TED Programming Language Final Report

Gideon Mendels, Konstantin Itskov, Theodore Ahlfeld, Matthew Haigh
{gm2597, koi2104, twa2108, mlh2196} @columbia.edu

1. Introduction

The TED programming language is a web-parsing language designed to simplify web scraping and serve as a bridge between complex high-level web-scraping languages like Javascript and imperative programming languages like C. The syntax of Ted combines both types of languages into one. TED programs are written in a C-like style language, that ultimately compiles into intel-based x86 64-bit assembly, which is then linked together with other C libraries, to be converted into machine code.

The primary object in a TED program is a Page which represents a website. All other functions are designed to allow the programmer to access and manipulate data within that page. This makes the Document Object Model structure of websites easier to access for programmers who are not experts in CSS and Javascript, while also providing enough functionality for experienced web developers to obtain and manipulate data.

2. Background

Web scraping is a complex process by which a computer program automatically collects data from across the internet. In this process, the program exchanges a large number of messages with the server in order to find and acquire data. Because of the intricacies involved with the exchange, new high level programming languages, such as javascript, python and PHP, rose to dominate the internet. Unlike their low level counterparts C/C++, these programming languages introduced new, easy to use, and easy to understand interfaces that allowed programmers to develop automated internet tools without having to worry about string manipulation or http protocol message interchange. However, what these high level languages lack is speed and the ability to connect directly with the operating system. Due to their scripting nature, these languages require a native virtual machine or interpreter that creates a layer of indirection.

Web scraping has become increasingly important as modern computers have been created with a significant amount of computational power. With this computational power, computers can now sift through massive quantities of data in a timely manner, which has in turn increased the demand by users for the automated analysis of larger amounts of data. Web scraping has become common practice, because it is the only method for aggregating large amounts of public information across the internet.

Given the importance of web scraping, our team developed a low-level language (TED) with the ability to scrape data from the web, while eliminating the need to use cumbersome interfaces for web manipulation. In order to make an optimal language, we incorporated ways to easily connect to the internet and extract data from web pages. We developed TED to manipulate web strings using CSS selection, similar to the way that languages, such as AWK, manipulate strings using regular expressions.

3. Language Tutorial

3.1. Setup and Dependencies

Before compilation can be utilized a number of dependencies must be installed on the target system. Our programming language is depended on a set of programs listed in the software development environment section of the project plan chapter. Please refer to that information for the system requirements specifications. However, for the purpose of this tutorial please follow either the instructions below for install the dependencies or simply execute the `setup.sh` file found inside of `src/` folder.

```
> sudo apt-get install phantomjs nodejs nodejs-legacy npm
> sudo npm install phantom -g
> sudo npm install phantom
```

3.2. Using the Compiler

Inside the `src/` folder, type `make` which creates the TED compiler executable `ted`, which, given a `.ted` file in the TED language, compiles an executable file of the same name (without the `.ted` extension). A TED program `hello_world.ted` must first be compiled using the following sequence of commands:

```
> ted hello_world.ted (produces assembly .asm file)
> nasm -f elf64 filename.asm (produces object file)
> gcc hello_world.o -I../libparse -L../libparse -lm -lparse -o hello_world (produces
executable binary file)
```

Alternatively, type the following command and the same set of commands will execute:

```
> ./tedc.sh hello_world
```

The following sample TED code demonstrates the following features:

- The mandatory *main* function, with type int and return value 0 (must return int)
- Calling the built-in `print` function, with output *stdout* selected
- A string with new-line escape sequence

```
int main() {
  print(stdout, "Hello World!\n");
}
```

After compiling and running the program above you should expect the output to be:

```
> Hello World!
```

3.3. Data Manipulation

TED's declaration and assignments work as follows:

- A newly-declared variable is preceded by its type, which can be int, string, List, Page, Element, or file

- All variable declarations must occur before function calls.

```
int main() { /* main is a function that returns int */
  /* this is a comment */
  /* comments can span multiple lines */
  int i;
  int j = 21;
  i = 21 + j;
  print(stdout, "%d\n", i);
  return 0;
} /* every function must return a value */
```

The output is:

```
> 42
```

3.4. Lists

Lists are linked lists which can contain data of any type. There is a small library of list functions in TED: listNew, listAddAfter, listAddLast, listRemove, listConcat, listHead, and listTail

```
int main() {
  List list; /*all variable declarations occur in the beginning of a function */
  List list1;
  List list2;
  int i;
  int j;

  list = listNew(); /* creates a new empty list */
  listAddLast(list, 1); /* calling listAddLast on an empty list creates a list head */
  listAddAfter(list, 2);  /* appends int 2 after the element list */
  listRemove(list, 0); /* removes whatever is in the 0th location (int 1) */
  list1 = listNew();
  listAddLast(list1, 3); /* appends element to end of list */
  list2 = listConcat(list, list1); /* concatenate list1 to the end of list and store in
list2 */

  for (i = 0; i < 2; i = i + 1) {
    j = listHead(list2); /* listHead returns the data in the first item of a list */
    print(stdout, "%d", j);
    list2 = listTail(list2); /* listTail returns the list without the first element */
  }

  return 0;
}
```

The output is:

```
> 13
```

3.5. Page

Page is the primary data element in TED. The user creates a Page by invoking the built in pageFetch function. There are 4 built in functions to retrieve Page data: pageURL, pageHTML, pageRoot, and pageFind

```
int main() {

  Page p;
  string s;
  string s1;
  string s2;
  List list;
  Element e;

  p = pageFetch("http://example.com"); /* retrieve the data from the website */
  s = pageURL(p); /* the url */
  s1 = pageHTML(p); /* the full html data */
  list = pageFind(p, "h1"); /*returns a list of all html type h1 as Element objects*/
  e = listHead(list);
  s2 = elementText(e); /* returns the text inside of the h1 */
  print(stdout, "%s\n\n%s\n\n%s\n", s, s1, s2);

  return 0;
}
```

The output is:

```
http://example.com

<html><head>\n    <title>Example Domain</title>\n\n    <meta charset=\"utf-8\">\n    <meta
http-equiv=\"Content-type\" content=\"text/html; charset=utf-8\">\n    <meta
name=\"viewport\" content=\"width=device-width, initial-scale=1\">\n    <style
type=\"text/css\">\n    body {\n        background-color: #f0f0f2;\n        margin: 0;\n
padding: 0;\n        font-family: \"Open Sans\", \"Helvetica Neue\", Helvetica, Arial,
sans-serif;\n        \n    }\n    div {\n        width: 600px;\n        margin: 5em auto;\n
padding: 50px;\n        background-color: #fff;\n        border-radius: 1em;\n    }\n
a:link, a:visited {\n        color: #38488f;\n        text-decoration: none;\n    }\n
@media (max-width: 700px) {\n        body {\n            background-color: #fff;\n
}\n        div {\n            width: auto;\n            margin: 0 auto;\n
border-radius: 0;\n            padding: 1em;\n        }\n    }\n    </style>
\n</head>\n\n<body>\n<div>\n    <h1>Example Domain</h1>\n    <p>This domain is established
to be used for illustrative examples in documents. You may use this\n    domain in examples
without prior coordination or asking for permission.</p>\n    <p><a
href=\"http://www.iana.org/domains/example\">More
information...</a></p>\n</div>\n\n\n<script
src=\"http://code.jquery.com/jquery-1.11.3.min.js\"></script></body></html>

Example Domain
```

3.6. Element

Element represents an element of the HTML Document Object Model. The built in functions for working with Elements are elementChildren, elementType, elementText, and ElementAttr

```
int main() {

  Page p;
  Element e;
  Element f;
  List l;
  List l1;
  List l2;
  string s;
  string t;

  /* the final variable declaration can use a function */
  Page p = pageFetch("http://example.com");

  e = pageRoot(p); /* pageRoot will usually be the "head" element" */
  l = elementChildren(p, e);
  e = listHead(l);
  l1 = listTail(l);
  f = listHead(l1);
  s = elementText(f);
  print(stdout, "%s/n", s);
  return 0;
}
```

The output is:

```
     Example Domain
     This domain is established to be used for illustrative examples in documents. You may use this
     domain in examples without prior coordination or asking for permission.
     More information...
```

4. Reference Manual

4.1. Lexical Elements

This chapter describes the lexical elements that make up a TED source code file. Lexical elements will be referred to as tokens. These tokens will be categorized as: Identifiers, Keywords, Constants, Operators, Separators, and Whitespace.

4.1.1. Identifiers

Identifiers are sequences of characters used for naming variables, functions, or data types. These identifiers can include letters, digits, and underscores '_', but must begin with a letter. Identifiers are case sensitive.

4.1.2. Keywords

Keywords are special identifiers reserved for the compiler and the language itself.

```
if, else, for, while, int, string, list, FILE, Page, Element, return
```

4.1.3. Constants

A constant is a literal value. All constants belong to a language defined data types.

4.1.3.1. Integer Constants

All integer constants must be in the base ten number system. Any integer value prefix with a hyphen '-' is treated as a negative integer.

4.1.3.2. String Constants

TED does not allow character manipulation explicitly. To prevent unsafe string manipulation that has plagued both C and C++, strings are a primitive type. String constants are a double quoted sequence of characters.

```
"Example"
"String Constants"
```

4.1.3.3. String Escape Constants

String escape constants are two character sequences with a backslash followed by a character. Escape sequences are as follows

```
Backslash:                   "\\"
Single quotation mark:       "\'"
Double quotation mark:       "\""
Newline:                     "\n"
```

4.1.4. Operators

An operator is a special token that performs either a unary or binary operation. They can perform operations such as addition, complements, etc. Full coverage of operation is provided in chapter 3, “Expressions and Operators”.

4.1.5. Separators

Separators are single characters which separate tokens such as:

```
 ( ) { } ; , . :
```

In addition white spaces, which are still considered separators, and are not considered tokens and are ignored during parsing.

