Margs Programming Language

A Simplified, Arithmetic Based JavaScript Implementation

Eric Kerr

August 12, 2010

The Ohio State University

CSE 655, Dr. Bob Mathis

Table of Contents

Summary

The Computer Science and Engineering 655 project during Summer quarter of 2010 was to design a programming language processor to take a high-level language (like Pascal, C, Lisp, or Scala) and map it to a machine-level language (like PL/0, Java byte-code, or Assembly) for simulation.

A programming language, Margs, was developed to encapsulate basic arithmetic functionality with the familiar syntax of JavaScript. A Grammar of the programming language is included on page 12 and is identical to the basic arithmetic capabilities of JavaScript. A Python interpreter was developed to tokenize the input Margs program, build an abstract syntax tree using a recursive descent parser, and compile the program to machine-level PL/0 code. Pypl0, a third-party Python PL/0 simulator [1], was then used to execute the machine code and prompt output. The Pypl0 library was modified to add support for I/O operations.

Margs is a simplified subset of the JavaScript language and includes support for assignment, variables, constants, iteration, if conditionals, if-else conditionals, subprograms, input and output, error messages, and function parameters. The Python Margs interpreter includes capabilities to compile and simulate sequentially without having to deal with the intermediary PL/0 program.

Abstract

This report will include a revised project proposal, the goals and scope of the language, a summary of the approach in compiling and interpreting the language, language syntax and semantics, an overview of processing support, a description of

how the Margs interpreter can be used, a complete program listing, and a list of tests run on the system. A discussion of the overall project will also be included with information on the project design, planning and execution as well as testing strategies and language capability demos.

Conclusion

The project involved many facets of programming language design including grammar development, tokenizing, recursive descent parsing, abstract syntax tree generation, and compiling. The programming language is fully capable of functioning as a base for non-trivial arithmetic computation as programs can be written to compute factorials, greatest common divisors, least squares regressions, and many other commonly used algorithms. Because the compiler and simulator are written in Python (which, by default, overflows integers to longs), arithmetic can be done which would not otherwise be possible.

The PL/0 language itself is fairly limited and does not include support for function parameters, if-else blocks, or true/false keywords. The Margs programming language abstracts all of this functionality away from PL/0 and makes it possible to include JavaScript-esque parameters, conditionals, and boolean keywords.

Revised Proposal

The course project will include a JavaScript parser compiled to PL/0 written in Python known as Margs. Margs will be an arithmetic subset of the language capabilities of JavaScript as it will not allow support for objects or arrays, but it will support assignment, conditionals, iteration, subprograms, input and output, error messages, and function parameters. Python will be used to parse Margs and map it to PL/0 machine-level language. Execution of the compiled PL/0 will be accomplished with the Pypl0 Python library which simulates PL/0 behavior.

The goals of the project are to be able to a program written in Margs (simplified JavaScript), break it into tokens with the parser, build and maintain a stack of memory to correspond with execution, and compile the language into PL/0, an executable machine-level language.

The Python portion of the project will be broken down into several parts. The code will be organized in an object oriented scheme with components to manage the various portions of execution. The parser will make use of a tokenizer which will recursively break down the input Margs. The compiler will use a traditional Abstract Syntax Tree to hold the data structures in memory and will interpret the incoming tokens with a recursive descent parser based on the Grammar. A central Error handling class will allow parsing and execution errors to be propagated to the user and will report detailed error messages where applicable (token information, runtime errors, etc.). Utility classes will also be used to break apart conditional expressions to allow support for IF-ELSE statements by taking the inverse of a conditional statement and generating two IF statements (to allow support in PL/0). Expressions will be parsed in a similar way such that parts are recursively parsed and evaluated preserving order of operations.

Project success will be determined by a language which is successfully able to evaluate tests with complete coverage. As Margs is a subset of JavaScript, a grammar will be constructed to encompass all valid syntax. A successful project will be able to parse tests completely covering the grammar. A successful project will also be able to evaluate the language in a quick and memory efficient manner. A good project will include proper data structures so parsing is quick. Execution of the parsing and compilation process should not take more than a few seconds for any of the tests provided.

