Brian Cullinan

Project 10: Final Compiler

12/03/2008

**Objectives:**

Create a code generator in order to complete the compiler. This is the final bridge to cover the gap between the language code and the compiled assembly code that can be run by the interpreter.

**Overview:**

I created a code generator and then a wrapper function for putting it all together. I made lots of changes to my assembler and interpreter. I fixed many little problems with those things. The code generator works pretty well using special functions for inserting code into different positions.

**Approach:**

I first implemented loading and saving of variables and jumping around. Then I implemented the loops. Then I implemented the rest. I started by just looking at the output code and making sure that it would function properly. Then I started running the VM on it and fixing the problems between the two.

Some of the difficulties I had were making the jumps go to the right place. Also for some reason when I just put a new line character between generated commands, it doesn’t parse the line characters properly. I had to add both the carriage return and the new line character in order to get it to read the lines and parse them out.

I also added a few commands for printing out characters and manipulating variables. Development went pretty smoothly, I just kept adding handlers to my generator for each type of node. I had to figure out a way for code to be inserted in a different position, instead of reformatting my entire tree and potentially breaking other aspects, I decided to add functions for inserting code at a later time. Also, generating the code first and then inserting it somewhere else. So my handlers for each node type all have to work together.

There are a few inefficiencies. The generated code isn’t quite as compact as it could be in order to achieve the same functionality, but it still works pretty well, and very fast.

**Conclusion:**

The compiler works well, it does not have the bonus functionality of things like arrays. It is fast because PHP is very fast. I doubt I could rewrite the compiler in my language. It can calculate 1000 Fibonacci numbers in about a second and a half, including sending the output to the browser. My language is not nearly as cool as I hoped it would be.

Generator.php:

<?php

require\_once 'visitor.php';

require\_once 'parser.php';

require\_once 'printer.php';

class Generator extends Visitor

{

var $loop\_counter = 0; // used for assigning a number to each loop

var $code = ''; // code generated by visiting each node

function Generator($tree)

{

$this->tree = $tree;

}

function generate()

{

$this->visit($this->tree);

return $this->code;

}

function visitAlways($tree)

{

if(isset($tree->label))

{

$this->code .= $tree->label . ':' . "\r\n";

}

if(isset($tree->insertCode))

{

$this->code .= $tree->insertCode;

}

if(isset($tree->insertTo) || isset($tree->insertToAfter))

{

// save code length so we can find the changes later

$tree->codeLength = strlen($this->code);

}

}

function leaveAlways($tree)

{

if(isset($tree->insertTo))

{

// take the difference between what it was when the node was visited

// and what it is now and insert that difference in the insertTo spot

if(!isset($tree->insertTo->insertCode)) $tree->insertTo->insertCode = '';

$tree->insertTo->insertCode .= substr($this->code, $tree->codeLength, strlen($this->code) - $tree->codeLength);

$this->code = substr($this->code, 0, $tree->codeLength);

}

if(isset($tree->insertToAfter))

{

if(!isset($tree->insertToAfter->insertCodeAfter)) $tree->insertToAfter->insertCodeAfter = '';

$tree->insertToAfter->insertCodeAfter .= substr($this->code, $tree->codeLength, strlen($this->code) - $tree->codeLength);

$this->code = substr($this->code, 0, $tree->codeLength);

}

if(isset($tree->insertCodeAfter))

{

$this->code .= $tree->insertCodeAfter;

}

}

function visitExpression($tree)

{

if(!isset($tree->tree['Operator']))

{

if(isset($tree->tree['Literal']->tree['BinaryLiteral'])) $this->code .= 'push ' . $tree->tree['Literal']->tree['BinaryLiteral'] . "\r\n";

if(isset($tree->tree['Literal']->tree['IntLiteral'])) $this->code .= 'push ' . $tree->tree['Literal']->tree['IntLiteral'] . "\r\n";

if(isset($tree->tree['Literal']->tree['RealLiteral'])) $this->code .= 'push ' . $tree->tree['Literal']->tree['RealLiteral'] . "\r\n";

if(isset($tree->tree['Literal']->tree['HexLiteral'])) $this->code .= 'push ' . $tree->tree['Literal']->tree['HexLiteral'] . "\r\n";

//if(isset($tree->tree['Literal']->tree['StringLiteral'])) $this->code .= 'push ' . $tree->tree['Literal']->tree['StringLiteral'];

if(isset($tree->tree['VariableName'])) $this->code .= 'gload ' . $tree->tree['VariableName']->info->index . "\r\n";

}

elseif(!isset($tree->tree['VariableName']))

