VC:
A Virtual Classroom Language

Carolyn Fine (crf2133), Marissa Golden (mjg2238),
Michelle Levine (mal2291)

December 22, 2015

Table of Contents

Introduction

The effects of technology have permeated every space in modern life, from robotic alarm clocks and hoverboard transportation to virtual realities and wireless activity trackers like fitbit. One focus of the future in technology is improving educational experiences through teachers' use of smartboards and programs for students to engage on their ipads. Often times the students are more technologically equipped than the teachers, having grown up holding an iphone before they are even old enough to attend school. Our Virtual Classroom (VC) language allows teachers to use coding for an educational purpose and generate arithmetic examinations using a simple, readable programming language. The language provides a higher level of abstraction from the usual coding syntax to enable teachers to create functions for a personalized test. By using VC, a teacher can create mathematic tests for the students with different types of questions, which are then displayed on a browser and auto evaluated once the student clicks submit. The automatic scoring of the computerized exams

Language Goals

Creating arithmetic tests can often be tedious and time-consuming for teachers, especially when writing similar expressions but using different integer values or operations. Furthermore, composing a variety of multiple choice answers using logical mistakes that a student might make can be a repetitive process, performing the same errors for each individual question. VC aims to automate this process, allowing teachers to auto-generate similar question forms with randomly generated integers and operators, as well as create functions to evaluate expressions in an understandably incorrect way (such as skipping over parentheses or not following the order of operations).

Overview

The VC language is mainly composed of arithmetic questions and functions composed by the teacher. These functions can include ways to randomly generate values for questions in a specific format or manipulate evaluation strategies to create incorrect multiple choice answers. The teacher can also call built-in library functions, such as display_radio and display_fillin, for the formatting of a multiple choice/true-false and fill_in questions respectively. The functions are called within a run function.
Chapter 2

Language Description

An examination question in VC is composed of a string expression saved in a primitive string variable. These questions can be manually constructed by the teacher or generated in a particular format specified in a function by the teacher. A helpful generator component is using an array of operator strings, and then using the built-in function rand to choose a random index in the operator array (generate random integer and then get the operator value stored in that index). Each of the random integers in the expression must be cast as strings and concatenated with the string operators to form the full expression. The run function can then call this function to generate multiple questions. Additionally, the teacher is able to create functions to evaluate the expression in various ways (correctly and incorrectly). They can then call one of the library display functions, passing in the question, all answer options in a list (with the correct answer repeated in the last index of that list - used for scoring evaluations), and the question number. The fill in type questions would call display_fillin and just pass the correct answer in a list of length 1.

Built in Functions

VC has a number of built-in functions that ease the programmer's own writing of functions and ability to utilize parts of strings or arrays. There are 8 built-in functions:

1. A print function that takes in any primitive data types and prints the value of the expression as it does in JavaScript. For example,

```
print(3 + 7); -> 10
```

would print the sum of the value of a plus 7.

2. A rand function that takes in an integer as its upper bound and returns a random integer between 0 (inclusive) and that integer (exclusive). For example,

```
int r = rand(10); -> r = 8
```

would return an integer between 0 and 9. Can be used by the teacher to generate questions with similar formats and varying random integer values.

3. An str function that takes in a variable or expression and converts it to a string. For example,

```
string s = str(503); -> s = "503"
```

converts integer 503 into string "503". Used most often to create string versions of questions.

4. An evalDouble function that takes in an expression string and evaluates it into a double. For example,

```
double a = evalDouble("1.0 * 3.2"); -> a = 0.32
```

This can be used to evaluate question strings to determine double answers.

5. An evalInt function that takes in an expression string and evaluates it into a integer. For example,

```
int a = evalInt("25 / 5"); -> a = 5
```

This can be used to evaluate question strings to determine integer answers.

6. A get_char_at function takes a string and index and returns the character at the specified index. For example,

```
string question1 = "1+5";
string c = get_char_at(question1, 1); -> c = "+"
```

returns the plus sign located at index 1 of the string question1. This can be used to manipulate question strings.

7. A length function that takes in a string or list and returns an integer of the length of argument. For example,

```
int i = length("hi there"); -> i = 8
```

This function can be used for loops, creating functions, and manipulating answers array.

8. An strReplace function that takes an input string, a string to replace, and a string to replace it with and returns the input string with the replacement. For example,

```
string s = strReplace( "(5+2)*3", "(", "" );
s = strReplace( s, ")", "" );
-> s = "5+2*3"
```

The first strReplace replace the open parentheses with an empty string (removed the open parentheses and the second strReplace did the same for the closing parentheses, thereby removing parentheses in the string. This can be used to manipulate string questions in order to generate incorrect multiple choice answers.

Compiling and Running a VC Program

In the src with the Makefile and the source code, run make, which compiles all vc source files. Use the ./vc operator to run your program as follows:

```
$ ./vc -[option] source/_files.vc
```

This will compile the vc code to JavaScript and HTML so you can open the HTML file and view the display in the browser.
Options for compiling vc code:

1. -a for printing the AST
2. -s for running semantic analysis on the source code
3. -j for compiling and generating the JavaScript and HTML code of the program
4. -h for help

Sample Code

This is an example program that creates a 10 question test. It creates the questions using the function createQ() that randomly selects an operator from the operators list, generates random integers for the operands, and creates the question string by concatenating the converted to string version of each integer operand and the selected operator. The program also has a function evalWithoutParens that removes the parentheses in the input question string and evaluates the newly formed string to provide an incorrect multiple choice answer. The function produceWrongAns takes in a string and adds a random integer (between 0 and 4) to that answer in order to offer another incorrect multiple choice answer. The function run then calls createQ, evalDouble, evalWithoutParens, produceWrongAnswer, and display_radio in a loop, thereby generating 10 questions and corresponding multiple choice answers. The function display_radio is a function in our library that takes in the question string, an array of answers, both incorrect and correct (and the correct answer is repeated in the last index of the array in order to calculate the scoring), and the name of the question (q1, q2, ...).

```
function evalWithoutParens returns double (string q){
      string newq=strReplace(q, "(", "");
      newq=strReplace(newq, ")","");
      double ans = evalDouble(newq);
      return ans;
}
function produceWrongAns returns double(string q){
      double b=evalDouble(q);
      b=b+rand(5);
      return b;
}
string list operators = ["+", "-", "*", "/"];
function createQ returns string (){
      int len=length(operators);
      int randInd= rand(len)-1;
      string q;
      string b= operators[randInd];
      randInd=rand(len) -1;
      string c = operators[randInd];
      q= str(rand(100))~b~"("~str(rand(100))~c~str(rand(100))~")";
      return q;
}
function run returns none (){

      int a;
      loop conditions (start: a=0; check: a < 10; change: a=a+1) do {
            string name = "q"~str(a);
            string q1=createQ();
            int len=length(operators);
            int randInd= rand(len)-1;
            string b= operators[randInd];
            q1=q1~b~createQ();
            double a1 = evalDouble(q1);
            double wa = evalWithoutParens(q1);
            double wa2 = produceWrongAns(q1);
            display_radio(q1,[wa,a1, wa2,a1],name);
      }

}
```

Language Reference Manual

Lexical Conventions

Comments

Multi-line comments are done using `/* */`, where anything in between those characters is ignored by the scanner.

Reserved Keywords

- `if`
- `else`
- `return`
- `loop`
- `start`
- `do`
- `conditions`
- `check`
- `change`
- `int`
- `bool`
- `double`
- `string`
- `list`
- `function`
- `none`
- `returns`
- `true`
- `false`

Identifiers

An identifier is a sequence of letters and digits with no spaces between them. The first character must be alphabetic but can be followed by any alphanumeric character. Underscore characters may be included in the identifier.

Literals

There are four types of literals in our language: string literals, integer literals, boolean literals, and double literals. A string literal is a sequence of characters surrounded by double quotes. Integer literals are a sequence of digits in base 10. Boolean literals can hold either the value `true` or `false`. A double literal is a sequence of digits that represents a fractional number.

Separators

In order to separate different statements a semicolon must be placed at the end of each statement.

Whitespace

We use the separators explained above to distinguish between statements. All whitespace (i.e. tabs and newlines) is ignored, but should be included for formatting as a good practice.

Identifiers and Scoping

The type of the identifier defines the type of values that are allowed to be stored in the variable. Functions define the scope and lifetime of the identifier. The scope of identifiers that are declared within a function lasts until the return statement of the function. Once an identifier is outside the scope of the function in which it was declared it is discarded. If a local variable shares its name with a global variable, the local variable (within the function) has precedence.

Types

VC supports the following data types:

Primitive Types

Primitive types include `int`, `double`, `bool`, and `string`. Type `none` can be used as a return type for functions.

Complex Types

Our complex type is `list` which maintains an ordered collection of elements. All of elements in a list are of one of the primitive types.

Declarations

Variable Declaration and Assignment

The type of the identifier determines the meaning of the values that are allowed to be stored in the variable. Thus the type must be explicitly specified when declaring an identifier as follows:

Declaration

Variables are declared using a type followed by an identifier and can additionally include an assignment using the = followed by an expression to be assigned to the variable. It would look as follows:

```
var_type var_name;
```

or

```
var_type var_name = expr;
```

Assignment

Once an identifier has been declared it can be assigned or reassigned by using the identifier followed by the assignment operator and and expression. For example:

```
var_name = expr;
```

Function Declaration

A function is declared by specifying a function name, return type, parameters (optional), for example:

```
function func_name returns var_type (params){ statements }
```

Parameters are formatted as follows:

```
var_type var_name
```

Each parameter of a function is comma separated. The body of the function is made up of a list of statements. These statements may include local variables and function calls. Variables that are declared within are considered local variables and can have the same name as global variables. References to local variables take precedence over global variables when being referenced within a function. There must be a function called `run` in order to run the program.

Expressions

1. Primitive literals of type `int`, `double`, `bool`, and `string`.
2. Complex literals of type `var_type list`.
3. Identifiers, given it has been declared properly.
4. Binary operators include arithmetic operators, concatenation, as well as comparison operators
5. Unary operators which include negative and not operators
6. Assignment operator
7. Call, which is used to run an existing function. The format of a call is the function name followed by open parenthesis, arguments separated by commas (if any), and close parenthesis. For example:

```
func_name (arg1, arg2, …)
```

8. List expressions is formated with an open bracket followed by expressions separated by commas and a close bracket. For example:

```
[ 1, 2, 3, 4]
```

9. List reference is formatted by using an identifier followed by an open bracket and integer or identifier and a close bracket. For example:

```
var_name[0]
```

Operators

Unary Operator

Symbol	Function
-	negative
!	not

Negative operator followed by an expression returns the negative value of the expression. The not operator followed by an expression returns a boolean value that is the opposite of the original expression. For example `!true` returns a value that is `false`.

Arithmetic Operators

Symbol	Function	format	accepted types

^	exponent	`expr1 ^ expr2`	`Int ^ Int -> Int,`
			`Double ^ Int -> Double,`
			`Double^Double -> Double`

*	multiplication	`expr1 * expr2`	`Int * Int -> Int,`
			`Int * Int -> Int,`
			`Double*Double -> Double,`
			`Int * Double -> Double,`
			`Double * Int -> Double`

/	division	`expr1 / expr2`	`Int / Int -> Int,`
			`Double/Double -> Double,`
			`Int / Double -> Double,`
			`Double / Int -> Double`

+	addition	`expr1 + expr2`	`Int + Int -> Int,`

			`Double+Double -> Double,`
			`Int + Double -> Double,`
			`Double + Int -> Double`

-	subtraction	`expr1 - expr2`	`Int - Int -> Int,`
			`Double-Double -> Double,`
			`Int - Double -> Double,`
			`Double - Int -> Double`

Note: the arrow in accepted types indicated the type of expression these operators return with the given expression types used.

Relational Operators

Symbol	Function
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal
!=	Not Equal
&&	Logical and
||	Logical or

The <, >, <=, >=, ==, != operators can only be used for types `int` and `double` and they return a boolean `true` or `false` depending on the relation between the two expressions. The && and || operators can be used with expressions which evaluate to `true` or `false` or with booleans.