Testing will be done in a top-down fashion using unit tests. Top-down testing will be used to test the program, at whole, for expected output so all of the components are proven to work well together. A python testing framework/suite will be used to aid in the unit testing process so everything is streamlined. Because the development and testing are written by the same person, the tests will be white-box as the testers are aware of how the inner workings of functions and procedures work.

Description of Approach

The development process was broken up into several stages. The first step was to find an adequate PL/0 simulator written in Python so the Margs programming language can be seamlessly simulated without having to worry about the intermediary compilation process. The next step was to modify the Pypl0 library to include support for basic I/O operations as the one I found did not natively include support for either. After I found a working PL/0 simulator (making the desired project goals feasible), I began to develop the grammar for Margs by looking at the grammar for JavaScript [2] and simplifying it to its core arithmetic components. After the grammar was developed and I had a valid list of language keyword tokens, I wrote the Parser class to tokenize the input Margs program. I then wrote the Compiler class to take the tokenized program and build an abstract syntax tree by consuming the input tokens. During this time, I developed an internal error logger to aid in the debugging process which uses Python's sys._getframe() method to dynamically report stack call information to trace where in the AST calls were coming from. To build the abstract syntax tree and consume the parsed input tokens, I took a literal translation of the grammar and developed an abstract Node class which serves as the basis for all declarations in the grammar. Each specialization of the abstract Node class is a declaration in the grammar and knows how to parse its own tokens, re-arrange itself for PL/0 compatibility, and generate its own compiled code. After I was able to build an syntax tree of the input program and generate semi-correct PL/0 code, I wrote a run.py script which allows an input Margs program to be compiled and simulated simultaneously.

I then wrote a test framework which parses all .margs files in the ./tests/ directory and compares the observed simulated output to the expected simulated output. The unit tests exposed a bunch of nuances about stylistic requirements about PL/0 which I was previously unaware of. I then added another method to the abstract Node class, clean(), which gets recursively called after build() for each Node type. The clean() method for each node knows how it must translate itself so that the compile() method can generate valid compiled PL/0. For example, the clean method for the Program(Node) class re-arranges all global variable and constant declaration statements to appear before functions are defined because of requirements in the PL/0 interpreter. The Function(Node) class also re-arranges variable declarations so that local variables are initialized before all other statements and before the Procedure's BEGIN PL/0 keyword.

After I had code generating valid PL/0 code for all of my test cases, I wanted to add support for JavaScript style parameters because PL/0 does not natively do this. I decided to pass everything through global variables and keep track of the parameter order to ensure the proper assignments are made before the call and inside the Procedure. I ensured that there would be no identifier name conflict by requiring that valid identifiers (function names and variables) be all lowercase and using an uppercase conversion of the Function's name appended with the parameter number being referenced. After parameters were added and I was comfortable with the level of functionality in the Margs programming language, I went through and thoroughly added unit tests to ensure valid execution for by complete coverage.

All development was done on a Macbook pro running Snow Leopard using TextMate. I kept a code repository by creating a symlink to my DropBox account [3] which stores versioned data on Amazon S3. I would have used a remote GIT repository, but since it was just me I opted for using DropBox since you don't need to commit periodically as every saved copy of your files is its own version. All code was written using Python 2.6 [4]. The class documentation was generated using Python's built in pydoc module [5].

Project Goals (scope of language)

The main goal of the project was to develop a fully functioning language parser and compiler. I wanted to add additional functionality to the existing PL/0 language, so I have added support for boolean keywords (true/false), function parameters, and IF-ELSE conditionals. Another goal of the project was also to automate as much of the testing process as possible. Every time I added functionality to the code in terms of finalizing statement types in the grammar or modifying the behavior of any portion of the project I verified that I had a test which would validate its behavior and re-run the test suite (by running $ python run.py tests). This made it extremely easy to mentally focus on development rather than trying to figure out when stuff was breaking afterwords.

I chose Python for the project because I am very familiar with it and have been using it professionally for several years. Python is platform agnostic and is well respected for its speed, brevity, and ease of use. Although it is not a compiled language, there are packages like Psycho [6] and PyPy [7] which natively compile Python to machine code. I chose JavaScript to model my syntax after because it is familiar to anyone with experience writing Java as well as C, the two most well-known programming languages [8].

