YAGL: Yet Another Graphics Language

Final Report

Edgar Aroutiounian, Jeff Barg, Robert Cohen

August 15, 2014

Abstract

YAGL is a programming language for generating graphics in SVG format from JSON formatted data.

1 The Proposal

1.1 Description of YAGL

YAGL is a new programming language for constructing graphics, with the intent of allowing programmers to construct graphics with various different output formats.

The language features a static type system with a void type that can conditionally downcast to the other primitives (Int, Array, etc..). Types are indicated by the type name declarator followed by the variable name. All types are objects. Each object holds an enumeration for what its type is, along with the underlying data. There is no string type. There are integer and array literals.

There are limited control structures: an iterated for loop, an if statement and break. A for loop can iterate on Stream and Array types and will yield a void type as the iterated item which needs to be downcast.

YAGL formatted JSON data must contain the following keys:

1.2 What Does YAGL Solve?

YAGL allows users to take JSON formatted data and produce clear and concise graphs. The properties of these graphs (such as color) can be manipulated utilizing algorithms. The output of the YAGL program can be determined by the user (SVG, ASCII, or PDF). YAGL is essentially and clean and minimal language for producing graphs quickly and easily.

1.3 Examples of YAGL Syntax

YAGL will have a void type, which cannot be operated upon. The void type can be conditionally downcast using the syntax:

```
if let b = a as Int {
      # scope of b is only in this function
}
```

This allows us to safely operate on JSON streams.

YAGL does not have a string type. In order to pipe in filenames to open JSON files, the open open keyword takes in $[n] which specifies the *nth* command line parameter. *open* constructs a Stream.

```
Stream s = open $0
```

YAGL has an iterated for loop and array literals:


```
Stream s = open $0
```

YAGL has an iterated for loop and array literals:


```
for a in [[4,1,5], 3, 5, [2, 3]] {
      if (let int_a = a as Int) {
      }
      elif (let int_a = a as Array) { }
}
```

Table 1 YAGL PRIMITIVES

KEYWORDS	OPERATORS	TYPES
if	++	Void
elif	+	Array
for	-	Canvas
in	*	Int
break	/	Stream
let		
in		
func		

2 Language Tutorial

2.1 Example YAGL Program

```
func main()
{
    canvas(500, 500);
    open("demo1.json");

      iterate {
            addRect(this["width"], this["height"], this["x"], this["y"], "red", "blue");
      }
}
```

2.2 More Example Code

Below is another example program written in the YAGL language.

```
func main() {
 Int a = numberOfRings();
 Int b = a;
 String c = "red";
```

```
 Int d = 0;

 canvas(500, 500);

 while (a) {
   if ((a/2 + a/2) == a) {
     c = "red";
   } else {
     c = "blue";
   }

   d = (250 * a) / b;

   addCircle(d, 250, 250, c, c);

   a = a - 1;
 }

 text("A Target!", 40, 40, 300);
}

func numberOfRings() {
 return 50;
}
```

3 Language Reference Manual

3.1 Introduction

This manual describes the YAGL programming language as specified by the YAGL team.

3.2 Lexical Conventions

A YAGL program is written in the 7-bit ASCII character set and is divided into a number of logical lines. A logical line terminates by the token SEMI, where a logical line is constructed from one or more physical lines. A physical line is a sequence of ASCII characters that are terminated by a semi-colon character.

3.2.1 Tokens

A logical line may consist of the following tokens assuming correct syntax is used:
SEMI, LPAREN, RPAREN, LBRACE, RBRACE, COMMA, COLON, IN, LBRACK, RBRACK, PLUS, MINUS, TIMES, DIVIDE, ASSIGN, FUNC, EQ, NEQ, LT, LEQ, GT, GEQ, RETURN, IF, ELSE, FOR, WHILE.

3.2.2 Comments

Comments are introduced by '/*' and last until they encounter the next '*/'.

3.2.3 Identifiers

An identifier is a sequence of ASCII alphanumeric characters and the underscore character where upper and lower case are distinct; ASCII digits may not be included in an identifier. Identifiers must start with a lowercase letter.

