Wig Compiler Report

Ioannis Fytilis
Shanshan Ruan
David Thibodeau

December 8, 2013

Contents

1 Introduction

1.1 Clarifications

Some of the provided examples do not match the grammar that was provided. We chose to stick more closely to the provided grammar and use those other examples as examples of invalid code.

1.2 Restrictions

Our version of WIG does not have any restriction when compared to the WIG10 version.

1.3 Extensions

The provided grammar was restricting the use of wig variables in html code to inside regular text. We allow it to appear inside attributes in html tags. This allows for some more parametric code. For example, one could change the url of a link depending on a particular variable or modify the idenfifier of a div tag so that it would load a different css style.

1.4 Implementation Status

A feature that has not been yet implemented is a better exception handler that refers to errors by line number, instead of only showing part of the code where the error occured, as well as being able to output more than one error before stopping its execution.

2 Parsing and Abstract Syntax Trees

2.1 The Grammar

The following is the grammar for our wig compiler. It appears at it appears in the file parser.mly. Since Menhir allows the modifiers + and * to create a non-empty and possibly empty list, respectively, these tokens appear in our grammar as those representations.

```
parse:
| service EOF {s}

service:
| SERVICE LCURLY html+ schema* nevariables func* session+ RCURLY
| SERVICE LCURLY html+ schema* func* session+ RCURLY

html:
| CONST HTML identifier EQ htmlbody* SEMICOLON

htmlbody:
| identifier attribute* CLOSING
| id = SIDENT
| id = BIDENT
| m = META
| INPUT inputattr+ CLOSING
| SELECT inputattr+ CLOSING htmlbody* CLSELECT
| WHATEVER
```

```
inputattr:
| NAME EQ attr
| TYPE EQ inputtype
| iattribute

inputtype:
| STR
| ID

iattribute:
| iattr
| iattr EQ iattr

iattr:
| identifier
| stringconst
| BIDENT

attribute:
| attr
| attr EQ attr

attr:
| identifier
| stringconst
| TYPE
| NAME
| BIDENT

schema:
| SCHEMA identifier LCURLY field* RCURLY

field:
| simpletype identifier SEMICOLON

nevariables:
| variable
| nevariables; variable

variable:
| tp identifiers SEMICOLON

identifiers:
| identifier
| identifier COMMA identifiers

simpletype:
| INT
| BOOL
| STRING
| VOID

tp:
```

```
| simpletype
| TUPLE identifier

func:
| tp identifier LPAREN arguments RPAREN compoundstm

arguments:
| (* empty *)
| argument
| argument COMMA arguments

argument:
| tp identifier

session:
| SESSION identifier LPAREN RPAREN compoundstm

stm:
| SEMICOLON
| SHOW document receive SEMICOLON
| EXIT document SEMICOLON
| RETURN SEMICOLON
| RETURN exp SEMICOLON
| IF LPAREN exp RPAREN stm
| IF LPAREN exp RPAREN stm ELSE stm
| WHILE LPAREN exp RPAREN stm
| compoundstm
| exp SEMICOLON

document:
| identifier
| PLUG identifier LBRACKET plugs RBRACKET

receive:
| (* empty *)
| RECEIVE LBRACKET inputs RBRACKET

compoundstm:
| LCURLY nevariables stm* RCURLY
| LCURLY stm* RCURLY

plugs:
| plug
| plug COMMA plugs


plug:
| identifier EQ exp


inputs:
| (* empty *)
| input
| input COMMA inputs


input:
| lvalue EQ identifier

exp:
| lvalue
| lvalue EQ exp
| exp ISEQ  exp
| exp NEQ   exp
| exp LE    exp
| exp GE    exp
| exp LEQ   exp
| exp GEQ   exp
| BANG exp
| MINUS exp
| exp PLUS   exp
| exp MINUS  exp
| exp TIMES  exp
| exp DIV    exp
| exp MOD    exp
| exp AND    exp
| exp OR     exp
| exp ASSIGN exp
| exp BPLUS identifier
| exp BMINUS identifier
| exp BPLUS LPAREN identifiers RPAREN
| exp BMINUS LPAREN identifiers RPAREN
| identifier LPAREN exps RPAREN
| intconst
| TRUE
| FALSE
| stringconst
| TUPLE LCURLY fieldvalues RCURLY
| LPAREN exp RPAREN

exps:
| (* empty *)
| exp
| exp COMMA exps

lvalue:
| identifier
| identifier PERIOD identifier


fieldvalues:
| (* empty *)
| fieldvalue
| fieldvalue COMMA fieldvalues
```

```
fieldvalue:
| identifier EQ exp


stringconst:
| str = STR


intconst:
| i = NUM


identifier:
| id = ID
```

2.2 Using the `Ocamllex` Tool

Our lexer has five different functions that handle five different states. The parser will query the different functions according to the current state using a reference to a state datatype. The lexer will change this state when it comes across some particular input. The states are the following:

- BaseLevel which is used for tokens in the regular flow of the program
- CommentLevel which is used to allow arbitrary nesting of multiline comments
- HTMLLevel which is used to avoid keyword stealing when parsing html code. It also uses an integer argument to count the nesting of select tags.
- InputAttrLevel which is used when inside html tags
- AttrIDLevel which is used to force the first word of an html tag to be registered as a string or a wig variable.

2.3 Using the `Menhir` Tool

Parsing is done through the parser generator Menhir. `Menhir` accepted most of the grammar as at was previously presented. The available tokens are defined in parser.mly and generated by Menhir. Some of them can take an argument which is indicated in triangular brackets before the token. As explained in the grammar, we create lists of different production rules using the + and * modifiers.

