McGill University
School of Computer Science
COMP 520

Wig Compiler Report

Report No. 2013-12

Furkan Tufekci

December 6, 2013

www.cs.mcgill.ca

Table of Contents

1 Introduction

1.1 Clarifications

Overall, the WIG language definition was clearly defined. The only confusing part was the operations on tuples.

For example if we had:

schema Type1{
int a;
}
	schema Type2{
	bool b;
	}

Schema Type3{
string s;
}
	tuple Type1 t1;
	tuple Type2 t2;
	tuple Type3 t3;
	t1 = t2 << t3;
	
	t1.b = true;
	Can we access it ? The schema doesn't specify an element named “b”.

Table 1: Combining tuples of different schema types

What would happen for example when we wanted to combine 2 tuples of different types into a tuple of a different type (i.e.: t1 = t2 << t3) ? In the generated code, I decided to keep all the fields even if the schema didn't specify it. Even though I keep the fields, the script won't be able to access it, because during the typechecking phase, it will generate an error saying that the corresponding schema doesn't contain that field. If user wanted to modify the generated code, it could potentially access those fields not present in the schema.

1.2 Restrictions

I didn't make any restrictions in my version of WIG. The generated code supports all the constructs that are available in WIG (tuples, functions, global variables, nested show/exit statements, etc.).

1.3 Extensions

I didn't made any extensions in my version of WIG. I think implementing a “WIG standard library” would have been interesting.

1.4 Implementation Status

Almost all the proposed features are implemented. They have been tested and they work. The compiler takes as input the wig file and outputs a python cgi script.

I didn't implement typechecking for html inputs. I don't think it's worth having because it would be very fragile. For example, what would happen if in the form there was no input named “size” but that it was added using Javascript in the browser ? Compilation would fail indicating that the input “size” isn't present in the html, but in theory the code should work.

The only weeder I implemented is responsible for checking that all the functions and sessions return. I didn't put a lot of time into developing other weeders, because a lot of the errors in the script are detected in the parsing and typechecking phases.

2 Parsing and Abstract Syntax Trees

2.1 The Grammar

```
service: tSERVICE '{' htmls schemas nevariables functions sessions '}'
     | tSERVICE '{' htmls schemas functions sessions '}'

htmls: html
      | htmls html


html: tCONST ttHTML tID '=' tHtmlOpen nehtmlbodies tHtmlClose ';'
     | tCONST ttHTML tID '=' tHtmlOpen tHtmlClose ';'

nehtmlbodies: htmlbody
            | nehtmlbodies htmlbody

htmlbody: '<' tID attributes '>'
        | tTagClose tID '>'
        | tGapOpen tID tGapClose
        | tWHATEVER
        | tMetaOpen tWHATEVER tMetaClose
        | '<' tInput inputattrs '>'
        | '<' tSelect inputattrs '>' nehtmlbodies tTagClose tSelect '>'
        | '<' tSelect inputattrs '>' tTagClose tSelect '>'

inputattrs: inputattr
        | inputattrs inputattr

inputattr: tName '=' attr
         | tType '=' tInputType
         | attribute

attributes: /* emtpy */
          | neattributes

neattributes: attribute
            | neattributes attribute

attribute: attr
         | attr '=' attr

attr: tID
    | tSTR

schemas: /* empty */
       | neschemas

neschemas: schema
        | neschemas schema

schema: tSchema tID '{' fields '}'
```

```
fields: /* empty */
      | nefields

nefields: field
        | nefields field

field: simpletype tID ';'

simpletype: tInt
           | tBool
           | tString
          | tVoid

type: simpletype
    | tTuple tID

nevariables: variable
    | nevariables variable

variable: type identifiers ';'

identifiers: tID
    | identifiers ',' tID

functions: /* empty */
    | nefunctions

nefunctions: function
    | nefunctions function

function: type tID '(' arguments ')' compoundstm

arguments: /* empty */
    | nearguments

nearguments: argument
    | nearguments ',' argument

argument: type tID

compoundstm: '{' nevariables stms '}'
            | '{' stms '}'


sessions: session
    | sessions session

session: tSESSION tID '(' ')' compoundstm

 stms: /* empty */
    | nestms
```

```
nestms: stm
    | nestms stm

stm: ';'
    | tSHOW document receive ';'
    | tEXIT document ';'
    | tRETURN ';'
    | tRETURN exp ';'
    | tIF '(' exp ')' stm
    | tIF '(' exp ')' stm tELSE stm
    | tWHILE '(' exp ')' stm
    | compoundstm
    | exp ';'

document: tID
    | tPLUG tID '[' plugs ']'

plugs: plug
    | plugs ',' plug

plug: tID '=' exp

receive: /* empty */
    | tRECEIVE '[' inputs ']'

exp: lvalue
    | lvalue '=' exp
    | exp tEQ exp
    | exp tNEQ exp
    | exp '<' exp
    | exp '>' exp
    | exp tLEQ exp
    | exp tHEQ exp
    | '-' exp %prec UMINUS
    | '!' exp
    | exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | exp '%' exp
    | exp tLAND exp
    | exp tLOR exp
    | exp tTCOMBINE exp
    | exp tTKEEP tID
    | exp tTKEEP '(' identifiers ')'
    | exp tTDISCARD tID
    | exp tTDISCARD '(' identifiers ')'
    | tID '(' exps ')'
    | tINT
    | tTRUE
    | tFALSE
```

```
    | tSTR
    | tTuple '{' fieldvalues '}'
    | '(' exp ')'

fieldvalues: /* empty */
    | nefieldvalues

nefieldvalues: fieldvalue
    | nefieldvalues ',' fieldvalue

fieldvalue: tID '=' exp

exps: /* empty */
    | neexps

neexps: exp
    | neexps ',' exp

inputs: /* empty */
    | neinputs

neinputs: input
    | neinputs ',' input

input: lvalue '=' tID

lvalue: tID
    | tID '.' tID
```