Concatenate Operator

The concatenate operator, ~, can be used to combine expressions of type string.

Built In Functions

VC provides a number of built in functions to allow easy use of string manipulation and displaying.

1. `print` function accepts any valid expression and prints this to the console.log in the browser.

```
print(expr);
```

2. `rand` function takes an integer and returns a random number between 1 and that integer

```
rand(5);
```

3. `str` function takes expression and converts it to a string

```
str(5+7);
```

4. `evalDouble` function takes an expression of doubles and returns the evaluated double value

```
evalDouble(5.4/9.3);
```

5. `evalInt` function takes and expression of integers and returns the evaluated integer value

```
evalInt(5/9);
```

6. `display_radio` function takes a string (question), list (answers), and string (name of the question) as parameters. The list of answer must be formatted so that the last element of the list is the correct answer (in addition to the correct answer being listed as a separate element). This is used for multiple choice questions. For example:

```
display_radio("(3+2)*2", [9,10,7,11,10],"q1");
```

7. display_fillin function takes a string (question), list with one element, and a string(name of question) as parameters. For example:

```
display_fillin("(2^3)+5*2", [18], "q2");
```

8. `get_char_at` function takes a string and an integer as parameters and returns the character at that index. For example:

```
get_char_at("2+5*4", 3);
```

9. `length function` takes a string or list and returns the length. For example:

```
length([2,3,1,55]);
```

10. strReplace function takes a three strings as parameters, the original string, the string you are looking for, and the string you want to replace it with. For example:

```
strReplace("(4+10)", "+", "-");
```

Statements

A statement is anything that can makeup a line of VC code.

Statement Blocks

A statement block is a list of statements surrounded by curly braces. For example:

```
{
        statement;
        statement;
        statement;
        ....
}
```

Expressions

Any of the expression mentioned above followed by a semicolon are considered statements.

Conditional Statements

A conditional statement consists of an `if` statement with an optional `else` statement. For example:

```
if(expression){
  statements
}
```

or

```
if(expression){
  statements
}
else{
  statements
}
```

The compiler will check if the expression in the `if` statement evaluates to `true` and if it does it will evaluate the `statements` inside the if block. If the `if` statement evaluate to `false` and there is an `else` statement then the `statements` within the `else` block will be evaluated.

Return Statements

To return a value to the caller you can use the return keyword followed by an expression. It will return the resulting value of the evaluated expression. For example:

```
return expression;
```

Loops

A loop statement takes a variable assignment, and two expressions as parameters. For example:

```
int a;
loop conditions (starts: a=0; check: a<10; change: a=a+1) do{
   statements
}
```

The statements within the loop will be performed how ever many times the loop runs based on the conditions entered.

Variable Declarations

Variable declarations are statements that exist either globally or locally within function definitions.

Project Plan

Identify process used for planning, specification, development and testing

As a team we were lucky in the sense that we already knew each other. Also, we all lived off campus and were able to meet up closer to home when it worked out. Our schedules varied, but we found that after class was usually a convenient time to meet. During the semester we had a short checkup meeting on Mondays and a longer meeting on Wednesdays. For a stretch of time we also met our TA,

Prachi Shukla, on Wednesday evenings. When the semester hit its climax though, we decided to skip the TA meetings and devote more of Wednesday night to our own meeting and programming. We found that it was more efficient for us to do this and simply email Prachi our updates and questions.

At each meeting we discussed where we were in terms of our progress and what our goal was for the meeting/week. It was helpful in theory to have deadlines throughout the semester, but unfortunately after our Hello World demo we found out that we essentially had to rework our whole project.

None of us had prior experience in OCaml. We discovered that we worked best when we were all together, since each of our individual knowledge of OCaml was not sufficient to get productive work done on our own. When we worked together, we were able to fill in each other's gaps and debug better. We would take turns typing while the other two members of the team either offered suggestions and checked for errors or worked on separate files.

Programming style guide used by the team

- Use tab to indent
- "let" and "and" functions should be on same indentation line
- each block of code following "let" or "and" statements should be indented
- indentation should be consistent between different functions
- function names should be meaningful
- break into a new line if line of code is significantly longer than other lines in the file
- use '_' for naming variables (as opposed to camelCase)
- a bracket should open on the same line as its function declaration and close on its own line
- comments should be indented to the same tab as nearby comments

Project timeline

Date	Milestone
September 30	Proposal
October 26	LRM
November 15	Finished scanner, parser and ast, checked syntax with Menhir

November 16	Finished compiler, ran Hello World
November 30	Restarted project
December 22	Final Presentation and Final Report

As mentioned above, we had to restart our project with a month left in the semester. We were working at a reasonable pace up until that point, but starting nearly from scratch certainly set us back. We still managed to complete everything on time, but this required a very intense effort in the midst of studying for other finals. We developed each file in order (scanner, ast, parser, sast, semantic_check, code_gen, compiler), but as we went along we often had to return to files we thought were finished to change or add something new. We were able to test with Menhir fairly early on, but it was difficult to not be able to run other tests until the final stage of development.

Identify roles and responsibilities of each team member

Owing to our personalities and friendship, we were a very democratic team. We chose roles early on in the semester, but our responsibilities overlapped heavily. Marissa was the manager, Michelle was the language guru, and Carolyn was the system architect. We did not have a 4th member, so the original intention was to split responsibility for testing. Marissa was usually the one to coordinate our meetings and set reasonable goals for our meetings. Michelle was consulted for all language related questions, usually having to do with our preferred syntax. Carolyn set up the bitbucket repo that we used and organized the structure of our files.

Describe the software development environment used (tools and languages)

We started out by using a public github repo, but switched to Bitbucket after a suggestion from our TA. We developed in the Mac OS X environment and used Sublime Text 2 as our text editor. We wrote our compiler in OCaml and a few specific library function in JavaScript (those needed to display inputs and submit tests). We automated our compilation using a Makefile and ran our tests via a shellscript. We used ocamlc - c for our .ml and .mli files and ocamlyacc for parser.mly. For testing we use menhir on the parser.

Include your project log

There are two logs here. The first is our commit log from our public github repo, and the second is the log from bitbucket. Note that we usually worked on our project together with one person typing, so the number of commits per author does not reflect the division of labor.

Github log:

```
2e535cf - carolynfine: moving repo to bitbucket
718c8fc - carolynfine: Merge branch 'master' of
https://github.com/carolynfine/VC
6033ee2 - carolynfine: deleted unnecessary types from ast.ml
105a026 - malevin1: first commit
cc0d44f - malevin1: Merge branch 'master' of
https://github.com/carolynfine/VC
f131724 - malevin1: first commit
52f0c0d - carolynfine: added single line comment to scanner
fd252de - carolynfine: simplified scanner.mll
a43f9c4 - carolynfine: copied microc code into new folder
99d786c - Carolyn Fine: Added source code and test cases from class for
microc language
75f0ab8 - carolynfine: Initial commit
```

```
d760bbc - carolynfine: removed new_tests folder (tests are in
tests/vc_tests)
754de22 - carolynfine: fixed indentation 3
534d634 - carolynfine: fixed indentation 2
7859590 - carolynfine: fixed indentation
333e024 - Marissa Golden: testall.sh runs all scripts now
b54d591 - carolynfine: trying to clean up folders and make a test
script
a790989 - carolynfine: merge conflict w .ds store
973701a - carolynfine: added tests.  loop var needs to be declared
earlier
a9b5581 - carolynfine: added ; to code gen
f8720cf - Marissa Golden: added main example to tests
bbeb73d - Marissa Golden: Merge branch 'master' of
https://bitbucket.org/vc4115/vc
0707a18 - Marissa Golden: test remove paren
83d4a86 - carolynfine: redid .depend file
47c95af - carolynfine: deleted parser.output
413c42f - carolynfine: copied proposal intro to readme
ebd9a64 - carolynfine: deleted microc tests, deleted sast.ml
e3cc69c - carolynfine: Merge branch 'master' of
bitbucket.org:vc4115/vc
d0fa5ee - Marissa Golden: test removing parens
7cb30cc - carolynfine: code clean up
75b1837 - carolynfine: added test for if conditions
```

```
7bfafc9 - carolynfine: changed array to accept expr, renamed tests
982ba52 - Marissa Golden: added math.floor to evalInt
6451a06 - Marissa Golden: added strReplace function
2bc7761 - Marissa Golden: test14 produces random wrong ans
e7e11ab - Marissa Golden: updated test13
7b0999f - Marissa Golden: changed string checking back in semantic
check
6e2b418 - Marissa Golden: test13 works!!!!!!!"
b7eff4d - Marissa Golden: fixed an issue with passing arguments in
javascript
7d89d31 - Marissa Golden: issues with test11 using loops...
bc0d969 - Marissa Golden: added charAt function, only works inside
another functions
8d7835e - Marissa Golden: added html genertation
35fba8f - carolynfine: added library function to calculate number of
correct answers
bd032ef - carolynfine: added display functions as built ins and
tested
cc43d33 - carolynfine: added eval for int and double
6f70872 - carolynfine: built in functions are working beautifully
4ff64a9 - carolynfine: changed ans to li and allowed expr lists
03a16b8 - carolynfine: added tabs to make code prettier
1a0ef42 - carolynfine: tested list ref, fixed list reverse issue
5d16dbd - carolynfine: don't allow empty array decl, added ans_lit
for some reason
7227c69 - carolynfine: rewriting Ans and Ans list
3e36840 - Marissa Golden: fixed semi colon error in expressions
6332d6a - carolynfine: prelim tests
eb582d9 - Marissa Golden: fixed naming of js files
27e8b83 - carolynfine: Testing files
92e9c88 - carolynfine: IT COMPILES
c76901e - carolynfine: committing current  vc.ml
d6fe0b4 - carolynfine: wrote library function for displaying in
html/js
90ba2ce - carolynfine: no compile errors in code_gen but prolly has
issues
679b65d - carolynfine: working on code_gen
06059fd - Marissa Golden: changed makefile, deleted bytecode.ml
d4a7553 - carolynfine: added expr to statement check
fd42864 - carolynfine: fixed unused op error in check_stmt
294b245 - Marissa Golden: need to add matching for expr and Ans_list
0add0f6 - Marissa Golden: need to fix pattern matching stuff in
semantic check
217cfbe - Marissa Golden: got rid of things from n2n that we dont
have
916246b - Marissa Golden: fixed a syntax error, still have another
syntax error
28e13dd - Marissa Golden: error with mapping Ans_list
80546f1 - carolynfine: using a cheat to get error to disappear
33e80e6 - carolynfine: moved error down, but probably did not fix
issue with ans_list
```

```
796917e - Marissa Golden: syntax error
1144603 - Marissa Golden: added return types to functions
b3c298f - Marissa Golden: fixed one type error 500 more to go
a962697 - carolynfine: added vc.ml and commented out most of the
parts we don't need
361edfa - carolynfine: I finished going through this but there\'s
errors and I feel confused
6cc1aaa - carolynfine: updated Makefile and added sast.mli
6706eba - malevin1: created sast.ml and started working on
semantic_check.ml until line 481
44f6bde - carolynfine: added list reference
f02da2c - carolynfine: added lists
b907326 - carolynfine: cleaned up comments, restricted syntax
2c8307c - carolynfine: made expr optional, no reduce errors
45e555c - Marissa Golden: removed var_assign, and tried to to do it
as an expression
9cd862b - Marissa Golden: cleaned with reduce errors
1abdc88 - Marissa Golden: reduce error in loop
3a42ff7 - Marissa Golden: no reduce errors with loop!
879252b - malevin1: fixed FUNCTION in parser and started working on
FOR loop in parser, FOR still rejected
0c5da1a - malevin1: did make clean on the previous version
d5c0e91 - malevin1: changed program in parser and scanner and works
with make and menhir
d8a090b - malevin1: moved type program in ast
ef85a04 - Marissa Golden: reduce error
20884ab - Marissa Golden: issues with parser
6907969 - Marissa Golden: upadtes to ast, parser, and scanner
ccbdca4 - Marissa Golden: fixed ast
cd364a9 - Marissa Golden: upadtes to scanner and parser
20794d5 - Marissa Golden: starting over
50009cd - Marissa Golden: deleted comments
3b6bad8 - carolynfine: added print program
900ea2f - carolynfine: OMG hello world
f59ac30 - Marissa Golden: removed references to sast in vc.ml
a0da7d2 - Marissa Golden: removed code_gen.cmo target from Makefile,
fixed syntax error in vc.ml
d83a601 - carolynfine: vc_compile.ml compiles now
925716c - malevin1: A few more changes
1673c98 - malevin1: Merged changes from Caroline and Michelle in
vc_compile
7d484f7 - malevin1: Edited vc_compile and a few other files to fix
errors
a3af4df - carolynfine: bug fixes to vc_compile
01d3476 - carolynfine: everything up to scanner seems to compile
a59c554 - carolynfine: changed ast. parser now seems to compile
e1ceee1 - carolynfine: changed parser and ast based on TA feedback
69c2402 - carolynfine: works with menhir
86eeb85 - carolynfine: compiles with hello world cheat
76615ef - carolynfine: Merge branch 'master' of
https://bitbucket.org/vc4115/vc
```

```
a02f07f - carolynfine: debugging compiler
dc616c1 - Marissa Golden: edited test.sh
2e31b62 - Marissa Golden: updated paths in test.sh
9ae5a76 - Marissa Golden: edited the Makefile
aa71816 - malevin1: added unedited vc.ml, test.sh, Makefile
6e9fe81 - malevin1: small edit to vc_compile
2f79627 - malevin1: Merge branch 'master' of
https://bitbucket.org/vc4115/vc
6f73aba - malevin1: added first version vc_compile and edited ast.ml
a98438b - Marissa Golden: added a test to the test folder
7f10dab - malevin1: updated parser.mly - took away decls
ecb23a1 - malevin1: updated parser.mly - took away expr_opt
9a71ba4 - malevin1: updated scanner.mll, parser.mly, and ast.ml
b326be2 - malevin1: updated scanner.mll and parser.mly
0744bb1 - malevin1: updated scanner.mll
461dbe6 - Marissa Golden: testing push to bitbucket
c08124c - carolynfine: testing push
383ca00 - carolynfine: added readme
c4f4af4 - carolynfine: moved git repo to bitbucket
```