3.2.4 Keywords

The following is an enumeration of all the keywords required by a YAGL implementation: 'if', 'else', 'in', 'func', 'return', 'iterate', 'while', 'this'. Identifiers cannot have the same name as keywords.

3.2.5 Built-in Functions

The following functions are built-in:
print(String: x), print_int(Int: x), addRect(Int: width, Int: height, Int: x, Int: y, String: color, String: border_color), addCircle(Int: r, Int: cx, Int: cy, String: color, String: border_color), text(String: title, Int: x, Int: y, Int: size), canvas(Int: width, Int: height)

3.2.6 String Literals

YAGL string literals begin with a double-quote (") followed by a finite sequence of non-double-quote ASCII characters and close with a double-quote ("). YAGL strings do not recognize escape sequences. An example of a YAGL string literal is String bar = "Hello World".

3.2.7 Integer Literals

YAGL supports integer literals. An integer literal is a sequence of digits that does not lead with 0. Integer literals are base 10 and are the only type of numeric literal recognized by the language, Integer literals can only be positive. Additionally, integer literals will use 64 bits.

3.2.8 Operators

The following tokens are operators: +, -, *, /, >, <, <=, >=, ==, =

3.2.9 Basic Types

YAGL features four built-in data types, String, Array, Integer, and Dictionary. String represents string objects, Arrays represent an ordered sequence of Integers or Dictionaries. Arrays must have all elements of the same type. Dictionaries represent an implementation of a key-value storage system and are mainly used as a YAGL container for JSON data. Dictionaries may have only Strings as keys and their values are integers. Arrays and Dictionaries are iterable and hence may be used in the declaration of a for loop.

3.3 Data Model

As YAGLs primarily purpose is a language to programmatically create SVGs (scalable vector graphics), therefore it makes sense to have just one global SVG Object. In addition, YAGL supports one global reference to a Dictionary of JSON data. The reference does not need an explicit handle, a simple call to open is enough to establish this global variable. A YAGL environment also provides a handle to a "this" object. The "this" object refers to a previously opened Dictionary via an open call.

3.4 JSON Protocol

The YAGL spec recommends that JSON data passed to the builtin function **open** be of the following format:

```
[{"height":<Integer Literal>, "width":<Integer Literal>,
 "x":<Integer Literal>, "y":<Integer Literal>}...]
```

3.5 Expressions

The precedence of expression operators will first prioritize function calls, then parentheses, then multiplication/division/modulo, then addition and subtraction.

3.6 Array References

An array identifier followed by a set of square brackets enclosing an integer value to denote the index denotes array indexing, i.e. myArray[4]. Array indexing returns an integer value. Indexing out of bounds in an array causes a compile time error. Arrays start at index 0, and a legal index is any index from 0 to the number of elements in the array minus 1.

3.7 Dictionary References

A Dictionary identifier followed by a set of square brackets enclosing a string literal performs a lookup. Performing a lookup on a Dictionary where the key does not exist returns -1, else it returns the integer value associated with the key.

3.8 Function Calls

A function call is a postfix expression which is performed by the identifier of the function followed by a possibly empty set of parentheses. Functions may return an explicit value to their caller if they have a return expression defined in their body, else they return 0. return is a statement that takes an expression, evaluates it, then returns it as the value of the expression where the function is called.

3.9 Operators

3.9.1 Multiplicative Operators

The multiplicator operators *, /, follow the usual rules of mathematics and group left to right, where denotes multiplication, / denotes division. Division by 0 is undefined. Division will also truncate decimal parts of numbers to the nearest integer lower than the mathematical division.

3.9.2 Additive Operators

The additive operators +, - group left to right where + denotes addition and - denotes subtraction

3.9.3 Relational Operators

The relational operators group left to right and return back 1 if the operator evaluates to true, 0 if the operator evaluates to false.

3.9.4 Equality Operators

The equality operator == is only valid for either Integer or String types and returns 1 if the operands are equal, 0 otherwise.

3.9.5 Assignment Expressions

There is only one assignment operator, =. The equals operator accepts a type declaration along with a NAME token for its left operand and an expression for its right operand.

