Vowel

The Way for Wordsmiths

Name	UNI
Coby Simler	zys2102
Aidai Beishekeeva	ab5248
Lex Mengenhauser	am4958
Vikram Rajan	vjr2123

1 Introduction

1.1 Description

Vowel is an imperative, statically-typed programming language intended to be used to intuitively iterate over, operate on, and manipulate bodies of natural language. The Vowel language is distinguished by its powerful string data type, which has in-house support for certain set operations on space-delimited natural language strings. The language has C-style control flow, function declaration syntax, and typing, while providing the Python-style scripting ability that is preferable for string manipulation.

1.2 Motivation

Certain tasks require programmatic processing of large amounts of natural text data. Such tasks often involve analyzing, operating on, and comparing large bodies of structured and unstructured streams of space-delimited natural language. To perform such tasks, commonly used languages such as Python or Java rely on generic control flow and syntax such as loops and slicing, or otherwise importing third-party libraries to get the job done. Vowel aims to be the shorthand tool that enables programmers to more quickly and intuitively express these manipulations and compositions on natural text.

2 Language Tutorial

Note: This tutorial assumes familiarity with git and bash/shell.

2.1 Git Clone

The first step to using Vowel requires cloning the git repository into your local machine. To do this, you can use the *git clone* command.

```
$ git clone git@github.com:abeishekeeva/vowel.git
```

2.2 Docker

To ensure you have the appropriate libraries and configuration files, Docker must be installed on your machine. If this is not yet the case, you can install Docker here. You can verify that Docker is running correctly by running the command:

```
$ docker run hello-world
```

Next, navigate to the Vowel repository on your machine and invoke the Docker image as follows:

```
$ docker run --rm -it -v ‘pwd‘:/home/vowel -w=/home/vowel columbiasedwards/plt
```

2.3 Compilation and Execution

Once you’ve cloned the git repository and have the Docker image running, you must now compile the Vowel executable. To do this, you can simply run the command:

```
$ make
```

This will generate a *./vowel.native* executable, which can be used to run any Vowel program. For example, save the following Vowel program into *consonant.vwl*:

```
string x = “consonant”;
printstr(x);
```

You can now run *consonant.vwl* as follows:

```
$ ./vowel.native consonant.vwl
consonant
```

Congratulations — you ran your first Vowel program!

3 Language Manual

3.1 Reserved Words

There are a total of 15 keywords in Vowel.

```
while, for, in, if, else, return, true, false, int, float, char, bool,
string, len, slice
```

3.1.2 Identifiers

Variable identifiers are sequences of alphanumeric characters (letters and digits), including the underscore ‘_’. Variable names may start with an underscore, but not with a number, and cannot include any symbols. Furthermore, variables may not have the same name as a reserved word.

3.1.3 Comments

All comments will be opened with /* and will comment out everything until a closing */ is found. If no closing */ is found, an error will be thrown. There is no single-line comment syntax, as everything will follow this open and close comment convention.

```
/* Single-line comment */

/* Multi-line comment
This is still commented */
```

3.2 Types and variables

To declare a variable, first state the data type, then a legal variable name followed by the assignment operator, '=', and then initialize the variable by setting it equal to a literal or function statement corresponding to the stated type, followed by a semicolon. Declared variables must be initialized in place; to simplify scripting Vowel does not allow declaration without initialization.

3.2.0 Primitive Types

Vowel supports five primitive types.

- ➔ Int: Integer number
- ➔ Bool: True or false
- ➔ Float: floating point integer
- ➔ String: Text type
- ➔ Void: Null type

```
/* int year = 2021; */
/* bool tired = true; */
/* float balloon = 3.09; */
/* string s = "my string!"; */
/* void my func(){  print(1);}*/
```

3.2.1 Int Literal

An int literal is any sequence of numbers 0-9. The int type is 4 bytes.

```
int x = 4; int x = 345678;
```

3.2.2 FloatLiteral

A float literal is a floating pointer integer.

```
Float temp = 5.99;
```

3.2.3 String Literal

A string literal is any sequence of characters enclosed by double quotes.

```
string temp = “Hello World!”;
```

3.2.4 Boolean Literal

A boolean literal is either true or false. The boolean type is 1 byte.

```
bool x = true;
```

3.2.5 Array Literal

A list of elements of the same type separated by commas, and enclosed in square brackets.

```
int[] arr = [1, 2, 3, 4, 5];
string[] str_arr = [“first”, “second”, “third”];
```

3.3 Numerical Operators for Ints and Floats

Vowel supports numerical operators `+ - * / %` on Ints and floats. When used with two Ints, the operator will return type Int and when used with two floats it will return a float.

```
4.0 + 4.2  /*8.2*/
16 - 8     /*8*/
4.5 * 3    /*13.5*/
12 / 4     /*4*/
12 % 3     /*0*/
```

3.4 Comparison Operators

Vowel supports comparison operators `==, >, <, >=, <=, !=` for Ints and floats and `==, !=, &&, ||, !` for boolean types.

```
4.0 == 4.0    /*1*/
16 > 8        /*1*/
4 < 3         /*0*/
12 >= 4       /*1*/
12 <= 12      /*1*/
3.6 != 20.1   /*1*/
true == false /*0*/
true != false /*1*/
true && true  /*1*/
false || true /*1*/
!true         /*0*/
```

3.5 Assignment Operators

Assignment operators `=, +=` are supported for strings, ints, and floats with expressions of the same type. Additionally, ints support a decrement operator `-=.` Vowel provides the assignment operator = for boolean types. Variables must be declared and initialized in the same line.

```
int my_int = 5;
string great_string = “This is a great string”;
bool my_bool = true;

int my_int += 2;          /*my_int = 7*/
string a = “hi”; a += a; /* a = “hihi”*/

my_int -= 2               /*my_int =5*/
```

3.6 Strings

Key to Vowel is the native `string` data type, which is intended to hold text data. Vowel operators (e.g. *, +, -, &, |) perform distinct manipulations on string parameters, enumerated in the following table.

3.6.1 Operators for String data type

Operator	Data Types	Description

&	`string[] S = string A + string B`	Produces an array of space-delimited words that are contained in both the first and the second string. The array does not contain duplicate words, and is ordered by the appearance of the words in the first string.
*	`string A * int x = string A`	Modifies `string A` to contain the text of `string A` repeated `x` number of times.
-	`string[] C = string A - string B`	Produces an array of words in A which do not appear in B
+	`string c = string a + string b`	The plus operators produces a string that contains the first string, concatenated with the second string.
+=	`string a += string b`	This operator increments the value in string a with the value in string a, and reassigns this new value to string a.
slice(string a, int start, int end)	`string g =slice(string A, int start, int end)`	Creates a new `string g` that is a substring of `string A` from index `start` (inclusive) to index `end` (exclusive).
len(string a)	`int a = len(string a)`	Returns the number of characters in the input string in int format.

3.7 Arrays

Array is a collection of elements that contains elements of the same type. Arrays have to be declared and initialized. They can contain all primitive data types of Vowel. Arrays of arrays are not permitted in Vowel. The syntax for array is the following:

```
string[] names = ["coby", "aidai", "lex", "vik"]; /*for array of strings*/
int[] nums = [1,2,3];
```

3.7.1 Operations on Arrays

Access by index

Each element of an array can be accessed by index. Indexes start from 0 to n-1, where n is the length of the array. The value of the accessed element has to be saved to a variable.

```
int[] arr = [1, 2, 3, 4];
```

```
int x = arr[2];
```

Update
The values of the existing arrays can be updated by accessing the corresponding element by index and assigning a new value to it.

```
string[] str_arr = ["first", "second", "third"];
str_arr[1] = "fourth";
print(str_arr[1]); /* Prints fourth */
```

3.8 Statements & Expressions

Vowel has the following statements:

- While
- For
- Return
- If-else
- Expression

While loop
Statement that contains a while statement executes until the condition is evaluated to false. An example of a while loop that will run 3 iterations.

```
int i = 0;
while (i < 3) {
      printstr(i);
      i += 1;
}
/* 0 1 2 */
```

For loop
For statements in Vowel execute an initial statement, evaluates an expression, updates the statement and execute the loop body until the expression is false. General syntax is:

```
for(initial; expression; update) {
      statement;
}
```

Example:

```
int j = 0;
```

```
for (int j; j < 4; j=j+1) {
      print(j);
}
/* 0 1 2 3 */
```

If-else

If statements in Vowel allow for the conditional execution of a statement.

```
if (condition) {
      statement;
} else {
      statement;
}
```

The expression for condition has to be boolean type, otherwise it will throw an error. *If* condition evaluates to true inside if, the following statement will be executed, otherwise the statement inside else will be executed. Statements must be placed inside {}.

Return

Return statements are used to return control to the function that called the method. The return type must be of the same type as the function type.

```
int add(int x, int y) {
      int c = x + y;
      return c; /* will return int */
}
```

Expressions

Expressions are statement types that include:

- Binary operators
- Unary operators

Binary operators are described under section 3.2

3.9 Operations

In Vowel, each operator has a different level of precedence, and they are listed below along with their associativity, with the topmost operators having the highest levels of precedence.

Symbol	Operator	Associativity

[] ()	Array access Parentheses	Left
!	Not	Right
* / %	Multiply Divide Modulo	Left
+ -	Plus Minus/String difference	Left
< > <= >=	Less than Greater than Less than or equal Greater than or equal	Left
== !=	Equal Not equal	Left
&&	And	Left
||	Or	Left
&	Intersect	Left
+= -= =	Increment Decrement Assign	Right

3.10 Functions

Functions in Vowel are run sequentially. The syntax of a function definition is to have the return type of the function, the name of the function, followed by parentheses containing zero or more comma separated arguments formatted as *type name* pairs. Functions that do not return anything are type `void` which is a reserved keyword in Vowel. Arguments in Vowel are passed by value meaning a copy of them is made to be passed to the function. The body of the function is contained by curly braces that open after the function argument's closing parenthesis, and close after the last line of the function definition.

Functions are called by using the function name followed by parenthesis including the required number of arguments dictated by the function definition. No 'Main Function' is required. Vowel does not support functionality to include more arguments than dictated in the function definition. Calling the function above looks like:

```
myFunc(myString);
→ Hello World!
```

Variables can be set to contain the results of functions so long as they have the same type as the functions return value. Using the examples above, this looks like:

```
string w = myFunc(myString);
print(w);
→ Hello World!
```

4 Project Plan

4.1 Planning process

The project scope and outline was determined in the early Fall of 2021. After identifying a shared frustration about the limited extent to which existing programming languages are tailored toward solving text based manipulation problems, the team members consulted with T.A. Max Levitach and T.A. John Hui about an early conception of Vowel. Through weekly meetings, we iterated on our initial ideas by logging all action items in a shared weekly agenda and notes document.

4.2. Specification

The process of determining the specifications of our language began with two brainstorm meetings at which we discussed what we felt were shortcomings of existing programming languages for string analysis, manipulation, and parsing. We arrived at three major specification categories in which we saw room for improvement: control flow, string operations, and typing. Within each category, we enumerated the existing tools used for string analysis and manipulation. For example, in the control flow category, we listed existing norms and behaviors in Java, Python, and C that are commonly used to parse over words in a string, such as for loops and while loops. In string operations, we noticed the limited operation set on strings in Python, generally limited to “+” for string concatenation and * for string repetition.

With an understanding of these shortcomings, we then brainstormed a comprehensive list of ideas to augment the built in functionality of those languages to better express the kind of manipulation that we wanted. We then shortened the list to about 5 of the most desirable features. Otherwise, we sought to emulate the simplicity of Python by not requiring a main method, while ensuring that the typing guardrails of Java were in place for beginner programmers.

4.3. Development process

We developed this project iteratively by first dividing up foundational tasks and functionality, and subsequently splitting Vowel-specific tasks to each team member. In all cases, our implementation followed the stages of the compiler—starting with parser and scanner, moving to AST, semant, and SAST, and finally moving to code generation. For example, we began by implementing strings and arrays as the foundation for our later functionality. In certain cases, when two team members were simultaneously stuck on their given responsibilities, the two team members switched tasks so as to increase the chance of a breakthrough and expose each team member to different parts of the code base. We implemented in increments of one to two weeks throughout the semester, and then at a higher velocity in the final month of the semester.

For each new feature, the directly responsible individual for that feature would create a new git branch. All development related to that feature would occur on the branch. Branches were iteratively merged into the master branch as development for that feature finished.

4.4. Programming Style Guide

Vowel programs are written in a style that closely resembles that of Java's editing and formatting style. Specific rules include:

- Functions are declared at the top of Vowel files in camel case.
- Variable names use an underscore between words. Variable names do not start with numbers. Numbers should be used after alphabetical characters within variable names. E.g. variabel1 and variable2 are valid variable names. 1variable and vari2able are not.
- Indentation (4 spaces) is used within functions and loops to indicate sub statements.
- Comments are started and ended at the same level of indentation at which the statement being commented out is expressed.

