NYCGCS LANGUAGE

Ankur Goyal
ag3625@columbia.edu
August 14, 2015

Table of Contents

INTRODUCTION

The NYCGCS (New York City Garbage Collection Statistics) language is targeted towards fetching the garbage collection records and statistics from City of New York open data. This language is designed to be developer friendly and consists of various statements such as if, for, repeat, sendrequest etc. As a result, it frees the developer from writing extensive code just to create an object, send a REST request and then parse the response. Therefore, there is no pricing issue, library stability issue or EULA.

LANGUAGE TUTORIAL

Environment Requirements

The language compiler can be used on any Linux machine that has Ocaml. It also requires that the following Ocaml and Linux packages are installed in the machine:-

- M4
- Opam
- Cohttp
- Lwt

Compiling and Running the program

NYCGCS can take the code written in a text file and can execute it. It takes two arguments:-

- -b :- This would allow the developer to see the intermediate representation of the program that is being executed
- -e :- This would compile and execute the program

Example:-

```
Ocamlrun project.byte –b < forin.gs
Ocamlrun project.byte –e < forin.gs
```

Basic Types

NYCGCS treats everything it receives as a string. Therefore, all input constants to it should be enclosed within quotes.

Example:-

```
var i;
i="90";
```

An identifier can only be alphanumeric and must begin with an alphabet. A string constant can be alphanumeric. However, a string constant can contain only the following special characters: - space (“ “), period (“.”), hyphen (“-“) and an ampersand (“&”) .

Functions

Functions can be defined anywhere in the program, but they can be called only after they are defined. Function definitions in NYCGCS are defined in order to facilitate minimal coding for the developer. Function definition includes function identifier, a list of function arguments and the function body. The program should always contain a function called “main”, which serves as the point of entry.

Function signatures must be unique and they cannot be overloaded.

Example:-

```
beginfunction testudf t1 t2
{
printtoscreen(t1);
printtoscreen(t2);
}
endfunction
```

We can also return values from a function. The user can use the “return” keyword in order to return a value.

```
return “20”;
```

NYCGCS also provides a number of built-in-functions which the developer can use in order to perform various operations. The built-in-functions abstract the core functionalities and help in minimizing the code effort. NYCGCS provides the following built-in-functions :-

- printtoscreen :- This prints the value of the variable or the string constant.

```
printtoscreen(t1);
printtoscreen("i is less than j");
```

- fetchvalue :- This fetches the value of the object given its property name.

```
fetchvalue(o1 "papertonscollected");
```

- compare:- This compares two values. It returns 0 or 1 if the first value is greater than or equal to the second value. It returns -1 if the first value is less than the second value.

```
t1=compare(“90” “80”);
```

- sendrequest :- This function is used to request data from NYC Open data website. Internally, this function sends a REST request to the Open Data website, parses the response and creates an object from the response.

```
o1=sendrequest("brooklyn" "03");
```

- add :- This function can be used to add two numeric values.

```
i=add(i "10");
```

Condition block

NYCGCS also allows executing statements based on conditions. It exposes if-else keywords that can help the user in writing conditional flows.

```
var i;
var j;
i="90";
j="80";
if compare(i j) then
printtoscreen("i is greater than or equal to j");
else
printtoscreen("i is less than j");
```

Iterative Statements

NYCGCS also exposes iterative statements that can be used to execute statements repeatedly based on some condition. The iterative statements are:-

- for-in loop
  The for-in loop executes statements present in its statement body once for every argument present in its “in” clause. The value of the argument for which the loop is running can be accessed by using the identifier present in the “for” clause.

```
for i in ("120" "150" "180")
printtoscreen(i);

for l in (i j k)
{
printtoscreen(l);
}
```

- repeat-until loop
  The repeat-until loop executes statements present in its statement body until the condition present in the “until” clause evaluates to true (return value is greater than or equal to 0). The “until” clause should be a clause that returns a numeric value.

```
repeat
{
printtoscreen(i);
i=add(i "10");
}
until compare("90" i);
```

Comments

NYCGCS allows comments in the source code. A comment should be enclosed with “*” and “*/”. A comment can be alphanumeric and can contain the following special characters:- space (“ “), hyphen (“-“) and period (“.”).

LANGUAGE REFERENCE MANUAL

A1. Introduction

This manual describes the NYCGCS language specified by the Project Report submitted to CVN on 14th August, 2015 for approval as “New York City Garbage Collection Statistics Language”.

A2. Lexical Conventions

A program consists of variable declarations and function names. Once the scanning is done, the program has been reduced to a sequence of tokens.

A2.1 Tokens

There are six classes of token: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, tabs, newlines and comments as described below are ignored. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been separated into token up to a given character, the next token is the longest string of characters that could constitute a token.

A2.2 Comments

Any text between /* and */ is considered as a comment and is ignored by the compiler. The text can include alphanumeric characters. The only special characters that are allowed are: - {space}, {hyphen}, {period}. Comments do not nest, and they do not occur within string literals.

A2.3 Identifiers

An identifier is a string of letters and digits. The first character must be a letter. Upper and lower case letters are different. Identifiers may have any length.

A2.4 Keywords

The following identifiers are reserved for use as keywords and therefore, cannot be used otherwise:

`beginfunction`	`compare`	`return`	`else`
`if`	`then`	`endfunction`	`var`
`sendrequest`	`fetchvalue`	`printtoscreen`	`repeat`
`until`	`for`	`in`	`add`

A2.5 Constants

The languages can recognize only string constants. The string constants are represented by a series of letters and digits surrounded by quotation marks (“ “). A string constant can be alphanumeric and can contain only the following special characters :- space (“ “), hyphen (“-“), period (“.”) and ampersand (“&”).

Constant:
String-constants

A3. Syntax Notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal words and characters in `typewriter` style. Alternative categories are usually listed on separate lines. An optional terminal or nonterminal symbol carries the subscript “opt” so that, for example,

$\{ \textit{expression}_{opt} \}$

means an optional expression, enclosed in braces.

A4. Meaning of Identifiers

A4.1 Types

Variables and objects can be declared by using the `var` keyword. The variable can be assigned values either from a string constant, an identifier or from the return value of a function.

A5. Conversions

Since, the variables are declared by a `var`, it is possible to convert an object type to a string type and vice-versa.

A5.1 Object to string conversion

An object type can be converted to a string type by simple assignment.

A5.2 String to object conversion

A string type can be converted to an object type by simple assignment

A6. Expressions

An expression is a combination of variables, operators, constants and functions. The handling of overflow and other exceptions in expression evaluation is not defined by the language. The language does not handle any exceptions.

