Scirch Final Report

By: Jeff Sinckler
Brian Hunter

Scirch Introduction and Motivation

Scirch is designed to be a utility for anybody who uses circuits on a regular basis. Users can write Scirch programs, which are synonymous to circuits, and execute them with selected inputs. Scirch lets users see the results of the circuits without having to physically build the entire circuit. The language is meant to be small, quick, compact, and easy so that users can spend as little time as possible writing up and testing their circuits. Scirch can be used to see the results of one simple circuit with a bunch of basic gates, or a series of circuits that are constantly being used inside of a larger circuit. The underlying concept behind Scirch is connectivity between circuits and repetition of already built components.

The motivation behind Scirch lies mainly in the fact that the two creators have taken courses in circuit logic. The most annoying part of these classes was having to loop the results of circuits back into the same circuit and re-think the result. There should be a way to easily recalculate the results of this circuit, as well as quickly calculate the outputs of larger circuits. Scirch was thought up and developed to allow saving of circuit routines so that the logic can easily be executed, repeated, tested, and analyzed. Also, there are times when we were trying to build a circuit that will return certain output in certain situations. Scirch's test functions allow for easy testing of complete inputs, to make sure that they actually perform as expected. (Not actually in the submitted iteration, but would be given more time) The basic functionalities of a programming language naturally couple with the building of components in circuits and then executing them. The functions in Scirch are synonymous with the smaller circuit components. When implemented as a function, these circuits are able to be used anytime and as many times as you want from within the main circuit. They are also able to modify and use any number of inputs.

Scirch Language Reference Manual

Elements of Scirch

Scirch contains tokens of the following "types":
keywords
identifiers
operators
punctuators

Scirch only has one keyword:

declare

Declare is used to initialize a variable that is not being given a value immediately. This can be done from inside a function if desired, and must be done outside of functions. Variables cannot be given values outside of functions.

Identifiers in Scirch are comprised of one or more characters.

identifier:
one or more of: a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 beginning with a character

Operators in Scirch define how to manipulate certain values.

operators:
^ o - a n

Comments in Scirch are placed in between </ and /> tokens.

Program Structure

Every source file in Scirch has an entry point. This entry point is enclosed in {} curly braces. From the entry point, users are able to perform basic logic and call other declared functions. Functions have their own individual sets of instructions that are run when the function is called. The main function gives execution to the function to run through its statements, then execution is passed back to the function in the same place.

{ *stmtone; stmttwo; function();* }

Scoping in Scirch manages the visibility of elements throughout the program. Particularly, variables can be declared and set from within any function. If they are declared in the main function, then they are accessible to other parts of the main function, as well as declared functions. Functions, however, are not able to mutate these values, only use them as part of their arithmetic. This is meant to mimic the process of circuits. Components are able to make use of values belonging to the main circuit, but not permanently modify them.

In addition, global variables can be declared. These variables can be accessed and set. They serve as inputs to the main function as well as inputs to the components. These variables act as the main thoroughput for the circuit.

Expressions and Assignments

Scirch only contains one L-value: identifiers. Variables (identifiers that refer to locations in memory) are the only things that can be set using the ‘=’ equals operator.
Every other expression is an R-value that is evaluated.

Operators in Scirch perform basic operations on the binary values (0 and 1):

- Binary
    - ‘^’ - and
    - ‘o’ - or
    - ‘a’ - nand
    - ‘n’ - nor
- Unary
    - ‘-’ - not

All operators in Scirch have the same precedence. A string of operators is evaluated left to right. If these operators are used on values other than 0 and 1, the expression will return a -1. This trickles down to all expressions, meaning if you have a string of expressions and one of them is bad, the entire string becomes -1.

Functions

Functions can be declared before and after the main function. The declaration must have an identifier name. When the main function calls declared functions, execution is transferred from the main function to the declared function. When the function finishes executing, execution will be returned to the main function. Functions must be declared in the following fashion:

sampleFunction{ *stmtone; stmttwo;* }

When a function finishes, the last feasable calculation that was done is returned. For example, if the last expression in a function was *testVar;* the return value of that function is the value of testVar. If the last expression was *declare testVar;* then the return value of the function is 0 representing the value of the testVar function.

Scirch Tutorial

Scirch is meant to be an easy language to pick up, learn, and use. These instructions should assist in figuring out how to write some meaningful Scirch code.

