Brian Cullinan

Project 8: Assembler

11/05/08

Objective:

Create and assembler that converts out instruction set into a binary byte-code file.

Overview:

I accomplished this task by creating a parser that reads in my assembly file code, then a Scanner that checks all the commands and the correct number of arguments. It then uses a Visitor to go through the list of commands and create the hex values, which are then converted to binary output.

Approach:

I first copied my Scanner, and made it recognize the simple commands like ‘load’ and ‘save’. All of those are considered keywords. I then had it recognize the arguments that are passed to those commands; they can be either identifiers, or literals. The way I am handling memory access is based upon the type of literal. Hex literals are going to be my global memory registers, which hold the current values of variables for example. Binary literals will handle the stack access. All the other literals are just used as is.

I then decided to use a Visitor in order to produce the output. When a command is visited it inserts the opcode into the output string. When a literal is visited, it inserts a value for the type of literal and its value. When an identifier is visited, it is added to the named address list, and the index of that list is inserted in its place. This allows for all the different named values to have an index that can be referenced at any time. Function definitions can be converted to labels, and then referenced using those indices.

My output is a binary file that includes some information about it, and then all the commands. Here is the print out from vim.

0000000: 534d 564d 0000 0001 0000 0003 0000 0084 SMVM............

0000010: 2b01 0000 0007 5a01 0000 0001 2b01 00bc +.....Z.....+...

0000020: 614e 2b01 00bc 614e 3c01 00bc 614f 2a01 aN+...aN<...aO\*.

0000030: 00bc 6150 5b03 0000 0010 0d0a ..aP[.......

As you can see the first 8 bytes are the file type. The next 8 byes is an integer version number. The next 8 bytes describe how many global variables should be allocated immediately. The next 8 is the total length of the byte-code.

The first byte-code command is 2B which is a load command. The arguments for this command are the integer value, or a reference to an object using a hex value. Since 01 comes right after the opcode, we know that it is a literal integer value, so it can be used by the virtual machine.

Here is some code:

Assembly.txt (this file contains some example generated code):

load 7

push 1

load n

load n

call mult

save i

pop h10

Assembler.php (this class contains all the functionality needed to assemble):

<?php

require\_once 'reader.php';

require\_once 'token.php';

require\_once 'error.php';

require\_once 'visitor.php';

$scanner = new Assembler('assembly.txt');

$commands = new VMParser();

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

//print\_r($tree);

class BinaryOutput extends Visitor

{

var $named = array();

var $output = '';

var $named\_counter = 12345678;

function visitAIdentifier($tree)

{

if(!isset($this->named[$tree->token->content]))

{

$this->named[$tree->token->content] = $this->named\_counter;

$this->named\_counter++;

}

$i = dechex($this->named[$tree->token->content]);

$this->output .= '01' . str\_pad($i, 8, 0, STR\_PAD\_LEFT);

}

function visitAKeyword($tree)

{

switch($tree->token->content)

{

case '32load': $this->output .= '0A'; break;

case '64load': $this->output .= '1A'; break;

case 'save': $this->output .= '2A'; break;

case 'load': $this->output .= '2B'; break;

case 'move': $this->output .= '2C'; break;

case 'jump': $this->output .= '3A'; break;

case 'jumpif': $this->output .= '3B'; break;

case 'call': $this->output .= '3C'; break;

case 'return': $this->output .= '3D'; break;

case 'gload': $this->output .= '4A'; break;

case 'gsave': $this->output .= '4B'; break;

case 'push': $this->output .= '5A'; break;

case 'pop': $this->output .= '5B'; break;

case 'peek': $this->output .= '5C'; break;

case 'add': $this->output .= '7A'; break;

case 'sub': $this->output .= '7B'; break;

case 'xor': $this->output .= '6A'; break;

case 'or': $this->output .= '6B'; break;

case 'and': $this->output .= '6C'; break;

case 'not': $this->output .= '6D'; break;

case 'mul': $this->output .= '7C'; break;

case 'div': $this->output .= '7D'; break;

case 'pow': $this->output .= '7E'; break;

case 'noop': $this->output .= '00'; break;

}

}

function visitALiteral($tree)