A6.1 String Expressions

A string expression can be an identifier or a literal.

String-expression:
Identifiers
Literals

An identifier is specified during declaration by using the `var` keyword. All identifiers should be declared at the beginning of the function.

Literals are string constants. It takes the form as discussed in A2.5.

A6.2 Assignment Expressions

The assignment operator = groups from right to left.

Assignment-expression:
Variable
Constant
Function calls
Conditional-expression
Variable assignment-operator assignment-expression

Assignment-operator:
=

These expressions are evaluated from right to left. Since, NYCGCS is a type-independent language, we can change a string type to an object type by assignment and vice-versa. Assignment operator expects a variable identifier on the left and a combination of variables, constants and method calls on the right. In an assignment operation, the value of the expression replaces that of the left operand.

A6.3 Function Calls

A function call is followed by an optional space-separated list of variables which constitute the arguments to the function.

In preparing for the call to a function, a copy is made of each argument; all argument-passing is strictly by value. A function may change the values of its parameter objects, which are copies of the argument expressions, but these changes cannot affect the values of the arguments.

Func-call:

Identifier(argument-list$_{opt}$)

Identifier(func-call)

A7. Declarations

Declarations specify the interpretation given to each identifier. Once an identifier is defined, space is allocated to it.

A7.1 Type Specifiers

The type-specifiers are

Type-specifier:
var

A value cannot be assigned to a variable at its point of declaration. The value must be assigned separately as an *assignment-expression*.

A7.2 Function Declaration

Function declarations are created along with their definition and have the following format :

Function-declaration:
start-function identifier identifier-list$_{opt}$ {statement-list} end-function

start-function:
beginfunction

end-function
endfunction

The `beginfunction` is the keyword that signifies that a function is starting. The identifier is the name of the function. The *identifier-list* is optional. If present, the argument-list is a space delimited list of identifiers (formal parameters).

The function definition occurs inside curly braces. The definition can contain any number of *statements*, *expressions*, and *declarations*. A `return` statement is not required at the end of the function. If we assign the result of a function that does not contain a return statement, then the result of the last operation in the function is used as the assignment value. You cannot overload functions.

The `endfunction` signifies that the function has finished.

A8. Statements

Statements are executed in sequence. Statements should be ended by a semi-colon (";"). Statements fall into several groups:-

Statement:
Expression-statement
Return-statement
Statement-block
Conditional-statement
Iterative-statement
Variable-declaration
Function-declaration

Statements are separated by a semi-colon and a series of statements will be called a *statement-list*.

Statement-list:
Statement
Statement-list statement

A8.1 Expression-statement

Expression-statement:
Expression

An expression statement is made up of one or more expressions as defined in A6.

A8.2 Return-statement

A return statement is included in a function and indicates the value that would be returned by that function.

Return-statements:
return expression

A8.3 Statement-block

A statement block groups multiple statements together. We cannot declare a variable inside a statement block unless that statement block is also the body of a function. A statement-block can access the variables defined in the parent block.

Statement-block:
{ statement-list }

A8.3 Conditional statements

Conditional statements control the flow of a program by performing different sets of statements depending on some numeric value.

Conditional-statement:
If expression then statement-block
If expression then statement-block else statement-block

The expression after `if` must be an expression that returns a value. If the value that is returned by the expression is greater than or equal to 0, then execute the corresponding *statement-block* and jump out of the conditional expression. If the value returned by the expression is less than 0, then execute the *statement-block* corresponding to the `else` statement (if present) and then jump out of the conditional expression.

A8.4 Iterative statements

The Iterative statements execute a statement-block depending on some particular condition. They are grouped into "`for-in`" and "`repeat-until`" loops.

Iterative statements:
For identifier in (argument-list) {statement-list}
Repeat {statement-list} until expression

- `for-in` loop
  The `for-in` loop executes the statements present in the statement-list once for each argument in the argument-list. The value of the current loop argument can be accessed by using the identifier.
- `repeat-until` loop
  The `repeat-until` loop executes the statements present in the *statement-list* till the expression returns a value greater than or equal to 0. If the expression returns a value less than 0, then the control would exit the loop.

A9. Built-In-Functions

The language contains many built-in-functions that can help the developer to code efficiently with minimal effort.

A9.1 Compare

compare(expression expression)

The `compare` function can be used to compare two *expressions*. Both the *expressions* should evaluate to a number. If the first *expression* is equal to the second expression, then the function returns "0". If

the first *expression* is greater than the second *expression*, then the function returns “1”. If the first *expression* is less than the second *expression*, then the function returns “-1”.

A9.2 Add

add(expression expression)

The `add` function can be used to add two *expressions*. Both the *expressions* should evaluate to a number. It returns the result of the addition.

A9.3 Fetchvalue

fetchvalue(object-identifier string-identifier)

The `fetchvalue` function can be used to fetch value of a property from an *object*. The object-identifier indicates an object whereas the string-identifier denotes the property. The string-identifier can take the following values :-

- "papertonscollected”
- "mgptonscollected”
- "communitydistrict”
- "borough”

A9.4 Sendrequest

sendrequest(string-identifier string-identifier)

The `sendrequest` function can be used to fetch an object from NYC Open Data website. The first string-identifier indicates the “borough” and the second string-identifier denotes the “communitydistrict”. Data will be fetched by sending the “borough” and “communitydistrict” values to the REST API.

A9.5 Printtoscreen

printtoscreen(expression)

The `printtoscreen` function prints the value returned by the *expression* to the screen.

PROJECT PLAN

Process

Planning and Specification

Specification for the NYCGCS language came from the language proposal. It was eventually modified to be more consistent and logical. The entire project was broken down into different parts. The parts were prioritized and they were executed in order of their priority. I followed an "automated" process. In this process, every time I did a build, it automatically triggered the testing script thus executing the regression testing.

Development

I started working on the scanner, parser and the abstract syntax tree. The earlier versions of the scanner, parser and the abstract syntax tree were built to recognize keywords. Once that was done, I moved on to implementing the execution process for user defined functions and pre-defined functions. While implementing sendrequest method, I also had to work on parsing the JSON response that I was getting for my web request. After I had tested the code, I then started work on implementing if-else condition. In the end, I also decided to implement loops such as For loop and Repeat-Until loop (similar to while loop in C). Loops have a similar implementation architecture to the if-else condition (check for condition – branch if required) and therefore, they were implemented quickly.

Testing

The testing driver script (testall.sh) was added early on in the development process, so that tests could be added in easily. Every time a new feature was implemented, I added in a new test which verified that feature was working. Moreover, the testing script is tied with the build script. As a result, regression testing was done automatically whenever the build script was ran.