4.5 Project timeline

Date	Description
September 26	Language conceptualized
September 29	Initial language proposal authored
October 10	Identified string operations as preliminary focus
October 23	First version of parser completed
October 24	Language Reference Manual Completed
November 10	Second version of Scanner and Parser completed

November 23	Began work on AST and Semantically Checked AST
December 1	Began String Operator: Increment, Concatenation, Equality
December 10	Developed test suite and debugging
December 20	All language files complete
December 21	Project presentation complete
December 22	Submitted project final report

4.6. Roles & responsibilities

The following is a broad breakdown of roles and responsibilities in this project. In practice, every team member helped out across all areas and made substantial contributions to the codebase.

- Coby Simler (Language guru): Coby initially offered the idea of a programming language tailored toward parsing natural language, and identified shortcomings of existing languages at solving these tasks. He took the lead role in conceptualized string set operations, built-in functionality, and more.
- Lex Mengenhauser (System architect): Lex spearheaded the language architecture efforts, contributing most of all team members in the area of linking vowel_func.c and code generations. She also was pivotal in debugging.
- Vikram Rajan (Tester): Vik spearheaded the testing process and ensured that all areas of Vowel were adequately tested. He also took charge of implementing critical pieces of the languages such as arrays and several binary operators.
- Aidai Beishekeeva (Manager): Aidai consistently held team members responsible for setting and meeting deadlines and following the pace for development. She managed version control, branching, and merging everyone's contributions, and effectively led team meetings.

Decrement	Lex
Increment	Aidai
Modulus	Vik
Strings as data type	Coby
String concatenation	Coby
String comparison	Lex
Arrays as data type	Aidai & Vik
Array access	Vik
Array literal	Aidai

String intersection & difference (ocaml)	Coby & Lex
String intersection & difference (c functions)	Aidai & Vik & Lex & Coby
Removing main	Aidai & Vik
String length, slice, multiplication	Lex

4.7 Development Environment

The Vowel team relied on a technical stack including the following tools:

- GCC — to perform linking between our generated LLVM code and the C code which defines Vowel string operations.
- Ocaml 4.08.1, Ocamlyacc, and Ocamllex — for scanning, parsing, building a syntax tree, and generating LLVM code for our source program.
- Github-hosted Git Repository — for hosting our code and version control.
- IDEs — Group members relied on various integrated development environments including VSCode, Vim, and XCode. Plugins with these systems were used for debugging and merging different branches of the codebase.

4.8 Project Log

```
commit e25c0cd0e3c860b0e9090d16ff2fdf4de7455682
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Tue Dec 21 14:23:20 2021 -0500

    changed file extensions to vwl

commit 41a2c1c0fdbf1b32971b3d44f740f255b94f2ae2
Merge: 3720e27 62d0ac0
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Tue Dec 21 14:15:51 2021 -0500

    merged string slice

commit 3720e2711095b6cd31d27c20bf5c9ca4f2f3e550
Merge: 31674b7 c84a4c1
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Tue Dec 21 14:09:59 2021 -0500
```

```
    merged slice

commit 31674b73740be4770632ea9fd23173791271b524
Merge: 9a33fdf c9b602e
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Tue Dec 21 13:58:21 2021 -0500

    merged string minus

commit 9a33fdf8030670dc65f1adbb2c21af308677a583
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Tue Dec 21 13:47:22 2021 -0500

    merged intersection

commit ed60b05d5197e199b77c59d7424bec53d1d60b7e
Merge: 1e89385 c9a5242
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Tue Dec 21 13:41:34 2021 -0500

    merged intersection

commit 1e89385911e27ce234c447f9de9bcaee7f9d6c93
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Tue Dec 21 13:19:11 2021 -0500

    fixed tests

commit c9b602eb961b4ca88fe861c89d3d566bdcb1fdcb
Author: root <am4958@columbia.edu>
Date:   Tue Dec 21 12:26:10 2021 -0500

    string subtraction working

commit c9a5242e901ca629173361074168533b007a935c
Author: Coby <simlercoby@gmail.com>
Date:   Tue Dec 21 12:03:29 2021 -0500

    fixed quote mark issue with string intersection

commit a50df6349b56b4555649b6230826ab52a2b17f4f
Author: Coby <simlercoby@gmail.com>
Date:   Tue Dec 21 11:21:17 2021 -0500
```

```
    added c program for string intersection in vowel_func.c. Removed
comments and substring method, which was causing a segfault

commit fc0168c8be0b0a56c56c813fae0854cbc54226d9
Author: Coby <simlercoby@gmail.com>
Date:   Tue Dec 21 11:19:54 2021 -0500

    modified ast and codegen to accomodate for string intersection

commit 2fc4e58710cca5877c0ccac1e5f16d7555aa1cf1
Merge: da9341f f88f927
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Mon Dec 20 21:25:00 2021 -0500

    Merge branch 'new_tests' of https://github.com/abeishekeeva/vowel into
new_tests

commit da9341fefec6c5001e86c9360ff2fab69d1cdb2a
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Mon Dec 20 21:24:56 2021 -0500

    first fail test modified

commit f88f9278768e5f94ed71805b6825f0e7867f068f
Author: Vikram Rajan <vikramrajan90@gmail.com>
Date:   Mon Dec 20 21:10:28 2021 -0500

    Updated tests, but globals still failing

commit f08e68350d776fe089e2c29bf14242919ba693a1
Author: root <am4958@columbia.edu>
Date:   Mon Dec 20 20:20:00 2021 -0500

    working but backwards

commit 1b2882c9e4375f37eebd144fb582014c07e051ca
Author: root <am4958@columbia.edu>
Date:   Mon Dec 20 19:33:15 2021 -0500

    working expect answer is opposite of what it should be

commit fb8d11e8db44fc73165c063f3981c2d4cade96e4
```

```
Merge: 1aeb1fd d74a5b3
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Mon Dec 20 19:28:40 2021 -0500

    merged variable decl & initialization into master

commit d74a5b310d75209fc5a88092970567bb93d46cbf
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Mon Dec 20 19:23:11 2021 -0500

    variable declaration & initialization in one line

commit c779a9dfdd43fc686347eeb46978c070a054e59d
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Mon Dec 20 14:14:04 2021 -0500

    string intersection wip

commit 65136a927910611f0cea6148a988014f90945824
Author: Coby <simlercoby@gmail.com>
Date:   Mon Dec 20 14:13:29 2021 -0500

    idenitified source of identiifer error as id

commit de6173fdee0fc86a9955c7d468fed107c42f5194
Author: Coby <simlercoby@gmail.com>
Date:   Sun Dec 19 23:17:38 2021 -0500

    updated semant.ml

commit 52fec757d401f3376c8ec6baa0601807e1d07e18
Author: Coby <simlercoby@gmail.com>
Date:   Sun Dec 19 21:51:35 2021 -0500

    declaration and initialization - todo: change symbol table to instead
rely on StringMaps

commit c84a4c1e36a1806091a00a5dd513a48728918a8a
Author: root <am4958@columbia.edu>
Date:   Sun Dec 19 18:47:32 2021 -0500

    slice working
```

```
commit 9a71ebc991ca26255d8b77942f3bd63c2b8929ee
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sun Dec 19 18:41:15 2021 -0500

    merge conflict in parse

commit 1aeb1fdbf7fce31318b7ebf6863d95b52330c5a5
Merge: 53374e0 7f1b9bd
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sun Dec 19 18:22:39 2021 -0500

    merged with incr & decr

commit 33909e8d5500c3cf90600c34a73c3352f0931c04
Author: root <am4958@columbia.edu>
Date:   Sun Dec 19 17:38:59 2021 -0500

    Slice op version not working trying something new

commit 53374e055eaef16f6b5125f7ce6fe87253238f6c
Author: Coby <simlercoby@gmail.com>
Date:   Sun Dec 19 01:21:05 2021 -0500

    updated decl and assign

commit f295108a8fb7e5d8566513e7c46b2735c431881d
Author: Coby <simlercoby@gmail.com>
Date:   Sun Dec 19 00:38:27 2021 -0500

    adding test-strconcat

commit 2120af2d28dbf3da80e2fff39d741ee43963c93c
Author: Coby <simlercoby@gmail.com>
Date:   Sun Dec 19 00:36:36 2021 -0500

    adding declaration and assignment in the same line functionality

commit aea8671073dbcbe619bdb8e01801428219336a8d
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 20:29:14 2021 -0500

    updated gitignore
```

```
commit 9322c35fae59c352d58b59aa23f2b671eac122ec
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 20:26:13 2021 -0500

    deleted unnecessary tests

commit 0562e38263d9d32469bb6013da0bf697c541eb78
Merge: 3f759c8 1c53a35
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 20:20:39 2021 -0500

    fixing conflict after array merge

commit 3f759c819dad1669618ae63f0da2149041eb89a1
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 20:13:13 2021 -0500

    cleaning up test files

commit 3b05f3399c13cfd0cab5dc799589f2d6894bbccc
Merge: c296486 7d2c6cd
Author: abeish <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 20:04:51 2021 -0500

    Merge pull request #1 from abeishekeeva/equalityop

    Equalityop

commit 1c53a357fe017d475f16ea9500979f1766d6570e
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 19:57:56 2021 -0500

    cleaned up the tests

commit f51cb37b29e004e43202e32b756dcd0120945e0e
Merge: 7929f11 ab27569
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 19:52:39 2021 -0500

    fixed conflicts

commit 7929f1105bb1ba9433a5312c47cf43eb77b0a633
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
```

```
Date:   Sat Dec 18 19:51:39 2021 -0500

    committing before pull

commit ab27569ad4cbf1d74507022788e3bf29bdf6e975
Author: Vikram Rajan <vikramrajan90@gmail.com>
Date:   Sat Dec 18 18:59:41 2021 -0500

    Arrays fully working

commit 9ad0dcb7e7ce36af5feb69eb2825d20190bd2764
Merge: f1196a0 71fd7b1
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 18:19:35 2021 -0500

    resolved conflicts

commit f1196a0bb81d99838975288f5d99d6177cc9cf57
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 18:18:22 2021 -0500

    fixing array access bug

commit 71fd7b18f0b271b36f52bac421b05a5941b4e5b9
Author: Vikram Rajan <vikramrajan90@gmail.com>
Date:   Sat Dec 18 18:16:32 2021 -0500

    Second try. Can declare arrays but having issues accessing

commit 5f98d253febc9755c0c48d861b58c383d03a6613
Author: root <am4958@columbia.edu>
Date:   Sat Dec 18 18:00:55 2021 -0500

    first draft of slicing

commit d16db4a4e24e0c829168d32d729028294cb2acab
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 16:29:17 2021 -0500

    first try on arrays

commit 62d0ac0f625e1cac06f81e74a6ac1dc8f4268d86
Author: root <am4958@columbia.edu>
```

```
Date:   Sat Dec 18 14:55:28 2021 -0500

    strlen worksgit add .!

commit 7f1b9bd0828fed9a411b111ba728da6cb5b637f7
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sat Dec 18 14:40:05 2021 -0500

    increment & decrement

commit 50d7b7549f0b06dde0b29715b1c6d8a37bac140f
Author: root <am4958@columbia.edu>
Date:   Sat Dec 18 14:04:28 2021 -0500

    weird error combining ints ans strings in one test case

commit 4553bf6714a2379a325741cd1dd23e78ebc3cf57
Author: root <am4958@columbia.edu>
Date:   Sat Dec 18 13:43:58 2021 -0500

    still parse error but looking into codegen

commit d60b08000373336202995e76502a8e959fd93d8f
Author: root <am4958@columbia.edu>
Date:   Sat Dec 18 12:16:38 2021 -0500

    first attempt at string len getting parse error

commit 7d2c6cd1f19485de80980fc1958dd5f9f1bf23ef
Author: root <am4958@columbia.edu>
Date:   Wed Dec 15 15:14:44 2021 -0500

    cleaned up files for wokring string equality and inequality

commit 5dbed82de7c1ec9de5a2cd12149a1f74333c56c7
Author: root <am4958@columbia.edu>
Date:   Wed Dec 15 15:11:45 2021 -0500

    string equality and inequality workgit add .!

commit 744deaa81ece449ce5339ec238d90502d4b60b65
Author: root <am4958@columbia.edu>
Date:   Wed Dec 15 14:56:26 2021 -0500
```

```
    string inequality worksgit add .git add .!

commit 426d6e65f2ddb6cbecc8ca05bb694e2e2a4e42c1
Author: root <am4958@columbia.edu>
Date:   Wed Dec 15 14:52:58 2021 -0500

    string inequality is opposite of what it should be and converted ops
back to ints

commit 84173077c30b8de2a9a28ffd1eda26d979579ce2
Author: root <am4958@columbia.edu>
Date:   Wed Dec 15 14:45:41 2021 -0500

    string not equal to compiling but saying diff. Equality parse error

commit a30566f80556af191d10232bfd1b62d6766ad937
Author: root <am4958@columbia.edu>
Date:   Tue Dec 14 21:33:13 2021 -0500

    progress but not working string equality:(

commit ce7af16864743aef7e962b0dd4143257c79d202b
Author: root <am4958@columbia.edu>
Date:   Tue Dec 14 14:09:05 2021 -0500

    still not working but closest version getting parse error

commit 656501d4a8c1e36ec92210da6947c93331fcebdd
Author: root <am4958@columbia.edu>
Date:   Tue Dec 14 13:37:59 2021 -0500

    updated string eqalop still not working tho

commit f99bfce28abccd2cff705c93367592dd7b6cba16
Author: root <am4958@columbia.edu>
Date:   Tue Dec 14 12:21:46 2021 -0500

    changes to equlity operator int type verison

commit 4ae415cfd667c9628aa8cadce5898028657c7545
Author: root <am4958@columbia.edu>
Date:   Sun Dec 5 16:02:58 2021 -0500
```

```
    fixing function pointers in strequality

commit f693c2211f55eab8154e127f28e239a2d0f081ab
Author: root <am4958@columbia.edu>
Date:   Sun Dec 5 15:48:58 2021 -0500

    work in progress on string equality

commit c296486bf35314de2d42679f77ba3847e6151c3d
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sun Dec 5 14:09:36 2021 -0500

    fixed merge conflicts & accidental deletions

commit 19767d277e2274f26e3c08dfbe20b322cbad4305
Merge: fa31fc1 d080368
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sun Dec 5 13:55:18 2021 -0500

    Merge remote-tracking branch 'origin/coby_branch'

commit fa31fc15e4e2781515391582 45f8838d28701b0e
Merge: bd90ab7 906d4cb
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sun Dec 5 13:54:47 2021 -0500

    merged first version of concat

commit d080368c9c18338d82f16c8f1c30b047ad7afe7c
Author: Coby <simlercoby@gmail.com>
Date:   Sun Dec 5 13:53:40 2021 -0500

    revised Makefile to include linking of vowel_func.c and codegen

commit bd90ab718b8cef389d4af3de693845886d973046
Merge: 5b88ca0 b4af4ac
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sun Dec 5 01:12:17 2021 -0500

    merged modulus operator into master

commit 5b88ca0e3babc8d33d3669a65efe4de8cbbefadf
```

```
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sun Dec 5 01:05:28 2021 -0500

    committing test output files

commit cb743e27636abb706b6ee1acf17491c47e20be01
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Sun Dec 5 00:56:22 2021 -0500

    renamed files for vowel

commit 906d4cb70f36a2d46344564c92926c821283c33a
Author: Coby <simlercoby@gmail.com>
Date:   Sun Dec 5 04:04:44 2021 +0000

    added string concatenation functionality with the '+' Binop

commit 6ac5d6345d39d41f9373b79e326c635fdee63afe
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Wed Dec 1 13:23:51 2021 -0500

    fixed test increment output file

commit da438b3f7fefa77b9fd3d1aec4c82b62a3fc34ad
Merge: 31c0989 1d9c070
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Wed Dec 1 13:21:21 2021 -0500

    fixed conflict after merging with aidai branch

commit 1d9c070afd21207ed7ae8361364f52dd3a865b41
Author: aidai beishekeeva <aidai.beishekeeva@gmail.com>
Date:   Wed Dec 1 13:17:57 2021 -0500

    merged increment & decrement

commit 784c9158a7873b46796688fdb6b97465bea6c73a
Merge: d498742 12f0bc8
Author: abeishekeeva <aidai.beishekeeva@gmail.com>
Date:   Wed Dec 1 13:04:11 2021 -0500

    resolved conflicts after merging decrement from lex
```

```
commit 12f0bc8fb23982ce22fac31b9f4a8ade2e9aed94
Author: am4958 <am4958@columbia.edu>
Date:   Wed Dec 1 13:00:18 2021 -0500

    decrement but parse error

commit d4987423ca18f022cf8ab5e687c77e15f4e6dfd2
Author: abeishekeeva <aidai.beishekeeva@gmail.com>
Date:   Wed Dec 1 12:41:20 2021 -0500

    increment operator for integers

commit b4af4acec2efe8efcebfeb286fa925da998d1544
Author: Vikram Rajan <vikramrajan90@gmail.com>
Date:   Wed Dec 1 12:05:18 2021 -0500

    Modulus function working for int and float

commit 65b7f118824c390d1945063ee4bf2d6300efc5ef
Author: am4958 <am4958@columbia.edu>
Date:   Wed Dec 1 11:13:35 2021 -0500

    first update of decrement

commit c4268a43d01fcbb480f8f82b4ca067ed28f2655c
Author: abeish <aidai.beishekeeva@gmail.com>
Date:   Sat Nov 13 21:30:07 2021 -0500

    Update README

commit 04fbd7f7fba6a2a1297efce22c5c57f5a61ca19f
Author: abeish <aidai.beishekeeva@gmail.com>
Date:   Sat Nov 13 11:09:36 2021 -0500

    Update README

commit f13ff644c6e31eb5939d9c54ceed8d7edaa86a3d
Author: root <simlercoby@gmail.com>
Date:   Sat Nov 13 00:52:03 2021 -0500

    a long night of bug bashing. This version now supports strings and
printing strings
```