We fixed the reduce/shift conflicts that arose mostly through the use of directives, but we had on some occasions to refactor the grammar. In particular, we had to duplicate the attribute and attr production rules in order to allow the general use of the keyword "type" in arbitrary html tags while restricting its use in a select or input tag.

2.4 Abstract Syntax Trees

The abstract syntax tree represents the grammar quite directly due to the way we define algebraic datatypes. Each kind of terms presented in the grammar is defined as an ocaml type. We define collections of a particular items (such as func, session, variable, attribute) as a simple list. This allows us to use the Ocaml higher order functions on lists such as map, fold, iterate instead of having to write such functions for each of those collections.

2.5 Desugaring

In order to reduce the total amount of node types, also known as constructors in the context of OCaml, certain forms of desugaring were used. First, we created only a single constructor for both of these rules.

```
    | "if" "(" exp ")" stm
    | "if" "(" exp ")" stm "else" stm
```

In the cases when an "else" branch is not present, the compiler automatically creates an "else" branch whose associated statement is an empty statement. By an empty statement, we simply refer to a semicolon. Quite obviously, this case of desugaring does not change the behaviour of the compiler because executing the empty statement is similar to having nothing to execute. By creating the abstract syntax tree this way, we do not need to differentiate between nodes of type StmIf and nodes of type StmIfElse because a single node type can be used for both grammar rules.

Second, we noticed that it was easier to reason about the structure of our abstract syntax tree if all statements had similar structure. In the case of "if/else" and "while" statements, the grammar allows for these statements to be followed by compound statements. While it is generally good style to do so, and is actually what most programs tend to use, it is possible to have, for instance, an "if/else" statement whose branches are statements not contained in compound statements. In other words, the grammar (like most languages) allows for branches to avoid using curly brackets.

```
| IF LPAREN exp RPAREN stm
| IF LPAREN exp RPAREN stm ELSE stm
| WHILE LPAREN exp RPAREN stm
| compoundstm
```

In our compiler, we automatically wrapped the associated statements of control structures in compound statements (curly brackets). It is not a major change, and certainly does not change the semantics of the program, but it makes it easier to visually represent (pretty print) the source code.

Third, we were able to eliminate a node type from the following grammar rule.

```
receive:
| (* empty *)
| RECEIVE LBRACKET inputs RBRACKET
```

A "receive" is part of a "show" and comes after a "document". While the grammar allows us to either have or not to have a "receive" in a "show", we were able to simplify the abstract syntax tree by realizing that the lack of a "receive" can simply be represented by a "receive" taking no inputs. This means that if a program does not use a "receive", our compiler will, in its abstract tree representation, analyze the program as if a "receive" with an empty list of inputs was used. Of course, this does not change the semantics of the program because receiving a list of no inputs is essentially the same as not having to do anything at all. This allows us not to have to use two different node types to differentiate these similar cases.

Finally, the four following rules were made more concise by only using two constructors to represent them.

```
| exp BPLUS identifier
| exp BMINUS identifier
| exp BPLUS LPAREN identifiers RPAREN
| exp BMINUS LPAREN identifiers RPAREN
```

While the two rules using multiple identifiers have a different representation because they use parentheses, they act no differently from the other two cases where a single identifier is used. Because there was no need to differentiate whether parentheses were used or not, we represented these rules in the abstract syntax tree as if parentheses were always used. That is, even if a single identifier was used it would instead become a list with a single element and be analyzed as if it had come from the rules taking multiple identifiers. The semantics of the program do not change even if we add parentheses where there were none before, because the operations using these identifiers will behave independantly from the other identifiers. That is, the operators “\+” and “\-” act in the same way whether they are passed a list of zero, one, two or more identifiers.

2.6 Weeding

Before performing advanced steps such as building a symbol table, type checking or code generation, we go through a quick weeding phase. The weeding phase is meant to detect errors in the structure of the abstract syntax tree, but because an error in the structure of the abstract syntax tree would naturally make code generation, or other advanced steps, fail eventually, it is not necessary to catch every possible invalid abstract syntax tree structure error right away in this phase. Nonetheless, it is meant to remain simple and filters out some very simple cases.

In our compiler, the weeder ensures that every function which is defined to have a non-void return type must return an expression at all times. Although expressions do not have a type associated to them at this stage, we do verify that all paths from the start of the function to the end of the function, through any sequence of branches, eventually leads to a return node. A similar procedure is used for sessions, making sure that every session eventually reaches an exit or a show statement.

2.7 Testing

We do testing of the parser and the abstract syntax tree using a unit test. We have a series of benchmark examples and we have a script that runs the parser phase on them to verify our parser parses correctly or refuses good or bad examples, respectively. We also test the parsing and abstract syntax tree through the pretty printer by testing on our benchmarks the expression

$$pretty(parse(s)) = pretty(parse(pretty(parse(s))))$$

We also have intentionally invalid programs who do not need the rules of the grammar. For these programs, we verify that the property above fails.

3 Symbol Tables

3.1 Scope Rules

We established as global scope all the code, the schemas, the functions and the sessions. This allows for mutually recursive functions and mutually recursives sessions. Then, when we get inside functions and sessions, inner scopes are introduced when going inside curly bracketed statements which are defined as `compoundstm` in the grammar.

Since the wig grammar forces all the variables to be defined at the beginning of a compound statement, then there is no problem of out of scope variables later in the curly-braced block. All the new names for this scope are added to the symbol table, then the actual computations are done.