Programming Style Guide

When I was writing the compiler I followed a particular coding style. It included the following items :-

- Indent as much as possible in order to differentiate between nested blocks. As an example, if there is a "if" block, then its body should be indented one level further.
- Comment reasonably

Project Timeline

The timeline for the project looked like the following:-

- June 10 :- Project Proposal submitted
- June 20 :- Project broken down into modules. Initial commit.
- July 12 :- Project sending request to NYC Open Data and parsing it
- July 20 :- Project generating bytecodes and doing simple execution
- August 2:- Loops implemented. Project executing all commands

Roles and Responsibilities

I was the only person working on this project and therefore, I was working as the Manager, Language Guru, System Architect, Developer and Tester.

Development Environment

The project was built on an Ubuntu 14.04 (which was installed as a second OS) on my laptop. The compiler was written in OCaml. I am using "Make" to compile my code. The Makefile executes the Ocamlbuild command.

Project log

June 20	Initial commit of the code.
July 8	Incorporate features similar to Microc
July 12	Sending request to Open Data
July 18	Start printing bytecode
July 19	Checking in initial execute.ml
July 25	Bug Fixes
July 26	Initial check in of if-else
August 1	Complete if-else
August 2	Complete For loop, Repeat-until
August 5	Adding function
August 10	Bug fixes

LANGUAGE EVOLUTION

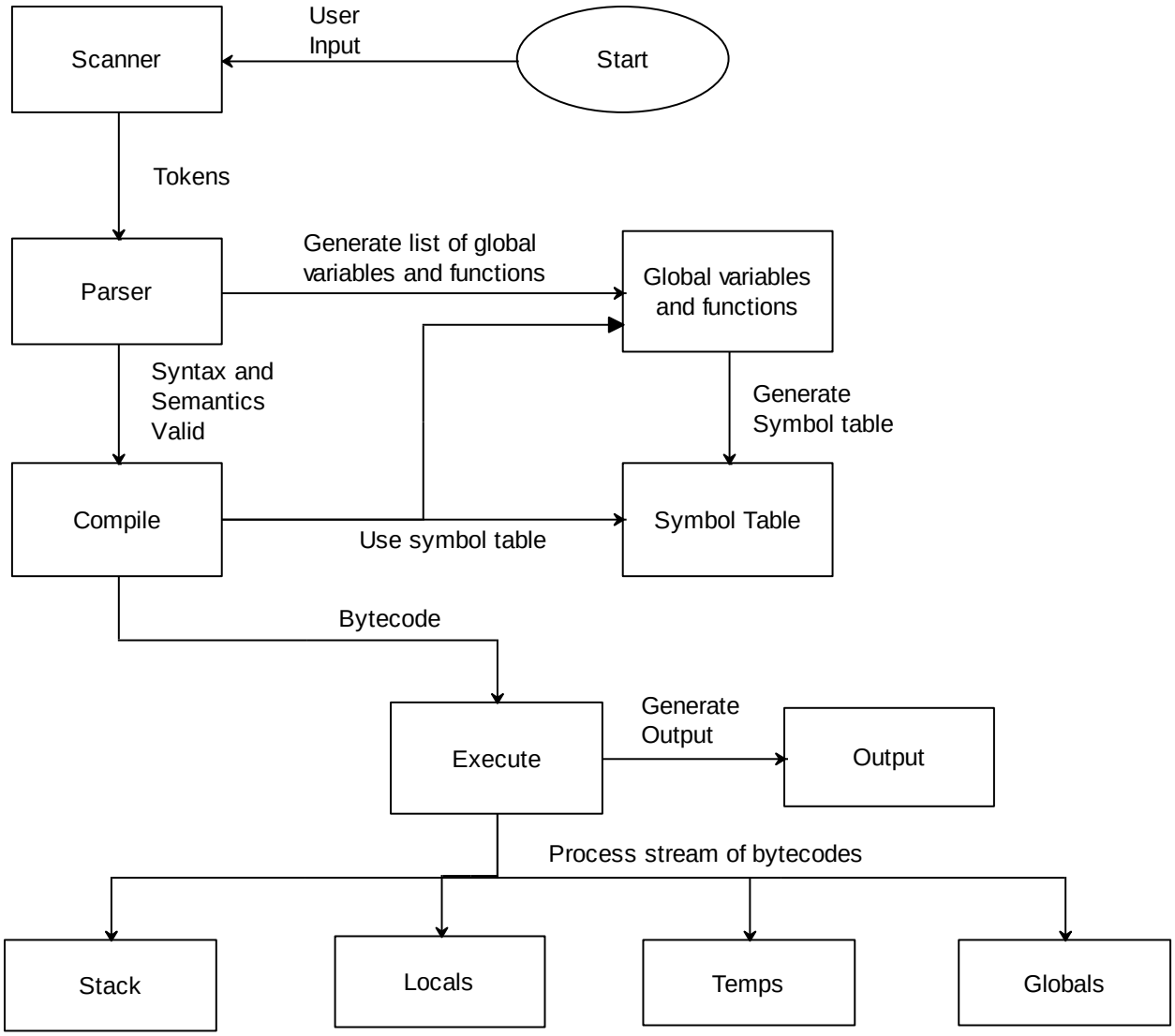
In the language proposal that I had submitted, I had mentioned that I would also be giving the developer the ability to change the web request URL and to create custom types. I had also mentioned that the developer can access two macros: - MAX_NUM (equal to 32767) and MIN_NUM (equal to -32768).

However, the language that I have implemented does not contain those items and instead contains the following additional functionalities:-

- Comments
- Repeat-until loop
- For-in loop
- Add function

One other item that I omitted from my language is giving a number and a string keyword. Every value in NYCGCS is being considered as a string and therefore, there is no “string” or “number” keyword. The user has to use “var” keyword in order to declare a type.

TRANSLATOR ARCHITECTURE



Scanner

Scans the input program and produces a tokenized output. The scanner discards all whitespace and comments and reports an error if any invalid character sequences are encountered. The scanner was written for ocamllex.

Parser

Parses the tokenized output of the scanner and generates the syntax tree. This tree was generated after the parser determines that the input program was written in a correct manner.

It generates information about the global variables and functions that are used for setting up the “environment” in Compile phase

Compile

It creates the symbol tables using the information about the global variables and functions that were generated in Parser phase. It then uses that information to generate the bytecode information

Execute

This is the phase where we process the bytecode information and execute the operations that are mentioned in that bytecode. If we need to send a web request to the NYC Open Data, we send that request and parse the response.

