PyTuring Compiler Overview

Max Miller
University of New Hampshire

May 16, 2018

Why Compile?

The main purpose of PyTuring is to be an educational tool for learning about computability and Turing machines. As a part of this, I see it beneficial to create a compiler with verbose output to show how the instructions in PyTuring are compiled into machine code (x86 Assembly). As a resource, I will be using the "Dragon Book", or rather Compilers: Principles, Techniques, and Tools by Aho, Lam, Sethi, and Ullman. This book will both act as a sort of "instruction manual" as well as a general reference book. I encourage anyone interested to start reading it and join in on the fun.

Overview

If you are interested in seeing the PyTuring documentation, head over to `https://github.com/mam1101/pyturing/blob/master/PyTuring%C2%A0Documentation.pdf`, where the complete documentation for the general Unicode interpreter is hosted.

We can break down the compiler into multiple parts, and address each one as we come to them. The parts are as follows (Aho & Ullman, 1986):

- Lexical Analyzer
- Syntax Analyzer
- Intermediate Code Generator (optional)
- Code Optimizer (optional, but cool)
- Code Generator (absolutely required)

Each of these can be created in parts. This paper will serve as more of an analysis of the theory behind the compiler than actually making the compiler, though some code may be present.

Lexical Analyzer

The purpose of the lexical analyzer is to convert the various symbols and whitespace used within a PyTuring program and turn into compiler readable tokens.

This first requires us to create a directory of the allowed symbols and their meanings, as well as a map of these to their corresponding tokens. Luckily, PyTuring only has a few special-case lines, and the rest are broken down into the command structure. Command structure allows for a very simple token delineation and mapping, in the form of:

(CMD, $state$, $read$, $action$, $goto$)

This allows for an **id** token state of CMD, with a vector for storing each part of the command structure. We can put this in a hashtable when this gets translated into code. When parsing, a look ahead will be necissary to ensure the different aspects of the vector are properly deliniated. In this case, we just need to account for a space character, with special cases of the <;> to end a line and the >> and << as actions on the machine head rather than a placement, though this can be held through the process and determined at the end.

We can check for keywords as we move through the program as well, seeing as there are only a few of them. SUB, and ENDSUB can be broken down into their own tokens, with SUB having the form:

(SUB, $name$, $startState$, $tokenList$)

The token list contains a list of tokens in the command token structure. In code, we can just use a hashtable array for this.

The other keywords can be put into simple 1x2 vectors. Each of the top three commands involves simply storing what type of token they are, along with the value. These do not need to hold anymore information, and cannot by the language's syntax checker.

Syntax Analyzer

References

Aho, S., Lam, & Ullman. (1986). *Compilers: Principles, Techniques, and Tools.* Pearson.