4.1.6. Whitespace

Whitespace is any permutation of either spaces, tabs, newline, carriage return, or form feeds. White space is ignored, outside of string constants, and is therefore considered as an option with an exception of separating tokens. White space is not required to separate operators from their operand, and any whitespacing between an operand and operators are ignored.

4.1.7. Comments

Comments are indicated by the symbol “/*”. Any input read after a comment symbol will be ignored until the end symbol “*/” is encountered. Nested comments are not allowed.

4.2. Data Types

4.2.1. Primitive Data Types

4.2.1.1. Integer Data Types

int : The 32-bit int data type can hold integer values in the range of −2,147,483,648 to 2,147,483,647

4.2.1.2. String Data Types

string is a sequence of 8 bit ASCII characters followed by a 0 terminating character.

4.2.1.3. List Data Types

List is a linked list of memory references to objects. Can refer to any type but all objects in a list must be of the same type.

4.2.2. Object Data Types

4.2.2.1. Element Data Types

Element represents an element of the HTML Document Object Model. The built in functions for working with Elements are elementChildren, elementType, elementText, and elementAttr. elementType, elementText, and elementAttr are invoked by providing the Element as an argument to the function. elementChildren requires as arguments the page and the element and returns a list of all of the DOM children of an element. elementType returns the HTML type associated with the Element. elementText returns the inner text value of the HTML element. elementAttr returns a string that represents all of the attributes of the HTML element.

4.2.2.2. Page Data Types

Page is the primary data element in TED. The user creates a Page by invoking the built in pageFetch function as demonstrated in section 3.5. There are 4 built in functions to retrieve Page data: pageURL, pageHTML, pageRoot, and pageFind. pageURL, pageHTML, and pageRoot are invoked by providing the Page as an argument to the function. pageFind requires two arguments: the Page and the CSS selector to be invoked. pageURL returns a string representing the URL of the page. pageHTML returns a string representing the full HTML text. pageRoot returns and Element object for the html element of the DOM, namely the root node or top level element. pageFind searches for the CSS selector entered by the user, using the standard webkit model.

4.3. Expressions and Operators

4.3.1. Expressions

Expressions consist of at least one operand and zero or more operators. Operands are typed objects, such as constants, variables, and function calls that must return values (non void functions). Examples:

```
42
40 + 2
f(42)  /* This assumes function f returns a non void value */
```

Parentheses group subexpressions in order to override default precedence.

```
(2 + 3) * (5*(2+2))/4 - 0
```

Innermost expressions are evaluated first. In the above example, 2+2 is first evaluated to be 4 before being multiplied by 5. Next, 2 + 3 = 5 is calculated, followed by the multiplication of the already calculated 5 * 20 to be 100, which is then divided by 4, ended with a minus 0.

4.3.2. Assignment Operators

Assignment operators store values in variables.

The standard assignment operator = simply stores the value of its right operand in the variable specified by its left operand. As with all assignment operators, the left operand cannot be a literal or constant value. For example:

```
int x = 10;
int y = 45 + 2;
string a = "ted";
```

4.3.3. Arithmetic Operators

TED provides operators for standard arithmetic operations: addition, subtraction, multiplication, and division. Usage of these operators is straightforward; here are some examples:

```
/* Addition. */
x = 5 + 3;

/* Subtraction. */
x = 5 - 3;

/* Multiplication. */
x = 5 * 3;

/* Division. */
x = 5 / 3;
```

Integer division of positive values truncates towards zero, so 5/3 is 1.

4.3.4. Comparison Operators

The equal-to operator == tests its two operands for equality. The result is true if the operands are equal, and false if the operands are not equal.

```
if (x == y)
    print (stdout, "%s", "x is equal to y");
else
    print (stdout, "%s", "x is not equal to y");
```

The not-equal-to operator != tests its two operands for inequality. The result is true if the operands are not equal, and false if the operands *are* equal.

```
if (x != y)
    print (stdout, "%s", "x is not equal to y");
else
    print (stdout, "%s", "x is equal to y");
```

Beyond equality and inequality, there are operators you can use to test if one value is less than, greater than, less-than-or-equal-to, or greater-than-or-equal-to another value. Here are some code samples that exemplify usage of these operators:

```
if (x < y)
    print (stdout, "%s", "x is less than y");

if (x <= y)
    print (stdout, "%s", "x is less than or equal to y");

if (x > y)
    print (stdout, "%s", "x is greater than y");

if (x >= y)
    print (stdout, "%s", "x is greater than or equal to y");
```

4.3.5. Function Calls as Expressions

A call to any function which returns a value is an expression.

```
int function(do something);
...
a = function();
```

4.3.6. The Comma Operator

The comma operator is used in for statements as follows:

```
/* Using the comma operator in a for statement. */
for (x = 0;  x <= 10;  x = x + 1) {
    ...
}
```

This allows the user to conveniently set, monitor, and modify multiple control expressions for the for statement.

A comma is also used to separate function parameters. Thus,

```
foo (a,  b,  c,  d);
```

is interpreted as a function call with four arguments.

4.3.7. Statements and Declarations in Expressions

Variables must be declared outside of expressions, including for for loops. The incrementer variable must be declared outside of the conditional expression.

```
int i;
for (i = 0; i < 10; i = i +1) {
    do something ...
}

not allowed: b = (int a = 3) // must define a in a separate expression
int add (x,y) { return x + y }
int c = add(a = 3, b  = 2); //c can be declared here but not a or b
```

4.3.8. Operator Precedence

When an expression contains multiple operators, such as a + b * f(), the operators are grouped based on rules of precedence. For instance, the meaning of that expression is to call the function f with no arguments, multiply the result by b, then add that result to a. That is what the TED rules of operator precedence determine for this expression.

The following is a list of types of expressions, presented in order of highest precedence first. Sometimes two or more operators have equal precedence; all those operators are applied from left to right unless stated otherwise.

1. Function calls, array subscripting, and membership access operator expressions.
2. Multiplication, division, and modular division expressions.

3. Addition and subtraction expressions.
4. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions.
5. Equal-to and not-equal-to expressions.
6. All assignment expressions.

4.4. Expressions and Operators

4.4.1. Expression Statements

You can turn any expression into a statement by adding a semicolon to the end of the expression. Here are some examples:

```
5;
2 + 2;
10 >= 9;
```

In each of the examples above, all that happens is that each expression is evaluated. However, they are useless because they do not store a value anywhere, nor do they actually do anything, other than the evaluation itself. The compiler is free to ignore such statements.

Expression statements are only useful when they have some kind of side effect, such as storing a value, calling a function, or (this is esoteric) causing a fault in the program. Here are some more useful examples:

```
x = x + 1;
y = x + 25;
print (stdout, "%s", "Hello, user!");
```

4.4.2. The if Statement

You can use the if statement to conditionally execute part of your program, based on the truth value of a given expression. Here is the general form of an if statement:

```
if (test) {
    then-statement
} else {
    else-statement
}
```

If test evaluates to true, then the then-statement is executed and else-statement is not. If test evaluates to false, then the else-statement is executed and then-statement is not. The else clause is optional. The execution code must be enclosed in braces.

Here is an actual example:

```
if (x == 10) {
    print (stdout, "%s", "x is 10");
}
```

If x == 10 evaluates to true, then the statement print (stdout, “%s”, "x is 10"); is executed. If x == 10 evaluates to false, then the statement print (stdout, “%s”, "x is 10"); is not executed. Here is an example using else:

```
if (x == 10) {
    print (stdout, "%s", "x is 10");
} else {
    print (stdout, "%s", "x is not 10");
}
```

You can use a series of if statements to test for multiple conditions:

```
if (x == 1) {
    print (stdout, "%s", "x is 1");
} else if (x == 2) {
    print (stdout, "%s", "x is 2");
} else if (x == 3) {
    print (stdout, "%s", "x is 13");
} else {
    print (stdout, "%s", "x is something else");
}
```

4.4.3. The while Statement

The while statement is a loop statement with an exit test at the beginning of the loop. Here is the general form of the while statement:

```
while (test) {
    statement
}
```

The while statement first evaluates test. If test evaluates to true, statement is executed, and then test is evaluated again. statement continues to execute repeatedly as long as test is true after each execution of statement. The execution code must be enclosed in brackets. The example below shows a while statement, which prints the integers from zero through nine:

```
int counter = 0;
while (counter < 10) {
    print (stdout, "%d ", counter);
    counter = counter + 1;
}
```

4.4.4. The for Statement

The for statement is a loop statement whose structure allows for easy expression testing, and variable modification. It is very convenient for making counter controlled loops. Here is the general form of the for statement:

```
for (initialize; test; step)
    statement
```

The for statement first evaluates the expression initialize and then it evaluates the expression test. If test is false, then the loop ends and program control resumes after statement. Otherwise, if test is true, then statement is executed. Finally, step is evaluated, and the next iteration of the loop begins with evaluating test again. Most often, initialize assigns values to one or more variables, which are generally used as counters, test compares those variables to a predefined expression, and step modifies those variables' values. Below is another example that prints the integers from zero through nine:

```
int x;
for (x = 0; x < 10; x = x + 1)
    print (stdout, "%d ", x);
```

First, the variable x is declared outside the loop, then it evaluates initialize, which assigns x the value 0. Then, as long as x is less than 10, the value of x is printed (in the body of the loop). Then x is incremented in the step clause and the test re-evaluated. All three of the expressions in a for statement are optional, and any combination of the three is valid. Since the first expression is evaluated only once, it is perhaps the most commonly omitted expression.

4.4.5. Blocks

A block is a set of zero or more statements enclosed in braces. Blocks are also known as compound statements. Often, a block is used as the body of an if statement or a loop statement, to group statements together.

```
for (x = 1; x <= 10; x = x + 1) {
    print (stdout, "x is %d\n", x);
    if ((x - 2) == 0)
        print (stdout, "%d is two\n", x);
    else
        print (stdout, "%d is not two\n", x);
}
```

You can also put blocks inside other blocks:

```
for (x = 1; x <= 10; x = x + 1) {
    if ((x - 2) == 0) {
        print (stdout, "x is %d\n", x);
        print (stdout, "%d is two\n", x);
    } else {
        print (stdout, "x is %d\n", x);
        print (stdout, "%d is not two\n", x);
    }
}
```

Blocks have their own variable scope.