Internal Implementation

The compiler internally uses 4 arrays to keep track of the various data moving around:-

- Locals :- This array is used for keeping track of the values for the local variables.
- Globals :- This array is used for keeping track of the values for the global variables.
- Temps :- We store the temporary values in this array. When we execute any operation, we also store the result in this array unless the compiler determines that storing is not required.
- Stack :- This data structure is useful when executing the branch operation and also when returning from a branch. When we branch, we store the following items in the array :-
    - Program counter of the next statement in caller function (which is invoking the branched method).
    - The caller function’s offset in the locals array. This offset is comparable to the frame pointer in the activation record.
    - Index of the last usable local element corresponding to the caller function.

Intermediate Representation Details

- LLocal (i) -> This loads the element present at the ‘ith’ index of the local array
- LGlobal (i) -> This loads the element present at the ‘ith’ index of the global array
- Bin -> This performs the binary operation
- Jsr (i) -> This jumps to the ‘ith’ location of the intermediate representation
- Gnf (i) -> This gets the next field at offset ‘I’. This is useful when we are fetching a property of an object.
- PLocal (i) -> This prints the value of the ‘ith’ element of the local array
- PGlobal (i) -> This prints the value of the ‘ith’ element of the global array
- PTemp -> This prints the last value that is present on the temporary array.

- SendReq -> This sends the request using the values that are present on the temporary array
- SLocal (i) -> This saves the value present on the temporary array to the 'Ith' location of the local array
- SGlobal (i) -> This saves the value present on the temporary array to the 'ith' location of the global array
- STemp (s) -> This saves the string s on the temporary array
- Hlt -> Stops the program
- Ent (i) ->It tells the number of local variables present
- Rts (i) -> This tells the total number of variables (locals and formals) present. This helps in resetting the local array to its original state
- Skip (i) -> This skips over 'I' instructions present in the instructions array
- Loop -> This indicates that a loop would start. This would push the location of the next statement onto the stack array. Therefore, we consider a loop as an anonymous function.
- GoBackLoop -> This goes back to the last Loop bytecode that was encountered. It gets the location from the stack array and resets the temporary array
- EndLoop -> This corrects the entries in the stack. As a result, the loop is no longer accessible
- LSkip (i) -> This instruction is used in the "For" loop and can be used to skip over 'I' instructions. It is different from the Skip bytecode.
- EndForLoop -> This corrects the entries in the stack array so that the "For" loop would restart at the correct location

Loops and Conditions Internal Details

- If –else condition :-
  - It is converted to a series of Load, Compare and Skip instructions.
- Repeat-Until loop :-
  - It is converted a series of Loop, Load, Compare, Skip, GoBackLoop and EndLoop instructions.
- For-in loop :-
  - For-in loop can be used for iterating over a list of values. Internally, it is translated to a series of Load, SLocal, SGlobal, LSkip, Skip and EndForLoop instructions.

TEST PLAN AND SCRIPTS

Send-Request

This program demonstrates sending a web request and then assigning the returned value to another object.

Source Program (sendrequest.gs)

```
beginfunction main
{
var o1;
var o2;
o1=sendrequest("brooklyn" "03");
printtoscreen(fetchvalue(o1 "papertonscollected"));
printtoscreen(fetchvalue(o1 "mgptonscollected"));
o2=o1;
printtoscreen(fetchvalue(o2 "papertonscollected"));
printtoscreen(fetchvalue(o2 "mgptonscollected"));
}
endfunction
```

Intermediate Representation

```
0 Jsr 2
1 Hlt
2 Ent 2
3 STemp "brooklyn"
4 STemp "03"
5 SendReq
6 SLocal 1
7 LLocal 1
8 Gnf 0
9 Fetch
10 PTemp
11 LLocal 1
12 Gnf 1
13 Fetch
14 PTemp
15 LLocal 1
16 SLocal 2
17 LLocal 2
18 Gnf 0
19 Fetch
20 PTemp
```

```
21 LLocal 2
22 Gnf 1
23 Fetch
24 PTemp
25 Rts 2
```

Output

```
"271"
"246"
"271"
"246"
```

For-in loop

This program demonstrates a for-in loop. We are iterating over a set of values and then printing them to the screen.

Source Program (forin.gs)

```
beginfunction main
{
var i;
var j;
var k;
var l;
for i in ("120" "150" "180")
printtoscreen(i);
i="90";
j="100";
k="80";
for l in (i j k)
printtoscreen(l);
}
endfunction
```

Intermediate Representation

```
0 Jsr 2
1 Hlt
2 Ent 4
3 STemp "120"
```

```
4 LSkip 5
5 STemp "150"
6 LSkip 3
7 STemp "180"
8 LSkip 1
9 Skip 3
10 SLocal 1
11 PLocal 1
12 EndForLoop
13 STemp "90"
14 SLocal 1
15 STemp "100"
16 SLocal 2
17 STemp "80"
18 SLocal 3
19 LLocal 1
20 LSkip 5
21 LLocal 2
22 LSkip 3
23 LLocal 3
24 LSkip 1
25 Skip 3
26 SLocal 4
27 PLocal 4
28 EndForLoop
29 Rts 4
```

Output

```
"120"
"150"
"180"
"90"
"100"
"80"
```

Repeat-Until Loop

This program demonstrates a repeat-until loop. We are printing multiples of 10 between 10 and 90.

Source Program(repeatuntil.gs)

```
beginfunction main
{
var i;
i="10";
/* Repeat until i is less than or equal to 90 */
repeat
{
printtoscreen(i);
i=add(i "10");
}
until compare("90" i);
}
endfunction
```

Intermediate Representation

```
0 Jsr 2
1 Hlt
2 Ent 1
3 STemp "10"
4 SLocal 1
5 Loop
6 PLocal 1
7 LLocal 1
8 STemp "10"
9 Add
10 SLocal 1
11 STemp "90"
12 LLocal 1
13 Compare
14 Skip 1
15 GoBackLoop
16 EndLoop
17 Rts 1
```

Output

```
"10"
"20"
"30"
"40"
"50"
"60"
"70"
"80"
"90"
```

Test Suite

Every time a new functionality was added to the language, I added in a few tests to test that functionality. A shell script (testall.sh) was written which automated running through these tests. The shell script compiles and executes a specific program, and diffs the output of execution against the expected output. This shell script was called in the MakeFile. Therefore, anytime I did a build, it would also execute the automated script. As a result, I was performing regression testing on every build that I did.

