THE ENGLISH LANGUAGE LANGUAGE

Rabia Akhtar (ra2805)
Emily Bau (eb3029)
Candace Johnson (crj2121)
Michele Lin (ml3706)
Nivita Arora (na2464)

December 20th, 2017

Contents

1 Introduction

1.1 What is The English Language Language?

In 2010, nearly one out of three college students across the United States reported having plagiarized assignments from the Internet, and about 10% of college students have plagiarized work at least once from another student. These shocking statistics reveal the prevalent issue of plagiarism across schools in the country, as well as around the world.

The English Language Language solves problems specific to document manipulation and data extrapolation. In traditionally utilized languages, it is often difficult to write scripts that can analyze multiple documents quickly, and can cross-compare them. Storing a document itself is tough, but on top of that, being able to compare multiple documents at once is very rare. Our unique language provides core file manipulation operations and storage structures, which allow us to mine statistics that will check for plagiarism between given documents, as well as other tasks related to document manipulation. This is especially useful for teaching, grading, publishing, and other such projects and work. We hope to facilitate the efforts of teachers and publishers by offering them a service that will allow them to mine statistics about documents and cross-compare them quickly and easily, without having to peruse hundreds of documents manually.

2 Usage

Here is a brief overview on how to use our language, including writing, compiling, and running your ELL program.

2.1 Getting Started

First, confirm that you have the LLVM version 5.0.0 environment set up on your system. Then, download the language project folder onto your local system. In your terminal, move into this directory, and then move further into the `english-llvm` subdirectory. You are all set to start running your own ELL programs!

2.2 Writing Your Program

Create a new file using your favorite text editor, with the extension `.ell`. As a demonstration, let us start with a simple "Hello World" program, named `helloworld.ell`. Here is what the program looks like:

```
// helloworld.ell
int main() {
    print_string("hi world");
    return 0;
}
```

2.3 Compiling

Make sure you are in the `eng-lang-master/english-llvm` directory. Type the command `make` in your terminal window, which will generate our ELL compiler. Then, run the following commands in your shell:

```
$ make
$ ./run_english helloworld.ell
```

2.4 Manual Compiling

You can also manually make and compile using the following commands.

```
$ ./english.native < hello_world.ell > hello_world.ll
$ /usr/local/opt/llvm/bin/llc hello_world.ll > hello_world.s
$ cc -o hello_world.exe out.s c-code.o
$ ./hello_world.exe
```

3 Language Tutorial

Now we can get started with writing more complicated programs. The English Language language allows for various data types, structures, and functions. Here we will give an overview of each one, as well as sample code to demonstrate its usage. For more detailed reference on usage and syntax, please refer to the Language Reference Manual in Section 4.

3.1 Numerical Operations

The two numerical types that ELL supports are `integers` and `double`. Various operations can be performed on these types, such as basic arithmetic computations. Below is an example of simple addition between integers:

```
int sum = 3 + 5;
```

Doubles can be declared as follows and follow IEEE floating point standards.

```
double a = 3.0;
```

3.2 Boolean

Use a boolean to specify true or false statements. These are useful for conditional statements and loops, discussed later on.

```
bool max = true;
bool min = false;
```

3.3 Char

Declare an ASCII char with single quotes.

```
char a = ’a’;
```

3.4 String

Declare a series of chars as a string literal. Built in string operations are found in Section 3.11.4. Note that a **file ptr** is a string specific for holding a file open.

```
string word = "hello world";
```

3.5 Array

Arrays allow you to store elements in memory. Each arrays must contain items of the same type. The user can implement arrays for objects of type `int`, `double`, or `string`. Some common uses for arrays are accessing elements, indexing the array, and reassigning elements, and the syntax for these operations are displayed here for reference, but are elaborated upon in the Language Reference Manual. Example:

```
string[] names; /* declare an array */
names = ["emily", "rabia", "michele"]; /* initialize array elements */
string test = names{|0|}; /* accessing an element in the array */
names[3] = "fun"; /* reassigning an element in the array */
```

3.6 Struct

A struct is a data type declaration that defines a grouped list of variables to be placed under one name in a single block of memory, allowing the different variables to be assigned and accessed through their field names via the dot operator. Example:

```
struct Doc {
    string File_name;
    string[] Content;
    int Word_count;
};

struct Doc document;
document.File_name = "hello.txt";
print_all(document.File_name);
```

3.7 Variable Declaration

Variables can be declared without any value assigned. Variables of type `int`, `bool`, `double`, `string`, and `char` can also be declared and initialized at once. Example:

```
int a;
int b = 1;
```

3.7.1 Global Variables

Global variables can be initialized with literals, but not expressions. Example:

```
int a = 2;
double b = 1.5;
```

```

int main() {
    print(a);
}
```

3.7.2 Local Variables

Local variables can be initialized with literals or expressions. Example:

```
int main() {
    int a = 2;
    int b = a + 2 * 3;
}
```

3.8 For Loop

Use a for loop to iterate until a certain condition renders false. Example:

```
for (int i=0; i<5; i++) {
    <expr>
}
```

3.9 While Loop

Use a while loop to iterate while a given condition remains true. Example:

```
int count = 0;
while (count < 5) {
    <expr>
    count++;
}
```

3.10 If/else Statement

Use an if/else statement to have an event occur once based on the truth of a single condition. Example:

```
int i = 0;
if (int i < 5) {
    <expr>
}
else {
    <expr>
}
```

4 Language Reference Manual

4.1 Lexical Elements

4.1.1 Identifiers

Identifiers are used for naming user-defined objects in an ELL program, such as variables and arrays. They are defined by at least one lowercase letter and can be followed by any combination of letters, numbers, or underscores. The following regular expression defines identifier names:

```
['a'-'z'][['a'-'z 'A' - 'Z' '0' - '9' '_']*]
```

4.1.2 Reserved Keywords

int	boolean	double	string
char	void	struct	file_ptr
while	for	if	else
read	write	open	close
return	print	print_string	print_char
print_double	print_all	printbig	printb
strcat	strcmp	strlen	strcpy
strget	to_lower	word_count	string_at
is_stop_word	calloc	free	

4.1.3 Literals

Integer Literal: a series of one or more digits from 0-9
double Literal: a series of digits followed by a '.' character and another series of digits. Must have digits either preceding or following the '.'
Boolean Literal: a value of either 'true' or 'false'
String Literal: a series of one or more characters

4.1.4 Operators

+, -, *, /, %, ++, - -	arithmetic integer operators
==, <, >, <=, >=	numerical integer operators
++, –	post fix increment/decrement operators
| |, &&, !	logical operators
=	assignment

4.1.5 Delimiters

Parentheses: used to contain arguments in function calls, as well as to ensure precedence in expressions
Semicolon: denotes the end of a statement
Curly Braces: used to enclose block logic in conditionals and loops, as well as to contain code in functions

COMMAS: used to separate arguments in function calls, as well as elements in arrays

4.1.6 Whitespace

Whitespace is only used to separate tokens.

4.1.7 Comments

All comments (both single and multi line) are started with /* and ended with */

4.2 Data Types

4.2.1 Primitive Data Types

INTEGERS
An integer number is declared as type int. Integer values can be assigned to a variable, assigned to an array element, passed as function arguments, or utilized to iterate over loops. Example:

```
int x = 5;
int [] count = [1,2,3];
```

DOUBLES
doubleing point numbers are similar to integers in usage, but are declared as type double. Example:

```
double x = 2.3;
```

BOOLEANS
Boolean values will be declared as type bool and can be either true or false. Example:

```
bool match = true;
```

4.2.2 Non-Primitive Data Types

STRINGS
strings are defined by the type string. They will begin and end with a double quote, as shown, " ". Example:

```
string intro = "Hello World"
```

String operations:
strlen(a) returns the number of characters in string a
strcat(a, b) appends string b to the end of string a

strcpy(a, b) copies contents of string b into string a
strget(a, i) gets character in string a at int i
to_lower(a) returns string a in all lowercase characters
strcmp(b) compares string a to string b and returns 0 if they are equal, a negative integer if a < b, and a positive integer if a > b

Structures

Structures are defined by the type struct. Structure names must begin with a capital letter, and can be followed by any combination of letters, numbers, or underscores. The following outlines the declaration of a struct:

```
struct <struct_name> {
    <type> <field1_name>;
    <type> <field2_name>;
    (...)
};
```

The following demonstrates an example of defining a struct:

```
struct Object {
    int x;
    string y;
};
```

In order to declare and initialize a struct, assign or reassign its fields, call it, or perform any operations on it, we use the dot operator. The following demonstrates an example of using the previously declared struct:

```
/* initialize a struct of struct type Object */
struct Object example;
example.x = 5;
example.y = "hi";
```

Arrays

Arrays are a collection of int, double, or string data types, and can be defined by simply typing the data type followed by an open and close bracket, followed by the name of the array. Each arrays must contain items of the same type specified in the declaration of the array. The size of the array is never specified between the brackets. Array indexes are initially set to zero. The following demonstrates an example of declaring an array:

```
int [] a; /* an empty array of integers */
string [] names; /* an empty array of strings */
```

To initialize an array, we call the array name, followed by an equal sign for assignment, and then define its elements between brackets and separated by commas. Example:

```
names = ["emily", "rabia", "michele"];
```

To reassign the values of an array, we call the array name, directly followed by the integer index number of the element we wish to assign, placed between brackets. However, to assign the elements of an array to another variable, or simply access already-defined elements, we specify the index between curly braces and pipe characters, as shown below. Example:

```
names[3] = "nivita"; /* reassigns the element "michele" to "nivita" */
string test = names{|0|}; /* sets the string test to "emily" */
```

File Pointers

File pointers are defined by the type `file_ptr`, and point to a an opened text file. Example:

```
file_ptr fp = open(file, "rb");
```

4.2.3 Functions

Built-in Functions

These functions are predefined in the compiler. In ELL, these are the string operations specified above, as well as the following functions:

MEMORY:

```
 string calloc (int num, int size);
```

Allocates a block of memory for an array of num elements, each of them size bytes long, and initializes all its bits to zero.

```
 void free (string alloced_memory);
```

A block of memory previously allocated by a call to calloc is made available again.

IO:

```
 void print (int x);
```

Print an object of type `int`

```
 void print_double (double x);
```

Print an object of type `double`

```
 void print_char (char x);
```

Print an object of type `char`

```
void print_string (string x);
```

Print an object of type string

```
void print_all (<type> x);
```

Print an object of any of the following types, including all primitive types: int, double, float, boolean, single and binary operators

FILE IO:

```
file_ptr open(string filename, string mode);
```

- r - open for reading
- w - open for writing (file need not exist)
- a - open for appending (file need not exist)
- r+ - open for reading and writing, start at beginning
- w+ - open for reading and writing (overwrite file)
- a+ - open for reading and writing (append if file exists)

Opens the file specified and returns a file pointer. Once you've opened a file, you can use the file pointer to perform input and output functions on the file.

```
int close(file_ptr file);
```

Close returns zero if the file is closed successfully.

```
int read(string buf, int size_of_elements, int number_of_elements,
    file_ptr file);
```

The first argument is the name of the array or the address of the structure you want to write to the file. The second argument is the size of each element of the array in bytes. The third argument is the number of elements you want to read in. Finally, file is the return pointer of open.

```
int write (string data, file_ptr file);
```

Writes the string pointed by data to the file. On success, a non-negative value is returned.

4.2.4 String Operations

```
int strlen(string str)
```

Computes the length of the string str up to but not including the terminating null character.

```
int strcmp(string st1, string str2)
```

Compares the string pointed to, by str1 to the string pointed to by str2.

```
string strcat(string des, string str2)
```

Appends a copy of the source string to the destination string. "strcpy"; "strget"; "to_lower";

```
int is_stop_word(string str)
```

Returns 1 if string passed in is a stop word, 0 otherwise. Stopwords from here: https://www.ranks.nl/stopwords

```
int word_count(string str)
```

Returns number of words in inputted string.

```
int string_at(string str, int indice, int )
```

Returns number of words in inputted string.