I modified the Pypl0 library that I found to include support for Inout and Output operations, but all other functionality was left intact. By default Pypl0 writes all output to screen, but by setting `sys.stdout = open(filename, ‘w’)` the output can be redirected to a file - this is utilized for the -o command line option as well as the testing framework.

I chose Margs for the programming language's name because I like frozen margaritas and I planned the first draft of the project proposal while at Cazuela's drinking $2 margaritas at happy hour. Margs was only fitting.

Project Design

The project code was developed during a two week period with the majority of time being spent refactoring the Node inherited classes. The initial code took about a week to develop, but issues discovered during testing (like variable declaration) took another week or so to thoroughly implement and test.

My initial project proposal is very similar to what I ended up implementing. I originally intended to support hash-map data structures as well as arrays, but PL/0's limited functionality rendered both unrealistic if not impossible. Other than that my final project is very similar to what I originally intended.

I think the strongest part of my design is how well laid out the objects are in relation to one another. Looking at the code, associated documentation, and comments, it is very easy (relatively speaking) to figure out what's happening. I originally had a less object oriented design as I was initially developing the compiler, however I was able to refactor it once the functionality was finalized. This would prove to be essential for the clean() methods as they literally re-build the program's abstract syntax tree so the compile() methods are able to generate valid PL/0 code.

If I had to redo the project, I would pick a different machine level language because the functionality of PL/0 is very limiting. If I used Java byte-code, I would have been able to use a greater variety of data structures like strings, objects, hashmaps, and arrays, *but* the Margs language would be more complicated - there is beauty in simplicity. As it turns out, Margs is a very simple programming language to compute basic arithmetic.

Implementation of the parser and tokenizer was very straight forward because it involved checking a known list of literal keywords and doing a few regular expressions to determine which type of token an incoming text string was. I was able to implement the final parser in about five hours of development time.

The most difficult portion of development was re-arranging the abstract syntax tree to be able to produce valid machine-level PL/0 code. This involved dynamically generating new Statement objects and placing them throughout the program, ignoring certain flows of execution when necessary, and "rendering" the compiled portions of PL/0 at the right time.

Further extensions of the project might include developing a GUI to interact with the programming language easier. It would be straight forward to parse a program for all of the INPUT statements and render the corresponding input fields to a GUI window, and a list of programs to execute could be selected in a dropdown menu so common mathematical operations were made easier for workers. This might be helpful if one were to build a suite of financial calculation programs on top of the Margs programming language.

Language Syntax Implemented

Symbol	Syntax Rule
<program>	<elements> | empty_string
<elements>	<element> | <elements> <element>
<elements>	<stmt> | <function>
<stmt_list>	<stmt> | <stmt_list> <stmt>
<stmt>	<stmt_block> | <stmt_var> | <stmt_cond> | <stmt_io> | <stmt_fn_call> | <stmt_iteration> | <stmt_empty>
<stmt_block>	“{“ “}” | “{“ <stmt_list> “}”
<stmt_var>	“var” <decl_list> “;” | “const” <decl_list> “;” | <decl_list> “;”
<stmt_cond>	"if" "(" <condition> ")" <statement> "else" <statement> | "if" "(" <condition> ")" <statement>
<stmt_io>	"INPUT" <identifier> ";" | "OUTPUT" <expression> ";"
<stmt_fn_call>	<identifier> "(" <parameters> ")" ";" | <identifier> "(" ")" ";"
<stmt_iteration>	"while" "(" <condition> ")" <statement>
<stmt_empty>	“;”
<function>	"function" <identifier> "(" <parameters> ")" "{" <stmt_list> "}" | "function" <identifier> "(" ")" "{" <stmt_list> "}"
<parameters>	<identifier> | <parameters> "," <identifier> | <number> | <parameters> "," <number>
<decl_list>	<decl> | <decl_list> "," <decl>
<decl>	<identifier> | <identifier> "=" <expression>
<expression>	<boolean> | <identifier> | <term_list>
<term_list>	<term> | <addsub> <term> {<addsub> <term>}
<term>	<factor> {<muldiv> <factor>}
<factor>	<identifier> | <number> | "(" <term_list> ")"
<condition>	<expression> <comparison> <expression> | <boolean>