Values in Scirch

The basic elements of a Scirch program are the same as the basic elements of a circuit, the binary values 0 and 1. These numbers are passed into circuits and operations are called on these values. Using these values in Scirch is as easy as typing a 1 or a 0 into the source code.

Basic Operations

Now that we know how to use 1 and 0, we can talk about the basic operations in Scirch. There are five in total:

- ‘^’ - and
- ‘o’ - or
- ‘-’ - not

- ‘a’ - nand
- ‘n’ - nor

Each of these is a binary operator except for not, which is unary. Use these symbols in order to do basic logic:

- 0^0 -> 0
- 1^1 -> 1

NOTE: FOR OPERATORS o, a, and n, THE ARGUMENTS MUST HAVE SPACES BETWEEN THEM.

- 0o1 is WRONG
- 0 o 1 is RIGHT

Building a Program

Scirch source files can be called anything you like. In order to create one that Scirch can successfully read, you only need one thing, curly braces. The simplest Scirch program is as follows:

{}

This is a program with a main function, but no body. Essentially, a circuit with nothing inside of it! To make this more meaningful, you may add as many or as few basic operations as you like. Be sure to separate each statement with semicolons.

{ 0 o 1; 1 n 1; 1 a 1^1^1;}

Variables

Scirch allows users to set, get, and store values in variables. Variables can be named and are accessed by those names. Variables can be declared inside of declared functions, inside the main function, and outside of functions. Here is an example of getting and setting a variable:

{ testVar = 1; testVarTwo = 1; testVarThree = testVar^testVarTwo; }

Functions

This is where the routines for the circuits are meant to be written. Basic arithmetic and functions can be used inside of these functions and are declared like this:

sampleFunction{ *stmt list;* }

Declare any functions that you wish to make. Write and call them as many times as you like within the main function.

SampleFunction { 0 ^ 0; 0 o 1; }
sampleFunctionTwo {1 n 1; 1 a 0; }
{
sampleFunction();

```
sampleFunctionTwo();
}
```

Functions are meant to be sets of instructions, meaning nested functions are not possible. That would be the equivalent of putting another circuit component inside of another circuit component, which does not make sense in terms of the circuit.

Printing values

All of the function elements just discussed perform operations, but don't print anything to the console for the user to see. In order to print to the console, use the built in function, print()

```
{ print(0); print(1); print(0^1); print(0 o 1^1); }
```

This will print output to the console. Any values that you want to see the result of can be printed to the console using this method. Print can be used from the main function or from declared functions.

Scirch Manual

Development Process

We used an iterative process for creating Scirch. Planning was done at the beginning/middle of October with the following workflow:

- Basic scanner
    - Reading in our syntax
- Abstract syntax tree
    - Types and basic reduction
- Implementation
    - Basic operations
    - Storing Variables
    - Getting Variables
    - Creating functions
    - Calling functions
    - Printing values
- Bytecode
    - Translating into bytecode
    - Printing the bytecode
    - Implementing functions for each opcode
- Test cases
    - Write test code
- Functionality testing

The work has been iterative. A feature would be implemented and tested for bugs and errors. As components and features were added, integration testing was performed to make sure new components did not break old components.

Style guide

- All variable names and function names will be declared in camelCase.
- There will be a layer of indentation for each level of scoping.
- New lines will be created whenever you come across an “open”, “in”, “and”, ‘|’, or a “let”.
- Lines can be split at ‘;’ if they are too long, but this is not a strict rule.
- New lines can be inserted at 'if', 'then' and 'else'

Project Timeline

October 19th - Began work on project.
November 10th - Basic logic working, reading from source file, reads multiple lines of code.
November 17th - Continued work on scanner and ast.
November 25th - Added functionality for main function and variable storage (was not perfect yet)
November 29th - Added functionality for functions (was not perfect yet)
December 4th - Variables and Functions work as intended.
December 15th - Started bytecode.
December 19th - Eliminated shift/reduce conflicts and got printing bytecode listing to work.
December 20th - Basic bytecode operations working.
December 21st - Jump to subroutine bytecode working
December 21st - Test cases written
December 21st - Functionality testing begins

Primary Member Roles

Jeff Sinckler: Jeff was primarily responsible for every element of Scirch. Jeff built the AST interpreter and made source code work for the first time. He did the testing and refining to confirm operation. He also built the skeleton and translator from AST code into bytecode, as well as the executor that actually runs from a list of bytecodes or a list of AST expressions. Wrote the test case files and found the kinks in Scirch and fixed them. Also wrote the reports and slides. All of the code that is a part of Scirch right now was written by Jeff.