Table 2: Bison grammar

I used the grammar provided at: http://www.cs.mcgill.ca/~cs520/2012/wiggrammar_bison.txt

2.2 Using the flex or SableCC Tool

Instead of explicitly keeping track of the line number, I used the `%option yylineno` which automatically update yylineno variable to be the line number. In theory, that option is supposed to slow down the scanner and be expensive in performance costs, but in practice, it's not noticeable.

I also used start conditions in order to avoid keyword stealing. The states are `SERVICE, HTML, HTMLCODE, METACODE, GAPCODE, COMMENT`. Other than SERVICE state, they are all exclusive states because same letters can have different meanings in them. For example, if we had:

`session A(){`
`if(true){...}`
`}`
	`const html a = <html>if(true)`
	`{...}</html>;`

Table 3: Scanner start condition example

In the first case, "`if(true)`" is supposed to return the tokens `tIF` and `tTRUE` whereas in the second case, it should just return `tSTR` with the content of the string being "`if(true)`". So the rules inside these states make sure that the appropriate token is returned.

```
%token tSERVICE tCONST tSESSION
%token ttHTML
%token <str> tID tWHATEVER tSTR tInputType
%token <integer> tINT

/* HTML RELATED TOKENS */
%token tHtmlOpen /* "<html>" */
%token tHtmlClose /* "</html>" */
%token tTagClose /* "</" */
%token tGapOpen /* "<[" */
%token tGapClose /* "]>" */
%token tMetaOpen /* "<!--" */
%token tMetaClose /* "-->" */
%token tInput /* "input" */
%token tSelect /* "select" */
%token tName /* "name" */
%token tType /* "type" */
/* END OF HTML RELATED TOKENS */

%token tSchema tInt tBool tString tVoid tTuple
%token tSHOW tPLUG tRECEIVE tEXIT tRETURN tIF tELSE tWHILE
%token tEQ tNEQ tLEQ tHEQ tLAND tLOR tTCOMBINE tTKEEP tTDISCARD
%token tTRUE tFALSE
```

Table 4: Types of tokens

2.3 Using the bison or SableCC Tool

My compiler is written in C++ and although bison works with C++, the support is not great. For example, I wanted to use `shared_ptr` in the `%union` of bison in order to make sure that there was no memory leak, but the `%union` only supports POD (Plain Old Data) variables, so I had to use raw pointers with naked `new` and `delete`. Initially, I didn't manage it very well and I had around 40mb of memory leaks for small inputs (around 10 lines), but I managed to bring it down to around 1-2mb of leaks for fairly large inputs (around 50-60 lines).

The other interesting point is that I construct the AST in a typesafe manner. All my AST classes derive from `ast::Base` class. So I could have just used `ast::Base *base;` in the `%union` and chose all the bison's action to return that type. But instead, I created a `%union` variable for the different AST classes and bison's action return the appropriate type, so the AST is created in a typesafe way.

