C Processor

@author Thomas Loh

**Abstract**

The purpose of this project is to build a language processor. Specifically, this language processor takes a high-level language, in this case a C-like language, and maps it to a low-level language/machine-level language. In the end, the compiled / processed / interpreted language will be executed by an interpreter. This language processor and the interpreter will be implemented in Ruby.

The functionalities of the C processor are these:

* parser
* code generator

And the sole function of the interpreter is to simulate a virtual machine and execute the compiled C-like language in Ruby’s context.

This language is basically a subset of the C language with some minor tweaks. It supports various data types like integer, character, string, floats and Boolean. Conditionals like if else clause are implemented, as well as while loops. Also, global variables can be declared before the main program along with subroutines. But these are optional. Aside from that, this language supports arithmetic operations such as addition, subtraction, division and multiplication. Along with that, comparative operations such as equal, not equal, greater than, less than etc., are also implemented. This language also supports I/O such as getting input from user and printing output to screen.

**Project Goals**

* To understand the underlying principle of how a programming language is created and processed.
* To be able to use context-free grammars / BNF / PEG to describe the structure of a language.
* To understand the pros and cons of using a compiler or interpreter for a language.
* To master recursive descent approach to parsing
* To master delegation pattern approach in building a compiler.

**Approach**

The main structure of this project is compilation-based. The main approach is to divide the project into many smaller subtasks, where each task depends on the product of the last task. By the end of the project, I should have these programs:

* A **parser** thattakes a file as an input and converts it to string, then parses it to produce an abstract syntax tree.
* A **compiler** that takes an abstract syntax tree in hash form and compiles it to produce bytecode.
* An **interpreter** that acts as a virtual machine (hypothetical processor) for executing the C-like language. It takes a bytecode(in hash form) as an input.

Here’s a chronological view of how this language will be processed and executed:

Source code

[Parser]

[Compiler]

[Interpreter]

Specifically, here’s a big picture of how source code went through each phases.

**In Parser**

1. Source code is being fed into the parser.
2. The parser builds an abstract syntax tree with syntax nodes represented by each important part of the code such as assignments, if else clause, while loops, output etc. The abstract syntax tree consists a gigantic tree with many nodes stored within nodes. Each syntax node has four attributes: offset, name, text\_value and element.
3. The abstract syntax tree is then converted into a hash with hashes stored within hashes.

**In Compiler**

1. The hash tree produced by parser is fed into the compiler.
2. Each syntax node (hash) in the tree is computed by corresponding helper function.
3. Each helper function returns the appropriate bytecode (array) and in the end all bytecodes are concatenated.

**In VM**

1. The bytecode produced by compiler is fed into the virtual machine / interpreter.
2. The interpreter interprets the bytecode sequentially, namely, from left to right and executes each instruction appropriately.

**Tools**

For the parser, I will be using a Ruby gem called Treetop that takes a grammar file and produces a parser class that does parsing based on the rules defined in the grammar file. The grammar file is based on PEG (Parsing Expression Grammar) and consists of many rules of the language.

The output of the parser is an AST (Abstract Syntax Tree), which is then converted into a hash.

The AST hash is fed into the compiler as input. Based on the AST, the compiler would compile the program using a delegation approach. For each syntax node in the tree, the compiler would call corresponding class to handle that node and return the opcode. When all the syntax nodes have been compiled, an array of opcode (bytecode) will be fed into the virtual machine.

The virtual machine implemented is based on an open-source virtual machine. It has stack data structure and is very similar to the PL/0 interpreter.