Architectural Design

Block diagram

The VC compiler takes a .vc input file and passes it through the scanner to tokenize it. The parser then takes those tokens and builds and AST. Next, the program is checked for semantics and the compiler generates an SAST. The SAST is translated into JavaScript code, which is outputted to a .js file along with a skeleton HTML file.

Components

Note: in this report we did not specify who wrote each component because all the files were the result of a joint collaboration.

Scanner

The scanner is used for lexical analysis. It takes an input program and reads it as a stream of characters. It discards irrelevant details such as whitespace and

comments and outputs a stream of tokens. Here we reject programs that have illegal syntax.
The stream of tokens can include identifiers, operators, keywords, punctuation, numbers and strings.

Parser and AST

The tokens produce in the scanner are run through the parser, where we defined our context free grammar. Here the program may be rejected in the syntax is not correct (i.e. tokens appear in the wrong order). Otherwise, the parser generates an Abstract Syntax Tree. The AST represents the overall structure of the program defined in the input program. The AST includes a variety of nodes, such as data types, function calls, identifiers, statements and the program itself.

Semantic Check and SAST

The scanner, parser and AST combine to form the frontend of the compile. The semantic check represents the beginning of the backend. At this point we are ensured that the program is syntactically correct - tokens are legal and appear in the correct order. However, the program may not be consistent so we need to check for that here. Semantic analysis takes the AST as input and either raises an error or outputs an SAST.
We verify many things in the semantic check. We ensure that variables are of the right type, that they are declared correctly and that their use within the scope is correct. We also check that function are not called before they are declared and that operations are done on the correct types. Variables are stored in an environment table along with their types so we can reference them later and check that their use is consistent. We verify that all elements in a list are of the same type here as well.

Code Generation

The SAST (semantically checked AST) is used by code_gen to translate the original input program into correct JavaScript syntax. We translate our variable type, statements and function calls into syntax compatible with JavaScript. Here we also handle a few built in function that are included in our language such as methods to display test questions and manipulate strings. All the tokens in the SAST are converted to strings that will be used in our target file.

Compilation

The final step is to take the output from the JavaScript code generation and put it in a .js file. We do this in our compilation and also create a static HTML page

that can be opened in the browser. A very important note about our language is that the file library.js must be included in the same directory as the .html and .js files produced by the VC compilation. This library file is fairly simple. It has a few JavaScript methods we wrote that the programmer can call as built-in functions. The two the programmer is likely to use are “display_radio” and “display_fillin”. These functions take a question string, answer array, and name as input and populate the html page with the correct format. The library also has an “onclick” function which calculates and alerts the score of a user who takes the generated arithmetic test. It was necessary to write the library in JavaScript because we had to call functions that would only be semantically correct in that language (e.g. “document.createElement()”).

Test Plan

- Show two or three representative source language programs along with the target language program generated for each
- Show the test suites used to test your translator
- Explain why and how these test cases were chosen
- What kind of automation was used in testing
- State who did what

Development of the Test Plan

Our test plan includes a test suite of comprehensive tests intended to check the various functionalities of a vc program, as well as a shell script to test the command to generate the JavaScript code for each of these tests.

Test Suites

The tests included in our test suite are in the table below. Many of the tests focus on checking the functionality of the smallest building blocks of a vc program, making it easier to locate bugs and ensure every part is functional. Some of the tests are a little more complicated as they test numerous functionalities integrated together, in a format more similar to how the programmer may use the coding language.

Component Testing

```
$ ./testall.sh
```

File	Functionality Tested

example.vc	Testing generating exam with multiple choice qs in for loop
testDisplayCalls.vc	Testing display calls from library-radio and fillin
testEmptyFunction.vc	Testing empty function that returns none
testEval.vc	Testing eval functions-evalDouble and evalInt
testExprIndex.vc	Testing using expression as index for array reference
testFunctionCall.vc	Testing function call
testGetCharAt.vc	Testing built-in function get_char_at
testGlobalVar.vc	Testing creation of global variable
testIf.vc	Testing if statements, using booleans
testIfConditions.vc	Testing if statements, using comparison conditions
testLength.vc	Testing built-in function length
testLengthInLoop.vc	Testing using length function within loop conditions
testList.vc	Testing list creations and references
testLoop.vc	Testing loop with proper conditions
testProduceAnswers.vc	Testing fillin and multiple choice questions with answers
testRandandStr.vc	Testing built-in rand and str functions
testRandQuestionDisplay.vc	Testing displaying questions generated using rand
testRemoveParen.vc	Testing multiple choice answer with remove parentheses
testReverseQuestion.vc	Testing reverse question string
testStrReplace.vc	Testing built-in function strReplace

Automation

We use an automated test script that compiles and generates the JavaScript and HTML code for each of the vc test files. Here is the code for the script:

```
#!/bin/bash

for file in vc_tests/*.vc
do
      ../vc -j "$file" &> "$file.out"
done
```

vc to JavaScript

Random Question Display

This vc code has a function to produce a question using random integers. The run function calls produceQuestion() twice, evaluates the answer for each, and then displays one as a fill in question and one as a multiple choice question.

VC file:

```
function produceQuestion returns string (){
      int a= rand(5);
      int b = rand(10);
      string q= str(a) ~ "+" ~ str(b);
      return q;
}
function run returns none (){
      string q = produceQuestion();
      int a = evalInt(q);
      display_fillin(q,[a], "q1");
      string q2= produceQuestion();
      int a2= evalInt(q2);
      display_fillin(q2,[a2], "q2");
      print(q);
}
```

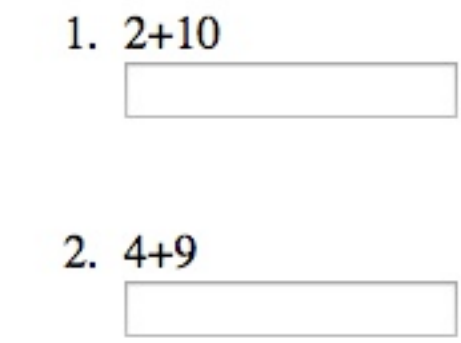
JavaScript file:

```
$( document ).ready(function() {

var q = produceQuestion();

      var a = Math.floor(eval(q));

      display_fillin(q, [a], "q1");
      var q2 = produceQuestion();

      var a2 = Math.floor(eval(q2));

      display_fillin(q2, [a2], "q2");
      console.log(q);
            function produceQuestion() {
      var a = Math.floor((Math.random() * 5 + 1));

      var b = Math.floor((Math.random() * 10 + 1));

      var q = a.toString().concat("+").concat(b.toString());

            return q;
      }

});
```

HTML file:

```
<!DOCTYPE html>

    <html lang="en-US">

    <meta charset="UTF-8">

    <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></scr

    <script src="./library.js" type="text/javascript"></script>

    <script src="./testRandQuestionDisplay.js" type="text/javascript"></script

     <body>

     <title>Test</title>

     <ol id="content">

     </ol>

     <button type="button" id="submit">Submit</button>

     </body>
</html>
```

Browser Display:



VC file (example explained earlier):

```
function evalWithoutParens returns double (string q){
     string newq=strReplace(q, "(", "");
     newq=strReplace(newq, ")","");
```

```
      double ans = evalDouble(newq);
      return ans;
}
function produceWrongAns returns double(string q){
      double b=evalDouble(q);
      b=b+rand(5);
      return b;
}
string list operators = ["+", "-", "*", "/"];
function createQ returns string (){
      int len=length(operators);
      int randInd= rand(len)-1;
      string q;
      string b= operators[randInd];
      randInd=rand(len) -1;
      string c = operators[randInd];
      q= str(rand(100))~b~"("~str(rand(100))~c~str(rand(100))~")";
      return q;
}
function run returns none (){

      int a;
      loop conditions (start: a=0; check: a < 10; change: a=a+1) do {
            string name = "q"~str(a);
            string q1=createQ();
            int len=length(operators);
            int randInd= rand(len)-1;
            string b= operators[randInd];
            q1=q1~b~createQ();
            double a1 = evalDouble(q1);
            double wa = evalWithoutParens(q1);
            double wa2 = produceWrongAns(q1);
            display_radio(q1,[wa,a1, wa2,a1],name);
      }

}
```

JavaScript file:

```
$( document ).ready(function() {
      var operators = ["+","-","*","/"];

var a;

            for (a = 0; a<10; a = a+1){
            var name = "q".concat(a.toString());

      var q1 = createQ();

      var len = operators.length;
```

```
     var randInd = Math.floor((Math.random() * len + 1))-1;

     var b = operators[randInd];

     q1 = q1.concat(b).concat(createQ());
     var a1 = eval(q1);

     var wa = evalWithoutParens(q1);

     var wa2 = produceWrongAns(q1);

     display_radio(q1, [wa,a1,wa2,a1], name);
     }
     function createQ() {
     var len = operators.length;

     var randInd = Math.floor((Math.random() * len + 1))-1;

     var q;

     var b = operators[randInd];

     randInd = Math.floor((Math.random() * len + 1))-1;
     var c = operators[randInd];

     q = Math.floor((Math.random() * 100 +
1)).toString().concat(b).concat("(").concat(Math.floor((Math.random()
* 100 + 1)).toString()).concat(c).concat(Math.floor((Math.random() *
100 + 1)).toString()).concat(")");
          return q;
     }
     function produceWrongAns(q) {
     var b = eval(q);

     b = b+Math.floor((Math.random() * 5 + 1));
          return b;
     }
     function evalWithoutParens(q) {
     var newq = q.replace("(","");

     newq = newq.replace(")","");
     var ans = eval(newq);

          return ans;
     }

});
```

