COMS W4115 Programming Languages and Translator

TAPE: A File Manipulation Language

Tianhua Fang (tf2377)
Alexander Sato (as4628)
Priscilla Wang (pyw2102)
Edwin Chan (cc3919)

Fall 2016

Table of Contents

1. Introduction

Motivation

The way in which we store data and information has drastically changed in modern times. In the past, important pieces of data were often physically inscribed and stored in ancient relics such as “file cabinets” and “rolodexes”. In current times, we have started to shift away from these hard copies to digital ones. Digital copies of data offer many benefits, such as the obvious saving of space. Yet, organizing the data became much more difficult. With hard copies, one is dealing with tangible files which can be physically stored and retrieved. With digital copies, the files are no longer tangible, so organization can be a bit more difficult. Currently, there is a huge, thriving industry specifically dedicated to storing and manipulating these digital files.

With Tape, we are aiming to streamline and simplify the process of file manipulation. We wanted to create a language in which the user can have more control over how to search through and manipulate the given data. By looking at other similar languages, such as AWK and C, we aimed to improve and to simplify the task of handling data.

So, what is Tape?

Tape is a file manipulating language created by Tianhua Fang, Alexander Sato, Priscilla Wang, and Edwin Chan. It was inspired by the languages AWK and C. The name Tape is derived from the first letter of the first names of the creators. It also representative the functionality of adhesive tapes since users are essentially “taping” files together when they are using Tape.

2. Language Tutorial

This section contain information on how users can get started with Tape. For a more in-depth overview on the language, including syntax, please refer to the Language Reference Manual.

Getting Started

Inside the Tape project folder, type “make”. This will create the tape compiler. To run your own tape program, type the following in the command line:

```
./tape.native <  path_to/file_name.tape > path_to/file_name.ll
```

This will compile the program you wrote in file_name.tape into a tape program. `path_to/file_name.tape` is the program that you wrote. `path_to/file_name.ll` is the llvm file that you want to name as the compiled version of your program file. We recommend that you named the `.ll` file the same name as your `.tape file to avoid confusion.`

To run the program, type the following in the command line:

```
lli path_to/file_name.ll
```

Running a Tape Program: Hello World!

Let’s start with the most basic example, Hello World!

After you have downloaded and saved the Tape project folder, create a new file named `helloworld.tape`. In the file, write the following code:

```
int main(){
  # This is hello world.
  print_s("hello tape");
  return 0;
}
```

The above code demonstrates the following features of Tape:

1. Mandatory `main()` function, with no parameters and returns 0 as success, otherwise as failure.
2. Calling the built-in print string function, `print_s()`.

To compile and run the code above, type the following in the command line:

```
make
```

```
./tape.native <  helloworld.tape > helloworld.ll
lli helloworld.ll
```

You will see the output:

```
hello tape
```

Don't forget the ; !

Each line within two curly braces, { and }, needs to end with a semicolon, ;, or else the compiler will throw an error. This includes variable declaration, variable assignment, and actions. Please note that semicolons do not follow the right curly brace.

Variable Declaration

Tape requires users to declare all variables before using them in the program. Each variable has to have a name and a type. Tape supports the following types:

```
int string void char
```

Below is an example of a Tape program where we declare `my_string`, a string, assign it to the string `"I love plt.",` and printing out `my_string`. In addition to that, there is a variable named `i`, which is assigned to the integer 3, and printed out by the built-in function `print_i`.

```
int main() {
    string my_string;
    int i;

    my_string = "I love plt.";
    i = 3;

    print_s(my_string);
    print_i(i);

    return 0;
}
```

Using the same way we ran `helloworld.tape` above, we will get the following as output:

```
i love plt.
3
```

Integer

Users can perform mathematical operations such as + and - on integers. Below is an example of a Tape program that adds two integers together and returns the result.

Strings and Arrays

Strings are declared in the format: `string variable_name` . Strings are assigned to text within quotation marks. The following example demonstrates declaring a string named `my_string` and assigning it to the string value of `"I love plt."`.

```
int main() {
    string my_string;
    my_string = "I love plt.";
    return 0;
}
```

In Tape, arrays are also declared as a string. To create an array, declare the variable as a string type, and assign it to `new[index]` , where index is the length of the array. Below is an example array of size 5 named `my_array`.

```
int main() {
    string my_array;
    my_array=new[5];
    return 0;
}
```

For

The syntax of a for loop in Tape is in the following format:

```
for () { some action }
```

Note that unlike C,C++, and Java, the index within the for loop has to be declared outside of the parenthesis.

While

The syntax of the whole loop in Tape is in the following format:

```
while( some condition )  {  some action }
```

If/Else

The syntax for conditionals in Tape is in the following format:

```
if (some condition) { some action } else { other action }
```

If `else{}` is missing from the conditional, then the compiler will throw an error. Users can leave the space between `{` and `}` if they do not wish to perform any action in the else case.

Regular Expression

The regular expression can only be applied to library function:

```
tape(file filename, string re)
```

Function `tape` accept string or a certain regular expression and search it in the specified file. The regular expression must be in form of "`/string/operation/string /`".

Tape has 6 regular expression operations, and they are as follow:

Symbol	Name	Syntax and Example
`&`	And	`/apple/&/melo/` Searches lines that contain both apple and melo
`|`	Or	`/apple/|/melo/` Searches lines that contain either apple or melo
`^`	Xor	`/apple/^/melo/` Searches lines that contain neither apple nor melo
`-`	Between	`/a/-/e/` Searches for lines that contain a string that starts with a and ends with e.
`*`	Kleene	`/ap/*/le/` Searches for lines that contain ale, aple, apple, appple, apppple, etc.
`?`	Subexpression	`/will/?/iam/` Searches for lines that contain will or william.

Below is an example on how to use the `tape` function.

If you've downloaded the folder with all of Tape's source code, there is a file called `dictionary.txt` under main directory that contains a list of words. You can use the tape to find all words starting with a and end with z by calling the below statement in your tape program.

```
tape("dictionary.txt", "/a/-/z/");
```

For more information regarding the tape function, please refer to the section on Standard Library Functions in the Language Reference Manual.

Library Functions

Tape has several library functions that users may use in their program. To use library functions, simply call the library function in the program in the following format: `function_name(input);` .

3. Language Manual

Introduction

Tape is a data-driven scripting language that allows users to read, write, and manipulate documents easily. This is the ideal language to use for file manipulation because it perform actions based on patterns. Tape compiles to LLVM.

Quick Overview

There are 3 basic rules for a Tape program:

1) Each Tape program must have a main function, much like a C program.
2) Scope in Tape is determined with curly braces.
3) Every statement must end with a semicolon.

Lexical Conventions

Identifiers

In Tape, identifiers are strings that represent names of different elements such as variables and functions. Identifiers are also case sensitive. User created identifiers must begin with a letter [a-z]/[A-Z], and it can be followed with more letters, numbers, or an underscore. Identifiers are case sensitive and once declared, they are unique within the scope.

Keywords and Reserved Names

Tape has a few words that are reserved, meaning users cannot create variables or functions with these names. The following keywords and names are reserved for Tape:

```
if else while for return void string int file new
```

Types

Tape supports 4 different types: `int char string void`.

Name	Description and Examples

`int`	This is a 32 bit signed integer value. An integer constant must be at least 1 digit long, and it must use the values [0-9].
	`int myInt;`
	`myInt = 3;`

`char`	This is a 8 bit ASCII character value. A character can be any single lowercase or uppercase letter [a-z]/[A-z], any numerical digit [0-9].
	`char myChar;`
	`myChar = 'h';`

`string`	Users can assign strings to characters within quotes. Example:
	`string myString;`
	`myString = "hello";`
	Users can also assign strings to an array. Users can modify the values within each index of the array they same way they would modify values within an array in Java or C.
	The array required user to use "new". Without the init will result an error. The "new" will allocate memory on the stack and used for the array.
	The following example demonstrates the creation of a string as an array of 2 characters, and assigning the string to say "hi".
	`string myString;`
	`myString = new[2];`
	`mystring[0] = 'h';`
	`mystring[1] = 'i';`

`void`	Void type does not return a result value to its caller.

