) that make up a program and rejects a program that contains invalid tokens.

For this compiler the lex program from the PLY package provide by python was used. This tool was chosen for it's simplicity and ease of use in creating a scanner. It is very similar to the lex program provided with most unix systems and has a relatively easy setup compared to other scanner generators such as ANTLR. Initially ANTLR was the chosen scanner for this project but the very specific dependencies proved to be too complicated and a switch to PLY was decided.

Implementing the scanner was pretty simple. The provided guide was used heavily to get used to using PLY and to visualize what was needed to use PLY Every token was defined such as keyword or operator and a regular expression was used to define each one of these rules. Most of the regular expressions were pretty simple with a couple exceptions. One has to be careful that a token is not matched by a different token. For example, "ENDWHILE" will be matched by the rule that defines "END". To get around this a lookahead was used

so "END" would only match if it wasn't followed by "WHILE". Another tricky part was finding a way of excluding the comments from the output file. It was finally determined that a prefix of t_error was needed for the comment regular expression so it would be ignored.

3.2 Parser

The next step to creating a compiler is creating a parser. It was decide that using the yacc tool within PLY would be the best parser generator to use because Lex was used to generate a scanner. These two are made to be used together and the PLY yacc can directly use the output from PLY lex. The guide at http://www.dabeaz.com/ply/ply.html was heavily used to create the bare bones of the code. From there it was essentially copy and pasting the grammar rules defined for the LITTLE programming language into functions.

One of the main difficulties that was encountered was the fact that the initial scanner that was made did not match the parser correctly. Initially, keywords or operators were matched as a KEYWORD or an OPERATOR. For the parser more specific token recognition was needed such as differentiating a comma from a semi colon. To fix this new rules were added for the operators and keywords. Another difficulty was the fact that the parser would assume a keyword was an identifier since the identifier rule in the scanner had higher precedence. To fix this a rule was added to the identifier rule that would only match an identifier if it wasn't in the keyword list.

In addition there was a problem with mixed syntax from several examples which through some errors that were hard to find. Finally it was found that some rules in the scanner started with a capital T instead of a lower case t. Changing this somehow fixed it.

3.3 Symbol Table

One thing that needs to be done while creating a compiler is making a symbol table. You have to do this manually

Problems: Didn't know how to perilate information up through

Had to add some rules to find start and end of blocks and functions

3.4 Semantic Routines

3.5 Full-fledged Compiler

4 Conclusion and Future Work