bridge™

A programming language for novice users to easily generate HTML forms

F I N A L   P R O J E C T   R E P O R T

Vinay Ahuja
va2199@columbia.edu

COMS W4115 – Programming Languages and Translators

May 14, 2009
Table of Contents

1. Introduction ................................................................................................................. 4

1.1 Motivation ................................................................................................................... 4

2. Language Tutorial ........................................................................................................ 5


3.1 Execution ................................................................................................................... 10

3.2 Lexical Conventions ............................................................................................... 10

3.2.1 ASCII ............................................................................................................... 10

3.2.2 Whitespace ....................................................................................................... 10

3.2.3 Identifiers ......................................................................................................... 10

3.2.4 Keywords ......................................................................................................... 11

3.3 Types ......................................................................................................................... 11

3.3.1 Comments ....................................................................................................... 11

3.3.2 Whitespace ....................................................................................................... 12

3.3.3 declare ............................................................................................................. 12

3.3.4 link= ............................................................................................................... 12

3.3.5 form= .............................................................................................................. 12

3.3.6 input= ............................................................................................................. 12

3.3.7 image= ........................................................................................................... 13

3.3.8 int= ............................................................................................................... 13

3.3.9 header ............................................................................................................. 13

3.3.10 para .............................................................................................................. 13

3.3.11 bold ............................................................................................................. 13

3.3.12 line ............................................................................................................... 14

3.3.13 button ......................................................................................................... 14

3.3.14 open .............................................................................................................. 14
1. Introduction

The Bridge programming language provides an ability for novice users to develop web pages without necessarily being familiar with any HTML syntax. As implied by the name, the language provides novice users a bridge to web based applications. It does so by providing users an intuitive and easy to learn syntax which is simple enough to read and understand, but at the same time powerful enough to create complex HTML pages. The language is also intended to be used by student developers as a tool for quickly creating prototypes. While HTML is certainly the most fundamental and simple building block of any web based application, it is tedious, time consuming and error prone to write manually. Bridge addresses these issues by providing a fairly concise set of keywords and an intuitive way to use them. From the perspective of novice users, the language will allow them to be shielded from all of the underlying HTML and from the perspective of student developers, the language will allow them to focus on their application code – for example web services running on a Java backend, and use Bridge to quickly validate and demonstrate web service calls with a web page. Programs written in Bridge will be all one-file programs. In the process of interpreting, if the Bridge compiler comes across any errors, it will inform the user of the problem with user friendly error messages and when possible also provide a recommended way of solving the problem, similar to how common programming languages do. Once interpreted, the resulting HTML code needs to be saved in a .html file which can then be opened in any browser.

1.1 Motivation

The idea for developing the Bridge programming language stemmed from two main motivations. First, it provided me an opportunity to learn how to build a compiler for a language and in the process, understand the many intricacies involved with which I was largely unfamiliar at the beginning of the project. While the idea of a programming language to generate HTML code is simple, it nevertheless provided me ample challenges to work my way through the concepts learned in the class and to apply them in the form of an elegant, concise language and its compiler. The second motivation came from having felt the need of such a language as a student developer myself in the past semesters – a language which would allow me to quickly prototype a web page to demonstrate or simply test some of the backend application features. Having a concise, yet robust language that will help me accomplish this will be useful for me in my other classes in the upcoming semesters as it will save me the time of developing a test user interface for the application, time that I can then use for focusing on the backend application features.
2. Language Tutorial

Generating HTML pages using Bridge is as easy as the following –

1. Run bridge.byte executable
2. Type in a Bridge program, ctrl-Z to run
3. Save the resulting HTML code in a .html file and open in a browser

Bridge Program

While the formal usage of the language is defined in the language reference manual, this section presents a simple practical example to demonstrate how a novice user can easily get up to speed with understanding and writing Bridge programs. The following is a complete Bridge program which generates a basic Hello World HTML page –

```bridge
run() {
open(""");
title("Hello World Program in Bridge");
header1("Hello World Program in Bridge");
close("");
}
```

Generated HTML Code

```html
<html>
<body>
<title>Hello World Program in Bridge</title>
<h1>Hello World Program in Bridge</h1>
</body>
</html>
```