4.4.6. The return Statement

The return statement can be used to end the execution of a function and return program control to the function that called it. Here is the general form of the return statement is as follows:

```
return return-value;
```

Functions must specify return type and must return that type under all conditions.

4.5. Functions

4.5.1. Function Declarations

A function declaration should be written to specify the name of a function, a list of parameters, and the function's return type. A function declaration ends with a semicolon. Here is the general form:

```
return-type function-name (parameter-list);
```

return-type indicates the datatype of the value returned by the function. function-name can be any valid identifier. parameter-list consists of zero or more parameters, separated by commas. A typical parameter consists of a data type and a name for the parameter. Here is an example of a function declaration with two parameters:

```
int zaphod (int x, Element y);
```

The parameter names can be any identifier, and if there is more than one parameter, the same name cannot be used more than once within a single declaration. The parameter names in the declaration need not match the names in the definition.
The function declaration should be written above the first use of the function.

4.5.2. Function Definitions

You write a function definition to specify what a function actually does. A function definition consists of information regarding the function's name, return type, and types and names of parameters, along with the body of the function. All parameters must be assigned to separate local variables within the function body. The function body is a series of statements enclosed in braces; in fact it is simply a block. Here is the general form of a function definition:

```
return-type function-name (parameter-list) {
    parameters to local variables
    function-body
}
```

return-type and function-name are the same as what is used in the function declaration. parameter-list is the same as the parameter list used in the function declaration. Names for the parameters must be included in a function definition. Here is an simple example of a function definition, which take two integers as its parameters and returns the sum of them as its return value:

```
int add_values (int x, int y) {
    int a = x;
    int b = y;
    return a + b;
}
```

4.5.3. Calling Functions

A function can be called by using its name and supplying any needed parameters. Here is the general form of a function is:

```
function-name (parameters);
```

A function call can make up an entire statement, or it can be used as a subexpression. Here is an example of a standalone function call:

```
foo (5);
```

In the above example, the function 'foo' is called with the parameter 5.
Here is an example of a function call used as a subexpression:

```
a = square (5);
```

Supposing that the function 'square' squares its parameter, the above example assigns the value 25 to a.

If a parameter takes more than one argument, the parameters as separated with commas:

```
a = vogon (5, 10);
```

4.5.4. Function Parameters

Function parameters can be literal values or variables. Within the function body, the parameter is a local copy of the value passed into the function; you cannot change the value passed in cannot be changed by changing the local copy.

```
int x = 23;
foo (x);
...
/* Definition for function foo. */
int foo (int b) {
    int a = b;
    a = 2 * a;
    return a;
}
```

In the above example, even though the parameter a is modified in the function ‘foo’, the variable x that is passed to the function does not change. In order to use the function to change the original value of x, the function call would need to be incorporated into an assignment statement:

```
x = foo (x);
```

4.5.5. Built in Functions

TED comes with a small set of built in functions for basic functionality.

4.5.5.1. The open Function

TED offers file I/O. The programmer must explicitly open a file for I/O

The open function opens the file whose name is the string pointed to by filename and associates a stream with it.

The argument mode points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

“r” Open text file for reading. The stream is positioned at the beginning of the file.

“r+” Open for reading and writing. The stream is positioned at the beginning of the file.

“w” Truncate to zero length or create text file for writing. The stream is positioned at the beginning of the file.

"w+” Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

“a” Open for writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file.

“a+” Open for reading and writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file.

To open to a specified file into read and write mode would be.

```
file ted_fp = open("C:\temp\tedoutput.txt", “w+”);
```

4.5.5.2. The print Function

TED offers limited output. The only output function is print, which behaves very similarly to the write system call for linux and prints a string to specified file.

```
print(stdout, "This will output to standard out");
```

To print to a specified file would be similar to

```
file ted_fp = open("C:\temp\tedoutput.txt", “w”);
print(ted_fp, "Prints this string to file");
```

4.5.5.3. The pageFetch Function

TED offers only one way to connect over the network and retrieve pages. The pageFetch function is the only way to create a page. It connects to the specified web address and loads the url into the page structure. The function will create a new HTTP GET request to the specified URL, and save the response to memory. The response http code returned from the server will be saved into the Page structure and the url will be recorded to remember to location the page was scraped from. There get function is called by:

```
Page p = pageFetch(“http address”);
```

4.5.5.4. The pageFind Function

The Page structure is supported by the pageFind function. This function parses through the children list of their data looking for CSS matching. pageFind will employ the entire CSS selection language which allows for easy element selection on the page. The element that is returned from this function is a List Node that could be traversed using the provided List functionalities.

```
Page p;
Element e;
list l;
p = pageFetch("http://www.example.com/");
l = pageFind(p, "class");
e = listHead(l);
```

4.5.5.5. The pageURL Function

The pageURL function extracts the original url that the page was searched on. This variable is preserved in order for the developer to keep track which pages have been scraped and which were not.

```
Page p;
string s;
p = pageFetch("http://www.example.com/");
s = pageURL(p);
```

4.5.5.6. The pageHTML Function

The pageHTML function retrieves the html data that was returned directly from the website in its string from. The developer can then manipulate or print this string to the screen in order to examine the data closer.

```
Page p;
string s;
p = pageFetch("http://www.example.com/");
s = pageHTML(p);
```

4.5.5.7. The pageRoot Function

In order to traverse the html tree the developer must obtain the root element of the page which is returned in the Element data structure. From this element the developer can retrieve children and traverse them.

```
Page p;
Element e;
p = pageFetch("http://www.example.com/");
e = pageRoot(p);
```

4.5.5.8. The elementText Function

This function provides access to the text data contained within a particular DOM element. The data returned is in the string primitive type. For example if the text inside the selected root is `<html>Sample Text</html>` then this function will return `Sample Text`.

```
Page p;
Element e;
string s;
p = pageFetch("http://www.example.com/");
e = pageRoot(p);
s = elementText(e);
```

4.5.5.9. The elementType Function

The elementType function returns the type of the html tag that represents this DOM element. For example if a tag `<div></div>` is selected then the type is DIV.

```
Page p;
Element e;
string s;
p = pageFetch("http://www.example.com/");
e = pageRoot(p);
s = elementType(e);
```

4.5.5.10. The elementAttr Function

The elementAttr function returns the attribute of the html tag that represents this DOM element. For example if a tag `<div class=”blue”></div>` is selected then the attribute return by the search for the class attribute will be `blue`.

```
Page p;
Element e;
string s;
p = pageFetch("http://www.example.com/");
e = pageRoot(p);
s = elementAttr(e, “class”);
```

4.5.5.11. The elementChildren Function

In order for the developer to traverse through the children of a given Element, one must retrieve the list of children that such node contains. The return value of this function is a list Node that can be walked through using the list functions.

```
Page p;
Element e;
string s;
List l;
p = pageFetch("http://www.example.com/");
e = pageRoot(p);
s = elementAttr(e, “class”);
l = elementChildren(p, e);
```

4.5.5.12. The listHead Function

The head function returns the data inside the head of the list

```
listHead(lst);
```

4.5.5.13. The listTail Function

Returns the next element in the list, and will return NULL if it does not exist.

```
listTail(lst);
```

4.5.5.14. The listAddAfter Function

Adds a new element into the list after the specified node. Note: This cannot be used to add the first element. listAddLast must be used for this purpose.

```
listAddAfter(lst, data);
```

4.5.5.15. The listAddLast Function

Adds a new element into the end of the list. Note: This is required to add the first node to the list.

```
listAddLast(lst, data);
```

4.5.5.16. The listRemove Function

Removes an element from the list.

```
listRemove(lst, idx);
```

4.6. Program Structure and Scope

4.6.1. Program Structure

TED can be compiled from a source file with the extension .ted. In order to compile the developer must run the ted compiler which will produce the assembly equivalent. If the developer wants to compile directly to an executable they need to run the tedc.sh script provided together with the language. This script invokes the compilation process of the *.ted source program and compiles it down to an assembly object. The script performs a set of instructions that compile and link the assembly code with the TED extended library. The TED compiler depends on a number of structural dependencies which must be installed on the target system. Please refer to the system requirements in order to confirm the set of installed dependencies in case of use.

4.6.2. Scope

Scope refers to what parts of the program can “see” a declared object. A declared object can be visible only within a particular function, or within a particular file, or may be visible to an entire set of files by way of import statements. Unless explicitly stated otherwise, declarations made at the top-level of a file (i.e., not within a function) are visible to the entire file, including from within functions.

Declarations made within functions are visible only within those functions.
A declaration is not visible to declarations that came before it; for example:

```
int x = 5;
int y = x + 10;
```

will work, but:

```
int x = y + 10;
int y = 5;
```

will not.

5. Project Plan

5.1. Planning Process

Our team had a recurring meeting scheduled once per week, in which we would review the overall milestones set by Professor Edwards, discuss upcoming class deliverables, and determine our short term goals accordingly. The original design of our language was not a compiler, but rather an interpreter; however, after a discussion with Professor Edwards a week prior to the submission of our reference manual, we shifted our focus based on his feedback.

5.2. Specification Process

When we initially determined the specifications, we were planning to create a language that would convert java-like syntax into C-like syntax, with additional functionality for web scraping. Based on this approach, we wrote the proposal and implemented the language reference manual; however, a week prior to our Hello World Demo, we felt that the language we were writing looked like more of a Library extension of a language, rather than being a language itself. After a discussion with Professor Edwards, who suggested that we focus on creating a C to assembly compiler instead, we decided to change direction and implement the same type of web scraping in a different way. This required us to change the language reference manual, as well as all of the other specifications. Through this process, our team learned the importance of continuously revisiting the project specifications, and being flexible and willing to make adjustments when the initial plan does not turn out as expected. Throughout the development process, at each successful implementation point, our language would become increasingly robust, which forced us to revisit the specifications and often add or remove components of the specifications based on the results we were seeing.