User-Defined Functions

These functions are constructed by the user in their ELL program, and can be defined and called as follows:

```
return_type function_name(<args>) {
    ...
    return return_type;
}

function_name(<args>);
```

4.2.5 Control Flow Statements

Conditionals

Conditional statements use the key words `if` and `else` to allow you to only run a series of operations based on a specified condition.

```
if (<bool>) {
    <expr>
}
```

```
else {
    <expr>
}
```

Loops

Loops will iterate while a given conditional statement is true. In ELL, these are called `for` and `while` loops.

```
for (int i=0; i<5; i=i+1) {
    <expr>
}
while (<bool>) {
    <expr>
}
```

5 Project Plan

5.1 Process

As a team, we started from Stephen Edwards' MicroC compiler shown in class. After ensuring our understanding of it, we edited it to produce an output of "hello world". Once that stage in the programming was completed, we split up tasks and began building off of this new compiler to implement the various types, structures, and functions we needed for our own language. Throughout the process, we continued helping each other out and debugging each others code.

1. Worked together to build the Hello World program that uses the skeleton MicroC compiler provided by Professor Edwards, and to have it print out the string "hello world."

2. Split up the work based on the difficulty of each part, as well as on our role designations assigned at the beginning of the semester

3. Implemented the basic data types needed, namely integers, doubles, and strings.

4. Implemented struct declaration, allowing the previously defined data types to be declared within struct definitions.

5. Built up from struct declarations to allow for struct access, operations, and usage.

6. Basic built-in functions defined, such as various functions that each print objects of a different data type.

7. String functions added, some linked from C libraries, and other unique ones defined in the compiler.

8. Added the character data type.

9. Implemented arrays, including declaring, accessing, and manipulating.

10. Fixed bugs in string functions, passing arrays as arguments, and other areas.

11. Demo code programs built and tested.

5.2 Team Roles

Emily Bau: Project Manager
Michele Lin: System Architect
Candace John: System Architect
Nivita Arora: Language Guru
Rabia Akhtar: Tester

5.3 Software Development Environment

We utilized GitHub's version control system for this project, as we had a relatively large group with five team members. We created a GitHub repository for our project, and each pulled and committed the code we worked on separately, using our own programming environment preferences.

In terms of languages, we used OCaml LLVM for our main compiler files, including the abstract syntax tree builder, the LLVM IR, the semantic checker, and the code generation. We used OCamllex for our MicroC scanner.

5.4 Style Guide

We did not have a strict style guide to adhere to as we were coding.

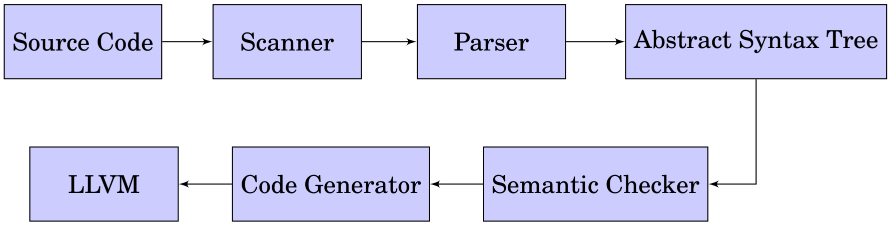
In our source code, we always did these things

1. Indent clearly to show dependencies
2. Use helper functions judiciously in order to increase readability and clarity
3. Name functions and variables using the underscore convention
4. Use descriptive names in order to uncover the details of the function
5. Do not leave any warnings, especially parser warnings, in the build
6. Use comments where the code is not clear

6 Architectural Design

Our compiler begins with the source code, passes that through the scanner and tokenizes it. This output is then passed through a parser, from which an abstract syntax tree is constructed. Then, the semantic analyzer checks the semantics of the program to detect any issues in structures, declarations, arguments, etc., and then passes that output through the code generator. Finally, this output is translated into LLVM code.

6.1 Block Diagram



6.2 Dividing the Work

Hello World	All
Char	Rabia
Increment / Decrement Operator	Rabia
Structs	Rabia, Nivita
String Functions	Rabia
File IO	Rabia
Strings	Rabia, Candace
Arrays	Candace
Doubles	Michele
Variable Init	Michele
Function Returns	Michele
Word Manipulation Functions	Emily
Print Functions	Rabia, Emily, Candace, Michele
Code Demos	Emily
Final Report	Nivita

7 Test Plan

Here we demonstrate three programs written in ELL that demonstrate most of the functionalities of our language.

7.1 Code Demos

Who did what: Emily Bau mainly worked on developing the demo codes, but everyone contributed ideas and debugging help.

7.1.1 Longest Common Substring

The following program demonstrates an example of how an instructor can find the longest common overlap between two text files, to assist with detecting plagiarism between those two students.

```
/* lcs.ell */

/* this demo finds the longest common substring of two text files */
/* function finds longest common substring */

/* struct for storing longest common substring and length */
struct LongestC {
   int L_count;
   string L_string;
};

/* function finds longest common substring of two strings and returns the substring and length in struct */
struct LongestC lcs(string a, string b){

   /* declare variables */
   int a_s = strlen(a);
   int a_t;
   int b_s = strlen(b);
   int b_t;
   int i;
   int j;
   int equal = 1;
   int longest = 0;
   string temp;
   int count = 0;
   string x;
   string y;
   struct LongestC result;
   result.L_string = calloc(1, a_s);
   result.L_count = 0;

   /* iterate to find longest substring */
   for (i = 0; i< a_s; i ++) {
      for (j = 0; j < b_s; j++){
         x = string_at(a, i, 1 , 2);
         y = string_at(b, j, 1, 2);
         temp = calloc(1, a_s);
         if (strcmp(x, y) == 0){
            a_t = i;
            b_t = j;
            while(equal == 1 && a_t< a_s && b_t< b_s){
               free(x);
               free(y);
               x = string_at(a, a_t, 1, 2);
               y = string_at(b, b_t, 1, 2);
               if (strcmp(x, y) == 0){
                  strcat(temp, x);
                  count++;

               }
               if(strcmp(x, y) != 0){
                  equal = 0;
               }
               a_t++;
               b_t++;

            }

            if (count > result.L_count){
               result.L_count = count;
               strcpy(result.L_string, temp);

            }
            count = 0;
            equal = 1;
         }
         /* free allocated memory */
         free(x);
         free(y);
         free(temp);
      }
   }
   /* return struct containing longest common substring */
   return result;
}

/* function takes in filepath, reads in text and returns string */
string read_Essay(string file){
   string s1 = calloc(1, 2000);
   file_ptr fp = open(file, "rb");
   int size = read(s1, 1, 2000, fp);
   close(fp);
   return s1;

}

int main()
{
   struct LongestC l;

   /* read in essays */
   string rabia = read_Essay("tests/demo_lcs_one.txt");
   string nivita = read_Essay("tests/demo_lcs_two.txt");

   /* find longest common substring */
   l = lcs(rabia, nivita);

   /* print result */
   print(l.L_count);
   print_string(l.L_string);

   /* free allocated memory */
   free(rabia);
   free(nivita);
   free(l.L_string);
   return 0;
}
```

Output:

```
/* test-demo-lcs.out */
174
a and this is my Essay. PLT is a great class and I enjoy learning about
    building my own language. My language is called the English
    Language Language and it is really great.
```

7.1.2 Word Count

The following program demonstrates an example of how an instructor can see which students remained within the word count limit for an assignment.

```
/* this demo shows using our language to check if a list of essays
    follows a word count. In this example there is a class struct with
    students and submitted essays. */

struct Class{

   int Class_size;
   string [] Students;
   string [] Essays;
};
```

```

/* function checks wordcounts of all submitted essays in class struct */
int follow_word_count(struct Class c, int word_limit){
   int i;
   string name;
   string file;
   string text;
   int count;
   for (i = 0; i<c.Class_size; i++){
      name = c.Students{|i|};
      file = c.Essays{|i|};
      text = read_Essay(file);
      count = word_count(text);
      /* check if text file follows word count */
      if (count < word_limit + 1){
         print_string(name);
         print_string("followed the word count. Their essay had");
         print(count);
         print_string("words. Here is their essay:");
         print_string(text);
      }
      if(count > word_limit){
         print_string(name);
         print_string("did not follow the word count. Their essay had");
         print(count);
         print_string("words. Here is their essay:");
         print_string(text);
      }
      print_string("");
      free(text);
   }
   return 0;
}

/* function takes in filepath, reads in text and returns string */
string read_Essay(string file){
   string s1 = calloc(1, 2000);
   file_ptr fp = open(file, "rb");
   int size = read(s1, 1, 2000, fp);
   close(fp);
   return s1;
}

int main() {
  struct Class cl;
  string [] s = ["Candace","Emily", "Michele", "Nivita", "Rabia"];
  string [] e = ["tests/candace.txt", "tests/emily.txt",
      "tests/michele.txt", "tests/nivita.txt", "tests/rabia.txt"];
  cl.Students = s;
  cl.Essays = e;
  cl.Class_size = 5;
  follow_word_count(cl, 90);
  return 0;

}
```

Output:

```
/* test-demo-wordcount.out */
Candace
did not follow the word count. Their essay had
92
words. Here is their essay:
The Moon is a barren, rocky world without air and water. It has dark
    lava plain on its surface. The Moon is filled wit craters. It has
    no light of its own. It gets its light from the Sun. The Moo keeps
    changing its shape as it moves round the Earth. It spins on its
    axis in 27.3 days stars were named after the Edwin Aldrin were the
    first ones to set their foot on the Moon on 21 July 1969 They
    reached the Moon in their space craft named Apollo II.

Emily
followed the word count. Their essay had
81
words. Here is their essay:
The doctor is a person who looks after the sick people and prescribes
    medicines so that the patient recovers fast. In order to become a
    doctor, a person has to study medicine. Doctors lead a hard life.
    Their life is very busy. They get up early in the morning and go to
    the hospital. They work without taking a break. They always remain
    polite so that patients feel comfortable with them. Since doctors
    work so hard we must realise their value.

Michele
did not follow the word count. Their essay had
109
words. Here is their essay:
The Taj Mahal is a beautiful monument built in 1631 by an Emperor named
    Shah Jahan in memory of his wife Mumtaz Mahal. It is situated on
    the banks of river Yamuna at Agra. It looks beautiful in the
    moonlight. The Taj Mahal is made up of white marble. In front of
    the monument, there is a beautiful garden known as the Charbagh.
    Inside the monument, there are two tombs. These tombs are of Shah
    Jahan and his wife Mumtaz Mahal. The Taj Mahal is considered as one
    of the Seven Wonders of the World. Many tourists come to see this
    beautiful structure from different parts of the world.

Nivita
did not follow the word count. Their essay had
96
```

```
words. Here is their essay:
A snake charmer is a person who moves the streets with different types of the banks of the river Yamuna. It is snakes in his basket. He goes from one place to another to show various types of snakes and their tricks. He carries a pipe with which he plays music and snakes dance to his tune. He usually wears a colourful dress. The job of a snake charmer is quite dangerous. Some snakes are quite poisonous and can even bite him. It is not an easy task to catch and train them for the shows.

Rabia
followed the word count. Their essay had
71
words. Here is their essay:
The Solar System consists of the Sun Moon and Planets. It also consists of comets, meteoroids and asteroids. The Sun is the largest member of the Solar System. In order of distance from the Sun, the planets are Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune and Pluto; the dwarf planet. The Sun is at the centre of the Solar System and the planets, asteroids, comets and meteoroids revolve around it.
```