I also used the `%left` directive in order to fix shift-reduce errors of the operations that are in the `exp` grammar. For example, I declared the last `%left` directive to be for the unary minus, so it gets the highest priority and there are no parsing conflicts with the normal minus operation.

I used the grammar from the class's website with most of the conflicts solved, so it wasn't very difficult to get bison to accept the grammar. Whenever there was a parsing conflict, I used the bison utilities for debugging. I generated reports and graphviz diagrams to try to find out where the error was coming from. I also had to make sure that the result of an action was of the appropriate type.

2.4 Abstract Syntax Trees

I represented AST nodes as classes instead of using `struct` because it's easier to divide the code into different parts. Everthing related to AST is in the namespace `ast`. There are 3 main AST classes that act as base classes for derivation: `Base`, `Exp`, `Stm`. `Base` contains 2 fields that are useful to all the derived classes: the line number where the AST node was generated and associated symbol table (more information later). `Exp` contains the type of the expression, so all the derived expressions can have a type, these types are set at the time of typechecking (more information later). `Stm` doesn't contain any implementation details, I just made it in case I needed to add common implementation to all statements later.

The following table shows which nodes derive from which base class:

Derived from `Base`	Derived from `Stm`	Derived from `Exp`
`Service`	`EmptyStm`	`LValExp`
`Whatever`	`CompoundStm`	`BinopExp`
`Variable`	`ShowStm`	`UnopExp`
`Function`	`DocumentStm`	`TupleopExp`
`Field`	`PlugStm`	`FunctionExp`
`Empty`	`InputStm`	`IntegerExp`
`HtmlTag`	`ReceiveStm`	`TrueExp`
`Argument`	`ExitStm`	`FalseExp`
`MetaTag`	`ReturnStm`	`StringExp`
`Schema`	`IfStm`	`FieldValExp`
`String`	`WhileStm`	`TupleExp`
`List`	`ExpStm`	
`Type`		
`Session`		

Table 5: AST class derivation

Most of the AST classes are only responsible for holding references to their children AST nodes, they don't contain any logic. They are very similar to “case classes” in Scala or the “data types” in Haskell, SML, etc. I initially started by implementing compiling phases as being functions in the AST classes, for example: for typechecking, the `Base` class would have a virtual function called `get_type()` and all derived classes would implement that function. To typecheck, an AST class would call `get_type()` on it's children and use typechecking rules to confirm correctness. Once done, it could return it's own type. The problem with that approach is that for every phase of the compiler I would have to implement a new function in each of the AST nodes. This started becoming time consuming.

So I implemented the `Visitor` pattern. Now all the AST nodes have a function called `accept(Visitor *v)` and it will basically just call that visitor giving it access to itself. This way, I can have a lot of different visitors going through the AST. All the visitors are in the `visitors` namespace. I have the following visitors: `CodeGenerator`, `PrettyPrinter`, `SymTabler`, `TypeChecker`, `Weeder` and `ReturnCheck` (more information on each of these visitors later). More information on the visitor pattern can be found at: http://en.wikipedia.org/wiki/Visitor_pattern.

2.5 Desugaring

I have 1 case of syntactic sugaring when multiple variables of the same types are declared like this: "`int a,b,c;`". They are going to be transformed into: "`int a; int b; int c;`". In the bison grammar, I used lists for variable identifiers, so it becomes easier to handle this situation.

2.6 Weeding

I only weed the inputs where the functions and sessions don't have a `return` statements as the last statement (`show` and `exit` statements are also considered as being return statements). I also weed the inputs where a `void` function tries to return something. In a situation where the last statement of the function or the session is an `if` or a `while`, I check that the last statement inside all the possible branches contain a return statement. I do it by implementing a visitor named Weeder which itself uses ReturnCheck visitor. The weeder works by throwing a `NO_RETURN` error. It then stops compiling and displays the line number and the name of the function or the session where the error was produced to the user. This weeder is needed because there could be situations where the program could crash or go into an infinite loop at runtime while trying to use a function's inexistent return value, etc.

2.7 Testing

The testing of this phase is done using a python script. It's inside the `tests/parsing/` folder. When `run.py` inside that folder is run, it will go into the `input/` folder, parse all the wig files and output the result of pretty-printing into `pp1/` folder. Then it will repeat the process from `pp1/` folder to `pp2/` folder. Once it's done, it will check that all the files inside `pp1/` and `pp2/` are exactly the same. So it's basically checking that pretty(parse(pretty(parse(code)))) = pretty(parse(code)).