Basic Operators

Tape has the following basic operators:

Operators	Name	Associativity
=	Assign	Right
==	Equal to	Left
!=	Not equal to	Left
>	Greater than	Left
<	Less than	Left
+	Addition	Left
-	Subtraction	Left
*	Multiplication	Left
>=	Greater or Equal	Left
<=	Smaller or Equal	Left

Regular Expression Operators

Since Tape has a built in function for searching regular expressions within a file, Tape also has operators that are used exclusively for regular expression searches. To do a search of one string in a file, put the words in the string within two slashes like so:

```
/my word/
```

To do a search on two strings in a file, put both words in two slashes, and put a regular expression operator in between like so:

```
/first word/<Insert Operation Here>/second word/
```

The following are operators used for regular expression operations:

Operator	Name	Example
-	Between	Find all text between two brackets `[`, and `]`: `/[/-/]/`
`&`	And	Find all lines with the word “movie” and “manga”: `/movie/&/manga/`
`|`	Or	Find all lines with the word “movie” or “manga”: `/movie/|/manga/`
`?`	Subexpression	Find all lines with star or starwars `/star/?/wars/`
`^`	Exclusive Or	Find all lines with neither the word “movie” nor “manga”: `/movie/^/manga/`
`*`	Kleene Closure	Find a list of words ‘ggle’ with one or more ‘o’ between ‘g’ and ‘gle:. `/go/*/gle/`

Comments

The comment operator is #. Everything to the right of it will be ignored by the compiler. Below is an example of how comments are used.

```
# This is a comment. It will be ignored by the compiler
```

Types

Tape supports 4 different types: `int char string void.`

Name	Description and Examples

`int`	This is a 32 bit signed integer value. An integer constant must be at least 1 digit long, and it must use the values [0-9].
	`int myInt;`
	`myInt = 3;`

`char`	This is a 8 bit ASCII character value. A character can be any single lowercase or uppercase letter [a-z]/[A-z], any numerical digit [0-9].
	`char myChar;`
	`myChar = 'h';`

`string`	Users can assign strings to characters within quotes. Example:
	`string myString;`
	`myString = "hello";`
	Users can also assign strings to an array. Users can modify the values within each index of the array they same way they would modify values within an array in Java or C.
	The array required user to use "new". Without the init will result an error. The "new" will allocate memory on the stack and used for the array.
	The following example demonstrates the creation of a string as an array of 2 characters, and assigning the string to say "hi".
	`string myString;`
	`myString = new[2];`
	`mystring[0] = 'h';`
	`mystring[1] = 'i';`

`void`	Void type does not return a result value to its caller.

Keywords

The following keywords are reserved for Tape.

Name	Description
`for`	`for` provides a way to iterate over a range of values.
`while`	`while` provides a way to repeatedly execute a statement while condition is true.
`if... else...`	`if` and `else` provide a way to execute a certain section of code under certain conditions.
`#`	`#` provides a way to comment codes. To comment out a line of code, just add `#` to the beginning of the line.

Statements

A statement is a unit of code.

Expression Statement

An expression statement is an expression followed by a semicolon.

Declaration Statement

Tape requires users to declare all variables before assigning or using them in the program. Each variable has to have a name and a type. Tape supports the following types:

```
int string char
```

Below are some examples of variable declaration:

```
string my_string;
int i;
```

Assignment Statement

Users can assign variables to a value using the assignment operator. The types of the variable has to match the types of the value that users want to assign the variables to.

Variable type	Description and Examples

`int`	Users can assign variables of type int to integer values. Example:
	`int myVar;`
	`myVar = 3;`

`string`	Users can assign strings to characters within quotes. Example:
	`string myString;`
	`myString = "hello";`
	Users can also assign strings to an array. Users can modify the values within each index of the array they same way they would modify values within an array in Java or C. The following example demonstrates the creation of a string as an array of 2 characters, and assigning the string to say "hi".
	`string myString;`
	`myString = new[2];`
	`mystring[0] = 'h';`
	`mystring[1] = 'i';`

`char`	Users can assign variables of type character to a single character. Example:
	`char myChar;`
	`myChar = 'c';`

Keyword Statements

Name	Description and Examples

`for`	`for` provides a way to iterate over a range of values.
	
	This `for` loop prints out all odd numbers less than 10. The `for` loop has three fields. First is the initiator. Unlike C, variables cannot be declared here. The second field is for the test condition. The last field is the action to take at the end of the loop.
	`int odd;`
	`odd = 1;`
	`for (odd; odd < 10; odd=odd+2){`
	`  print_i(i);`
	`}`

`while`	`while` provides a way to repeatedly execute a statement while condition is true.
	
	This `while` loop prints out “in while loop” while the condition index > 0 remains true. This will print out the string “in while loop” five times.
	`int index;`
	`index = 5;`
	`while(index > 0) {`
	`  print_s(“in while loop”);`
	`  index = index - 1;`
	`}`

`if... else...`	`if` and `else` provide a way to execute a certain section of code under certain conditions. If `else{}` is missing from the conditional, then the compiler will throw an error. Users can leave the space between `{` and `}` if they do not wish to perform any action in the else case. This example shows that if `index` is greater than 0, it prints “positive”, otherwise, it prints “negative”.

	`if (index > 0) {`
	`  print_s("positive");`
	`}`
	`else {`
	`  print_s("negative");`
	`}`

`#`	`#` provides a way to comment codes. To comment out a line of code, just add `#` to the beginning of the line. This is an example of a line of comment. `# This is a comment`
`new`	Used for declaring arrays.

Scope

Scope refers to which variables and functions are accessible at a given point of the program.

Blocks

Each block is defined by a set of curly braces, `{` and `}`.

Variables in Scope

Variables can be declared locally inside a function or a block. This means that they can be used inside the function or the block. Variables within its own scope must have consistent type. If users try to assign it to a different type, the compiler will throw a syntax error. The code below will throw a syntax error because variable `var` is declared as a string, but later assigned to an int.

```
int main() {
    string var;
    var = "yay";
    var = 3; #this is not
allowed
    return 0;
}
```

Introduction to Functions

Main ( )

Each Tape program requires a `main()` function. Without `main()`, the program will not compile. Users can write their program in the body of main `main()`, or, they can write necessary functions after `main()`, and call those functions from `main()`.

`main()` is declared with a return type in the following format:

```
int main() {
    # do something
    return 0;
}
```

Calling Functions

To call a function, users simply have to write the function name with the correct input values and end it with a semicolon. If the functions have return values and users want to assign those values to variables, users have to make sure that the variables are of the same type as the return values.

User Defined Functions

Users can declare their own functions in their program. To do so, users need the following information: return type, function name, input types, and action of the function. The following example shows a user defined function called `print_hello`, which prints the string “hello” if the input equals to 1 and does nothing otherwise

```
void print_hello(int index) {
    if (index == 1) {
        print(“hello”);
    }
    else {}
}
```

Users can call these user-defined functions in the `main()` or in other user-defined functions. An example of how users can call the above function within `main()` is shown below. When the program runs, it will print nothing since the input value is not equal to 1.

```
int main() {
    print_hello(3);
    return 0;
}
```

Note that in order to write a user-defined function, users have to include a `main()` function at the top of the file. Without `main()`, the program will not compile.

Built-in Functions

Tape has a number of built-in functions. Each library accepts a certain number of arguments. If users omit or adds extra arguments, they will get an error message. These functions can be called by users in their own program.

print_i

```
void print_i(int x)
```

This is the basic print statement for integers. It does not return anything.

Example:

```
print_i(42);
```

Output:

```
42
```

print_s

```
void print_s(string x)
```

This is the basic print statement for strings. It does not return anything.

Example:

```
print_s("PLT");
```

Output:

```
PLT
```

print_c

```
void print_c(char x)
```

This is the basic print statement for char. It does not return anything.

Example:

```
print_c('c');
```

Output:

```
c
```

tolower

```
char tolower(char x)
```

This function take a input char `x` and returns a lowercase version of the character.

toupper

```
char toupper(char x)
```

This function take a input char `x` and return a uppercase version of the character.

fget

```
string fget(string x, int w, string z)
```

This function reads one line (until a `/n`) from with size smaller or equal to `w` from file `z` and copies it into string `x`. Returns null if nothing is read.

write

```
int write(string x, int w, int y, string z)
```

This function writes to a file `z` from `x`, each time `w` bytes of element for `y` times.

read

```
string read(string x, int w, int y, string z)
```

This function read from file `z` to `x`, each time `w` bytes of element for `y` times.

find

```
string find(string x,string z)
```

This function returns a pointer to the first appearance in `x` of any of the entire sequence of characters specified by `z`. Returns null if not found.

cpy

```
string cpy(string x, string z, int w)
```

This function copies `w` characters from `z` to `x`. The count starts from the very first character of string `z`.

length

```
int length(string x)
```

This function returns the length of string `x`.

open

```
open(string file, string attribute)
```

Open is used to open files in Tape. The string file is the name of the file. The attributes are as follows:

"r", open a file for reading
"w", create an empty file and erase the file if it exist
"a", append the material at the back of the file, will create the file if it is not exist
"r+", open a file for both reading and writing
"w+", create an empty file for read and write
"a+", open a file for both reading and appending

The following example is a program that opens the file `myList.txt` in order to add or delete items on the list.

```
string newFile;
File z;
newFile = "myList.txt"
Z = open(newFile, "w");
```

Tape Standard Library

Tape has a standard library of functions. Each library accepts a certain number of arguments. If users omit or adds extra arguments, they will get an error message. These functions can be called by users in their own program. To see how these library functions were implemented, please refer to `stdlib.tape` in the Appendix.

countWord

```
int countWord(string handle, string sourceFile)
```

This searches for the string `handle` in the `sourceFile`, which is the string representation of the path to the source file, and returns an integer value of the number of occurrences of `handle`.

Below is an example of a program that counts the number of occurrences of the word “apple” in the file `originalFile.txt` and prints out that number.

`originalFile.txt:`

```
Today is December 14th. I ate an apple for breakfast.
Then, at 1pm, I ate another apple
I also ate an apple after dinner.
```

`countExample.tape`

```
int main(){
  string handle;
  string target;
  int i ;
  target = "originalFile.txt";
  handle="apple";

  i = countWord(handle,target);
  print_s(“Number of apples: “);
  print_i(i);

  return 0;
}
```

Output:

```
Number of apples:
3
```

findreplace

`findreplace(string source, string search, string replace, string newFile)`

This searches for the string `search` in string `source`. Every occurrence of string search is replaced with string `replace`, and the result is saved in string `newFile`.