7.1.3 Essay Topics

The following program demonstrates an example of how someone can find the main topics in an Essay. The code has two functions, one for reading in a test file and the other for creating a string of the topic words that exclude stop words.

```
/* essay_topics.ell */
/* this demo code reads in a text document and finds the topic words in the file */

/* function builds array of topic words */
int main_topics(string a){

  /* set variables */
  int i;
  int len = strlen(a);
  int start = 0;
  string x;
  string b;
  string temp;
  string [] topics = [""];
  int j;
  int w = 0;
  int indice =0;
  int present = 0;
  int k;
  int h;
  int l;

/* iterate through string by character */
  for(i = 0; i<len; i++){
    temp = calloc(30,1);
    x = string_at(a,i,30, 1);

    /* if end of word, build previous word */
    if(strcmp(x, " ") == 0){
      j = start;
      for(; j < i; j++){
        b = string_at(a,j, 30, 1);
        if(strcmp(b,".") != 0){
        strcat(temp, b);
        }
        free(b);
      }

      /* put word in array if not a stop word*/
      if(is_stop_word(temp) == 0){
      topics [indice] = calloc(30, 1);
      strcpy(topics{|indice|}, temp);
      indice++;
      }
      start = i+1;
    }
    free(x);
  }

    /* print array of topic words */
    for(h = 0; h < indice; h++){
      print_string(topics{|h|});
      free(topics{|h|});
    }

    free(temp);
    return 0;
}

/* function takes in filepath, reads in text and returns string */
string read_Essay(string file){
   string s1 = calloc(1, 2000);
   file_ptr fp = open(file, "rb");
   int size = read(s1, 1, 2000, fp);
   close(fp);
   return s1;
}

int main() {
  int i;
  string essay;
  essay = read_Essay("tests/essay1.txt");
  main_topics(essay);
  free(essay);
  return 0;
}
```

Output:

```
Oceans
lakes
much
common
quite
different
well
Both
bodies
water
oceans
large
bodies
salt
water
lakes
much
smaller
bodies
fresh
water
Lakes
usually
surrounded
land
oceans
surround
```

7.2 Test Suites

We have dozens of different test suites that we use to test each part of our compiler. To see all of them, please refer to the folder `eng-lang/english-llvm/tests/`. We chose these test cases below in particular to display here because we felt that it highlights the key aspects and operations of our code, and the basis to the main functionalities of our language.

Here are some of the test suites we have for testing important operations, objects, and data types, along with their corresponding output files:

Integers:

```
/* test-add1.ell */
int add(int x, int y) {
  return x + y;
}

int main() {
  print(add(17, 25));
  return 0;
}
```

Output:

```
# test-add1.out
42
```

Allocating Memory

```
/* test-alloc.ell */
int main(){
   string s1 = calloc(1, 2000);
   file_ptr fp = open("tests/hello.txt", "rb");
      int size = read(s1, 1, 2000, fp);
      close(fp);
      print(size);
      print_string(s1);
      free(s1);
   return(0);
}
```

Output:

```
# test-alloc.out
11
hello world
```

Basic Arithmetic Operations

```
/* test-arith3.ell */
int foo(int a) {
  return a;
}

int main() {
  int a;
  a = 42;
  a = a + 5;
  print(a);
  return 0;
}
```

Output:

```
# test-arith3.out
47
```

Characters

```
/* test-char1.ell */
int main() {
   char c = ’A’;
   char a = to_lower(c);
   print_char(a);
   return (0);
}
```

Output:

```
# test-char1.out
a
```

Doubles

```
/* test-float4.ell */
int main() {
  double a;
  a = 42.1;
  a = a + 3.3;
  print_double(a);
  return 0;
}
```

Output:

```
# test-float4.ell
45.400000
```

Operatives

```
/* test-ops1.ell */
int main() {
  print(1 + 2);
  print(1 - 2);
  print(1 * 2);
  print(100 / 2);
  print(99);
  printb(1 == 2);
  printb(1 == 1);
  print(99);
  printb(1 != 2);
  printb(1 != 1);
  print(99);
  printb(1 < 2);
  printb(2 < 1);
  print(99);
  printb(1 <= 2);
  printb(1 <= 1);
  printb(2 <= 1);
  print(99);
  printb(1 > 2);
  printb(2 > 1);
  print(99);
  printb(1 >= 2);
  printb(1 >= 1);
  printb(2 >= 1);
  return 0;
}
```

Output:

```
# test-ops1.out
3
-1
2
50
99
0
1
99
1
0
99
1
0
99
1
1
0
99
0
1
99
0
1
1
```

Operatives with Booleans

```
/* test-ops2.ell */
int main() {
  printb(true);
  printb(false);
  printb(true && true);
  printb(true && false);
  printb(false && true);
  printb(false && false);
  printb(true || true);
  printb(true || false);
  printb(false || true);
  printb(false || false);
  printb(!false);
  printb(!true);
  print(-10);
  print(-42);
}
```

Output:

```
# test-ops2.out
1
0
1
0
0
0
1
1
1
0
1
0
-10
-42
```

Increment and Decrement

```
/* test-pops1.ell */
int main() {
  int i = 1;
  int j = 2;
  i++;
  j--;
  print(i);
  print(j);
  return(0);
}
```

Output:

```
# test-pops1.out
2
1
```

Fibonacci Recursion

```
/* test-fib.ell */
int fib(int x) {
  if (x < 2) return 1;
  return fib(x-1) + fib(x-2);
}

int main() {
  print(fib(0));
  print(fib(1));
  print(fib(2));
  print(fib(3));
  print(fib(4));
  print(fib(5));
  return 0;
}
```

Output:

```
# test-fib.out
1
1
2
3
5
8
```

For Loops

```
/* test-for1.ell */
int main() {
  int i;
  for (i = 0 ; i < 5 ; i = i + 1) {
    print(i);
  }
  print(42);
  return 0;
}
```

Output:

```
# test-for1.out
0
1
2
3
4
42
```

While Loops

```
/* test-while2.ell */
int foo(int a) {
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}

int main() {
  print(foo(7));
  return 0;
}
```

Output:

```
# test-while2.out
14
```

Basic Functions

```
/* test-func7.ell */
int a;

void foo(int c) {
  a = c + 42;
}

int main() {
  foo(73);
  print(a);
  return 0;
}
```

Output:

```
# test-func7.out
```

```
115
```

More Functions

```
/* test-func10.ell */
string test(string a) {
   return a;
}

int main() {
   string b = "hello";
   print_string(test(b));
}
```

Output:

```
# test-func10.out
hello
```

Greatest Common Denominator

```
/* test-gcd2.ell */
int gcd(int a, int b) {
  while (a != b)
    if (a > b) a = a - b;
    else b = b - a;
  return a;
}

int main() {
  print(gcd(14,21));
  print(gcd(8,36));
  print(gcd(99,121));
  return 0;
}
```

Output:

```
# test-gcd2.out
7
4
11
```

Variables

```
/* test-var2.ell */
int a;

void foo(int c) {
  a = c + 42;
}

int main() {
  foo(73);
  print(a);
  return 0;
}
```

```
# test-var2.out
115
```

Global Variables

```
/* test-global3.ell */
int i;
bool b;
int j;

int main() {
  i = 42;
  j = 10;
  print(i + j);
  return 0;
}
```

Output:

```
# test-global3.out
52
```

Local Variables

```
/* test-local2.ell */
int foo(int a, bool b) {
  int c;
  bool d;

  c = a;

  return c + 10;
}

int main() {
 print(foo(37, false));
 return 0;
}
```

Output:

```
# test-local2.out
47
```

Initializing Primitive Typed Variables

```
/* test-init2.ell */
int main() {
  int a = 1;
  int b = a+1;
  double c = 1.5;
  bool d = true;

  print(b);
  print_double(c);
  printb(d);
  return 0;
}
```

Output:

```
# test-init2.out
2
1.500000
1
```

If Statements

```
/* test-if2.ell */
int main() {
  if (true) print(42); else print(8);
  print(17);
  return 0;
}
```

Output:

```
42
17
```

Print Function

```
/* test-printall.ell */
int main() {
  print_all("Hello World");
  print_all(7);
  print_all(100.98);
  print_all(true);
  print_all(-1);
  print_all(100+1);
  print_all('a');
  return 0;
}
```

Output:

```
# test-printall.out
Hello World
7
100.980000
1
-1
101
a
```

Structures

```
/* test-struct2.ell */
/* Test returning structs from functions and assigning the value */
struct Doc {
   string File;
   int Word_count;
};

struct Doc returnDoc() {
  struct Doc doc;
  doc.File = "this is an essay";
  doc.Word_count = 20;
  print_string(doc.File);
  print(doc.Word_count);
  return doc;
}

int main() {
  struct Doc essay = returnDoc();
  print_string(essay.File);
  print(essay.Word_count);
  return 0;
}
```

Output:

```
# test-struct2.out
this is an essay
20
this is an essay
20
```

File Opening and Closing

```
/* test-open2.ell */
int main() {
   file_ptr fp;
   fp = open("tests/open.txt", "wb");
   close(fp);
   print_string("done");
   return(0);
}
```

Output:

```
# test-open2.out
done
```

is_stop_word() Function

```
/* test-stopword.ell */
int main() {
  int test1 = is_stop_word("is");
  int test2 = is_stop_word("plt");
  int test3 = is_stop_word("who");
  int test4 = is_stop_word("didn't");
  print(test1);
  print(test2);
  print(test3);
  print(test4);
}
```

Output:

```
# test-stopword.out
1
0
1
1
```

string_at() Function

```
/* test-string_at.ell */
int main() {
  string a;
  string b = "plt!";
  a = string_at(b, 2, 3, 1);
  print_string(a);
  free(a);
  return 0;
}
```

Output:

```
# test-string_at.out
t
```

Strings

```
/* test-stringinit.ell */
int main() {
   string s = "hello world";
   print_string(s);
   return(0);
}
```

Output:

```
# test-stringinit.out
hello world
```

String Functions

```
/* test-stringfunc2.ell */
int main() {
   string s1 = "hello world";
   string s2 = calloc(1, 30);
   string s3 = calloc(1, 30);
   char x = strget(s1, 2);
   strcpy(s2, s1);
   strcpy(s3, s1);
   strcat(s3, s1);
   print_string(s1);
   print_string(s2);
   print_string(s3);
   print_char(x);
   free(s2);
   free(s3);
   return(0);
}
```

Output:

```
# test-stringfunc2.out
hello world
hello world
hello worldhello world
l
```

word_count() Function

```
/* test-wordcount.ell */
```

```
int main() {
  string essay = "This is a test to see how many words are in this
      essay. There are two sentences.";
  int count = word_count(essay);
  print_string(essay);
  print(count);
}
```

Output:

```
# test-wordcount.out
This is a test to see how many words are in this essay. There are two
    sentences.
17
```

WRITE TO FIILE

```
/* test-write1.ell */
int main() {
   file_ptr fp;
   fp = open("tests/test-write.txt", "wb");
   write("Testing Write", fp);
   close(fp);
   print_string("done");
   return(0);
}
```

Output:

```
# test-write1.out
done
```

7.3 Automation

We used the testing automation provided by Professor Edwards to run all the test scripts at once, with a single shell command. The following demonstrates how to run all test suites via this automation.

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

8 Lessons Learned

8.1 Rabia Akhtar

Understanding the MicroC code and coding in Ocaml will never be easy. You have to drown in the code before you can swim. There is really no way to figure things out otherwise. Use your standard debugging techniques and you'll learn bug by bug. Most of the procrastination happens because people are scared of the code. Also, even if you are not the manager making sure everyone is on track is vital.

As Professor Edwards said in the beginning of the semester, group projects are most successful when people communicate clearly, understand each other, and empathize with others.

8.2 Michele Lin

Gaining an understanding of MicroC before starting the code makes things easier. Not really understanding how each step works results in a lot of guess work. Towards the end of the project, making changes was much faster once the logic of the code made sense.