We also collcet the wig variables appearing html code (denoted by $<$ [ ] $>$ brackets) in a special scope that is accessed when we refer to this particular page. This allows us to make sure the variables used when invoking a particular piece of html code actually exist. A similar process is done for fields inside a schema.

3.2 Symbol Data

The symbol table is defined as a tree of hashtables. A given node will have in scope each of the variables in each hashtable belonging to its parent nodes. The path from the node to the root represents the actual symbol table for this particular scope.

Then, each hashtable contains, for every name, the information about what kind of identifier it is, a type for the identifier, if needed and some possible additional information. This additional information can be a hashtable used to collect all the wig variables in an html text or the fields of a schema and will be associated with html or schemas symbol. Otherwise, it can be a boolean to indicate whether a value has been declared for this identifier. This case happens when we deal with variables. The last possible additional information that can be carried around is a reference to te integer variable counting the number of plugs of html pages done by a session, which is used only by session identifiers. This allows the code generation phase to create the different goto statements to hanlde control flow in the compiled code.

3.3 Algorithm

The compiler will traverse the abstract syntax tree to collect all the identifiers into the table. It first starts with the global scope, going through the html constants, schemas, global variables, functions and sessions to fill the root hashtable.

For the html constants, it will collect information about all the wig variables in order to verify that wig variables used when plugging an html page actually exist. It also collects all the names of input tags in order match if the expected output from the page will actually be provided by the page. Then, the algorithm will collect the schema identifiers for the global scope. In this process, it will create an additional hashtable for the fields to keep track of the fields of the schema and their types. After that, it will collect the global variables and then the function and session names.

After that, it will go into the function and session bodies to collect variable declarations and to verify that every identifier used was defined. The function and session bodies introduce a new local scope. Then, whenever we get into a compound statement, we introduce a level. Since we are now dealing with local scopes, we must now consider how they are maintained. In section 3.2, we explained that the whole symbol table is represented as a tree. This tree represents each of the scope at any given point in the program. We implement the current scope as a stack. When we enter a new compound statement, we add a new hashtable to the stack that will collect all the identifiers for that level. Hence, looking up an identifier in scope reduces to iterate over all the hashtables in the stack. However, we want to be able to reuse the symbol table during type checking. To do so, when we are to pop an hashtable from the stack, we add it to the list of children of the parent node, thus creating a tree structure for it.

In addition, we do several other tests during the construction of the symbol table such as verifying that the wig variables exist or that we are obtaining existing values from the html code.

3.4 Testing

We also have a testing unit for this phase that makes sure no exception is thrown when collecting and testing the symbols. We also have capabilities for printing the symbol tables to make sure the collected information is correct. We did some manual tests in addition to the unit test. All our benchmarks work correctly.

4 Type Checking

4.1 Types

The expressions in our wig source code can take any of our four simple types: int, bool, string, and void. In addition, it can be a tuple type that has fields with any of the four simple types. We also have special type constructs for html variables, functions variables, and session variables. Since those things are not considered as first class values, they don't appear in expressions but are still considered during typechecking.

4.2 Type Rules

Let R be the return type of a function, and Γ be the list of variables in the scope. The typing rule $R, \Gamma \vdash \mathit{stm}$ denotes that the statement stm in well typed in context Γ with return type of the enclosing function R. The typing rule $\Gamma \vdash \mathit{exp} : \mathit{tp}$ denotes that the expression exp has type tp in context Γ. The typing rule $\Gamma \vdash_p \mathit{id} = \mathit{exp}$ denotes that the expression exp matchedSince variables are added together with their defined types to the symbol table, we do not have typing rules involving variable declarations.

We note that since we are dealing with simple types and we have no notion of subtyping in our types, we simply reuse the same variable to mean that they should check against the same type instead of defining a type equality predicate. $\vec{t}$ denotes the type of a tuple. Each $t_i \in \vec{t}$ is one of int, bool or string.

The typing rules for the statements are the following.

$$\frac{\Gamma \vdash \mathit{exp} : \text{ bool} \quad R, \Gamma \vdash \mathit{stm}}{R, \Gamma \vdash \text{if } (\mathit{exp}) \ \mathit{stm}} \qquad \frac{\Gamma \vdash \mathit{exp} : \text{ bool} \quad R, \Gamma \vdash \mathit{stm}_1 \quad R, \Gamma \vdash \mathit{stm}_2}{R, \Gamma \vdash \text{if } (\mathit{exp}) \ \mathit{stm}_1 \text{ else } \mathit{stm}_2}$$

$$\frac{}{\text{void}, \Gamma \vdash \ \text{return}} \qquad \frac{\Gamma \vdash e : R}{R, \Gamma \vdash \ \text{return } e} \qquad \frac{\Gamma \vdash \mathit{exp} : \text{ bool} \quad R, \Gamma \vdash \mathit{stm}}{R, \Gamma \vdash \text{while } (\mathit{exp}) \ \mathit{stm}}$$

$$\frac{\Gamma(\mathit{id}) = \text{html}}{R, \Gamma \vdash \text{show } \mathit{id}} \qquad \frac{\Gamma(\mathit{id}) = \text{html} \quad \forall k.(\Gamma \vdash_p p_k)}{R, \Gamma \vdash \text{show plug } \mathit{id}\{\vec{p}\}} \qquad \frac{\Gamma(\mathit{id}) = \text{html}}{R, \Gamma \vdash \text{exit } \mathit{id}} \qquad \frac{\Gamma(\mathit{id}) = \text{html} \quad \forall k.(\Gamma \vdash_p p_k)}{R, \Gamma \vdash \text{exit plug } \mathit{id}\{\vec{p}\}}$$

$$\frac{\Gamma(\mathit{id}) = \text{html} \quad \forall k.(\Gamma \vdash_i \mathit{inp}_k)}{R, \Gamma \vdash \text{show } \mathit{id} \text{ receive } [\vec{\mathit{inp}}]} \qquad \frac{\Gamma(\mathit{id}) = \text{html} \quad \forall k.(\Gamma \vdash_p p_k) \quad \forall k.(\Gamma \vdash_i \mathit{inp}_k)}{R, \Gamma \vdash \text{show plug } \mathit{id}\{\vec{p}\} \text{ receive } [\vec{\mathit{inp}}]}$$