Below is an example of a program that replaces all occurrences of the string “ an apple” in the file `originalFile.txt` with the string “Professor Edwards” and saves the altered text in a file called `destinationFile.txt` .

`originalFile.txt:`

```
Today is December 14th. I ate an apple for breakfast.
Then, at 1pm, I ate another apple
I also ate an apple after dinner.
```

countExample.tape

```
int main(){
  string handle;
  string newword;
  string target;

  target = "originalFile.txt";
  handle="apple";
  newword="Professor Edwards";

  findreplace("an apple",newword, target, "destinationFile.txt");

  return 0;
}
```

After running the program, `destinationFile.txt` will look like this:

```
Today is December 14th. I ate Professor Edwards for
breakfast.
Then, at 1pm, I ate another apple
I also ate Professor Edwards after dinner.
```

tape

`int tape(string fn, string re)`

The tape function will open the file `fn` and search the string `re` line by line in the file. If result found, return the line that `re` occur. Before do searching, the function will analysis the input `re` to check if it is a normal string or a regular expression start with '/':

The input variable re can be a normal string between two double quotes, such as "hello". Then it will call searchLine(file f, string a) function to do a normal word searching.

The input variable re can be one of a type defined regular expressions in form of "/string/op/string/", where op can be *, |, &, ?, -, ^.

When tape function is called with re in form of regular expression, it will check what regular expression operator it has. Then call the corresponding searching function to do the work.

The corresponding searching function are list below:

```
int searchAnd(file f, string re)
int searchOr(file f, string re)
int searchEOR(file f, string re)
int searchQ(file f, string re)
int searchBetween(file f, string re)
Int searchKleene(file a, string re)
```

indexOf

```
int indexOf(string t, char c)
```

This function searches the string `t` for the first occurrence of the character `c`, and returns the position where that occurrence begin in `t`.

substring

```
string substring(int index1, int index2, string s)
```

This function returns a length-character-long substring of `s1`. It starts at the index `index1` of `s` and ends at `index2` of `s`.

searchLine

```
int searchLine(file f, string a)
```

This function searches for the occurrence of string `a` in file `f`. If `a` is found, the function prints out the lines in `f` that contain `a`.

str2Upper

```
string str2Upper(string a)
```

This takes the string word and changes all lowercase letters to uppercase.

str2Lower

```
string str2Lower(string a)
```

This takes the string word and changes all uppercase letters to lowercase.

mergeString

```
string mergeString(string a, string b)
```

This function takes in two strings and concatenates the two strings. The returning string will be a string that starts with `a` and ends with `b`.

appendChar

```
string appendChar(string s, char a)
```

This takes string `s` and character `a` and returns a string that is the same as string `s` but has the character a appended to it.

strEq

```
int strEq(string a, string b)
```

This takes in two strings and checks whether they are equivalent. If `a` and `b` are the same, the function returns `1`, otherwise, it returns `0`.

4. Project Plan

Process

As a team, we first started from understanding the MicroC implementation examples showed in Professor Stephen Edward's Programming Languages and Translators course. Then, with those examples, we built our own parser, scanner, AST, and codgen with the basic functionalities we wanted Tape to have. The more we understood those files the more we were able to implement our own features. This includes implementing additional types such as arrays and using C library functions. After this was all done, we implemented library functions that would be helpful in file manipulation using the Tape language.

What we found extremely helpful in the process was having very specific demo products in mind while we were creating Tape. By having these demos in mind, we were able to think about the functionalities that users may find very helpful, and focus on implementing those features.

Initial Proposal vs. Final Product

Our initial proposal for this project was to create a language that directly mimics AWK, another file manipulating language. However, as time went on, we ended up molding Tape into a language that has similar syntax to Java. We chose to do this because most developers, including us, are more familiar with Java. We want Tape to be an intuitive language in which users would not have to spend a lot of time learning new syntax.

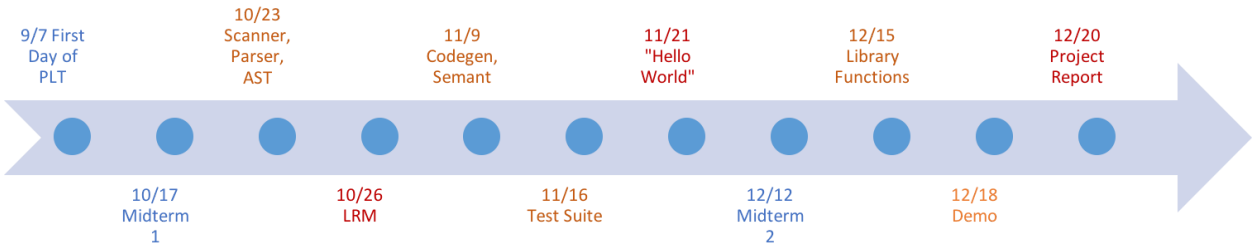
Style Guide

1. 4 spaces for indentation in all files.
2. Comments for specific code should be placed above those lines or after those lines of code.
3. Variable names consist of lowercase characters and underscores such as `myvariable` or `my_variable`.
4. After using a semicolon, `;`, start the following code in the next line.
5. "`in`" keyword (most common in `semant.ml`) is placed at a single line.

Project Plan

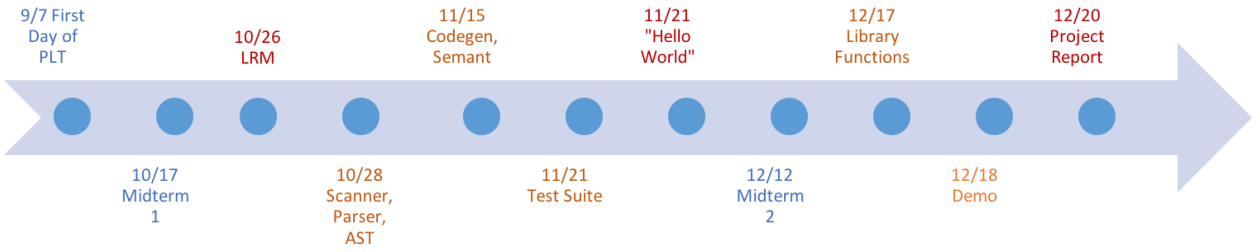
Our plan was to work on Tape consistently, not just towards the end of semester right before the presentation because we understand that it is a large project that we would not be able to complete within one sitting. We created a timeline around the deadlines for the Language Reference Manual and "Hello World" as shown below.

The red dates indicate hard deadlines, blue dates indicate other events related to PLT, and the orange dates indicate soft deadlines that we set for ourselves.



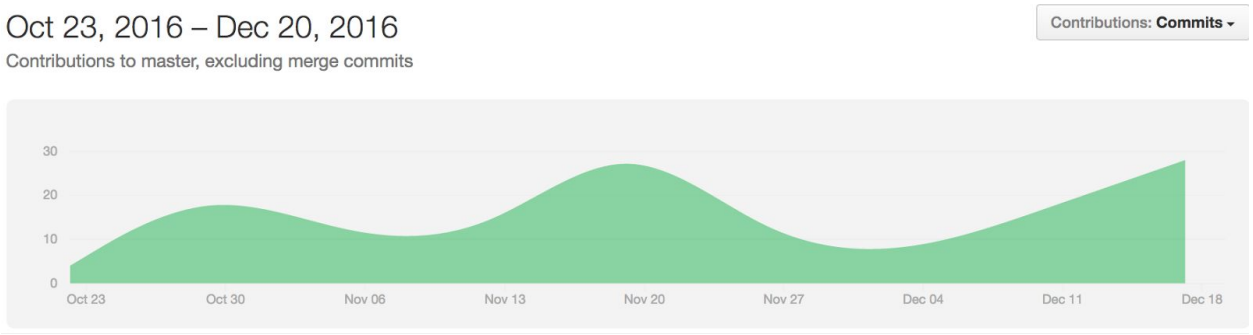
Project Log

We were able to follow our plan pretty closely, however, we were running a bit behind by a few days for each milestone that we set for ourselves. We believe this happened due to unexpected issues and bugs we encountered that we we were not able to solve immediately. As shown in the project log below, we had a harder time getting started with the scanner, parser, and AST. This was because we had some issues trying to get the environment to work. In addition to that, we were still getting to know how LLVM works and what we want Tape to be. As we moved towards the end of the semester, we had a better sense of what our vision for Tape is, so we had an easier time meeting our milestone deadlines within one grace day or two.



Commit Log

We were able to work pretty consistently throughout the semester as indicated by the commit log in our Github repository as shown in the graph below. We do, however, acknowledge that we tend to work extra hard right before the hard deadlines.