HTML file:

```
<!DOCTYPE html>

                        <html lang="en-US">

                        <meta charset="UTF-8">

                        <script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"

                        <script src="./library.js"
type="text/javascript"></script>

                        <script src="./example.js"
type="text/javascript"></script>

                        <body>

                        <title>Test</title>

                        <ol id="content">

                        </ol>

                        <button type="button" id="submit">Submit</button

                        </body>

                        </html>
```

Browser Display:

1. 52/(93+46)+39-(6+61)
   ○ 18.55913978494624 ○ -27.625899280575 54 ○ -26.6258992805

2. 57+(12+96)*47/(14*37)
   ○ 77.71042471042472 ○ 66.7992277992278 ○ 70.799227799227

3. 8+(5*38)+78*(90*50)
   ○ 351198 ○ 351198 ○ 351202

4. 96*(29+77)+84-(3+67)
   ○ 2875 ○ 10190 ○ 10193

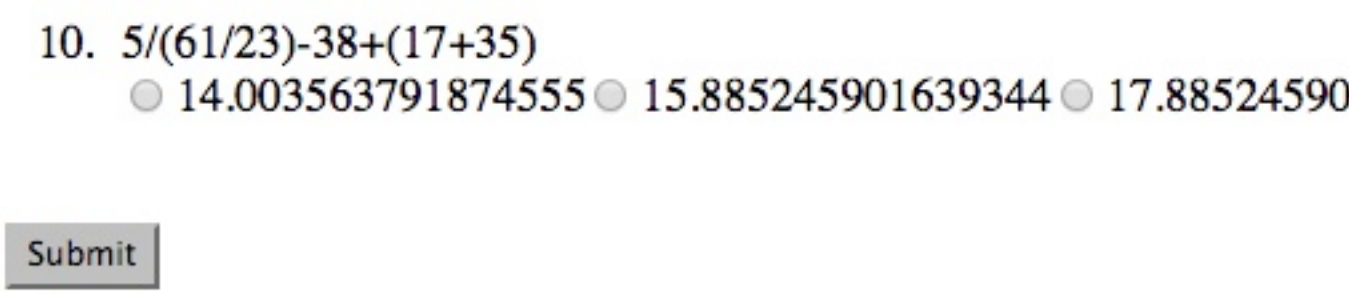
5. 86+(66/46)/49*(91-69)
   ○ 86.644188110026 62 ○ 86.64418811002662 ○ 87.6441881100 2

6. 57+(91+90)-10-(67+56)
   ○ 105 ○ 105 ○ 108

7. 11/(58*94)+85+(81+48)
   ○ 231.82758620689657 ○ 214.00201760821716 ○ 216.00201760

8. 54/(70/16)/19/(33+55)
   ○ 0.0000288362952836637 07 ○ 0.007382091592617909 ○ 5.007

9. 27*(13-58)+56-(39/20)
   ○ 347.05 ○ -1160.95 ○ -1159.95



Lessons Learned

Marissa Golden
If you get assigned a TA as your advisor for your project always double check with Edwards that you are taking your project in the direction he wants. Don't end up like us needing to start from scratch with a month left in the semester because you only checked in with the TA. This will make the end of your semester awful. And when he tells you to start early, just remember that it's not really possible to start early because you won't know enough to work on your project at the beginning of the semester. I also learned that I hate Ocaml and never want to program in it again. I also came to appreciate how much work goes into compiling a simple source code into as working program.

Carolyn Fine
I certainly learned a lot about the structure of a compiler while working on VC. It was exciting to come up with a new language and decide on its syntax. I have a newfound appreciation for all the languages I code in now that I have a better understanding for all the detail that goes into semantic analysis. I am also way more impressed with languages that don't require the programmer to distinguish between types because I realize now how hard it would be to verify that the variables are used correctly. I learned how valuable testing can be as well. Getting everything to compile was a big deal, but often we would discover through testing some gigantic issue that we hadn't thought of. It's hard to be thorough in testing when there's limited time, but if you're organized enough and don't think you'll change your language syntax you can probably write tests fairly early on and just wait until your compiler is done to test them. I learned that it is possible to cram this project into a shortened amount of time, but I really wish we knew earlier that we were not on the right track with our language compiler. My advice to the next generation would be that if you have a fundamental question about your design, go straight to the top and ask Prof. Edwards in person.

Michelle Levine
It would definitely be really helpful to get started early. Make sure to check in with Professor Edwards toward the very beginning, when you formulate your

project proposal, so that you don't end up putting hours of work into the code, only to discover that your edited proposal was not what he was looking for. Definitely pay a lot of attention to the lectures when he goes through the microc files. Those were helpful in terms of navigating the skeleton code. In the end, I still didn't enjoy using Ocaml, but I did become significantly more familiar with its syntax through working on the project. I also gained an appreciation for how complex it must be for languages like python to do semantic checking without defining any types or declaring variables. I now better understand certain compile errors that either came up throughout the project or that have shown up when compiling other code.

Appendix

Scanner.mll

```
{ open Parser }
let integer=['0'-'9']
let dec = ((integer+ '.' integer*) | ('.' integer+))
rule token = parse
[' ' '\t' '\r' '\n']     { token lexbuf }                    (*
Whitespace *)
| "/*"                          { comment lexbuf }            (*
Comments *)

| '('                          { LPAREN }
          (* Operators *)
| ')'                          { RPAREN }
| '{'                          { LBRACE }
| '}'                          { RBRACE }
| '['                          { LBRACKET }
| ']'                          { RBRACKET }
| ';'                          { SEMI }
| ','                          { COMMA }
| '+'                          { PLUS }
| '-'                          { MINUS }
| '*'                          { TIMES }
| '/'                          { DIVIDE }
| '='                          { ASSIGN }
| ':'                          { COLON }
| '~'                          { CONCAT }
| '^'                          { EXP }
| '!'                          { NOT }
| "=="                         { EQ }
| "!="                         { NEQ }
| '<'                          { LT }
```

```
| '>'                          { GT }
| "<="                         { LEQ }
| ">="                         { GEQ }
| "&&"                         { AND }
| "||"                         { OR }

| "if"                         { IF }
          (* Statements *)
| "else"                       { ELSE }
| "return"                     { RETURN }
| "loop"                       { LOOP }
| "start"                      { START }
| "do"                         { DO }
| "conditions"           { CONDITIONS }
| "check"                { CHECK }
| "change"                     { CHANGE }

| "int"                        { INT }
          (* Data types *)
| "bool"                       { BOOL }
| "double"                     { DOUBLE }
| "string"                     { STRING }
| "list"                 { LI }
| "function"             { FUNCTION }
| "none"                 { NONE }
| "returns"                    { RETURNS }

(* literals *)
| integer+ as lxm              { INT_LITERAL(int_of_string lxm)
}
| dec as lxm                   { DOUBLE_LITERAL(float_of_string
lxm) }
| '"' ([^'"']* as lit) '"'   { STRING_LITERAL(lit) }
| ("true" | "false") as lit    { BOOL_LITERAL(bool_of_string
lit) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm {
ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^
Char.escaped char)) }

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

Parser.mly

```
%{
  open Ast
%}

%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET
%token SEMI COMMA CONCAT
%token LOOP START CONDITIONS CHECK CHANGE DO COLON
%token PLUS MINUS TIMES DIVIDE EXP ASSIGN
%token EQ NEQ LT LEQ GT GEQ AND OR NOT
%token IF ELSE
%token FUNCTION RETURN RETURNS
%token INT DOUBLE STRING BOOL LI NONE
%token <int> INT_LITERAL
%token <string> STRING_LITERAL ID
%token <float> DOUBLE_LITERAL
%token <bool> BOOL_LITERAL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left CONCAT
%left PLUS MINUS
%left TIMES DIVIDE
%left EXP
%left NEG NOT

%start program
%type <Ast.program> program

%%

program:
 | decls EOF         { $1 }

decls:
 | /* nothing */                  { ([], []) }
 | decls global_var_declaration { ($2 :: fst $1), snd $1 }
 | decls function_declaration   { fst $1, ($2 :: snd $1) }

var_declaration:
```

```
 | var_type ID SEMI                    { Var($1, $2)
}                      /* int a; */
 | var_type ID ASSIGN expr SEMI       { Var_Decl_Assign($1, $2,
$4) }     /* int a = 1; */
 | var_type LI ID ASSIGN expr SEMI  { Var_Decl_Assign($1, $3,
$5) }      /* int list a = [1,2,3] */

global_var_declaration:
 var_declaration { $1 }

var_type:
 | INT                 { Int }
 | STRING              { String }
 | DOUBLE              { Double }
 | BOOL                { Bool }
/*  | var_type LI         { Li($1) } */
 | NONE                { None }


function_declaration:
/* function foo return int (int a, int b) { ... } */
| FUNCTION ID RETURNS var_type LPAREN formal_parameters RPAREN
LBRACE statements RBRACE
    {
      { fname = $2;
        formals = $6;
        return_type = $4;
        body = List.rev $9;
      }
    }

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

formal_list:
 | parameter                   { [$1] }     /* foo: Int */
 | formal_list COMMA parameter { $3 :: $1 } /* foo: Int, bar:
String */

parameter:
 | var_type ID    { Formal($1, $2) }        /* foo: Int */

statements:
 | /* nothing */        { [] }
 | statements statement { $2 :: $1 }

statement:
```

```
 | expr SEMI                                                          {
Expr($1) }                  /* 1 + 2 */
 | RETURN expr SEMI                                                 {
Return($2) }                /* return 1 + 2 */
 | LBRACE statements RBRACE                                         {
Block(List.rev $2) }        /* { 1 + 2 \n 3 + 4 } */
 | IF LPAREN expr RPAREN statement %prec NOELSE                     {
If($3, $5, Block([])) }
 | IF LPAREN expr RPAREN statement ELSE statement                   {
If($3, $5, $7) }
 | LOOP CONDITIONS LPAREN START COLON assign_opt SEMI
  CHECK COLON expr_comp SEMI
  CHANGE COLON expr_opt RPAREN DO statement                          {
Loop($6, $10, $14, $17) }   /* loop coditions (start: a=1;
check: a<5; a=a+1) { ... } */
 | var_declaration                                                  {
Var_decl($1) }              /* see above */


expr_opt:
 | /*nothing*/    { Noexpr }
 | expr           { $1 }

expr:
 | literal                      { Literal($1)
}               /* 18, "Awesome", 3.14, true */
 | binary_operation             { $1
}                         /* 4 + 3, "Two " ~ "words" */
 | unary_operation              { $1
}                         /* -1 */
 | ID                           { Id($1)
}                     /* question, answer, option */
 | ID LPAREN actuals_opt RPAREN { Call($1, $3)
}               /* display_radio(question, answer, name) */
 | LPAREN expr RPAREN           { $2
}                         /* (4 + 6) */
 | assign                       { $1
}                         /* a = 1 */
 | LBRACKET list_opt RBRACKET   { Li_list($2)
}                /* [1,2,3,4] */
 | ID LBRACKET expr RBRACKET    { Li_list_ref($1,
$3)}         /* answer[0] */



list_opt:
 | /*nothing*/   { [] }
```

```
 | li_list       { List.rev $1 }

li_list:
 | expr                  { [$1] }
 | li_list COMMA expr    { $3 :: $1 }

assign_opt:
 | /*nothing*/    { Noexpr }
 | assign           { $1 }

 assign:
 | ID ASSIGN expr              { Assign($1, $3)}

literal:
 | INT_LITERAL                  { Int_Literal($1) }          /*
1, 7, 100 */
 | STRING_LITERAL               { String_Literal($1) }       /*
"Marissa", "Michelle", "Carolyn" */
 | DOUBLE_LITERAL               { Double_Literal($1) }       /*
2.78 */
 | BOOL_LITERAL                 { Bool_Literal($1) }         /*
true, false */