Brian Hunter: Brian was the secondary coder for Scrich. He went over Jeff’s code and while Jeff built the basis for most of the code, Brian went through and added functionality to them as he worked on adding in branching, if, while, and for loops, and returns, and did error checking and testing while doing so. Unfortunately none of those elements were finalized and are not in the final project because of it. Brian worked on additional testing of the code, including test case files.

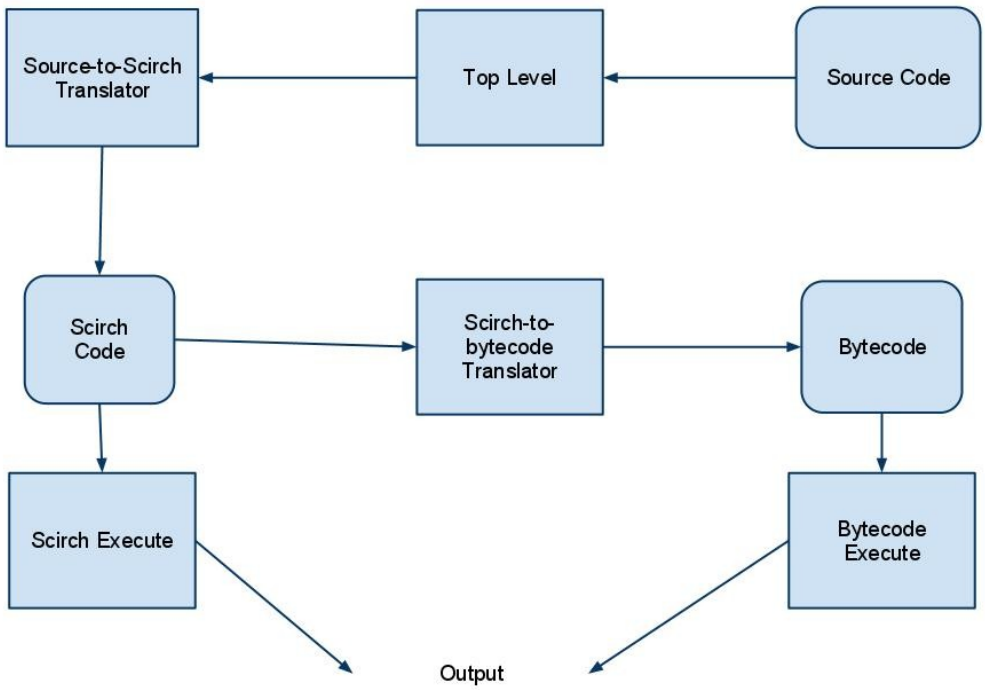
Resources

The compiler was, of course, written in oCaml. No other programming languages were used in this project. The bytecode that the compiler produces is also faux bytecode meant to represent processor opcodes.
Jeff primarily did his programming on a Mac OS X environment using vim as his editor.
Brian primarily did his programming on an Ubuntu distribution.
Google code was the repository used in this assignment. The link can be found here:

http://code.google.com/p/pltfuncproglang/

Project log: http://code.google.com/p/pltfuncproglang/updates/list
Google code subversion keeps track of all commits to the code as well as the date, and the comment associated with the commit. Please go to this link in order to view the project log.

Architectural Design



Rounded corners - String lists of code.
Rectangles - Elements of scirch that modify the string lists.
The source code is passed into the top level of Scirch. Top level passes the source code to the translator for translating. With Scirch code, there are two options. The Scirch code can be passed to the Scirch execution unit or the Scirch to bytecode unit. Scirch to bytecode returns a

list of bytecode strings, which is in turn passed to the bytecode executable. All executables return output.