Whenever there's an error during the parsing, an error indicating the line number and the token that caused the error will be outputted to the file or the console depending on the arguments given to the compiler.

3 Symbol Tables

3.1 Scope Rules

The scoping rules are very similar to C-like languages where the brackets (“{“, “}”) indicate a new scope. The only exceptions are the htmls and function arguments. Every symbol inside the html is represented in a single scope related to that html variable. For the function arguments, even though they are not inside the brackets of the function declaration, their scope is generated as if they were in the brackets.

3.2 Symbol Data

The symbol table entries are represented by the `Symbol` class inside the `st` namespace.
The symbol data contains the symbol's name, most of the time the name corresponds to the id of the AST node. The only exception is the html tag symbols where the name is gap's name or the html tag's name attribute.
The symbol also contains the symbol identifier which can be any of the following:

```
HTML
SELECT_TAG
INPUT_TAG
HOLE_TAG
SCHEMA
FIELD
VARIABLE
ARGUMENT
FUNCTION
SESSION
```

Table 6: Symbol identifiers

The symbol also keeps a reference to the original AST node for which it was generated, this is useful in some cases where we need to find out more information about the symbol later, it's mostly used when we try to access a tuple field, we can access the schema using that reference to the node.
There's also a field for keeping track of the symbol's type, it can be any of the following:

```
INT
BOOL
STRING
VOID
TUPLE
HTML
SCHEMA
UNDEFINED
```

Table 7: Symbol type

Here are some examples of symbol data generation:

Original code	Generated symbols (doesn't take into account scoping...)
const html Return = <html><[gap]><input type=”text” name=”YorN”></html>;	| Return, symbol_identifier: variable, type: html, + pointer to Return's AST node | | gap, symbol_identifier: hole_tag, type: html, + pointer to gap's AST node | | YorN, symbol_identifier: input_tag, type: html, + pointer to YorN's AST node |
int a;	a, symbol_identifier: variable, type: int, + pointer to a's AST node
int add(int a, int b){...}	add, symbol_identifier: function, type: int, + pointer to add's AST node
session B(){...}	B, symbol_identifier: session, type: void, + pointer to B's AST node

Table 8: Symbol data generation examples

3.3 Algorithm

The symbol table is represented by a `stack` of `map`s. Each level in the stack corresponds to a new scope.

The symbol table generation is implemented as a visitor class named `SymTabler`. It keeps track of a global symbol table. Whenever a `CompoundStm`, which represents statements inside a bracket, is encountered, it will create a new scope in this symbol table. It will traverse all the elements inside it and add each corresponding symbol to the symbol table. Then, it will create a copy of the global symbol table, link it into the `CompoundStm` node (`CompoundStm` derives from `Base` which has a field to keep a symbol table) and pop the last scope from the global symbol table and continue scanning the rest of the code.

Whenever 2 symbols with the same name are presented in the same scope, `SymTabler` will produce `SYMBOL_ALREADY_PRESENT` error. This situation is detected by checking that the top `map` element of the `stack` doesn't contain a symbol with the same name.

If we try to add a html tag that isn't an input or a gap into the symbol table, it will generate `HTML_TAG_NOT_A_SYMBOL` error.

The symbol table generation also associates with each symbol a unique identifier, it will become useful in the code generation phase.

3.4 Testing

My compiler was able to generate correct symbol tables for all different positions in the source code. The symbol table generation was tested using 2 different techniques. The first one is to print the symbol table for each scope in the pretty-printing code. So basically, the compiler parsed the input source code and pretty-printed it with the symbol tables and I checked that they did match the manually constructed symbol tables. The problem with that approach was that it's very manual and there's no easy way to do regression testing.

So I developed another way to write tests. It's basically using an augmented input file which includes extra metadata before the source code for the script. That metadata represents the compilation errors that this file must produce and the python script checks that these errors are produced. It overlaps with the testing for the typechecking system, so more information about this is given later in the next part.