Generated HTML Page in a Browser

The following is a more realistic User Information From generated using Bridge –
Bridge Program

```plaintext
userinfo() {
    declare ssn;
    declare fName;
    declare mName;
    declare lName;
    declare email;

    ssn input= "SSN: ", "11", "11";
    fName input= "First Name: ", "25", "20";
    lName input= "Last Name: ", "25", "20";
    mName input= "Middle Name: ", "15", "10";
    email input= "Email Address: ", "25", "20";

    insert(ssn);
    insert(fName);
    insert(lName);
    insert(mName);
    insert(email);
}

address() {
    declare apt;
    declare street;
    declare city;
    declare state;
    declare zip;
    declare country;

    apt input= "Apartment Number: ", "5", "5";
    street input= "Street: ", "45", "40";
    city input= "City: ", "35", "30";
    state input= "State: ", "2", "2";
    zip input= "Zip: ", "5", "5";
    country input= "Country: ", "20", "20";

    insert(apt);
    insert(street);
    insert(city);
    insert(state);
}
```
insert(zip);
insert(country);
}

horizontalLine() {
insert("line");
}

image(source, width, height) {
declare imagel;
imagel image= source, width, height;
insert(imagel);
}

form() {
declare form1;
form1 form= "POST", "/action.do";
insert(form1);
}

run() {
open(" ");
title("User Information Form");
form();
header3("User Information Form");
userinfo();
address();
horizontalLine();
image("100", "100",
"C:\Users\Vinay\Pictures\cu_home_alma_mater.jpg");

insert(button "submit", "Submit");
insert(button "reset", "Reset");
close(" ");
}
<html>
<body>
<title>User Information Form</title>
<form onsubmit="return true" method="POST" action="/action.do">
<h3>User Information Form</h3>
<p>SSN:  <input type="text" maxlength="11" name="11" size="11"></p>
<p>First Name:  <input type="text" maxlength="25" name="20" size="25"></p>
<p>Last Name:  <input type="text" maxlength="25" name="20" size="25"></p>
<p>Middle Name:  <input type="text" maxlength="15" name="10" size="15"></p>
<p>Email Address:  <input type="text" maxlength="25" name="20" size="25"></p>
<p>Apartement Number:  <input type="text" maxlength="5" name="5" size="5"></p>
<p>Street:  <input type="text" maxlength="45" name="40" size="45"></p>
<p>City:  <input type="text" maxlength="35" name="30" size="35"></p>
<p>State:  <input type="text" maxlength="2" name="2" size="2"></p>
<p>Zip:  <input type="text" maxlength="5" name="5" size="5"></p>
<p>Country:  <input type="text" maxlength="20" name="20" size="20"></p>
<hr>
<img src="C:\Users\Vinay\Pictures\cu_home_alma_mater.jpg" width="100" height="100">
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>
</body>
</html>
User Information Form

SSN: 
First Name: 
Last Name: 
Middle Name: 
Email Address: 
Apartment Number: 
Street: 
City: 
State: 
Zip: 
Country: 

[Submit] [Reset]

This section formally defines the Bridge programming language, its syntax and semantics so that users and developers can understand the basic elements and start to write programs in this language.

3.1 Execution

The Bridge language can be run by executing the bridge.byte executable. The second step is to type in the Bridge program itself. In the process, the interpreter scans the input line by line in the lexical analysis stage, which results in tokens explained later in this document. The tokens are then parsed to verify if the input is a valid, syntactically correct program. This stage is followed by a semantic analysis and the generation of the HTML code as an output which can then be viewed in any browser.

3.2 Lexical Conventions

A Bridge program is comprised of a single file written in ASCII character set. Following the convention of other common languages – comments, tabs and newlines as defined below are ignored. Whitespace is required to separate tokens and is used by the parser to identify the beginning and the end of a token. At the lexical analysis stage a Bridge program can be thought of as a single line program containing the tokens described below separated by whitespace. It is also important to note that Bridge is a single-typed language. Everything in Bridge is internally handled as a String.

3.2.1 ASCII

Includes all of the 128 characters defined in the USASCII code chart. Every single element of a Bridge program is made up of some combination of these characters which the parser determines to be a part of the language.

3.2.2 Whitespace

Whitespace is used to identify one token apart from another.

3.2.3 Identifiers

An identifier is defined as a combination of alphanumeric characters $[a-z][A-Z][0-9]$ and must start with a lower case alphabet $[a-z]$. 
3.2.4 Keywords

The following identifiers are reserved keywords and therefore may not be used in any other context:

 declare
 link=
 form=
 input=
 image=
 int=
 header[1-6]
 para
 bold
 line
 button
 open
 close
 insert
 comment

3.3 Types

Bridge is a single typed language and at the most basic level everything is a string. As documented in the following keyword definitions, it is how a declaration is made which determines the usage of a given variable.