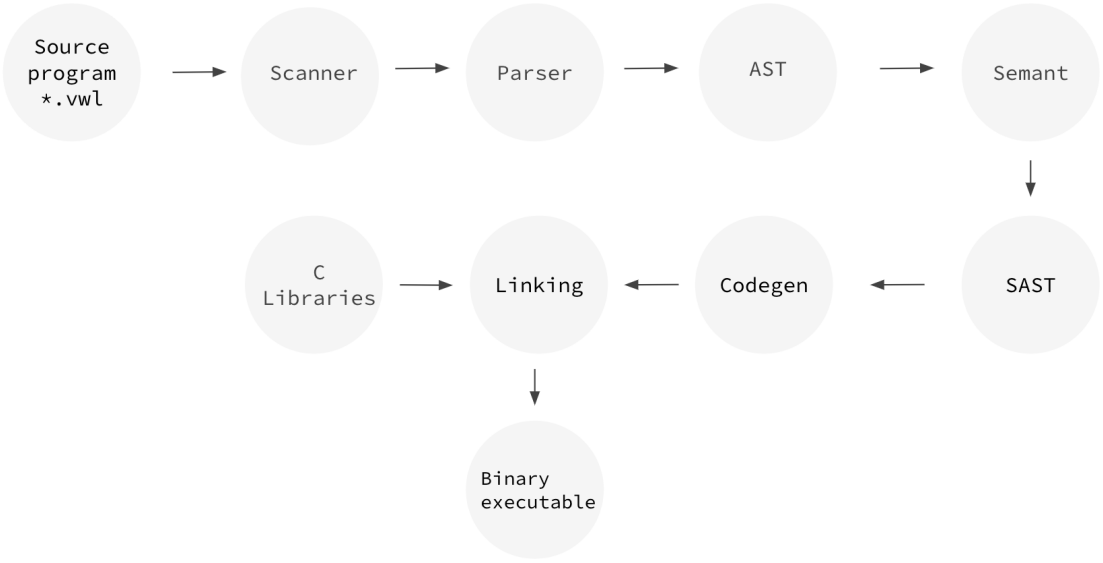
```
commit 31c0989e7c3aba0a832cececd65e00ab04e4a677
Author: Vikram Rajan <vikramrajan90@gmail.com>
Date:   Wed Nov 10 20:27:55 2021 -0500

    Added same ast.ml for testing purposes, still need to change to ours

commit d4032f9fd7962cafa0d55d83593c2fefe9ccf1f7
Author: aidai beishekeeva
<aidaibeishekeeva@dyn-160-39-244-161.dyn.columbia.edu>
Date:   Wed Nov 10 16:40:10 2021 -0500

    updating parser in accordance with microc example

commit 2ffc84adb52b511711f628bc0ecec1da6f7a108b
Merge: 6ec9626 91242d2
Author: aidai beishekeeva
<aidaibeishekeeva@dyn-160-39-244-135.dyn.columbia.edu>
Date:   Wed Nov 10 11:37:22 2021 -0500

    fixing conflict in parser, removing char

commit 6ec9626399b4106fa3beea1a2512c3d90d8fbae8
Author: aidai beishekeeva
<aidaibeishekeeva@dyn-160-39-244-135.dyn.columbia.edu>
Date:   Wed Nov 10 11:36:26 2021 -0500

    scanner. first version

commit 91242d2a28c5bdc49422c114c380ad569e7ad620
Author: am4958 <46966950+am4958@users.noreply.github.com>
Date:   Mon Nov 8 11:20:24 2021 -0500

    added statments error with for loop init

    because comparison and increment are not defined and need to be written

commit dc8e25967c92a1d8f153f4a4d8ae9648ce022652
Author: am4958 <46966950+am4958@users.noreply.github.com>
Date:   Mon Nov 8 11:09:41 2021 -0500

    no new no char

commit cdd1a8839ec1065a75a1d604671927858d77a60a
```

```
Author: aidai beishekeeva <aidaibeishekeeva@aidais-MBP.lan>
Date:   Sat Nov 6 14:38:56 2021 -0400

    conditional & loops

commit 7109349d27608ce8f4b29aadcbaef9c6b23f18be
Author: Vikram Rajan <vikramrajan90@gmail.com>
Date:   Sat Nov 6 14:05:30 2021 -0400

    Added slicing

commit de237bbf2a7bee74d39479fc3d88e515100fcfab
Author: am4958 <46966950+am4958@users.noreply.github.com>
Date:   Sat Nov 6 12:49:14 2021 -0400

    added structs

commit 6276fd13a47a5ad3eeb057516abe9c833093dd92
Author: am4958 <46966950+am4958@users.noreply.github.com>
Date:   Sat Nov 6 12:39:21 2021 -0400

    fixed some of the variable names from micro c to math our code

commit 3af2c06905d5b6ea3278e74d5c741a038ae8fc06
Merge: 7557d1e 3450734
Author: Coby <simlercoby@gmail.com>
Date:   Fri Nov 5 17:18:48 2021 -0400

    Added declaration to parser.mly from MicroC compiler

commit 7557d1e2b60e476d9c353ac95cf62a2e2e2100c4
Author: Coby <simlercoby@gmail.com>
Date:   Fri Nov 5 17:09:59 2021 -0400

    Added var and function declaration from the microC example.

commit 34507348131831f3e5372c1d90130a2a993f2745
Author: am4958 <46966950+am4958@users.noreply.github.com>
Date:   Thu Nov 4 16:17:13 2021 -0400

    Added chars - no errors

commit 9c2135ce5c3c143d4b088fe6d976cda9e5e5c3e4
```

```
Author: am4958 <46966950+am4958@users.noreply.github.com>
Date:   Thu Nov 4 16:08:11 2021 -0400

    Implemented Arrays-- no errors

commit 37f60179fba7606f8ebdb48a46686c5bc5597d47
Author: Coby <simlercoby@gmail.com>
Date:   Mon Nov 1 22:28:21 2021 -0400

    adding parser.output for visibility

commit bf04b4b3d8562ac68d42d6a51a0ec774a9fbf92e
Author: Coby <simlercoby@gmail.com>
Date:   Mon Nov 1 22:25:27 2021 -0400

    rebuilt from scratch. No more shift/reduce errors, but this is far from
being able too express everything in the language. This doesn't include
arrays, loops, semicolons, and more

commit 632a4c77f3968b09ad2ff85a0e9431d17adb0a89
Author: Coby <simlercoby@gmail.com>
Date:   Sat Oct 30 19:39:15 2021 -0400

    Added EQUAL NEQUAL; Added associativity for INCR DECR

commit f35a6ebc30b2442870fd2c59c58764f14705f853
Author: Vikram Rajan <vikramrajan90@gmail.com>
Date:   Fri Oct 29 16:52:12 2021 -0400

    Went from 42 shift/reduce to 11 errors, also commented out IF THEN

commit 7c30307e65db100ce0e6085a1c8487bb74559d0d
Author: aidai beishekeeva
<aidaibeishekeeva@dyn-160-39-244-100.dyn.columbia.edu>
Date:   Fri Oct 29 16:02:15 2021 -0400

    first version of parser
```

5 Architectural design

5.1. Block diagram



5.2. Scanner

File: scanner.mll (Appendix 8.1)

The scanner file takes in a Vowel source program (.vwl extension) and converts it into a stream of tokens for identifiers, keywords, operators, and literals. Comments and whitespaces are ignored at this stage, and there is support for both single line and multiline comments. The scanner is implemented in Ocamllex, and uses regular expressions to scan string literals, sequences of digits, escape characters, and so on. If any characters are found to be illegal—meaning not identified by the scanner—then the scanner throws an error. The tokens from the scanner are read in by the parser next.