{

if(isset($tree->tree['Literal']->tree['BinaryLiteral'])) $this->code .= 'push ' . $tree->tree['Literal']->tree['BinaryLiteral'] . "\r\n";

if(isset($tree->tree['Literal']->tree['IntLiteral'])) $this->code .= 'push ' . $tree->tree['Literal']->tree['IntLiteral'] . "\r\n";

if(isset($tree->tree['Literal']->tree['RealLiteral'])) $this->code .= 'push ' . $tree->tree['Literal']->tree['RealLiteral'] . "\r\n";

if(isset($tree->tree['Literal']->tree['HexLiteral'])) $this->code .= 'push ' . $tree->tree['Literal']->tree['HexLiteral'] . "\r\n";

}

else

{

$this->code .= 'gload ' . $tree->tree['VariableName']->info->index . "\r\n";

}

}

function leaveExpression($tree)

{

if(isset($tree->tree['Operator']))

{

if($tree->tree['Operator'] == '+') $this->code .= 'add' . "\r\n";

}

}

function leaveAssignment($tree)

{

$this->code .= 'gsave ' . $tree->tree['Assignment']['VariableName']->info->index . "\r\n";

}

function visitForLoop($tree)

{

$tree->tree['Expression']->label = 'loop' . $this->loop\_counter;

$tree->tree['Counter']->insertCode = 'jgt end' . $tree->tree['Expression']->label . "\r\n";

$tree->tree['Counter']->insertToAfter = $tree->tree['CommandList'];

}

function leaveForLoop($tree)

{

$this->code .= 'jump ' . $tree->tree['Expression']->label . "\r\n";

$this->loop\_counter++;

$this->code .= 'end' . $tree->tree['Expression']->label . ':' . "\r\n";

}

function visitParamCall($tree)

{

}

function leaveFunctionCall($tree)

{

$this->code .= 'call ' . $tree->info->name . "\r\n";

}

function visitFunctionDef($tree)

{

$this->code .= 'jump end' . $tree->tree['F-Name']->tree['Identifier'] . "\r\n" . $tree->tree['F-Name']->tree['Identifier'] . ':' . "\r\n";

}

function leaveFunctionDef($tree)

{

$this->code .= 'return' . "\r\n" . 'end' . $tree->tree['F-Name']->tree['Identifier'] . ':' . "\r\n";

}

function visitParamDef($tree)

{

$this->code .= 'pop ' . 0xFFFFFFFF . "\r\n";

foreach($tree->tree['Params'] as $i => $param)

{

$this->code .= 'gsave ' . $param->info->index . "\r\n";

}

$this->code .= 'gload ' . 0xFFFFFFFF . "\r\n";

}

function leaveKeyword($tree)

{

if($tree->tree['Keyword'] == 'p')

{

$this->code .= 'print' . "\r\n";

}

elseif($tree->tree['Keyword'] == 'pc')

{

$this->code .= 'printc' . "\r\n";

}

}

function leaveProgram($tree)

{

$this->code .= 'return' . "\r\n";

}

}

?>

Code.txt:

[

v fib1 = 0 + 1

v fib2 = 1

fo(i = 0, i < i1000, i = 1 + i)

[

fprint fib1

fprint fib2

fib1 = fib1 + fib2

l newfib = fib2 + fib1

fib2 = newfib

]

f fprint(pvar)[

p pvar #prints out 5.08

pc 32

]

Generated Code:

push 0

push 1

add

gsave 0

push 1

gsave 1

push 0

gsave 2

loop0:

gload 2

push 1000

jgt endloop0

gload 0

call fprint

gload 1

call fprint

gload 0

gload 1

add

gsave 0

gload 1

gload 0

add

gsave 3

gload 3

gsave 1

push 1

gload 2

add

gsave 2

jump loop0

endloop0:

jump endfprint

fprint:

pop 4294967295

gsave 4

gload 4294967295

gload 4

print

push 32

printc

return

endfprint:

return

Wrapper.php:

<?php

$file\_name = 'temp\_'; // . md5(time());

require\_once 'parser.php';

require\_once 'visitor.php';

require\_once 'generator.php';

require\_once 'assembler.php';

require\_once 'interpreter.php';

// create a reader object

print "parsing<br />\n";

$parser = new Parser('code.txt');

$tree = $parser->parse();

print "analyzing<br />\n";

$analyzer = new Analyzer($tree);

$tree = $analyzer->analyze();

print "generating<br />\n";

$generator = new Generator($tree);

$code = $generator->generate();

print "saving<br />\n";

$fp = fopen($file\_name, 'w');

fwrite($fp, $code);

fclose($fp);

print "parsing assembly<br />\n";

$scanner = new Assembler($file\_name);

$commands = new VMParser();

$tree = $commands->parseVM($scanner->tokens);

print "assembling<br />\n";

$analyzer = new BinaryOutput($tree);

$output = $analyzer->assemble();

print "executing<br />\n";

$vm = new VM($output, false);

?>