3.9.6 Comma Operator

A parameter list is a comma-separated series of identifiers with type names, for example **(Int: a, Int: b, Int: c)**. The parameter list maybe empty in which case it is an empty pair of parenthesis.

3.10 Scope

3.10.1 Lexical Scope

Identifiers are placed into non-intersecting namespaces. The two namespaces are functions and file level. The lexical scope of an object or function identifier that appears in a block begins at the end of its declarator and persists to the end of the block in which it appears. The scope of a parameter of a function begins at the start of the function block and extends to the end. If an identifier is reused at the head of a block or as a function parameter, any other declaration of the identifier is shadowed until the end of the block or function.

3.10.2 Iteration Statements

While loops expects an expression and a statement body. The while loop repeats the expression body until the expression yields zero.

The *iterate* keyword introduces a block which is executed for the length of the JSON data under the assumption that *open* was previously called.

3.10.3 If Statements

In an *if* statement, the expression is evaluated, including side-effects, and if the result is not 0, the first sub-statement is executed. If it is equal to 0, the else sub-statement is executed provided that it exists.

3.11 Grammar

This is the comprehensive Grammar for the YAGL programming language. The grammar has terminal symbols NAME, INTEGER, STRING, OPERATORs, which includes "+", "-", "/", "*", and PUNCTUATION which includes "(", ")", "", and "". Note that * after a regular expression denotes zero or more instances of the expression, + denotes one or more instances and ? denotes one or zero.

```
type_spec: Array | Dict | Int | String
return_stmt: return [expr]
func_definition: 'func' NAME parameters { suite }
suite: simple_stmt | compound_stmt
simple_stmt: (expr_stmt | print_stmt) NEWLINE
```

```
expr_stmt: asn_stmt, ; | expr
asn_stmt: type_specifier, NAME = expr
exefinition | ( if_stmt | while_stmt | for_stmt | simple_stmt )+
if_stmt:  if    (   bool_expr )   suite ( elif  suite)* ( else  suite)?
while_stmt:  while    (   bool_expr )   suite
iterate_stmt: 'iterate' '(' NAME ')' suite
bool_expr: 1 | 0 | logic_and | logic_or
comp_op:  <   |   >   |   ==   |   >=   |   <=
parameters: (   [args_list] )
args_list: (type_spec NAME)*
```

4 Project Plan

4.1 Plan for YAGL Developing

Table 2 YAGL Development Calendar

Week 1	Week 2	Week 3	Week 4	Week 5	Week 6
Proposal, Plan	Parser	Semantic Analysis	Bytecode	Compiler	Test Suite and Final Project

5 Architectural Design

The YAGL internal stack begins with the a lexical tokenizer. The tokenizer converts a stream of chars into a list of tokens in accordance with YAGL's grammar. The resultant list is then passed to a parser which creates an abstract syntax tree. This AST is then passed to a semantic analyzer which stops the compilation process upon encountering a semantic error, i.e. adding a String to an Integer. If the semantic analysis succeeds, the AST passes to the code generator. At this stage, code generation starts and outputs a single C++ source code file.

6 Test Plan

The test suite is executed via a python script called run_tests.py. Instructions are in the README.md of the YAGL project.

7 Lessons Learned

The following lessons were learned during the course of this project:

7.1 Type System

First, we learned that although we designed the language to have a fairly strong type system, at compile time most of the type information is thrown away. The semantic analysis is where the type information comes into play and where the compiler has the chance to usefully determine whether types are being misused. The semantic analysis itself is a non-trivial task, therefore we restricted our semantic analysis to only integer expressions.

7.2 Byte-Code

We also learned that using low-level bytecode and converting back to C++ is actually fairly efficient, because GCC will actually optimize the C code after it's been generated using our code-gen. Additionally, using a slightly higher-level language C++ allowed us to use better libraries for the tasks we had while stile maintaining the amount of optimization possible.

7.3 Stack

We learned the utility of using a stack on the lower-level to manage memory and the allocation of variables that we had. This allowed us to very easily have several different types and execute logic wherever appropriate by using the stack. Additionally, this stack allowed us to create our iterate keyword without too much difficulty.

