THE ENGLISH LANGUAGE LANGUAGE

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

December 20th, 2017

Contents

1 Introduction **2**
1.1 What is The English Language Language? 2

2 Usage **3**
2.1 Getting Started 3
2.2 Writing Your Program 3
2.3 Compiling 3

3 Language Tutorial **4**
3.1 Integer 4
3.2 String 4
3.3 Array 4
3.4 Struct 4
3.5 Variable Declaration 4
3.6 For Loop 4
3.7 While Loop 4
3.8 If/else Statement 4
3.9 Library Functions 4

4 Language Reference Manual **5**
4.1 Lexical Elements 5
4.2 Data Types 6

5 Project Plan **8**
5.1 Process 8
5.2 Project Timeline 8

6 Architectural Design **9**

7 Test Plan **10**
7.1 Sort Names 10
7.2 Word Count 11
7.3 Plagiarism 13

8 Lessons Learned **17**

9 Appendix **18**
9.1 scanner.mll 18
9.2 parser.mly 19

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, and we wanted to find a way to battle it.

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 this service that will allow them to check for plagiarism quickly and easily, without having to peruse hundreds of documents manually.

2 Usage

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

2.1 Getting Started

First, 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 command in your shell:

```
\$
```

3 Language Tutorial

Now we can get started with writing more complicated programs. The English Language language allows for various data types, structures, functions, and other such implementations. Here we will give an overview of each one, as well as sample code to demonstrate its usage.

3.1 Integer

Various operations can be performed on integers in ELL, such as basic arithmetic operations. Below is an example of simple addition between integers:

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

3.2 String

3.3 Array

3.4 Struct

3.5 Variable Declaration

3.6 For Loop

3.7 While Loop

3.8 If/else Statement

3.9 Library Functions

ELL utilizes several library functions, including strcmp(), open(), and close(). A full list of all available library functions are detailed later, and any of these can be utilized simply by calling the function name and passing in the appropriate input. Here is an example implementation of the strcmp() function.

```
if (strcmp(words[1], words[2]) == 0)
```

4 Language Reference Manual

4.1 Lexical Elements

IDENTIFIERS
Identifiers are used for naming Docs and other data types. They are defined by at least one lowercase letter and can be followed by any combination of letters, numbers, or underscores.

RESERVED KEYWORDS

int	boolean	float	string
char	void	struct	file_ptr
while	for	if	else
return	print	print_string	

LITERALS
Integer literal: a series of one or more digits from 0-9
Float 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

OPERATORS

+, -, *, /, %	arithmetic integer operators
==, <, >, <=, >=	numerical integer operators
||, &&, !	logical operators
=	assignment

DELIMITERS
Parentheses: used to contain arguments in function calls, as well as to ensure precedence in expressions
Semicolon: end of statement
Curly braces: used to enclose block logic in conditionals and loops, as well as to contain code in functions Commas: used to separate inputs in function calls

WHITESPACE
Whitespace is only used to separate tokens.

COMMENTS
Single line comments are started with // and multiple line comments are started with /* and */

4.2 Data Types

PRIMITIVE DATA TYPES

Integers: most numbers will be declared as type `int`. Example:

```
int x = 5
```

Floats: floating point numbers will be declared as type `float`. Example:

```
float x = 2.3
```

Booleans: boolean values will be declared as type `bool` and can be either True or False. Example:

```
bool match = True
```

NON-PRIMITIVE DATA TYPES

Strings: strings are defined by the type `string`. They will be defined with two double quotes 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 should begin with a capital letter. Example:

```
struct Object = {
    int x;
    char y;
}
```

Arrays: arrays are a collection of any other data type, and can be defined by simply typing the data type followed by an open and close bracket, optionally containing the number of elements in between, and the brackets are then followed by the name of the array. Example:

```
string [5] a;
```

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");
```

Functions

Built-in Functions: functions predefined in the compiler. In ELL, these are:
`print()` prints to the screen any data type passed as an argument
`print_string()` prints to the screen an argument of type `string`
`open()` opens a text file by passing in the file name as type `string` as the first argument, and the read/write conditions as the second argument
`close()` closes a text file by passing in its file pointer of type `file_ptr` as an argument

User-defined Functions: these functions are defined and called as follows:

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

function_name(<args>);
```

Control Flow Statements

Conditionals:

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

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 began building off of this new compiler to implement the various types, structures, and functions we needed for our own language. We split up the work based on how difficult each section seemed, as well as on our role designations at the beginning of the semester, while also continuing to help each other throughout the process.