Implemented by Vik, Coby, Lex, Aidai for the features each member was responsible for as described in section 4.5.1

5.3. Parser and AST

Files: vowelparse.mly, ast.ml (Appendix 8.2, 8.3)

The parser takes tokens generated by the lexer and derives an abstract syntax tree (ast) with the grammar created in vowelparse.mly. Specifically, the parser contains a context-free grammar that is composed of production rules and their precedence. The parser defines the grammar rules, beginning with a top level program non-terminal, which expands to declarations and in turn, all variables, functions, and statements. The expression non-terminal contains rules for recursive generation of operator statements. Finally, the parser, written in Ocamlyacc, defines the precedence and associativity of various operator tokens. If the syntax of the source program is incorrect, a Stdlib.parse_error will be thrown.

Ast defines a token syntax tree, including binary and unary operator types, a type for types, and expressions and statements.

Implemented by Vik, Coby, Lex, Aidai for the features each member was responsible for as described in section 4.5.1

5.4 Semantic Checking

File: semant.ml (Appendix 8.4)

The semantic checker converts the abstract syntax tree (AST) into a semantically checked syntax tree (SAST). This step ensures that the variables are declared and initialized to proper types and in a correct scope and that there are no duplicate variables and functions. In addition, the SAST validates that built-in functions are passed required types. If the semantic checking is successful, it will return a properly constructed SAST.

Implemented by Vik, Coby, Lex, Aidai for the features each member was responsible for as described in section 4.5.1

5.5 Code Generation

File: codegen.ml (Appendix 8.6)
The code generator takes in SAST and builds a LLVM code. We use ocaml's llvm library to map expressions to LLVM types. The LLVM module then gets compiled from the intermediate representation (IR) into native machine code. Along with binary operations for int, float and boolean, we added support for string binary operations. This was accomplished by using LLVM's pointercast that generates a pointer to instruction. Codegen also allocates memory for both our string and array types. It is also where we import built-in functions that we wrote with the help of C (described below).

Implemented by Vik, Coby, Lex, Aidai for the features each member was responsible for as described in section 4.5.1

5.6 C Libraries

Files: vowel_func.c

Codegen interfaces with vowel_func.c to support built-in functions supported by external c libraries. This includes the implementation of our complex string manipulation operators. Each of these built-in methods are detailed in section 3.2.

Implemented by Vik, Coby, Lex, Aidai for the features each member was responsible for as described in section 4.5.1

6 Test Plan

6.1 Example Vowel Programs

The following are two example programs written in the Vowel language that demonstrate the ease with which programmers can parse English language text.

This program returns the longest shared word between two inputs. The program uses Vowel's intersection operator, which returns an array of words that are contained in both String a and String b, ordered by their appearance in String A. The example program below also includes Vowel's built-in len() operator to determine the length of each word in the for loop below. Note the lack of main method and single line declaration and initialization requirement.

Vowel Source Program:

```
string getLongestSharedWord(string a, string b){
    string[] sharedWords = a & b;
    int longestWordLen = 0;
    string longestWord = "";
    int i = 0;
    for ( ; i<2; i=i+1){
        int currentlen = len(sharedWords[i]);
        if (currentlen > longestWordLen){
            longestWordLen = currentlen;
            longestWord =  sharedWords[i];
        }
    }
    return longestWord;
}
```

```
string x = getLongestSharedWord("abra cadabra baba","babra cadabra baba");
printstr(x);
```

LLVM Target Program:

```
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00", align 1
@fmt.2 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@string = private unnamed_addr constant [21 x i8] c"\22babra cadabra
baba\22\00", align 1
@string.3 = private unnamed_addr constant [20 x i8] c"\22abra cadabra
baba\22\00", align 1
@fmt.4 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.5 = private unnamed_addr constant [4 x i8] c"%g\0A\00", align 1
@fmt.6 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@string.7 = private unnamed_addr constant [3 x i8] c"\22\22\00", align 1

declare i8* @string_concat(i8*, i8*)
declare i32 @string_inequality(i8*, i8*)
declare i8** @string_intersection(i8*, i8*)
declare i8** @string_sub(i8*, i8*)
declare i8* @slice(i8*, i32, i32)
declare i32 @len(i8*)
declare i8* @string_mult(i8*, i32)
declare i32 @printf(i8*, ...)
declare i32 @printbig(i32)
define i32 @main() {
entry:
  %getLongestSharedWord_result = call i8* @getLongestSharedWord(i8*
getelementptr inbounds ([20 x i8], [20 x i8]* @string.3, i32 0, i32 0), i8*
getelementptr inbounds ([21 x i8], [21 x i8]* @string, i32 0, i32 0))
  %x = alloca i8*
  store i8* %getLongestSharedWord_result, i8** %x
  %x1 = load i8*, i8** %x
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt.2, i32 0, i32 0), i8* %x1)
  ret i32 0
}

define i8* @getLongestSharedWord(i8* %a, i8* %b) {
entry:
```

```
  %a1 = alloca i8*
  store i8* %a, i8** %a1
  %b2 = alloca i8*
  store i8* %b, i8** %b2
  %a3 = load i8*, i8** %a1
  %b4 = load i8*, i8** %b2
  %string_intersection = call i8** @string_intersection(i8* %a3, i8* %b4)
  %sharedWords = alloca i8**
  store i8** %string_intersection, i8*** %sharedWords
  %longestWordLen = alloca i32
  store i32 0, i32* %longestWordLen
  %longestWord = alloca i8*
  store i8* getelementptr inbounds ([3 x i8], [3 x i8]* @string.7, i32 0,
i32 0), i8** %longestWord
  %i = alloca i32
  store i32 0, i32* %i
  store i32 0, i32* %i
  br label %while

while:                                            ; preds = %merge, %entry
  %i17 = load i32, i32* %i
  %tmp18 = icmp slt i32 %i17, 2
  br i1 %tmp18, label %while_body, label %merge19

while_body:                                       ; preds = %while
  %i5 = load i32, i32* %i
  %accpos = add i32 %i5, 0
  %sharedWords6 = load i8**, i8*** %sharedWords
  %acceltptr = getelementptr i8*, i8** %sharedWords6, i32 %accpos
  %accelt = load i8*, i8** %acceltptr
  %len = call i32 @len(i8* %accelt)
  %currentlen = alloca i32
  store i32 %len, i32* %currentlen
  %currentlen7 = load i32, i32* %currentlen
  %longestWordLen8 = load i32, i32* %longestWordLen
  %tmp = icmp sgt i32 %currentlen7, %longestWordLen8
  br i1 %tmp, label %then, label %else

merge:                                            ; preds = %else, %then
  %i15 = load i32, i32* %i
  %tmp16 = add i32 %i15, 1
  store i32 %tmp16, i32* %i
```

```
  br label %while

then:                                             ; preds = %while_body
  %currentlen9 = load i32, i32* %currentlen
  store i32 %currentlen9, i32* %longestWordLen
  %i10 = load i32, i32* %i
  %accpos11 = add i32 %i10, 0
  %sharedWords12 = load i8**, i8*** %sharedWords
  %acceltptr13 = getelementptr i8*, i8** %sharedWords12, i32 %accpos11
  %accelt14 = load i8*, i8** %acceltptr13
  store i8* %accelt14, i8** %longestWord
  br label %merge

else:                                             ; preds = %while_body
  br label %merge

merge19:                                          ; preds = %while
  %longestWord20 = load i8*, i8** %longestWord
  ret i8* %longestWord20
}
```

The following is another example of a Vowel program. This program leverages Vowel's string subtraction operator to return an ordered list of those whiskey brands contained in the first string (order1) but not in the second string (order 2).

Vowel Source Program:

```
string order1 = "Your receipt contains: Jameson |
                 Glenlivet | Seagrams | Fireball |
                 Glenfiddich | Glenmorangie";
string order2 = "Your receipt contains: Glenmorangie |
                 Oban | Glenlivet | Glenfiddich |
                 Bowmore | Laphroaig";

string[] unique1 = order1 - order2;
int i = 0;
for ( ; i < 3 ; i=i+1 ){
```

```
    printstr(unique1[i]);
}
```

LLVM Target Program:

```
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00", align 1
@fmt.2 = private unnamed_addr constant [4 x i8] c"%s\0A\00", align 1
@string = private unnamed_addr constant [96 x i8] c"\22Your receipt
contains: Jameson | Glenlivet | Seagrams | Fireball | Glenfiddich |
Glenmorangie\22\00", align 1
@string.3 = private unnamed_addr constant [93 x i8] c"\22Your receipt
contains: Glenmorangie | Oban | Glenlivet | Glenfiddich | Bowmore |
Laphroaig\22\00", align 1

declare i8* @string_concat(i8*, i8*)

declare i32 @string_inequality(i8*, i8*)

declare i8** @string_intersection(i8*, i8*)

declare i8** @string_sub(i8*, i8*)

declare i8* @slice(i8*, i32, i32)

declare i32 @len(i8*)

declare i8* @string_mult(i8*, i32)

declare i32 @printf(i8*, ...)

declare i32 @printbig(i32)

define i32 @main() {
entry:
  %order1 = alloca i8*
  store i8* getelementptr inbounds ([96 x i8], [96 x i8]* @string, i32 0,
i32 0), i8** %order1
  %order2 = alloca i8*
  store i8* getelementptr inbounds ([93 x i8], [93 x i8]* @string.3, i32 0,
i32 0), i8** %order2
  %order11 = load i8*, i8** %order1
```

```
  %order22 = load i8*, i8** %order2
  %string_sub = call i8** @string_sub(i8* %order11, i8* %order22)
  %unique1 = alloca i8**
  store i8** %string_sub, i8*** %unique1
  %i = alloca i32
  store i32 0, i32* %i
  store i32 0, i32* %i
  br label %while

while:                                            ; preds = %while_body,
%entry
  %i6 = load i32, i32* %i
  %tmp7 = icmp slt i32 %i6, 3
  br i1 %tmp7, label %while_body, label %merge

while_body:                                       ; preds = %while
  %i3 = load i32, i32* %i
  %accpos = add i32 %i3, 0
  %unique14 = load i8**, i8*** %unique1
  %acceltptr = getelementptr i8*, i8** %unique14, i32 %accpos
  %accelt = load i8*, i8** %acceltptr
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x
i8], [4 x i8]* @fmt.2, i32 0, i32 0), i8* %accelt)
  %i5 = load i32, i32* %i
  %tmp = add i32 %i5, 1
  store i32 %tmp, i32* %i
  br label %while

merge:                                            ; preds = %while
  ret i32 0
}
```

6.2 Translator Test Suite

Tests were written throughout the development process by the individual directly responsible for the respective feature. For example, all tests for string intersection features were originally on the string intersection branch before being merged into master. This approach has advantages and disadvantages. On the one hand, this ensures that the specifics of every feature was tested and that cases did not fall through the cracks. On the other hand, this led to additional time being expended upon merging tests from different branches. The branch on which the main method requirement was removed, for instance, was merged after the branches for string incrementation were merged. As a result, we had to go back and remove the main methods from all previous tests.

6.3 Testing Automation

To programmatically test our language, we relied on the logic from the testall.sh file from MicroC combined with our own test suite. This file iterates over all tests in the tests directory and runs the vowel executable (vowel.native) on the tests, returning OK for a successful test and outputting errors and differences where appropriate. See the appendix for this file.

6.4 Test Roles

With each individual contributor responsible for generating tests for the features they created, the work for creating tests was distributed. With that said, Vik, as the tester point person, spearheaded the testing process after all branches were merged into master.

7 Lessons Learned

7.1 Coby Simler

The creation of a programming language is an exercise in reducing complexity into its simpler, constituent parts. From this journey, I learned about my own learning process: I learn most effectively when I have a high-level, basic understanding of the task of each constituent part. Only then do I find it productive to dive into the specific lines of code of each file. For example, in the first several weeks, I attempted to get my bearing by studying each file line-by-line. This “bottom-up” approach left me confused. I was better able to understand the material when I stepped back and first understood the broader objective of each file/part.

Building Vowel, too, required asking my teammates plenty of “dumb” questions. Why does this line of Ocaml need to be before this block of code? How does this regular expression match strings? And so on. Developing a shared codebase with this level of complexity helped me re-learn the importance of being willing to ask these questions openly and early.

Finally, the process bolstered my confidence that in the future, I’ll be able to approach complex engineering tasks and, with a little bit of effort, contribute productively.