5.3. Development Process

Our team first built the tokenizer and the parser, followed by writing the code generation, which allowed for printing and compiling of the Hello World Program. Although we initially attempted to implement all of the web scraping functionality of our language before the Hello World Presentation, we found that our language was missing certain building blocks, such as string manipulation, which were necessary in order to use the web scraping functionality. We therefore ended up building the web scraping functionality and building blocks almost in parallel, simultaneously testing each and seeing what worked in combination. Once these components began communicating with each other, we implemented the SAST, which allowed us to perform type checking on all of the variables. Our language was then mature enough that we were able to use our language and test its functionality.

5.4. Testing Process

We continuously tested functionality throughout the development process, but because we adjusted our language specification part way through the semester, we also needed to adjust our testing accordingly. Because our language is web-oriented, our testing process included fetching pieces of information from the internet and comparing them to our expectations of what the retrieved information should be. If the two aligned, the system would confirm that they were the same, and therefore confirm that a specific functionality of our language worked. Unfortunately, our approach to testing the web turned out to be quite difficult, because often the data being returned included special characters that were hidden within the data, and therefore could not be seen by the human eye. As a result, the assertion information we wrote often would not match the data being fetched, not because the data was necessarily incorrect, but because of the hidden characters. We were therefore required to meticulously examine the data we were calculating when creating the assertion, in order to ensure accurate testing.

5.5. Team Responsibilities

Team Member	Responsibility
Gideon Mendels	Architectural Design
Konstantin Itskov	Project Manager
Theodore Ahlfeld	Coding Guru
Matthew Haigh	Regression Testing

5.6. Project Log

Date	Milestone
September 19	Language proposal and definition
September 26	Proposal writing
September 30	Language proposal deliverable
October 3	Feature deciding documentation
October 14	Grammar definition
October 17	Library specifications and design
October 26	Language reference manual deliverable
October 31	Lexer and parser building
November 7	Project forking to assembly compiler instead of interpreter
November 14	TED to assembly complete generation
November 16	Hello World demo deliverable
November 21	Language control flow and remaining feature completion
November 28	Web library implementation
December 5	Semantic and type checking completion
December 12	Regression testing implementation
December 19	Language and library touch-ups

December 21	Demo day

5.7. Software Development Environment

TED requires a x86_64bit Ubuntu Linux environment, since the code is assembled into 64 bit x86_64bit assembly

6. Architectural Design

6.1. The Compiler

6.1.1. scanner.mll

The file containing tokens for the parser. Consist of the ascii representation of the tokens to their actual tokens.

6.1.2. parser.mly

Parser.mly transforms a list of token and creates an abstract syntax tree of functions and expression for the compiler.

6.1.3. ast.mli

Creates the interface for what an abstract syntax tree can consist of.

6.1.4. opcode.ml

Opcodes are an intermediate representation between and AST and actual assembly directors. After a linear list of Opcodes exists it parses each opcode element linearly to produce the NASM syntax like assembly directives.

6.1.5. compile.ml

Compiles the AST into its linear opcode representation.

6.1.6. ted.ml

The main file that passes a file into a the scanner which produces a list of tokens that will be passed to the parse to produce its abstract syntax tree. It follows up by ensuring all the variable types, argument types, and operator types match. If passed it passes the AST into to compiler to get a list of opcode. It takes the list of opcodes and attaches a program header and tail to ensure accurate program detail and lastly translates all opcodes into NASM assembly text.

6.2. The C Library

In order to introduce web functionalities into our language, and to allow these web functionalities to be easily used and of the same format as modern high-level languages, such as javascript, we built an external support library for our language that would abstract the entire implementation of such complex functionality, and enable TED developers to easily scrape information from the web. Additionally, in order for this type of web scraping to work, we had to implement a linked list for our language, through which we could iterate in order to find the retrieved internet information.

6.2.1. parse.h

parse.h is the primary container for our external library. This module contains a set of functions that perform the web communication with the internet, and it is divided into two groups of functions: a set of functions that are related to the page data type, and a set of functions that pertain to the element data type. These functionalities range from fetching the information from the internet, to searching for particular pieces of information. In order to implement the searching and fetching mechanisms, we employed an external headless browser, called PhantomJS, which internally uses the Chrome Webkit engine. Because of this integration, our functions that fetch information from the internet all contained the entire capability of a modern web browser, which allowed for the use of a sublanguage of CSS Selector. This language was therefore integrated into the parse.h functions.

6.2.2. list.h

list.h implements the linked list functionality, and allows for the creation and storage of arbitrary data types.

6.2.3. cJSON.h

cJSON.h is an open source library for parsing and processing JSON serialized strings. In order to maintain communication between the PhantomJS program,, which is javascript-based, and the TED compiler, we adopted this library, which is built for C to allow the parsing of the incoming messages from the PhantomJS program.

7. Test Plan

The initial regression suite was adapted from the MicroC example. We adapted the testall script to our needs and created a basic set of tests for scanning, parsing, and basic syntax. We also developed a script that executes menhir for testing parsing. Once the print function was implemented, a new set of tests was developed to check output. Tests for more syntax and for/while loops followed. Next we implemented types so tests were created for that. Finally, when web functionality was achieved, the final round of tests was created with fully realized TED programs. A few adjustments were made as compromises were made during implementation. Overall, testing worked well and proved to be a critical part of the development process.

8. Lessons Learned

8.1. Theodore Ahlfeld

When structuring a program you must have the full structure in mind from the beginning. Bottom-Up approach to programming is not the way to go. OCaml will punish you for it by forcing you to rework the entire program and structure to implement and feature that was not foreseen. Weekly meetings are extremely important for the entire group to know that status of the compiler. It is also important that all member as comfortable in the target language.

8.2. Matthew Haigh

First I learned about compiler design, and the basic concepts of the PLT class, such as scanning, parsing, abstract syntax trees, semantic checking, and functional programming. I also learned about bash scripting since I had little to no experience and I had to develop scripts for testing. I learned some ocaml and functional programming which helps me to understand more about programming and computers in general. I gained experience in version control and teamwork that will be invaluable in future projects. I also learned about assembly language and compilation, which was a nice side effect to choosing to compile to x86.

8.3. Konstantin Itskov

Since my background is not in functional programming, OCaml was very challenging for me to use. I often found myself trying to research examples or sample material on how to best use the language, but was unable to find much information. As a result of this project, I have learned to use Ocaml and although it was difficult for me to learn, it was ultimately beneficial and I can apply my experience to learning additional functional languages in the future. One of the key lessons that I learned during the course is how to construct a compiler from scratch using yacc and lex. Going forward, although I may not always utilize Ocaml, the same concept exists in every language, so I can leverage my experience and learnings from this course in future projects.

In addition, through this course I learned the value of using multiple team-based tools, such as integrated development environments that allow for easier code writing, utilizing systems to ensure version control, and using google hangout or other types of video communication to allow for team meetings even when all members could not be physically present.

Advice: With a project like this, that lasts throughout the semester, it is imperative that the team does not push off work or procrastinate, because the amount of work is significant and it cannot be done successfully if you wait until the last minute.

8.4. Gideon Mendels

There are a few major points that i'm going to take with me for future work. First and foremost is the concept of the critical path. In a project like TED the work on some of the tasks must be sequential. In order for us to add the Web functionality we needed the complete the code generation part. Although I spent a lot of time researching different alternatives such as C XML parsers, building a parsing server and OCaml html interpretation I could not have move forward without the linking part working. Also, since our compiler is similar to GCC in its input and output I learned a lot about how code actually gets to run on a machine. The hands on work with the additional theoretical knowledge provided in class proved to be a good balance.

Advice: The only people who likes OCaml are those that suffered and worked hard to learn it through the steep learning curve. Either way you'll have to use it so it's better to put more effort in learning it so you would actually enjoy coding.