CONCLUSIONS

I am currently working as a full-time IT professional in AT&T R&D Labs and it is hard to balance both work and studies. There are some things that I learned along the way. They are:-

1. Concentrate on the basic things first. I was thinking on how to implement “for” and “while” loops. However, at the core, “for” and “while” loops are only condition check and branch statements. They are similar to how “if-else” condition is implemented. So, once I implemented “if-else”, “for” and “while” (“repeat-until”) were easy.
2. Every project (whether big or small) needs some kind of automated regression test tool. I coupled the regression test script with the build process and as a result, it helped me in automatically detecting bugs whenever I did a build.

Advice to future students

1. Studying while working is hard. Therefore, break the language into different smaller modules and assign those modules to “Sprints”. The code prototype should be built first.
2. Start as soon as possible
3. Keep an automated test script around and if possible, tie it to your build process. This way, whenever a build is done, it would automatically trigger the test script.

FULL CODE LISTING

Ast.ml

```
type op = Compare | FetchValue | Add
type sendop = SendRequest
type printop = Print

type expr =
  Id of string
| Text of string
| PrintVal of printop * expr
| CompareVal of expr * expr
| FetchVal of expr * string
| AddVal of expr * expr
| Assign of string * expr
| SendWebRequest of sendop * expr list
| Call of string * expr list
| Noexpr

type stmt =
  Block of stmt list
| Expr of expr
| Return of expr
| If of expr * stmt * stmt
| Repeat of stmt * expr
| Forin of string * expr list * stmt

type func_decl = {
        fname : string;
        formals : string list;
        locals : string list;
        body: stmt list;
}

type program = string list * func_decl list
```

Bytecode.ml

```
type bstmt =
            LLocal of int (* Load from local *)
          | LGlobal of int (* Load from global *)
          | Bin of Ast.op (* Perform binary operation *)
          | Jsr of int (* Jump to location *)
          | Gnf of int (* Gets next field at offset *)
          | PLocal of int     (* Prints the value present on local array location *)
          | PGlobal of int (* Prints the value present on global array location *)
          | PTemp     (* Prints the last value present in temporary *)
          | SendReq  (* Sends the request using the values present in call array *)
          | SLocal of int (* Saves the value at specified location in local array *)
          | SGlobal of int (* Saves the value at specified location in global array *)
          | STemp of string (* Saves the string to the temporary array *)
          | Hlt (* Halt the program *)
          | Ent of int    (* Tells the number of local variables present *)
          | Rts of int    (* Tells the total number of variables (local+formals) present. This helps in resetting
the local array to its original state *)
          | Skip of int   (* Skips over 'i' instructions present in instructions array*)
          | Loop              (* A loop would start. This would push the location of the next statement onto
the stack and treat a loop as an anonymous inner function *)
          | GoBackLoop    (* Go back to the last loop. Get the location from the stack and start execution
again. Reset temporary array *)
          | EndLoop         (* This corrects the entries in the stack so the anonymous function (loop) is no
longer accessible*)
          | LSkip of int      (* This skips the i instructions while iterating the for loop. These skipped
instructions are executed in the next iteration *)
          | EndForLoop    (* This corrects the entries in the stack so that the for loop would restart at the
correct location *)

type prog = {
   num_globals : int;   (* Number of global variables *)
   text : bstmt array; (* Code for all the functions *)
  }

let string_of_stmt = function
          LLocal(i) -> "LLocal " ^ string_of_int i
          | LGlobal(i) -> "LGlobal " ^ string_of_int i
          | PTemp -> "PTemp"
          | Jsr(i) -> "Jsr " ^ string_of_int i
          | Bin(Ast.Compare) -> "Compare"
          | Bin(Ast.FetchValue) -> "Fetch"
          | Bin(Ast.Add) -> "Add"
          | Gnf(i) -> "Gnf " ^ string_of_int i
          | PLocal(i) -> "PLocal " ^ string_of_int i
          | PGlobal(i) -> "PGlobal " ^ string_of_int i
```

```
          | SendReq -> "SendReq"
          | SLocal(i) -> "SLocal " ^ string_of_int i
          | SGlobal(i) -> "SGlobal " ^ string_of_int i
          | STemp(s) -> "STemp " ^ s
          | Hlt -> "Hlt"
          | Ent(i) -> "Ent " ^ string_of_int i
          | Rts(i) -> "Rts " ^ string_of_int i
          | Skip(i) -> "Skip " ^ string_of_int i
          | Loop -> "Loop"
          | GoBackLoop -> "GoBackLoop"
          | EndLoop -> "EndLoop"
          | LSkip(i) -> "LSkip " ^ string_of_int i
          | EndForLoop -> "EndForLoop"

let string_of_prog p =
  string_of_int p.num_globals ^ " global variables\n" ^
  let funca = Array.mapi
     (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
  in String.concat "\n" (Array.to_list funca)
```