8 Complete Listing of Code

```
(* Scanner.mll *)

{ open Parser }

rule token = parse
| [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"        { comment lexbuf }
| ';'         { SEMI}
| '('         { LPAREN }
| ')'         { RPAREN }
| '['         { LBRACK }
| ']'         { RBRACK }
| '{'         { LBRACE }
| '}'         { RBRACE }
| ','       { COMMA }
| ':'       { COLON }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '='       { ASSIGN }
| "in"      { IN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "if"      { IF }
| "else"    { ELSE }
| "iterate" { FOR }
| "while"   { WHILE }
| "return" { RETURN }
| "Int"     { INT }
| "Dict"    { DICT }
| "func"    { FUNC }
| "Array"   { ARRAY }
| "String" { STRING }
| ['0'-'9']+ as lxm { INTLITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '"' [^'"']* '"' as l { STRINGLITERAL(l) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }


(* Parser.mly *)

%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA COLON IN LBRACK RBRACK
%token PLUS MINUS TIMES DIVIDE ASSIGN FUNC
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE
%token <int> INTLITERAL
%token INT DICT ARRAY STRING
%token <string> ID STRINGLITERAL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

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

fdecl:
   FUNC ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { fname = $2;
   formals = $4;
   locals = List.rev $7;
   body = List.rev $8 } }

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

formal_list:
    INT COLON ID                   { [(Int, $3)] }
  | STRING COLON ID                { [(String, $3)]}
  | ARRAY COLON ID                 { [(Array, $3)]}
  | DICT COLON ID                  { [(Dict, $3)]}
  | formal_list COMMA qual COLON ID { ($3, $5) :: $1 }

vdecl_list:
  |              { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:/* Need to come back to this, looks suspicious */
/* YAGL only knows string literals and int literals */
  | INT ID ASSIGN expr SEMI { {id=$2;
```

```
105             v_type=Int;
106             rhs=$4}}
107   | INT ID SEMI { {id=$2;
108             v_type=Int;
109             rhs=LiteralInt(0)}}
110   | STRING ID ASSIGN expr SEMI { {id=$2;
111           v_type=String;
112                                  rhs=$4}}
113   | STRING ID SEMI { {id=$2;
114           v_type=String;
115                                  rhs=LiteralString("")}}
116
117 stmt_list:
118     /* nothing */  { [] }
119   | stmt_list stmt { $2 :: $1 }
120
121 stmt:
122     expr SEMI { Expr($1) }
123   | RETURN expr SEMI { Return($2) }
124   | LBRACE stmt_list RBRACE { Block(List.rev $2) }
125   | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
126   | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
127   | FOR stmt { For($2) }
128   | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
129   | ID ASSIGN expr SEMI { Variable($1, $3) }
130
131 qual:
132   | ARRAY { Array }
133   | INT { Int }
134   | DICT { Dict }
135   | STRING { String}
136
137 expr:
138     INTLITERAL       { LiteralInt($1) }
139   | STRINGLITERAL    { LiteralString($1) }
140   | ID               { Id($1) }
141   | expr PLUS   expr { Binop($1, Add,   $3) }
142   | expr MINUS  expr { Binop($1, Sub,   $3) }
143   | expr TIMES  expr { Binop($1, Mult,  $3) }
144   | expr DIVIDE expr { Binop($1, Div,   $3) }
145   | expr EQ     expr { Binop($1, Equal, $3) }
146   | expr NEQ    expr { Binop($1, Neq,   $3) }
147   | expr LT     expr { Binop($1, Less,  $3) }
148   | expr LEQ    expr { Binop($1, Leq,   $3) }