3.3.1 Comments

Comments in Bridge are identified by starting a line with /* which identifies the entire given line as a comment. The Bridge interpreter does not support nested comments. Multi-line comments must be prefixed with /* on each line.

 comment: /* commentString
 commentString: ASCII characters

Example:
 /* This is a comment in Bridge language
3.3.2 Whitespace
Whitespace helps identify one token from another.

whitespace: ‘ ’+

3.3.3 declare
The declare keyword is used to make a declaration of a variable

declare: ['_']('[a'-'z']|'[a'-'z' 'A'-'Z' '0'-'9'])*

Example:
   declare x;

3.3.4 link=
The link= keyword is used to assign a link to a variable

link=: identifier link= url, caption
identifier: as defined in 3.2.3
url: ASCII characters
caption: ASCII characters

3.3.5 form=
The form= keyword is used to assign a form to a variable

form=: identifier form= method, action
identifier: as defined in 3.2.3
method: ASCII characters
action: ASCII characters

3.3.6 input=
The input= keyword is used to assign an input text box to a variable

input=: identifier input= label, maxLength/size, name
identifier: as defined in 3.2.3
maxLength/Size: ASCII characters
name: ASCII characters
3.3.7 image=
The image= keyword is used to assign an image to a variable

image=: identifier image= source, width, height
identifier: as defined in 3.2.3
width: ASCII characters
name: ASCII characters

3.3.8 int=
The int= keyword is used to assign an image to a variable

int=: identifier int= int
identifier: as defined in 3.2.3
int: [0-9]

3.3.9 header
The header keyword is used to create and insert headings in the generated HTML page

header: [header1 | header2 | header3 | header4 | header5| header6] headerString
headerString: ASCII characters

3.3.10 para
The para keyword is used to create and insert HTML paragraphs in the generated HTML page

para: para paragraphString
paragraphString: ASCII characters

3.3.11 bold
The bold keyword is used to create and insert bold text in the generated HTML page

bold: bold boldString
boldString: ASCII characters
3.3.12 line
The line keyword is used to create and insert a horizontal line in the generated HTML page

line: insert("line")

3.3.13 button
The button keyword is used to create and insert a button in the generated HTML page

button: insert(button type, value)
type: ASCII characters
value: ASCII characters

3.3.14 open
The open keyword is used to generate the standard opening tags of a HTML page

open: open(""")

3.3.15 close
The close keyword is used to generate the standard closing tags of a HTML page

close: close(""")

3.3.16 insert
The insert keyword is used to insert a link, form, input text box, image and a horizontal line in the generated HTML page

insert: insert(id)
id: link, form, input, image, line
link: as defined in section 3.3.4
form: as defined in section 3.3.5
input: as defined in section 3.3.6
image: as defined in section 3.3.7
line: as defined in section 3.3.12

3.3.17 comment
The comment keyword is used to insert comments in the generated HTML page
3.4 Punctuations

/*
   Used for adding comments in the program text

() 
   Used for passing arguments to a function
   f(arg1, arg2, argn)
   f: function
   arg1-argn: arguments to the function

{} 
   Used for grouping a block of statements and associate it with a given function
   {
      statement1;
      statement2;
      ...
      statemenn;
   }

; 
   Used for terminating a statements
   statement;

, 
   Used for separating arguments
   arg1, arg2, ... , argn

= 
   Used as an assignment operator
   id = value
   id: identifier
   identifier: as defined in 3.2.3
   value: ASCII characters

" 
   Used to identify a string
   “string”
   string: ASCII characters
4. Project Plan

Since the project for writing an interpreter included a steep learning curve with both a new language oCaml as well new concepts like scanning, parsing, lexical analysis, etc., it was essential to have a detailed yet practical project plan that would allow me, as a one-person team, to accomplish both the goals that were set for this class – 1. To understand “what” programming languages are, and 2. “How” to build compilers / interpreters for these languages.

I started off by first focusing on getting up to speed with oCaml. With Java being the only language I have used over the past four years, it was a challenge to think “functionally” in order to be able to write programs in oCaml. As a student totally new to oCaml, I realized that writing programs in a notepad text file was inefficient and very error-prone for a beginner, as other languages like Java have well developed IDEs like Eclipse which for example help you identify and therefore correct syntax errors which a beginner is bound to make numerous times, right at the time as you are typing the program.