Symbol	Lexical Rule
<comparison>	"<" | "<=" | ">" | ">=" | "==" | "!="
<addsub>	"+" | "-"
<muldiv>	"*" | "/"
<identifier>	[a-z] {[a-z]}
<number>	"-" <num> | <num>
<num>	<digits> | <digits> "." <digits>
<digits	[0-9] {[0-9]}

Token Descriptions

Literal	Token	Description
<=	LTEQ	The "<=" comparison.
>=	GTEQ	The ">=" comparison.
<	LT	The "<" comparison.
>	GT	The ">" comparison.
+	PLUS	The addition operator.
-	MINUS	The subtraction operator.
*	MUL	The multiply operator.
/	DIV	The division operator.
(	LPAREN	Used to L bracket conditionals and order of operations.
)	RPAREN	Used to R bracket conditionals and order of operations.
{	LBLOCK	Used to L bracket blocks of statements
}	RBLOCK	Used to R bracket blocks of statements.
.	DOT	Used in numbers to signify decimal values

Literal	Token	Description
=	EQUALS	Used in assignment
==	EQUAL	The “==” comparison.
!=	NOTEQUAL	The “!=” comparison.
,	COMMA	Used to delimit parameters and declarations.
;	SEMI	Used to terminate statements.
true	TRUE	The true boolean.
false	FALSE	The false boolean.
var	VAR	Used to signify a variable identifier declaration.
const	CONST	Used to signify a constant identifier declaration.
function	FUNCTION	Used to begin a function declaration within the program.
INPUT	INPUT	Used to signify a user input stored in an identifier.
OUTPUT	OUTPUT	Used to signify an expression to print to screen.
if	IF	The beginning of an IF statement.
else	ELSE	The beginning of the ELSE portion of the IF statement.
while	WHILE	The beginning of a WHILE statement.
	ILLEGAL	An invalid token (not a literal, number, or identifier).
	NUMBER	A token which is a number.
	IDENTIFIER	An identifier (a function or a variable).

Language Demonstrations

Simple I/O (file: demos/margs/simple_io.margs)

Margs Program	var a; INPUT a; OUTPUT a + a;
Explained	Defines a variable, asks the user for input and prints out 2x the value.

Compiled PL/0	VAR
	  a;
	
	BEGIN
	  @ a;
	  ! a+a;
	END.

Input	INPUT a: 5
Simulated Output	10

Function Call (file: demos/margs/function_call.margs)

Margs Program	function callme(){
	    OUTPUT 5 * (4 + 5);
	}
	callme();
	callme();

Explained	Calls function twice to print out the value 5 * (4 + 5), which is 5 * 9, which is 45.

Compiled PL/0	PROCEDURE callme;
	BEGIN
	  ! 5*(4+5);
	END;
	
	BEGIN
	  CALL callme;
	  CALL callme;
	END.

Simulated Output	45 45

Parameters (file: demos/margs/parameters.margs)

Margs Program	function test(a, b, c){
	    OUTPUT a + b + (c * 2);
	}
	var a = 6;
	test(2, 3, a);

Explained	Calls function with parameters to output: 2 + 3 + (6 * 2) = 5 + 12 = 17

Compiled PL/0	VAR
	  a, TEST0, TEST1, TEST2;
	
	PROCEDURE test;
	VAR a, b, c;
	BEGIN
	  a := TEST0;
	  b := TEST1;
	  c := TEST2;
	  ! a+b+(c*2);
	END;
	
	BEGIN
	  a := 6;
	  TEST0 := 2;
	  TEST1 := 3;
	  TEST2 := a;
	  CALL test;
	END.

Simulated Output	17

While, If/Else (file: demos/margs/while_if_else.margs)

Margs Program	var i = 1;
	while(i <= 4){
	    if(i == 4){
	        OUTPUT 4;
	    } else {
	        OUTPUT 0;
	    }
	    i = i + 1;
	}

Explained	Iteration which prints out 0 three times and then 4 at the end.