Roles and Responsibility

Below were the roles and responsibilities that we assigned to each member of the team before starting the project:

Tianhua Fang: System Architect
Test Suite, Demo, Scanner/Parser, AST, Codegen, Semantics
Alexander Sato: Language Guru
Test Suite, Demo, Scanner/Parser, AST, Codegen, Semantics
Priscilla Wang: Project Manager
Documentation, Demo, Scanner/Parser, AST, Codegen, Semantics
Edwin Chan: Tester
Test Suite, Scanner/Parser, AST, Codegen, Semantics

As the semester passes by, the roles of each of the team members remained the same, however, the responsibilities changed. Each team member was able to learn more about what they are good at and start taking more responsibilities in that field. This is what the roles and responsibilities for each member ended up being by the end of the semester:

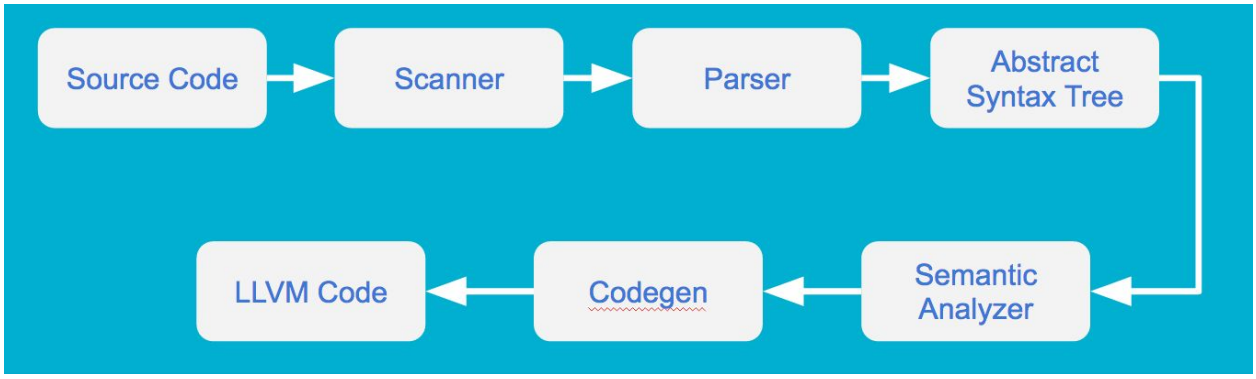
Tianhua Fang: System Architect
Some tests, Library Functions, Demo, Scanner/Parser, AST, Codegen, Semantics
Alexander Sato: Language Guru
Some tests, Scanner/Parser, AST, Codegen
Priscilla Wang: Project Manager
Some tests, Documentation, Library Functions, Demo, Scanner/Parser, AST
Edwin Chan: Tester
Test Suite, Library Functions, Documentation, Scanner/Parser, AST, Codegen, Semantics

Software Development Environment

For this project, we used Ocaml and LLVM 3.6 on Ubuntu 15.20. Prior to running a Tape program, please make sure that you have the correct environment set up. Instructions on how to setup your environment can be found in `setup.md` in the Appendix.

5. Architectural Design

Block Diagram



About the Architecture

The sources code first passes through the scanner and be translated into tokens. Next, the tokens are passed into the Parser and AST for building up the syntax tree.

The semantic analyzer performs a static analysis on the syntax tree to detect any problem about code structure, type matching, duplicate name, function argument and return type.

The tree that passes the semantic checking will passed to codegen.ml for generating the assembly code. Finally, the assembly code is compiled into LLVM for execution.

Files

To see the source code of the different components of the architecture, please refer to the following files in the Appendix:

```
scanner.mll parser.mly ast.ml semant.ml codegen.mll
```

Who Did What?

All four team members contributed to the architecture of Tape, however, Edwin Chan and Tianhua Fang took the lead.

6. Test Plan

We tested the Tape language in the following two ways:

1. Manually compiling a file, running that program, and checking to see if the output is what we expected it to be.
2. Running a test suite.

These tests were ran all throughout the development of Tape to ensure that new implemented features were not affecting the rest of the language.

Unit Testing

Unit tests were the first kind of test that we used since it tests the smallest testable parts of the application. Unit tests are designed to be short and focused so we can quickly determine where the error lies. An example of a unit test was simply compiling and running a file with a `main()` function and nothing else.

Integration Testing

Integration testing allows us to build bigger programs and see whether all the smaller components of the language work well together. This was extremely important because we were able to find and fix features of our language that may have been fine during unit tests but incompatible when used together in a program.

Example 1: While Loop

This is a simple test program that prints 5, 4, 3, 2, 1 using a while loop. This would be an example of a unit test because it tests a small component of the language. Below is the code for a while loop test.

```
int main(){

   int a;
   a=5;
   while (a>0){
       print_i(a);
       a=a-1;
   }
   return 0;
}
```

This is the output:

```
5
4
3
2
1
```

Example 2: fget

This is a test program that looks through a file, and prints every line in the array. Below is the code:

```
int main()
{
  file filea;
  string array;
  string null;
  int i;

  null = find("","a");

  array = new[10000];

  filea = open("test2.txt", "r");
  while(fget(array,1000,filea) !=
null){
    print_s(array);
  }

  return 0;
}
```

The output for the following:

```
Testing for read

Second line for read

Third line

end
```

Test Suite

Throughout the development of Tape, we added tests (such as the ones seen in the two examples above) in our test suite that was called by the command `sh testall.sh`. This test suite allows the system to compile all the programs in the suite, run all of the programs, and check that the output of the program is the same as the expected output. A single line is printed to stdout indicating whether the test has passed

or failed. If a tests case did not passed, a testall.log will generated and record the test record.

We also included failed cases in the test suite. Failed cases are there to ensure that the system will throw the correct error message when users make a mistake in their code.

Having a test suite was helpful because as the number of features of Tape grows, the more things we as developers of the language have to test for, and running the tests one by one would no longer be feasible. We ended up having around 100 test cases for Tape covering various features such as variable declaration, file manipulation, user defined functions, and use of library functions.

To see how `testall.sh` is implemented, please refer to the Appendix.

Who Did What?

Edwin Chan was the major contributor of the test suite. However, the other three members of the group also contributed to the test suite.

7. Learned from Course

The major take-away for this project is that communication and discussions are extremely important. When we first started this project, we had a completely wrong idea about what writing our own language even means. We knew we wanted a file manipulation language, but we did not know how to execute it correctly. By discussing our proposal with our TA, we were able to get a better sense of what Tape is, what should be part of the Tape language, and what kind of library functions should be written in tape. The TA was also able to provide us with various examples of existing languages that we could have drawn inspirations from.

Along the same line as communication, consistently asking for help from each other and from the TA simplified our lives much more. In the beginning, when we were stuck on a problem, we would hesitate to ask other team members for help. However, we realized that by having a fresh pair of eyes look at our code, we could find bugs more more easily. Same thing goes for asking the TA for help. Often times, the TA had already spoken to other teams who experienced similar issues. Most of the teams end up resolving these issues, so the TA would have a better sense of how to resolve the issues that our team encounters.

Like previous groups, we also suggest future groups to start on the project early in the semester because there will be so many issues that come out throughout the semester that we cannot fix overnight. In addition to that, we recommend future groups to start testing their project early and with small test cases. We found that it was much easier to find bugs when we start by running small programs using tape.

Last but not least, we recommend that future groups have a very clear sense of what they want their demo to be. We were able to narrow our focus significantly once we figured out what we wanted our demo to do. There are so many features that we envisioned Tape to have, however, we only have so much time in a school semester, so it is very important to pinpoint the more significant features and focus on making those features work.