4 Type Checking

4.1 Types

The types for the WIG language are in the `kType` enum class inside the `ast` namespace. They are the following:

Type	Information
`INT`	Integer, can be considered a BigInteger since the target language is Python which handles BigInteger natively
`BOOL`	Boolean, take true or false as values
`STRING`	Can be any string, multiline strings are not supported
`VOID`	Can only be the return value for a function, cannot declare variables of type void
`TUPLE`	Represents variables which are objects of a certain schema type
`HTML`	Global html variables
`SCHEMA`	Schema declaration

Table 9: WIG types

4.2 Type Rules

The following table describes the type rules I used to typecheck the input WIG file. I typecheck like 99% of all statements, the only things missing are the combine, keep and discard operations on tuples. I did typechecking on other uses of tuples, but didn't had enough time to implement it for these operations.

If at any moment, the checks on the types fail, the type of the statement or the expression will be set to `UNDEFINED`.

Statement or Expression	**Checks performed**	**Type of the Statement or Expression IFF the checks are successful**
if (exp) { ... }	exp = BOOL	VOID, cannot assign if statement to a variable
while (exp) { ... }	exp = BOOL	VOID, cannot assign while statement to a variable
lvalue		- if lvalue corresponds to a variable, RETURN that variable's type - if lvalue is a tuple field, find the corresponding schema and the type of the corresponding field, RETURN that type

lvalue = exp	lvalue's type = exp's type	lvalue's type
exp1 == exp2 exp1 != exp2	exp1's type = exp2's type	BOOL
exp1 < exp2 exp1 <= exp2 exp1 > exp2 exp1 >= exp2	exp1's type = INT AND exp2's type = INT	BOOL
exp1 + exp2	exp1's type = exp2's type AND exp1's type = INT OR STRING	exp1's type
exp1 – exp2 exp1 * exp2 exp1 / exp2 exp1 % exp2	exp1's type = INT AND exp2's type = INT	INT
exp1 && exp2 exp1 || exp2	exp1's type = BOOL AND exp2' type = BOOL	BOOL
function(arg1, arg2, ..., argn)	all the args have the same type as expected for the function input	Function's return type
show doc receive; exit doc;	doc's type = HTML AND none of the plugs have an undefined type (which can happen if we had 2+true) AND receive identifiers exist	VOID
function declaration	If function's return type isn't void, CHECK that the last statement is a return which has the same type as the function's return type	Function's return type
!exp (logic negate operation)	exp's type = BOOL	BOOL
-exp (unary minus)	exp's type = INT	INT
anything related to tuples	CHECK that the corresponding field has the same type as in the schema definition	TUPLE
Any integer		INT
true		BOOL
false		BOOL

Any string		STRING

Table 10: Type rules

4.3 Algorithm

The typechecking is implemented in the visitor class named `TypeChecker`. It uses global variables to keep track of the symbol table and expression type of the last seen expression. Whenever a new `CompoundStm` is encountered, the symbol table is updated to reflect the change. Then it's just a question of applying the rules described above.

At each step, the visitor is run on the children AST nodes which will update the last seen expression's type and we can then check these types and if the checks pass, we can update the last seen expression's type. If at anytime the checks fail, we can set the last seen expression's type to UNDEFINED.

The following is an example of how the typechecking works:

Typecheck the following:
int a;
a = 2 + 2;

1st step:
Update symbol table to have: a, symbol_identifier: variable, type: int (this happens when we reach a CompoundStm)

2nd step:
Typecheck:
a = 2 + 2;

3rd step:
Typecheck:
2 + 2;
First run typecheck on the left expression, once it finishes running, check that the last expression's type is INT.
Than run typecheck on the right expression, once it finishes running, check that the last expression's type is INT.
Update last expression's type to INT.

4th step (continuation of 2nd step):
a = 2 + 2;
First run typecheck on the left hand side, once it finishes running, store the last expression's type in a temporary variable.
Then run typecheck on the right hand side, once it finishes running, check that the type in the temporary variable is the same as the type of the last expression.
Update last expression's type to be INT.

Table 11: Typechecking example

4.4 Testing

I tested this phase using 2 methods. The first one is a manual method where I print all the types for the expressions next to them and than manually check that they correspond to the required types. For example:

Input	Output
if(2 == (a+b)){...}	If((2/*int*/ == (a/*int*/+b/*int*/))/*bool*/){...}

Table 12: Manual test example

The problem with that approach is it's really time consuming and it's difficult to do regression testing.