**Language Syntax and Semantics (BNF)**

Syntax Rules

<block> ::= global\* body

<global> ::= <assignment> | <subroutine>

<body> ::= <begin\_program> <statement>+ <end\_program>

<statement> ::= <assignment> | <conditional> | <loop> | <input> | <output> | <procedureCall>

<assignment> ::= [ <type> ] <identifier> “=” <expression> “;”

<conditional> ::= “if” “(“ <expression> “)” { <statement>+ } [ “else” { <statement>+ } ]

<loop> ::= “while” “(“ <expression> “)” { <statement>+ }

<subroutine> ::= <begin\_subroutine> <statement>+ <end\_subroutine>

<subroutine\_call> ::= <identifier> "(" <identifer> | <number> ")" ";"

<output> ::= “puts” “(“ <begin\_quote> <string> <end\_quote> | <identifier> "+" “)” “;”

<input> ::= "gets" "()" ";"

<expression> ::= <comparative> | <additive>

<comparative> ::= <additive> <comparative\_operator> <additive>

<additive> ::= <multitve> <additive\_operator> <additive>

<multitive> ::= <term> <multitive\_operator> <multitive>

<term> ::= <identifier> | <number> | ( <expression> )

Lexical Rules

<comparative\_operator> ::= == | != | < | > | <= | >=

<additive\_operator> ::= + | -

<multitive\_operator> ::= \* | /

<identifier> ::= <letter> <ld>\*

<ld> ::= <letter> | <digit>

<number> ::= <digit>+

<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<type> ::= “bool” | “char” | “int” | "string" | "float"

<begin\_quote> ::= """

<end\_quote> ::= """

<begin\_program> ::= "main(){"

<end\_program> ::= "}"

<begin\_subroutine> ::= "void(" (<type> <identifier)? "){"

<end\_subroutine> ::= "}"

NOTE:

* Global variables are declared with a ‘$’ in front.
* Functions only take one parameter.
* Declarations of global variables or functions are optional.
* Variables must be defined when declared.

**Processing Support**

To support the processing/simulating of my language, I’ve decided to modify an open source virtual machine for this project. This modified virtual machine is heavily influenced by an open source virtual machine. It is modified accordingly to support all the features of my C-like language. Here are some of the features that I’ve utilized for my C-processor.

Overview

Stack

Support for various data types

Support for input/output

Support for arithmetic and comparative operations

Support for subroutine definition and calling

Arrays for storing local/global variables

Register for instruction pointer

Register for current instruction register

Supports different levels of scope

Support Jumps to simulate conditionals and loops

Data Types

Integer

Float

String

Character

Arithmetic Operations

Add

Subtract

Multiply

Divide

Comparative Operations

Equal

Not Equal

Greater Than

Less Than

Greater Than Or Equal

Less Than Or Equal

**How To Use**

This language shares many similarities with the C language. In fact, it is a subset of the C language with some minor tweaks. To write a program in this language, these rules should be referenced.

1. The beginning of the program can have declarations of global variables or functions. Global variables start with a “$” sign. Functions start with the keyword “void” and can have at most one parameter.

i.e. Global variable

int $c = 0;

i.e. Functions

void Square(int i) {}

1. The body of the program starts off with the keyword “main()” followed by open curly bracket and ended by a close curly bracket.

i.e. Program body

main(){}

1. The body of the program consists of different kinds of statements like assignment, if else conditionals, while loop, output, input and subroutine call.
2. A function cannot be assigned to any variable and can be called only on its own with parameter. Upon function return, the result is stored in the passed variable in the parameter

i.e. Function call

Square(x)

1. All declared variables must be assigned a value during declaration. If not, it will throw an error.
2. All variables’ original value are modified and lost once they are passed into a parameter of a function as the result is stored in that parameter.
3. Here’s a sample program that works.

void SayHelloNTimes(int n)

{

while(n > 0)

{

puts("Hello World: " + n);

n = n - 1;

}

}

int $d = 0;

void Sqr(int s)

{

$d = s \* s;

}

void Fibonacci(int r)

{

int a = 0;

int b = 1;

int i = 0;

int t = 0;

while(i < r)

{

puts("a: " + a);

t = b;

b = a + b;

a = t;

i = i + 1;

}

}

main(){

int d = 0;

string s = "hello";

float f = 0.0;

char c = 's';

d = gets();

if(d >= 10)

{

if(d > 10)

{

puts ("Value of d (" + d + ") is greater than or equal 10");

}

else

{

puts ("Value of d (" + d + ") is equal 10");

}

}

else

{

puts ("Value of d: (" + d + ") is less than 10");

}

puts("");

puts("Demo of subroutine 'SayHelloNTimes' ");

puts("------------------------------------");

puts("");

puts("Say hello how many times?");

d = gets();

SayHelloNTimes(d);

puts("");

puts("Demo of subroutine 'Sqr' ");

puts("------------------------------------");

puts("");

puts ("Find square of this number:");

d = gets();

Sqr(d);

puts ("Answer: " + d);

}

\* This demo is called sample.c and is provided in the project.