9. Appendix

Ted.ml

```
open Opcode
open Ast
module StringMap = Map.Make(String)
module StringSet = Set.Make(String)

let _ =
  (* Reading File or stdin(put) *)
  let rec file_in =
    if Array.length Sys.argv > 1 then
      (try
        open_in Sys.argv.(1)
      with e->
        Printf.printf "Error Opening File %s\n" Sys.argv.(1);
        raise e;
      )
    else
      stdin
  in
```

```
(* Creates output file or print to stdout *)
let file_out =
  if Array.length Sys.argv > 1 then
    open_out ((String.sub (Sys.argv.(1)) 0 (String.index Sys.argv.(1)
    '.'))^".asm")
  else
    stdout
in
(* Lexing *)
let lexbuf = Lexing.from_channel file_in in
(* AST Building *)
let ast = Parser.program Scanner.token lexbuf in
let prgglob (var, fdecl) =
  List.map (fun x -> match x with Var(_, n, _) -> n) var
in
let externfunc =
    {ftype   = Int;
     fname   = "print";
     formals =
       Arg(File, "stream")::Arg(String, "format")::Arg(Any, "vararg")::[];
     locals  = [];
     body    = [];
    }::{
     ftype   = File;
     fname   = "open";
     formals = Arg(String, "file")::Arg(String, "attr")::[];
     locals  = [];
     body    = []
    }::{
     ftype   = List;
     fname   = "listNew";
     formals = [];
     locals  = [];
     body    = []
    }::{
     ftype   = Any;
     fname   = "listHead";
     formals = Arg(List, "list")::[];
     locals  = [];
     body    = [];
    }::{
     ftype   = Any;
     fname   = "listAddLast";
     formals = Arg(List, "list")::Arg(Any, "data")::[];
     locals  = [];
     body    = [];
    }::{
     ftype   = Any;
     fname   = "listAddAfter";
     formals = Arg(List, "list")::Arg(Any, "data")::[];
```

```
 locals  = [];
 body    = [];
}::{
 ftype   = Any;
 fname   = "listSet";
 formals = Arg(List, "list")::Arg(Any, "data")::[];
 locals  = [];
 body    = [];
}::{
 ftype   = Any;
 fname   = "listRemove";
 formals = Arg(List, "list")::Arg(Int, "index")::[];
 locals  = [];
 body    = [];
}::{
 ftype   = List;
 fname   = "listTail";
 formals = Arg(List, "list")::[];
 locals  = [];
 body    = [];
}::{
 ftype   = List;
 fname   = "listConcat";
 formals = Arg(List, "list1")::Arg(List, "list2")::[];
 locals  = [];
 body    = []
}::{
 ftype   = Page;
 fname   = "pageFetch";
 formals = Arg(String, "url")::[];
 locals  = [];
 body    = []
}::{
 ftype   = String;
 fname   = "pageURL";
 formals = Arg(Page, "page")::[];
 locals  = [];
 body    = []
}::{
 ftype   = String;
 fname   = "pageHTML";
 formals = Arg(Page, "page")::[];
 locals  = [];
 body    = []
}::{
 ftype   = Element;
 fname   = "pageRoot";
 formals = Arg(Page, "page")::[];
 locals  = [];
 body    = []
```

```
    }::{
     ftype   = List;
     fname   = "pageFind";
     formals = Arg(Page, "page")::Arg(String, "selector")::[];
     locals  = [];
     body    = []
    }::{
      ftype = String;
      fname = "elementAttr";
      formals = Arg(Element, "element")::Arg(String, "selector")::[];
      locals = [];
      body = []
    }::{
      ftype = String;
      fname = "elementText";
      formals = Arg(Element, "element")::[];
      locals = [];
      body = []
    }::{
      ftype = String;
      fname = "elementType";
      formals = Arg(Element, "element")::[];
      locals = [];
      body = []
    }::{
      ftype = List;
      fname = "elementChildren";
      formals = Arg(Page, "page")::Arg(Element, "element")::[];
      locals = [];
      body = [];
    }::[]
in
let externfset =
    List.fold_left
      (fun set x -> x.fname::set) [] externfunc
in
let globals = prgglob ast in
let complete_ast (var, fdecl) =
  let newvar = [Var(File, "stdout", Noexpr)] in
  (newvar@var, externfunc@fdecl)
in
let localfnameset (var, fdecl) =
  List.fold_left (fun m fd -> StringSet.add fd.fname m) StringSet.empty fdecl
in
let fnameset = localfnameset ast in
let ast = complete_ast ast in
(* SAST Building *)
let sast (var, fdecl) =
  let fnamemap = List.fold_left
    (fun m func -> StringMap.add func.fname func.ftype m)
  StringMap.empty fdecl
in
let sc_fdecl fd =
  let varmap =
    List.fold_left (fun m x -> match x with Ast.Var(t, n, _) -> StringMap.add n t m)
    StringMap.empty (fd.locals@var)
  in
  let argmap =
    List.fold_left (fun m x -> match x with Ast.Arg(t, n) -> StringMap.add n t m)
    StringMap.empty (fd.formals)
  in
  let rec sc_expr = function
    | Literal s         -> Int
    | Id s              -> (try StringMap.find s varmap
                              with Not_found -> try StringMap.find s argmap
                              with Not_found -> try StringMap.find s fnamemap
                              with Not_found -> raise (Failure ("Can't find ID")))
    | Stringlit s       -> String
    | Binop(lhs,op,rhs) -> let isvalid e =
                              (match sc_expr e with
                              | Int -> Int
                              | _ -> raise (Failure ("Binop type mismatch")))
                           in
                           ignore (isvalid lhs);
                           ignore (isvalid rhs);
                           Int
    | Assign(v, e)      -> let vtype = StringMap.find v varmap in
                           let expr = sc_expr e in
                           if vtype = expr || expr = Any then vtype
                           else raise (Failure ("Type Mismatch"))
    | Call(s, e)        -> let rec matching lst1 lst2 =
                              let hd = function
                                | [] -> Nil
                                | (Td t)::_ -> t
                                | (Te t)::_ -> sc_expr t
                              in
                              let hd1 = hd lst1 and hd2 = hd lst2 in
                              if hd2 != Any then (
                              if hd1 != hd2 then raise (Failure ("Type Mismatch"))
                              else if lst1 != [] then
                                matching (List.tl lst1) (List.tl lst2)
                              else StringMap.find s fnamemap)
                              else StringMap.find s fnamemap
                           in
                           let rec flist = function
                              | [] -> []
                              | hd::tl -> if s = hd.fname then hd.formals else flist tl
                           in
                           let encaps lst =
                              List.map (fun x -> match x with
                                   | Literal _    | Id _          | Stringlit _
                                   | Binop(_,_,_) | Assign(_, _) | Call _
                                   | Noexpr -> Te x
                                   ) lst
                                in
                                matching (encaps e) (
                                  List.map (fun x -> match x with
                                    | Arg(t, _) -> Td t) (flist fdecl))
            | Noexpr            -> Nil
          in
          let rec sc_stmt = function
            | Block stmt_list   -> List.iter sc_stmt stmt_list
            | Expr e            -> ignore (sc_expr e)
            | Return e          -> if fd.ftype != (sc_expr e) then
                                raise (Failure ("Binop type mismatch"))
            | If (p, tb, fb)    -> if (sc_expr p) != Int then
                                  raise (Failure ("Type Mismatch"))
                                else
                                (sc_stmt tb);
                                (sc_stmt tb);
            | For(e1, e2, e3,s) -> ignore (sc_expr e1);
                                ignore (sc_expr e2);
                                ignore (sc_expr e3);
                                sc_stmt s;
            | While(e, s)       -> if sc_expr e != Int then raise (Failure ("Type Mismatch")) else
                                sc_stmt s
          in
          let sc_var = function
            | Var(t, _, e) ->
              if e != Noexpr && t != (sc_expr e) && (sc_expr e) != Any then
                raise (Failure ("Variable type mismatch"))
          in
          List.iter sc_var fd.locals;
          List.iter sc_stmt fd.body
        in
        List.iter sc_fdecl fdecl
      in
      sast ast;
      let program = ast in
      (* Creates an array of correlating bstmts(opcode.ml) *)
      let prg = (Compile.translate program fnameset).text in
      (* Creates the stringmap for string literals *)
      let stringlit =
        let add_string str n map =
          if StringMap.mem str map then map
          else StringMap.add str ("Str" ^ (string_of_int n)) map
        in
        let rec filter_strings n = function
          | []     -> StringMap.empty
          | hd::tl ->
            (match hd with
            | Opcode.Str s | Opcode.Arg(_, Opcode.Str s)
            | Opcode.Arg(_, (Opcode.Arg(_, Opcode.Str s)))
            | Opcode.Assign(_, Opcode.Str s) ->
              add_string s n (filter_strings (n+1) tl)
            | _   -> filter_strings (n) tl
            )
        in
        filter_strings 0 (Array.to_list prg)
      in
      (*StringMap.iter (fun k v -> Printf.printf "%s->%s\n" k v) stringlit;*)
      (* Transfroms array into a list *)
      let prg_ops = Array.to_list prg in
      let rec makeheader = function
        | [] -> ""
        | hd :: tl -> (* Looks for functions *)
          (match hd with
            | Opcode.Prologue(s, _) ->
              (match s with
                | "fprintf" -> makeheader tl
                | "fopen"   -> makeheader tl
                | s         -> Printf.sprintf "%sglobal %s\n" (makeheader tl) s )
            | _ -> makeheader tl
          )
      in
      let full_prg =
          [Opcode.Header
            ((List.fold_left (fun s n -> Printf.sprintf "%sglobal %s\n" s n)
              (makeheader prg_ops) globals),
            (List.fold_left (fun s n ->
              let fn = match n with
                | "print"      -> "fprintf"
                | "open"       -> "fopen"
                | "listRemove" -> "list_remove"
                | "listConcat" -> "listConcate"
                | n            -> n
              in
              Printf.sprintf "%sextern %s\n" s fn) "\n" externfset))]
          @ prg_ops @ [Opcode.Tail ("", globals)]
            in
      List.iter
        (fun x -> Printf.fprintf file_out "%s" x)
        (List.map (Opcode.string_of_stmt stringlit) full_prg)
```

Ast.mli

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

type expr =
    Literal of int
```

```
  | Id of string
  | Stringlit of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr

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

type type_def =
    Int
    | String
    | File
    | List
    | Page
    | Element
    | Nil
    | Any

type t_match = Td of type_def | Te of expr

(* type * ID * value *)
type var = Var of type_def * string * expr

type arg = Arg of type_def * string

type func_decl = {
    ftype : type_def;
    fname : string;
    formals : arg list;
    locals : var list;
    body : stmt list;
}

type program = var list * func_decl list
```

scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "any"    { ANY }
| "file"   { FILE }
| "int"    { INT }
| "string" { STRING }
| "List"   { LIST }
| "Page"   { PAGE }
| "Element" { ELEMENT }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '\"' [' '-'!' '#'-'~']* '\"' as lxm { STRING_LIT(String.sub lxm 1 (String.length lxm - 2)) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE INT
%token <int> LITERAL
%token <string> ID STRING_LIT
%token EOF
%token STRING LIST ELEMENT PAGE FILE ANY

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

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

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

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

type_decl:
    INT { Int }
    | FILE { File }
    | ANY { Any }
    | STRING { String }
    | LIST { List }
    | ELEMENT { Element }
    | PAGE { Page }

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

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

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

stmt:
    expr SEMI { Expr($1) }
  | 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_opt SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

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

expr:
    LITERAL          { Literal($1) }
  | ID               { Id($1) }
  | STRING_LIT       { Stringlit($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr 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) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }
```