Compile.ml

```
open Ast
open Bytecode

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
    function_index : int StringMap.t; (* Index for each function *)
    global_index   : int StringMap.t; (* "Address" for global variables *)
    local_index    : int StringMap.t; (* FP offset for args, locals *)
  }

(* val enum : int -> 'a list -> (int * 'a) list *)
(* This creates "addresses"*)
(* enum 1 1 [8; 7; 6];;
- : (int * int) list = [(1, 8); (2, 7); (3, 6)]
*)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)
(* This creates "Symbol table" *)
(* when (int * int) list = [(1, 8); (2, 7); (3, 6)] is passed,
then it says that 8 is present at location 1, 7 is present at location 2 and so on *)
let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

let translate (globals, functions) =

  (* Allocate "addresses" for each global variable *)
  let global_indexes = string_map_pairs StringMap.empty (enum 1 0 globals) in

  (* Assign indexes to function names; built-in "printtoscreen" is special *)
  let funcmap1 = StringMap.add "printtoscreen" (-1) StringMap.empty in
  let funcmap2 = StringMap.add "fetchvalue" (-2) funcmap1 in
  let funcmap3 = StringMap.add "compare" (-3) funcmap2 in
  let built_in_functions = StringMap.add "sendrequest" (-4) funcmap3 in
  let function_indexes = string_map_pairs built_in_functions
      (enum 1 1 (List.map (fun f -> f.fname) functions)) in

  (* Translate a function in AST form into a list of bytecode statements *)
  let translate env fdecl =
    (* Bookkeeping: FP offsets for locals and arguments *)
    let num_formals = List.length fdecl.formals
```

```
   and num_locals = List.length fdecl.locals
   and formal_offsets = enum 1 1 fdecl.formals
   (* locals begin immediately after formals. Consider formals as local variables in code*)
   and local_offsets = enum 1 ((List.length fdecl.formals)+1) fdecl.locals in
   let env = { env with local_index = string_map_pairs
                        StringMap.empty (formal_offsets @ local_offsets) } in
(* Each operation would save its value in a temporary array. As a result, we should be able to execute
statements like "printtoscreen (Compare something)" *)
(* Save operation should also be able to use the temporary array. Therefore, we could execute
statements like "var t1=Compare something" *)
(* SendReq operation is where temporary array is most useful. Since, sendrequest uses 4 arguments, we
would store the local *)
   let rec expr = function
           Text s -> [STemp (s)]
     | Id s -> (* Check if s begins with double quotes. If it does, then add it to the temporary array
otherwise, it is a variable*)
           if s.[0] = '"' then [STemp (s)] else
            (try [LLocal (StringMap.find s env.local_index)]
        with Not_found -> try [LGlobal (StringMap.find s env.global_index)]
        with Not_found -> raise (Failure ("undeclared variable " ^ s)))
    |      FetchVal (e1,s) -> (match s with
                     "\"papertonscollected\"" -> expr e1 @ [Gnf 0] @ [Bin FetchValue]
                   | "\"mgptonscollected\"" -> expr e1 @ [Gnf 1] @ [Bin FetchValue]
                   | "\"communitydistrict\"" -> expr e1 @ [Gnf 2] @ [Bin FetchValue]
                   | "\"borough\"" -> expr e1 @ [Gnf 3] @ [Bin FetchValue]
                   | _ -> raise (Failure ("Unrecognized field name" ^ s))
                  )
    | AddVal (e1,e2) -> expr e1 @ expr e2 @ [Bin Add]
    |      CompareVal (e1, e2) -> expr e1 @ expr e2 @ [Bin Compare]
    | PrintVal (op, e1) -> (match e1 with
                     Text s -> [STemp(s)] @ [PTemp]
                     | Id s -> if s.[0] = '"' then [STemp(s)] @ [PTemp] else (try [PLocal (StringMap.find s
env.local_index)]
           with Not_found -> try [PGlobal (StringMap.find s env.global_index)]
           with Not_found -> raise (Failure ("undeclared variable " ^ s)))
                     | CompareVal (e1, e2) -> expr e1 @ expr e2 @ [Bin Compare] @ [PTemp]
                     | AddVal (e1,e2) -> expr e1 @ expr e2 @ [Bin Add] @ [PTemp]
                     | FetchVal (e1, s) -> (match s with
                      "\"papertonscollected\"" -> expr e1 @ [Gnf 0] @ [Bin FetchValue] @ [PTemp]
                   | "\"mgptonscollected\"" -> expr e1 @ [Gnf 1] @ [Bin FetchValue] @ [PTemp]
                   | "\"communitydistrict\"" -> expr e1 @ [Gnf 2] @ [Bin FetchValue] @ [PTemp]
                   | "\"borough\"" -> expr e1 @ [Gnf 3] @ [Bin FetchValue] @ [PTemp]
                   | _ -> raise (Failure ("Unrecognized field name"^s))
                  )
                     | _ -> raise (Failure ("Cannot call print on non-returning expression"))
           )
    | Assign (s, e) -> expr e @
            (try [SLocal (StringMap.find s env.local_index)]
```

```
            with Not_found -> try [SGlobal (StringMap.find s env.global_index)]
            with Not_found -> raise (Failure ("undeclared variable " ^ s)))
    | SendWebRequest (op, e1) -> (List.concat (List.map expr (List.rev e1))) @ [SendReq]
    | Call (fname, actuals) -> (try
            (List.concat (List.map expr (List.rev actuals))) @
            [Jsr (StringMap.find fname env.function_index) ]
      with Not_found -> raise (Failure ("undefined function " ^ fname)))
    | Noexpr -> []

(* This returns the bytecodes for a forin loop. Example :- if the list is (60 80), then the bytecode would be
:-
            STemp 60
            LSkip 3 // 3 because we have an additional skip statement at the end of list in forin
            STemp 80
            LSkip 2 *)
  in let rec loop_ignore_cmd = function
            [] -> []
    | hd :: [] -> expr hd @ [LSkip 1]
    | hd :: tl -> expr hd @ [LSkip (1 + (2 * List.length tl))] @ (loop_ignore_cmd tl)

  in let rec stmt = function
            Block sl    ->  List.concat (List.map stmt sl)
    | Expr e      -> expr e
    | Return e    -> expr e (* The returned value would automatically be saved in temporary array *)
    | If (p, t, f) -> let t' = stmt t and f' = stmt f in
            expr p @ [Skip(1 + List.length t')] @
            t' @ [Skip(List.length f')] @ f'
    | Repeat(sl, e) -> let t' = stmt sl in
            [Loop] @ t' @ expr e @ [Skip(1)] @ [GoBackLoop] @ [EndLoop]
    | Forin(s,el,sl) -> let t'=stmt sl in
            (* Adding 2 to the skip so that it can skip over EndForLoop and SLocal/SGlobal also *)
            (loop_ignore_cmd (List.rev el)) @ [Skip (2 + (List.length t'))] @
             (try [SLocal (StringMap.find s env.local_index)]
             with Not_found -> try [SGlobal (StringMap.find s env.global_index)]
             with Not_found -> raise (Failure ("undeclared variable " ^ s)))
            @ t' @ [EndForLoop]

  in let bytecodearray = stmt (Block fdecl.body)
  in [Ent (num_locals+num_formals)] @      (* Entry: allocate space for locals *)
   bytecodearray @        (* Body *)
  [Rts (num_formals+num_locals)]   (* Default = return 0 *)

 in let env = { function_index = function_indexes;
                       global_index = global_indexes;
                       local_index = StringMap.empty } in

 (* Code executed to start the program: Jsr main; halt *)
 let entry_function = try
```

```
  [Jsr (StringMap.find "main" function_indexes); Hlt]
with Not_found -> raise (Failure ("no \"main\" function"))
in

(* Compile the functions *)
let func_bodies = entry_function :: List.map (translate env) functions in

(* Calculate function entry points by adding their lengths *)
let (fun_offset_list, _) = List.fold_left
   (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0) func_bodies in
let func_offset = Array.of_list (List.rev fun_offset_list) in

{ num_globals = List.length globals;
  (* Concatenate the compiled functions and replace the function
     indexes in Jsr statements with PC values *)
  text = Array.of_list (List.map (function
          Jsr i when i > 0 -> Jsr func_offset.(i)
    | _ as s -> s) (List.concat func_bodies))
}
```