**Instructions for use on your system**

**For Unix machine:**

\* Make sure you have Ruby installed on your machine (stdsun doesn’t have rubygems installed, so please use your own machine).

\* To install rubygems, type these commands(might need to sudo)

**gem install rubygems-update**

and then

**update rubygems**

Now you should have rubygems installed.

Install these gems as well:

**sudo gem install treetop**

**sudo gem install polyglot**

**sudo gem install facets**

To write a program using this language, follow these instructions:

1. Open up Terminal.
2. Navigate to project655/processor/ using the terminal or Finder.
3. Create a [name].c file and write your source code in it. Save it when you’re done.
4. Type this command on the command line:

**ruby main.rb**

1. There should be a prompt for a file name. Please provide one. (type “sample.c” for a demo)
2. The program should run unless there’s a syntax error in your program.

**Project Design & Planning & Execution**

**How long did your project take you?**

This project took me approximately 6 weeks to design, develop and test. For the first 2 weeks, most of the work is focused on the project proposal. From 3rd to 5th week, most work is on coding and implementing the parser, compiler & virtual machine, chronologically. Testing and documentation are done on the 6th week.

**Where did you deviate from your project proposal, why?**

I deviated from my initial proposal in several ways. First, I’ve decided to use ruby gem called Treetop to build the parser instead of lex and yacc as I realized that using Treetop is a much quicker yet efficient way to build the parser. Another deviation is that I’ve decided to use PEG (Parsing Expression Grammar) instead of BNF to build the parser class for this language. After exploring Ruby quite some time, I realized there’s a ruby library (gem) called Treetop that takes a grammar file (PEG) as input and build a parser class based on that grammar file. It is a much elegant and cleaner solution as it recursive descent parsing. It is better than Context Free Grammar as it’s able to detect input like palindrome.

Also, instead of reinventing the wheel, I’ve decided to use an open-source virtual machine and modify it to process my language. I figured it would take too much time to build another virtual machine. This virtual machine behaves much like PL/0. Its features are stated above in page – Processing Support.

Some other changes are like: adding input / output functionalities, adding global variables and adding subroutine definition (with one parameter).

**What do you think was the strongest part of your design?**

I think the strongest part of my design is using Parsing Expression Grammar to build the parser that tokenizes and parses the source code. Using the Treetop library, the parser is built based on the PEG. One notable feature of PEG is that it has no lexical analyzer as it guarantees the same time complexity as other lexical analyzers. Therefore, everything is compiled nicely into one parser that scans, tokenizes and builds the abstract syntax tree. The abstract syntax tree is very useful during development as it allows us to see the “graphical” layout of the source code using syntax nodes. It’s very useful for debugging purposes.

**What do you think was the weakest part of your design?**

I think the weakest part of my design is the integration of the virtual machine with the parser and compiler. The abstract syntax tree produced by the parser is fed into the compiler, which then produces the bytecode. The bytecode is an array of numbers that only the virtual machine understands. I think this is the weakest part of my design. As it is a series of numbers, it’s very hard to understand and difficult to debug. I can say I’ve spent the most time in debugging the compiler to make sure it produces to correct bytecode.

**What were the easiest and hardest parts to implement?**

The easiest part of the project to implement is the parser. As aforementioned, it uses PEG to describe the grammar of the language. It’s easiest in the sense that it alone can be used to build the parser using the Treetop library. Although it’s the easiest, doesn’t mean it’s easy to create an elegant yet understandable grammar for the user. The tradeoff is that I have to handle spaces in the code as PEG takes everything into account. It actually took me more than 72 hours to come up with a good grammar file.