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

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

compile.ml

```
open Ast
open Opcode

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

(* Symbol table: Information about all the names in scope *)
type env = {
  function_index : string StringMap.t; (* Index for each function *)
  global_index   : StringSet.t; (* "Address" for global variables *)
  local_index    : int StringMap.t; (* Frame pointer offset for args, locals *)
}

(* enum : int -> int -> 'a list -> (int * 'a) list
 * enum 1 2 [14,23,42] = [ (2,14), (3,23), (4,43) ]
 *)
let rec enum stride n = function
    | []     -> []
    | hd::tl -> (n, hd) :: enum stride (n+stride) tl
;;

let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs
;;

(* Translate a program in AST form into a bytecode program.  Throw an
 * exception if something is wrong, e.g., a reference to an unknown
 * variable or function *)
let translate (globals, functions) localfnameset =

  (* Allocate "addresses for each global variable" *)
  let global_indexes =
    List.fold_left
      (fun m x -> match x with Ast.Var(_, n, _) -> StringSet.add n m)
      StringSet.empty globals
  in
  (* Assign indexes to built-in functions is special *)
  let rec string_map_create = function
    | []     -> StringMap.empty
    | (fn, fp)::tl -> StringMap.add fn fp (string_map_create tl)
  in
  let function_indexes = string_map_create ([
    ("print", "fprintf");
    ("open", "fopen");
    ("get", "get_title");
    ("find", "find");
    ("head", "head");
    ("addafter", "addafter");
    ("addbefore", "addbefore");
    ("remove", "remove");
    ("getdata", "getdata")] @ List.map (fun x -> (x.fname, x.fname)) functions)
  in

  (* Translate a function in AST form into a list of bytecode statements *)
  let translate env fdecl =
    if not (StringSet.mem fdecl.fname localfnameset) then [Fakenop] else
    (* Bookkeeping: frame pointer offsets for locals and arguments *)
    let formal_strings = (List.map (fun x -> match x with Ast.Arg(_, s) -> s) fdecl.formals)
in
    let local_offsets = (* -8 because we are in 64bit system *)
      enum (-8) (-8) (List.map (fun x -> match x with Ast.Var(_, s, _) -> s) fdecl.locals)
    and formal_offsets = (* +1 because of ret pointer *)
      enum (-8) (((List.length fdecl.locals)+1)*(-8)) formal_strings in
    let env = { env with local_index = string_map_pairs
      StringMap.empty (formal_offsets @ local_offsets) } in
    let unlist = function [x] -> x | _ -> Fakenop in
    let int_to_var = function
      | 1 -> "rdi"
      | 2 -> "rsi"
      | 3 -> "rdx"
      | 4 -> "rcx"
      | 5 -> "r8"
      | 6 -> "r9"
      | x -> if x > 6 then Printf.sprintf "%d" ((x-7))
             else Printf.sprintf "[rbp-%xH]" (abs x)
    in
    let rec to_arg acc hd =
      Arg((int_to_var (acc+1)), hd)
    in
    let rec expr = function
      | Literal i -> [Lit i]
      | Id s ->
        (try [Local_var (StringMap.find s env.local_index)]
          with Not_found -> try [Glob_var (StringSet.find s env.global_index)]
          with Not_found -> raise (Failure ("undeclared variable " ^ s)))
      | Stringlit s    -> [Str s]
      | Binop (lhs, op, rhs) -> [Mov("rcx", unlist (expr rhs))] @ [Mov("rax", unlist (expr
lhs))] @ [Bin op]
      | Assign (s, e) ->
```

```
        let asn =
          (try [Str_var (int_to_var (StringMap.find s env.local_index))]
            with Not_found -> try [Get_gvar (StringSet.find s env.global_index)]
            with Not_found -> raise (Failure ("undeclared variable" ^ s)))
        in
        (match e with
        | Call(fn, a)    -> (expr e) @ [Opcode.Assign(unlist asn, Call(fn, List.length a))]
        | Binop(_, _, _) -> (expr e) @ [Opcode.Assign(unlist asn, Fakenop)]
        | Stringlit(s)   -> [Opcode.Assign(unlist asn, Str s)]
        | _              -> [Opcode.Assign(unlist asn, unlist (expr e))]
        )
      | Call (fname, actuals) ->
        (try (List.rev (List.mapi to_arg (List.concat (List.map expr actuals)))) @
          [Opcode.Call ((StringMap.find fname env.function_index), (List.length actuals))]
          with Not_found ->
            StringMap.iter (fun k v -> Printf.printf "%s->%s\n" k v) env.function_index;
            Printf.printf "TESTING:%s\n" (StringMap.find fname env.function_index);
            raise (Failure ("undeclared function " ^ fname)))
      | Noexpr -> []
    in

    let rec stmt fnameb n = function
      | Block sl        -> List.concat (List.mapi (stmt fnameb) sl)
      | Expr e          -> expr e @ []
      | Return e        ->
        (match e with
        | Binop(_, _, _) -> expr e
        | _              -> [Ret (unlist (expr e))]
        )
      | If(p, t, f) ->
        let fnameb = fnameb ^ (string_of_int n) in
        let tblock = stmt (fnameb^"t") n t and fblock = stmt (fnameb^"f") n f in
        expr p @ [Jmp_true (fnameb ^ "t")] @ fblock @ [Jmp (fnameb^"bend")] @
        [Label (fnameb ^ "t")] @ tblock @ [Label (fnameb^"bend")]
      | While(e, b) ->
        let fnameb = fnameb ^ (string_of_int n) ^ "w" in
        let blk = stmt fnameb n b and cond = expr e in
        [Label fnameb] @ cond @ [Jmp_false (fnameb ^ "end")] @ blk @
        [Jmp (fnameb)] @ [Label (fnameb^"end")]
      | For(e1, e2, e3, b) ->
        stmt (fnameb ^ (string_of_int n) ^ "f") n
        (Block([Expr(e1); While(e2, Block([b; Expr(e3)]))]))
    in
    let rec var_asn_list = function
      | []     -> []
      | Var(_, s, e)::tl ->
        (match e with
        | Noexpr -> var_asn_list tl
        | _      -> (Expr (Ast.Assign(s, e))) :: var_asn_list tl
        )
    in
    let arg_to_var =
        (try List.mapi (fun x arg -> Arg_to_var(int_to_var (StringMap.find arg
env.local_index), (int_to_var (x+1)))) formal_strings
          with Not_found -> raise (Failure ("Undefined Problem")))
    in
    [Prologue(fdecl.fname, (((StringMap.cardinal env.local_index)+(List.length
fdecl.formals))*8))] @
      (stmt fdecl.fname 0 (Block (var_asn_list fdecl.locals))) @ (arg_to_var) @
      (stmt fdecl.fname 0 (Block fdecl.body)) @ [Epilogue]
  in
  let env = {
    function_index = function_indexes;
    global_index = global_indexes;
    local_index = StringMap.empty}
  in

  let func_bodies = List.map (translate env) functions in

  { num_globals = List.length globals;
    (* Concatenate the compiled functions and replace the function
       indexes in Jsr statements with PC values *)
    text = Array.of_list (
      List.map (
        function
          (*| Jsr i when i > 0 -> Jsr func_offset.(i)*)
          | _ as s -> s) (List.concat func_bodies)
    )
  }
```

opcode.ml

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

type bstmt =
  | Lit of int                    (* Integer Literal *)
  | Str of string                 (* String Literal *)
  | Arg of string * bstmt
  | Reg of string
  | Bin of Ast.op                 (* Binary Operators *)
  (*| Unop of Ast.unop            (* Unary Operators *)*)
  | Mov of string * bstmt         (* Mov instruction *)
  | Local_var of int              (* Local Variables, Relative Frame Pointer offset *)
  | Glob_var of string            (* Global Variables, by absolute label *)
  | Get_gvar of string            (* Gets Global Variables *)
  | Set_gvar of string            (* Sets Global Variables  *)
  | Call of string * int          (* Call function by name or address *)
  | Fdecl of string               (* Function Declaration *)
  | Imprt                         (* Import/Extern function *)
  | Prologue of string * int      (* Start of every stack frame *)
  | Epilogue                    (* End of every stack frame *)
  | Assign of bstmt * bstmt     (* Set variable *)
  | Ld_var of string            (* Load variable *)
  | Ld_reg of string            (* Load register into id *)
  | Ld_lit of int               (* Load lit into register *)
  | Str_var of string           (* Stores variable *)
  | Jmp_true of string          (* Jump if equal to zero *)
  | Jmp_false of string         (* Jump if not equal to zero*)
  | Jmp of string               (* Unconditional Jump to label *)
  | Label of string             (* Label for jumps *)
  | Header of string * string   (* Creates standard header *)
  | Tail of string * string list(* Creates a standard string *)
  | Arg_to_var of string * string
  | Fakenop
  | Ret of bstmt
;;

type prog = {
    num_globals : int;  (* Number of global variables *)
    text : bstmt array; (* Code for all the functions *)
  }

let explode s =
  let rec exp i l =
    if i < 0 then l
    else
      let ch = s.[i] in
      exp (i - 1) (Char.code ch :: l)
  in
  exp (String.length s - 1) []
;;

let rec define_global acc = function
  | [] ->
    if acc = 0 then "\n\t\tdb 00H, 00H, 00H, 00H\n"
    else if (acc+1) mod 4 = 0 then "00H\n"
    else "00H, " ^ define_global (acc+1) []
  | hd :: tl ->
    if acc = 0 then
      "\n\t\tdb " ^ (Printf.sprintf "%02XH, " hd) ^ define_global (acc+1) tl
    else if acc = 7 then
      (Printf.sprintf "%02XH" hd) ^ define_global 0 tl
    else
      (Printf.sprintf "%02XH, " hd) ^ define_global (acc+1) tl
;;


let unescape s =
    Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun u -> u);;
```