Execute.ml

```
open Ast
open Bytecode
open Lwt
open Cohttp
open Cohttp_lwt_unix

let execute_prog prog =
        let locals = Array.make 1024 "0" (* This is local array *)
        and globals = Array.make prog.num_globals "0"
        and temps = Array.make 100 "0" (* Temporary array *)
        and stack = Array.make 100 0
        and currentln j l = if j < l then l else j
        and sentinel = "thisisanobject" in (*This would hold the program states and would be useful for
jumping to functions and returning. *)
(* sp = Stack pointer,
  pc = index used to go through bytecode,
  tc = index used to go through temporary array,
  ln = offset into the local array. This signifies when a new block (which has its own locals) starts,
  lst = index of the last object that was added
  If the 4th element after the 'i' matches "thisisanobject", then assume that we are processing an
object*)
 let rec exec sp pc tc ln lst= match prog.text.(pc) with
        LLocal i -> if locals.(ln+i+4) = sentinel then
                        ( for k = 0 to 4 do
                        temps.(tc + k) <- locals.(ln+i + k)
                        done ; exec sp (pc+1) (tc+5) ln lst)
                    else ( temps.(tc) <- locals.(ln+i); exec sp (pc+1) (tc+1) ln lst)
  |     LGlobal i -> if globals.(i+4) = sentinel then
                        ( for k = 0 to 4 do
                        temps.(tc + k) <- globals.(i + k)
                        done ; exec sp (pc+1) (tc+5) ln lst)
                    else ( temps.(tc) <- globals.(i);exec sp (pc+1) (tc+1) ln lst)
  |     Bin op ->
          temps.(0) <- (  match op with
                Add -> "\"" ^ string_of_int ((int_of_string(Str.global_replace (Str.regexp "\"") ""
temps.(tc - 2))) + (int_of_string(Str.global_replace (Str.regexp "\"") "" temps.(tc - 1)))) ^ "\""
            |   Compare -> string_of_int (compare (int_of_string(Str.global_replace (Str.regexp "\"") ""
temps.(tc - 2))) (int_of_string(Str.global_replace (Str.regexp "\"") "" temps.(tc - 1))))
            |   FetchValue -> "\"" ^ temps.(tc - 1) ^ "\"" (* The last value in temp array is required val *)
        ); temps.(4) <- "0"; exec sp (pc+1) 1 ln lst (*The temps array has been reset with this call *)
  |     Jsr i -> stack.(sp) <- (pc + 1);stack.(sp+1) <- ln; (* Push the program counter on stack. When
subroutine completes, execution should start from here. *)
                exec (sp+2) i tc ln lst
```

```
  |       Gnf i -> temps.(tc - 5) <- temps.(tc - 5 + i); (* The temps array contains the response object.
Fetch the property at offset and store it at the place where the response object is stored thus reusing
the temps array *)
                temps.(tc -1) <- "0"; exec sp (pc + 1) (tc-4) ln lst
  |       PLocal i -> print_endline (locals.(ln+i)); exec sp (pc+1) 0 ln lst
  |       PGlobal i -> print_endline (globals.(i)); exec sp (pc+1) 0 ln lst
  |       PTemp -> print_endline(temps.(tc-1)); exec sp (pc+1) 0 ln lst
  |       SLocal i -> (* Need to check if we are storing object *)
                if temps.(tc-1) = sentinel then
                        ( for k = 5 downto 1 do
                        locals.(ln+i+5-k) <- temps.(tc-k)
                        done; exec sp (pc+1) 0 ln (currentln i lst))
                else (locals.(ln+i) <- temps.(tc-1); exec sp (pc+1) 0 ln (currentln i lst))
  |       SGlobal i -> (* Need to check if we are storing object *)
                if temps.(tc-1) = sentinel then
                        ( for k=5 downto 1 do
                        globals.(i+5-k) <- temps.(tc-k)
                        done; exec sp (pc+1) 0 ln lst)
                else ( globals.(i) <- temps.(tc-1); exec sp (pc+1) 0 ln lst)
  |       STemp s -> temps.(tc) <- s; exec sp (pc+1) (tc+1) ln lst
  |       Hlt     -> ()
  |       Ent i -> (* Push all temporary values to local stack and reset temporary. The temporary values
were sent as arguments. *)
        if tc > 0 then
        ( for k=(tc-1) downto 0 do
                (* lst -1 + i + tc - 1 - k *)
          locals.(lst-1+i+tc-1-k) <- temps.(k)
          done; exec sp (pc+1) 0 lst (lst+tc)
        )
        else exec sp (pc+1) 0 lst lst
  |       Skip i -> if tc > 0 then
                  begin
                        if temps.(tc-1) = "0" then
                                exec sp (pc+1) tc ln lst
                        else if temps.(tc-1) = "1" then
                                exec sp (pc+1) tc ln lst
                        else
                                exec sp (pc+i+1) tc ln lst
                  end
                  else exec sp (pc+i+1) tc ln lst
  |       Rts i -> let new_pc=stack.(sp-2) and new_ln=stack.(sp-1) in
                  temps.(4) <- "0"; exec (sp-2) new_pc tc new_ln 0
  |       Loop  -> stack.(sp) <- (pc + 1);stack.(sp+1) <- ln;
                exec (sp+2) (pc+1) tc ln lst (* Similar to Jsr *)
  |       GoBackLoop -> let new_pc=stack.(sp-2) and new_ln=stack.(sp-1) in
                        exec sp new_pc tc new_ln 0 (* Similar to Rts *)
  |       EndLoop -> exec (sp-2) (pc+1) tc ln lst (* Perform cleanup *)
  |       LSkip i -> stack.(sp) <- (pc + 1); stack.(sp+1) <- ln;
```

```
                    exec (sp+2) (pc+i+1) tc ln lst (* Similar to Jsr *)
  |       EndForLoop -> let new_pc=stack.(sp-2) and new_ln=stack.(sp-1) in
                          exec (sp-2) new_pc tc new_ln 0 (* Similar to Rts *)
  |       SendReq ->

          let test =
          let body =
          Client.get (Uri.of_string ("http://data.cityofnewyork.us/resource/ebb7-
mvp5.json?month=2014%20/%2001&borough="^(Str.global_replace (Str.regexp_string "\"") ""
(temps.(tc-2)))^"&communitydistrict="^(Str.global_replace (Str.regexp_string "\"") "" (temps.(tc-1)))))
>>= fun (resp, body) ->
  Cohttp_lwt_body.to_string body
          in Lwt_main.run body
          in let splitBody = Str.split (Str.regexp "[,]+") (Str.global_replace (Str.regexp "[^0-9,./]") "" test) in
          temps.(tc) <- string_of_int (truncate (float_of_string(List.nth splitBody 0)));
          temps.(tc+1) <- string_of_int (truncate(float_of_string(List.nth splitBody 1)));
          temps.(tc+2) <- (List.nth splitBody 2);
          temps.(tc+3) <- (List.nth splitBody 4);
          temps.(tc+4) <- sentinel;
          exec sp (pc+1) (tc+5) ln lst;

  in exec 0 0 0 0 0
```