Planning the project ahead of time is important. It creates a clear idea of the features the project needs, and a time line to mark our progress. Staying on track reduces a lot of stress, and makes sure we don't have to cram working on the project into our finals week.

8.3 Nivita Arora

Put a lot of time into the Hello World part of the project, because that is the stepping stone to everything else that comes down the line. Make sure you fully understand how Professor Edwards' MicroC compiler works and the various parts of the code, before you end up in the depths of building your project itself.

Group projects can also be difficult to navigate, so make sure you have a good team right from the start. One of the main lessons I learned from this project was that everyone's contribution matters, so if one person cannot keep up for whatever reason, someone else will have to do that work. Be sure to also be flexible and accommodating when working in a group dynamic.

8.4 Emily Bau

This project has made me appreciate coding so much more! I've learned all the details and effort that go into making a language from the variables to llvm and everything in-between. Most important lesson learned is planning in the beginning and setting deadlines for yourself is the best way to stay on track. Because before you know it, it's the end of the semester! I also learned that the code base of other projects is the most valuable recourse. The amount of variety in previous projects makes it so you can find pretty much anything in the code base. Reading the code and the different ways groups structured their code taught me a lot about OCaml. Also, writing tests for every little feature ensured reliability in our language. We worked separately on our parts so having tests made sure our own code wouldn't break someone else's. Lastly, have fun with your group!! I really enjoyed the conversations we had and getting to know everyone better :)

8.5 Candace Johnson

Understanding each file in MicroC, what it does, and why its important really helped me have a firm understanding of the basics. When I understood what each file aimed to achieve, I was able to work faster. The biggest challenge I had was merging my branch with master. I often tried to finish a whole new feature before I pushed. I should have incrementally added new changes I made. This would have made it easier to fully integrate new features with our existing code. Since, we were all working on new features at the same time, the master code would change frequently. Thus, when I added a feature, it would result in breaking a lot of tests, and I would have to go through all the code and fix breaking changes. Overall, it was a challenging project, but rewarding once everything started working!

9 Appendix

9.1 scanner.mll

```
(* Authors: Rabia, Michele, Candace, Emily, Nivita *)
(* Ocamllex scanner for MicroC *)

{ open Parser }



rule token = parse
  [’ ’ ’\t’ ’\r’ ’\n’] { token lexbuf } (* Whitespace *)
| "/*"    { comment lexbuf }        (* Comments *)
| ’(’     { LPAREN }
| ’)’     { RPAREN }
| ’{’     { LBRACE }
| ’}’     { RBRACE }
| "{|"    { LINDEX }
| "|}"    { RINDEX }
| "of"    { OF }
| "["     { LSQUARE }
| "]"     { RSQUARE }
| ’;’     { SEMI }
| ’,’     { COMMA }
| ’+’     { PLUS }
| ’-’     { MINUS }
| ’*’     { TIMES }
| ’/’     { DIVIDE }
| "++"    { INCREMENT }
| "--"    { DECREMENT }
| ’=’     { ASSIGN }
| "=="    { EQ }
| "!="    { NEQ }
| ’<’     { LT }
| "<="    { LEQ }
| ">"     { GT }
| ">="    { GEQ }
| "&&"    { AND }
| "||"    { OR }
| ’.’     { DOT }
| "!"     { NOT }
| "if"    { IF }
| "else"  { ELSE }
| "for"   { FOR }
| "while" { WHILE }
| "return" { RETURN }
| "int"   { INT }
| "double" { FLOAT }
| "bool"  { BOOL }
| "void"  { VOID }
| "true"  { TRUE }
| "false" { FALSE }
| "string" { STRING }
| "char"  { CHAR }
| "file_ptr" { STRING }
| "struct" { STRUCT }
| ['0'-'9']+ as lxm { NUM_LIT(int_of_string lxm) }
| ['0'-'9']+'.'['0'-'9']* | ['0'-'9']*'.'['0'-'9']+
  as lxm { FLOAT_LIT(float_of_string lxm)}
| '"' (([^ '"'] | "\\\"")* as strlit) '"' { STRING_LIT(strlit) }
| '''([' '-'!' '#'-'[' ']'-'~' ]|['0'-'9'])''' as lxm {CHAR_LITERAL(
    String.get lxm 1)}
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

9.2 parser.mly

```
/* Authors: Rabia, Michele, Candace, Emily, Nivita */
/* Ocamlyacc parser for MicroC */

%{
open Ast

let fst (a,_,_) = a;;
let snd (_,b,_) = b;;
let trd (_,_,c) = c;;

%}

%token SEMI LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT DECREMENT INCREMENT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR DOT

%token RETURN IF ELSE FOR WHILE INT FLOAT BOOL VOID LENGTH
%token INT CHAR FLOAT BOOL VOID STRING OF STRUCT TRUE FALSE LINDEX RINDEX
%token <int> NUM_LIT
%token <float> FLOAT_LIT
%token <string> STRING_LIT
%token <char> CHAR_LITERAL
%token <string> ID
%token EOF
```

```

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN

%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left DOT
%right NOT NEG
%left LINDEX

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

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

fdecl:
   typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { typ = $1;
    fname = $2;
    formals = $4;
    locals = List.rev $7;
    body = List.rev $8 } }

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

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

dtyp:
  INT { Int }
| STRING {String}
| FLOAT {Float}
| CHAR {Char}
```

```

atyp:
  dtyp dim_list { Array($1, $2) }

typ:
    dtyp { Simple($1)}
  | atyp { $1 }
  | BOOL { Bool }
  | VOID { Void}
  | STRUCT ID { Struct ($2) }

dim_list:
  LSQUARE RSQUARE { 1 }
| LSQUARE RSQUARE dim_list { 1 + $3 }



index:
| LINDEX expr RINDEX { $2}

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

vdecl:
    typ ID SEMI          { VarDecl($1, $2, Noexpr) }
  | typ ID ASSIGN expr SEMI { VarDecl($1, $2, $4) }


sdecl:
    STRUCT ID LBRACE vdecl_list RBRACE SEMI
      {
        { sname = $2;
          sformals = $4;
      }
    }

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

stmt:
    expr SEMI { Expr $1 }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr 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 }

id:
  ID             { Id($1) }

val_list:
    expr              { [ $1 ] }
  | expr COMMA val_list { [ $1 ] @ $3 }

simple_arr_literal:
    LSQUARE val_list RSQUARE { $2 }


expr:
    NUM_LIT        { NumLit($1) }
  | FLOAT_LIT      { FloatLit($1) }
  | STRING_LIT     { StringLit($1) }
  | CHAR_LITERAL   { CharLit($1)}
  | simple_arr_literal { ArrayLit($1)}
  | expr index     { Index($1, [$2]) }
  | TRUE           { BoolLit(true) }
  | FALSE          { BoolLit(false) }
  | ID            { Id($1) }
  | id INCREMENT { Pop($1, Inc) }
  | id DECREMENT { Pop($1, Dec) }
  | 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 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) }
  | MINUS expr %prec NEG { Unop(Neg, $2) }
  | NOT expr       { Unop(Not, $2) }
  | expr ASSIGN expr { Assign($1, $3) }
  | expr DOT ID { Dot($1,      $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | ID LSQUARE expr RSQUARE ASSIGN expr { ArrayAssign($1, [$3], $6) }
  | LPAREN expr RPAREN { $2 }

```

```

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

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

9.3 ast.ml

```
(* Authors: Rabia, Michele, Candace, Emily, Nivita *)
(* Abstract Syntax Tree and functions for printing it *)

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

type pop =
  | Dec
  | Inc

type dtyp = Int | String | Float |Char

type uop = Neg | Not

type typ = Simple of dtyp

    | Bool
    | Void
    | Array of dtyp * int
    | Struct of string

type bind = typ * string

type expr =
    (* Literal of int *)
    NumLit of int
  | FloatLit of float
  | BoolLit of bool
  | StringLit of string
  | ArrayLit of expr list
  | Index of expr * expr list
  | StructLit of string
  | CharLit of char
  | Id of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Pop of expr * pop
  | Assign of expr * expr
  | ArrayAccess of string * expr
  | ArrayAssign of string * expr list * expr
  | Call of string * expr list
  | Dot of expr * string
  | Noexpr

type var_decl = VarDecl of typ * string * expr

type struct_decl = {
    sname: string;
    sformals: var_decl list;
 }

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 : var_decl list;
    body : stmt list;
  }

type program = var_decl list * func_decl list * struct_decl list

(* Pretty-printing functions *)

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

let string_of_uop = function
```

```
  Neg -> "-"
 | Not -> "!"

let string_of_pop = function
  Inc -> "++"
 | Dec -> "--"

let convert_array l conversion joiner =
  let glob_item original data = original ^ (conversion data) ^ joiner
    in
  let full = (List.fold_left glob_item "" l) in
  "[" ^ String.sub full 0 ((String.length full) - 2) ^ "]"

let rec string_of_expr = function
  NumLit(l) -> string_of_int l
 | FloatLit(f) -> string_of_float f
 | BoolLit(true) -> "true"
 | BoolLit(false) -> "false"
 | StringLit(s) -> s
 | ArrayLit(l) -> convert_array l string_of_expr ", "
 | Index(e, l) -> string_of_expr e ^
         "{|" ^ string_of_expr (List.hd l) ^ "|}"
 | StructLit(s) -> "Struct " ^ s
 | CharLit(s) -> Char.escaped s
 | 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
 | Pop(v, p) -> string_of_expr v ^ string_of_pop p
 | Assign(v, e) -> string_of_expr v ^ " = " ^ string_of_expr e
 | Dot(e, s) -> string_of_expr e ^ "." ^ s
 | Call(f, el) ->
   f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
 | Noexpr -> ""
 | ArrayAccess(s,e2) -> (s) ^ "[" ^ (string_of_expr e2) ^ "]"
 | ArrayAssign(v, l, e) -> v ^ "[" ^ string_of_expr (List.hd l) ^ "]" ^
   " = " ^ string_of_expr e

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_d_typ = function
  Int -> "int"
| String -> "string"
| Float -> "double"
| Char -> "char"

let rec repeat c = function
    0 -> ""
  | n -> c ^ (repeat c (n - 1))

let string_of_typ = function
    Bool -> "bool"
  | Void -> "void"
  | Simple(d) -> string_of_d_typ d
  | Array(d,n) -> string_of_d_typ d ^ repeat "[]" n
  | Struct(id) -> "struct" ^ id

let string_of_vdecl = function
  VarDecl(t, id, e) -> string_of_typ t ^ " " ^ id ^ "=" ^ string_of_expr
      e ^ ";\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_sdecl sdecl =
 "struct " ^ sdecl.sname ^ String.concat
 "{\n" (List.map string_of_vdecl sdecl.sformals) ^ "\n}\n"

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

9.4 semant.ml

```
(* Authors: Rabia, Michele, Candace, Emily, Nivita *)
(* Semantic checking for the ELL compiler *)

open Ast
module A = Ast

module StringMap = Map.Make(String)
module StringSet = Set.Make(String)

(* Semantic checking of a program. Returns void if successful,
   throws an exception if something is wrong.
   Check each global variable, then check each function *)

let check (globals, functions, structs) =

  (* Raise an exception if the given list has a duplicate *)
  let report_duplicate exceptf list =
    let rec helper = function
   n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
      | _ :: t -> helper t
      | [] -> ()
    in helper (List.sort compare list)
  in

  (* Check struct name and recursive definition *)
  let find_sdecl_from_sname struct_t_name =
    try List.find (fun t-> t.sname= struct_t_name) structs
      with Not_found -> raise (Failure("Struct " ^ struct_t_name ^ "not
          found"))
  in
  let rec check_rec_struct_h sdecl structs_known_set =
    let check_for_repetition struct_t_name =
      if StringSet.mem struct_t_name structs_known_set
      then raise (Failure ("recursive struct definition"))
      else check_rec_struct_h (find_sdecl_from_sname struct_t_name)
      (StringSet.add struct_t_name structs_known_set)
    in
    let struct_field_check = function
      (Struct s, _) -> check_for_repetition s
      | _ -> ()
    in
    let sformals_list = List.map (fun (VarDecl(t, n, _)) -> (t, n))
        sdecl.sformals in
    List.iter (struct_field_check) sformals_list
  in
  let check_recursive_struct sdecl =
     check_rec_struct_h sdecl StringSet.empty
  in
  let _ = List.map check_recursive_struct structs
  in