Work Division

Scirch Executable - Jeff Sinckler/Brian Hunter
Bytecode Executable - Jeff Sinckler/Brian Hunter
Source-to-Scirch Translator - Jeff Sinckler
Top Level - Jeff Sinckler
Scirch to Bytecode Translator - Jeff Sinckler/Brian Hunter

Test Plan

Sample Programs

Test case code

```
testfunc{ 0^0; 1^1; }
testfunctwo{print(1^0);}
{
testfunc();
testfunctwo();
}
```

-- test-callfunc.sc

```
{
print(0^0);
print(0^1);
print(1^0);
print(1^1);
print(0o0);
print(0o1);
print(1o0);
print(1o1);
print(0n0);
print(0n1);
print(1n0);
print(1n1);
print(0a0);
print(0a1);
print(1a0);
print(1a1);
}
```

--test_gates.sc

```
testvar = 1;
testvartwo = 1;
{
testvarthree = 0;
print(testvar^testvartwo);
print(testvar^testvarthree);
}
```
--test_getvars.sc

```
{
print(0);
print(1);
print(1^1);
print("Hello, world!");
}
```
--test_print.sc

```
{
testvarone = 1;
testvartwo = 1;
testvarthree = 0;
print(testvarone^testvartwo);
testvartwo = testvarthree;
print(testvarone^testvartwo);
}
```
--test_setvars.sc

```
declare input1;
declare input2;
declare input3;
func1{input1 = -input1; input2 = -input2; input3 = -input3;}
func2{input1=input2^input3; input2=input3 o input1;}
{
input1 = 1;
input2 = 0;
input3 = 1;
func1();
print(input1);
print(input2);
print(input3);
func2();
print(input1);
print(input2);
print(input3);
```

}
--test-multioutput.sc

These tests were chosen to test the basic functionality of Scirch. Setting variables is tested to make sure values can properly be stored away in pertinent storage. Printing is tested to make sure results can actually be printed to the console. Getting variables is also tested to make sure that values can hold items and those items can be properly obtained. All of the basic gates are tested in a truth table style fashion to make sure they are all working properly. Global variables are tested to make sure inputs can be used in circuit components. Lastly, functions are tested in order to ensure the creation and calling of circuit subroutines.

Lessons Learned

Jeff Sinckler: In doing this project, I learned more about the general compilation process. Before, it was just magic, but now I realize how it works. I won't use this knowledge to start building compilers, but it will help me in my everyday programming. Being able to look at a program from a different perspective is always helpful, and writing code that generates fake machine code gave me a deeper perspective on how compilers integrate with stacks and such. As stated before, understanding this will allow me to look at all of the programming I will do in the future from a different perspective. For future students, I would suggest thinking outside of the box. You're language can be anything you want and can have any kind of syntax that you want. You don't have to be limited to "something like c" or "something like ocaml". Have fun with it and make something cool that you really enjoy. I am not saying that I didn't make a language that I didn't like, but I wish I could go back and make it look different (try to think of a cool new syntax unlike c syntax)

Brian Hunter: As a computer scientist its good to understand how the tools we use work. This better understanding of a compiler and how it works will certainly help with future programming design choices I make. There is lots of advice I have for future students. First off, meet with your group early and often, so you can decide on design choices you will make before you start coding, so that everyone is on the same page and time isn't wasted trying to understand what each person wrote. Of course, the standard advice, is start early, as waiting too long can cause lots of work to be invalidated if you can't finalize it in time. I would suggest working on all aspects together from the start, rather than each working on separate parts, so it is easier to bring it all together in the end.

Appendix A: Source code listings

bytecode.ml

```
type opcodes =
Lite of int
| Ent of int
| Ret of int
| Lod of int
| Str of int
| Slc of int
| Llc of int
| Bin of Gates_ast.operator
| Cal of int
| Hlt
| Drp
```