```ocaml
let rec build_str kv_list =
  match kv_list with
    | []      -> ""
    | (k, v)::tl -> v ^ ":\t\t;\"" ^ k ^ "\"" ^
      (define_global 0 (explode (unescape k))) ^ (build_str tl)
;;

let rec string_of_stmt strlit_map blist =
  let to_string x = string_of_stmt strlit_map x in
  match blist with
  | Lit(x)            -> string_of_int x
  | Str(s)            -> (try StringMap.find s strlit_map with Not_found ->
                            raise (Failure ("Undeclared string " ^ s)))
  | Arg(lhs, rhs)     ->
    (match rhs with
    | Call(s, n) -> let suffix =
                      if n-6 > 0 then Printf.sprintf "\tadd rsp, %XH\n" ((n-6)*8)
                      else ""
                    in
                    "\tmov\t" ^ lhs ^ ", rax\nRHS:"^ (to_string rhs) ^ suffix
    | Str   s     -> Printf.sprintf "\tmov\t%s, %s\n" lhs (to_string rhs)
    | _           -> (try Printf.sprintf "\tmov\trax, %s\n\tmov\t[rsp+%XH], rax\n"
                      (to_string rhs) (int_of_string lhs)
                      with Failure "int_of_string" ->
                        "\tmov\t" ^ lhs ^ ", " ^ (to_string rhs) ^ "\n")
    )
  | Reg s             -> s
  | Bin(Ast.Add)      -> "\tadd\trax, rcx\n"
  | Bin(Ast.Sub)      -> "\tsub\trax, rcx\n"
  | Bin(Ast.Mult)     -> "\timul\trcx\n"
  | Bin(Ast.Div)      -> "\tcdq\n\tidiv\trcx\n"
  | Bin(Ast.Equal)    -> "\txor\trax, rcx\n\tcmp\trax, 0\n"
  | Bin(Ast.Neq)      -> "\tcmp\trax, rcx\n\tsetne\tdl\n\tcmp\tdl, 1\n"
  | Bin(Ast.Less)     -> "\tcmp\trax, rcx\n\tsetl\tdl\n\tcmp\tdl, 1\n"
  | Bin(Ast.Leq)      -> "\tcmp\trax, rcx\n" ^
                           "\tsetle dl\n" ^
                           "\tcmp\tdl, 1\n"
  | Bin(Ast.Greater)  -> "\tcmp\trax, rcx\n" ^
                           "\tsetg dl\n" ^
                           "\tcmp\tdl, 1\n"
  | Bin(Ast.Geq)      -> "\tcmp\trax, rcx\n" ^
                           "\tsetge dl\n" ^
                           "\tcmp\tdl, 1\n"
  | Mov(dst, src)     -> Printf.sprintf "\tmov\t%s, %s\n" dst (to_string src)
  | Ret(b)            ->
    (match b with
    | Lit _ | Str _
    | Glob_var _ | Local_var _ -> "\tmov\trax, " ^ (to_string b) ^ "\n"
    | Call(_, _)               -> (to_string b)
    | Bin _                    -> (to_string b)
    | _                         -> ""
    )
  | Prologue(s, n)       ->
      let offset = if n mod 16 = 0 then n+16 else n+24 in
      s ^ ":\n\tpush\trbp\n\tmov\trbp, rsp\n\tsub\trsp, " ^
      Printf.sprintf "%02XH\n" offset
  | Epilogue             -> "\tleave\n\tret\n"
  | Local_var(x)         -> Printf.sprintf "[rbp-%XH]" (abs x)
  | Arg_to_var(var, arg) -> "\tmov\t" ^ var ^ ", " ^ arg ^ "\n"
  | Glob_var(s)          -> "["^s^"]"
  | Set_gvar(s)          -> "\tmov\t" ^ s ^ ", rax\n"
  | Get_gvar(s)          -> "\tmov\trax, " ^ s ^ "\n"
  | Call(s, n)           -> let name = match s with
                               | "listRemove"     -> "list_remove"
                               | "listConcat" -> "listConcate"
                               | _                  -> s
                            in
                            "\tcall\t" ^ name ^ "\n"
  | Fdecl(s)             -> "global " ^ s ^ "\n"
  | Imprt                -> "extern fprintf\nextern fopen\n"
  | Assign(dst, src)     ->
    (match src with
    | Call(s, n) -> (to_string dst)
    | Fakenop    -> (to_string dst)
    | _          -> "\tmov\trax, " ^ (to_string src)  ^ "\n" ^ (to_string dst)
    )
  | Jmp_true(lbl)     -> "\tjz " ^ lbl ^ "\n"
  | Jmp_false(lbl)    -> "\tjnz " ^ lbl ^ "\n"
  | Jmp(lbl)          -> "\tjmp " ^ lbl ^ "\n"
  | Label(lbl)        -> lbl ^ ":\n"
  | Ld_var(var)       -> "\tmov\trdx, rax\n\tmov\t" ^ var ^ ", rax\n"
  | Ld_reg(reg)       -> "\tmov\trax, " ^ reg ^ "\n"
  | Ld_lit(lit)       -> "\tmov\trax, " ^ (string_of_int lit) ^ "\n"
  | Str_var(var)      -> "\tmov\tqword " ^ var ^", rax\n"
  | Header(s, extn)   -> s ^ extn ^
                         "extern stdout\n" ^
                         "\nSECTION .text\n"
  | Tail(s, g)        -> "\nSECTION .data\n" ^
                         (List.fold_left
                           (fun s n ->
                             Printf.sprintf "%s\n%s:\n\t\tdd 00000005H\n" s n) "" g) ^
                         "SECTION .bss\n" ^
                         "SECTION .rodata\n" ^
                         (build_str (StringMap.bindings strlit_map)) ^ "\n" ^
                         s ^ "\n"
  | Fakenop           -> ""
;;
```

parse.c

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include "cJSON.h"
#include <stdint.h>
#include <ctype.h>
#include "parse.h"

char *_scrape(char *script) {
    int pipefd[2];
    pid_t cpid;
    int status;

    FILE *file_ptr;
    char *buffer = NULL, *temp = NULL;

    if (pipe(pipefd) == -1) return NULL;

    if ((cpid = fork()) == -1) {
        return NULL;
    } else if (cpid == 0) {
        close(pipefd[0]);
        dup2(pipefd[1], STDOUT_FILENO);
        dup2(pipefd[1], STDERR_FILENO);

        execl("/usr/bin/nodejs", "nodejs", "-e", script, NULL);
        return NULL;
    }


    close(pipefd[1]);

    size_t sz = 0, idx = 0;
    int chr = EOF;

    file_ptr = fdopen(pipefd[0], "r");
    if (file_ptr != NULL) {
        while (chr) {
            chr = fgetc(file_ptr);
            if (chr == EOF) chr = 0;

            if (sz <= idx) {
                sz++;
                if ((temp = realloc(buffer, sz)) == NULL) {
                    free(buffer);
                    break;
                }
                buffer = temp;
            }
            buffer[idx++] = (char) chr;
        }
    }

    if (waitpid(cpid, &status, 0) == -1) return NULL;

    return buffer;
}
Page *pageFetch(char *url) {
    Page *page = malloc(sizeof(Page));
    char *buffer = NULL;
    asprintf(&buffer, "var phantom = require('phantom'); phantom.create(function (ph) {
ph.createPage(function (page) { page.set('viewportSize', {width: 1366, height: 768});
page.open('%s', function () { page.includeJs('http://code.jquery.com/jquery-1.11.3.min.js',
function () { page.evaluate(function () { return window.document.documentElement.outerHTML; },
function (result) { var str = JSON.stringify(result); console.log(str.substring(1, str.length
- 1)); ph.exit(); }); }); }); }); });", url);
    if (buffer == NULL) return NULL;

    page->url = strdup(url);
    char *str = _scrape(buffer);
    str[strlen(str) - 1] = 0;
    page->html = str;


    free(buffer);
    return page;
}
NODE *pageFind(Page *page, char *sel) {
    char *buffer = NULL;
    asprintf(&buffer, "var phantom = require('phantom'); phantom.create(function (ph) {
ph.createPage(function (page) { page.set('viewportSize', {width: 1366, height: 768});
page.set('content', \"%s\"); page.includeJs('http://code.jquery.com/jquery-1.11.3.min.js',
function () { page.evaluate(function () { var css_path = function(el) { if (!(el instanceof
Element)) return; var path = []; while (el.nodeType === Node.ELEMENT_NODE) { var selector =
el.nodeName.toLowerCase(); if (el.id) { selector += '#' + el.id; path.unshift(selector);
break; } else { var sib = el, nth = 1; while (sib = sib.previousElementSibling) { if
(sib.nodeName.toLowerCase() == selector) nth++; } if (nth != 1) selector +=
':nth-of-type('+nth+')'; } path.unshift(selector); el = el.parentNode; } return path.join(' >
'); }; return $.map($('%s'), function (elm) { return {html: elm.outerHTML, path:
css_path(elm)} }); /*return $.find(blue[0]).length;*/ }, function (result) {
console.log(JSON.stringify(result)); ph.exit(); }); }); }); });", page->html, sel);
    if (buffer == NULL) return NULL;

    cJSON *json;
    char *str = _scrape(buffer);
    json = cJSON_Parse(str);
    if (json == NULL)
        return NULL;

    NODE * head = listNew();

    int i;
    for (i = 0; i < cJSON_GetArraySize(json); i++) {
        Element * e = malloc(sizeof(Element));
        e->html = strdup(cJSON_GetObjectItem(cJSON_GetArrayItem(json, i),
"html")->valuestring);
        e->path = strdup(cJSON_GetObjectItem(cJSON_GetArrayItem(json, i),
"path")->valuestring);
        listAddLast(head,e);
    }

    free(buffer);
    cJSON_Delete(json);
    return head;
}


/*Element*/
char * _runOnElement(Element * element,char * code){
    int size = sizeof(char) * strlen(element->html)*2;
    char * innerHtml = malloc(size);
    str_escape(innerHtml,element->html,size);
    char *buffer = NULL;
    asprintf(&buffer, "var phantom = require('phantom'); phantom.create(function (ph) {
ph.createPage(function (page) { page.set('viewportSize', {width: 1366, height: 768});
page.set('content',\"\"); page.includeJs('http://code.jquery.com/jquery-1.11.3.min.js',
function () { page.evaluate(function () { return obj = {text:$(\"%s\").%s}; }, function
(result) { console.log(JSON.stringify(result)); ph.exit(); }); }); }); });", innerHtml,code);
    if (buffer == NULL) return NULL;

    cJSON *json;
    char *str = _scrape(buffer);
    //printf("str: %s",str);
    json = cJSON_Parse(str);
    if (json == NULL)
        return NULL;
    json = cJSON_GetObjectItem(json,"text");
    if (json == NULL)
        return NULL;
    free(innerHtml);
    return json->valuestring;
}
char * elementAttr(Element * element, char * attr){
    if (element->html == NULL)
        return NULL;
    int size = sizeof(char) * strlen(element->html)*2;
    char * innerHtml = malloc(size);
    str_escape(innerHtml,element->html,size);
    char *buffer = NULL;
    asprintf(&buffer, "var phantom = require('phantom'); phantom.create(function (ph) {
ph.createPage(function (page) { page.set('viewportSize', {width: 1366, height: 768});
page.set('content',\"\"); page.includeJs('http://code.jquery.com/jquery-1.11.3.min.js',
function () { page.evaluate(function () { return obj = {attr:$(\"%s\").attr('%s')}; },
function (result) { console.log(JSON.stringify(result)); ph.exit(); }); }); }); });",
innerHtml, attr);
    if (buffer == NULL) return NULL;

    cJSON *json;
    char *str = _scrape(buffer);
    json = cJSON_Parse(str);
    if (json == NULL)
        return NULL;
    json = cJSON_GetObjectItem(json,"attr");
    if (json == NULL)
        return NULL;
    free(innerHtml);
    return json->valuestring;
}
char * elementText(Element * element){
    return _runOnElement(element,"text()");
}
char * elementType(Element * element){
    return _runOnElement(element,"get(0).tagName");
}
NODE * elementChildren(Page * page, Element * element) {
    char *buffer = NULL;
    asprintf(&buffer, "var phantom = require('phantom'); phantom.create(function (ph) {
ph.createPage(function (page) { page.set('viewportSize', {width: 1366, height: 768});
page.set('content', '%s'); page.includeJs('http://code.jquery.com/jquery-1.11.3.min.js',
function () { page.evaluate(function () { var css_path = function(el) { if (!(el instanceof
Element)) return; var path = []; while (el.nodeType === Node.ELEMENT_NODE) { var selector =
el.nodeName.toLowerCase(); if (el.id) { selector += '#' + el.id; path.unshift(selector);
break; } else { var sib = el, nth = 1; while (sib = sib.previousElementSibling) { if
(sib.nodeName.toLowerCase() == selector) nth++; } if (nth != 1) selector +=
\":nth-of-type(\"+nth+\")\"; } path.unshift(selector); el = el.parentNode; } return
path.join(\" > \"); }; return $.map($('%s').children(), function (elm) { return {html:
elm.outerHTML, path: css_path(elm)} }); /*return $.find(blue[0]).length;*/ }, function
(result) { console.log(JSON.stringify(result)); ph.exit(); }); }); }); });",
page->html,element->path);
    if (buffer == NULL) return NULL;
    cJSON *json;
    char *str = _scrape(buffer);
    json = cJSON_Parse(str);
    if (json == NULL)
        return NULL;
    NODE * head = listNew();

    int i;
    for (i = 0; i < cJSON_GetArraySize(json); i++) {
        Element * e = malloc(sizeof(Element));
        e->html = strdup(cJSON_GetObjectItem(cJSON_GetArrayItem(json, i),
"html")->valuestring);
        e->path = strdup(cJSON_GetObjectItem(cJSON_GetArrayItem(json, i),
"path")->valuestring);
        listAddLast(head,e);
    }

    free(buffer);
    cJSON_Delete(json);
    return head;
}

/* Page */
char * pageURL(Page * p){
    return p->url;
}
char * pageHTML(Page * p){
    return p->html;
}
Element * pageRoot(Page * p){
    Element * element = malloc(sizeof(element));
    element->path = "html";
    element->html = p->html;
    return element;
}


char * elementHTML(Element * element){
    return element->html;
}

size_t str_escape(char *dst, const char *src, size_t dstLen)
{
    const char complexCharMap[] = "abtnvfr";

    size_t i;
    size_t srcLen = strlen(src);
    size_t dstIdx = 0;

    // If caller wants to determine required length (supplying NULL for dst)
    // then we set dstLen to SIZE_MAX and pretend the buffer is the largest
    // possible, but we never write to it. Caller can also provide dstLen
    // as 0 if no limit is wanted.
    if (dst == NULL || dstLen == 0) dstLen = SIZE_MAX;
```

```
for (i = 0; i < srcLen && dstIdx < dstLen; i++)
{
    size_t complexIdx = 0;

    switch (src[i])
    {
        case '\'':
        case '\"':
        case '\\':
            if (dst && dstIdx <= dstLen - 2)
            {
                dst[dstIdx++] = '\\';
                dst[dstIdx++] = src[i];
            }
            else dstIdx += 2;
            break;

        case '\r': complexIdx++;
        case '\f': complexIdx++;
        case '\v': complexIdx++;
        case '\n': complexIdx++;
        case '\t': complexIdx++;
        case '\b': complexIdx++;
        case '\a':
            if (dst && dstIdx <= dstLen - 2)
            {
                dst[dstIdx++] = '\\';
                dst[dstIdx++] = complexCharMap[complexIdx];
            }
            else dstIdx += 2;
            break;

        default:
            if (isprint(src[i]))
            {
                // simply copy the character
                if (dst)
                    dst[dstIdx++] = src[i];
                else
                    dstIdx++;
            }
            else
            {
                // produce octal escape sequence
                if (dst && dstIdx <= dstLen - 4)
                {
                    dst[dstIdx++] = '\\';
                    dst[dstIdx++] = ((src[i] & 0300) >> 6) + '0';
                    dst[dstIdx++] = ((src[i] & 0070) >> 3) + '0';
                    dst[dstIdx++] = ((src[i] & 0007) >> 0) + '0';
                    }
                    else
                    {
                        dstIdx += 4;
                    }
                }
        }
    }

    if (dst && dstIdx <= dstLen)
        dst[dstIdx] = '\0';

    return dstIdx;
}
```