  (* Raise an exception if a given binding is to a void type *)
  let check_not_void_f exceptf = function
      (Void, n) -> raise (Failure (exceptf n))
    | _ -> ()
  in
```

```
 let check_not_void_v exceptf = function
   (VarDecl(Void, n,_)) -> raise (Failure (exceptf n))
  | _ -> ()
in

(* Raise an exception of the given rvalue type cannot be assigned to
   the given lvalue type *)
(* let check_assign lvaluet rvaluet err =
   if lvaluet == rvaluet then lvaluet else raise err
in  *)

let resolve_struct_access sname field =
  let  s = try List.find (fun t -> t.sname = sname) structs
    with Not_found -> raise (Failure("Struct " ^ sname ^ " not
        found")) in
  let sformals = List.map (fun (VarDecl(t, n, _)) -> (t, n))
      s.sformals in
  try fst( List.find (fun s -> snd(s) = field) sformals) with
Not_found -> raise (Failure("Field " ^ field ^ " not found in Struct"
    ^ sname))
in

let check_access lhs rhs =
   match lhs with
    Struct s -> resolve_struct_access s rhs
    | _ -> raise (Failure(string_of_typ lhs^ " is not a struct"))

in

(* Check function declrations *)
let check_func_decl func_name =
  if List.mem func_name (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function may not be defined as " ^ func_name))
in

(* check all reserved function names *)
check_func_decl "printb";
check_func_decl "printbig";
check_func_decl "print_double";
check_func_decl "print_all";
check_func_decl "open";
check_func_decl "close";
check_func_decl "read";
check_func_decl "write";
check_func_decl "strlen";
check_func_decl "strcmp";
check_func_decl "strcat";
check_func_decl "strcpy";
check_func_decl "strget";
check_func_decl "to_lower";
check_func_decl "calloc";
check_func_decl "free";
check_func_decl "print_char";
check_func_decl "is_stop_word";
check_func_decl "word_count";
check_func_decl "print_string";
check_func_decl "string_at";


(**** Checking Global Variables ****)

List.iter (check_not_void_v (fun n -> "illegal void global " ^ n))
    globals;

report_duplicate (fun n -> "duplicate global " ^ n)
  (List.map (fun (VarDecl(_,n,_)) -> n) globals);

(* allowed initiation types *)
let globalInitTyps = function
    NumLit _ -> A.Simple(A.Int)
    | FloatLit _ -> A.Simple(A.Float)
    | BoolLit _ -> Bool
    | StringLit _ -> A.Simple(A.String)
    | CharLit _ -> A.Simple(A.Char)
    | StructLit s -> Struct s
    | _ -> raise (Failure ("Illegal global initialization"))
in

let check_type lvaluet rvaluet err =
   if (String.compare (string_of_typ lvaluet) (string_of_typ rvaluet))
       == 0 then lvaluet else raise err
in

let checkGlobalInit = function
  VarDecl(t,n,e) -> if e != Noexpr then
    let typ = globalInitTyps e in
      ignore (check_type t typ(Failure ("Global initialization type
          does not match " ^ n ^ " " ^ string_of_expr e)))
in

(* check assignment types *)
List.iter checkGlobalInit globals;

(**** Checking Functions ****)

if List.mem "print" (List.map (fun fd -> fd.fname) functions)
then raise (Failure ("function print may not be defined")) else ();

report_duplicate (fun n -> "duplicate function " ^ n)
```

```
  (List.map (fun fd -> fd.fname) functions);

(* Function declaration for a named function *)
let built_in_decls =

    StringMap.add "print"
   { typ = Void; fname = "print"; formals = [(A.Simple(A.Int), "x")];
     locals = []; body = [] }

     (StringMap.add "printb"
   { typ = Void; fname = "printb"; formals = [(Bool, "x")];
     locals = []; body = [] }

      (StringMap.add "printbig"
   { typ = Void; fname = "printbig"; formals = [(A.Simple(A.Int),
       "x")];
     locals = []; body = [] }

      (StringMap.add "print_double"
   { typ = Void; fname = "print_double"; formals =
       [(A.Simple(A.Float), "x")];
     locals = []; body = [] }

     (StringMap.add "print_all"
   { typ = Void; fname = "print_all"; formals = [(A.Simple(A.String),
       "x")];
     locals = []; body = [] }

     (StringMap.add "open"
   { typ = A.Simple(A.String); fname = "open"; formals =
   [(A.Simple(String), "x"); (A.Simple(A.String), "y")]; locals = [];
       body = []}

     (StringMap.add "close"
   { typ = Void; fname = "close"; formals =
   [(A.Simple(A.String), "x")]; locals = []; body = []}

     (StringMap.add "read"
   { typ = A.Simple(A.Int); fname = "read"; formals =
   [(A.Simple(A.String), "a"); (A.Simple(A.Int), "b");
       (A.Simple(A.Int), "c"); (A.Simple(A.String), "d")];
     locals = []; body = [] }

     (StringMap.add "write"
   { typ = A.Simple(Int); fname = "write"; formals =
   [(A.Simple(String), "x"); (A.Simple(String), "y")];
     locals = []; body = [] }


     (StringMap.add "strlen"
   { typ = A.Simple(A.Int); fname = "strlen"; formals =
   [(A.Simple(A.String), "x")];

     locals = []; body = [] }

     (StringMap.add "strcmp"
   { typ = A.Simple(A.Int); fname = "strcmp"; formals =
   [(A.Simple(A.String), "x"); (A.Simple(A.String), "x")];
     locals = []; body = [] }

      (StringMap.add "strcat"
   { typ = A.Simple(A.String); fname = "strcat"; formals =
   [(A.Simple(A.String), "x"); (A.Simple(A.String), "x")];
     locals = []; body = [] }

      (StringMap.add "strcpy"
   { typ = A.Simple(A.String); fname = "strcpy"; formals =
   [(A.Simple(A.String), "x"); (A.Simple(A.String), "x")];
     locals = []; body = [] }

      (StringMap.add "strget"
   { typ = A.Simple(A.Char); fname = "strcat"; formals =
   [(A.Simple(A.String), "x"); (A.Simple(A.Int), "y")];
     locals = []; body = [] }

     (StringMap.add "to_lower"
   { typ = A.Simple(A.Char); fname = "to_lower"; formals =
   [(A.Simple(A.Char), "x")];
     locals = []; body = [] }

     (StringMap.add "calloc"
   { typ = A.Simple(A.String); fname = "calloc"; formals =
   [(A.Simple(A.Int), "x"); (A.Simple(A.Int), "x")];
     locals = []; body = [] }

      (StringMap.add "free"
   { typ = A.Simple(A.String); fname = "free"; formals =
   [(A.Simple(A.String), "x") ];
     locals = []; body = [] }

     (StringMap.add"print_char"
   { typ = Void; fname = "print_char"; formals = [(A.Simple(A.Char), "x")];
     locals = []; body = [] }

       (StringMap.add"is_stop_word"
   { typ = A.Simple(A.Int); fname = "is_stop_word"; formals = [(A.Simple(A.String), "x")];
     locals = []; body = [] }

```

```
         (StringMap.add"string_at"
    { typ = A.Simple(A.String); fname = "string_at"; formals = [(A.Simple(A.String), "x"); (A.Simple(A.Int), "x"); (A.Simple(A.Int), "x"); (A.Simple(A.Int), "x")];
      locals = []; body = [] }

       (StringMap.add"word_count"
    { typ = A.Simple(A.Int); fname = "word_count"; formals = [(A.Simple(A.String), "x")];
      locals = []; body = [] }


    (StringMap.singleton "print_string"
    { typ = Void; fname = "print_string"; formals = [(A.Simple(A.String), "x")];
      locals = []; body = [] }))))))))))))))))))))))


  in

 (* Accepted types for print_all *)
 let print_types = [A.Simple(String); A.Simple(Int); Bool; A.Simple(Float); A.Simple(A.Char)] in

 let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
                     built_in_decls functions
 in

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

 (* let struct_decls = List.fold_left (fun m st -> StringMap.add st.sname st m)
                      StringMap.empty structs *)

 let check_type lvaluet rvaluet err =
    if (String.compare (string_of_typ lvaluet) (string_of_typ rvaluet)) == 0 then lvaluet else raise err
 in

(* let struct_decl s = try StringMap.find s struct_decls
     with Not_found -> raise (Failure ("unrecognized struct" ^ s)) *)


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

 let check_function func =

    List.iter (check_not_void_f (fun n -> "illegal void formal " ^ n ^
      " in " ^ func.fname)) func.formals;

    report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^
        func.fname)
      (List.map snd func.formals);

    List.iter (check_not_void_v (fun n -> "illegal void local " ^ n ^
      " in " ^ func.fname)) func.locals;

    report_duplicate (fun n -> "duplicate local " ^ n ^ " in " ^
        func.fname)
      (List.map (fun (VarDecl(_,n,_)) -> n) func.locals);

    (* Type of each variable (global, formal, or local *)
    let var_symbols = List.fold_left (fun m (t, n) -> StringMap.add n t
        m)
       StringMap.empty func.formals in

    let symbols = List.fold_left (fun m (VarDecl(t,n,_)) ->
        StringMap.add n t m)
       var_symbols (globals @ func.locals) in

    let type_of_identifier s =
      try StringMap.find s symbols
      with Not_found -> raise (Failure ("undeclared identifier " ^ s))
    in

    let array_access_type = function
      Array(t,_) -> Simple(t)
      | _ -> raise(Failure("Can only access a[x] from an array a"))
    in

    (* Return the type of an expression or throw an exception *)
    let rec expr = function
     NumLit _ -> A.Simple(A.Int)
      | FloatLit _ -> A.Simple(A.Float)
      | BoolLit _ -> Bool
      | CharLit _ -> A.Simple(A.Char)
      | StringLit _ -> A.Simple(A.String)
      | ArrayLit(l) -> let first_type = expr (List.hd l) in
                       let _ = (match first_type with
                                 Simple _ -> ()
                               | _ -> raise (Failure ("’" ^
                                   string_of_expr (List.hd l) ^ "’ is
                                   not simple and is in array"))
                              ) in
                       let _ = List.iter (fun x -> if string_of_typ(expr
                           x) == string_of_typ first_type then ()
```

```
                                              else raise (Failure ("'"
                                                  ^ string_of_expr x ^
                                                  "' doesn't match
                                                  array's type"))) l in
                     Array((match first_type with Simple(x) -> x
                            | _ -> raise(Failure("not array type"))),
                                1)
    | ArrayAccess(s, e1) -> let _ = (match (expr e1) with
                                  Simple(Int) -> Simple(Int) (* ||
                                      A.Simple(A.String) ->
                                      A.Simple(A.String) ||
                                      A.Simple(A.Float) ->
                                      A.Simple(A.Float) *)
                                 | _ -> raise (Failure ("attempting
                                     to access with a non integer
                                     type"))) in
                         array_access_type (type_of_identifier s)
    | Index (a, i) -> if string_of_typ(expr (List.hd i)) !=
        string_of_typ(Simple(Int))
                   then raise ( Failure("Array index ('" ^
                       string_of_expr (List.hd i) ^ "') is not an
                       integer") )
                   else
                     let type_of_entity = expr a in
                     (match type_of_entity with
                        Array(d, _) -> Simple(d)
                      | Simple(String) -> Simple(String)
                      | _ -> raise (Failure ("Entity being indexed ('"
                          ^ string_of_expr a ^"') cannot be array")))

    | StructLit s -> Struct s
    | Id s -> type_of_identifier s
    | ArrayAssign(v, i, e) as ex -> let type_of_left_side =
                                if string_of_typ(expr (List.hd i)) !=
                                    string_of_typ(Simple(Int))
                                then raise ( Failure("Array index ('"
                                    ^ string_of_expr (List.hd i) ^
                                    "') is not an integer") )
                                else
                                  let type_of_entity =
                                      type_of_identifier v in
                                  (match type_of_entity with
                                     Array(d, _) -> Simple(d)
                                   | _ -> raise (Failure ("Entity
                                       being indexed ('" ^ v ^"')
                                       cannot be array"))) in
                                let type_of_right_side = expr e in
                                check_type type_of_left_side
                                    type_of_right_side
```