compile.ml

```
open Gates_ast;;
open Bytecode;;

let varIndex = ref (-1);;
let checkLocalref = ref (-1);;
let funcIndex = ref 0;;
let numLocals = ref 0;;
let localstrings = ref (Array.make 20 "null");;

let rec expr = fun a -> function
Print(e1) -> (fst a @ fst (expr ([], snd a) e1) @ [Cal (-1)], snd a)
| Print2(s1) -> a
| Lit(i) -> (fst a @ [Lite i], snd a)
| Assign(s, e1) -> varIndex := -1; Array.iteri (fun b c -> if (c.varName = s)
then varIndex := b
else ()) (snd a).varList;
if (fun a -> a = -1) !varIndex
then (
if List.mem s (Array.to_list !localstrings)
then ((Array.iteri (fun x y -> if (y = s) then checkLocalref := x else ()) !localstrings); fst a @ fst
(expr ([], snd a) e1) @ [Slc !checkLocalref], snd a)
else (!localstrings.(!numLocals) <- s; numLocals := (!numLocals+1); fst a @ fst (expr ([], snd a)
e1) @ [Slc (!numLocals-1)], snd a))(*fst a @ fst (expr ([], snd a) e1) @ [], snd a*)
else fst a @ fst (expr ([], snd a) e1) @ [Str !varIndex], snd a
| Var(s1) -> checkLocalref := (-1);
```

```
if List.mem s1 (Array.to_list !localstrings)
then (Array.iteri (fun d e -> if (e = s1) then checkLocalref := d else ()) !localstrings; (fst a @ [Llc !
checkLocalref], snd a))
else (Array.iteri (fun b c -> if (c.varName = s1) then varIndex := b else ()) ((snd a).varList); (fst a
@ [Lod !varIndex], snd a))
| Call(s1) -> Array.iteri (fun b c -> if (c.funcName = s1) then funcIndex := b else ()) (Array.of_list
(snd a).funcList); (fst a @ [Cal !funcIndex], snd a)
| Initialize(e1) -> (fst a @ [], snd a)
| Binop(e1, op, e2) -> (fst a @ fst (expr ([], snd a) e1) @ fst (expr ([], snd a) e2) @ [Bin op], snd
a)
```

execute.ml

```
open Gates_ast
open Firstgates2;;
open Bytecode;;

let execute_prgm prog =
let stack = Array.make 1024 0
and globals = Array.make (snd prog) 0
and locals = Array.make 100 0 in
let rec exec fpt spt pcont = match (fst prog).(pcont) with
 Lite i -> stack.(spt) <- i ; exec fpt (spt+1) (pcont+1)
 | Ret i ->
let next_fpt = stack.(fpt)
and next_pcont = stack.(fpt-1)
in stack.(fpt-1) <- stack.(spt-1); exec next_fpt fpt next_pcont
 | Lod i -> stack.(spt) <- globals.(i); exec fpt (spt+1) (pcont+1)
 | Str i -> globals.(i) <- stack.(spt-1); exec fpt spt (pcont+1)
 | Drp -> exec fpt (spt - 1) (pcont + 1)
 | Hlt -> ()
 | Cal(-1) -> print_endline (string_of_int stack.(spt-1)); exec fpt spt (pcont+1)
 | Cal i -> stack.(spt) <- pcont+1; exec fpt (spt+1) i
 | Ent i -> stack.(spt) <- fpt; exec spt (spt+1) (pcont+1)
 | Slc i -> locals.(i) <- stack.(spt-1); exec fpt spt (pcont+1)
 | Llc i -> stack.(spt) <- locals.(i); exec fpt (spt+1) (pcont+1)
 (*| Cal i -> print_endline "Cal: "; if (fun a -> a < 0) i
then print_endline (string_of_int stack.(spt)); exec fpt spt pcont+1 else stack.(spt) <- pcont + 1;
exec fpt (spt+1) i*)
 | Bin op1 -> let op2 = stack.(spt-2) and op3 = stack.(spt-1) in stack.(spt-2) <- (match op1 with
And -> andgate(op2, op3)
| Or -> orgate(op2, op3)
| Not -> notgate(op2)
```

```
| Nand -> nandgate(op2, op3)
| Nor -> norgate(op2, op3)); exec fpt (spt-1) (pcont+1)
in exec 0 0 0
```

firstgates2.ml

```
let andgate x = match x with
   (0,0) |  (0, 1) |  (1, 0) -> 0
    | (1, 1) -> 1
    | _ -> -1
let orgate x = match x with
   (0, 0) -> 0
 |  (0, 1) |  (1, 0) | (1, 1) -> 1
 |  _ -> -1
let notgate x = match x with
   0 -> 1
 |  1 -> 0
 | _ -> -1
let nandgate x = match x with
   (0, 0) |  (0, 1) |  (1, 0) -> 1
      |  (1, 1) -> 0
      | _ -> -1
let norgate x = match x with
   (0, 0) -> 1
 |  (0, 1) |  (1, 0) |  (1, 1) -> 0
 |  _ -> -1
```

gates_ast.ml

```
type operator = And | Or | Not | Nand | Nor

type expr =
 Binop of expr * operator * expr
 | Initialize of string
 | Assign of string * expr
 | Lit of int
 | Var of string
 | Call of string
 | Print of expr
 | Print2 of string
```

```
type stmt_list = expr list

type stmt = expr

type func = {
 funcName : string;
 body : stmt list;
}

type variable = {
 varName : string;
 varValue : expr;
}

type prgm = {
 mutable varList : variable array;
 funcList : func list;
 stmtList : stmt list;
}
```