149   | expr GT     expr { Binop($1, Greater,  $3) }
150   | expr GEQ    expr { Binop($1, Geq,   $3) }
151   | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
152   | expr LBRACK expr RBRACK { ArrayIndex($1, $3) }
153   | LPAREN expr RPAREN { $2 }
154
155 actuals_opt:
156     /* nothing */ { [] }
157   | actuals_list  { List.rev $1 }
158
159 actuals_list:
160     expr                    { [$1] }
161   | actuals_list COMMA expr { $3 :: $1 }
162
163 (* ast.ml *)
164
165  type op = Add | Sub | Mult | Div
166      | Equal | Neq | Less | Leq
167      | Greater | Geq
168
169  and expr = LiteralInt of int
170     | LiteralString of string
171     | Id of string
172     | Binop of expr * op * expr
173     | Call of string * expr list
174     | ArrayIndex of expr * expr
175     | Noexpr
176
177  and stmt = Block of stmt list
178     | Expr of expr
179     | Return of expr
180     | If of expr * stmt * stmt
181     | For of stmt
182     | While of expr * stmt
183     | Variable of string * expr
184
185  and qual = Dict | Array | Int | String
186
187  and variable = {id:string; v_type:qual;rhs:expr}
188
189  and func_decl = {fname : string;
190       formals : (qual * string) list;
191       locals : variable list;
192       body : stmt list}
193
194 type program = variable list * func_decl list
195
196 let string_of_qual = function
197   | Dict -> "Dict"
198   | Array -> "Array"
199   | String -> "String"
200   | Int -> "Int"
201
202 (* semantic.ml *)
203
204 open Ast
205 (* Could be made more informative? *)
206 exception SemanticError of string
207
208 type typ = TInt | TString
209 (* Not an exhaustive match *)
```

```
let rec infer_typ = function
  | LiteralInt _ -> TInt
  | LiteralString _ -> TString
  | Binop (e1, op, e2) ->
     let (t1, t2, ret_typ) = infer_op_typ op in
     if check_expr e1 t1 && check_expr e2 t2
     then ret_typ (* Make this more informative *)
     else raise (SemanticError "Type problem with: ")
  | _ -> TInt

and infer_op_typ = function
  | Add | Mult | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq -> (TInt, TInt, TInt)

and check_expr e typ =
  let inf_typ = infer_typ e in
  typ = inf_typ

let infer_statement = function
  (* Obviously more, but just focusing on low fruit *)
  | Expr(e) -> infer_typ e
  (* Just a stop gap *)
  | _ -> TInt

let infer_func_body = function
  | {fname=_;formals=_;locals=_; body=a} -> List.fold_left  (* Only doing this for side effect *)
                (fun accum s -> infer_statement s :: accum)
                []
                a
let var_check q rhs = match (q, rhs) with
  | (Int, TInt) -> true

let infer_vars = function
  | {id=_; v_type=Int; rhs=e} -> var_check Int (infer_typ e)

let verify (globals, funcs) =
(* Basically doing these maps just for the possibility of a side effect going off,
   aka the exception, but if no exception, then we just give back what we are given *)
  let g_result = List.map infer_vars globals in
  let f_result = List.map infer_func_body funcs in
  (globals, funcs)


(* bytecode.ml *)

open Ast

type bstmt =
LitInt of int (* Push a literal *)
  | LitStr of string (* Push a literal *)
  | Drp (* Discard a value *)
  | Bin of Ast.op (* Perform arithmetic on top of stack *)
  | Lod of int
  | Str of int
  | Lfp of int
  | Sfp of int
  | Jsr of int
  | Ent of int
  | Rts of int
  | Beq of int
  | Bne of int
  | Bra of int
  | Index
  | NextItr
  | Hlt

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

(* 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 vars *)