```
                              (Failure ("illegal assignment " ^
                                  string_of_typ type_of_left_side ^
                                      " = " ^ string_of_typ
                                           type_of_right_side ^ "
                                           in " ^
                                      string_of_expr ex))
 | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
    (match op with
     Add | Sub | Mult | Div when t1 = A.Simple(A.Int) && t2 =
         A.Simple(A.Int) -> A.Simple(A.Int)
   | Add | Sub | Mult | Div when t1 = A.Simple(A.Float) && t2 =
       A.Simple(A.Float) -> A.Simple(A.Float)
   | Add | Sub | Mult | Div when t1 = A.Simple(A.Char) && t2 =
       A.Simple(A.Char) -> A.Simple(A.Char)
    | Equal | Neq when t1 = t2 -> Bool
    | Less | Leq | Greater | Geq when t1 = A.Simple(A.Int) && t2 =
         A.Simple(Int) -> Bool
   | Less | Leq | Greater | Geq when t1 = A.Simple(A.Float) && t2
       = A.Simple(A.Float) -> Bool
    | And | Or when t1 = Bool && t2 = Bool -> Bool
   | _ -> raise (Failure ("illegal binary operator " ^
        string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
        string_of_typ t2 ^ " in " ^ string_of_expr e))
  )
 | Dot(e, field) -> check_access (expr e) field
 | Unop(op, e) as ex -> let t = expr e in
   (match op with
      Neg when t = A.Simple(A.Int) -> A.Simple(A.Int)
    | Neg when t = A.Simple(A.Float) -> A.Simple(A.Float)
    | Not when t = Bool -> Bool
    | _ -> raise (Failure ("illegal unary operator " ^
        string_of_uop op ^
      string_of_typ t ^ " in " ^ string_of_expr ex)))
| Pop(e, op) as ex -> let t = expr e in
   (match op with
    | Inc | Dec -> (match t with
                    A.Simple(A.Int) -> A.Simple(A.Int)
                  | _ -> raise (Failure ("illegal postfix operator
                      " ^ string_of_pop op ^ " used with a " ^
                                     string_of_typ t ^ " in " ^
                                         string_of_expr ex)))

  )
 | Noexpr -> Void
 | Assign(var, e) as ex -> let lt = expr var
                        and rt = expr e in
  check_type lt rt (Failure ("illegal assignment " ^ string_of_typ
      lt ^
          " = " ^ string_of_typ rt ^ " in " ^
          string_of_expr ex))
  | Call(fname, actuals) as call -> let fd = function_decl fname in
     if List.length actuals != List.length fd.formals then
       raise (Failure ("expecting " ^ string_of_int
         (List.length fd.formals) ^ " arguments in " ^ string_of_expr call))
     else
       let _ =
           (match fname with
             "print_all" ->
               ignore (List.iter (fun e ->
                 let etyp = expr e in
                 if (List.mem etyp print_types) == false then
                   raise (Failure ("illegal actual argument found " ^ string_of_typ etyp ^ " in " ^ string_of_expr e))) actuals);
             | _ ->
             List.iter2 (fun (ftyp, _) e ->
               let etyp = expr e in
               ignore (check_type ftyp etyp (Failure ("illegal actual argument found " ^ string_of_typ etyp ^ " expected " ^ string_of_typ ftyp ^ " in " ^ string_of_expr e)))
             ) fd.formals actuals
          ) in
        fd.typ
 in

 let check_bool_expr e = if expr e != Bool
  then raise (Failure ("expected Boolean expression in " ^ string_of_expr e))
  else () in

 (* Verify a statement or throw an exception *)
 let rec stmt = function
Block sl -> let rec check_block = function
       [Return _ as s] -> stmt s
     | Return _ :: _ -> raise (Failure "nothing may follow a return")
     | Block sl :: ss -> check_block (sl @ ss)
     | s :: ss -> stmt s ; check_block ss
     | [] -> ()
    in check_block sl
  | Expr e -> ignore (expr e)
  | Return e -> let t = expr e in if t = func.typ then () else
     raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                    string_of_typ func.typ ^ " in " ^ string_of_expr e))

  | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2
  | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
```

```
                            ignore (expr e3); stmt st
      | While(p, s) -> check_bool_expr p; stmt s
    in

    let check_var_init = function
      VarDecl(t,_,e) as ex -> if e != Noexpr then
       let v = expr e in
         ignore (check_type t v(Failure ("illegal initialization of" ^
             string_of_typ t ^
            " = " ^ string_of_typ v ^ " in " ^ string_of_vdecl ex))) in

    stmt (Block func.body);
    List.iter check_var_init func.locals
  in
  List.iter check_function functions
```

9.5 codegen.ml

```
(* Authors: Rabia, Michele, Candace, Emily, Nivita *)
(* 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

module StringMap = Map.Make(String)
module String = String

let translate (globals, functions, structs) =
  let context = L.global_context () in
  let the_module = L.create_module context "English"
  and i32_t = L.i32_type context
  and i8_t  = L.i8_type  context
  and p_t = L.pointer_type (L.i8_type (context))
  and i1_t  = L.i1_type  context
  and f_t   = L.double_type context
  and void_t = L.void_type context
in

  let rec int_range = function
      0 -> [ ]
    | 1 -> [ 0 ]
    | n -> int_range (n - 1) @ [ n - 1 ] in


let struct_type_table:(string, L.lltype) Hashtbl.t = Hashtbl.create 10
in

let make_struct_type sdecl =
  let struct_t = L.named_struct_type context sdecl.A.sname in
  Hashtbl.add struct_type_table sdecl.A.sname struct_t in
  let _ = List.map make_struct_type structs
in

let lookup_struct_type sname = try Hashtbl.find struct_type_table sname
  with Not_found -> raise(Failure("Struct name not found"))
in
let rec ltype_of_typ = function
      A.Simple(A.Int) -> i32_t
    | A.Simple(A.Float) -> f_t
    | A.Bool -> i1_t
    | A.Void -> void_t
    | A.Simple(A.String) -> p_t
    | A.Array(d, _) -> L.struct_type context [| i32_t ; L.pointer_type
        (ltype_of_typ (A.Simple(d))) |]
    | A.Simple(A.Char) -> i8_t
    | A.Struct(sname) -> lookup_struct_type sname
    in

(* Define structs and fill hashtable *)
let make_struct_body sdecl =
  let struct_typ = try Hashtbl.find struct_type_table sdecl.A.sname
    with Not_found -> raise(Failure("struct type not defined")) in
  let sformals_types = List.map (fun (A.VarDecl(t, _, _)) -> t)
      sdecl.A.sformals in
  let sformals_lltypes = Array.of_list (List.map ltype_of_typ
      sformals_types) in
  L.struct_set_body struct_typ sformals_lltypes true
in  ignore(List.map make_struct_body structs);

let struct_field_indices =
  let handles m one_struct =
    let struct_field_names = List.map (fun (A.VarDecl(_, n, _)) -> n)
        one_struct.A.sformals in
    let add_one n = n + 1 in
    let add_fieldindex (m, i) field_name =
     (StringMap.add field_name (add_one i) m, add_one i) in
    let struct_field_map =
     List.fold_left add_fieldindex (StringMap.empty, -1)
          struct_field_names
    in
    StringMap.add one_struct.A.sname (fst struct_field_map) m
```

```
in
List.fold_left handles StringMap.empty structs
in


(* Declare printf(), which the print built-in function will call *)
let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func = L.declare_function "printf" printf_t the_module in

(* Declare the built-in printbig() function *)
let printbig_t = L.function_type i32_t [| i32_t |] in
let printbig_func = L.declare_function "printbig" printbig_t the_module in

(* Declare the built-in open() function *)
let open_t = L.function_type p_t [| L.pointer_type i8_t; L.pointer_type i8_t |] in
let open_func = L.declare_function "fopen" open_t the_module in

(* Declare the built-in close() function *)
let close_t = L.function_type i32_t [| p_t |] in
let close_func = L.declare_function "fclose" close_t the_module in

(* Declare the built-in fputs() function as write() *)
let write_t = L.function_type i32_t [| L.pointer_type i8_t; p_t |] in
let write_func = L.declare_function "fputs" write_t the_module in

(* Declare the built-in fread() function as read() *)
let read_t = L.function_type i32_t [| p_t; i32_t; i32_t; p_t |] in
let read_func = L.declare_function "fread" read_t the_module in

(* Declare the built-in strlen() function *)
let strlen_t = L.function_type i32_t [| p_t |] in
let strlen_func = L.declare_function "strlen" strlen_t the_module in

(* Declare the built-in strcmp() function *)
let strcmp_t = L.function_type i32_t [| p_t; p_t|] in
let strcmp_func = L.declare_function "strcmp" strcmp_t the_module in

(* Declare the built-in strcat() function *)
let strcat_t = L.function_type p_t [| p_t; p_t|] in
let strcat_func = L.declare_function "strcat" strcat_t the_module in

(* Declare the built-in strcpy() function *)
let strcpy_t = L.function_type p_t [| p_t; p_t|] in
let strcpy_func = L.declare_function "strcpy" strcpy_t the_module in

(* Declare the built-in strget() function *)
let strget_t = L.function_type i8_t [| p_t; i32_t|] in
let strget_func = L.declare_function "strget" strget_t the_module in

(* Declare c code as string_lower() *)
let to_lower_t = L.function_type i8_t [| i8_t |] in
let to_lower_func = L.declare_function "char_lower" to_lower_t
    the_module in

(* Declare c code as is_stop_word() *)
let is_stop_word_t = L.function_type i32_t [| p_t |] in
let is_stop_word_func = L.declare_function "is_stop_word"
    is_stop_word_t the_module in

(* Declare c code as is_stop_word() *)

let word_count_t = L.function_type i32_t [| p_t |] in
let word_count_func = L.declare_function "word_count" word_count_t
    the_module in

let string_at_t = L.function_type p_t [| p_t; i32_t; i32_t; i32_t|] in
let string_at_func = L.declare_function "string_at" string_at_t
    the_module in

(* Declare heap storage function *)
let calloc_t = L.function_type p_t [| i32_t ; i32_t|] in
let calloc_func = L.declare_function "calloc" calloc_t the_module in

(* Declare free from heap *)
let free_t = L.function_type p_t [| p_t |] in
let free_func = L.declare_function "free" free_t the_module in

let int_format_str builder = L.build_global_stringptr "%d\n" "fmt"
    builder in
let float_format_str builder = L.build_global_stringptr "%f\n" "fmt"
    builder in
let string_format_str builder = L.build_global_stringptr "%s\n" "fmt"
    builder in
let char_format_str builder = L.build_global_stringptr "%c\n" "fmt"
    builder in


(* Return the value for a variable or formal argument *)
let lookup g_map l_map n = try StringMap.find n l_map
     with Not_found -> StringMap.find n g_map in