7.2 Lex Mengenhauser

The start of this project is very intimidating and can feel overwhelming. I think the biggest help to getting the first few parts of the projects implemented (especially hello world) is programming as a team a few times. Doing this made it feel less intimidating and we were all able to fill in gaps in knowledge for one another by working through questions as a group. As the project went on and I got more familiar with the code structure it became much easier and quicker to implement new built-in functions and operators. It can be really frustrating to get stuck on the implementation of a single feature which we solved by frequently exchanging assignments after getting to a point of frustration. I really liked that we did this as it helped to get familiar with every file and aspect

of the project and would often help solving the original problem too. I think this is a great thing for future teams to try as well as setting internal deadlines for features to keep on track!

7.3 Vikram Rajan

This project is much bigger than it seems at the start, and it already seemed pretty big at the start! To successfully tackle most of what you set out to do requires really good planning. Setting milestones and defining exactly what everyone is currently working on is huge. This prevents procrastination and the stress of having to do a semester's worth of work in the last few weeks. It also forces everyone in the group to learn, which unfortunately can only be done after going through painful hours of debugging and screaming.
As we got further into the project, we found that pair programming was one of the most effective techniques to make significant progress. Personally, I found some parts of this project demotivating while working alone, as constantly running into dead ends can hurt anyone's spirits. Just the advantage of having another pair of eyes and another mind to catch the small mistakes that may go unnoticed to us is huge. Another positive aspect of pair programming is that more people are caught up and understand what more of the code does.

7.4 Aidai Beishekeeva

Implementing our own language was challenging every step of the way. However, with each little goal that we accomplished as a team, we felt more confident about our own knowledge and understanding of compilers.
As professor Edwards said in the beginning of the semester, this course actually taught a lot more about the importance of teamwork and empathy towards my teammates. Of course, the goal was to deepen the understanding of compilers but I realize now that it was only successful because we did it in a group.
As a project manager, I learned that I need to do a better job of distributing the tasks more evenly in a timeline and pairing group members to complete more complicated features. I came to the conclusion that it was productive and efficient in terms of communication and accomplishing milestones.

8 Appendix

8.1 scanner.mll

```
(* MicroC template is used for this scanner *)

{ open Vowelparse }
```

```
let digit = ['0' - '9']
let digits = digit+
let float = (digits) '.' (digits)
let alphabet = ['a'-'z' 'A'-'Z']
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
```

```
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let str_literal = '"' (ascii | escape)* '"'


rule token = parse
 [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'      { MODULUS }
| '='      { ASSIGN }
| "+="     { INCREMENT }
| "-="     { DECREMENT }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| '&'      { INTERSEC }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "string" { STRING }
| "bool"   { BOOL }
| "float"  { FLOAT }
| "[]"     { ARRAY }
| "void"   { VOID }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
```

```
 | digits as lxm { LITERAL(int_of_string lxm) }
 | digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
 | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*     as lxm { ID(lxm) }
 | str_literal as lxm { SLIT(lxm) }
 | eof { EOF }
 | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

8.2 vowelparse.mly

```
/* Vowel parser. MicroC template is used. */

 %{ open Ast %}

 %token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA PLUS MINUS
TIMES
 DIVIDE INTERSEC
 %token MODULUS ASSIGN
 %token INCREMENT DECREMENT
 %token NOT EQ NEQ LT LEQ GT GEQ AND OR
 %token RETURN IF ELSE FOR WHILE INT BOOL FLOAT VOID STRING
 %token ARRAY
 %token <int> LITERAL
 %token <bool> BLIT
 %token <string> ID FLIT SLIT
 %token EOF

 %start program
 %type <Ast.program> program

 %nonassoc NOELSE
 %nonassoc ELSE
 %right INCREMENT DECREMENT ASSIGN
 %left INTERSEC
 %left OR
 %left AND
 %left EQ NEQ
 %left LT GT LEQ GEQ
 %left PLUS MINUS
 %left TIMES DIVIDE MODULUS
 %right NOT
```

```
%%

program:
 decls EOF { $1 }

decls:
  /* nothing */ { ([], [], [])         }
| decls stmt { let (stmt, vdecl, fdecl) = $1 in ($2::stmt, vdecl, fdecl) }
| decls vdecl { let (stmt, vdecl, fdecl) = $1 in (stmt, $2::vdecl, fdecl) }
| decls fdecl { let (stmt, vdecl, fdecl) = $1 in (stmt, vdecl, $2::fdecl) }

fdecl:
  typ ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
    { { typ = $1;
  fname = $2;
  formals = List.rev $4;
  locals = [];
  body = List.rev $7 } }

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

formal_list:
   typ ID                   { [($1,$2)]     }
 | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ:
   INT    { Int   }
 | BOOL   { Bool  }
 | FLOAT  { Float }
 | VOID   { Void  }
 | STRING { String }
 | typ ARRAY { Arr($1, 0) }


vdecl:
   typ ID SEMI { ($1, $2) }


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


stmt:
```

```
    expr SEMI                                  { Expr $1                }
  | RETURN expr_opt SEMI                       { Return $2              }
  | LBRACE stmt_list RBRACE                    { Block(List.rev $2)     }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt     { If($3, $5, $7)         }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                             { For($3, $5, $7, $9)    }
  | WHILE LPAREN expr RPAREN stmt            { While($3, $5)          }

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

/* assignment_op:
 ASSIGN       { Assign }
| INCREMENT    { Incr }
| DECREMENT    { Decr } */

expr:
    LITERAL          { Literal($1)            }
  | FLIT             { Fliteral($1)           }
  | SLIT             { STRliteral($1)         }
  | BLIT             { BoolLit($1)            }
  | ID               { Id($1)                 }
  | 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 MODULUS expr{ Binop($1, Mod,   $3)   }
  | 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)   }
  | expr INTERSEC expr {Binop($1, Intersec, $3)}
  | MINUS expr %prec NOT { Unop(Neg, $2)      }
  | NOT expr         { Unop(Not, $2)          }
  | ID LPAREN args_opt RPAREN { Call($1, $3)  }
  | LPAREN expr RPAREN { $2                   }
  | ID ASSIGN expr    { Assign($1, $3)        }
  | ID INCREMENT expr { Increment($1, $3)    }
  | ID DECREMENT expr { Decrement($1, $3)     }
 /* Arrays */
```

```
  | ID LBRACKET expr RBRACKET { ArrayAccess($1, $3) }
  | LBRACKET args_list RBRACKET { ArrayLit($2) }
  | ID LBRACKET expr RBRACKET ASSIGN expr { ArrAssign($1, $3, $6) }
 /* VARIABLE DECLARATION */
  | typ ID ASSIGN expr { DeclAssn($1, $2, $4) }


args_opt:
   /* nothing */ { [] }
  | args_list  { List.rev $1 }

args_list:
   expr                    { [$1] }
  | args_list COMMA expr { $3 :: $1 }
```

8.3 ast.ml

```
(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater
| Geq |
          And | Or | Intersec

type uop = Neg | Not

type typ = Int | Bool | Float | Void | String | Arr of (typ * int)

type bind = typ * string

type expr =
   Literal of int
  | Fliteral of string
  | STRliteral of string
  | BoolLit of bool
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | DeclAssn of typ * string * expr
  | Increment of string * expr
  | Decrement of string * expr
  | Call of string * expr list
  | ArrayAccess of string * expr
  | ArrayLit of expr list
  | ArrAssign of string * expr * expr
```

```
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

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

type program = stmt list * bind list * func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"
  | Intersec -> "&"

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

let rec string_of_typ = function
  Int -> "int"
| String -> "string"
| Bool -> "bool"
```

```
  | Float -> "float"
  | Void -> "void"
  (* | Arr(t, _) -> string_of_typ t ^ "[]" *)
  | Arr(t, _) -> string_of_typ t ^ "[]"

  let rec string_of_expr = function
     Literal(l) -> string_of_int l
   | Fliteral(l) -> l
   | STRliteral(s) -> s
   | BoolLit(true) -> "true"
   | BoolLit(false) -> "false"
   | Id(s) -> s
   | Binop(e1, o, e2) ->
       string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
   | Unop(o, e) -> string_of_uop o ^ string_of_expr e
   | Assign(v, e) -> v ^ " = " ^ string_of_expr e
   | DeclAssn(t, s, e) -> string_of_typ t ^ " " ^ s ^ " = " ^ string_of_expr e
   | Increment(v, e) -> v ^ "+= " ^ string_of_expr e
   | Decrement(v, e) -> v ^ "-="  ^ string_of_expr e
   | Call(f, el) ->
       f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
   | ArrayAccess (s, e) ->  s ^ "[" ^ string_of_expr e ^ "]"
   | ArrayLit(e) -> "[" ^ String.concat "," (List.map string_of_expr (List.rev
e)) ^ "]"
   | ArrAssign(s, e1, e2) -> s ^ "[" ^ string_of_expr e1 ^ "] = " ^
string_of_expr  e2
   | Noexpr -> ""

  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
   | For(e1, e2, e3, s) ->
       "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
       string_of_expr e3  ^ ") " ^ string_of_stmt s
   | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

