Brian Cullinan

Project 6: Advanced Parser

10/22/08

**Objective:**

Implement an advanced language design parser. Improve the functionality of the simple language parser to handle such things as structures.

**Overview:**

I gave my parser the ability to parse a more advanced language that includes control structures, multiple data types, and conditionals. I accomplished this by modifying my previous parser to recognize more types of tokens.

**Summary:**

I started by modifying my scanner class to handle a few extra types of tokens. I needed square brackets for use with the control structures; I also needed it to recognize Boolean and logical operators. Then I moved on to reorganize my existing code and the types of nodes. With exception to the AST top level node, the code is organized such that smaller expected structures are at the top, and larger structures are at the bottom. For example, a CommandList has a very large structure underneath it that consists of commands and if-statements, whereas a literal has hardly any structure under it so it is at the top. I had a hard time, adding structure functionality at first, but once I accomplished it adding different types of structures was simple. First I implemented a for-loop structure. I accomplished this by adding a handler for the ‘fo’ keyword. I added the functionality to match each of the three pieces of a for-loop definition, and then I added the functionality for recognizing a command-list. Command lists are expressed with the square brackets. As a result I ended up with a nice feature; a command list can be “sectioned off” in any part of the code. In many languages brackets can be placed anywhere without a preceding if or function declaration. This is useful when you want to give you program more block-like appearance, and really accent important code. This is the kind of thing I am going for.

Finally I was able to add multiple structure types, I proceeded to add while loop functionality, ‘if’ and ‘else’ statement functionally, and function declarations. Lastly, I added the extended support for explicitly defining literal types. My parser checks for a specific preceding character to define what type it is, or it will evaluate the numerical value.

**Conclusion:**

I have taken the name and structure changes with operators and loops into consideration. I don’t think it makes sense to proceed with that step before fully considering the analyzer and how I want to handle it. Other than that the project was completed smoothly.

**Index.php:**

<code>

<pre>

<?php

// include some needed classes

require\_once 'parser.php';

require\_once 'printer.php';

// create a reader object

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

$tree = $parser->parse();

print\_r($tree);

$printer = new PrinterVisitor($tree);

$printer->visit($tree);

//$analyzer = new Analyzer($tree);

//$analyzer->analyze();

?>

</pre>

</code>

**Output:**

AST(Program(CommandList(Command(Assignment(Identifier(fib1)Expression(Literal(0))))Command(Keyword(Assignment(Identifier(fib2)Expression(Literal(1)))))Command(Keyword(ForLoop(Assignment(Identifier(i)Expression(Literal(0)))Expression(Identifier(i)Expression(Literal(100)))Assignment(Identifier(i)Expression(Identifier(i)Expression(Literal(1))))CommandList(Command(FunctionCall(ParamCall(Expression(Identifier(fib1)))))Command(FunctionCall(ParamCall(Expression(Identifier(fib2)))))Command(Assignment(Identifier(fib1)Expression(Identifier(fib1)Expression(Identifier(fib2)))))Command(Keyword(Assignment(Identifier(newfib)Expression(Identifier(fib2)Expression(Identifier(fib1))))))Command(Assignment(Identifier(fib2)Expression(Identifier(newfib))))))))Command(Keyword(IfStatement(Expression(Identifier(fib1)Expression(Literal(5)))CommandList(Command(FunctionCall(ParamCall(Expression(Identifier(fib1)))))))))Command(Keyword(ElseStatement(CommandList(Command(Keyword(Identifier(fib2)))))))Command(Keyword(FunctionDef(Identifier(print)ParamDef(Expression(Identifier(pvar)))CommandList(Command(Keyword(Identifier(pvar))))))))))

**Code.txt:**

v fib1 = 0

v fib2 = 1

fo(i = 0, i < i100, i = i + 1)

[

print fib1

print fib2

fib1 = fib1 + fib2

l newfib = fib2 + fib1

fib2 = newfib

]

if(fib1 > 5)[

print fib1

]el[

p fib2

]

f print(pvar)[

p pvar #prints out 5.08

]

**Scanner.php:**

<?php

require\_once 'reader.php';

require\_once 'token.php';

require\_once 'error.php';

class Scanner

{

var $file;

var $tokens = array();

function Scanner($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[ABCDEF]+|b[01]+|[0-9]\*\.?[0-9]\*|\"[^\"]\*\"|\'[^\']\*\')$/', $buffer . chr($int)) != 0)