  local_index : int StringMap.t; (* FP offset for args, locals *)
}

(* enum : int -> 'a list -> (int * 'a) list *)
let rec enum stride n = function
  [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* string_map_pairs:StringMap 'a -> (int * 'a) list -> StringMap 'a *)
let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs


(** Translate a program in AST form into a bytecode program.
Throw an exception if something is wrong, e.g., a reference to an unknown variable or function *)
let translate (globals, functions) =
  (* Allocate "addresses" for each global variable *)

  let global_indexes = string_map_pairs StringMap.empty (enum 1 0 (List.map (fun l -> l.id) globals)) in

  (* Assign indexes to function names; built-in functions are special *)
  let built_in_functions =
    StringMap.add "print"     (-1) StringMap.empty in
  let built_in_functions =
```

```
    StringMap.add "canvas" (-2) built_in_functions in
  let built_in_functions =
    StringMap.add "text"   (-3) built_in_functions in
  let built_in_functions =
    StringMap.add "addCircle"     (-4) built_in_functions in
  let built_in_functions =
    StringMap.add "addRect"     (-5) built_in_functions in
  let built_in_functions =
    StringMap.add "open"     (-6) built_in_functions in

  let function_indexes = string_map_pairs built_in_functions
    (enum 1 1 (List.map (fun f -> f.fname) functions)) in

  (* Translate an AST function to 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 local_offsets = enum 1 1 (List.map (fun l -> l.id) fdecl.locals)
  and formal_offsets = enum (-1) (-2) (List.map (fun (q, s) -> s) fdecl.formals) in

  let env = {
    env with
      local_index =
        string_map_pairs
        StringMap.empty (local_offsets @ formal_offsets)
    } in

(* Translate an expression *)
let rec expr = function
  LiteralInt i -> [LitInt i]
  | LiteralString s -> [LitStr s]
  | Id s ->
    (try [Lfp (StringMap.find s env.local_index)]
  with Not_found -> try
    [Lod (StringMap.find s env.global_index)]
  with Not_found ->
    if (s = "this") then [] else
    raise (Failure ("undeclared variable " ^ s)))
  | Binop (e1, op, e2) -> expr e1 @ expr e2 @ [Bin op]
  | 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)))
  | ArrayIndex (v, i) -> expr i @ [Index]
  | Noexpr -> []

(* Translate a statement *)
in let rec stmt = function
  Block sl -> List.concat (List.map stmt sl)
  | Expr e -> expr e @ [Drp] (* Discard result *)
  | Return e -> expr e @ [Rts num_formals]
  | If (p, t, f) -> let t' = stmt t and f' = stmt f in
    expr p @ [Beq(2 + List.length t')] @
    t' @ [Bra(1 + List.length f')] @ f'
  | For (b) ->
    let b' = stmt b and e' = [NextItr] in
    [Bra (1+ List.length b')] @ b' @ e' @
    [Bne (-(List.length b' + List.length e'))]

  | While (e, b) ->
    let b' = stmt b and e' = expr e in
    [Bra (1+ List.length b')] @ b' @ e' @
    [Bne (-(List.length b' + List.length e'))]
  | Variable (id, e) -> expr e @
    (try [Sfp (StringMap.find id env.local_index)]
    with Not_found -> try
    [Str (StringMap.find id env.global_index)]
    with Not_found ->
    raise (Failure ("undeclared variable " ^ id)))

(* Translate a whole function *)
in [Ent num_locals] @ (* Entry: allocate space for locals *)
  (List.fold_left (fun ls v -> ls @ expr v.rhs @
    (try [Sfp (StringMap.find v.id env.local_index)]
    with Not_found -> try
    [Str (StringMap.find v.id env.global_index)]
    with Not_found ->
    raise (Failure ("undeclared variable " ^ v.id)))) [] fdecl.locals) @