The hardest part of the project is undoubtedly on implementing the compiler and making sure the virtual machine integrates seamlessly with it. Debugging the compiler took most of the time, as aforementioned, and after producing the correct bytecode, the hardest part is to make sure the virtual machine handles and executes the bytecode properly. This part took me more than 100 hours to implement.

**How well did you implement your design?**

I believe I’ve implemented my design extremely well according to my initial design plan. Although there were challenges/obstacles along the way and some things have to be changed, the overall design didn’t change. The change I made is that subroutine doesn’t preserve the value of the passed parameter. Initially, the design is to let the subroutine preserve it. But due to time constraints, I was unable to debug it as there’re many other bugs waiting to be fixed. Another change is that variable must be declared and assigned a value. The initial design plan is that variable can be declared but not defined. This however leads to complications in the grammar file and I have to put a restriction on the language on having variable defined when declared. However, letting subroutine alter the parameter and having variable defined when declared doesn’t affect the overall design of the project.

**Review of coding style and internal comments.**

My coding style is based on delegation pattern approach. The idea is to split the main function into many smaller sub-functions that handle different parts of the source code, in this case, syntax node. Instead of having one large block of program that handles everything sequentially, I have a series of helper functions that executes certain parts of the code. In this way, those helper functions can be reused multiple times and the program is much cleaner looking.

Internal comments are inserted appropriately in the program. I believe it’s sufficient to be understood by a first-time user of the code.

**Overall project functionality**

The overall functionality of this project is quite sophisticated. It has most basic features of a regular programming language. These features are demonstrated in the test cases provided. It can be used to write some simple programs, from saying “Hello World!” to printing Fibonacci sequence up to n times. Although there are a lot of spaces to improve, but it has all the features as far as this course is concerned. All in all, the overall project functionality is complete and relevant to this course.

**Testing Strategies**

**What strategies did you use to develop your test cases?**

The main strategy I’ve used to develop my test cases is bottom-up approach. There are 3 phases in processing the language. First it goes through parsing. Second it goes through compilation. Finally it is simulated. The input of every stage depends on the product of the phase before it except for the first phase – parsing. In the first, phase, a file name is provided and that’s all. Hence in this case bottom-up approach is appropriate because I would start testing at the parsing phase. After making sure everything parses correctly, I can guarantee that I have a valid abstract syntax tree for input into the compiler. So then I would move on to the second phase and start testing the compilation process. After thorough testing, I can guarantee that the bytecode produced by the compiler is valid. Then finally I move on to testing in the final phase—simulation. By inputting various bytecode produced by the compiler, I tests if the results produced are the results expected. If it is, then the test case passes, if not, the test case fails.

**What was each test or set of tests meant to expose?**

The set of tests in “project655/processor/parser/Tests” folder is meant to expose / catch any errors found in parsing the source code.

The set of tests in “project655/processor/compiler/Tests” folder is meant to expose / catch any errors found during compilation process of the abstract syntax tree.

The set of tests in “project655/vm/Tests” folder is meant to expose / catch any errors found during simulation of the bytecode.

**Given more time, what would you test further?**

If given more time, I would test for functionality of recursion of the language.

**What errors did you discover with your tests?**

From the tests, I have discovered several errors, but mostly minor errors. Some of the bugs found are such as: expression in if conditions cannot be the word “true” or “false”, there must be at least zero statement in the body and so on…\*Refer to test cases.

**What tests convinced you that project goals had been met?**

The test results from each set of tests in parser, compiler and vm convinced me that the project goals have been met. Through testing of each phase, I understand the underlying principle of how a programming language is created and processed. I’m also able to use BNF / PEG to describe the structure of the language. In testing of the parser, I have mastered the technique of recursive descent approach to parsing. As for testing of the compiler, I have mastered the delegation pattern approach in building the compiler.

**Demos (in class)**

Depth of purpose and functionality

What is the core purpose of your project?

Lessons learned to share with class

Professionalism & Organization of demo

Overall success of project

In-class demo / discussion as a supplement.