8. Appendix

setup.md

```
The tape compiler

Coded in OCaml and compiles it into LLVM IR.

It needs the OCaml llvm library, which is most easily installed through opam.

Install LLVM and its development libraries, the m4 macro preprocessor,
and opam, then use opam to install llvm.

The version of the OCaml llvm library should match the version of the LLVM
system installed on your system.

------------------------------
Installation under Ubuntu 15.10

LLVM 3.6 is the default under 15.10, so we ask for a matching version of the
OCaml library.

sudo apt-get install -y ocaml m4 llvm opam
opam init
opam install llvm.3.6 ocamlfind
eval `opam config env`

make
./testall.sh

------------------------------
Installation under Ubuntu 14.04

The default LLVM package is 3.4, so we install the matching OCaml
library using opam.  The default version of opam under 14.04 is too
old; we need to use a newer package.

sudo apt-get install m4 llvm software-properties-common

sudo add-apt-repository --yes ppa:avsm/ppa
sudo apt-get update -qq
sudo apt-get install -y opam
opam init

eval `opam config env`

opam install llvm.3.4 ocamlfind

------------------------------
Installation under OS X
```

```
1. Install Homebrew:

   ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"

2. Verify Homebrew is installed correctly:

   brew doctor

3. Install opam:

   brew install opam

4. Set up opam:

   opam init

5. Install llvm:

   brew install llvm

   Take note of where brew places the llvm executables. It will show
   you the path to them under the CAVEATS section of the post-install
   terminal output. For me, they were in /usr/local/opt/llvm/bin. Also
   take note of the llvm version installed. For me, it was 3.6.2.

6. Have opam set up your enviroment:

   eval `opam config env`

7. Install the OCaml llvm library:

   opam install llvm.3.6

   Ensure that the version of llvm you install here matches the
   version you installed via brew. Brew installed llvm version 3.6.2,
   so I install llvm.3.6 with opam.

   IF YOU HAVE PROBLEMS ON THIS STEP, it's probably because you are
   missing some external dependencies. Ensure that libffi is installed
   on your machine. It can be installed with

   brew install libffi

   If, after this, opam install llvm.3.6 is still not working, try
   running

   opam list --external --required-by=llvm.3.6

   This will list all of the external dependencies required by
   llvm.3.6. Install all the dependencies listed by this command.
```

```
   IF THE PREVIOUS STEPS DO NOT SOLVE THE ISSUE, it may be a problem
   with using your system's default version of llvm. Install a
   different version of llvm and opam install llvm with that version
   by running:

   brew install homebrew/versions/llvm37
   opam install llvm.3.7

   Where the number at the end of both commands is a version different
   from the one your system currently has.

8. Create a symbolic link to the lli command:

   sudo ln -s /usr/local/opt/llvm/bin/lli /usr/bin/lli

   Create the symlink from wherever brew installs the llvm executables
   and place it in your bin. From step 5, I know that brew installed
   the lli executable in the folder, /usr/local/opt/llvm/bin/, so this
   is where I symlink to. Brew might install the lli executables in a
   different location for you, so make sure you symlink to the right
   directory.

   IF YOU GET OPERATION NOT PERMITTED ERROR, then this is probably a
   result of OSX's System Integrity Protection.

   One way to get around this is to reboot your machine into recovery
   mode (by holding cmd-r when restarting). Open a terminal from
   recovery mode by going to Utilities -> Terminal, and enter the
   following commands:

   csrutil disable
   reboot

   After your machine has restarted, try the `ln....` command again,
   and it should succeed.

   IMPORTANT: the prevous step disables System Integrity Protection,
   which can leave your machine vulnerable. It's highly advisable to
   reenable System Integrity Protection when you are done by
   rebooting your machine into recovery mode and entering the following
   command in the terminal:

   csrutil enable
   reboot

   Another solution is to update your path, e.g.,

   export PATH=$PATH:/usr/local/opt/llvm/bin

   A third solution is to modify the definition of LLI in testall.sh to
   point to the absolute path, e.g., LLI="/usr/local/opt/llvm/bin/lli"

9. To run and test, navigate to the Tape folder. Once there, run
```

```
   make ; ./testall.sh

   Tape should build without any complaints and all tests should
   pass.

   IF RUNNING ./testall.sh FAILS ON SOME TESTS, check to make sure you
   have symlinked the correct executable from your llvm installation.
   For example, if the executable is named lli-[version], then the
   previous step should have looked something like:

   sudo ln -s /usr/local/opt/llvm/bin/lli-3.7 /usr/bin/lli

   As before, you may also modify the path to lli in testall.sh

------------------------------
To run and test:

$ make
ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 tape.native
Finished, 22 targets (22 cached) in 00:00:01.

$ ./testall.sh
test-appendchar1...OK
test-appendchar2...OK
test-countword1...OK
test-countword2...OK
test-countword3...OK
...
fail-print_i...OK
fail-return1...OK
fail-tolower...OK
fail-toupper1...OK
```

ast.ml

```
type typ = Int | Char | String | Void | Bool
type op = Plus | Minus | Times | Equal | Less | LessEQ | Great | GreatEQ | Unequal

type uop = Not

type crement = INCREMENT | DECREMENT

type bind = typ * string

type expr = Literal of int
       | StringLit of string
       | Binop of expr * op * expr
       | Assign of string * expr
```

```
       | Unop of uop * expr
       | Noexpr
       | BoolLit of bool
       | Call of string * expr list
       | NewstringLit of string
       | Char_Lit of char
       | Array of string * expr
       | Arrayassign of string * expr * expr
       | Init of string * expr
       | Arrayaccess of string*string*expr
type stmt = Block of stmt list
   | Expr of expr
   | If of expr * stmt * stmt
   | For of expr * expr * expr * stmt
   | While of expr * stmt
   | Return of expr

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


type program = bind list * func_decl list

let string_of_op = function
       Plus -> "+"
  | Minus -> "-"
  | Times -> "*"
  | Equal -> "=="
 (* | Neq -> "!="*)
  | Less -> "<"
  | LessEQ -> "<="
  | Great -> ">"
  | GreatEQ -> ">="
  | Unequal -> "!="

let string_of_uop = function
       (* Neg -> "-" *)
  Not -> "!"

let string_of_typ = function
       Int    -> "int"
  | Bool   -> "bool"
  | Void   -> "void"
  | String -> "string"
  | Char -> "char"
let rec string_of_expr = function
       Literal(l) -> string_of_int l
  | BoolLit(true) -> "true"
```

```
  | BoolLit(false) -> "false"
  | NewstringLit(s) -> s
  | Char_Lit(s) -> Char.escaped s
  | StringLit(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Array(v,e) -> v^"["^ string_of_expr e ^ "]"
  | Arrayassign(v,e1,e2) -> v ^ "[" ^ string_of_expr e1 ^ "]" ^ string_of_expr e2
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  | Init(v,e) -> v^ "=" ^ "new" ^ "[" ^ 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
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1 ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3 ^ ")" ^ string_of_stmt s
  | While(e,s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

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

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^ "}\n"


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

codegen.ml

```
module L = Llvm
module A = Ast
module Fcmp = Llvm.Fcmp
module StringMap = Map.Make(String)

let translate(globals,functions) =
      let context = L.global_context () in
      let the_module = L.create_module context "tape"

      and i32_t  = L.i32_type context
```

```
    and i8_t   = L.i8_type context
    and i1_t   = L.i1_type context (*bool*)
    and ptr_t  = L.pointer_type (L.i8_type context)
    and void_t = L.void_type context in

       let ltype_of_typ = function
      A.Int -> i32_t
    | A.String -> ptr_t
    | A.Void -> void_t
    | A.Bool -> i1_t
       | A.Char -> i8_t

       (*There may have more things need to be put*)
       in
       (*global variable*)
       let global_vars =
    let global_var m (t,n) =
       let init = L.const_int (ltype_of_typ t) 0
       in StringMap.add n (L.define_global n init the_module) m in
    List.fold_left global_var StringMap.empty globals in

       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


       (*declare external function printf*)
       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

       let prints_t = L.var_arg_function_type ptr_t [|L.pointer_type i8_t|] in
       let prints_func = L.declare_function "puts" prints_t the_module in

       (*file open and close*)
       let open_file_t = L.function_type ptr_t [| L.pointer_type i8_t;L.pointer_type
i8_t |] in
       let open_file_func = L.declare_function "fopen" open_file_t the_module in

       let close_file_t = L.function_type i32_t [| i32_t |] in
       let close_file_func = L.declare_function "fclose" close_file_t the_module in

       let write_t = L.function_type i32_t [| i32_t; ptr_t |] in
       let write_func = L.declare_function "fputs" write_t the_module in

       let get_t = L.function_type ptr_t [|ptr_t; i32_t; ptr_t|] in
```

```
    let get_func = L.declare_function "fgets" get_t the_module in

    let fwrite_t = L.function_type i32_t [|ptr_t; i32_t; i32_t; ptr_t|] in
    let fwrite_func = L.declare_function "fwrite" fwrite_t the_module in

    let read_t = L.function_type i32_t [|ptr_t; i32_t; i32_t; ptr_t|] in
    let read_func = L.declare_function "fread" read_t the_module in

    let toupper_t = L.function_type i8_t [| i8_t |] in
    let toupper_func = L.declare_function "toupper" toupper_t the_module in

    let tolower_t = L.function_type i8_t [| i8_t |] in
    let tolower_func = L.declare_function "tolower" tolower_t the_module in

    let calloc_t = L.function_type ptr_t [|i32_t; i32_t|] in
    let calloc_fun = L.declare_function "calloc" calloc_t the_module in

    let strfind_t = L.function_type ptr_t [|ptr_t;ptr_t|] in
    let strfind_func = L.declare_function "strstr" strfind_t the_module in

    let memcpy_t = L.function_type ptr_t [|ptr_t; ptr_t; i32_t|] in
    let memcpy_func = L.declare_function "memcpy" memcpy_t the_module in

    let strlen_t = L.function_type i32_t [|ptr_t|] in
    let strlen_func = L.declare_function "strlen" strlen_t the_module in

    (*build function body - 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 = (*create an instruction builder*)
    L.builder_at_end context (L.entry_block the_function) in
  let int_format_str = (*Format string for printf calls*)
    L.build_global_stringptr "%d\n" "fmt" builder in
    let char_format_str = L.build_global_stringptr "%c\n" "fmt" builder in

    (* formals and locals  *)
    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 (t, n) =
    let local_var = L.build_alloca (ltype_of_typ t) n builder
    in StringMap.add n local_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

    (*look up  for a variable among locals/formal arguments, then the golbals*)
    let lookup n = try StringMap.find n local_vars
```