5.2 Project Timeline

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.

[block diagram of compiler]

7 Test Plan

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

7.1 Sort Names

The following program demonstrates an example of how an instructor can sort their students' names in alphabetical order, by implementing mergesort on a list of strings.

```
// sortnames.ell

/* merge sort helper function*/
int merge_helper(int a[], int l, int m, int r){

int n1 = m-l +1;
int n2 = r -m;
int left [n1];
int right [n2];

int i;
int j;

/* copy data to temporary arrays */
for(i =0; i <n1; i = i+1){
  left[i] = a[1 + i];
}
for(j =0; j<n2; j= j+1){
  right[j] = a[m +1 + j];
}

/* merge arrays */
i = 0;
j = 0;
int k = 1;

while ( i < n1 && j< n2){

    if (strcmp(left[i],right[j]) > 0){
      a[k] = left[i];
      i = i +1;
    }
    else{
      a[k] = r[j];
      j = j+1;
    }
    k = k + 1;
    }
```

```
    /* copy remainder of left */
    while (i <n1){

    a[k] = left[i];
    i = i +1;
    k = k+1;
    }

    /* copy remainder of right */
    while(j <n2){
    a[k] = right[j];
    j = j+1;
    k = k+1;
    }
   return 0;
}

/* merge sort function */
int merge_sort(string a[], int l, int r){
  if(l <r){
    int m = l+r/2;
    merge_sort(a, l, m);
    merge_sort(a, m+1, r);

    merge_helper(a, l, m, r);
  }
  return 0;
}

int main() {
  string [] test;
  test = ["emily", "rabia", "nvita", "michele", "candace"];
  merge_sort(test);
  return 0;
}
```

7.2 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.

```
// follow_word_count.ell

/* student struct*/
struct Student {
   string Name;
   string Essay;
```

```
};

/* function checks if a string is 100 words or under */
int follows_word_count(string essay){

   int words = word_count(essay);
   if (words < 101){
      return 1;
   }
   return 0;
}

/* function reads in a file and returns the content in string format */
string import_essay(string file_name){
   string s1 = calloc(1, 2000);
   file_ptr fp = open("tests/hello.txt", "rb");
      int size = read(s1, 1, 2000, fp);
      close(fp);
      return s1;
}


int main(){
   /* declare array of students who submitted an essay */
   struct Student [5] students;

   struct Student candice;
   candice.Name = "Candace";
   candice.Essay = import_essay("candace_essay.txt");
   students[0] = candace;

   struct Student emily;
   emily.Name = "Emily";
   emily.Essay = import_essay("emily_essay.txt");
   students[1] = emily;

   struct Student michele;
   michele.Name = "Michele";
   michele.Essay = import_essay("michele_essay.txt");
   students[2] = michele;

   struct Student nvita;
   nvita.Name = "Nvita";
   nvita.Essay = import_essay("nvita_essay.txt");
   students[3] = nvita;

   struct Student rabia;
   rabia.Name = "Rabia";
   rabia.Essay = import_essay("rabia_essay.txt");
   students[4] = rabia;
```

```
  /* check which students followed the word count */
  int i;
  for (i = 0; i< 5; i = i + 1){
     if(follows_word_count(students[i].Essay) == 1){
        print_string(students[i].Name);
        print_string("'s essay follows the word count.");
     }
     else{
        print_string(students[i].Name);
        print_string("'s essay has too many words.");
     }
  }

  /* free allocated memory */
     free(candace.Essay);
     free(emily.Essay);
     free(michele.Essay);
     free(nvita.Essay);
     free(rabia.Essay);

  return(0);
}
```