parse.h

```c
#ifndef PARSE_PARSE_H
#define PARSE_PARSE_H
#include "list.h"

#define mu_assert(message, test) do { if (!(test)) return message; } while (0)
#define mu_run_test(test) do { char *message = test(); tests_run++; if (message) return
message; } while (0)
extern int tests_run;

typedef struct Page {
    char *url;
    char *html;
} Page;

typedef struct Element {
    char *html;
    char *path;
} Element;


char *_scrape(char *script);
size_t str_escape(char *dst, const char *src, size_t dstLen);


/*Page*/
Page *pageFetch(char *url);
NODE *pageFind(Page *page, char *sel);
char *pageURL(Page *p);
char *pageHTML(Page *p);
Element *pageRoot(Page *p);


/*Element*/
char *_runOnElement(Element *element, char *code);
char *elementText(Element *element);
char *elementType(Element *element);
char *elementAttr(Element *element, char *attr);
NODE * elementChildren(Page *page, Element *element);


#endif
```

list.c

```c
#include <stdlib.h>
#include <stdio.h>
#include "list.h"
```

```
NODE *listNew()
{

    NODE *node;
    if(!(node=malloc(sizeof(NODE)))) return NULL;
    node->data=NULL;
    node->next=NULL;
    return node;
}

NODE *_list_create(void *data)
{
    NODE *node;
    if(!(node=malloc(sizeof(NODE)))) return NULL;
    node->data=data;
    node->next=NULL;
    return node;
}

void * listHead(NODE * n){
    return n->data;
}

NODE * listTail(NODE * n){
    return n->next;
}


void listSet(NODE * n, void * data){
    n->data = data;
}


NODE *listAddAfter(NODE *node, void *data)
{
    NODE *newnode;
    newnode= _list_create(data);
    newnode->next = node->next;
    node->next = newnode;
    return newnode;
}

NODE * listAddLast(NODE * first, void * data){

    if(first->next == NULL && first->data == NULL){
        first->data = data;
        return first;
    }

    NODE *newnode;
    newnode= _list_create(data);

    NODE *current = first;
    while (current->next != NULL) {
        current = current->next;
    }
    current->next = newnode;
    return newnode;

}

//list remove

void list_remove(NODE *list, int index) {//todo:test!
    int i = 0;

    while (list->next && i < index-1) {
        list = list->next;
        i++;
    }

    if(!list->next)//last one,can't delete
        return;

    NODE * tmp = list->next;
    if(!list->next->next)//this is one before the last one
        list->next = NULL;
    else
        list->next = list->next->next;

    free(tmp);
}


//list concat
NODE * listConcate (NODE *head1, NODE *head2)
{
    NODE  *p;
    if (head1==NULL)                            //if the first linked
        return (head2);
    if (head2==NULL)                            //if second linked
        return (head1);

    p=head1;                             //place p on the first
    while (p->next!=NULL)                 //move p to the last node
        p=p->next;
    p->next=head2;                           //address

    return (head1);
}
```

list.h

```c
typedef struct node_s {
    void *data;
    struct node_s *next;
} NODE;

NODE *listNew();
NODE *_list_create(void *data);
void * listHead(NODE * n);
NODE * listTail(NODE * n);
void listSet(NODE * n, void * data);
NODE *listAddAfter(NODE *node, void *data);
void list_remove(NODE *list, int index);
NODE * listConcate (NODE *head1, NODE *head2);
NODE * listAddLast(NODE * first, void * data);
void listPrint(NODE *start);
```