As I described in the previous part, I developed a python script which basically checks that appropriate errors are thrown from scripts which have problems. Metadatas are placed just before the beginning of the source code and that scripts check that they are all equal (their order isn't important, but the number of times they appear is because an error can happen multiple times in different places and the number of times it appears must be the same for both), for example:

Input file with metadata	Output file with metadata

/* **Expected errors:**
TYPES_DONT_MATCH(available = 2)
TYPES_DONT_MATCH(name = true)
TYPES_DONT_MATCH(price = "a")
SCHEMA_DONT_HAVE_FIELD(aaaa = "maybe")
SCHEMA_DONT_HAVE_FIELD(inexistent = false)
*/
service {
  const html a = <html></html>;
  schema Item {
  string name;
  int price;
  bool available;
  }
  session d () {
  tuple Item mi1;
  tuple Item mi2;
  tuple Item mi3;
  mi1 = tuple { name = "a",
  price = 3,
  available = true,
  inexistent = false};
  mi2 = tuple { name = true,
  price = "a",
  available = 2,
  aaaa = "maybe"};
  mi3 = mi1;
  mi3 = tuple { name = "a",
	/* **Produced errors:**
	SCHEMA_DONT_HAVE_FIELD(inexistent = false)
	TYPES_DONT_MATCH(name = true)
	TYPES_DONT_MATCH(price = "a")
	TYPES_DONT_MATCH(available = 2)
	SCHEMA_DONT_HAVE_FIELD(aaaa = "maybe")
	*/
	
	[... same code as on the left side ...]

`price = 1,`
`available = true};`
`exit a;`
`}`
`}`	

Table 13: Example of testing script

The typechecking phase can throw the following errors:

Error code	Example of code that would throw the error
SHOULD_BE_BOOL	if(2){} OR bool b; b = “true”;
TYPES_DONT_MATCH	int a; a = true;
CAN_COMPARE_INTEGERS_ONLY	“a” < “b” true >= false
OP_INT_ONLY	“test” - “st” true / false “maybe” % false
OP_INT_OR_STR_ONLY	true + false
OP_BOOL_ONLY	!2 “not bool” && true
NOT_A_FUNCTION	inexistent_function(“cool”);
NOT_SAME_NUMBER_PARAMETERS	int power(int a, int b){...} [...] power(2);
ARGUMENT_TYPE_DONT_MATCH	int power(int a, int b){...} [...] power(true, “a”);
HTML_DOESNT_EXIST	show inexistent_html;
SHOULD_BE_HTML	int a; show a;
FUNCTION_RETURN_TYPE_DONT_MATCH	int cool(){return true;}
PLUG_TYPE_UNDEFINED	show plug a[text=”a”+3];
SCHEMA_NOT_DEFINED	tuple inexistentSchema s1;
SCHEMA_DONT_HAVE_FIELD	tuple Item i1; i1.inexistent_field = 3;

5 Resource Computation

5.1 Resources

Since the target language for code generation is Python which is dynamic by nature, I didn't really need to compute any resources in advance.

5.2 Algorithm

5.3 Testing

6 Code Generation

6.1 Strategy

The overall strategy for generating code for the service involves finding equivalents in Python for the WIG constructs. The following table gives an overview:

WIG construct	**Python equivalent**
Html variables	Functions which take a dictionary representing the local variables. Inside the function, the template is represented using python's multiline string. In original wig templates, gaps are represented with "<[...]>", I replaced them with "{...}", so then I can just use ".format(**variableDictionaryFromInput)" to plug the variables where appropriate. A layout function which adds the extra html and form tags around the page's content.
Variables	Variables are represented in a global dictionary named __vars. Every variable is associated a unique identifier.
Global variables	Global variables use the same idea as variables, the only addition is that there's a global list in Python indicating which elements of __vars are global variables. It's used to serialize and deserialize them separately from the session variables.
Schemas/tuples	Python's class represents WIG's schema. In order to use same kind of syntax to initialize the tuple, I added a constructor which takes a dictionary and fills the respective object values. Tuple operations are implemented as object methods in the class.
Functions	Functions use the same algorithm as the sessions (described below) to separate the logic inside them into different labels. The only difference between the sessions and the functions is that a function can be recursive. Since it can also have show/exit statements inside, I decided to simulate a call stack. Whenever a function is called, it will create a new element on top of the simulated call stack, it will also copy all the local variables in order to be able to restore them when the function call returns. If we reach a position in the function where there is a show/exit statement, we will suspend and save the call stack to the same file as the session variables. On the subsequent requests, we will check if the call stack isn't empty, we will simulate it until it's not empty and then continue running the session's code from where it was left.
Sessions	This part is probably the most complicated, because there can be show/exit statements anywhere and then we will have to return the html client and when we receive a response back, we will have to figure out where we were and decide what's the next step we should take. I divided the logic inside a session into chunks. Each of these chunks is implemented as a function. When executed, they can output html or branch into other logic function. The logic functions are numbered starting from 1.