The typing rules for the expressions are the following. We omit the rules for typing symbols (such as a number, true, false, a string constant) since they don't provide useful insights. The operations in tuple types handle the set operations by using field names as a way to distinguish elements.

$$\frac{\forall i. t_i \in \{ \text{ int}, \text{ bool}, \text{ string}\} \quad \Gamma \vdash \mathit{exp}_1 : \vec{t'} \quad \Gamma \vdash \mathit{exp}_2 : \vec{t''} \quad \vec{t} = \vec{t'} \cup \vec{t''}}{\Gamma \vdash \mathit{exp}_1 << \mathit{exp}_2 : \vec{t}} \qquad \frac{\Gamma(\mathit{id}) = t}{\Gamma \vdash \mathit{id} : t}$$

$$\frac{\forall i. t_i \in \{ \text{ int}, \text{ bool}, \text{ string}\} \quad \Gamma \vdash \mathit{exp}_1 : \vec{t'} \quad \Gamma \vdash \mathit{exp}_2 : \vec{t} \quad \vec{t} \subset \vec{t'}}{\Gamma \vdash \mathit{exp}_1 \backslash + \mathit{exp}_2 : \vec{t}} \qquad \frac{\Gamma(\mathit{id}) = t \quad \Gamma \vdash \mathit{exp} : t}{\Gamma \vdash \mathit{id} = \mathit{exp} : t}$$

$$\frac{\forall i. t_i \in \{ \text{ int}, \text{ bool}, \text{ string}\} \quad \Gamma \vdash \mathit{exp}_1 : \vec{t'} \quad \Gamma \vdash \mathit{exp}_2 : \vec{t''} \quad \vec{t} = \vec{t'} \backslash \vec{t''}}{\Gamma \vdash \mathit{exp}_1 \backslash - \mathit{exp}_2 : \vec{t}} \qquad \frac{\Gamma \vdash \mathit{exp}_1 : \text{ int}}{\Gamma \vdash -\mathit{exp}_1 : \text{ int}}$$

$$\frac{\text{rel} \in \{==, !=, <=, =>, <, >\} \quad \Gamma \vdash \mathit{exp}_1 : t \quad \Gamma \vdash \mathit{exp}_2 : t}{\Gamma \vdash \mathit{exp}_1 \text{ rel } \mathit{exp}_2 : \text{ bool}} \qquad \frac{\Gamma \vdash \mathit{exp}_1 : \text{ bool}}{\Gamma \vdash ! \mathit{exp}_1 : \text{ bool}}$$

$$\frac{\text{op} \in \{+, -, *, /, \%\} \quad \Gamma \vdash \mathit{exp}_1 : \text{ int} \quad \Gamma \vdash \mathit{exp}_2 : \text{ int}}{\Gamma \vdash \mathit{exp}_1 \text{ op } \mathit{exp}_2 : \text{ int}} \qquad \frac{\Gamma(f) = (t, \vec{id}) \quad \forall i. \Gamma(\mathit{id}_i) = t_i \quad \Gamma \vdash \mathit{exp}_i : t_i}{\Gamma \vdash f(\vec{\mathit{exp}}) : t}$$

$$\frac{\Gamma \vdash \mathit{exp}_1 : \text{ string} \quad \Gamma \vdash \mathit{exp}_2 : t \quad t \in \{ \text{ int}, \text{ string}\}}{\Gamma \vdash \mathit{exp}_1 + \mathit{exp}_2 : \text{ string}} \qquad \frac{\text{op} \in \{\&\&, ||\} \quad \Gamma \vdash \mathit{exp}_1 : \text{ bool} \quad \Gamma \vdash \mathit{exp}_2 : \text{ bool}}{\Gamma \vdash \mathit{exp}_1 \text{ op } \mathit{exp}_2 : \text{ bool}}$$

The typing rules for plugs are the following.

$$\frac{\Gamma \vdash \mathit{exp} : t \quad t \in \{ \text{ bool}, \text{ int}, \text{ string}\}}{\Gamma \vdash_p \mathit{id} = \mathit{exp}}$$

The typing rules for inputs are the following.

$$\frac{\Gamma(\mathit{id}_1) = t \quad t \in \{ \text{ string}, \text{ int}\}}{\Gamma \vdash_i \mathit{id}_1 = \mathit{id}_2}$$

4.3 Algorithm

The typecheking algorithm will go through the abstract syntax tree one more time. It takes also the symbol table as argument and uses it to reconstruct the context by looking at the current scope. It simply puts in the stack the path from the current node to the root and navigates through it the same way it did in the symbol table phase. The typechecker then implements the typing rules shown in the previous section.

It will output a modified abstract syntax tree in which the expression will have a type annotation. This is used for the type annotations in the pretty printing, and for some operations duringcode generation.

4.4 Testing

Again, we have implemented a unit test scripts to try to typecheck our benchmarks. In addition, we have analyzed the typed pretty printed syntax tree manually. For all our benchmarks, our tests gave expected results.