7.3 Plagiarism

The following program demonstrates an example of how someone can detect plagiarism between documents. The code has two functions, one for finding top word count and one for finding common top word counts between data sets, in order to check for similarity between two documents.

```
// plagiarism.ell

/* struct for keeping track of word frequency */
struct WordFreq{
      string [100] Top_words;
      int[100] Top_counts;
  };

/* function returns top int top number of most frequent words in string,
    exclusing common stop words */
string [] relevant_words(string essay, int top){

  int size = word_count(essay);

  string [size] words;

  /* todo */
```

```
words = split_at(essay, " ", words);

/* count the frequency of each word in the text document and store in
    count */
/* still considering a data structure for this */

int [size] count;

int i;
int j;
int found = 0;
string current;

for (i = 0; i < size; i = i + 1){
   current = words[i];
   /* don't count stop words */
   if (is_stop_word(current) == 0){
   for (j = 0; j< size; j = j+1){
       /* check if same word has been found */
      if (found == 0){
         if (strcmp(current, words[j]) == 0){
            found = 1;
            count [j]++;
         }
      }
   }
   found = 0;

}
}

/* find top words */
struct WordFreq word_freq;

int max = 0;
int max_index = 0;
string max_word;
int k;

for (k = 0; k< top; k = k +1){
   for (i = 0; i < size; i = i + 1){
      if (max < count[i]){
         int present = 0;
         for (j = 0; j<top; j = j+1){
            /* check if value already put in top words */
            if (count[i] == word_freq.top_counts[j]){              if
                (string_compare(words[i] == word_freq.top_words[j])){
                 present = 1;
            }
            }
```

```
            }
            if(present == 0){
               max_index = i;
               max = count[i];

            }
      }
      }

      word_freq.top_words[k] = words[max_index];
      word_freq.top_counts[k] = count[max_index];
      max = 0;
         }
    return word_frequency.Top_words;
}

/* check for simlarity of content between two files, takes in top, top i
    words are compared, returns number of common top words. */
int check_similar(string file1, string file2, int top){

   /* find relevant words in each document */
   /* the hard coded numbers can be changed or made input */ // input
       would be ideal
   string [top] relevant1 = find_relevant(file1, top);
   string [top] relevant2 = find_relevant(file2, top);

   /* check for similarity in content */
   int i;
   int j;
   int similar = 0;
   for(i = 0; i< top; i = i + 1){
      for(j = 0; j< top; j = j + 1){
         if(strcmp(relevant1[i], relevant2[i]) == 0){
         similar = similar + 1;
         }
      }
   }

   return similar;
   }

int main(){

   string s1 = calloc(1, 2000);
   file_ptr fp1 = open(file_name, "rb");
   int size1 = read(s1, 1, 2000, fp);
   close(fp1);

   string s2 = calloc(1, 2000);
   file_ptr fp2 = open(file_name, "rb");
```

```
    int size2 = read(s1, 1, 2000, fp);
    close(fp2);

    int result = check_similarity(s1, s2, 30);

    print(result);

    free(s1);
    free(s2);
    return 0;

}
```

8 Lessons Learned

[lessons learned and future advice from each team member]

9 Appendix

9.1 scanner.mll

```
(* Ocamllex scanner for MicroC *)

{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }        (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACK}
| ']'      { RBRACK}
| ';'      { 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 }
| "float"  { FLOAT }
| "bool"   { BOOL }
| "void"   { VOID }
| "true"   { TRUE }
| "false"  { FALSE }
| "string" { STRING }
| "array"  { ARRAY }
```

```
| "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

```
/* 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 {ArrayAccess($1, $3)}
  | LPAREN expr RPAREN { $2 }


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

actuals_list:
```

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

[full code listing as separate subsections with authors as a comment on top]