I therefore spent some time looking for IDEs that would assist in writing, compiling and running oCaml programs quickly and efficiently. I settled down with OcaIDE, available at http://ocaml.eclipse.ortsa.com:8480/ocaide/ which was also briefly discussed in one of the class lectures later on. I found this Eclipse plug-in for oCaml to be of great help. Features like auto-completion were very helpful as they present the available API and provide a brief description about each of the available options and then allow you to make a choice and proceed. In many cases, like the ‘if then else’, the IDE even provides one with a template to fill in the values with, much like the features available in Java. For example, using the Map for scope management, if one types in the keyword Map followed by a ‘.’ the IDE presents the available options and a brief description as shown in the snapshot below.

The above is of course possible by searching online as one is writing the program in a notepad text file, however from a beginners point of view, having features like this available all in one place while one is writing the program is a great help and over the
course, helps save a lot of time which can then be utilized for learning and implementing the concepts.

Using the OcalIDE also helped as it visually arranged all the source code in an organized manner as shown in this screenshot, with the source code file names in the Navigator pane on the left hand side, the source code file contents in the right hand side top panel and a convenient Console window to run Bridge programs –

This also fully automated the build process by simply configuring the IDE to use the source files for generating the bridge.byte executable -
4.1 Programming Style

With an IDE and the development environment all set up, my first goal was to have a small “Hello World” type of interpreter working, and even though looking back in the semester this now seems like a simple and easily attainable goal, I believe having this simple example working was my first major milestone as well as the most challenging milestone in the entire timeline of this project. It was the most challenging because even though it was an extremely basic interpreter, it still required all the different pieces like the scanner, parser, abstract syntax tree, the interpreter and the top level to fit in properly and work in harmony with each other. Everything else in the Bridge project is built upon this simple example, one piece at a time.

Working as a single-person team presented me with the privilege of managing everything by myself and therefore I chose to follow an iterative programming style. Under this mechanism, a project is developed in iterations and each iteration delivers a set of functionalities, which are then tested, documented before moving on two the next iteration. I deliberately chose this style of programming because of its following plus points –

1. At all times, the project is always functional meaning that after a particular iteration has been completed, then the project has a working, deliverable functionality that has been implemented as of that iteration.

2. The iterative style of programming helps divide the overall set of functionalities into smaller, more manageable chunks. This helps as it breaks one single deadline at the end of the semester into a number of deadlines, therefore giving the opportunity to make adjustments at the end of each iteration as compared to reaching the end of the semester and realizing there is a full set of important functionalities that cannot be completed in the remaining time.

3. Developing in iterations also helps as an efficient backup management. Since this was a single person project, I did not make use of source code control systems as all files were being edited and managed by just one person. Nevertheless, this does not reduce the need of having backups. Each successful iteration during the development this project also became a backup to revert to when necessary.

4. Each iteration has its own set of test cases which specifically test the functionality delivered in that particular iteration. This helps keep all
functionality tested throughout the iterations provided that after each iteration, the test cases developed in all the previous iterations are run to ensure no functionality is broken.

### 4.2 Project Timeline

<table>
<thead>
<tr>
<th>Description</th>
<th>Functionalities</th>
<th>Date Finalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Project Proposal</td>
<td>N/A</td>
<td>February 10, 2009</td>
</tr>
<tr>
<td>Eclipse OcalDE Dev Environment Setup</td>
<td>N/A</td>
<td>February 18, 2009</td>
</tr>
<tr>
<td>Iteration One</td>
<td>“Hello World” interpreter</td>
<td>March 8, 2009</td>
</tr>
<tr>
<td>Language Reference Manual</td>
<td>N/A</td>
<td>March 10, 2009</td>
</tr>
<tr>
<td>Iteration Two</td>
<td>Scope Management</td>
<td>April 3, 2009</td>
</tr>
<tr>
<td>Iteration Three</td>
<td>HTML links, input text boxes generation</td>
<td>April 10, 2009</td>
</tr>
<tr>
<td>Iteration Four</td>
<td>HTML forms, image setup generation</td>
<td>April 19, 2009</td>
</tr>
<tr>
<td>Iteration Five</td>
<td>General HTML tags generation</td>
<td>April 26, 2009</td>
</tr>
<tr>
<td>Iteration Six</td>
<td>Finalizing test cases developed in all iterations</td>
<td>May 9, 2009</td>
</tr>
<tr>
<td>Final Report</td>
<td>N/A</td>
<td>May 14, 2009</td>
</tr>
</tbody>
</table>
5. Architectural Design

- Bridge Program
- Scanner
- Parser
- AST
- HTML Page
- Browser
6. Test Plan

Since Bridge is a non-mathematical language and generates HTML pages intended to be viewed in a browser, the testing had to be a manual effort and required opening each generated HTML file in a browser and inspecting its results. To maintain functionality throughout all the software development iterations, a separate folder was kept for each of the functionality being tested and was run after each iteration. The following table provides a list of all the test cases and a brief description of the functionality they test –