  let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

  let string_of_fdecl fdecl =
   string_of_typ fdecl.typ ^ " " ^
```

```
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

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

8.4 semant.ml

```
(* Semantic checking for the Vowel compiler *)

open Ast
open Sast
module StringMap = Map.Make(String);;

 module HashtblString =
 struct
  type t = string
  let equal = ( = )
  let hash = Hashtbl.hash
 end;;

 module StringHash = Hashtbl.Make(HashtblString);;

(* Semantic checking of the AST. Returns an SAST if successful,
  throws an exception if something is wrong.

  Check each global variable, then check each function *)

let check (statements, globals, functions) =

 (* Verify a list of bindings has no void types or duplicate names *)
let check_binds (kind : string) (binds : bind list) =
  List.iter (function
(Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
    | _ -> ()) binds;
  let rec dups = function
      [] -> ()
    | ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
  raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
```

```
    | _ :: t -> dups t
  in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
in

let check_bind tbl (t, n) =
  if t = Void then raise (Failure ("illegal void " ^ n)) else
  if (StringHash.mem tbl n) then raise (Failure ("duplicate " ^ n)) else
    StringHash.add tbl n t; tbl
in

(**** Check global variables ****)

check_binds "global" globals;

(**** Check functions ****)

(* Collect function declarations for built-in functions: no bodies *)
let built_in_decls =
  let add_bind map (name, ty) = StringMap.add name {
    typ = Void;
    fname = name;
    formals = [(ty, "x")];
    locals = []; body = [] } map
  in List.fold_left add_bind StringMap.empty [ ("print", Int);
                               ("printb", Bool);
                               ("printf", Float);
                               ("printbig", Int);
                               ("printstr", String)]
in
let built_in_decls =
    StringMap.add "string_sub" {
    typ = Arr(String,99);
    fname = "string_sub";
    formals = [(String, "str"); (String, "str2")];
    locals = [];
    body = [] } built_in_decls
in

(* declare main function with all statements *)
let main_decl =
  {
    typ = Int;
    fname = "main";
    formals = [];
    locals = [];
    body = List.rev statements }
```

```
  in
let functions = main_decl :: functions in

let built_in_decls =
  StringMap.add "slice" {
  typ = String;
  fname = "slice";
  formals = [(String, "str"); (Int, "from"); (Int, "to")];
  locals = [];
  body = [] } built_in_decls
in

let built_in_decls =
  StringMap.add "string_mult" {
  typ = String;
  fname = "string_mult";
  formals = [(String, "str"); (Int, "a")];
  locals = [];
  body = [] } built_in_decls
in

let built_in_decls =
  StringMap.add "len"{
  typ = Int;
  fname = "len";
  formals = [(String, "str")];
  locals = [];
  body = []} built_in_decls
in

(* Add function name to symbol table *)
let add_func map fd =
  let built_in_err = "function " ^ fd.fname ^ " may not be defined"
  and dup_err = "duplicate function " ^ fd.fname
  and make_err er = raise (Failure er)
  and n = fd.fname (* Name of the function *)
  in match fd with (* No duplicate functions or redefinitions of built-ins *)
       _ when StringMap.mem n built_in_decls -> make_err built_in_err
     | _ when StringMap.mem n map -> make_err dup_err
     | _ ->  StringMap.add n fd map
in

(* Collect all function names into one symbol table *)
let function_decls = List.fold_left add_func built_in_decls functions
in
 (* Return a function from our symbol table *)
```

```
 let find_func s =
   try StringMap.find s function_decls
   with Not_found -> raise (Failure ("unrecognized function " ^ s))
 in

 let rec check_arr (arrtyp, lv) =
   (match arrtyp with
     Arr(ty,_) ->  check_arr (ty, lv+1)
   | _ -> (arrtyp, lv))

   in

 let _ = find_func "main" in (* Ensure "main" is defined *)

 let check_function func =
   (* Make sure no formals or locals are void or duplicates *)
   check_binds "formal" func.formals;
   check_binds "local" func.locals;
   let tbl = StringHash.create 10 in
   let formal_tbl = StringHash.create 5 in

   (* Raise an exception if the given rvalue type cannot be assigned to
      the given lvalue type *)
   let check_assign lvaluet rvaluet err =
       if lvaluet = rvaluet then lvaluet else
         (match lvaluet with
           (* Object(_) -> if rvaluet = Null then lvaluet else raise (Failure
err) *)
         | Arr _ -> (match rvaluet with
                         Arr _ -> let r_arr = check_arr (rvaluet, 0) in
                                  let l_arr = check_arr (lvaluet, 0) in
                                    if r_arr = l_arr then rvaluet else raise
(Failure err)
                       | _ -> raise (Failure err))
         | _ -> raise (Failure err))
   in

   (* Build local symbol table of variables for this function *)
   let _ = List.fold_left check_bind
     formal_tbl (func.formals @ func.locals )
   in

   (* Return a variable from our local symbol table *)
   let type_of_identifier s =
     try StringHash.find tbl s
     with Not_found ->
```

```
      try StringHash.find formal_tbl s
      with Not_found -> raise (Failure ("undeclared identifier " ^ s))
  in

  (* check if of array type, return element type *)
  let is_arr_ty (v, ty) = match ty with
      Arr(t,_) ->
        if t = Void then raise (Failure ("void type array " ^ v ^ " is not
allowed"))
        else t
    | _ -> raise (Failure ("cannot access an element in variable " ^ v ^ " of
type " ^ string_of_typ ty))
  in

  (* Return a semantically-checked expression, i.e., with a type *)
  let rec expr = function
      Literal  l -> (Int, SLiteral l)
    | Fliteral l -> (Float, SFliteral l)
    | BoolLit l  -> (Bool, SBoolLit l)
    | STRliteral l -> (String, SSTRliteral l)
    | Noexpr     -> (Void, SNoexpr)
    | Id s       -> (type_of_identifier s, SId s)

    | Assign(var, e) as ex ->
        let lt = type_of_identifier var
        and (rt, e') = expr e in
        let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
          string_of_typ rt ^ " in " ^ string_of_expr ex
        in let _ = (match lt with
          Arr _ -> StringHash.replace tbl var lt
          | _ -> ignore 1)
        in (check_assign lt rt err, SAssign(var, (rt, e')))


    | DeclAssn(ty, var, e) as declassn ->
        (* raise (Failure ("I'm Being Called From DeclAssn"));*)
        ignore (check_bind tbl (ty, var));
        (* check_bind tbl (ty, var);    *)
        let (rt, e') = expr e in
        let _err = "illegal assignment " ^ string_of_typ ty ^ " = " ^
            string_of_typ rt ^ " in " ^ string_of_expr declassn

        (*  update array size *)
        in let _ = (match ty with
          Arr _ -> StringHash.replace tbl var ty
        | _ -> ignore 1)
```

```
      in (ty, SDeclAssn(ty, var, (ty, e')))

  | Increment(var, e) as ex ->
      let l = Binop(Id(var), Add, e) in
      let lt = type_of_identifier var
      and (rt, e') = expr l in
      let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
        string_of_typ rt ^ " in " ^ string_of_expr ex
      in (check_assign lt rt err, SAssign(var, (rt, e')))

  | Decrement(var, e) as ex ->
      let l = Binop(Id(var), Sub, e) in
      let lt = type_of_identifier var
      and (rt, e') = expr l in
      let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
        string_of_typ rt ^ " in " ^ string_of_expr ex
      in (check_assign lt rt err, SAssign(var, (rt, e')))


  | Unop(op, e) as ex ->
      let (t, e') = expr e in
      let ty = match op with
        Neg when t = Int || t = Float -> t
      | Not when t = Bool -> Bool
      | _ -> raise (Failure ("illegal unary operator " ^
                             string_of_uop op ^ string_of_typ t ^
                             " in " ^ string_of_expr ex))
      in (ty, SUnop(op, (t, e')))
  | Binop(e1, op, e2) as e ->
      let (t1, e1') = expr e1
      and (t2, e2') = expr e2 in
      (* All binary operators require operands of the same type *)
      let same = t1 = t2 in
      (* Determine expression type based on operator and operand types *)
      let ty = match op with
        Add | Sub | Mult | Div | Mod when same && t1 = Int   -> Int
      | Add | Sub | Mult | Div | Mod when same && t1 = Float -> Float
      | Add  when same && t1 = String -> String
      | Sub when same && t1 = String -> Arr(String,99)
      | Equal | Neq            when same               -> Bool
      | Mult when t1 = String -> String
      | Intersec when same && t1 = String -> Arr(String, 99)
      | Less | Leq | Greater | Geq
                when same && (t1 = Int || t1 = Float) -> Bool
        | And | Or when same && t1 = Bool -> Bool
        | _ -> raise (Failure ("illegal binary operator " ^
```

```
                    string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                    string_of_typ t2 ^ " in " ^ string_of_expr e))
        in (ty, SBinop((t1, e1'), op, (t2, e2')))

    | Call(fname, args) as call ->
        let fd = find_func fname in
        let param_length = List.length fd.formals in
        if List.length args != param_length then
          raise (Failure ("expecting " ^ string_of_int param_length ^
                          " arguments in " ^ string_of_expr call))
        else let check_call (ft, n) e =
          let (et, e') = expr e
          in let err = "illegal argument found " ^ string_of_typ et ^
            " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
          in let _ = (check_assign ft et err, e')
          in let _ = (match et with
            Arr _ -> StringHash.replace formal_tbl n et
            | _ -> ignore 1) in (et, e')
        in let args' = List.map2 check_call fd.formals args
        in (fd.typ, SCall(fname, args'))


    | ArrayLit(el) as arraylit -> (* check if types of expr are consistent *)
        let ty_inconsistent_err = "inconsistent types in array " ^
string_of_expr arraylit in
        let fst_e = List.hd el in
        let (fst_ty, _) = expr fst_e in
        let (arr_ty_len, arr_ty_e) = List.fold_left (fun (t, l) e ->
            let (et, e') = expr e in
            let is_arr = (match et with
                Arr _ -> true
                | _ -> false) in if ((et = fst_ty) || is_arr)
                              then (t+1, (et, e')::l)
                              else (t, (et, e')::l)) (0,[]) el
        in if arr_ty_len != List.length el
          then raise (Failure ty_inconsistent_err)
          (* determine arr type *)
          else let arr_ty = Arr(fst_ty, arr_ty_len)
          in (arr_ty, SArrayLit(arr_ty_e))


    | ArrayAccess(v, e) as arrayacess ->(* check if type of e is an int *)
        let (t, e') = expr e in
        if t != Int then raise
        (Failure (string_of_expr e ^ " is not of int -> type in " ^
string_of_expr arrayacess)) else
```

```
          (* check if variable is array type *)
        let v_ty = type_of_identifier v in
        let e_ty = is_arr_ty (v, v_ty)
      in (e_ty, SArrayAccess(v, (t, e')))
    | ArrAssign(v, e1, e2) as arrassign ->(* check if type of e is an int *)
        let (t, e1') = expr e1 in
        if t != Int then raise (Failure (string_of_expr e1 ^ " is not of int
type → in " ^ string_of_expr arrassign))
        else (* check if variable is array type *)
        let v_ty = type_of_identifier v in
        let e_ty = is_arr_ty (v, v_ty) in
        let (rt, e2') = expr e2 in
        (e_ty, SArrAssign(v, (t,e1'), (rt,e2')))

  in

  let check_bool_expr e =
    let (t', e') = expr e
    and err = "expected Boolean expression in " ^ string_of_expr e
    in if t' != Bool then raise (Failure err) else (t', e')
  in

  (* Return a semantically-checked statement i.e. containing sexprs *)
  let rec check_stmt = function
      Expr e -> SExpr (expr e)
    | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
    | For(e1, e2, e3, st) ->
      SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
    | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
    | Return e -> let (t, e') = expr e in
      if t = func.typ then SReturn (t, e')
      else raise (
  Failure ("return gives " ^ string_of_typ t ^ " expected " ^
     string_of_typ func.typ ^ " in " ^ string_of_expr e))

    (* A block is correct if each statement is correct and nothing
       follows any Return statement.  Nested blocks are flattened. *)
    | Block sl ->
        let rec check_stmt_list = function
            [Return _ as s] -> [check_stmt s]
          | Return _ :: _   -> raise (Failure "nothing may follow a return")
          | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten blocks *)
          | s :: ss         -> let c = check_stmt s in
                               c :: check_stmt_list ss
                               (* check_stmt s :: check_stmt_list ss *)
          | []              -> []
```

```
        in SBlock(check_stmt_list sl)

  in (* body of check_function *)
  { styp = func.typ;
    sfname = func.fname;
    sformals = func.formals;
    slocals  = func.locals;
    sbody = match check_stmt (Block func.body) with
SBlock(sl) -> sl
    | _ -> raise (Failure ("internal error: block didn't become a block?"))
  }
in (globals, List.map check_function functions)
```