Parser.mly

```
%{ open Ast %}

%token SEMI FUNCTION IF COMPARE RETURN ELSE THEN ENDFUNCTION VARIABLE
%token SENDREQUEST FETCHVAL PRINTVALUE LBRACE RBRACE LPAREN RPAREN
%token ASSIGN EOF COMMA ADD
%token REPEAT UNTIL FOR IN
%token <string> ID
%token <string> TEXTLINE

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN

%start program
%type <Ast.program> program

%%

program:
  /* nothing */ { [], [] }
 | program vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   FUNCTION ID formals_opt LBRACE vdecl_list stmt_list RBRACE ENDFUNCTION
    { { fname = $2;
            formals = $3;
            locals = List.rev $5;
            body = List.rev $6 } }

formals_opt:
   /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    ID                { [$1] }
  | formal_list ID { $2 :: $1 }

vdecl_list:
   /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
  VARIABLE ID SEMI { $2 }
```

```
stmt_list:
   /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
   expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF expr THEN stmt %prec NOELSE { If($2, $4, Block([])) }
  | IF expr THEN stmt ELSE stmt    { If($2, $4, $6) }
  | REPEAT stmt UNTIL expr SEMI              { Repeat($2,$4) }
  | FOR ID IN LPAREN actuals_list RPAREN stmt { Forin($2,$5,$7) }

textexpr :
   TEXTLINE { Text($1) }

expr:
   ID                { Id($1) }
|   ID ASSIGN expr { Assign($1,$3) }
|   ID LPAREN actuals_opt RPAREN { Call($1,$3) }
|   COMPARE LPAREN expr expr RPAREN { CompareVal($3,$4) }
|   ADD LPAREN expr expr RPAREN { AddVal($3,$4) }
|   FETCHVAL LPAREN expr ID RPAREN { FetchVal($3,$4) }
|   FETCHVAL LPAREN expr TEXTLINE RPAREN { FetchVal($3,$4) }
|   PRINTVALUE LPAREN expr RPAREN { PrintVal(Print,$3) }
|   PRINTVALUE LPAREN textexpr RPAREN { PrintVal(Print,$3) }
|   SENDREQUEST LPAREN actuals_list RPAREN { SendWebRequest(SendRequest,$3) }

actuals_opt:
   /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
   expr            { [$1] }
  | textexpr                { [$1] }
  | actuals_list expr { $2 :: $1 }
```

Scanner.ml

```
{ open Parser }

rule token = parse
  [' ' '\t' '\n' '\r'] {token lexbuf}
| "/*" ['a'-'z' 'A'-'Z' '0'-'9' ' ' '-' '.']* "*/"      { token lexbuf } (* Comments *)
| ';'           { SEMI }
| "beginfunction" { FUNCTION }
| "if"                  { IF }
| "compare"      { COMPARE }
| "return"       { RETURN }
| "else"   { ELSE }
| "then"              { THEN }
| "endfunction"   { ENDFUNCTION }
| "var"           { VARIABLE }
| "sendrequest"               { SENDREQUEST }
| "fetchvalue"     { FETCHVAL }
| "printtoscreen" { PRINTVALUE }
| '{'                   { LBRACE }
| '}'                   { RBRACE }
| '('                   { LPAREN }
| ')'                   { RPAREN }
| '='                   { ASSIGN }
| ','                   { COMMA }
| "repeat"          { REPEAT }
| "until"             { UNTIL }
| "for"               { FOR }
| "in"                { IN }
| "add"               { ADD }
| ['a'-'z' 'A'-'Z' ']+ ['a'-'z' 'A'-'Z' '0'-'9']* as lxm { ID(lxm) }
| '"' ['0'-'9']+ '"' as lxm { ID(lxm) }
| '"' ['a'-'z' 'A'-'Z' '0'-'9' ' ' '-' '.' '&']* '"' as lxm { TEXTLINE(lxm) }
| eof                   { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
```

Project.ml

```
type action = Bytecode | Execute

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-b", Bytecode);
                              ("-e", Execute) ]
  else Execute (* action = Execute at this point*)
  in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  match action with
   Bytecode -> let listing =
     Bytecode.string_of_prog (Compile.translate program)
    in print_endline listing
  | Execute -> Execute.execute_prog (Compile.translate program)
```

Testall.sh

```
#!/bin/sh

cd _build
NYCGCS="ocamlrun project.byte"

# Set time limit for all operations
ulimit -t 30

globallog=../tests/testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

SignalError() {
    if [ $error -eq 0 ] ; then
            echo "FAILED"
            error=1
    fi
    echo "  $1"
}
```

```
# Run <args>
# Report the command, run it, and report any errors
Run() {
   echo $* 1>&2
   eval $* || {
          SignalError "$1 failed on $*"
          return 1
   }
}

Check() {
   error=0
   basename=`echo $1 | sed 's/.*\\///
                        s/.gs//'`
   reffile=`echo $1 | sed 's/.gs$//'`
   basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

   echo -n "$basename...\n"

   echo 1>&2
   echo "###### Testing $basename" 1>&2

   generatedfiles=""

   generatedfiles="$generatedfiles ${basename}.b.out" &&
   Run "$NYCGCS" "-b" "<" $1 ">" ../tests/${basename}.b.out

   generatedfiles="$generatedfiles ${basename}.e.out" &&
   Run "$NYCGCS" "-e" "<" $1 ">" ../tests/${basename}.e.out

   # If the expected output is different from actual output, error

   notmatched=`diff -q ../tests/${basename}.expect.out ../tests/${basename}.e.out | wc -c`
   if [ "$notmatched" -gt "0" ]; then
          echo "Following file failed: ${basename}"
   fi

}

files="../tests/*.gs"

for file in $files
do
   Check $file 2>> $globallog
done

exit $globalerror
```

Makefile

```
project :
	ocamlbuild -clean
	ocamlbuild -use-ocamlfind -pkgs cohttp.lwt project.byte
	sh testall.sh
```