5 Resource Computation

5.1 Resources

We compute two different resources for the code generation. First we have the number of plugs that appear in a session. It is used to be able to have the compiled code resume a session that was paused in order to interact with the user. The resume is then done via a goto statement that replaces us at the right position in the cgi program.

The second resource is the list of input tags appearing in html code. It is maintained to correctly create the functions interacting with html code and correctly parse the input from the user as cgi stream. We keep an this list ordered in the same way the stream orders them so the information is properly handled.

5.2 Algorithm

We do not have a separate phase for resource computation. Both are computed during the construction of the symbol table as this allows us to do a single pass on the abstract syntax tree and thus makes it more efficient while not being problematics since the computing of those resources does not need to refer to the completed symbol table.

For the number of plugs, we simply go through the compound statement of a session and count the number of show and exit it meets along the way, recursively going through substatement. The taglist is done in a similar way by reading each html code and accumulating the identifiers whenever it passes by an input tag. Then it reverses the list to maintain the order in which they appear in the html code.

5.3 Testing

Since we do not have a separate phase for the computing of resources, we do not have a specific unit test for it. It's testing is done through the testing of the code generation phase as the correctness of the computation is required for the output code to be correctly generated. Since the algorithms are quite simple and our tests for code generations work, we assume those functions do what we expect them to do.

6 Code Generation

6.1 Strategy

After examining generated code for wig files by different compilers provided from previous years, we decided to choose c language as our target language. Moreover, we realize the importance of providing a lightweight, easy to use compiler, so we allow our wigA compiler to generate a c file and a associated install file (a script). The user can simply compiles the generated c file into the corresponding wig service (self-contained cgi scripts) by executing the install file.

The most challenging problem we are faced with is how to achieve multiple threads during the execution of cgi script and how to support the interaction between the user and the server.Since each show statement displays a html page and results in a stop in the cgi-thread, we need to store all information of local variables and states. Also since the user could provide some information and submit them through the browser, we need to restore all local variables and states when resuming our cgi thread.

Wig is a web programming language which provides support for multiple threads and sequential sessions. In order to realize such functionality, we choose to divide a session into different stages based on the show or exit document statements it contains, and we implement a main function to link these stages together. We need to maintain two files (gf and lf) for global variables and local variables respectively. Besides, we define two global variables for every wig file: url and session id. They are implemented as strings in the generated c files. When we enter a new session for the first time, we create a new session id for the connection to store local states and global states. We read all local and global variables from these temporary files before performing any information. Before we exit a session, the state of variables may get lost, hence we preserve this information by writing them into the temporary files. These temporary files and appropriate read and write command provides a reasonable solution for suspending the cgi execution at any time and resuming it afterwards without loss of any information. More specifically, the thread needs to be interrupted every time it reaches a a show document statement, and it should be resumed once it receives the data from the user through form submission.

6.2 Code Templates

Our implementation of code generation is inspired by pwig compiler and wig10 compiler which both generate c code for wig files. In particular, our run time support library is adapted from the wig10 running library since we closely follow the design idea of wig10 compiler. We first describe how a generated c file should look like by presenting the following skeleton of filename.c

```
/*****************************************
#     wig compiler-A by:
#
#    David Thibodeau
#    Ioannis Fytilis
#    Sherry Shanshan Ruan
#
#    copyright @ group-A 2013
#
 *****************************************/

/* Include library functions */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

```
#include "wigA_run.c"

/* Booleans are represented as ints */

#define true 1
#define false 0

typedef int bool;

/* Define global variables */

char * url;
char * sessionid;
char gb_name[] = "filename.gb";

FILE *gf;
FILE *lf;

/* Wig schema section */

typedef struct {
fields
} tuple name;

/* Wig global function declaration */

void func_name();

/* Wig global variable section */

type variable_name;

/* Wig html section */

void html_GetSeed(char *url, char *sessionid, ..body tag variables... )
{
    /* the standard staring printf for each html page */

    printf("Content-type: text/html\n\n");
    printf("<html>\n") ;
    printf("<form method=\"POST\" action=\"%s?%s\">\n", url, sessionid);
    printf("<body>\n");

    body of the html page

    printf("%s",body tag variable which should be plugged in);

    /* the standard ending printf for each html page */
    printf("</body>\n");
    printf("<br><input type=\"submit\" value=\"Continue\">\n");
    printf("</form><html>\n");
}
```

```
/* Wig function section */

void func_name()
{
    return ( );
}

/* Wig session section */

void session_Seed(int stage)
{
    /* all local variables in the session are declared here*/
    type varname;

    /* we need to instantiate all variables of string type */
    char *stringname = "";

    /* Control flow for different stages */
    int step = 0;
    if (stage == 0) goto step_0;

    /* Get the stage where we branch from the file */
    lf = fopen(sessionid, "r");
    step = fgetc(lf);
    fclose(lf);
    if (step == 1) goto step_1;
    ......
the number of steps correspond to the number show/exit documents the session contains
    ......
    exit(1);

/* step_0 : first time into this session */
step_0:

    /* Create a new sessionid for the connection to store locals and globals */
    sessionid = randomString("Seed", 20);

    /* Read all global variables first */
    gf = fopen(gb_name, "r");
    if (gf != NULL)
    {
    /* how we read a global variable of int type from a shared file */
fread(&varname, sizeof(int), 1, gf);

 /* how we read a global variable of string type from a shared file */
        {
            int tmpi;
            fread(&tmpi, sizeof(int), 1, gf);
            varname = (char *)malloc(tmpi+1) ;
            fread(varname, sizeof(char), tmpi, gf);
            varname[tmpi] = '\0';
        }
        fclose(gf);
    }