8.5 sast.ml

```
 (* Semantically-checked Abstract Syntax Tree and functions for printing it
*)

 open Ast

 type sexpr = typ * sx
 and sx =
    SLiteral of int
  | SFliteral of string
  | SSTRliteral of string
  | SBoolLit of bool
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of string * sexpr
  | SDeclAssn of typ * string * sexpr
  | SArrayAccess of string * sexpr
  | SArrayLit of sexpr list
  | SArrAssign of string * sexpr * sexpr
  | SCall of string * sexpr list
  | SNoexpr

 type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt

 type sfunc_decl = {
    styp : typ;
```

```
    sfname : string;
    sformals : bind list;
    slocals : bind list;
    sbody : sstmt list;
  }

 type sprogram = sstmt list * bind list * sfunc_decl list

 (* Pretty-printing functions *)

 let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    SLiteral(l) -> string_of_int l
  | SSTRliteral(l) -> l
  | SBoolLit(true) -> "true"
  | SBoolLit(false) -> "false"
  | SFliteral(l) -> l
  | SId(s) -> s
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
  | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
  | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
  (*\ | SIncr(v, e) -> v ^ " += " ^ string_of_sexpr e
  | SDecr (v, e) -> v ^ " -= " ^ string_of_sexpr e  *)
  | SCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
  | SArrayAccess(s, e) -> s ^ "[" ^ string_of_sexpr e ^ "]"
  | SArrayLit(e) -> "[" ^ String.concat "," (List.map string_of_sexpr
(List.rev e )) ^ "]"
  | SArrAssign(s, e1, e2) -> s ^ "[" ^ string_of_sexpr e1 ^ "] = " ^
string_of_sexpr e2
  | SDeclAssn(t, s, e) -> string_of_typ t ^ " " ^ s ^ " = " ^ string_of_sexpr
e
  | SNoexpr -> ""
  ) ^ ")"

 let rec string_of_sstmt = function
    SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s, SBlock([])) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
```

```
      "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
      string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s

let string_of_sfdecl fdecl =
 string_of_typ fdecl.styp ^ " " ^
 fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals) ^
 ")\n{\n" ^
 String.concat "" (List.map string_of_vdecl fdecl.slocals) ^
 String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
 "}\n"

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

8.6 codegen.ml

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR
LLVM tutorial: Make sure to read the OCaml version of the tutorial
http://llvm.org/docs/tutorial/index.html
Detailed documentation on the OCaml LLVM library:
http://llvm.moe/
http://llvm.moe/ocaml/ *)

module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String);;

module HashtblString =
  struct
   type t = string
   let equal = ( = )
   let hash = Hashtbl.hash
  end;;

module StringHash = Hashtbl.Make(HashtblString);;

(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
 let context    = L.global_context () in
  (* Create the LLVM compilation module into which
```

```
      we will generate code *)
   let the_module = L.create_module context "MicroC" in

   (* Get types from the context *)
   let i32_t      = L.i32_type    context
   and i8_t       = L.i8_type     context
   and str_t      = L.pointer_type (L.i8_type context)
   and i1_t       = L.i1_type     context
   and float_t    = L.double_type context
   and void_t     = L.void_type   context in

   (* Return the LLVM type for a MicroC type *)
   let rec ltype_of_typ = function
       A.Int   -> i32_t
     | A.Bool  -> i1_t
     | A.Float -> float_t
     | A.Void  -> void_t
     | A.String -> str_t
     | A.Arr(ty,_) -> L.pointer_type (ltype_of_typ ty)
   in


   let string_concat_t : L.lltype =
     L.function_type str_t [| str_t; str_t |] in
   let string_concat_f : L.llvalue =
     L.declare_function "string_concat" string_concat_t the_module in
    let string_inequality_t : L.lltype =
     L.function_type i32_t [| str_t; str_t |] in
   let string_inequality_f : L.llvalue =
     L.declare_function "string_inequality" string_inequality_t the_module in

   let string_intersection_t : L.lltype =
       L.function_type (L.pointer_type str_t) [| str_t; str_t |] in
   let string_intersection_f : L.llvalue =
       L.declare_function "string_intersection" string_intersection_t
the_module in
   let string_sub_t : L.lltype =
     L.function_type (L.pointer_type str_t) [| str_t; str_t |] in
   let string_sub_f : L.llvalue =
     L.declare_function "string_sub" string_sub_t the_module in

  let slice_t : L.lltype =
     L.function_type str_t [| str_t; i32_t; i32_t |] in
   let slice_f : L.llvalue =
     L.declare_function "slice" slice_t the_module in
```

```
   let len_t : L.lltype =
     L.function_type i32_t [| str_t |] in
   let len_f : L.llvalue =
     L.declare_function "len" len_t the_module in
      let string_mult_t : L.lltype =
       L.function_type str_t [| str_t; i32_t |] in
     let string_mult_f : L.llvalue =
       L.declare_function "string_mult" string_mult_t the_module in

   let printf_t : L.lltype =
       L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
   let printf_func : L.llvalue =
       L.declare_function "printf" printf_t the_module in
   let printbig_t : L.lltype =
       L.function_type i32_t [| i32_t |] in
   let printbig_func : L.llvalue =
       L.declare_function "printbig" printbig_t the_module in

   (* Define each function (arguments and return type) so we can
      call it even before we've created its body *)
   let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
     let function_decl m fdecl =
       let name = fdecl.sfname
       and formal_types =
   Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
       in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types
in
       StringMap.add name (L.define_function name ftype the_module, fdecl) m
in
     List.fold_left function_decl StringMap.empty functions in

    (* Fill in the body of the given function *)
   let build_function_body fdecl =
     let (the_function, _) = StringMap.find fdecl.sfname function_decls in
     let builder = L.builder_at_end context (L.entry_block the_function) in

     let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
     and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
     and string_format_str = L.build_global_stringptr "%s\n" "fmt" builder in


     (* Construct a hash table for function formals and locals
        add all the formals first *)
     let tbl = StringHash.create 10 in
     let formal_tbl = StringHash.create 5 in
       let add_formal tbl (t, n) p =
```

```
        L.set_value_name n p;
        let local = L.build_alloca (ltype_of_typ t) n builder in
            ignore (L.build_store p local builder);
        StringHash.add tbl n local; tbl in
      let _ = List.fold_left2 add_formal formal_tbl fdecl.sformals
        (Array.to_list (L.params the_function)) in

    (* Return the value for a variable or formal argument.
       Check local names first, then global names *)
    let lookup n = try StringHash.find tbl n(*StringMap.find tbl n n
local_vars*)
                   with Not_found -> try
                   StringHash.find formal_tbl n(*StringMap.find formal_tbl
n n global_vars*)
                  with Not_found -> raise (Failure ("variable " ^ n ^ " not
found in lookup"))
    in
    (* Construct code for an expression; return its value *)
    let rec expr builder ((_, e) : sexpr) = match e with
        SLiteral i  -> L.const_int i32_t i
      | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
      | SSTRliteral s -> L.build_global_stringptr s "string" builder
      | SFliteral l -> L.const_float_of_string float_t l
      | SNoexpr     -> L.const_int i32_t 0
      | SId s       -> L.build_load (lookup s) s builder
      | SArrayLit arr -> let len = L.const_int i32_t (List.length arr) in
            let size = L.const_int i32_t ((List.length arr) + 1) in
            let (fst_t, _) = List.hd arr in
            let ty = ltype_of_typ (A.Arr(fst_t, (List.length arr))) in
            (* allocate memory for array *)
            let arr_alloca = L.build_array_alloca ty size "arr" builder in
            (* bitcast *)
            let arr_ptr = L.build_pointercast arr_alloca ty "arrptr" builder
in
            (* store all elements *)
            let elts = List.map (expr builder) arr in
            let store_elt ind elt =
            let pos = L.const_int i32_t (ind) in
            let elt_ptr = L.build_gep arr_ptr [| pos |] "arrelt" builder in
            ignore(L.build_store elt elt_ptr builder)
            in List.iteri store_elt elts;
            let elt_ptr = L.build_gep arr_ptr [| len |] "arrlast" builder in
            let null_elt = L.const_null (L.element_type ty) in
            ignore(L.build_store null_elt elt_ptr builder);
            arr_ptr
      | SArrayAccess (s, e) ->
```

```
            let ind = expr builder e in
            let (ty, _) = e in
            (* increment index by one to get actual ptr position *)
            let pos = L.build_add ind (L.const_int i32_t 0) "accpos" builder
in
            let arr = expr builder (ty, (SId s)) in
            let elt = L.build_gep arr [| pos |] "acceltptr" builder in
            L.build_load elt "accelt" builder
      | SArrAssign (s, e1, e2) ->
            let ind = expr builder e1 in
            let (ty, _) = e1 in
            (* increment index by one to get actual ptr position *)
            let pos = L.build_add ind (L.const_int i32_t 0) "accpos" builder
in
            let arr = expr builder (ty, (SId s)) in
            let new_val = expr builder e2 in
            let elt_ptr = L.build_gep arr [| pos |] "arrelt" builder in
            L.build_store new_val elt_ptr builder
      | SAssign (s, e) -> let e' = expr builder e in
                          ignore(L.build_store e' (lookup s) builder); e'
      | SDeclAssn (t, s, e) ->
        let e' = expr builder e in
        let var = L.build_alloca (ltype_of_typ t) s builder in
            ignore (L.build_store e' var builder);
        StringHash.add tbl s var; e'

      | SBinop ((A.Float,_ ) as e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
            A.Add     -> L.build_fadd
          | A.Sub     -> L.build_fsub
          | A.Mult    -> L.build_fmul
          | A.Div     -> L.build_fdiv
          | A.Mod     -> L.build_frem
          | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
          | A.Neq     -> L.build_fcmp L.Fcmp.One
          | A.Less    -> L.build_fcmp L.Fcmp.Olt
          | A.Leq     -> L.build_fcmp L.Fcmp.Ole
          | A.Greater -> L.build_fcmp L.Fcmp.Ogt
          | A.Geq     -> L.build_fcmp L.Fcmp.Oge
          | A.Intersec -> raise (Failure "internal error: semant should have
rejected intersection binop with floats")
          | A.And | A.Or ->
            raise (Failure "internal error: semant should have rejected
and/or on float")
```

```
         ) e1' e2' "tmp" builder

       | SBinop ((A.String,_ ) as e1, op, e2) ->
         let e1' = expr builder e1
         and e2' = expr builder e2 in
         (match op with
             A.Add     -> L.build_call string_concat_f [| e1'; e2' |]
"string_concat" builder
           | A.Sub     -> L.build_call string_sub_f [| e1'; e2' |]
"string_sub" builder
           | A.Equal ->  (L.build_icmp L.Icmp.Eq) (L.const_int i32_t 0)
(L.build_call string_inequality_f [| e1'; e2' |] "string_inequality" builder)
"tmp" builder
           | A.Neq     -> (L.build_icmp L.Icmp.Ne) (L.const_int i32_t 0)
(L.build_call string_inequality_f [| e1';e2' |] "string_inequality" builder)
"tmp" builder
           | A.Intersec -> L.build_call string_intersection_f [| e1'; e2' |]
"string_intersection" builder
           | A.Mult ->  L.build_call string_mult_f [| e1'; e2' |]
"string_mult" builder
           | _ -> raise (Failure ("operation " ^ (A.string_of_op op) ^ " not
implemented")))


       | SBinop (e1, op, e2) ->
         let e1' = expr builder e1
         and e2' = expr builder e2 in
         (match op with
             A.Add     -> L.build_add
           | A.Sub     -> L.build_sub
           | A.Mult    -> L.build_mul
           | A.Div     -> L.build_sdiv
           | A.Mod     -> L.build_srem
           | A.And     -> L.build_and
           | A.Or      -> L.build_or
           | A.Equal   -> L.build_icmp L.Icmp.Eq
           | A.Neq     -> L.build_icmp L.Icmp.Ne
           | A.Less    -> L.build_icmp L.Icmp.Slt
           | A.Leq     -> L.build_icmp L.Icmp.Sle
           | A.Greater -> L.build_icmp L.Icmp.Sgt
           | A.Geq     -> L.build_icmp L.Icmp.Sge
           | A.Intersec -> raise (Failure "internal error: intersection
operator should have triggered on binop with string")
         ) e1' e2' "tmp" builder
       | SUnop(op, ((t, _) as e)) ->
           let e' = expr builder e in
         (match op with
```

```
          A.Neg when t = A.Float   -> L.build_fneg
          | A.Neg                  -> L.build_neg
          | A.Not                  -> L.build_not) e' "tmp" builder


      | SCall ("print", [e]) | SCall ("printb", [e]) -> L.build_call
printf_func [| int_format_str ; (expr builder e) |] "printf" builder
      | SCall ("printbig", [e]) -> L.build_call printbig_func [| (expr
builder e) |] "printbig" builder
      | SCall ("printf", [e]) -> L.build_call printf_func [|
float_format_str ; (expr builder e) |] "printf" builder
      | SCall ("printstr", [e]) -> L.build_call printf_func [|
string_format_str ; (expr builder e) |] "printf" builder
      | SCall ("slice", [v;e1;e2]) -> L.build_call slice_f [| (expr builder
v); (expr builder e1); (expr builder e2) |] "slice" builder
      | SCall ("len", [e]) -> L.build_call len_f [| (expr builder e) |]
"len" builder
      | SCall (f, args) ->
        let (fdef, fdecl) = StringMap.find f function_decls in
    let llargs = List.rev (List.map (expr builder) (List.rev args)) in
    let result = (match fdecl.styp with
                        A.Void -> ""
                      | _ -> f ^ "_result") in
         L.build_call fdef (Array.of_list llargs) result builder
     in

     (* LLVM insists each basic block end with exactly one "terminator"
        instruction that transfers control.  This function runs "instr
builder"
        if the current block does not already have a terminator.  Used,
        e.g., to handle the "fall off the end of the function" case. *)
     let add_terminal builder instr =
       match L.block_terminator (L.insertion_block builder) with
   Some _ -> ()
      | None -> ignore (instr builder) in
      (* Build the code for the given statement; return the builder for
        the statement's successor (i.e., the next instruction will be built
        after the one generated by this call) *)

     let rec stmt builder = function
   SBlock sl -> List.fold_left stmt builder sl
      | SExpr e -> ignore(expr builder e); builder
      | SReturn e -> ignore(match fdecl.styp with
                            (* Special "return nothing" instr *)
                            A.Void -> L.build_ret_void builder
                            (* Build return statement *)
```

```
                          | _ -> L.build_ret (expr builder e) builder );
                  builder
    | SIf (predicate, then_stmt, else_stmt) ->
       let bool_val = expr builder predicate in
let merge_bb = L.append_block context "merge" the_function in
       let build_br_merge = L.build_br merge_bb in (* partial function *)

let then_bb = L.append_block context "then" the_function in
add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
  build_br_merge;

let else_bb = L.append_block context "else" the_function in
add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
  build_br_merge;

ignore(L.build_cond_br bool_val then_bb else_bb builder);
L.builder_at_end context merge_bb

    | SWhile (predicate, body) ->
 let pred_bb = L.append_block context "while" the_function in
 ignore(L.build_br pred_bb builder);

 let body_bb = L.append_block context "while_body" the_function in
 add_terminal (stmt (L.builder_at_end context body_bb) body)
   (L.build_br pred_bb);

 let pred_builder = L.builder_at_end context pred_bb in
 let bool_val = expr pred_builder predicate in

 let merge_bb = L.append_block context "merge" the_function in
 ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
 L.builder_at_end context merge_bb

    (* Implement for loops as while loops *)
    | SFor (e1, e2, e3, body) -> stmt builder
    ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
 in

 (* Build the code for each statement in the function *)
 let builder = stmt builder (SBlock fdecl.sbody) in

 (* Add a return if the last block falls off the end *)
 add_terminal builder (match fdecl.styp with
     A.Void -> L.build_ret_void
   | A.Float -> L.build_ret (L.const_float float_t 0.0)
   | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
```

```
  in

  List.iter build_function_body functions;
  the_module
```