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

actuals_list:
 | expr                     { [$1] }
 | actuals_list COMMA expr { $3 :: $1 }

binary_operation:
 | expr PLUS   expr        { Binop($1, Add,   $3) }
 | expr MINUS  expr        { Binop($1, Sub,   $3) }
 | expr TIMES  expr        { Binop($1, Mult,  $3) }
 | expr DIVIDE expr        { Binop($1, Div,   $3) }
 | expr CONCAT expr        { Binop($1, Concat, $3) }
 | expr EXP    expr        { Binop($1, Exp, $3)}
 | expr_comp               { $1 }

expr_comp:
 | expr EQ     expr        { Binop($1, Equal, $3) }
 | expr NEQ    expr        { Binop($1, Neq,   $3) }
 | expr LT     expr        { Binop($1, Less,  $3) }
 | expr LEQ    expr        { Binop($1, Leq,   $3) }
 | expr GT     expr        { Binop($1, Greater,  $3) }
 | expr GEQ    expr        { Binop($1, Geq,   $3) }
```

```
 | expr AND    expr        { Binop($1, And, $3) }
 | expr OR     expr        { Binop($1, Or, $3) }

unary_operation:
 | NOT expr                      { Unop(Not, $2) }
 | MINUS expr %prec NEG          { Unop(Neg, $2) }
```

Ast.ml

```
(* operators *)
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq |
Greater | Geq | Exp | Concat | And | Or
type unop = Neg | Not

(* expressions *)
type expr =
   Literal of literal
 | Id of string
 | Binop of expr * op * expr
 | Assign of string * expr
 | Unop of unop * expr
 | Call of string * expr list
 | Li_list of expr list
 | Li_list_ref of string * expr
 | Noexpr

(* literal *)
and literal =
 | Int_Literal of int
 | Double_Literal of float
 | String_Literal of string
 | Bool_Literal of bool
 | Li_Literal of expr list

type var_type =
 | Int
 | String
 | Bool
 | Double
 | Li of var_type
 | None

(* variable declarations *)
type var_decl =
 | Var of var_type * string                              (* int a
*)
```

```
 | Var_Decl_Assign of var_type * string * expr          (*
double b = 1.1 *)

type stmt =
   Block of stmt list
 | Expr of expr
 | Return of expr
 | If of expr * stmt * stmt
 | Loop of expr * expr * expr * stmt
 | Var_decl of var_decl
type formal =
 | Formal of var_type * string

(* function declaration *)
type func_decl = {
   fname : string;
   formals : formal list;
   body : stmt list;
   return_type: var_type;
 }

type program = var_decl list * func_decl list

(* get string version of all types *)
let string_of_binop = function
 | Add          -> "+"
 | Sub          -> "-"
 | Mult         -> "*"
 | Div          -> "/"
 | Equal        -> "=="
 | Neq          -> "!="
 | Less         -> "<"
 | Leq          -> "<="
 | Greater      -> ">"
 | Geq          -> ">="
 | And          -> "&&"
 | Or           -> "||"
 | Concat       -> "~"
 | Exp          -> "^"

let string_of_unop = function
 | Neg -> "-"
 | Not -> "!"

let rec string_of_expr = function
   Literal(l) -> string_of_literal l
 | Id(s) -> s
```

```
 | Binop(e1, o, e2) ->
     string_of_expr e1 ^ " " ^
     string_of_binop o ^ " " ^
     string_of_expr e2
 | Assign(v, e) -> v ^ " = " ^ string_of_expr e
 | Call(f, el) ->
     f ^ "(" ^ String.concat ", " (List.map string_of_expr el)
^ ")"
 | Unop(o, e) ->
     string_of_unop o ^ " " ^
     string_of_expr e
 | Li_list(li) -> "[" ^ String.concat ", " (List.map
string_of_expr li) ^ "]"
 | Li_list_ref(li, ind) -> li ^ "[" ^ string_of_expr ind ^ "]"
 | Noexpr -> ""

and string_of_literal = function
 | Int_Literal(l)    -> string_of_int l
 | Double_Literal(l) -> string_of_float l
 | String_Literal(l) -> "\"" ^ l ^ "\""
 | Bool_Literal(l)   -> string_of_bool l
 | Li_Literal(l)    -> "[" ^ String.concat ", " (List.map
string_of_expr l) ^ "]"


let rec string_of_var_type = function
 | Int -> "int"
 | Double -> "double"
 | String -> "string"
 | Li(t) -> string_of_var_type t ^ " list"
 | Bool -> "bool"
 | None -> "none"

let string_of_formal = function
 | Formal(t, id) -> string_of_var_type t ^ " " ^ id ^ ";"


let string_of_vdecl = function
 | Var(t, id) -> string_of_var_type t ^ " " ^ id ^ ";"
 | Var_Decl_Assign(t, id, expr) -> string_of_var_type t ^ " " ^
id ^ "=" ^ string_of_expr expr ^ ";"

let rec string_of_stmt = function
   Block(stmts) ->
     "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^
"}\n"
 | Expr(expr) -> string_of_expr expr ^ ";\n";
```

```
 | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
 | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
 | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
     string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
 | Loop(v1, e, v2, s) -> "loop conditions ( start: " ^
string_of_expr v1 ^ "; check: " ^ string_of_expr e ^ "; change:
" ^ string_of_expr v2 ^ ") do " ^ string_of_stmt s
 | Var_decl(v) -> string_of_vdecl v

let string_of_fdecl fdecl =
 "function " ^ fdecl.fname ^ " returns "^ string_of_var_type
fdecl.return_type ^ " (" ^ String.concat ", " (List.map
string_of_formal fdecl.formals) ^ ")\n{\n" ^
 String.concat "" (List.map string_of_stmt fdecl.body) ^
 "}\n"

let string_of_program (vars, funcs) =
 String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
 String.concat "\n" (List.map string_of_fdecl funcs)
```

Sast.mli

```
open Ast

type sformal =
 SFormal of Ast.var_type * string

type svar_decl =
   SVar of Ast.var_type * string
 | SVar_Decl_Assign of Ast.var_type * string * sexpr

and sexpr =
   SLiteral of sliteral * Ast.var_type
 | SId of string * Ast.var_type
 | SBinop of sexpr * Ast.op * sexpr * Ast.var_type
 | SAssign of string * sexpr * Ast.var_type
 | SUnop of Ast.unop * sexpr * Ast.var_type
 | SCall of string * sexpr list * Ast.var_type
 | SLi_list of sexpr list * Ast.var_type
 | SLi_list_ref of string * expr * Ast.var_type
 | SNoexpr

and sliteral =
 SInt_Literal of int
```

```
 | SDouble_Literal of float
 | SString_Literal of string
 | SBool_Literal of bool

and sstatement =
 SBlock of sstatement list
 | SExpr of sexpr
 | SReturn of sexpr
 | SIf of sexpr * sstatement * sstatement
 | SLoop of sexpr * sexpr * sexpr * sstatement
 | SVar_Decl of svar_decl

type sfunc_decl = {
 sfname : string;
 sformals : sformal list;
 sbody : sstatement list;
 sreturn_type : Ast.var_type;
}

type sprogram =
 SProg of svar_decl list * sfunc_decl list
```

Semantic_check.ml

```
open Ast
open Sast

exception Error of string;;

type environment = {
     functions: func_decl list;
     scope: string;
     locals: var_scope;
     globals: var_scope;
     has_return: bool;
     return_val: expr;
     return_type: var_type;
}

and var_scope = {
     prims: (string * var_type * expr) list;
     lists: (string * var_type * expr list) list
}

let beginning_scope = { prims = []; lists = [] }

let beginning_environment = { functions = []; globals =
```

```
beginning_scope; locals = beginning_scope; scope = "global";
has_return = false; return_val = Id("None"); return_type =
None}

let rec print_type ty =
     match ty with
          String -> "string"
          | Int -> "int"
          | Double -> "double"
          | Bool -> "bool"
          | Li(t) -> print_type t ^ " list"
          | None -> "none"

let get_id_from_expr ex =
     match ex with
          Id(v) -> v
          | _ -> raise(Error("Trying to get id from a non-id
expression\n"))


let check_arithmetic_binary_op t1 t2 =
     match (t1, t2) with
     | (Int, Int) -> Int
     | (Int, Double) -> Double
     | (Double, Int) -> Double
     | (Double, Double) -> Double
     | (_,_) -> raise(Error("Binary operation fails, wrong
element type"))

let check_equality t1 t2 =
     prerr_string(print_type t1 ^ " " ^print_type t2 ^"\n");
     if t1 = t2 then Bool else
     match (t1, t2) with
     | (Int, Double) -> Bool
     | (Double, Int) -> Bool
     | (_, _) -> raise(Error("Equality operation fails,
arguments not same type"))

let check_logic t1 t2 =
     match(t1, t2) with
     | (Int, Int) -> Bool
     | (Int, Double) -> Bool
     | (Double, Int) -> Bool
     | (Double, Double) -> Bool
     | (String, String) -> Bool
     | (_,_) -> raise(Error("Logical operation fails, arguments
not of correct types"))
```

```
let get_literal_type l = match l with
     Int_Literal(i) -> Int
     | Double_Literal(d) -> Double
     | String_Literal(s) -> String
     | Bool_Literal(b) -> Bool
     | _ -> raise(Error("Should not be using _ here"))

let get_type_from_id var_table id =
     if List.exists (fun (lid,_,_) -> lid=id) var_table.lists
then
     (let (_,ty,_) = List.find(fun (lid,_,_) -> lid=id)
var_table.lists in ty)
     else
          let (_,ty,_) = try List.find (fun (vid, _, _) ->
vid=id) var_table.prims with
          Not_found -> raise Not_found in ty

let check_for_var_existence var_table id =
     (List.exists(fun (vid,_,_) -> vid=id) var_table.prims
     || List.exists(fun (lid,_,_) -> lid=id) var_table.lists)

let set_default_val ty =
     match ty with
     Int -> Literal(Int_Literal(0))
     | Double -> Literal(Double_Literal(0.0))
     | Bool -> Literal(Bool_Literal(false))
     | String -> Literal(String_Literal(""))
   | Li(t) -> Literal(Li_Literal([]))
     | _ -> raise(Error("Not a primitive type"))

let get_new_env env func =
     let new_locals = List.fold_left (fun l f -> let (t, id) =
(match f with

     Formal(ty, vid) -> (ty, vid)) in
                                                          if
check_for_var_existence env.locals id then
raise(Error("Variable: " ^ id ^ " already exists in local
scope"))

     else (match t with
                                                           |
None -> raise(Error("Can't have a variable of type None"))
                                                          | _
-> let new_prims = (id, t, set_default_val t) :: l.prims in {l
with prims = new_prims}) ) env.locals func.formals and
```

```
     new_functions = func :: env.functions in
     {env with functions = new_functions; locals = new_locals;
scope = func.fname; return_type = func.return_type}

let update_prim_table var_table id ty v =
     let does_exist = check_for_var_existence var_table id in
     match does_exist with
     true -> raise(Error("Variable to declare: " ^id^ " already
exists"))
     | false ->
          let new_prims = (id, ty, v)::var_table.prims in
               {var_table with prims = new_prims}

let update_list_table var_table id ty v =
     let does_exist = check_for_var_existence var_table id in
     match does_exist with
     true -> raise(Error("Variable to declare already exists"))
     | false ->
          let new_lists = (id, ty, v)::var_table.lists in
               {var_table with lists = new_lists}

let rec check_expr env expr =
     match expr with
     | Literal(l) -> get_literal_type l
     | Id(v) -> prerr_string("check_expr: " ^ v ^ " id
called\n");
          (try get_type_from_id env.locals v with
          Not_found -> try get_type_from_id env.globals v with
          Not_found -> raise(Error("Id does not appear in
program")))
     | Unop(u, e) -> (match u with
          Not -> if check_expr env e = Bool then Bool else
raise (Error("Using NOT on a non-boolean expr"))
          | Neg -> if check_expr env e = Double then Double
               else if check_expr env e = Int then Int
               else raise (Error("Using a neg on a non int or
float expr")))
     | Binop(e1, op, e2) ->
          let t1 = check_expr env e1
               and t2 = check_expr env e2 in
          let binop_t = (match op with
                Add -> check_arithmetic_binary_op t1 t2
               | Sub -> check_arithmetic_binary_op t1 t2
               | Mult -> check_arithmetic_binary_op t1 t2
               | Div -> check_arithmetic_binary_op t1 t2
               | Exp -> check_arithmetic_binary_op t1 t2
               | Equal -> check_equality t1 t2
```