/* Write local variables into temp file */
    lf = fopen(sessionid, "w");
    fputc(1, lf);
    fclose(lf);

/* Write global variables into temp file */
gf = fopen(gb_name, "w");

/* how we write a global variable of int type into a shared file */
    fwrite(&varname, sizeof(int), 1, gf);

    /* how we write a global variable of string type into a shared file */
    {
        int tmpi;
        tmpi = strlen(varname);
        fwrite(&tmpi, sizeof(int), 1, gf);
        fwrite(varname, sizeof(char), strlen(varname), gf);
    }
    fclose(gf);

    /* Call to display the html */
    html_htmlname(url, sessionid, ...body tag variables are instantiated by plugs....);
    exit(0);

/* a step correspond to a document */
step_1:

    /* Read local variables from temp file */
    lf = fopen(sessionid, "r");
    fgetc(lf);
    fclose(lf);

    /* Read global variables from temp file */
    gf = fopen(gb_name, "r");
    fread(&varname, sizeof(int), 1, gf);
    {
        int tmpi;
        fread(&tmpi, sizeof(int), 1, gf);
        varname = (char *)malloc(tmpi+1) ;
        fread(varname, sizeof(char), tmpi, gf);
        Info.holder[tmpi] = '\0';
    }
    fclose(gf);

    /* we need extra type casting depending on types of different expressions*/
    /* for example, we need string to integer casting type conversion here */
    varname1 = atoi(getField("string"));

    /* Write local variables into temp file */
    lf = fopen(sessionid, "w");
    fputc(1, lf);
    fclose(lf);

    /* Write global variables into temp file */
    gf = fopen(gb_name, "w");
    fwrite(&varname, sizeof(int), 1, gf);
    {
        int tmpi;
        tmpi = strlen(varname);
        fwrite(&tmpi, sizeof(int), 1, gf);
        fwrite(varname, sizeof(char), strlen(varname), gf);
    }
    fclose(gf);

    /* We remove session id before exiting */
    remove(sessionid);
    sessionid = "";

    /* Call to display the html */
    html_htmlname(url, sessionid);
    exit(0);
}

void session_sessionname2(int stage)
{
  .......
}

.......

/* Main function section */

int main (int argc, char *agrv[])
{
    /* Initialize random seed */
    srand48(time((time_t *)0));

    /* Parse fields */
    parseFields();
    url = catString("http://", catString(getenv("SERVER_NAME"), getenv("SCRIPT_NAME")));
    sessionid = getenv("QUERY_STRING");

    /* Decide which service to launch */
    if (sessionid == NULL) goto error_and_exit;

    if (strcmp(sessionid,"sessionname") == 0) session_Seed(0);
.........

/* Display the error page if an invalid sessionid is passed */
error_and_exit:
    printf("Content-type: text/html\n\n");
    printf("<html><body>\n");
    printf("<h3>Sorry, your session has expired or is not valid.</h3>\n");
    printf("<h3>Valid session(s) are:</h3>\n");
    printf("   --> sessionname");
    printf("<br>Example: <b><i>%s?Seed</i></b><br><br>\n", getenv("SCRIPT_NAME"));
    ......
    exit(1);
}
```

As we can see from above, the generated c file can be divided into five parts.

We first generate the "include library" information which provides run time support for the c file. After that, since booleans do not exist in programming language c, we represent them as integers. We also define global variables "url", "sessionid", "lf", "gf" to help implement multiple threads as we described in strategy section previously.

Secondly, we generated the declaration for schemas. Schemas are highly consistent with type struct in c language. Hence we implement them in a relatively straight forward way. The name of the schema corresponds to the struct name and fields are translated into declarations inside type struct.

Thirdly, we generate the declaration for wig functions and wig variables. They are simply c functions and c variables.

Then we start generate code for html parts. Each const html is implemented as a html function. The html function takes the global variables url and session id as input. In addition, it needs to take an ordered list of body tag variables since arguments as tag variables are implemented as arguments to functions in our strategy. Such information can be computed and collected when we traverse the abstract syntax tree in symbol table phase. But more precisely, they fall into the category of resource computation (see section Five: resource computation for more details). Once we realize the close relationship between body tag variables and arguments to a html function. The code generation for the rest of const html is straightforward, we have the standard starting printf template and ending printf template as illustrated above. One thing that we should pay careful attention to here is we must avoid all reserved words and symbols in c. For example, % in a wig file should change into %% in a c file.

Next we come to the code generation for wig functions. Since wig expressions adopt a similar style as c language, most of them can be translated directly into c statements. However, we need to be careful with wig identifier names as they may happen to be c reserved words. For instance, in some benchmark provided, continue (a reserved word in c) is used as a variable name in a wig file. We tackle such problem by adding a suffix after the identifier name (i.e. all continue are changed to continue_A in the generated c code). We should also be careful with some type conversion issues. For example, we need explicit casting "itoa" if we want to assign an integer variable with string expressions (which is allowed by our type system.)

The implementation of sessions follows from the strategy we introduced in the last section. Each session is a function which takes an interger stage as input. Such variable step is used to decide which step (or which html page) is going to be displayed. In the body of the session function, we define local variables inside a session first. Then we give the control flow of the step. Note that we need another resource computation here because to generate the correct control flow we must count the total number of show/exit documents inside a session, which need to be computed in advance. With the support of nice control flow and corresponding step labels for each show/exit document, we can achieve multiple threads and sequential sessions and switch between different html pages smoothly.

Finally, the main function acts as an interface among sessions. The code generation for main function is standard for every wig file except we need to provide meaning information of names of different sessions to help the user to identify sessions.

6.3 Algorithm

Our code generator takes a typed abstract syntax tree and a cactus stack of symbol tables as input. It generates corresponding c code by recursively traverse the abstract syntax tree. Every time it enters a new scope, it calls push function (i.e. retrieve the corresponding symbol table and push it on top of the stack). Similarly, every time it quits a scope, a symbol table is popped from the stack. This help establish the consistence between the current scope of the abstract syntax tree and the associated symbol table as we need to keep look up information for different identifiers in code generation phase. Basically, the code generator prints the appropriate c code to a c file. We already described in detail how the generated c code should look like in previous sections. Additionally, the code generator creates a new install file and writes corresponding gcc compilation command into it to automate the further cgi script generation.

6.4 Runtime System

Our compilers outputs C code that is to be compiled using a C compiler and then runned through CGI. We set it up on the McGill Computer Science servers but any server able to serve CGI requests would work.

6.5 Sample Code

Show the complete code generated for the service `tiny.wig`.