(* Define each function (arguments and return type) so we can call it
    *)
let function_decls =
  let function_decl m fdecl =
    let name = fdecl.A.fname
    and formal_types =
Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.A.formals)
    in let ftype = L.function_type (ltype_of_typ fdecl.A.typ)
      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

let format_str x_type builder =
  let b = builder in
    match x_type with
     A.Simple(A.Int)  -> int_format_str b
    | A.Simple(A.Float)  -> float_format_str b
    | A.Simple(A.String) -> string_format_str b
    | A.Bool    -> int_format_str b
    | A.Simple(A.Char)   -> char_format_str b
    | _ -> raise (Failure ("Invalid printf type"))
in

(* get type *)
let rec gen_type g_map l_map = function
    A.NumLit _ -> A.Simple(A.Int)
  | A.FloatLit _ -> A.Simple(A.Float)
  | A.StringLit _ -> A.Simple(A.String)
  | A.BoolLit _ -> A.Bool
  | A.CharLit _ -> A.Simple(A.Char)
  | A.Unop(_,e) -> (gen_type g_map l_map) e
  | A.Binop(e1,_,_) -> (gen_type g_map l_map) e1
  | A.Noexpr -> A.Void
  | _ -> raise (Failure ("Type not found"))

in

let get_init_val = function
     A.NumLit i -> L.const_int i32_t i
    | A.FloatLit f -> L.const_float f_t f
    | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
    | A.StringLit s -> let l = L.define_global "" (L.const_stringz
      context s) the_module in
    L.const_bitcast (L.const_gep l [|L.const_int i32_t 0|]) p_t
    | A.CharLit c -> L.const_int i8_t (Char.code c)
    | A.Noexpr -> L.const_int i32_t 0
    | _ -> raise (Failure ("not found"))
in

let get_init_noexpr = function
     A.Simple(A.Int) -> L.const_int i32_t 0
    | A.Simple(A.Float) -> L.const_float f_t 0.0
    | A.Bool -> L.const_int i1_t 0
    | A.Simple(A.Char) -> L.const_int i8_t 0
    | A.Simple(A.String) -> get_init_val(A.StringLit "")
```

```
     | A.Array(d, _) -> L.const_null (L.struct_type context [| i32_t ; L.pointer_type (ltype_of_typ (A.Simple(d))) |])
     | A.Struct(sname) -> L.const_named_struct (lookup_struct_type sname) [||]
     | _ -> raise (Failure ("not found"))
 in

let build_array_access g_map l_map s i1 i2 builder isAssign =
     if isAssign
        then L.build_gep(lookup g_map l_map s) [| i1; i2 |] s builder
        else L.build_load (L.build_gep(lookup g_map l_map s) [| i1; i2|] s builder) s builder
   in


(* Declare each global variable; remember its value in a map *)
 let global_vars =
   let global_var m (A.VarDecl(_, n, e)) =
     let init = get_init_val e in
     StringMap.add n (L.define_global n init the_module) m in
   List.fold_left global_var StringMap.empty globals in

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

 (* Return addr of lhs expr *)
 let addr_of_expr expr builder g_map l_map = match expr with
   A.Id(id) -> (lookup g_map l_map id)
 | A.StructLit (s) ->(lookup g_map l_map s)
 | A.Dot (e1, field) ->
      (match e1 with
     A.Id s -> let etype = fst(
       let fdecl_locals = List.map (fun (A.VarDecl(t, n, _)) -> (t, n)) fdecl.A.locals in
       try List.find (fun n -> snd(n) = s) fdecl_locals
       with Not_found -> raise (Failure("Unable to find" ^ s )))
       in
       (try match etype with
         A.Struct t->
           let index_number_list = StringMap.find t struct_field_indices in
           let index_number = StringMap.find field index_number_list in
           let struct_llvalue = lookup g_map l_map s in
           let access_llvalue = L.build_struct_gep struct_llvalue index_number "tmp" builder in
           access_llvalue
       | _ -> raise (Failure("not found"))
      with Not_found -> raise (Failure("not found" ^ s)))
      | _ -> raise (Failure("lhs not found")))
  | _ -> raise (Failure("addr not found"))

 in


 (* Construct code for an expression; return its value *)
 let rec expr builder g_map l_map = function
       A.NumLit i -> L.const_int i32_t i
     | A.FloatLit f -> L.const_float f_t f
     | A.StringLit s -> L.build_global_stringptr s "tmp" builder
     | A.CharLit c -> L.const_int i8_t (Char.code c)
     | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
     | A.StructLit t -> (lookup g_map l_map t)
     | A.Noexpr -> L.const_int i32_t 0
     | A.Id s -> L.build_load (lookup g_map l_map s) s builder
     | A.ArrayAccess(s, ind1) ->
       let i = expr builder g_map l_map ind1 in
         build_array_access g_map l_map s(L.const_int i32_t 0) i builder false
     | A.ArrayLit(l) -> let size = L.const_int i32_t (List.length l) in
                        let all = List.map (fun e -> expr builder g_map l_map e) l in
                        let new_array = L.build_array_malloc (L.type_of (List.hd all)) size "tmp" builder in
                        List.iter (fun x ->
                           let more = (L.build_gep new_array [| L.const_int i32_t x |] "tmp2" builder) in
                           let intermediate = List.nth all x in
                           ignore (L.build_store intermediate more builder)
                        ) (int_range (List.length l)) ;
                        let type_of_new_literal = L.struct_type context [| i32_t ; L.pointer_type (L.type_of (List.hd all)) |] in
                        let new_literal = L.build_malloc type_of_new_literal "arr_literal" builder in
                        let first_store = L.build_struct_gep new_literal 0 "first" builder in
                        let second_store = L.build_struct_gep new_literal 1 "second" builder in
                        ignore (L.build_store size first_store builder);
                        ignore (L.build_store new_array second_store builder);
                        let actual_literal = L.build_load new_literal "actual_arr_literal" builder in
                        actual_literal
     | A.ArrayAssign(v, i, e) -> let e' = expr builder g_map l_map e in
```

```
                                 let i' = expr builder g_map l_map
                                     (List.hd i) in
                                 let v' = L.build_load (lookup g_map l_map
                                     v) v builder in
                                 let extract_array = L.build_extractvalue
                                     v' 1 "extract_ptr" builder in
                                 let extract_value = L.build_gep
                                     extract_array [| i' |]
                                     "extract_value" builder in
                                 ignore (L.build_store e' extract_value
                                     builder); e'
        | A.Index(a, i) -> let a' = expr builder g_map l_map a in
                         let i' = expr builder g_map l_map (List.hd i) in
                         let extract_array = L.build_extractvalue a' 1
                             "extract_ptr" builder in
                         let extract_value = L.build_gep extract_array [|
                             i' |] "extract_value" builder in
                         if L.type_of extract_array == L.pointer_type i8_t
                         then let first_value = L.build_load extract_value
                             "value" builder in
                              let new_string = L.build_array_malloc i8_t
                                  (L.const_int i32_t 2) "tmp" builder in
                              let more = L.build_gep new_string [|
                                  L.const_int i32_t 0 |] "tmp2" builder in
                              ignore(L.build_store first_value more
                                  builder);
                              let more = L.build_gep new_string [|
                                  L.const_int i32_t 1 |] "tmp2" builder in
                              ignore(L.build_store (L.const_int i8_t 0)
                                  more builder);
                              let new_literal = L.build_malloc
                                  (ltype_of_typ (A.Simple(A.String)))
                                  "arr_literal" builder in
                              let first_store = L.build_struct_gep
                                  new_literal 0 "first" builder in
                              let second_store = L.build_struct_gep
                                  new_literal 1 "second" builder in
                              ignore(L.build_store (L.const_int i32_t 1)
                                  first_store builder);
                              ignore(L.build_store new_string second_store
                                  builder);
                              let actual_literal = L.build_load new_literal
                                  "actual_arr_literal" builder in
                              actual_literal
                         else L.build_load extract_value "value" builder
        | A.Binop (e1, op, e2) ->
         let e1' = expr builder g_map l_map e1
         and e2' = expr builder g_map l_map e2 in
          if (L.type_of e1' = f_t || L.type_of e2' = f_t) then
            (match op with
              A.Add    -> L.build_fadd
            | A.Sub    -> L.build_fsub
            | A.Mult   -> L.build_fmul
            | A.Div    -> L.build_fdiv
            | 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
            | _ -> raise (Failure ("operator not supported for operand"))
            ) e1' e2' "tmp" builder
          else
            (match op with
              A.Add    -> L.build_add
            | A.Sub    -> L.build_sub
            | A.Mult   -> L.build_mul
            | A.Div    -> L.build_sdiv
            | 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
             ) e1' e2' "tmp" builder
        | A.Unop(op, e) ->
         let e' = expr builder g_map l_map e in
           (match op with
              A.Neg    ->
                (if (L.type_of e' = f_t) then
                 L.build_fneg
                else
                 L.build_neg)
             | A.Not    -> L.build_not) e' "tmp" builder
         | A.Pop(e, op) -> let e' = expr builder g_map l_map e in
          (match op with
          | A.Inc -> ignore(expr builder g_map l_map (A.Assign(e, A.Binop(e, A.Add, A.NumLit(1))))); e'
          | A.Dec -> ignore(expr builder g_map l_map (A.Assign(e, A.Binop(e, A.Sub, A.NumLit(1))))); e')
        | A.Assign (e1, e2) -> let l_val = (addr_of_expr e1 builder g_map l_map) in
        let e2' = expr builder g_map l_map e2 in
         ignore (L.build_store e2' l_val builder); e2'

        | A.Dot (e, field) -> let llvalue = (addr_of_expr e builder g_map l_map) in
        let built_e = expr builder g_map l_map e in
```

```
        let built_e_lltype = L.type_of built_e in
        let built_e_opt = L.struct_name built_e_lltype in
        let built_e_name = (match built_e_opt with
                                  | None -> ""
                                  | Some(s) -> s)
        in
        let indices = StringMap.find built_e_name struct_field_indices in
        let index = StringMap.find field indices in
        let access_llvalue = L.build_struct_gep llvalue index "tmp" builder in
                           L.build_load access_llvalue "tmp" builder

        | A.Call ("print", [e])

        | A.Call ("printb", [e]) -> L.build_call printf_func [| int_format_str builder; (expr builder g_map l_map e) |]
                                "printf" builder
        | A.Call ("printbig", [e]) -> L.build_call printbig_func [| (expr builder g_map l_map e) |] "printbig" builder
        | A.Call("open", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
             L.build_call open_func (Array.of_list x) "fopen" builder
        | A.Call("close", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
             L.build_call close_func (Array.of_list x) "fclose" builder
        | A.Call ("read", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
             L.build_call read_func (Array.of_list x) "fread" builder
        | A.Call("write", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
             L.build_call write_func (Array.of_list x) "fputs" builder
        | A.Call("strlen", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
             L.build_call strlen_func (Array.of_list x) "strlen" builder
        | A.Call("strcmp", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
             L.build_call strcmp_func (Array.of_list x) "strcmp" builder
        | A.Call("strcat", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
             L.build_call strcat_func (Array.of_list x) "strcat" builder
        | A.Call("strcpy", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
             L.build_call strcpy_func (Array.of_list x) "strcpy" builder
        | A.Call("strget", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
             L.build_call strget_func (Array.of_list x) "strget" builder
        | A.Call("to_lower", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
             L.build_call to_lower_func (Array.of_list x) "char_lower" builder
```