gates_main.ml

```
open Gates_ast;;
open Firstgates2;;

let rec eval = fun info -> function
(* Lit just needs to take an integer *)
 Lit(x) -> (fst info, x)
 | Print(e1) -> print_endline (string_of_int (snd(eval info e1))); info
 | Print2(s1) -> print_endline s1; info
 | Assign(e1, e2) -> if List.exists (fun a -> a.varName = e1) (Array.to_list (fst info).varList)
then let e3 = snd (eval info e2) in Array.iteri (fun b c -> if (c.varName = e1) then (fst info).varList.
(b) <- {varName = e1; varValue = Lit(e3)} else ()) (fst info).varList; (fst info, snd (eval info e2))
else let e3 = snd (eval info (e2)) in
let newVariable = { varName = e1; varValue = Lit(e3) }
and oldInfo = fst info
in let newInfo = { varList = Array.append oldInfo.varList (Array.make 1 newVariable); funcList =
oldInfo.funcList; stmtList = oldInfo.stmtList }
in ();
newInfo, snd(eval info e2)
 | Call(name) -> let e1 = List.find (fun a -> a.funcName = name) ((fst info).funcList) in let e2 =
List.fold_left eval (info) (List.rev e1.body) in (fst info, snd e2)
 | Var(x) -> let e1 = List.find (fun a -> a.varName = x) (Array.to_list((fst info).varList))
```

```
in let e2 = eval info e1.varValue in (fst info, snd e2)
 | Initialize(e1) ->
if List.exists (fun a -> a.varName = e1) (Array.to_list (fst info).varList)
then info
else let newVar = {varName = e1; varValue = Lit(-1)}
in let info2 = {varList = Array.append (fst info).varList (Array.make 1 newVar); funcList = (fst
info).funcList; stmtList = (fst info).stmtList } in info2, 0
(* Binop needs to take an expression and an operator and an expression *)
 | Binop(e1, op, e2) ->
   let v1 = (snd (eval info e1)) and v2 = (snd (eval info e2))
in match op with
      And -> ((fst info), andgate(v1,v2))
      | Or -> ((fst info), orgate(v1,v2))
      | Nor -> ((fst info), norgate(v1,v2))
      | Not -> ((fst info), notgate(v1))
      | Nand -> ((fst info), nandgate(v1,v2))
```

gates_parser.mly

```
%{ open Gates_ast %}

%token AND OR NOR NOT NAND SEMI ASSIGN
%token LEFTPAREN RIGHTPAREN LEFTBRACE RIGHTBRACE DECL PRINT PRINTS EOF
%token IF ELSE FOR WHILE NOELSE
%token BINARY
%token <int> LITERAL
%token <string> ID

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left AND OR NOR NOT NAND
%left EQ NEQ

%start prgm
%type <Gates_ast.prgm> prgm

%%
prgm:
 { { varList = Array.make 0 {varName="null";varValue=Lit(0)}; funcList = []; stmtList = [] } }
 | prgm funcdecl { { varList = $1.varList; funcList = ($2 :: $1.funcList); stmtList = $1.stmtList } }
 | prgm LEFTBRACE stmt_list RIGHTBRACE { { varList = $1.varList; funcList = $1.funcList;
stmtList = ($1.stmtList @ $3) } }
```

```
 | prgm vardecl { { varList = Array.append $1.varList (Array.make 1 $2); funcList = $1.funcList;
stmtList = $1.stmtList } }
;
vardecl:
 DECL ID SEMI { { varName = $2; varValue = Lit(0) } }
;
funcdecl:
 ID LEFTBRACE stmt_list RIGHTBRACE { { funcName = $1;
    body = $3 } }
stmt_list:
 {[]}
 | stmt_list stmt {$2 :: $1}
;
stmt:
 expr SEMI {$1}
;
expr:
 expr AND expr{ Binop($1, And, $3) }
| expr OR expr{ Binop($1, Or, $3) }
| expr NOR expr{ Binop($1, Nor, $3) }
| NOT expr{ Binop($2, Not, Lit(0)) }
| expr NAND expr { Binop($1, Nand, $3) }
| ID ASSIGN expr { Assign($1, $3) }
| ID {Var($1)}
| DECL ID { Initialize($2) }
| ID LEFTPAREN RIGHTPAREN { Call($1) }
| LITERAL {Lit($1)}
| PRINT LEFTPAREN expr RIGHTPAREN { Print($3) }
| PRINTS LEFTPAREN ID RIGHTPAREN { Print2($3) }
;
```