```
                | Neq -> check_equality t1 t2
                | Less -> check_logic t1 t2
                | Leq -> check_logic t1 t2
                | Greater -> check_logic t1 t2
                | Geq -> check_logic t1 t2
                | And -> if(t1, t2) = (Bool, Bool) then Bool
else raise (Error("Using AND on a non-boolean expr"))
                | Or -> if(t1, t2) = (Bool, Bool) then Bool else
raise (Error("Using OR on a non-boolean expr"))
                | Concat -> if (t1, t2) = (String, String) then
String else raise (Error("Using Concat on non-string expr")))
in binop_t
     (* built in functions *)
     | Call("print", el) -> prerr_string("Print function is
being called\n"); List.iter(fun e -> ignore(check_expr env e) )
el; None
     | Call("rand", el) -> prerr_string("Rand function is being
called\n"); List.iter(fun e -> ignore(check_expr env e) ) el;
Int
     | Call("str", el) -> prerr_string("Str function is being
called\n"); List.iter(fun e -> ignore(check_expr env e) ) el;
String
     | Call("evalDouble", el) -> prerr_string("Evaluate
function is being called\n"); List.iter(fun e ->
ignore(check_expr env e) ) el; Double
     | Call("evalInt", el) -> prerr_string("Evaluate function
is being called\n"); List.iter(fun e -> ignore(check_expr env
e) ) el; Int

     | Call("display_radio", el) -> prerr_string("display radio
function is being called\n"); List.iter(fun e ->
ignore(check_expr env e) ) el; None
     | Call("display_fillin", el) -> prerr_string("display
fillin function is being called\n"); List.iter(fun e ->
ignore(check_expr env e) ) el; None
     | Call("get_char_at", el) -> prerr_string("get_char_at
function is being called\n"); List.iter(fun e ->
ignore(check_expr env e)) el; String
     | Call("length", el) -> prerr_string("length function is
being called\n"); List.iter(fun e -> ignore(check_expr env
e))el; Int
     | Call("strReplace", el) -> prerr_string("strReplace
function is being called\n"); List.iter(fun e ->
ignore(check_expr env e)) el; String
     | Call(id, el) -> let func = (try List.find (fun f ->
prerr_string("Looking for function: " ^ id ^ " but finding
```

```
function: " ^f.fname^".\n");

          f.fname = id) env.functions with
          Not_found -> raise(Error("Function definition not
found"))) in
          (try List.iter2 (fun e f -> let ty = (match f with
                                                  Formal(t,_) -
> t) in
                                              let t = check_expr
env e in
                                                  if t <> ty
then raise(Error("Argument does not match expected argument
type"))) el func.formals with
          Invalid_argument s -> raise(Error("Entered the wrong
number of arguments into function"))); func.return_type
     | Assign(id, e) -> prerr_string("Assign being called from
check_expr\n");
               let tl = (try get_type_from_id env.locals id
with
                    Not_found -> try get_type_from_id
env.globals id with
                    Not_found -> raise(Error("Id does not
appear in program")))
               and tr = check_expr env e in
                    prerr_string("tl = " ^ print_type tl ^ " tr
= " ^ print_type tr ^ ".\n" );
               if (tl = tr) then
                    tl
               else
                    raise(Error("Type mismatch in
assignment!"))

     | Li_list(li) -> check_list_expr li env
     | Li_list_ref(li, ind) -> let t = check_expr env (Id(li))
          and i = check_expr env ind in
          if (i = Int) then
               t
          else
          raise(Error("Index must be of int type"))
     | Noexpr -> None

and check_list_expr l env  = match l with
| [] -> raise (Error("no empty lists allowed"))
| _ ->
let e = List.hd l in
let t = check_expr env e in
let s_l = List.map (fun ex -> let t' = check_expr env ex in
```

```
                    if t = t' then
                         t
                    else
                         raise (Error("Elements of a list
literal must all be of the same type"))) l
in  List.hd s_l

and get_sformal form =
     (match form with
          Formal(ty, s) -> SFormal(ty, s))

and get_sformal_list fl sfl =
     match fl with
     [] -> List.rev sfl
     | head :: tail -> let new_sfl = (get_sformal head) :: sfl
in
          get_sformal_list tail new_sfl

and get_slit env ex = match ex with
     Int_Literal(i) -> SInt_Literal(i)
     | Double_Literal(d)-> SDouble_Literal(d)
     | String_Literal(s) -> SString_Literal(s)
     | Bool_Literal(b) -> SBool_Literal(b)
     | _ -> raise(Error("Can't get sliteral of this"))

and get_sexpr env ex = match ex with
     Literal(l) -> (match l with
          Int_Literal(i) -> SLiteral(SInt_Literal(i), Int)
          | Double_Literal(d)-> SLiteral(SDouble_Literal(d),
Double)
          | String_Literal(s) -> SLiteral(SString_Literal(s),
String)
          | Bool_Literal(b) -> SLiteral(SBool_Literal(b), Bool)
          | _ -> raise(Error("Can't get SLiteral of this")))
     | Id(v) -> SId(v, check_expr env ex)
     | Unop(u, e) -> SUnop(u, get_sexpr env e, check_expr env
ex)
     | Binop(e1, op, e2) -> SBinop(get_sexpr env e1, op,
get_sexpr env e2, check_expr env ex)
     | Call(str, el) -> SCall(str, List.map (fun e -> get_sexpr
env e) el, check_expr env ex)
     | Assign(str, e) -> SAssign(str, get_sexpr env e,
check_expr env ex)
     | Li_list(li) -> SLi_list(List.map (fun l -> get_sexpr env
l) li, check_expr env ex)
     | Li_list_ref(id, ind) -> SLi_list_ref(id, ind, check_expr
env ex)
```

```
     | Noexpr -> SNoexpr

and resolve_envs old_env new_env =
     let new_prims = List.map (fun (id, ty, e) -> let v = (try
List.find (fun (vid, _, _) -> vid=id) new_env.locals.prims with
          Not_found -> (id, ty, e)) in v) old_env.locals.prims
     and new_lists = List.map (fun (id, ty, el) -> let v = (try
List.find (fun (lid, _, _) -> lid=id) new_env.locals.lists with
          Not_found -> (id, ty, el)) in v) old_env.locals.lists
in
     {new_env with locals = {prims = new_prims; lists =
new_lists}}

and check_stmt env stmt =
     prerr_string("Calling check_stmt\n"); match stmt with
     | Block(stmt_list) ->
          prerr_string("Calling Block from check_stmt\n");
          let new_env = env in
          let (checked_stmts, up_env) = List.fold_left (fun (l,
e) s -> let (checked_statment, up_e) = check_stmt e s in

                (checked_statment :: l, up_e)) ([], env)
stmt_list in
          let resolved_env = resolve_envs new_env up_env in
          (SBlock(List.rev checked_stmts), resolved_env)
     | Loop(v1, e, v2, s) ->
          prerr_string("Calling loop from check_stmt\n");
          let (s1, new_env) = check_stmt env s in
          (SLoop((get_sexpr env v1), (get_sexpr env e),
(get_sexpr env v2), s1), new_env)
     | Expr(e) ->
          prerr_string("Calling expression from check_stmt");
          (SExpr(get_sexpr env e), env)
     | Return(e) ->
          prerr_string("Return from check_stmt");
          let t1 = check_expr env e in
               (if not((t1=env.return_type)) then
                    raise (Error("Incompatible Return Type
using return type: "^ print_type t1 ^" but should be using type
"^ print_type env.return_type)));
          let new_env = {env with has_return = true;
return_type = t1; return_val = e } in
               (SReturn(get_sexpr env e), new_env)
     | If(e,s1,s2) ->
          prerr_string("Calling If from check_stmt\n");
          let t1 = check_expr env e in
               (if not(t1=Bool) then
```

```
                    raise (Error("If statement must be a
boolean")));
          let (st1,new_env)= check_stmt env s1 in
          let (st2,new_env2) = check_stmt new_env s2 in
               (SIf((get_sexpr env e), st1, st2),new_env2)
     | Var_decl(decl) ->
     prerr_string("Calling Var_decl from check_stmt\n");
     let (checked_stmt, up_env) =
          (match decl with
          Var(ty, id) -> prerr_string("Local_Var: Checking " ^
id ^ "\n");
               let new_table = (match ty with
               None -> raise(Error("Can't declare none"))
               | _ -> update_prim_table env.locals id ty
(set_default_val ty)) in
               let new_env = {env with locals = new_table} in
               (SVar(ty, id), new_env)
          | Var_Decl_Assign(ty, id, e) ->
prerr_string("Var_Decl_Assign: Checking " ^ id ^ " type is "^
Ast.string_of_var_type ty ^ "\n");
               let t_ex = check_expr env e in
               if (t_ex = ty) then
                    let new_table = (match ty with
                    | Li(t) -> update_list_table env.locals id
t []
                    | None -> raise(Error("Can't declare
none"))
                    | _ -> update_prim_table env.locals id ty
e) in
                    let new_env = {env with locals = new_table}
in
                    (SVar_Decl_Assign(ty, id, get_sexpr env e),
new_env)
               else
                    raise(Error("Type mismatch in local
variable assignment")))
          in (SVar_Decl(checked_stmt), up_env)

and get_checked_statements env stmts checked_statments =
     match stmts with
     | stmt :: tail ->
          let (checked_statement, new_env) = check_stmt env
stmt in
          get_checked_statements new_env tail
(checked_statement::checked_statments)
     | [] -> (List.rev checked_statments, env)
```

```
let check_function env func =
     prerr_string("Starting to check function: " ^ func.fname ^
".\n");
     let (sfstatements, up_env) = get_checked_statements
(get_new_env env func) func.body [] in
     ({sfname = func.fname; sformals = get_sformal_list
func.formals []; sbody = sfstatements; sreturn_type =
func.return_type}, {up_env with locals = env.locals})

let rec check_functions env funcs checked_funcs =
     let checked_functions =
          (match funcs with
          func :: tail ->
               let (checked_func, up_env) = check_function env
func in
               check_functions up_env tail (checked_func ::
checked_funcs)
          | [] -> checked_funcs) in
     checked_functions

let check_global env var =
     let (checked_global, up_env) =
          (match var with
          Var(ty, id) -> prerr_string("Global_Var: Checking " ^
id ^ "\n");
               let new_table = (match ty with
               | None -> raise(Error("Can't declare none"))
               | _ -> update_prim_table env.globals id ty
(set_default_val ty)) in
               let new_env = {env with globals = new_table} in
               (SVar(ty, id), new_env)
          | Var_Decl_Assign(ty, id, e) ->
prerr_string("Var_Decl_Assign: Checking " ^ id ^ " and type is
" ^ Ast.string_of_var_type ty ^ "\n");
               let t_ex = check_expr env e in
               if (t_ex = ty) then
                    let new_table = (match ty with
                    | Li(t) -> update_list_table env.globals id
t []
                    | None -> raise(Error("Can't declare
none"))
                    | _ -> update_prim_table env.globals id ty
e) in
                    let new_env = {env with globals =
new_table} in
                    (SVar_Decl_Assign(ty, id, get_sexpr env e),
new_env)
```