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Functionality Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelloWorld</td>
<td>The first and the most basic Bridge program – generates an HTML page with the text “Hello World” printed in header1 font</td>
</tr>
<tr>
<td>Headers</td>
<td>Validates header1, header2, header3, header4, header5 and header6 HTML tags</td>
</tr>
<tr>
<td>Paragraphs</td>
<td>Generates an HTML page with three paragraphs</td>
</tr>
<tr>
<td>Links</td>
<td>Generates an HTML page with three HTML links</td>
</tr>
<tr>
<td>InputTextBoxes</td>
<td>Validates the generation of input text boxes</td>
</tr>
<tr>
<td>Images</td>
<td>Allows the insertion of images in an HTML page</td>
</tr>
<tr>
<td>Buttons</td>
<td>Generates an HTML page with ‘Submit’ and ‘Reset’ buttons</td>
</tr>
<tr>
<td>Forms</td>
<td>Generates an HTML POST form with a user’s first and last name as the two input fields. Also included are ‘Submit’ and ‘Reset’ buttons.</td>
</tr>
<tr>
<td>Lines</td>
<td>Validates the usage of a horizontal line generally used to divide different sections of a form</td>
</tr>
<tr>
<td>Bold</td>
<td>Validates the generation of text in bold font in the HTML page</td>
</tr>
<tr>
<td>Comments</td>
<td>Allows a user to insert comments in the generated HTML page</td>
</tr>
<tr>
<td>FunctionCall</td>
<td>Tests the functionality of being able to invoke functionA() from within the body of functionB()</td>
</tr>
<tr>
<td>FunctionArgs</td>
<td>Tests the functionality of passing in arguments to functionA(arg1, arg2) which then uses arg1 and arg2 for the relevant HTML task from within the body of functionB()</td>
</tr>
<tr>
<td>UserInfoForm</td>
<td>A realistic user information HTML form</td>
</tr>
</tbody>
</table>

The above test cases were chosen with two considerations – 1. To be able to use the Bridge language to generate the most commonly used HTML pages, 2. To be able to
demonstrate the usage of the compiler / interpreter concepts learned in the class. The following few test cases provide an example of these considerations which were taken into account throughout software development iterations –

- **Function Call Bridge Program**

```haskell
userName() {
    declare fName;
    declare lName;
    fName input= "First Name ", "20", "20";
    lName input= "Last Name ", "20", "20";
    insert(fName);
    insert(lName);
}

run() {
    open(" ");
    title("Function Call Test");
    userName();
    close(" ");
}
```

- **Generated HTML page**

```html
<html>
<body>
<title>Function Call Test</title>
<p>First Name  <input type="text" maxlength="20" name="20" size="20">
<p>Last Name  <input type="text" maxlength="20" name="20" size="20">
</body>
</html>
```

- **HTML page in a browser**

![HTML page in a browser](file:///C:/Users/Ninay/Eclipse/workspace_oCaml_Bridge/Tests/12-FunctionCall/functionCall.html)
- Function Call with Arguments Bridge Program

```c
createform(arg1, arg2) {
    insert(arg1);
    insert(arg2);
}

run() {
    declare fName;
    declare lName;

    fName input= "First Name ", "20", "20";
    lName input= "Last Name ", "20", "20";

    open(""");
    title("Function Argument Test");
    createform(fName, lName);
    close("");
}
```

- Generated HTML page

```html
<html>
<body>
<title>Function Argument Test</title>
<p>Last Name  <input type="text" maxlength="20" name="20" size="20"></p>
<p>First Name  <input type="text" maxlength="20" name="20" size="20"></p>
</body>
</html>
```
7. Lessons Learned

I think the three biggest lessons I learned in this class were –

1. To have an understanding of how programming languages operate. Before this class, Java has been the primary language that I have been using for the most part. And any other language seemed to be a completely different approach to solving a problem, largely due to the difference in the syntax. However after this class, having an understanding of the scanning process, converting the input stream into a set of tokens, the parsers and the abstract syntax trees, it became apparent that under the covers most language operate the same way and differ only in the way they present their syntax to the user. I think learning the above concept will prove very important as it changes the way one approaches a new programming language. When I am confronted with a new language now, there will obviously be a learning curve with the syntax, however besides that I will have an understanding that under the covers this new language is very likely operating the same way as one of the languages that I already know of.