{

if(isset($tree->tree['BinaryLiteral'])) $this->output .= '00' . str\_pad(dechex($tree->tree['BinaryLiteral']), 8, 0, STR\_PAD\_LEFT);

if(isset($tree->tree['IntLiteral'])) $this->output .= '01' . str\_pad(dechex($tree->tree['IntLiteral']), 8, 0, STR\_PAD\_LEFT);

if(isset($tree->tree['RealLiteral'])) $this->output .= '02' . str\_pad(dechex($tree->tree['RealLiteral']), 16, 0, STR\_PAD\_LEFT);

if(isset($tree->tree['HexLiteral'])) $this->output .= '03' . str\_pad($tree->tree['HexLiteral'], 8, 0, STR\_PAD\_LEFT);

if(isset($tree->tree['StringLiteral'])) $this->output .= '04' . str\_pad(dechex(strlen($tree->tree['StringLiteral'])), 8, 0, STR\_PAD\_LEFT) . $tree->tree['StringLiteral'];

}

}

// scan the parse tree and generate the binary output

$analyzer = new BinaryOutput($tree);

$analyzer->visit($tree);

header('ContentType: application/binary');

header('Content-Disposition: attachment; filename=' . 'assembly.vm');

//print\_r($analyzer->output);

$analyzer->output = '534d564d' . str\_pad(dechex(1), 8, 0, STR\_PAD\_LEFT) . str\_pad(count($analyzer->named), 8, 0, STR\_PAD\_LEFT) . str\_pad(strlen($analyzer->output), 8, 0, STR\_PAD\_LEFT) . $analyzer->output;

for($i = 0; $i < strlen($analyzer->output) / 2; $i++)

{

print\_r(chr(hexdec(substr($analyzer->output, $i\*2, 2))));

}

//print\_r($analyzer->output);

//print '<pre>';

//print\_r($tree);

class Assembler

{

var $file;

var $tokens = array();

function Assembler($file)

{

$this->file = $file;

$reader = new SourceReader($this->file);

$buffer = '';

$start = 1; # starting column

while(true)

{

$int = $reader->peek(1);

// these are not regular expressions for reading in tokens, they are just used to verify token content

if($int == -1)

{

break;

}

// check if it is a literal

elseif(preg\_match('/^(i[0-9]+|r[0-9]+\.?[0-9]\*|h[ABCDEF0-9]+|b[01]+|[0-9]\*\.?[0-9]\*|\"[^\"]\*\"|\'[^\']\*\')$/', $buffer . chr($int)) != 0)

{

$type = T\_LITERAL;

$buffer .= chr($reader->read());

}

// instruction

elseif(preg\_match('/^(32load|64load|save|load|move|jump|jumpif|call|return|gload|gsave|push|pop|peek|add|sub|xor|or|and|not|mul|div|pow|noop)$/', $buffer . chr($int)) != 0)

{

$type = T\_KEYWORD;

$buffer .= chr($reader->read());

}

// match named stuff

elseif(preg\_match('/^[a-zA-Z][a-zA-Z0-9]\*$/', $buffer . chr($int)) != 0)

{

// always assume it is an identifier until proven otherwise

$type = T\_IDENTIFIER;

$buffer .= chr($reader->read());

}

// check if it is some kind of white space delimiter

elseif(preg\_match('/^( \*|\(|\)|\[|\]|,)$/', $buffer . chr($int)) != 0)

{

$type = T\_WHITESP;

$buffer .= chr($reader->read());

}

elseif(preg\_match('/^(\s\*|\w\*|\r\*|\n\*)$/', $buffer . chr($int)) != 0)

{

$type = T\_EOL;

$buffer .= chr($reader->read());

}

// check for comments

elseif(preg\_match('/^#.\*[\r|\n]?$/', $buffer . chr($int)) != 0)

{

$type = T\_COMMENT;

$buffer .= chr($reader->read());

}

// clear out buffer then set it to new character

else

{

//var\_dump($buffer);

if($buffer != '')

{

if($type != T\_COMMENT && $type != T\_WHITESP)

{

// create token

$this->tokens[] = new Token($this->file, $reader->line, $start, $type, $buffer);

}

unset($type);

$buffer = '';

$start = $reader->col;

}

// must be an invalid character

else

{

// create error

$error = new CompileError($this->file, $reader->line, $reader->col, 'Invalid Character: ' . chr($int));

$error->error\_query();

}

}

}

}

}

class VMParser

{

var $token = array();

var $tree = array();

function accept(&$list)

{

//var\_dump(current($list)->type);

$this->token = current($list);

next($list);

}

function parseVM(&$list)

{

reset($list);

$program = new ACommandList();

$this->tree['VM'] = $program->parseCommandList($list);

return $this;