```
            with Not_found -> StringMap.find n global_vars
      in


      (*expression*)
      let rec expr builder = function
     A.Literal i -> L.const_int i32_t i   (*boolean not included*)
   | A.Noexpr ->    L.const_int i32_t 0
      | A.Char_Lit c -> L.const_int i8_t (Char.code c)
      | A.StringLit s -> L.build_load (lookup s) s builder
      | A.NewstringLit sl -> L.build_global_stringptr sl "string" builder
      | A.Array(e1,e2) -> let para1=(expr builder (A.StringLit e1))
      and  para2=(expr builder e2) in
      let k=L.build_in_bounds_gep para1 [|para2|] "tmpp" builder in
      L.build_load k "deref" builder
      | A.Arrayassign(e1,e2,e3) -> let para1 = (expr builder (A.StringLit e1))
      and para2= (expr builder e2)
      and para3= (expr builder e3)
      in let k=L.build_in_bounds_gep para1 [|para2|] "tmpp" builder in
      L.build_store para3 k builder
      | A.Init(e1,e2) -> let cnt1=(lookup e1) and cnt2= expr builder e2 in
      let tp= L.element_type (L.type_of cnt1) in
      let sz=L.size_of tp in
      let sz1=L.build_intcast sz (i32_t) "intc" builder in
      let dt=L.build_bitcast (L.build_call calloc_fun [|cnt2;sz1|] "tmpa" builder)
tp "tmpb" builder in
      L.build_store dt cnt1 builder
      | A.Assign (s,e) -> let e' = expr builder e in
       ignore (L.build_store e' (lookup s) builder); e'
   | A.Binop (e1, op, e2) ->
      let e1' = expr builder e1
      and e2' = expr builder e2 in
      (match op with
       A.Plus -> L.build_add
      | A.Minus -> L.build_sub
      | A.Times -> L.build_mul
      | A.Equal -> L.build_icmp L.Icmp.Eq  (*Fcmp is a module imported, no unequal*)
      | A.Unequal -> L.build_icmp L.Icmp.Ne
      | A.Less  -> L.build_icmp L.Icmp.Slt
      | A.Great -> L.build_icmp L.Icmp.Sgt
      | A.LessEQ -> L.build_icmp L.Icmp.Sle
      | A.GreatEQ -> L.build_icmp L.Icmp.Sge
      ) e1' e2' "tmp" builder
   | A.Unop(op, e) ->
      let e' = expr builder e in
      (match op with
       A.Not -> L.build_not) e' "tmp" builder

   (*build in function filled below*)
      | A.Call("print_i",[e]) ->
            L.build_call printf_func [|int_format_str; (expr builder e)|]
            "printf" builder
```

```
      | A.Call("print_s",[e]) ->
            L.build_call prints_func [|(expr builder e)|]
            "puts" builder
      | A.Call("print_c",[e]) ->
            L.build_call printf_func [|char_format_str; (expr builder e)|]
            "printf" builder


      | A.Call("toupper",[e]) ->
            L.build_call toupper_func [|(expr builder e)|]
            "toupper" builder

      | A.Call("tolower",[e]) ->
            L.build_call tolower_func [|(expr builder e)|]
            "tolower" builder
      | A.Call("write",e) -> let actuals= List.rev (List.map (expr builder)
(List.rev e)) in
            L.build_call fwrite_func (Array.of_list actuals) "tmpy" builder

      | A.Call("open", e) -> let actuals= List.rev (List.map (expr builder)
(List.rev e)) in
            L.build_call open_file_func (Array.of_list actuals) "fopen" builder
      | A.Call("fget",e) -> let actuals= List.rev (List.map (expr builder) (List.rev
e)) in
            L.build_call get_func (Array.of_list actuals) "tmpz" builder

      | A.Call("read", e) -> let actuals = List.rev (List.map (expr builder)
(List.rev e)) in
            L.build_call read_func (Array.of_list actuals) "tmpx" builder

      | A.Call("find", e)-> let actuals = List.rev (List.map (expr builder)
(List.rev e)) in
            L.build_call strfind_func (Array.of_list actuals) "find" builder

      | A.Call("cpy",e) -> let actuals = List.rev (List.map (expr builder) (List.rev
e)) in
            L.build_call memcpy_func (Array.of_list actuals) "mcpy" builder

      | A.Call("length",e) -> let actuals = List.rev (List.map (expr builder)
(List.rev e)) in
            L.build_call strlen_func (Array.of_list actuals) "len" builder

      | A.Call (f, act) ->
      let (fdef, fdecl) = StringMap.find f function_decls in

  let actuals = List.rev (List.map (expr builder) (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
      (*statements*)
      let add_terminal builder f =
    match L.block_terminator (L.insertion_block builder) with
```

```
    Some _ -> ()
   | None -> ignore (f builder) in
    let rec stmt builder = function
A.Block sl -> List.fold_left stmt builder sl

   | A.Expr e -> ignore (expr builder e); builder
   | A.Return e -> ignore (match fdecl.A.typ with
  A.Void -> L.build_ret_void builder
  | _ -> L.build_ret (expr builder e) builder); builder
   (*If Statement*)
   | A.If (predicate, then_stmt, else_stmt) ->
   let bool_val = expr builder 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
   (*While Statement *)
   | 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 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
   (*For Statements*)

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

   let builder = stmt builder (A.Block fdecl.A.body) in

   add_terminal builder (match fdecl.A.typ with
   A.Void -> L.build_ret_void
   | t-> L.build_ret(L.const_int (ltype_of_typ t) 0))

   in
```

```
      List.iter build_function_body functions;
      the_module
```

Makefile

```
.PHONY: tape.native

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

.PHONY: clean
clean:
      ocamlbuild -clean
      rm -rf scanner.ml parser.ml parser.mli
      rm -rf *.cmx *.cmi *.cmo *.cmx *.o
      rm -rf testall.log *.diff *.err *.ll
OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx tape.cmx

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

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 $<

ast.cmo:
ast.cmx:
codegen.cmo: ast.cmo
codegen.cmx: ast.cmx
tape.cmo   : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
tape.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


TARFILES = ast.ml codegen.ml Makefile microc.ml parser.mly README scanner.mll \
                semant.ml testall.sh $(TESTFILES:%=tests/%)
tape-llvm.tar.gz: $(TARFILES)
       cd .. && tar czf tape-llvm/tape-llvm.tar.gz \
       $(TARFILES:%=tape-llvm/%)
```

parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
%token IF ELSE WHILE FOR RETURN
%token PLUS MINUS TIMES ASSIGN BOOL TRUE FALSE
%token EQUAL LESS LESSEQ GREAT GREATEQ UNEQUAL
%token NOT NEW
%token INT CHAR VOID STRING

%token <int> LITERAL
%token <string> STRINGLIT NEWSTRINGLIT
%token <char> CHAR_LITERAL
%token EOF

%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQUAL UNEQUAL
%left MATCH
%left LESS GREAT LESSEQ GREATEQ
%left PLUS MINUS
%left TIMES
%right NEG NOT

%start program
%type <Ast.program> program

%%
program: decls EOF  { $1 }

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

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


typ: INT { Int }
   | VOID { Void }
   | BOOL { Bool }
   | STRING { String }
   | CHAR {Char}
vdecl_list: /*nothing*/ {[]}
   | vdecl_list vdecl {$2 :: $1}

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


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 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: LITERAL { Literal($1) }
       | STRINGLIT { StringLit($1) }
       | NEWSTRINGLIT { NewstringLit($1) }
       | STRINGLIT ASSIGN expr { Assign($1, $3) }
       | CHAR_LITERAL {Char_Lit($1)}
       | expr PLUS expr { Binop($1, Plus, $3) }
       | expr MINUS expr { Binop($1, Minus, $3) }
       | expr TIMES expr { Binop($1, Times, $3) }
       | expr EQUAL expr { Binop($1, Equal, $3) }
       | expr UNEQUAL expr {Binop($1, Unequal, $3)}
       | expr LESS expr { Binop($1, Less, $3) }
       | expr GREAT expr { Binop($1, Great, $3) }
       | expr LESSEQ expr { Binop($1, LessEQ, $3) }
       | expr GREATEQ expr { Binop($1, GreatEQ, $3) }
       | NOT expr { Unop(Not, $2) }
       | TRUE { BoolLit(true)}
       | FALSE { BoolLit(false)}
       | STRINGLIT LPAREN actual_opt RPAREN { Call($1, $3) }
       | STRINGLIT LBRACKET expr RBRACKET { Array($1,$3)}
       | STRINGLIT ASSIGN NEW LBRACKET expr RBRACKET {Init($1,$5)}
       | STRINGLIT LBRACKET expr RBRACKET ASSIGN expr {Arrayassign($1,$3,$6)}

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