```
               else
                    raise(Error("Type mismatch in global
variable assignment. Type of t_ex is " ^ Ast.string_of_var_type
t_ex )))
          in
     (checked_global, up_env)

let rec check_globals_and_update_env env vars checked_vars =
     let (checked_globals, new_env) =
          (match vars with
          | var :: tail ->
               let (checked_global, up_env) = check_global env
var in
               check_globals_and_update_env up_env tail
(checked_global::checked_vars)
          | [] -> (checked_vars, env))
     in (checked_globals, new_env)

let run_program program =
     let (vars, funcs) = program in
     let env = beginning_environment in
     let (checked_globals, new_env) =
check_globals_and_update_env env (List.rev vars) [] in
   let checked_functions = check_functions new_env (List.rev
funcs) [] in
     SProg(checked_globals, checked_functions)
```

Code_gen.nl

```
open Ast
open Sast
open Printf

let imports =
 "$( document ).ready(function() {\n\t"

let set_default_val ty = match ty with
   Int -> "0"
   | String -> "\"\""
   | Double -> "0.0"
   | Bool -> "false"
   | Li(t) -> "[]"
   | _ -> "\"\""
```

```
let rec gen_var_type = function _ -> "var"

let gen_binop = function
 | Add          -> "+"
 | Sub          -> "-"
 | Mult         -> "*"
 | Div          -> "/"
 | Equal        -> "=="
 | Neq          -> "!="
 | Less         -> "<"
 | Leq          -> "<="
 | Greater      -> ">"
 | Geq          -> ">="
 | And          -> "&&"
 | Or           -> "||"
 (* see below for actual concat/exp use *)
 | Concat       -> "~"
 | Exp          -> "^"

let gen_unop = function
 | Not -> "!"
 | Neg -> "-"

let gen_literal lit = match lit with
 | SInt_Literal(i)      -> string_of_int i
 | SDouble_Literal(d)   -> string_of_float d
 | SBool_Literal(b)     -> string_of_bool b
 | SString_Literal(str) -> "\"" ^ str ^ "\""

let rec gen_literal_list ll = match ll with
 | [] -> ""
 | head::[] -> gen_literal head
 | head::tail -> gen_literal head ^ ", " ^ gen_literal_list
tail

let rec gen_expr expr = match expr with
 | SLiteral(l,t)           -> gen_literal l
 | SId(v,t)                -> v
 | SUnop(u, e, t)          -> gen_unop u ^ "(" ^ gen_expr e ^
")"
 | SBinop(e1, op, e2, t)   -> (match op with
           | Concat -> gen_expr e1 ^ ".concat(" ^ gen_expr e2 ^
")"
           | Exp    -> "Math.pow(" ^ gen_expr e1 ^ ", " ^
gen_expr e2 ^ ")"
           | _ ->gen_expr e1 ^ gen_binop op ^ gen_expr e2)
 | SCall(id, e1, t)        -> if(id="print") then gen_print e1
```

```
                                else if(id="rand") then
"Math.floor((Math.random() * " ^ gen_expr (List.hd e1) ^" +
1))"
                                else if(id="str") then gen_expr
(List.hd e1) ^ ".toString()"
                                else if(id="evalDouble") then
"eval(" ^ gen_expr (List.hd e1) ^ ")"
                                else if(id="evalInt") then
"Math.floor(eval(" ^ gen_expr (List.hd e1) ^ "))"
                                else if (id="get_char_at") then
gen_expr(List.hd e1)^".charAt("^gen_expr(List.nth e1 1)^")"
                                else if (id="length") then
gen_expr (List.hd e1) ^".length"
                                else if (id="strReplace") then
gen_expr(List.hd e1)^".replace("^gen_expr(List.nth e1
1)^","^gen_expr(List.nth e1 2)^")"
                                else id ^ "(" ^ gen_expr_list e1
^ ")"
 | SAssign(str, e1, t)     -> str ^ " = " ^ gen_expr e1
 | SLi_list(li, t)           -> "[" ^ String.concat ","
(List.map gen_expr li) ^ "]"
 | SLi_list_ref(str, i, t)   -> str ^ "[" ^ string_of_expr i ^
"]"
 | SNoexpr                   -> "\"\""

and gen_expr_list expr_list = match expr_list with
 | [] -> ""
 | head::[] -> gen_expr head
 | head::tail -> gen_expr head ^ ", " ^ gen_expr_list tail

and gen_formal h = match h with
 SFormal(type_spec, id) -> id

and gen_formal_list fl = match fl with
 | [] -> ""
 | head::[] -> gen_formal head
 | head::tail -> gen_formal head ^ ", " ^ gen_formal_list tail

and gen_print p = match p with
 | []         -> ""
 | head::[]   -> "console.log(" ^ gen_expr head ^ ")"
 | head::tail -> "console.log(" ^ gen_expr head ^ ")" ^
gen_print tail


and gen_sstmt stmt = match stmt with
 | SBlock(stmt_list) -> gen_sstmt_list stmt_list
```

```
 | SExpr(expr)       -> gen_expr expr ^ ";\n\t"
 | SReturn(expr) -> "\treturn " ^ gen_expr expr ^ ";\n\t"
 | SIf(expr,s1,s2) -> "\tif(" ^ gen_expr expr ^ ") {\n\t\t" ^
gen_sstmt s1 ^ "}\n\t\telse {\n\t" ^ gen_sstmt s2 ^ "}\n\n"
 | SLoop(e1, e2, e3, s1) -> "\tfor (" ^ gen_expr e1 ^ "; " ^
gen_expr e2 ^ "; " ^ gen_expr e3 ^ "){\n\t\t" ^ gen_sstmt s1 ^
"}\n"
 | SVar_Decl(vdec) -> gen_var_dec vdec ^"\n\t"

and gen_sstmt_list stmt_list = match stmt_list with
 | [] -> ""
 | head::[] -> gen_sstmt head
 | head::tail -> gen_sstmt head ^ gen_sstmt_list tail

and gen_var_dec dec = match dec with
 | SVar(ty,id) -> gen_var_type ty ^ " " ^ id ^ ";\n\t"
 | SVar_Decl_Assign(ty,id,e) -> "var " ^ id ^ " = " ^ gen_expr
e ^ ";\n\t"

and gen_var_dec_list var_dec_list = match var_dec_list with
 | [] -> ""
 | head::[] -> gen_var_dec head
 | head::tail -> gen_var_dec head ^ gen_var_dec_list tail

and gen_global_var_dec_list var_dec_list = match var_dec_list
with
 | [] -> ""
 | head::[] -> gen_var_dec head
 | head::tail -> gen_var_dec head ^ gen_var_dec_list tail

and gen_func_dec func =
if(func.sfname = "run") then gen_sstmt_list func.sbody
else "\tfunction " ^ func.sfname ^ "(" ^ gen_formal_list
func.sformals ^ ") {\n\t" ^ gen_sstmt_list func.sbody ^ "}\n"

and gen_func_dec_list fl = match fl with
 | [] -> ""
 | head::[] -> gen_func_dec head
 | head::tail -> gen_func_dec head ^ gen_func_dec_list tail

let prog_gen = function
 SProg(checked_globals, checked_functions) ->
   imports ^
   gen_global_var_dec_list (List.rev checked_globals) ^ "\n" ^
   gen_func_dec_list checked_functions ^
   "\n});\n"
```

vc.ml

```
open Unix
open String
open Filename
type action = Ast | Sast | JavaScript

let usage (name:string) =
 "usage:\n" ^ name ^ "\n" ^
   "        -a source.vc                  (Print AST of an vc source)\n"
   "        -s source.vc                  (Run Semantic Analysis over so
   "        -j source.vc [target.js]   (Generate JavaScript code for vc



let backend_path = "../backend/src/"
let target_path = backend_path ^ "com/vc/"

let read_process command =
 let buffer_size = 2048 in
 let buffer = Buffer.create buffer_size in
 let string = Bytes.create buffer_size in
 let in_channel = Unix.open_process_in command in
 let chars_read = ref 1 in
 while !chars_read <> 0 do
   chars_read := input in_channel string 0 buffer_size;
   Buffer.add_substring buffer string 0 !chars_read
 done;
 ignore (Unix.close_process_in in_channel);
 Buffer.contents buffer

(* Create a pipe for the subprocess output. *)
let readme, writeme = Unix.pipe ()

let _ =
 let args = Array.length Sys.argv in
   let action = if args > 1 then
     List.assoc Sys.argv.(1) [ ("-a", Ast); ("-s", Sast); ("-j", JavaSc
   else JavaScript in
     let len = String.length Sys.argv.(2)-3 in
     let jname = Sys.argv.(2) in
     let fname = String.sub jname 0 len in
     let input = open_in jname in
     let lexbuf = Lexing.from_channel input in
```

```
     let program = Parser.program Scanner.token lexbuf in
     match action with
             Ast -> let listing = Ast.string_of_program program in
                    print_string listing
           | Sast -> let env = Semantic_check.run_program program in
                          ignore env; print_string "Passed Semantic Ana
           | JavaScript -> let sast = Semantic_check.run_program progra
                    let javaScript_source = Code_gen.prog_gen sast in
                    if args == 3 then
                      let output_file = open_out (fname ^ ".js") in
                      output_string output_file javaScript_source; clo
output_file;
                      let html_source = "<!DOCTYPE html>\n
                      <html lang=\"en-US\">\n
                      <meta charset=\"UTF-8\">\n
                      <script
src=\"https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js
                      <script src=\"./library.js\"
type=\"text/javascript\"></script>\n
                      <script src=\"./"^basename (fname^".js")^"\"
type=\"text/javascript\"></script>\n
                      <body>\n
                      <title>Test</title>\n
                      <ol id=\"content\">\n
                      </ol>\n
                      <button type=\"button\" id=\"submit\">Submit</bu
                      </body>\n
                      </html>" in
                      let html_output = open_out (fname ^ ".html") in
                      output_string html_output html_source; close_out
html_output;
```

Makefile

```
OBJS = ast.cmo parser.cmo scanner.cmo semantic_check.cmo
code_gen.cmo vc.cmo


# Choose one
YACC = ocamlyacc
# YACC = menhir --explain

TARFILES = Makefile testall.sh scanner.mll parser.mly \
     ast.ml sast.mli code_gen.ml vc.ml \
```

```
     $(TESTS:%=tests/test-%.mc) \
     $(TESTS:%=tests/test-%.out)

vc : $(OBJS)
     ocamlc -o vc -g unix.cma $(OBJS)

.PHONY : test
test : vc testall.sh
     ./testall.sh

scanner.ml : scanner.mll
     ocamllex scanner.mll

parser.ml parser.mli : parser.mly
     $(YACC) parser.mly

%.cmo : %.ml
     ocamlc -c $<

%.cmi : %.mli
     ocamlc -c $<

vc.tar.gz : $(TARFILES)
     cd .. && tar czf vc/vc.tar.gz $(TARFILES:%=vc/%)

.PHONY : clean
clean :
     rm -f vc parser.ml parser.mli scanner.ml testall.log \
     *.cmo *.cmi *.out *.diff tests/vc_tests/*.js
tests/vc_tests/*.html tests/vc_tests/*.out

# Generated by ocamldep *.ml *.mli
ast.cmo :
ast.cmx :
code_gen.cmo : sast.cmi ast.cmo
code_gen.cmx : sast.cmi ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semantic_check.cmo : sast.cmi ast.cmo
semantic_check.cmx : sast.cmx ast.cmx
vc.cmo : semantic_check.cmo scanner.cmo parser.cmi code_gen.cmo
ast.cmo
vc.cmx : semantic_check.cmx scanner.cmx parser.cmx code_gen.cmx
ast.cmx
parser.cmi : ast.cmo
sast.cmi : ast.cmo
```

menhir_tests

```
menhir --interpret --interpret-show-cst parser.mly

EOF

// global
INT ID SEMI

// function
FUNCTION ID RETURNS NONE LPAREN INT ID RPAREN LBRACE INT ID
SEMI RBRACE
FUNCTION ID RETURNS NONE LPAREN RPAREN LBRACE RBRACE
FUNCTION ID RETURNS NONE LPAREN RPAREN LBRACE ID SEMI RBRACE