  stmt (Block fdecl.body) @ (* Body *)
  [LitInt 0; Rts num_formals] (* Default = return 0 *)

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

StringMap.iter (fun x y -> print_endline x) function_indexes;

(* 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
let retval = { 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)
    )
}
in Array.iter (function Jsr i -> print_int i | _ -> ()) retval.text; retval


(* codeGen.ml *)

(*
let file_name_helper given_string = (Filename.chop_extension given_string) ^ ".c"
(* Just a test example *)
let generate_code byte_code = let gen_file = open_out (file_name_helper Sys.argv.(1)) in
            output_string gen_file "#include<stdio.h>\nint main(int argc, char **argv)\n\
                  {\nprintf(\"hello world!\\n\");\nreturn 0;\n}";
            flush gen_file

 *)
open Ast
open Bytecode
open Printf

let file = "example.cpp"

let generate prog =
  let oc = open_out file in
  fprintf oc
  "
  #include <fstream>
  #include <string>
  #include <stdlib.h>
  #include <json/json.h>

  #import <stdio.h>
  #import <iostream>
  #include <cstdint>

  typedef long int64;

  using namespace std;

string global_svg;
string style(string c, string s){return \"style=\\\"fill:\" + c + \";stroke:\" + s + \"\\\"\";}
void addRect(int width, int height, int x, int y, string color, string border_color)
{global_svg.append(\"<rect x=\\\"\" + to_string(x) + \"\\\" y=\\\"\" + to_string(y) + \"\\\" width=\\\"\" + to_string(width) + \"\\\" height=\\\"\" + to_s
void addCircle(int r, int cx, int cy, string color, string border_color)
{global_svg.append(\"<circle cx=\\\"\" + to_string(cx) + \"\\\" cy=\\\"\" + to_string(cy) + \"\\\" \" + \"r=\\\"\" + to_string(r) + \"\\\" \" + style(colo
void text(string title, int x, int y, int size)
{global_svg.append(\"<text x=\\\"\" + to_string(x) + \"\\\" y=\\\"\" + to_string(y) + \"\\\" font-family=\\\"Verdana\\\">\" + title + \"</text>\\n\");}
void canvas(int width, int height)
{global_svg.append(\"<?xml version=\\\"1.0\\\"?>\\n<svg width=\\\"\" + to_string(width) + \"\\\" height=\\\"\" + to_string(height) + \"\\\"  viewPort=\\\"
void _finished(){global_svg.append(\"\\n</svg>\");}

Json::Value openJson(string path)
{
  //Just creating node and reader on the stack.
  Json::Value root_node;
  Json::Reader reader;
  ifstream test(path);
    bool result = reader.parse(test, root_node, false);
  return root_node;
}

  #define DEBUG 0

  int main() {
    int64 num_globals = %i;
    void * globals[num_globals];
    void * stack[1024];

    Json::Value itr;
    Json::Value self;

    void * temp;

    string s1;
    string s2;
    string s3;

    char * temp_char_star;

    int64 i = 0;

    int itr_counter = 0;
    int64 size  = 0;

    int64 fp = 0;
    int64 sp = 0;
    int64 pc = 0;

    int64 op1;
    int64 op2;
    int64 result;

    int64 new_fp;
    int64 new_pc;
```

```
525
526     for (i = 0;i<1024;i++) {
527       stack[i] = 0;
528     }
529     for (i = 0; i < num_globals; i++) {
530       globals[i] = 0;
531     }
532     printf(\"\\ndddd\\n\");
533
534     ofstream svg_file;
535     svg_file.open(\"test.svg\");
536   " prog.num_globals;
537
538 let label_counter = ref (-1) in
539 let rec execute_prog fp sp pc =
540   Array.iter (fun x ->
541   incr label_counter;
542   fprintf oc " if(DEBUG)printf(\"PC: %%li \\n \", pc); goto gotopc;LABEL%i:\n" !label_counter;
543
544   match x with
545     LitInt i ->
546     fprintf oc
547       "stack[sp] = (void *)(%i);sp++;pc++;" i;