{

$type = T\_LITERAL;

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

}

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;

// check if it is a keyword

if(preg\_match('/^(v|c|l|p|f|fo|w|re|if|el)$/', $buffer . chr($int)) != 0)

{

$type = T\_KEYWORD;

}

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

}

// check if it is an operator

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

{

$type = T\_OPERATOR;

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

}

// check for boolean and logical operators

elseif(preg\_match('/^(\<|\>|==|\<=|\>=|\|\||&&|\!)$/', $buffer . chr($int)) != 0)

{

$type = T\_LOGICAL;

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

}

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

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

{

$type = T\_WHITESP;

// check for precidence tokens

if($buffer . chr($int) == '(' || $buffer . chr($int) == ')')

{

$type = T\_PREC;

}

// check for a statement list token

elseif($buffer . chr($int) == '[' || $buffer . chr($int) == ']')

{

$type = T\_STATL;

}

// check for delimiter

elseif($buffer . chr($int) == ',')

{

$type = T\_DELIM;

}

$buffer .= chr($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();

}

}

}

}

}

?>

**Nodes.php:**

<?php

class AST

{

var $tree = array();

function accept(&$list)

{

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

next($list);

}

function parseAST(&$list)

{

reset($list);

$program = new Program();

$this->tree['AST'] = $program->parseProgram($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 Literal extends AST

{

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($list)->content) && intval(current($list)->content) == current($list)->content)

{

$this->parseIntLiteral($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)

{

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

}

function parseRealLiteral(&$list, $haspreceder)

{

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

}

function parseHexLiteral(&$list)

{

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

}

function parseBinaryLiteral(&$list)

{

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

}

function parseStringLiteral(&$list)

{

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

}

}

class Identifier extends AST

{

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 Expression extends AST

{

function parseExpression(&$list)

{

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

{

$literal = new Literal();

$this->tree = $literal->parseLiteral($list);

}

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

{

$identifier = new Identifier();

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

if(current($list)->type == T\_OPERATOR || current($list)->type == T\_LOGICAL)

{

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

$this->accept($list);

$expression = new Expression();

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

}

else

{

//prev($list);

}

}

elseif(current($list)->type == T\_OPERATOR || current($list)->type == T\_LOGICAL)

{

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

$this->accept($list);

$expression = new Expression();

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

}

else

{

$expression = new Expression();

$this->tree['Expression1'] = $expression->parseExpression($list);

if(current($list)->type == T\_OPERATOR || current($list)->type == T\_LOGICAL)

{

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

$this->accept($list);

}

else

{

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

}

$expression = new Expression();

$this->tree['Expression2'] = $expression->parseExpression($list);

}

return $this;

}

}

class Assignment extends Command

{

function parseAssignment(&$list)

{

$identifier = new Identifier();

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

if(current($list)->type == T\_OPERATOR && current($list)->content == '=')

{

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

$this->accept($list);

}

else

{

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

}

$expression = new Expression();

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

return $this;

}

}

class ParamCall extends AST

{

function parseParamCall(&$list)

{

do

{

$continue = false;

$expression = new Expression();

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

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

{

$this->accept($list);

$continue = true;

}

}while($continue);

return $this;

}

}

class FunctionCall extends Command

{

function parseFunctionCall(&$list)

{

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

{

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

$this->accept($list);

$paramcall = new ParamCall();

$this->tree['ParamCall'] = $paramcall->parseParamCall($list);

}

return $this;

}

}

class WhileLoop extends Command

{

function parseWhileLoop(&$list)

{

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

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

$expression = new Expression();

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

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

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

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

$commandlist = new CommandList();

$this->tree['CommandList'] = $commandlist->parseCommandList($list);

return $this;

}

}

class ForLoop extends Command

{

function parseForLoop(&$list)

{

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

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

$assignment = new Assignment();

$this->tree['Start'] = $assignment->parseAssignment($list);

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

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

$expression = new Expression();

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

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

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

$assignment2 = new Assignment();

$this->tree['Counter'] = $assignment2->parseAssignment($list);

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

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

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

$commandlist = new CommandList();

$this->tree['CommandList'] = $commandlist->parseCommandList($list);

return $this;

}

}

class ElseStatement extends Command

{

function parseElseStatement(&$list)

{

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

$commandlist = new CommandList();