// loop
FUNCTION ID RETURNS NONE LPAREN RPAREN LBRACE LOOP CONDITIONS
LPAREN START COLON SEMI CHECK COLON ID LT INT_LITERAL SEMI
CHANGE COLON RPAREN DO LBRACE RBRACE RBRACE

FUNCTION ID RETURNS NONE LPAREN RPAREN LBRACE LOOP CONDITIONS
LPAREN START COLON SEMI CHECK COLON ID LT INT_LITERAL SEMI
CHANGE COLON ID ASSIGN ID PLUS INT_LITERAL RPAREN DO LBRACE
RBRACE RBRACE

FUNCTION ID RETURNS NONE LPAREN RPAREN LBRACE LOOP CONDITIONS
LPAREN START COLON ID ASSIGN INT_LITERAL SEMI CHECK COLON ID LT
INT_LITERAL SEMI CHANGE COLON ID ASSIGN ID PLUS INT_LITERAL
RPAREN DO LBRACE RBRACE RBRACE

FUNCTION ID RETURNS INT LPAREN RPAREN LBRACE LOOP CONDITIONS
LPAREN START COLON ID ASSIGN INT_LITERAL SEMI CHECK COLON ID LT
INT_LITERAL SEMI CHANGE COLON ID ASSIGN ID PLUS INT_LITERAL
RPAREN DO LBRACE ID ASSIGN LPAREN ID PLUS ID RPAREN SEMI RBRACE
RBRACE


// if
FUNCTION ID RETURNS NONE LPAREN RPAREN LBRACE IF LPAREN
BOOL_LITERAL RPAREN LBRACE RBRACE RBRACE

// else
FUNCTION ID RETURNS NONE LPAREN RPAREN LBRACE IF LPAREN
BOOL_LITERAL RPAREN LBRACE RBRACE ELSE LBRACE RBRACE RBRACE

// nested if
```

```
FUNCTION ID RETURNS NONE LPAREN RPAREN LBRACE IF LPAREN
BOOL_LITERAL RPAREN LBRACE IF LPAREN BOOL_LITERAL RPAREN LBRACE
RBRACE RBRACE RBRACE

// return
FUNCTION ID RETURNS NONE LPAREN RPAREN LBRACE RETURN
BOOL_LITERAL SEMI RBRACE


// arrays
INT LI ID ASSIGN LBRACKET RBRACKET SEMI
INT LI ID ASSIGN LBRACKET INT_LITERAL RBRACKET SEMI
INT LI ID ASSIGN LBRACKET ID RBRACKET SEMI
INT LI ID ASSIGN LBRACKET INT_LITERAL COMMA INT_LITERAL
RBRACKET SEMI


// reference array
INT LI ID ASSIGN LBRACKET INT_LITERAL RBRACKET SEMI INT ID
ASSIGN ID LBRACKET INT_LITERAL RBRACKET SEMI
```

testall.sh

```
#!/bin/bash

for file in vc_tests/*.vc
do
     ../vc -j "$file" &> "$file.out"
done
```

example.vc

```
function evalWithoutParens returns double (string q){
     string newq=strReplace(q, "(", "");
     newq=strReplace(newq, ")","");
     double ans = evalDouble(newq);
     return ans;
}
function produceWrongAns returns double(string q){
     double b=evalDouble(q);
     b=b+rand(5);
     return b;
}
string list operators = ["+", "-", "*", "/"];
```

```
function createQ returns string (){
     int len=length(operators);
     int randInd= rand(len)-1;
     string q;
     string b= operators[randInd];
     randInd=rand(len) -1;
     string c = operators[randInd];
     q=
str(rand(100))~b~"("~str(rand(100))~c~str(rand(100))~")";
     return q;
}
function run returns none (){

     int a;
     loop conditions (start: a=0; check: a < 10; change: a=a+1)
do {
          string name = "q"~str(a);
          string q1=createQ();
          int len=length(operators);
          int randInd= rand(len)-1;
          string b= operators[randInd];
          q1=q1~b~createQ();
          double a1 = evalDouble(q1);
          double wa = evalWithoutParens(q1);
          double wa2 = produceWrongAns(q1);
          display_radio(q1,[wa,a1, wa2,a1],name);
     }

}
```

testDiplayCalls.vc

```
function display returns none (){
     display_radio("2+2", [1,2,4,4], "name");
     display_fillin("question string", [1,2,3,4], "name");
}
function run returns none(){
     display();
}
```

testEpmtyFunction.vc

```
function myfunction returns none () {}
```

testEval.vc

```
string a = "10+100";
double b = evalDouble(a);
double d = evalDouble("10");
int e = evalInt("10+100");
int i = evalInt("1000");
```

testExprIndex.vc

```
string list operators = ["+", "-", "*", "/"];
function createQ returns string (){
     int len=length(operators);
     int randInd= rand(len)-1;
     string q;
     string b= operators[randInd];
     randInd=rand(len) -1;
     string c = operators[randInd];
     q= str(rand(4))~b~str(rand(12))~c~str(rand(6));
     return q;
}
function run returns none (){
     string q1 = createQ();
     string q2 = createQ();
     double a1 = evalDouble(q1);
     double a2 = evalDouble(q2);
     display_fillin(q1, [a1], "q1");
     display_fillin(q2, [a2], "q2");
}
```

testFunctionCall.vc

```
function myfunction returns none () {
     int a = 1;
     a = 1+2;
}

function myfunction2 returns int (double a) {
     double b = a;
     myfunction();
}
```

testGetCharAt.vc

```
string s = "hello";
function run returns none(){
     print(get_char_at(s, 3));

}
```

testGlobalVar.vc

```
int a = 5;
```

testIf.vc

```
/*FUNCTION ID RETURNS NONE LPAREN RPAREN LBRACE IF LPAREN
BOOL_LITERAL RPAREN LBRACE IF LPAREN BOOL_LITERAL RPAREN LBRACE
RBRACE RBRACE RBRACE*/

function ifel returns none (){
     if(true) {
     if(false) {
     int a=1;
     int b=2;
     int c=3;
     }
     }
}
```

testIfConditions.vc

```
function ifel returns none (){
     if(1 > 2) {
          int a = 2;
          int b = 3;
          if(b > a) {
               a=1;
          }
     }
}
```

testLength.vc

```
string h = "hello";
```

```
function run returns none(){
     int a=length(h);
}
```

testLengthInLoop.vc

```
int i=0;
function printIntLengthofString returns none (string s){
     int a;
     loop conditions (start: a=0; check: a < length(s); change:
a=a+1) do {
          i=i+1;
          print(i);
     }
}
function run returns none(){
     string h="hi";
     printIntLengthofString(h);
}
```

testList.vc

```
/*INT ANS ID ASSIGN LBRACKET RBRACKET SEMI
INT ANS ID ASSIGN LBRACKET INT_LITERAL RBRACKET SEMI
INT ANS ID ASSIGN LBRACKET INT_LITERAL COMMA INT_LITERAL
RBRACKET SEMI
*/

int list a =[1];
int list b = [1,2,3];
int c = b[0];
```

testLoop.vc

```
/*FUNCTION ID RETURNS INT LPAREN RPAREN LBRACE LOOP CONDITIONS
LPAREN START COLON ID ASSIGN INT_LITERAL SEMI CHECK COLON ID LT
INT_LITERAL SEMI CHANGE COLON ID ASSIGN ID PLUS INT_LITERAL
RPAREN DO LBRACE ID ASSIGN LPAREN ID PLUS ID RPAREN SEMI RBRACE
RBRACE*/
function myloop returns int () {
     int a=0;
     int b=1;
     int c=2;
     loop conditions (start: a=0; check: a < 10; change: a=a+1)
```

```
do {
          b = (c+c);
     }
}
```

testProduceAnswers.vc

```
function produceQuestion returns string (){
     int a= rand(5);
     int b = rand(10);
     string q= str(a) ~ "+" ~ str(b);
     return q;
}
function produceWrongAns returns int(string q){
     int b=evalInt(q);
     b=b+rand(5);
     return b;
}
function run returns none (){
     string q = produceQuestion();
     int a = evalInt(q);
     display_fillin(q,[a], "q1");
     string q2= produceQuestion();
     int a2= evalInt(q2);
     int w= produceWrongAns(q2);
     display_radio(q2,[a2,w, produceWrongAns(q2),a2], "q2");

}
```

testRandandStr.vc

```
int a = rand(10);
string b = str(a);
function callme returns none (){
     print(10);
     rand(100);
     str(a);
}
```

testRandQuestionDisplay.vc

```
function produceQuestion returns string (){
```

```
     int a= rand(5);
     int b = rand(10);
     string q= str(a) ~ "+" ~ str(b);
     return q;
}
function run returns none (){
     string q = produceQuestion();
     int a = evalInt(q);
     display_fillin(q,[a], "q1");
     string q2= produceQuestion();
     int a2= evalInt(q2);
     display_fillin(q2,[a2], "q2");
     print(q);
}
```

testRemoveParen.vc

```
/*test removing parenthesis*/

function evalWithoutParens returns int (string q){
     string newq=strReplace(q, "(", "");
     newq=strReplace(newq, ")","");
     int ans = evalInt(newq);
     return ans;
}
function run returns none (){

     string q1="(5+2)*3";
     int a1 = evalInt(q1);
     int wa = evalWithoutParens(q1);
     display_radio(q1,[wa,a1,a1],"q1");
}
```

testReverseQuestion.vc

```
function myloop2 returns string () {
     int a;
     int c=5;
     string q = "4+2*3-1";
     string newq = "";
     loop conditions (start: a=length(q)-1; check: a >= 0;
change: a=a-1) do {
          newq = newq~get_char_at(q, a);
     }
     return newq;
```

```
}

function run returns none() {
     string question = myloop2();
     print(question);
}
```

testStrReplace.vc

```
/*testing string replace*/
function run returns none () {
     string s = "hello world";
     print(strReplace(s, "hello", "hi"));
}
```

library.js

```
$( document ).ready(function() {
var content = document.getElementById("content"); // OL element

$("#submit").click(function(){
     var total = 0;
     var correct = 0;
     var answer = content.firstChild;
     while (answer) {
          var input = $(answer).find("input")[0];
          if (typeof input !== 'undefined') {
               total = total + 1;
               var type = $(input).attr("type");
               if (type == "text" && ($(input).val() ==
$(input).attr('id'))) {
                    correct = correct + 1;
               }
               else if (type == "radio") {
                    var name = $(input).attr('name');

     if($('input[name='+name+']:checked').attr('id')=='correct')
                         correct = correct + 1;
               }

          }
          answer = answer.nextSibling;
     }
     alert(correct + " correct out of " + total + " total
```

```
questions");

});
});


function display_radio(question, answer, name) {
     var li = document.createElement("LI");
     content.appendChild(li);
     var q = document.createTextNode(question);
     li.appendChild(q);
     var form = document.createElement("FORM");
     li.appendChild(form);
     for (var i=0; i<answer.length-1; i++) {
          var r = document.createElement("INPUT");
          r.setAttribute("type", "radio");
          r.setAttribute("value", answer[i]);
          r.setAttribute("name", name);
          if (answer[i] == answer[answer.length-1]) {
               r.id = "correct";
          }
          var label = document.createElement("LABEL");
          var text = document.createTextNode(answer[i]);
          label.appendChild(r);
          label.appendChild(text);
          form.appendChild(label);
     }
     content.appendChild(document.createElement("BR"));
}

function display_fillin(question, answer, name) {
     var li = document.createElement("LI");
     content.appendChild(li);
     var q = document.createTextNode(question);
     li.appendChild(q);
     var form = document.createElement("FORM");
     li.appendChild(form);
     var r = document.createElement("INPUT");
     r.setAttribute("type", "text");
     r.setAttribute("id", answer[0]);
     r.setAttribute("name", name);
     form.appendChild(r);
     li.appendChild(document.createElement("BR"));
     content.appendChild(document.createElement("BR"));
}

function clear() {
```

```
      while (content.firstChild) {
            content.removeChild(content.firstChild);
      }
}
```