```
actual_opt: /* nothing */ { [] }
       | actual_list { List.rev $1 }

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

scanner.mll

```
{ open Parser

  let unescape s =
       Scanf.sscanf ("\""^ s ^ "\"") "%S%!" (fun x->x)
}

let ascii=[' '-'!' '#'-'[' ']'-'~' ]
let escape= '\\' ['\\' ''' '"' 'n' 'r' 't']
let digit=['0'-'9']
let char='''(ascii|digit)'''
let escape_char='''(escape)'''
let newstring= '"' ((ascii|escape)* as s) '"'

rule token = parse
  [' ' '\t' '\r' '\n' '\\' ] {token lexbuf} (* Whitespace *)
| "(" {LPAREN}
| ")" {RPAREN}
| "{" {LBRACE}
| "}" {RBRACE}
| "[" {LBRACKET}
| "]" {RBRACKET}
| "void" {VOID}
| "true" {TRUE}
| "false" {FALSE}
| "if" {IF}
| "else" {ELSE}
| "while" {WHILE}
| "for" {FOR}
| "return" {RETURN}
| "bool" {BOOL}
| ';' {SEMI}
| '+' {PLUS}
| '-' {MINUS}
| '*' {TIMES}
| ',' {COMMA}
| '=' {ASSIGN}
| "new" {NEW}
| "==" {EQUAL}
| "!=" {UNEQUAL}
| '<' {LESS}
| "<=" {LESSEQ}
```

```
| '>' {GREAT}
| ">=" {GREATEQ}
| '!' {NOT}
| "char" {CHAR}
| "int" {INT}
| "string" {STRING}
| "file" {STRING}
| ['0'-'9']+ as lxm {LITERAL(int_of_string lxm)}
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { STRINGLIT(lxm)}
| char as lxm {CHAR_LITERAL( String.get lxm 1)}
| escape_char as lxm{CHAR_LITERAL(String.get (unescape lxm) 1)}
| newstring {NEWSTRINGLIT((unescape s))}
| eof {EOF}
| "#" { comment lexbuf } (* Comments *)

and comment = parse
  "\n" { token lexbuf }
| _ { comment lexbuf }
```

semant.ml

```
open Ast

module StringMap = Map.Make(String)

let check (globals, functions) =
  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

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

  let check_assign lvaluet rvaluet err =
       if lvaluet == rvaluet then lvaluet else raise err
  in

  (**** Checking Global Variables ****)
  List.iter (check_not_void (fun n -> "illegal void global " ^ n))
             globals;

  report_duplicate (fun n -> "duplicate global " ^ n)
                    (List.map snd globals);
```

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

  (*Check for duplicate. 2 functions cannot have same name, therefore also does not
allow overload*)
  report_duplicate (fun n -> "duplicate function " ^ n)
       (List.map (fun fd -> fd.fname) functions);

  (* Function declaratoin for a named function (build in function) *)
  (* Use 2 array to hold the details then throw to the built_in_decls by list.fold *)
  let built_in_decls_funcs = [

       { typ = Char; fname = "tolower"; formals = [(Char, "x")]; locals = []; body =
[]};

       { typ = Char; fname = "toupper"; formals = [(Char, "x")]; locals = []; body =
[]};

       { typ = Void; fname = "print_i"; formals = [(Int, "x")] ; locals = []; body =
[] };

       { typ = String; fname = "fget"; formals=[(String,"x");(Int,"y");(String,
"z")]; locals=[];body=[]};

       { typ = String; fname = "open"; formals = [(String, "x");(String,"x")]; locals
= []; body = [] };

       { typ = Int; fname = "write"; formals = [(String,
"x");(Int,"y");(Int,"z");(String, "a")]; locals = []; body = [] };

       { typ = Void; fname ="print_c" ; formals=[(Char, "x")]; locals=[]; body=[]};

       { typ = String; fname = "read" ; formals=[(String,"x");(Int, "w");(Int,
"y");(String, "z")]; locals=[]; body=[]};

       { typ =String; fname="find"; formals=[(String,"x");(String,"y")]; locals=[];
body=[]};

       { typ =String; fname="cpy"; formals=[(String, "x");(String, "y");(Int,"z")];
locals = []; body = []};

       { typ = Int; fname="length"; formals=[(String,"x")]; locals=[]; body=[]};
  ]

  in

  let built_in_decls_names = [ "tolower"; "toupper"; "print_i";"fget"; "open";
"write";"print_c";"read";"find";"cpy";"length"];

  in

  let built_in_decls = List.fold_right2 (StringMap.add)
```

```
                    built_in_decls_names
                    built_in_decls_funcs
                    (StringMap.singleton "print_s"
                                { typ = Void; fname = "print_s"; formals =
[(String, "x")]; locals = []; body = [] })
       in
  let function_decls =
       List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
                    built_in_decls functions
  in

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

  (*ensure "main" is defined*)
  let _=function_decls "main" in
  let check_function func=

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

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


  (*variable symbol Table*)
  let symbols = List.fold_left
        (fun m(t,n) -> StringMap.add n t m)
        StringMap.empty
        ( globals @ func.formals @ func.locals )
  in

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

  (*expression-return the type of an expression or throw an exception*)
  let rec expr = function
    Literal _ -> Int
       | BoolLit _ -> Bool
```

```
      | Char_Lit _ -> Char
      | StringLit s -> type_of_identifier s
      | NewstringLit _ -> String
      | Binop(e1, op, e2) as e-> let t1 = expr e1 and t2 = expr e2 in
      (match op with
      Plus -> (match(t1,t2) with (Int,Int)-> Int
                                 |(Char,Char) -> String)
      | Minus | Times when t1 = Int && t2 = Int -> Int
      | Equal | Unequal  when t1=t2 -> Bool
      | Less | Great |LessEQ|GreatEQ when t1 = Int && t2 = Int -> Bool
      | _ -> raise(Failure ("illegal binary operator "^
            string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
            string_of_typ t2 ^ " in " ^ string_of_expr e))
      )
      | Init(var, lit) -> let a = type_of_identifier var and b= expr lit in
            (match b with Int -> a
                  | _ -> raise (Failure("illegal "^ string_of_typ b ^", expected
int")))
      | Noexpr -> Void
      | Array(var, _)-> let k=function ptr_t -> Char
                                       | _->  raise(Failure("illegal argument of an
array"))
                  in k (type_of_identifier var)
      | Assign(var, e) as ex -> let lt = type_of_identifier var
                                and rt = expr e in
      check_assign lt rt (Failure ("illegal assignment " ^ string_of_typ lt ^
                  " = "  ^ string_of_typ rt ^ " in " ^ string_of_expr ex))
      | Arrayassign(var, _ , _ ) -> let k=function ptr_t -> Char
                                       | _ -> raise(Failure("illegal assign of
array"))
                                in k (type_of_identifier var)

      | Unop(op, e) as ex -> let t = expr e in
      (match op with
      Not when t = Bool -> Bool
      | _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op ^
            string_of_typ t ^ " in " ^ string_of_expr ex)))
      | Call(fname, actuals) as call -> let fd = function_decls 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
      List.iter2 (fun (ft,_) e -> let et = expr e in
            ignore (check_assign ft et
            (Failure ("illegal actual argument found " ^
            string_of_typ et ^ " expected " ^
            string_of_typ ft ^ " in " ^ string_of_expr e ))))
            fd.formals actuals;
      fd.typ

      in
      (*statment*)
      let check_bool_expr e = if expr e != Bool
```

```
      then raise (Failure ("expected Boolean expression in " ^ string_of_expr e))
      else () in

      let rec stmt = function
      Expr e -> ignore (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

      | 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))
      | Block s1 -> let rec check_block = function
                   [Return _ as s] -> stmt s
      | Return _ :: _ ->
             raise (Failure "nothing may follow a return")
      | Block s1 :: ss -> check_block (s1 @ ss)
      | s :: ss -> stmt s ; check_block ss
      | [] -> ()
      in check_block s1

  in stmt (Block func.body) (*body of check_function*)

  in List.iter check_function functions (*body of check*)
```

stdlib.tape

```
int main(){


    return 0;
}

int countWord(string a, string f){
       string temp;
       string null;
       file target;
       int intcount;
       intcount = 0;

       target=open(f, "r");
       temp=new[1000000];
       null=find("","a");

       while(fget(temp,1000,target)!=null){
       if(find(temp,a)!=null){
              intcount = intcount + 1;
       }else {

       }
       }
       print_i(intcount);
       return intcount;

}

int tape(string fn, string re){
       char key;
       file f;

       string null;
       null=find("","a");

       f=open(fn,"r");
       if(re[0]=='/'){
       if(find(re,"&")!=null){
              searchAnd(f,re);
       }else{}

       if(find(re,"|")!=null){
              searchOr(f,re);
       }else{}

       if(find(re,"?")!=null){
              searchQ(f,re);
       }else{}
```

```
	if(find(re,"^")!=null){
		searchEOR(f,re);
	}else{}

	if(find(re,"-")!=null){
		searchBetween(f,re);
	}else{}

	if(find(re,"*")!=null){
		searchKleene(f,re);
	}else{}
	}else{
	searchLine(f,re);
	}

	return 0;
}

int searchAnd(file f, string re){
	int idx;
	int lt;
	string t1;
	string t2;
	string temp;

	string null;
	null=find("","a");

	lt=length(re);
	idx=indexOf(re,'&');
	t1=substring(1,idx-2,re);
	t2=substring(idx+2,lt-2,re);
	temp=new[1000];
	while(fget(temp,1000,f)!=null){
	if(find(temp,t1)!=null){
		if(find(temp,t2)!=null){
		print_s(temp);
		}else{}
	}else{}

	}



	return 1;
}

int searchOr(file f, string re){
	int idx;
	int lt;
	string t1;
	string t2;
```

```
	string temp;

	string null;
	null=find("","a");

	lt=length(re);
	idx=indexOf(re,'|');
	t1=substring(1,idx-2,re);
	t2=substring(idx+2,lt-2,re);
	temp=new[1000];
	while(fget(temp,1000,f)!=null){
	if(find(temp,t1)!=null){
		print_s(temp);
	}else{
		if(find(temp,t2)!=null){
		print_s(temp);
		}else{}