2. The second most important lesson I learned in this class was to have a better understanding of compilers. By implementing a basic compiler / interpreter provided an opportunity to understand some of the many complexities that go on in building a compiler. Being exposed to some of these complexities during the term of the semester also made me learn to appreciate other well established languages and have a better understanding of the amount of effort it takes under the covers to make software development easier – for example, by providing as user friendly errors as possible and indicating the source of the problem as accurately as possible.

3. The third most important lesson I learned in this class was obviously to have an opportunity to learn a new programming language – oCaml, which is a language much different from other common languages like Java, C++ and C, per se. Learning oCaml required thinking differently. While in the beginning it seemed ‘unintuitive’, with enough practice it started to make sense and in the end it brings the confidence of being able to learn a new language as well as write an interpreter using it.
8. Appendix

- scanner.mll

```ml
{ open Parser }

rule token = parse
  [' ' '	' '' '
'] { token lexbuf }
| /**/ { comment lexbuf }
| '(' { LPAREN }
| ')' { RPAREN }
| '{' { LBRACE }
| '}' { RBRACE }
| ';' { SEMI }
| ',' { COMMA }
| '=' { ASSIGN }
| "int=" { INTASSIGN }
| "link=" { LINKASSIGN }
| "form=" { FORMASSIGN }
| "input=" { INPUTASSIGN }
| "image=" { IMAGEASSIGN }
| "button" { BUTTON }
| "return" { RETURN }
| "declare" { DECLARE }
| ['0'-'9']+ as lxm { INT_LITERAL(lxm) }
| "'[^[']'*" as lxm { STRING_LITERAL(lxm) }
| ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' '0'-'9']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  /**/ { token lexbuf }
| _ { comment lexbuf }
```
- parser.mly

```ml
%
open Ast %

%token SEMI
%token LPAREN
%token RPAREN
%token LBRACE
%token RBRACE
%token COMMA
%token QUOT
%token ASSIGN
%token INTASSIGN
%token LINKASSIGN
%token FORMASSIGN
%token INPUTASSIGN
%token IMAGEASSIGN
%token DECLARE
%token BUTTON
%token RETURN
%token <string> INT_LITERAL
%token <string> STRING_LITERAL
%token <string> ID
%token EOF

%left ASSIGN
%left INTASSIGN
%left LINKASSIGN
%left TITLEASSIGN
%left FORMASSIGN
%left INPUTASSIGN
%left IMAGEASSIGN

%start program
%type <Ast.program> program

%

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

fdecl:
  ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
  | { 
    fname = $1;
    formals = $3;
    locals = List.rev $6;
    body = List.rev $7 } }
```
- parser.mly contd.

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

formal_list:
  ID       { [$1] }
  | formal_list COMMA ID { $3 :: $1 }

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

vdecl:
  DECLARE ID SEMI { $2 }

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) }

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

eexpr:
  INT_LITERAL { Int_Literal($1) }
  | STRING_LITERAL { String_Literal($1) }
  | ID       { Id($1) }
  | ID ASSIGN expr { Assign($1, $3) }
  | ID INTASSIGN expr { IntAssign($1, $3) }
  | ID LINKASSIGN expr COMMA expr { LinkAssign($1, $3, $5) }
  | ID FORMASSIGN expr COMMA expr { FormAssign($1, $3, $5) }
  | ID INPUTASSIGN expr COMMA expr COMMA expr { InputAssign($1, $3, $5, $7) }
  | ID IMAGEASSIGN expr COMMA expr COMMA expr { ImageAssign($1, $3, $5, $7) }
  | ID LPAREN actuals_opt RPAREN { Invoke($1, $3) }
  | BUTTON expr COMMA expr { Button ($2, $4) }
  | LPAREN expr RPAREN { $2 }

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

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

- ast.mli

```ml
type expr =
  Int_Literal of string
  | String_Literal of string
  | Id of string
  | Assign of string * expr
    | IntAssign of string * expr
    | LinkAssign of string * expr * expr
    | FormAssign of string * expr * expr
  | InputAssign of string * expr * expr * expr
  | ImageAssign of string * expr * expr * expr
  | Button of expr * expr
  | Invoke of string * expr list
  | Noexpr

type stmt =
  Block of stmt list
  | Expr of expr
  | Return of expr

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

type program = string list * func_decl list
```
- interpret.ml

open Ast

module NameMap = Map.Make(struct
type t = string
  let compare x y = Pervasives.compare x y
end)

