Compilers

Group 12
Stuart Dilts
Joe Whitney

February 24, 2017

1 Introduction

When writing a computer program, many developers and scientists want to think about what the program is going accomplish, and focus on the algorithms that are deployed, not how the computer's state machine is actually performing each and every action. In order to achieve this abstraction, a compiler is needed to translate a programmer's ideas into the low level assembly code of the system. Compilers have other benefits as well, including simple code optimization and code portability. For our senior capstone project, we are implementing a compiler for the Tiny Language. To write a compiler, one needs to be able to apply various aspects of computer science including language parsing, discrete math, and finite automata, all while following good software engineering principles.

2 Background

Java and ANTLR are being used to build the compiler. ANTLR is generating both the scanner and the parser modules of the compiler. Both of these were chosen to speed up development time, as we are very familiar with Java, and ANTLR comes strongly recommended.

Before creating a compiler, the programing language's word list and grammar need to be defined. The Tiny Language is a subset of the

3 Methods and Discussion

3.1 Scanner

The purpose of a scanner is to take raw input (usually in the form of code) and *tokenize* it. Tokenization is the process of converting an input stream into a series of tokens, i.e. literals, keywords, identifiers, etc. Regular expressions are used in defining tokens, and subsequently they are central to the tokenization process. These tokens are equivalent to the formal grammar's word list.

As mentioned above, we used ANTLR in Java. We chose to use a scanner generator as it was recommended, and made the overall process of crating a scanner much simpler. The only real difficulties we faced were coming up with correct regular expressions for recognizing tokens; however, had we coded our own scanner we surely would have faced many more difficulties.

3.2 Parser

Parsing the stream of tokens is the first step in extracting meaning from the source code. On a high level, the productions defined in the grammar are used to organize the tokens into a hierarchy that allows the compiler to know how the code is organized. This hierarchy is placed into an abstract syntax tree (AST), where it can then be translated into another intermediate format.

For example, consider the following LITTLE code snippet:

```
WHILE (i < 10)
     WRITE (i * 2 + 1);
     i := i + 1;
ENDWHILE
```

The AST informs the compiler that `i < 10` needs to be evaluated before `WHILE` can decide if the program needs to continue looping. Likewise, it shows that `i * 2 + 1` is the argument for to `WRITE`, and knows the order of operations needed to compute `i * 2 + 1` correctly.

3.2.1 Building the AST

There are two main ways to parse a grammar: productions can either be derived starting from the smallest piece of the grammar, or from the largest. These two approaches are known as LR and LL parsing, respectively. LR parsers first discover the lowest leaf nodes on the above tree, then work up until the entire program is parsed. LL parsers start from the top of the tree and move downwards. In both cases, some parsers need to look beyond the next token to determine how to proceed. This is defined as the *look ahead*, and is denoted in parenthesis: an LL(1) parser will look ahead one token to determine which production should be used. In general, the more complicated the grammar, the more look-ahead is needed to correctly build the AST. ANTLR uses a variant of LL called LL(*), which

3.3 Semantic routines

3.4 Full-fledged compiler

4 Conclusion and Future Work