Compiled PL/0	VAR
	  aa;
	
	BEGIN
	  aa := 1;
	  WHILE aa<=4 DO BEGIN
	    IF aa=4 THEN BEGIN
	      ! 4;
	    END;
	    IF aa#4 THEN BEGIN
	      ! 0;
	    END;
	    aa := aa+1;
	  END;
	END.

Simulated Output	0 0 0 4

Testing Strategy

Top-down testing was employed to test the desired functionality of the compiler. A custom test framework was developed to read in source program files from the ./tests directory (anything with a .margs extension), capture the observed output by the simulator, and compare it with the corresponding expected results. Because the testing and development was done by the same person, a white-box testing strategy was used. The test programs were designed to test the lowest level blocks in the grammar and test every combination of valid statement within the grammar. Because the test framework encounters 0 total errors after exhausting a complete coverage test suite, it is assumed that the interpreter works as desired. The test suite can be run by executing:

```
$ python run.py tests
```

The test suite lists all files which are tested, whether they were successful or not, and the total time the tests took. The test suite takes about .12 seconds to run on my machine. This means that the Python interpreter and simulator can compile and simulate 49 Margs programs at about 3 thousandths of a second each.

If given more time, I would develop unit tests to run bottom-up tests on the individual modules. I would also thoroughly test the third party PL/0 library as my assumption is that “it just works”. Testing did expose a fair amount of bugs within my code which I was able to fix. I didn’t know that global VAR declarations cannot have assignment inline, so I had to generate a new statement within the body to handle the assignment. I also did not know that their had to be semi-colons after the END in an if/while block for their to be more statements following. One major bug I fixed as local variable declaration within functions, and I was able to devise a consistent method of removing variable identifier declarations to before the BEFORE portion of the function.

The demonstrations featured on page 14 as well as the quick execution time (3 thousands of a second to compile and simulate a test file) leads me to believe that my project goals have been met. The Margs programming language builds upon PL/0 and adds desirable functionality, namely function parameters, in a way that does not break any existing PL/0 behavior.

Test Files

All test files are located in the ./tests directory and have the .margs extension.

The expected results have the same filename but have a .expected extension.

Test File	Description
add	Adding two numbers together in an expression.
assignment	Variable assignment
call	Calling a function
calls	Multiple calls to a function
compare_equal	Comparison operator: ==
compare_greater	Comparison operator: >
compare_greaterequal	Comparison operator: >=
compare_less	Comparison operator: <
compare_lessequal	Comparison operator: <=
compare_notequal	Comparison operator: !=
divide	Dividing two numbers in an expression
equation1	More complicated order-operations eq using variables
equation2	More complicated order-operations eq using variables
equation3	More complicated order-operations eq using variables
FALSE	False keyword in a condition
function	Function declaration with a call
function_if	IF statement within a called function
function_ifelse	IF-ELSE statement within a called function
function_ifelse_false	If-ELSE statement within a called function where the else statement executes

Test File	Description
function_inner_call	A function called from within another function
function_statements	Multiple statements within a function
function_var	Variables defined locally within a function
function_while	While conditional within a function
functions	Program with multiple functions
if	IF statement
if_if	IF statement within an IF statement
if_ifelse	IF-ELSE statement within an IF Statement
if_ifelse_false	IF-ELSE inside IF where the ELSE block executes
ifelse	IF-ELSE statement
ifelse_false	IF-ELSE statement where the ELSE block executes
multiply	Multiplying two numbers inside an expression
negative	Negative number being output (becomes 0-50)
negative_add	Adding a negative number
negative_sub	Subtracting a negative number
parameter	A function with 1 parameter
parameters	A function with multiple parameters
simple_output	OUTPUT statement with an expression
subtract	Subtracting two numbers
TRUE	True keyword in a condition
var	Variable identifer declaration
var_add	Adding two variable identifiers together
var_divide	Dividing two variable identifiers
var_multiply	Multiplying two variable identifiers

Test File	Description
var_sub	Subtracting two variable identifiers
vars	Multiple variable identifiers being declared
while	WHILE statement
while_call	CALL statement inside a WHILE statement
while_if	IF statement inside a WHILE statement
while_ifelse	IF-ELSE statement inside a WHILE statement

Description of Virtual Machine

The Python Margs interpreter/compiler writes machine code in PL/0 (modified for I/O). The PL/0 is simulated by a third-party library called Pypl0 [1], which was also modified to support I/O functionality. The compiler and simulator are all part of the same package, so "scripting language"-like functionality is possible by bypassing the compilation stage and simultaneously compiling and simulating.