```
/*******************************************
#     wig compiler-A by:
#
#    David Thibodeau
#    Ioannis Fytilis
#    Sherry Shanshan Ruan
#
#    copyright @ group-A 2013
#
 *******************************************/

/* Include library functions */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "wigA_run.c"

/* Booleans are represented as ints */

#define true 1
#define false 0

typedef int bool;

/* Define global variables */

char * url;
char * sessionid;
char gb_name[] = "tiny.gb";

FILE *gf;
FILE *lf;

/* Wig global variable section */
```

```
int amount;

/* Wig html section */

void html_Welcome(char *url, char *sessionid)
{
printf("Content-type: text/html\n\n");
printf("<html>\n") ;
printf("<form method=\"POST\" action=\"%s?%s\">\n", url, sessionid);
printf("<body>\n");
printf("\n    Welcome!\n  ");
printf("</body>\n");
printf("<br><input type=\"submit\" value=\"Continue\">\n");
printf("</form><html>\n");
}

void html_Pledge(char *url, char *sessionid)
{
printf("Content-type: text/html\n\n");
printf("<html>\n") ;
printf("<form method=\"POST\" action=\"%s?%s\">\n", url, sessionid);
printf("<body>\n");
printf("\n    How much do you want to contribute?\n    ");
printf("<input name=\"contribution\" type=\"text\" size=4>");
printf("\n  ");
printf("</body>\n");
printf("<br><input type=\"submit\" value=\"Continue\">\n");
printf("</form><html>\n");
}

void html_Total(char *url, char *sessionid, char *total)
{
printf("Content-type: text/html\n\n");
printf("<html>\n") ;
printf("<form method=\"POST\" action=\"%s?%s\">\n", url, sessionid);
printf("<body>\n");
printf("\n    The total is now ");
printf("%s", total);
printf(".\n  ");
printf("</body>\n");
printf("<br><input type=\"submit\" value=\"Continue\">\n");
printf("</form><html>\n");
}

/* Wig session section */

void session_Contribute(int stage)
{
int i;


/* Control flow for different stages */
```

```
int step = 0;
if (stage == 0) goto step_0;

/* Get the stage where we branch from the file */
lf = fopen(sessionid, "r");
step = fgetc(lf);
fclose(lf);
if (step == 1) goto step_1;
if (step == 2) goto step_2;
exit(1);

/* step_0 : first time into this session */
step_0:

/* Create a new sessionid for the connection to store locals and globals */
sessionid = randomString("Contribute", 20);

/* Read all global variables first */
gf = fopen(gb_name, "r");
if (gf != NULL)
{
fread(&amount, sizeof(int), 1, gf);
fclose(gf);
}

i = 87;

/* Write local variables into temp file */
lf = fopen(sessionid, "w");
fputc(1, lf);
fwrite(&i, sizeof(int), 1, lf);
fclose(lf);

/* Write global variables into temp file */
gf = fopen(gb_name, "w");
fwrite(&amount, sizeof(int), 1, gf);
fclose(gf);

/* Call to display the html */
html_Welcome(url, sessionid);
exit(0);

step_1:

/* Read local variables from temp file */
lf = fopen(sessionid, "r");
fgetc(lf);
fread(&i, sizeof(int), 1, lf);
fclose(lf);

/* Read global variables from temp file */
gf = fopen(gb_name, "r");
fread(&amount, sizeof(int), 1, gf);
```

```
fclose(gf);


/* Write local variables into temp file */
lf = fopen(sessionid, "w");
fputc(2, lf);
fwrite(&i, sizeof(int), 1, lf);
fclose(lf);

/* Write global variables into temp file */
gf = fopen(gb_name, "w");
fwrite(&amount, sizeof(int), 1, gf);
fclose(gf);

/* Call to display the html */
html_Pledge(url, sessionid);
exit(0);

step_2:

/* Read local variables from temp file */
lf = fopen(sessionid, "r");
fgetc(lf);
fread(&i, sizeof(int), 1, lf);
fclose(lf);

/* Read global variables from temp file */
gf = fopen(gb_name, "r");
fread(&amount, sizeof(int), 1, gf);
fclose(gf);

i = atoi(getField("contribution"));
amount = (amount+i);

/* Write local variables into temp file */
lf = fopen(sessionid, "w");
fputc(2, lf);
fwrite(&i, sizeof(int), 1, lf);
fclose(lf);

/* Write global variables into temp file */
gf = fopen(gb_name, "w");
fwrite(&amount, sizeof(int), 1, gf);
fclose(gf);