548   | LitStr s ->
549     fprintf oc
550       "stack[sp] = (void *)((char *)(%s));sp++;pc++;" s;
551   | Drp -> fprintf oc "sp--;pc++;";
552   | Bin op ->
553     fprintf oc
554       "op1 = (int64) stack[sp-2];\nop2= (int64) stack[sp-1];";
555     (match op with
556       Add -> fprintf oc "result=(op1+op2);"
557     | Sub -> fprintf oc "result=(op1-op2);"
558     | Mult -> fprintf oc "result=(op1*op2);"
559     | Div -> fprintf oc "result=(op1/op2);"
560     | Equal -> fprintf oc "result=(op1==op2);"
561     | Neq -> fprintf oc "result=(op1!=op2);"
562     | Less -> fprintf oc "result=(op1<op2);"
563     | Leq -> fprintf oc "result=(op1<=op2);"
564     | Greater -> fprintf oc "result=(op1>op2);"
565     | Geq -> fprintf oc "result=(op1>=op2);");
566     fprintf oc
567       "stack[sp-2]=(void *)result;sp--;pc++;"
568   | Lod i -> fprintf oc "stack[sp]=globals[%i];" i; fprintf oc "sp++;pc++;";
569   | Str i -> fprintf oc "globals[%i]=stack[sp-1];" i; fprintf oc "pc++;";
570   | Lfp i -> fprintf oc "stack[sp]=stack[fp+%i];" i ; fprintf oc "sp++;pc++;";
571   | Sfp i -> fprintf oc "stack[fp+%i]=stack[sp-1];" i; fprintf oc "pc++;";
572   | Jsr(-1) ->
573     fprintf oc "%s"
574         "printf(\"\\n%li\\n\", (int64) stack[sp-1]);pc++;";
575   | Jsr(-2) ->
576     fprintf oc "%s"
577         "canvas((int64) stack[sp-1], (int64) stack[sp-2]);pc++;";
578   | Jsr(-3) ->
579     fprintf oc "%s"
580         "
581         s1 = (char *) stack[sp-1];
582         text(s1, (int64) stack[sp-2], (int64) stack[sp-3], (int64)stack[sp-4]);pc++;";
583   | Jsr(-4) ->
584     fprintf oc "%s"
585         "
586         s1 = (char *) stack[sp-4];
587         s2 = (char *) stack[sp-5];
588         addCircle((int64) stack[sp-1], (int64) stack[sp-2], (int64) stack[sp-3], s1, s2);pc++;";
589   | Jsr(-5) ->
590     fprintf oc "%s"
591         "s1 = (char *) stack[sp-5]; s2 = (char *)stack[sp-6];addRect((int64) stack[sp-1], (int64) stack[sp-2],(int64) stack[sp-3], (int64) stack[sp-4],s1,
592   | Jsr(-6) ->
593     fprintf oc "%s"
594         "s1 = (char *) stack[sp-1]; itr = openJson(s1); itr_counter =0;printf(\"itr_counter: %i\\n\", itr_counter);pc++;";
595   | Jsr i -> fprintf oc "stack[sp]=(void *)(pc+1);sp++;pc=%i;" i;
596   | NextItr -> fprintf oc "size = itr.size(); if(itr_counter >= size) { /* put 0 on stack */ stack[sp] = (void *)(0);sp++; } else { self = itr.get((Json::
597   | Index ->  fprintf oc "s1 = (char *)stack[sp-1]; op1 = (self.get(s1, &self).asInt()); stack[sp-1] = (void *)op1;pc++;";
598   | Ent i -> fprintf oc "stack[sp]=(void *)fp;fp=sp;sp+=(%i+1);pc++;" i;
599   | Rts i -> fprintf oc "new_fp=(int64)stack[fp];new_pc=(int64)stack[fp-1];stack[(fp-1-%i)]=stack[sp-1];sp=fp-%i;fp=new_fp;pc=new_pc;" i i;
600   | Beq i -> print_endline ("\n\n\n" ^ string_of_int i); fprintf oc "pc+=((stack[sp-1] != 0)?%i:1);sp--;" i;
601   | Bne i -> print_endline ("\n\n\n" ^ string_of_int i); fprintf oc "pc+=((stack[sp-1])?%i:1);sp--;" i;
602   | Bra i -> fprintf oc "pc+=%i;" i;
603   | Hlt -> fprintf oc "goto END;";
604 ) prog.text
605 in
606 let retval = execute_prog 0 0 0 in
607 let rec switchstatements n s = (if n < 0 then s else switchstatements (n-1) (s ^ "case " ^ string_of_int n ^ ": goto LABEL" ^ string_of_int n ^ ";break;")
608 fprintf oc "gotopc:\nswitch(pc){%sdefault:printf(\"\\nERROR: %%li\",pc);break;}\nEND:
609 _finished();
610 svg_file << global_svg;
611 svg_file.close();
612  ; }" (switchstatements !label_counter "");
613 close_out oc;
614 retval
```