exception ReturnException of string * string NameMap.t

let run (vars, funcs) =
  let func_decls = List.fold_left
    (fun func_decls fdecl -> NameMap.add fdecl.fname fdecl func𝓈)
    NameMap.empty funcs
  in
    let rec trimQuote string =
      let l = String.length string in
      if string.[0]='"' then
        trimQuote (String.sub string 1 (l-1))
      else if string.[l-1]='"' then
        trimQuote (String.sub string 0 (l-1))
      else
        string
  in
    let rec invoke fdecl actuals globals =
      let rec eval env = function
        | Int_Literal(i) -> i, env
        | String_Literal(i) -> i, env
        | Noexpr -> "1", env
        | Id(var) ->
          let locals, globals = env in
          if NameMap.mem var locals then
            (NameMap.find var locals), env
          else if NameMap.mem var globals then
            (NameMap.find var globals), env
          else
            raise (Failure ("undeclared identifier in Id " ^ var))
        | Assign(var, e) ->
          let v, (locals, globals) = eval env
          in
            if NameMap.mem var locals then
              v, (NameMap.add var v locals, globals)
            else if NameMap.mem var globals then
              v, (locals, NameMap.add var v globals)
            else
              raise (Failure ("undeclared identifier in Assign " ^ var))
- interpreter.ml contd.

  | IntAssign(var, e) ->
  |     let v, (locals, globals) = eval env e in
  |     if NameMap.mem var locals then
  |         v, (NameMap.add var v locals, globals)
  |     else if NameMap.mem var globals then
  |         v, (locals, NameMap.add var v globals)
  |     else raise (Failure ("undeclared identifier in Assign " ^ var))

  | LinkAssign(var, e, e2) ->
  |     let v, (locals, globals) = eval env e in
  |     eval env e2 in
  |         let v2, (locals, globals) = eval env e2 in
  |         let trimV2 = trimQuote v2 in
  |         if NameMap.mem var locals then
  |             v, (NameMap.add var ("<a href=" ^ v ^ ">") locals, globals)
  |         else if NameMap.mem var globals then
  |             v, (locals, NameMap.add var ("<a href=" ^
  |                 v ^ ">" ^ trimV2 ^ "</a><br>") globals)
  |         else raise (Failure ("undeclared identifier in LinkAssign " ^ var))

  | FormAssign(var, methodName, action) ->
  |     let v, (locals, globals) = eval env methodName in
  |     eval env action in
  |         if NameMap.mem var locals then
  |             v, (NameMap.add var ("<form onsubmit="return true" method=" ^ v
  |             ^ " action=" ^ v2 ^ ">") locals, globals)
  |         else if NameMap.mem var globals then
  |             v, (locals, NameMap.add var ("<a href=" ^
  |                 v ^ "dotcom") globals)
  |         else raise (Failure ("undeclared identifier in LinkAssign " ^ var))