}

function make\_error(&$list)

{

// create error

$error = new CompileError(current($list)->file, current($list)->line, current($list)->col, 'Invalid Token (' . current($list)->type . '): ' . current($list)->content);

$error->error\_query();

}

}

class AIdentifier extends VMParser

{

function parseIdentifier(&$list)

{

if(current($list)->type == T\_IDENTIFIER)

{

$this->tree['Identifier'] = current($list)->content;

$this->accept($list);

}

else

{

$this->make\_error(&$list);

}

return $this;

}

}

class AExpression extends VMParser

{

function parseExpression(&$list)

{

if(current($list)->type == T\_LITERAL)

{

$literal = new ALiteral();

$this->tree['Value'] = $literal->parseLiteral($list);

}

elseif(current($list)->type == T\_IDENTIFIER)

{

$identifier = new AIdentifier();

$this->tree['Value'] = $identifier->parseIdentifier($list);

}

else

{

$this->make\_error($list);

}

return $this;

}

}

class ALiteral extends VMParser

{

function parseLiteral(&$list)

{

if(current($list)->type == T\_LITERAL)

{

// first check front end for type declaration

switch(current($list)->content[0])

{

case 'i':

$this->parseIntLiteral($list, true);

break;

case 'r':

$this->parseRealLiteral($list, true);

break;

case 'h':

$this->parseHexLiteral($list);

break;

case 'b':

$this->parseBinaryLiteral($list);

break;

default:

if(is\_numeric(current($list)->content) && preg\_match('/^[0-9]\*$/', current($list)->content) != 0)

{

$this->parseIntLiteral($list);

}

elseif(is\_numeric(current($list)->content))

{

$this->parseRealLiteral($list);

}

else

{

$this->parseStringLiteral($list);

}

}

$this->accept($list);

}

else

{

$this->make\_error(&$list);

}

return $this;

}

function parseIntLiteral(&$list, $haspreceder = false)

{

$this->tree['IntLiteral'] = ($haspreceder)?substr(current($list)->content, 1):current($list)->content;

}

function parseRealLiteral(&$list, $haspreceder = false)

{

$this->tree['RealLiteral'] = ($haspreceder)?substr(current($list)->content, 1):current($list)->content;

}

function parseHexLiteral(&$list)

{

$this->tree['HexLiteral'] = substr(current($list)->content, 1);

}

function parseBinaryLiteral(&$list)

{

$this->tree['BinaryLiteral'] = substr(current($list)->content, 1);

}

function parseStringLiteral(&$list)

{

$this->tree['StringLiteral'] = current($list)->content;

}

}

class AKeyword extends VMParser

{

function parseKeyword(&$list)

{

switch(current($list)->content)

{

case '32load':

case 'save':

case 'load':

case 'move':

case 'call':

case 'gload':

case 'gsave':

case 'push':

case 'pop':

case 'peek':

case 'not':

$this->tree['Keyword'] = current($list)->content;

$this->accept($list);

$expression = new AExpression();

$this->tree['Arg2'] = $expression->parseExpression($list);

break;

case '64load':

case 'jump':

case 'jumpif':

case 'add':

case 'sub':

case 'xor':

case 'or':

case 'and':

case 'mul':

case 'div':

case 'pow':

$this->tree['Keyword'] = current($list)->content;

$this->accept($list);

$expression = new AExpression();

$this->tree['Arg1'] = $expression->parseExpression($list);

$expression = new AExpression();

$this->tree['Arg2'] = $expression->parseExpression($list);

break;

case 'return':

case 'noop':

$this->tree['Keyword'] = current($list)->content;

$this->accept($list);

break;

default:

$this->make\_error(&$list);

}

return $this;

}

}

class ACommand extends VMParser

{

function parseCommand(&$list)

{

switch(current($list)->type)

{

case T\_KEYWORD:

$keyword = new AKeyword();

$this->tree['Command'] = $keyword->parseKeyword($list);

if(current($list)->type == T\_EOL) $this->accept($list);

break;

default:

$this->make\_error(&$list);

}

return $this;

}

}

class ACommandList extends VMParser

{

function parseCommandList(&$list)

{

if(current($list)->type == T\_EOL) $this->accept($list);

do

{

// parse a command

$continue = true;

$command = new ACommand();

$this->tree['CommandList'][] = $command->parseCommand($list);

}while($continue && current($list) !== false);

return $this;

}

}

?>