gates_scanner.mll

```
{ open Gates_parser }

rule token =
 parse [' ' '\t' '\r' '\n'] { token lexbuf }
   | "</" { comment lexbuf }
   | '^' { AND }
   | '{' { LEFTBRACE }
   | '}' { RIGHTBRACE }
   | '(' { LEFTPAREN }
   | ')' { RIGHTPAREN }
   | 'o' { OR }
```

```
  | 'n' { NOR }
  | '-' { NOT }
  | 'a' { NAND }
  | '=' {ASSIGN}
  | ';' { SEMI }
  | "declare" {DECL}
  | "print" {PRINT}
  | "prints" {PRINTS}
  | ['0'-'9']+ as lit { LITERAL(int_of_string lit) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' ',' '!' '.']+ as word {ID(word)}
  | eof { EOF }
and comment = parse
  "/>" {token lexbuf}
  | _ {comment lexbuf}
```

scirch.ml

```
open Gates_ast;;
open Bytecode;;
open Compile;;

type action = Interpret | Compile | Execute

let _ =
let action = if Array.length Sys.argv > 2
then List.assoc Sys.argv.(1) [("-i", Interpret);("-c", Compile);("-e", Execute)]
else Interpret
in let source = Sys.argv.(2)
in let lexbuf = Lexing.from_channel (open_in source)
in let prgm = Gates_parser.prgm Gates_scanner.token lexbuf
in let add_function = fun b -> Ent 0 :: (fst (List.fold_left (Compile.expr) ([], prgm) (List.rev
b.body))) @ [Ret 0]
in let print_bytecode = function
Lite(_) -> print_endline "Lite"
| Hlt -> print_endline "Hlt"
| Str(_) -> print_endline "Str"
| Lod(_) -> print_endline "Lod"
| Bin(_) -> print_endline "Bin"
| Cal(_) -> print_endline "Cal"
| Ret(_) -> print_endline "Ret"
| Ent(_) -> print_endline "Ent"
| Slc(_) -> print_endline "Slc"
```

```
| Llc(_) -> print_endline "Llc"
| _ -> print_endline "something else"
in match action with
Interpret -> List.fold_left Gates_main.eval (prgm, 0) (List.rev prgm.stmtList)
| Compile -> let byt = (fst (List.fold_left (Compile.expr) ([], prgm) (List.rev prgm.stmtList)) @ [Hlt])
::
(List.map add_function prgm.funcList)
in let (offset, _) = List.fold_left (fun (a, b) c -> ( b::a, (b + List.length c))) ([], 0) byt
in let func_offset = Array.of_list(List.rev offset)
in let newBytecode = Array.of_list (List.map (function
Cal i when i > 0 -> Cal (func_offset.(i+1)) | _ as s -> s) (List.concat byt))
in List.iter print_bytecode (Array.to_list newBytecode);
(*(print_endline (string_of_int func_offset.(0))); (print_endline (string_of_int func_offset.(1)));
(print_endline (string_of_int func_offset.(2))); (print_endline (string_of_int func_offset.(3)));*)
(prgm, 0)
(*| Execute -> let oparray = List.fold_left (Compile.expr) ([], prgm) (List.rev prgm.stmtList)
in let byte = (Array.of_list ((fst oparray) @ [Hlt]), Array.length prgm.varList) in
Execute.execute_prgm byte; (prgm, 0)*)
| Execute -> let byt = (fst (List.fold_left (Compile.expr) ([], prgm) (List.rev prgm.stmtList)) @ [Hlt])
::
(List.map add_function prgm.funcList)
in let (offset, _) = List.fold_left (fun (a, b) c -> ( b::a, (b + List.length c))) ([], 0) byt
in let func_offset = Array.of_list(List.rev offset)
in let newBytecode = Array.of_list (List.map (function
Cal i when i > (-1) -> Cal (func_offset.(i+1)) | _ as s -> s) (List.concat byt))
in Execute.execute_prgm (newBytecode, Array.length prgm.varList); (prgm,0)
```