  | InputAssign(var, label, size, name) ->
  |     let labelStr, (locals, globals) = eval env label in
  |     let trimmedLabelStr = trimQuote labelStr in
  |     = eval env size in
  |     let sizeStr, (locals, globals) = eval env name in
  |     if NameMap.mem var locals then
  |         trimmedLabelStr, (NameMap.add var ("<p>" ^
  |         trimmedLabelStr
  |     else raise (Failure ("undeclared identifier in InputAssign " ^ var))
- interpreter.ml contd.

``` OCaml
  ^ " <input type="text">
  maxlength=" ^ sizeStr ^ " name=" ^ nameStr ^ " size=" ^ sizeStr ^ ">") locals, globals)
  else if NameMap.mem var globals then
  trimmedLabelStr, (locals, NameMap.add var
  ("<a href=" ^ trimmedLabelStr ^ "dotcom") globals)
  else raise (Failure ("undeclared identifier in LinkAssign " ^ var))
  | ImageAssign(var, src, width, height) ->
  let srcStr, (locals, globals) = eval
  env src in
  let widthStr,(locals, globals)
  = eval env width in
  let heightStr,(locals, globals)
  = eval env height in
  if NameMap.mem var locals then
  srcStr, (locals, NameMap.add var ("<img src=" ^ srcStr
  height=" ^ heightStr ^ ">") globals)
  else if NameMap.mem var globals then
  srcStr, (locals, NameMap.add var ("<img
  src=" ^ srcStr
  height=" ^ heightStr ^ ">") globals)
  else raise (Failure ("undeclared identifier in LinkAssign " ^ var))
  | Invoke ("bold", [e]) ->
  let v, env = eval env e in
  let vStr = trimQuote v in
  print_endline("<b>" ^ vStr ^ 
  "</b><br>");
  "0", env
  | Button (bType, value) ->
  eval env bType in
  let valueStr, (locals, globals) =
  eval env value in
  "<input type=" ^ bTypeStr ^ " value=" ^ valueStr ^ ">", (NameMap.add "dummyKey" ("")
  locals, globals)
  | Invoke ("close", [e]) ->
  let v, env = eval env e in
  print_endline("</body>\n</html>");
  "0", env
```
- interpreter.ml contd.

```ocaml
  | Invoke ("open", [e]) ->
  |   let v, env = eval env e in
  |   print_endline("<html>\n<body>")
  |   "0", env

  | Invoke ("header1", [e]) ->
  |   let v, env = eval env e in
  |   let vTrim = trimQuote v in
  |   print_endline("<h1>" ^ vTrim ^ "</h1>");
  |   "0", env

  | Invoke ("header2", [e]) ->
  |   let v, env = eval env e in
  |   let vTrim = trimQuote v in
  |   print_endline("<h2>" ^ vTrim ^ "</h2>");
  |   "0", env

  | Invoke ("header3", [e]) ->
  |   let v, env = eval env e in
  |   let vTrim = trimQuote v in
  |   print_endline("<h3>" ^ vTrim ^ "</h3>");
  |   "0", env

  | Invoke ("header4", [e]) ->
  |   let v, env = eval env e in
  |   let vTrim = trimQuote v in
  |   print_endline("<h4>" ^ vTrim ^ "</h4>");
  |   "0", env

  | Invoke ("header5", [e]) ->
  |   let v, env = eval env e in
  |   let vTrim = trimQuote v in
  |   print_endline("<h5>" ^ vTrim ^ "</h5>");
  |   "0", env

  | Invoke ("header6", [e]) ->
  |   let v, env = eval env e in
  |   let vTrim = trimQuote v in
  |   print_endline("<h6>" ^ vTrim ^ "</h6>");
  |   "0", env

  | Invoke ("para", [e]) ->
  |   let v, env = eval env e in
  |   let vTrim = trimQuote v in
  |   print_endline("<p>" ^ vTrim ^ "</p>");
  |   "0", env
```
- interpreter.ml contd.

```ml
| Invoke ("text", [e]) ->
  let v, env = eval env e in
  print_endline("<br>" ^ v ^ "</br>");
| Invoke ("comment", [e]) ->
  let v, env = eval env e in
  let vTrim = trimQuote v in
  print_endline("<!--" ^ vTrim ^ " -->");
| Invoke ("title", [e]) ->
  let v, env = eval env e in
  let vTrim = trimQuote v in
  print_endline("<title>" ^ vTrim ^ "</title>");
| Invoke ("insert", [e]) ->
  let v, env = eval env e in
  if (v = "\"line\") then
    print_endline("<hr>")
  else
    print_endline(v);
| Invoke (f, actuals) ->
  let fdecl =
    try NameMap.find f func_decls
    with Not_found -> raise (Failure ("undefined function " ^ f))
  in
  let actuals, env = List.fold_left (fun (actuals, env) actual ->
    let v, env = eval env actual in v ::
      actuals, env)
      ([], env) actuals
  in
  let (locals, globals) = env in
  try
    let globals = invoke fdecl actuals
    in
    Block(stmts) -> List.fold_left exec env stmts
    | Expr(e) -> let _, env = eval env e in env
    | Return(e) ->
      let v, (locals, globals) = eval env e in
      raise (ReturnException(v, globals))
  in
```
- interpreter.ml contd.

    let locals =
    try List.fold_left2
      (fun locals formal actual -> NameMap.add
       formal actual locals)
      NameMap.empty fdecl.formals actuals
    with Invalid_argument(_) ->
      raise (Failure ("wrong number of
       arguments passed to " ^ fdecl.fname))
    in

    (* Initialize local variables to 0 *)
    let locals = List.fold_left
      (fun locals local -> NameMap.add
       local "0" locals)
      locals fdecl.locals
    in

    (* Execute each statement in sequence, return
       updated global symbol table *)
    snd (List.fold_left exec (locals, globals)
       fdecl.body)

    (* Run a program: initialize global
       variables to 0, find and run "page" *)
    in let globals = List.fold_left
      (fun globals vdecl1 -> NameMap.add vdecl1 "0"
       globals) NameMap.empty vars
      in try
        invoke (NameMap.find "run" func_decls) []
      with Not_found -> raise (Failure ("did not
       find the run() function"))

- bridge.ml

    let insert = false

    let _ =
      let lexbuf = Lexing.from_channel stdin in
    let program = Parser.program Scanner.token lexbuf in
    ignore (Interpret.run program)