To interface with the simulator, the input file is read using the PL/0 abstract syntax tree generator and then passed to the interpreter for execution. All output is rendered to system's stdout, however output can be routed to a file by setting `sys.stdout = open(filename, 'w')`.

File Layout

```
/Margs/
----- /run.py
----- /translator/
----- ------------ /compiler.py
```

```
----- ------------ /parser.py

----- ------------ /nodes.py

----- /simulator/ (third party)

----- ----------- /astgen.py

----- ----------- /astnodes.py

----- ----------- /interp.py

----- ----------- /main.py

----- ----------- /parser.py

----- ----------- /scanner.py

----- ----------- /utils.py
```

Class Descriptions

Compiler

__init__(self, tokens)

Compiler initializer.
tokens (list): Tokens produced by the parser

__str__(self)

String representation of the compiler

error(self, text)

The compiler found a grammar error and will report it to the user. All errors are fatal
text (string):Error message to report to user.

next(self, pos=0)

Return the type of token pos positions ahead in the token list.
pos (int, optional): The position to lookup within the tokens.

run(self)

Compiles the code

skip(self, pos=1)

Tell the compiler to skip/discard pos number of tokens.
pos (int, optional): tokens to discard

Properties
valid_comparison_tokens = list()
valid_expression_tokens = list()
valid_statement_tokens = list()

Parser

__init__(self, filename)
The Parser initializer
filename (string): The file to open and tokenize.

__str__(self)
String representation of the Parser.

loadFile(self)
Loads self.filename to be parsed

parse(self)
Runs the parser to generate tokens

tokenize(self, line, line_num)
Tokenizes the given line and appends to self.tokens

Token

__init__(self, text, _type, legal=True, line=0)
Token initiazer
text (string): Raw text in the token
_type (string): Type of token
legal (bool): Whether the token is valid or not
line (int): Line number the token was found on

__str__(self)
String representation of the Token.

Node (Abstract Base Class)

__init__(self, compiler=None)
Initializer for the Node.

build(self)
Use and consume the tokens to build AST.

clean(self)

```
        Prepare the node to be translated to PL/0.

    compile(self, indent=0)
        Generate the compiled PL/0 code for the given node.
        indent (int, optional): The level of indentation.
```

Node (Specializations)

All used as direct translations from the Grammar on page 12.

```
Comparison
Condition
Declaration
Declaration_List
Expression
Function
Parameters
Program
Statement
Statement_Conditional
Statement_Empty
Statement_Function_Call
Statement_IO
Statement_Iteration
Statement_List
Statement_Var
```

Usage Prerequisites

- Python must be installed on your machine. I used Python version 2.6 [4].

- Must have command-line access to invoke the interpreter.

- The Marg's Python package should be available on your machine.

Usage Guide

Since Python is a high-level "scripting language" ,there is no compilation process to build the compiler or simulator. To execute, make sure your current working directory is in the Margs directory (which contains run.py). In the Margs directory, run the following python command where action is a valid “Command Line Action” (listed below):

```
python run.py action [-o outputfile] infile
```

Command Line Actions

Action	Description
tokenize	Tokenize a Margs program
translate	Compile a program from Margs to PL/0
simulate	Simulate a program compiled in PL/0 assembly language
both	Compile/translate and simulate a program written in Margs
tests	Run all test files through compiler and simulator

External References

[1] http://code.google.com/p/pypl0/
[2] http://web.2point1.com/wp-content/uploads/2009/03/jas.bnf
[3] http://dropbox.com
[4] http://www.python.org/download/releases/2.6.5/
[5] http://docs.python.org/library/pydoc.html
[6] http://psyco.sourceforge.net/
[7] http://codespeak.net/pypy/dist/pypy/doc/
[8] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html