$this->tree['CommandList'] = $commandlist->parseCommandList($list);

return $this;

}

}

class IfStatement extends Command

{

function parseIfStatement(&$list)

{

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

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

$expression = new Expression();

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

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

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

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

$commandlist = new CommandList();

$this->tree['CommandList'] = $commandlist->parseCommandList($list);

return $this;

}

}

class ParamDef extends AST

{

function parseParamDef(&$list)

{

do

{

$continue = false;

$expression = new Expression();

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

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

{

$this->accept($list);

$continue = true;

}

}while($continue);

return $this;

}

}

class FunctionDef extends Command

{

function parseFunctionDef(&$list)

{

$identifier = new Identifier();

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

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

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

$paramdef = new ParamDef();

$this->tree['ParamDef'] = $paramdef->parseParamDef($list);

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

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

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

$commandlist = new CommandList();

$this->tree['CommandList'] = $commandlist->parseCommandList($list);

return $this;

}

}

class Keyword extends AST

{

function parseKeyword(&$list)

{

switch(current($list)->content)

{

case 'v':

case 'c':

case 'l':

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

$this->accept($list);

$assignment = new Assignment();

$this->tree['Declaration'] = $assignment->parseAssignment($list);

break;

case 'p':

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

$this->accept($list);

$identifier = new Identifier();

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

break;

case 'fo':

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

$this->accept($list);

$forloop = new ForLoop();

$this->tree['ForLoop'] = $forloop->parseForLoop($list);

break;

case 'w':

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

$this->accept($list);

$whileloop = new WhileLoop();

$this->tree['WhileLoop'] = $whileloop->parseWhileLoop($list);

break;

case 'elif':

case 'if':

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

$this->accept($list);

$ifstatement = new IfStatement();

$this->tree['IfStatement'] = $ifstatement->parseIfStatement($list);

break;

case 'el':

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

$this->accept($list);

$elsestatement = new ElseStatement();

$this->tree['ElseStatement'] = $elsestatement->parseElseStatement($list);

break;

case 'f':

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

$this->accept($list);

$functiondef = new FunctionDef();

$this->tree['FunctionDef'] = $functiondef->parseFunctionDef($list);

break;

default:

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

}

return $this;

}

}

class Command extends AST

{

function parseCommand(&$list)

{

switch(current($list)->type)

{

case T\_KEYWORD:

$keyword = new Keyword();

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

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

break;

case T\_IDENTIFIER:

next($list);

if(current($list)->type == T\_OPERATOR && current($list)->content == '=')

{

prev($list);

$assignment = new Assignment();

$this->tree['Command'] = $assignment->parseAssignment($list);

}

else

{

prev($list);

$functioncall = new FunctionCall();

$this->tree['FunctionCall'] = $functioncall->parseFunctionCall($list);

}

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

break;

default:

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

}

return $this;

}

}

class CommandList extends Program

{

function parseCommandList(&$list)

{

if(!current($list)->type == T\_STATL || current($list)->content == '[') $this->accept($list);

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

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

do

{

// block structure niftyness

if(current($list)->type == T\_STATL && current($list)->content == '[')

{

$commandlist = new CommandList();

$this->tree[] = $commandlist->parseCommandList($list);

continue;

}

// parse a command

$continue = true;

$command = new Command();

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

if(current($list)->type == T\_STATL && current($list)->content == ']')

{

$this->accept($list);

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

$continue = false;

}

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

return $this;

}

}

class Program extends AST

{

function parseProgram(&$list)

{

$commandlist = new CommandList();

$this->tree['Program'] = $commandlist->parseCommandList($list);

return $this;

}

}

?>

**Printer.php:**

<?php

require\_once 'visitor.php';

class PrinterVisitor extends Visitor

{

function visit($tree)

{

if(is\_object($tree))

{

print get\_class($tree) . '(';

$f\_name = 'visit' . get\_class($tree);

if(method\_exists($this, $f\_name))

call\_user\_func(array($this, $f\_name), $tree);

else

$this->visitNode($tree);

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

print ')';

}

else

{

if(is\_array($tree))

{

foreach($tree as $key => $object)

{

$this->visit($object);

}

}

}

}

function visitNode($tree)

{

}

function visitIdentifier($tree)

{

print $tree->tree['Identifier'];

}

function visitLiteral($tree)

{

print current($tree->tree['Literal']);

}

}

?>