```
      | A.Call("calloc", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
           L.build_call calloc_func (Array.of_list x) "calloc" builder
      | A.Call("free", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
           L.build_call free_func (Array.of_list x) "free" builder
      | A.Call("is_stop_word", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
           L.build_call is_stop_word_func (Array.of_list x) "is_stop_word" builder
      | A.Call("word_count", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
         L.build_call word_count_func (Array.of_list x) "word_count" builder
      | A.Call("string_at", e) -> let x = List.rev (List.map (expr builder g_map l_map) (List.rev e)) in
         L.build_call string_at_func (Array.of_list x) "string_at" builder
      | A.Call ("print_double", [e]) ->
           L.build_call printf_func [| float_format_str builder ; (expr builder g_map l_map e) |] "printf" builder
      | A.Call ("print_string", [e]) ->
            L.build_call printf_func [| string_format_str builder ; (expr builder g_map l_map e) |] "printf" builder
      | A.Call ("print_all", [e]) ->
         let e' = expr builder g_map l_map e in
         let e_type = (gen_type) g_map l_map e in
         L.build_call printf_func [| (format_str e_type builder) ; e' |] "printf" builder
      | A.Call ("print_char", [e]) ->
            L.build_call printf_func [| char_format_str builder ; (expr builder g_map l_map e) |] "printf" builder
      | A.Call (f, act) ->
        let (fdef, fdecl) = StringMap.find f function_decls in
        let actuals = List.rev (List.map (expr builder g_map l_map) (List.rev act)) in
        let result = (match fdecl.A.typ with A.Void -> ""
                                                   | _ -> f ^ "_result") in
        L.build_call fdef (Array.of_list actuals) result builder
    in

(* Construct the function's "locals": formal arguments and locally
      declared variables. Allocate each on the stack, initialize their
      value, if appropriate, and remember their values in the "locals" map *)
    let local_vars =
      let add_formal m (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);
```

```
   StringMap.add n local m in

 let add_local m (A.VarDecl(t, n, e)) =
   let e' = match e with
        A.Noexpr -> get_init_noexpr t
      | _ -> expr builder global_vars m e
   in
   L.set_value_name n e';
   let l_var = L.build_alloca (ltype_of_typ t) n builder in
   ignore (L.build_store e' l_var builder);
   StringMap.add n l_var m in

 let formals = List.fold_left2 add_formal StringMap.empty
     fdecl.A.formals
      (Array.to_list (L.params the_function)) in
   List.fold_left add_local formals fdecl.A.locals in

 (* Invoke "f builder" if the current block doesn't already
    have a terminal (e.g., a branch). *)
 let add_terminal builder f =
   match L.block_terminator (L.insertion_block builder) with
Some _ -> ()
   | None -> ignore (f builder) in

 (* Build the code for the given statement; return the builder for
    the statement's successor *)
 let rec stmt builder = function
     A.Block sl -> List.fold_left stmt builder sl
   | A.Expr e -> ignore (expr builder global_vars local_vars e);
       builder
   | A.Return e -> ignore (match fdecl.A.typ with
  A.Void -> L.build_ret_void builder
| _ -> L.build_ret (expr builder global_vars local_vars e) builder);
    builder
   | A.If (predicate, then_stmt, else_stmt) ->
      let bool_val = expr builder global_vars local_vars predicate in
 let merge_bb = L.append_block context "merge" the_function in

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

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

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

   | A.While (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 global_vars local_vars 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

      | A.For (e1, e2, e3, body) -> stmt builder
       ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr e3]) ]
           )
    in

    (* Build the code for each statement in the function *)
    let builder = stmt builder (A.Block fdecl.A.body) in

    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.A.typ with
        A.Void -> L.build_ret_void
      | t -> L.build_ret (get_init_noexpr t))
  in

  List.iter build_function_body functions;
  the_module
```

9.6 english.ml

```
(* Authors: Rabia, Michele, Candace, Emily, Nivita *)
(* Top-level of the MicroC compiler: scan & parse the input,
   check the resulting AST, generate LLVM IR, and dump the module *)

module StringMap = Map.Make(String)

type action = Ast | 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 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: ./english.native [-a|-l|-c] [file.mc]" 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 = Parser.program Scanner.token lexbuf in
  Semant.check ast;
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate
      ast))
  | Compile -> let m = Codegen.translate ast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

9.7 c-code.c

```c
/* Authors: Rabia, Michele, Candace, Emily, Nivita */
/*
 *  C - Code
 */

#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
/*
 * Font information: one byte per row, 8 rows per character
 * In order, space, 0-9, A-Z
 */
static const char font[] = {
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x1c, 0x3e, 0x61, 0x41, 0x43, 0x3e, 0x1c, 0x00,
  0x00, 0x40, 0x42, 0x7f, 0x7f, 0x40, 0x40, 0x00,
  0x62, 0x73, 0x79, 0x59, 0x5d, 0x4f, 0x46, 0x00,
  0x20, 0x61, 0x49, 0x4d, 0x4f, 0x7b, 0x31, 0x00,
  0x18, 0x1c, 0x16, 0x13, 0x7f, 0x7f, 0x10, 0x00,
  0x27, 0x67, 0x45, 0x45, 0x45, 0x7d, 0x38, 0x00,
  0x3c, 0x7e, 0x4b, 0x49, 0x49, 0x79, 0x30, 0x00,
  0x03, 0x03, 0x71, 0x79, 0x0d, 0x07, 0x03, 0x00,
  0x36, 0x4f, 0x4d, 0x59, 0x59, 0x76, 0x30, 0x00,
  0x06, 0x4f, 0x49, 0x49, 0x69, 0x3f, 0x1e, 0x00,
  0x7c, 0x7e, 0x13, 0x11, 0x13, 0x7e, 0x7c, 0x00,
  0x7f, 0x7f, 0x49, 0x49, 0x49, 0x7f, 0x36, 0x00,
  0x1c, 0x3e, 0x63, 0x41, 0x41, 0x63, 0x22, 0x00,
  0x7f, 0x7f, 0x41, 0x41, 0x63, 0x3e, 0x1c, 0x00,
  0x00, 0x7f, 0x7f, 0x49, 0x49, 0x49, 0x41, 0x00,
  0x7f, 0x7f, 0x09, 0x09, 0x09, 0x09, 0x01, 0x00,
  0x1c, 0x3e, 0x63, 0x41, 0x49, 0x79, 0x79, 0x00,
  0x7f, 0x7f, 0x08, 0x08, 0x08, 0x7f, 0x7f, 0x00,
  0x00, 0x41, 0x41, 0x7f, 0x7f, 0x41, 0x41, 0x00,
  0x20, 0x60, 0x40, 0x40, 0x40, 0x7f, 0x3f, 0x00,
  0x7f, 0x7f, 0x18, 0x3c, 0x76, 0x63, 0x41, 0x00,
  0x00, 0x7f, 0x7f, 0x40, 0x40, 0x40, 0x40, 0x00,
  0x7f, 0x7f, 0x0e, 0x1c, 0x0e, 0x7f, 0x7f, 0x00,
  0x7f, 0x7f, 0x0e, 0x1c, 0x38, 0x7f, 0x7f, 0x00,
  0x3e, 0x7f, 0x41, 0x41, 0x41, 0x7f, 0x3e, 0x00,
  0x7f, 0x7f, 0x11, 0x11, 0x11, 0x1f, 0x0e, 0x00,
  0x3e, 0x7f, 0x41, 0x51, 0x71, 0x3f, 0x5e, 0x00,
  0x7f, 0x7f, 0x11, 0x31, 0x79, 0x6f, 0x4e, 0x00,
  0x26, 0x6f, 0x49, 0x49, 0x4b, 0x7a, 0x30, 0x00,
  0x00, 0x01, 0x01, 0x7f, 0x7f, 0x01, 0x01, 0x00,
  0x3f, 0x7f, 0x40, 0x40, 0x40, 0x7f, 0x3f, 0x00,
  0x0f, 0x1f, 0x38, 0x70, 0x38, 0x1f, 0x0f, 0x00,
  0x1f, 0x7f, 0x38, 0x1c, 0x38, 0x7f, 0x1f, 0x00,
  0x63, 0x77, 0x3e, 0x1c, 0x3e, 0x77, 0x63, 0x00,
  0x00, 0x03, 0x0f, 0x78, 0x78, 0x0f, 0x03, 0x00,
  0x61, 0x71, 0x79, 0x5d, 0x4f, 0x47, 0x43, 0x00
};

void printbig(int c)
{
  int index = 0;
  int col, data;
  if (c >= ’0’ && c <= ’9’) index = 8 + (c - ’0’) * 8;
  else if (c >= ’A’ && c <= ’Z’) index = 88 + (c - ’A’) * 8;
  do {
    data = font[index++];
    for (col = 0 ; col < 8 ; data <<= 1, col++) {
      char d = data & 0x80 ? ’X’ : ’ ’;
      putchar(d); putchar(d);
    }
    putchar(’\n’);
  } while (index & 0x7);
}

char char_lower(char c)
{
  return tolower(c);
}

char strget(char* c, int x)
{
  return *(c + x);
}

```

```
int is_stop_word(char * c){
  char word[100];
  char whitespace[100];

  FILE *file = fopen("stopwords.txt", "r");

  while(!feof(file)) {
      fscanf(file,"%[^ \n\t\r]s",word);
      if(strcmp(c, word) == 0){
        fclose(file);
        return 1;
      }
      fscanf(file,"%[ \n\t\r]s",whitespace);
  }

  fclose(file);
  return 0;

}

int word_count(char * str){
  int count = 0;
  int curr = 0;
  while(*str != '\0'){
    if (*str == ' ') {
      if (curr == 1){
        count = count + 1;
        curr = 0;
      }
    }

      if(*str != ' '){
        curr = 1;
      }
     str++;
  }

  if (curr == 1){
    count = count + 1;
  }

  return count;
}

char * string_at(char* str, int i, int size, int len){
  char char_string[2] = {str[i] , '\0'};
  char * buf = calloc(size, len);
  buf = strcpy(buf, char_string);
  return buf;
}
```

```c

#ifdef BUILD_TEST
int main()
{
  char s[] = "HELLO WORLD09AZ";
  char *c;
  for ( c = s ; *c ; c++) printbig(*c);
}
#endif
```

9.8 Makefile

```makefile
# Authors: Rabia, Michele, Candace, Emily, Nivita
# Make sure ocamlbuild can find opam-managed packages: first run
#
# eval ‘opam config env‘

# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

all : english.native c-code.o

english.native :
   ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \
      english.native

# "make clean" removes all generated files

.PHONY : clean
clean :
   ocamlbuild -clean
   rm -rf testall.log *.diff english scanner.ml parser.ml parser.mli
   rm -rf c-code
   rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.s *.ll *.out *.exe

# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx english.cmx

english : $(OBJS)
   ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis
       $(OBJS) -o english

scanner.ml : scanner.mll
   ocamllex scanner.mll

parser.ml parser.mli : parser.mly
   ocamlyacc parser.mly

```

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

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

%.cmx : %.ml
   ocamlfind ocamlopt -c -package llvm $<

# Testing the c-code

c-code : c-code.c
   cc -o c-code -DBUILD_TEST c-code.c


### Generated by "ocamldep *.ml *.mli" after building scanner.ml and parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
english.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
english.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo

# Building the tarball

TESTS = add1 arith1 arith2 arith3 fib for1 for2 func1 func2 func3 \
    func4 func5 func6 func7 func8 gcd2 gcd global1 global2 global3 \
    hello if1 if2 if3 if4 if5 local1 local2 ops1 ops2 var1 var2  \
    while1 while2 printbig printstring open1 write1 struct1 stringinit \
    stringfunc1 stringfunc2 char1 pops1 alloc

FAILS = assign1 assign2 assign3 dead1 dead2 expr1 expr2 for1 for2 \
    for3 for4 for5 func1 func2 func3 func4 func5 func6 func7 func8 \
    func9 global1 global2 if1 if2 if3 nomain return1 return2 while1 \
    while2 pops1

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

TARFILES = ast.ml codegen.ml Makefile english.ml parser.mly README scanner.mll \
   semant.ml testall.sh $(TESTFILES:%=tests/%) c-code.c arcade-font.pbm \
   font2c

english-llvm.tar.gz : $(TARFILES)
   cd .. && tar czf english-llvm/english-llvm.tar.gz \
     $(TARFILES:%=english-llvm/%)
```