	}

	}
	return 1;
}

int searchEOR(file f, string re){
	int idx;
	int lt;
	string t1;
	string t2;
	string temp;

	string null;
	null=find("","a");

	lt=length(re);
	idx=indexOf(re,'^');
	t1=substring(1,idx-2,re);
	t2=substring(idx+2,lt-2,re);
	temp=new[1000];
	while(fget(temp,1000,f)!=null){
	if(find(temp,t1)!=null){
		if(find(temp,t2)==null){
		print_s(temp);
		}else{}
	}else{
		if(find(temp,t2)!=null){
		print_s(temp);
		}else{}

	}

	}
```

```
	return 1;
}


int searchQ(file f, string re){
	int idx;
	int lt;
	string t1;
	string t2;
	string temp;
	int idx2;
	string t3;
	int len1;
	int len2;
	int len3;
	int idx3;

	string null;
	null=find("","a");

	lt=length(re);
	idx=indexOf(re,'?');
	t1=substring(1,idx-2,re);
	t2=substring(idx+2,lt-2,re);
	len1=length(t1);
	len2=length(t2);
	len3=len1+len2;
	t3=new[len3];
	idx2=0;
	idx3=0;
	while(idx2<len1){
	t3[idx3]=t1[idx2];
	idx3=idx3+1;
	idx2=idx2+1;
	}

	idx2=0;
	while(idx2<len2){
	t3[idx3]=t2[idx2];
	idx3=idx3+1;
	idx2=idx2+1;
	}

	temp=new[1000];



	while(fget(temp,1000,f)!=null){
	if(find(temp,t1)!=null){
		print_s(temp);
	}else{
```

```
              if(find(temp,t2)!=null){
              print_s(temp);
              }else{}

       }

       }


       return 1;
}

int searchBetween(file f, string re){
       int idx;
       int lt;
       string t1;
       string t2;
       string temp;


       #to find the intermediate string
       int begin;
       int end;
       string middle;
       string ans;
       string head;
       string tail;
       int len1;
       int len2;
       int len;
       int lent2;

       string null;
       null=find("","a");


       lt=length(re);
       idx=indexOf(re,'-');
       t1=substring(1,idx-2,re);
       t2=substring(idx+2,lt-2,re);

       temp=new[1000];
       head=new[1000];
       tail=new[1000];
       ans=new[1000];

       while(fget(temp,1000,f)!=null){
       head=find(temp,t1);
       if(head!=null){
              tail=find(temp,t2);
              if(tail!=null){
              len1=length(head);
              len2=length(tail);
```

```
              end=len1-len2;
              lent2=length(t2);
              end=end+lent2;
              ans=substring(0,end,head);
              print_s(ans);
              }else{}
       }else{}

       }



       return 1;
}


int  searchKleene(file a, string re){

       int idx;
       int la;
       int lre;
       int len;
       int lft;
       int lsn;
       int l;
       int right;
       int i;
       int j;
       int ltemp;
       char key;
       string ft;
       string sn;

       string temp;
       string word;
       string snmerge;
       string iffind;
       string null;

       null=find("","a");



       idx=indexOf(re,'*');
       lre=length(re);
       ft=substring(1,idx-3, re);
       sn=substring(idx+2, lre-2, re);
       key=re[idx-2];

   # print_c(key);

       la=length(a);
       lft=length(ft);
```

```
	lsn=length(sn);
	#/ab/*/c/
	l=lsn+lft-1;

	j=0;
	i=0;
	right=1;
	temp=new[1000];
	while(fget(temp,1000,a)!=null){
	while(l<=10){
		word=new[1000];
		word=mergeString(word,ft);

		while(j<i){
		word=appendChar(word,key);
		j=j+1;
		}

		word=mergeString(word,sn);
		if(find(temp,word)!=null){
		print_s(word);
		}else{
		word=new[1000];
		}

		j=0;
		i=i+1;
		l=l+1;
	}
	i=0;
	l=lsn+lft-1;
	j=0;
	}
	return 1;
}




int indexOf(string t, char c){
	int idx;
	int l;
	int ans;

	l=length(t);
	idx=0;

	ans=0;
	while(idx<l){
	if(t[idx]==c){
		ans=idx;
```

```
	}else{}

	idx=idx+1;
	}

	return ans;
}


string substring(int begin, int end, string s){
	string ans;
	string temp;
	int sublength;
	int len;
	int i;
	sublength=end-begin+1;
	len=length(s);

	ans=new[sublength];
	temp=new[len];
	temp=s;

	i=0;
	for(begin;begin<end+1;begin=begin+1){
	ans[i]=temp[begin];
	i=i+1;
	}

	return ans;
}

int searchLine(file f, string a){
	string temp;
	string null;

	temp=new[1000000];

	null=find("","a");

	while(fget(temp,1000,f)!=null){
	if(find(temp,a)!=null){
		print_s(temp);
	}else {

	}
	}

	return 1;

}

string str2Upper(string a){
	string temp;
```

```
	string tempup;
	string target;

	int i;
	string ans;
	char c;
	int l;

	l=length(a);

	target=new[l];
	target=a;
	temp=new[1];
	tempup=new[1];
	i=0;

	ans=new[l];
	while(i<l){
	temp[0]=target[i];
	tempup[0]=toupper(temp[0]);
	ans[i]=tempup[0];
	i=i+1;
	}



	return ans;
}

string str2Lower(string a){
	string temp;
	string tempup;
	string target;

	int i;
	string ans;
	char c;
	int l;

	l=length(a);

	target=new[l];
	target=a;
	temp=new[1];
	tempup=new[1];
	i=0;

	ans=new[l];
	while(i<l){
	temp[0]=target[i];
	tempup[0]=tolower(temp[0]);
	ans[i]=tempup[0];
	i=i+1;
```

```
	}



	return ans;
}

string mergeString(string a, string b){
	string temp;
	int la;
	int lb;
	int l;
	int i;
	int j;

	la=length(a);
	lb=length(b);

	l=la+lb;
	temp=new[l];

	i=0;
	j=0;
	while(j<la){
	temp[j]=a[i];
	i=i+1;
	j=j+1;
	}
	i=0;
	while(j<l){
	temp[j]=b[i];
	j=j+1;
	i=i+1;

	}

	return temp;
}

string appendChar(string s, char a){
	int ls;
	string temp;
	int i;
	i=0;
	ls=length(s);
	temp=new[ls+1];

	while(i<ls){
	temp[i]=s[i];
	i=i+1;
	}
	temp[ls]=a;
```

```
      return temp;

}

int findreplace(string a, string b, string orig, string dest){
  string temp;
  string newword;
  string null;
  int la;
  int lb;
  file origfile;
  file destfile;
  string index;
  string new_index;
  string ans_beg;
  string ans;

  int len_line;
  int len_index;
  int len_before_find;
  int len_diff;
  int len_ans;
  string line;
  int index_temp;
  string line_before_find;


  origfile=open(orig, "r");
  destfile=open(dest, "w");

  la=length(a);
  lb=length(b);
  newword=new[lb];
  newword=b;
  temp=new[100000];
  null=find("","a");

  line=fget(temp,1000,origfile);
  while(line!=null){
  len_line=length(line);
       index=find(temp,a);
       if(index!=null){
       len_index=length(index);
       len_before_find=len_line-len_index;
       if(la==lb){
       index_temp=0;
       while(index_temp < lb) {
       temp[len_before_find] = newword[index_temp];
       index_temp=index_temp+1;
       len_before_find=len_before_find+1;
       }
       write(temp,1,len_line,destfile);
       }
```

```
        else {
        new_index=substring(la, len_index, index);
        ans=mergeString(newword,new_index);
        ans_beg=substring(0, len_before_find-1, temp);
        ans=mergeString(ans_beg, ans);
        len_ans=length(ans);

        write(ans,1,len_ans,destfile);
        }
        }else {}
        line=fget(temp, 1000, origfile); #get next line
  }

  return 1;
}

int strEq(string a, string b){
        int la;
        int lb;

        int ans;
        int i;

        ans=1;
        i=0;
        la=length(a);
        lb=length(b);
        if(la==lb){
        while(i<la){
                if(a[i]!=b[i]){
                ans=0;
                }else{}
                i=i+1;
        }
        }else{ ans=0; }

        return ans;
}
```

testall.sh

```
#!/bin/sh

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

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

TAPE="./tape.native"
#TAPE="_build/tape.native"

# Set time limit for all operations
ulimit -t 30

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

keep=0

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

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

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

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

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

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

       echo -n "$basename..."

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

       generatedfiles=""

       generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
       Run "$TAPE" "<" $1 ">" "${basename}.ll" &&
       Run "$LLI" "${basename}.ll" ">" "${basename}.out" &&
       Compare ${basename}.out ${reffile}.out ${basename}.diff

       # Report the status and clean up the generated files

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

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

       echo -n "$basename..."
```

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

       generatedfiles=""

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

       # Report the status and clean up the generated files

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

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

shift `expr $OPTIND - 1`

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

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


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

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

exit $globalerror
```