8.7 vowel.ml

```
(* Top-level of the MicroC compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module *)

 type action = Ast | Sast | LLVM_IR | Compile

 let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./vowel.native [-a|-s|-l|-c] [file.vwl]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;
   let lexbuf = Lexing.from_channel !channel in
  let ast = Vowelparse.program Scanner.token lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> print_string (Sast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate
sast))
    | Compile -> let m = Codegen.translate sast in
  Llvm_analysis.assert_valid_module m;
 print_string (Llvm.string_of_llmodule m)
```

8.8 vowelfunc.c

```
 #include <stdlib.h>
 #include <stdio.h>
 #include <string.h>
```

```c
#include <stdbool.h>


char *string_concat(char *s1, char *s2) {
   char *new = (char *) malloc(strlen(s1) + strlen(s2));
   strncpy(new, s1, strlen(s1)-1);
   strncat(new, s2+1, strlen(s2)-1);
   return new;
}

bool string_inequality(char *s1, char *s2){
   bool result;
   int res = strcmp(s1, s2);
   bool bres = false;
   if (res == 0){
       bres = false;
   }
   else{
       bres = true;
   }
   result = bres;
   return result;
}

bool string_equality(char *s1, char *s2){
   bool result;
   int res = strcmp(s1, s2);
   bool bres = false;
   if (res == 0){
       bres = true;
   }
   else{
       bres = false;
   }
   result = bres;
   return result;
}

char **string_intersection(char s1[], char s2[]) {

   char *first = calloc(sizeof (char) * 10, strlen(s1) + 1 );
   char *second = calloc(sizeof (char) * 10, strlen(s2) + 1);
   strcpy(first, s1); strcpy(second, s2);
   first[strlen(first)-1] = '\0'; second[strlen(second)-1] = '\0';
   first[0] = ' '; second[0] = ' ';
```

```
    int space_counter1 = 1;
    int space_counter2 = 1;

    for(int i = 0; s1[i] != '\0'; i++)
    {
         if (s1[i] == ' ')
         {
              space_counter1++;
         }
    }
    for(int i = 0; s2[i] != '\0'; i++)
    {
         if (s2[i] == ' ')
         {
              space_counter2++;
         }
    }

    char **array = malloc (sizeof (char *) * space_counter1);
    char **array2 = malloc (sizeof (char *) * space_counter2);

    char *t = strtok(first, " ");
    int i = 0;
    while (t != NULL)
    {
        array[i] = t;
        i++;
        t = strtok (NULL, " ");
    }

    char *t2 = strtok (second, " ");
    int i2 = 0;
    while (t2 != NULL)
    {
        array2[i2] = t2;
        i2++;
        t2 = strtok (NULL, " ");
    }

    int counter = 0;
    for (int j = 0; j < i; j++) {
        for (int k = 0; k < i2; k++) {
            if (strcmp(array[j], array2[k]) == 0) {
                counter++;

            }
```

```
        }
    }
    char **arr_res = malloc (sizeof (char *) * counter);
    int l = 0;
    for (int j = 0; j < i; j++) {
        for (int k = 0; k < i2; k++) {
            if (strcmp(array[j], array2[k]) == 0) {
                arr_res[l] = array[j];
                l++;
            }
        }
    }
    return arr_res;
}

char **string_sub(char s1[], char s2[]) {
    char *first = calloc(sizeof (char) * 100, strlen(s1) + 1 );
    char *second = calloc(sizeof (char) * 100, strlen(s2) + 1);
    strcpy(first, s1); strcpy(second, s2);
    first[strlen(first)-1] = '\0'; second[strlen(second)-1] = '\0';
    first[0] = ' '; second[0] = ' ';

    int space_counter1 = 1;
    int space_counter2 = 1;

    for(int i = 0; s1[i] != '\0'; i++)
    {
         if (s1[i] == ' ')
         {
              space_counter1++;
         }
    }
    for(int i = 0; s2[i] != '\0'; i++)
    {
         if (s2[i] == ' ')
         {
              space_counter2++;
         }
    }

    char **array = malloc (sizeof (char *) * space_counter1);
    char **remove_array = malloc (sizeof (char *) * space_counter1);
    char **array2 = malloc (sizeof (char *) * space_counter2);

    char *zero = "0";
    for (int w = 0; w < space_counter1; w++){
```

```
        remove_array[w] = zero;
    }

    char *t = strtok(first, " ");
    int i = 0;
    while (t != NULL)
    {
        array[i] = t;
        i++;
        t = strtok (NULL, " ");
    }

    char *t2 = strtok (second, " ");
    int i2 = 0;
    while (t2 != NULL)
    {
        array2[i2] = t2;
        i2++;
        t2 = strtok (NULL, " ");
    }

    char *one = "1";
    int counter = 0;
    for (int j = 0; j < i; j++) {
        for (int k = 0; k < i2; k++) {
            if (strcmp(array[j], array2[k]) == 0) {
                counter++;
                remove_array[j] = one;

            }
        }
    }
     char **arr_res = malloc (sizeof (char *) * counter);
    int l = 0;
    for (int j = 0; j < i; j++) {
            if (remove_array[j]== zero) {
                arr_res[l] = array[j];
                l++;
            }
        }
    return arr_res;
}

char *slice(const char *str, size_t s, size_t e) {
    s++;
    e++;
```

```
    size_t index = 0;
    size_t length = strlen(str) ;
    char *slicestring = (char*)malloc(length +1);
    while (s < e && s <length){
        slicestring[index] = str[s];
        index++;
        s++;
    }
    slicestring[index] = '\0';

    return slicestring;
}
int len(const char *str){
    int l;
    size_t len = strlen(str);
    l = (int)(len) -2 ;

}

char *string_mult(char *s1, size_t s) {
    char *first = calloc(sizeof (char) * 100, strlen(s1) + 1 );
    char *second = calloc(sizeof (char) * 100, strlen(s1) + 1 );
    strcpy(first, s1);
    first[strlen(first)-1] = '\0';
    first[0] = ' ';
    first++;
    strcpy(second, first);
    size_t length = strlen(second) ;
    char *new = (char*)malloc(length*s);
    while (s>0){
    strcat(new,second);
    s--;
    }
    return new;
}
```

8.9 Makefile

```
# "make test" Compiles everything and runs the regression tests

.PHONY : test
test : all testall.sh
    ./testall.sh

# "make all" builds the executable as well as the "printbig" library
designed
```

```
# to test linking external code

.PHONY : all

all : vowel.native vowel_func.o printbig.o

# "make vowel.native" compiles the compiler
#
# The _tags file controls the operation of ocamlbuild, e.g., by including
# packages, enabling warnings
#
# See https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc

vowel.native :
   opam config exec -- \
   rm -f *.o
   ocamlbuild -use-ocamlfind -pkgs llvm.bitreader vowel.native
   gcc -c vowel_func.c
   cc -emit-llvm -o vowel_func.bc -c vowel_func.c -Wno-varargs
#used to be clang in the last line
# "make clean" removes all generated files
.PHONY : clean
clean :
   ocamlbuild -clean
   rm -rf testall.log ocamlllvm *.diff
   rm -f *.o *.output vowel_func.bc

# Testing the "printbig" example

vowel_func : vowel_func.c
   cc -o vowel_func -DBUILD_TEST vowel_func.c

# Building the tarball

TESTS = \
 aaaa add1 arith1 arith2 arith3 fib float1 float2 float3 for1 for2 func1 \
 func2 func3 func4 func5 func6 func7 func8 func9 gcd2 gcd global1 \
 global2 global3 hello if1 if2 if3 if4 if5 if6 local1 local2 ops1 \
 ops2 printbig var1 var2 while1 while2 declassn lex incr intarr incrstr
strequality \
 stringnotequal str-intersect stringsub slice strlen

FAILS = \
 assign1 assign2 assign3 dead1 dead2 expr1 expr2 expr3 float1 float2 \
 for1 for2 for3 for4 for5 func1 func2 func3 func4 func5 func6 func7 \
 func8 func9 global1 global2 if1 if2 if3 nomain printbig printb print \
```

```
  return1 return2 while1 while2

 TESTFILES = $(TESTS:%=test-%.vwl) $(TESTS:%=test-%.out) \
        $(FAILS:%=fail-%.vwl) $(FAILS:%=fail-%.err)

 TARFILES = ast.ml sast.ml codegen.ml Makefile _tags vowel.ml vowelparse.mly
\
    README scanner.mll semant.ml testall.sh \
    vowel_func.c vowel_func.c arcade-font.pbm font2c \
    Dockerfile \
    $(TESTFILES:%=tests/%)

 vowel.tar.gz : $(TARFILES)
    cd .. && tar czf vowel/vowel.tar.gz \
        $(TARFILES:%=microc/%)
```

8.10 testall.sh

```
 #!/bin/sh

 # Regression testing script for Vowel
 # Step through a list of files
 #  Compile, run, and check the output of each expected-to-work test
 #  Compile and check the error of each expected-to-fail test

 # Path to the LLVM interpreter
 LLI="lli"
 #LLI="/usr/local/opt/llvm/bin/lli"

 # Path to the LLVM compiler
 LLC="llc"

 # Path to the C compiler
 CC="cc"

 # Path to the microc compiler.  Usually "./microc.native"
 # Try "_build/microc.native" if ocamlbuild was unable to create a symbolic
link.
 MICROC="./vowel.native"
 #MICROC="_build/microc.native"

 # Set time limit for all operations
 ulimit -t 30
```

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

keep=0

Usage() {
   echo "Usage: testall.sh [options] [.vwl files]"
   echo "-k    Keep intermediate files"
   echo "-h    Print this help"
   exit 1
}

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

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to
difffile
Compare() {
   generatedfiles="$generatedfiles $3"
   echo diff -b $1 $2 ">" $3 1>&2
   diff -b "$1" "$2" > "$3" 2>&1 || {
   SignalError "$1 differs"
   echo "FAILED $1 differs from $2" 1>&2
   }
}


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

```
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
   echo $* 1>&2
   eval $* && {
   SignalError "failed: $* did not report an error"
   return 1
   }
   return 0
}

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

   echo -n "$basename..."

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

   generatedfiles=""

   generatedfiles="$generatedfiles ${basename}.ll ${basename}.s
${basename}.exe ${basename}.out" &&
   Run "$MICROC" "$1" ">" "${basename}.ll" &&
   Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s"
&&
   Run "$CC" "-o" "${basename}.exe" "${basename}.s" "vowel_func.o" &&
   Run "./${basename}.exe" > "${basename}.out" &&
```

```
    Compare ${basename}.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

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

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$MICROC" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

  # Report the status and clean up the generated files

  if [ $error -eq 0 ] ; then
  if [ $keep -eq 0 ] ; then
```

```
        rm -f $generatedfiles
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
    k) # Keep intermediate files
        keep=1
        ;;
    h) # Help
        Usage
        ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
 echo "Could not find the LLVM interpreter \"$LLI\"."
 echo "Check your LLVM installation and/or modify the LLI variable in
testall.sh"
 exit 1
}

which "$LLI" >> $globallog || LLIFail

if [ ! -f vowel_func.o ]
then
    echo "Could not find printbig.o"
    echo "Try \"make printbig.o\""
    exit 1
fi

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.vwl tests/fail-*.vwl"
fi
```

```
for file in $files
do
   case $file in
   *test-*)
       Check $file 2>> $globallog
       ;;
   *fail-*)
       CheckFail $file 2>> $globallog
       ;;
   *)
       echo "unknown file type $file"
       globalerror=1
       ;;
   esac
done

exit $globalerror
```

8.11 Test files

The following tests is subset of tests we wrote to make sure our language compiles and produces expected results. The rest is included in the zip file for the project submission.

8.11.1 Successful tests

test-stringsub.vwl

```
int foo() {
    string s1 = "hello this is a string";
    string s2 = "hello a";
    string[] arr = s1 - s2;

    int i = 0;

    for (i = 0; i < 3; i+=1) {
        printstr(arr[i]);
    }
}

foo();
```

test-strlen.vwl

```
int foo(){
    string a = "hellos";
    int x = 0;

    x = len(a);
    print(x);
    return 0;
  }

  foo();
```

test-strmult.vwl

```
int foo(){
    string a = "hello";
    string b = " ";
    b = a * 3;
    printstr(b);

}

foo();
```

test-stringnotequal.vwl

```
string a = "Hell";
string b = "Hello";
printb(a!=b);
```

test-str-intersect.vwl

```
int foo(){
    string a = "my name is julie";
    string b = "my name is bob";
    string[] c = a & b;

      int i = 0;
    for (i = 0 ; i < 3 ; i = i + 1) {
        printstr(c[i]);
    }
}
```

```
foo();
```

test-slice.vwl

```
int foo(){
      string r = "thisismystring";
      string k = slice(r,2,7);
      printstr(k);
      return 0;
}
foo();
```

test-incrstr.vwl

```
string a = "hello";
a += "world";
printstr(a);
```