/* We remove session id before exiting */
remove(sessionid);
sessionid = "";

/* Call to display the html */
html_Total(url, sessionid, itoa(amount));
exit(0);
```

```
}

/* Main function section */

int main (int argc, char *agrv[])
{
/* Initialize random seed */
srand48(time((time_t *)0));

/* Parse fields */
parseFields();
url = catString("http://", catString(getenv("SERVER_NAME"), getenv("SCRIPT_NAME")));
sessionid = getenv("QUERY_STRING");

/* Decide which service to launch */
if (sessionid == NULL) goto error_and_exit;

if (strcmp(sessionid,"Contribute") == 0) session_Contribute(0);
if (strncmp(sessionid,"Contribute$",11) == 0) session_Contribute(1);

/* Display the error page if an invalid sessionid is passed */
error_and_exit:
printf("Content-type: text/html\n\n");
printf("<html><body>\n");
printf("<h3>Sorry, your session has expired or is not valid.</h3>\n");
printf("<h3>Valid session(s) are:</h3>\n");
printf("   --> Contribute");
printf("<br>Example: <b><i>%s?Contribute</i></b><br><br>\n", getenv("SCRIPT_NAME"));
printf("</body></html>\n");
exit(1);
}
```

6.6 Testing

We wrote scripts to automate the unit test of all wig examples. The script first calls our wigA compiler to compile wig files into c files and install files. It then invoke the gcc compiler to compile generated c files into cgi scripts. During these two compilation process, if any exception or warning is thrown, it will be counted as failure and corresponding error message will be printed out. So far we pass 46 out of 48 valid examples.

In addition, we manually examined and verified the generated cgi scripts through browsers.

7 Availability and Group Dynamics

7.1 Manual

The Makefile interface is as follows:

- make code: builds the compiler
- make check: run all tests
- make syntax: tests the parser and lexer
- make weed: tests the weeder

- make pretty: tests the pretty printer
- make symbol: tests the symbol table builder
- make type: tests the type checker
- make prettytype: tests the pretty printer with type annotations
- make codegen: tests the code generator
- make wig10: run the wig10 compiler on some example files
- make pwig: run the pwig compiler on some example files
- make install: compiles generated C files into CGI scripts
- make html: builds an html index page for previously installed CGI scripts
- make clean: removes generated files

The main interface is as follows.:

- -p: Displays a pretty code representation
- -w: Enables weeding
- -s: Displays the symbol table
- -t: Enables typechecking
- -c: Generate a compiled C file"
- -f: Selects a file to compile

Example: "./wigA -f ../examples/valid/tiny.wig -w -p" will weed the given program and print a pretty reprsentation back to the user

When compiling WiG programs, you need to have an environment variable called WIGDIR pointing to the wig folder within the project (group-a/wig). Also, compiled CGI scripts will go directly to your public_html/cgi-bin folder.

In our case, the CGI scripts may only be compiled on the freebsd server which also happens not to have ocaml/menhir installed. As such, in order to be able to run the CGI scripts, we (and perhaps you) have to compile the project on another server such as mimi, generate all the C files that and required and, later, only compile them while on the freebsd server.

7.2 Demo Site

One of our team members hosted on his SOCS account an index page which links to every CGI script that is successfully generated by our compiler. The link is the following: iFytil's cgi-bin. Note that each compiled CGI script needs to start at a certain session. Like the wig10 compiler, our CGI scripts will tell you which sessions are available for the selected WiG program.

7.3 Division of Group Duties

Naturally, because the project was divided into multiple milestones we each worked on every part of the compiler. Nonetheless, there were periods where one member worked more on a specific milestone. Although all work is relatively shared, here is a rough breakdown:

- David: lexing, parsing, symbol table

- Ioannis: main interface, weeding, testing
- Sherry: type checking, code generation

Our group worked well, although more advanced knowledge of the Git version-control system could've allowed us to work more efficiently and avoid conflicts. For instance, in only one occasion did we separate our workflow into two branches, for a short period of time, and I don't believe any of us used rebasing over the entire semester. Git is one of the more complex VCSs though, so it takes a while to get a group used to an advanced workflow.

8 Conclusions and Future Work

8.1 Conclusions

Over the course of this project, some of us learned how compiler design is actually less intimidating than we thought it was. Of course, good compilers are difficult to build and we have not delved too deep in the field of optimization in this class but after this project we probably each have enough experience to develop our own small domain-specific language for a specialized area. After all, even Coffeescript, a language which compiles into JavaScript, once began as a tiny compiler made as a student project.

8.2 Future Work

Some basic data structures could make the WiG language much richer. After reading through some of the example programs provided, it seems many of them are essentially begging for the ability to use arrays or lists. It could be interesting to see how easy/difficult it would be to make a change to the grammar and whether it would be easy/difficult to extend the compiler to correctly use these new features.

8.3 Course Improvements

A minor point is that some of the course slides seem to need to be updated. Furthermore, after completing the first assignment (tiny expression evaluator) in both OCaml and C we quickly realized how much easier it is to deal with trees in a functional language. Of course, we're not sure how many teams completed this project using Java/C and whether or not they had any trouble or ease doing so but it might be worth it to teach some sections of this course using a different language. COMP302, which is a pre-requisite for this course, is now taught in OCaml.

8.4 Goodbye

David is working in type theory and language design and will continue working mostly with typecheckers, in particular for strongly typed (mostly dependently typed) functional languages but will most likely work with the earlier phases of compilation development in the process of developing interpreters for the languages. The generation of code is less an object of his study since he focusses on the result of the typechecking process.