	Logic divisions are required because of if and while statements, because these statements would change the “Program Counter” and there are no goto or labels in Python, so they need to be functions. There are extra global variables which keep track of the state. When the user makes the request for the first time, there's no session id parameter in his request, so the first logic chunk is called. At the same time, a session file for the user is initialized. When we reach a point where we have to return an html to the client, we will just save user's session into it's file and we will also save the number of the next logic function to call. When the user receives the html, it will include the session id which will be sent on subsequent requests. When we receive a request with a session id, we will load the needed information from the appropriate file and call the corresponding logic function by using the number for the next logic call. Loading and saving the session file is done using Python's pickle library which serializes and deserializes Python objects.

6.2 Code Templates

The code templates can be found in `src/codegen/template.h` file. They are basically functions which take as input a string or a list of strings and they return a string representing the generated code. Since python is sensitive to whitespace, I have helper functions which help with indentation.
Most of the code template functions are self explanatory. They use `stringstream` in the implementation to construct the final generated code.
Almost all the generated functions have “`global __vars`” and “`global __next_logic`” as the first statements because most of them modify these values. At the end of each logic chunk, there's statements to save the session file and to increment the next_logic variable.
Most of the generated code variables are prefixed by 2 underscores to separate them from what was inside the WIG file initially.
The filenames where the script is saving the session files and global files have a UUID (Universally unique identifier) appended to them in order to avoid session collisions and different scripts using the same global files.

6.3 Algorithm

Like all the other phases, the code generation goes through the AST and generates the appropriate code as explained in the strategy above.

6.4 Runtime System

The runtime system uses modules from Python's standard library. The used modules are the following:

Runtime module	Why is it required ?
Cgi	Helps simplify parsing request queries
Cgitb	Simplifies cgi debugging
Os	Used to find the filename of the running script
Uuid	Used for generating UUID for each session files
Pickle	Used for serializing and deserializing state variables
Copy	Used for deepcopy call stack during function calls
Sys	Used for directing script errors to the browser
Traceback	Used for debugging

Table 14: Runtime modules

6.5 Sample Code

```
#!/usr/bin/env python
import cgi
import cgitb
import os
import uuid
import pickle
import copy
import sys
import traceback
cgitb.enable()
__cgi_input = cgi.FieldStorage(keep_blank_values=1)
__session = os.environ["QUERY_STRING"].split("&")[0]
sys.stderr = sys.stdout
__vars = {}
__sid = 0
__next_logic = 1
__vars["__exited_from"] = -1

def __str_sid():
     if __sid != 0:
          return "&sid="+str(__sid)
     else:
          return ""

def __action_name():
     return os.path.basename(__file__)+"?"+__session+__str_sid()

def __layout(page):
```

```
     return """<html><form action="{action}" method="POST">
     {page} <input type="submit" value="go">
     </form></html>""".format(action=__action_name(),page=page)

def __Welcome(__varDict):
     return """ <body>
    Welcome!
  </body> """.format(**(__varDict))

def __Pledge(__varDict):
     return """ <body>
    How much do you want to contribute?
    <input name="contribution" size=4 type="text">
  </body> """.format(**(__varDict))

def __Total(__varDict):
     return """ <body>
    The total is now {total}.
  </body> """.format(**(__varDict))

__global_vars = []
def __save_global_vars():
     global_vars_file =
"GLOBAL_6c79c797-fcb4-4698-b0b4-113fd2edb3df"
     open(global_vars_file, 'w').close()
     global_vars = dict((k, __vars[k]) for k in __global_vars if k
in __vars)
     with open(global_vars_file, "w") as f:
          pickle.dump(global_vars, f)
     return

def __load_global_vars():
     global __vars
     global_vars_file =
"GLOBAL_6c79c797-fcb4-4698-b0b4-113fd2edb3df"
     try:
          with open(global_vars_file, "r") as f:
               global_vars = pickle.load(f)
               __vars = dict(__vars.items() + global_vars.items())
     except IOError:
          return

__vars["amount_15_7"] = 0
__global_vars.append("amount_15_7")
__vars["__call_stack"] = []
__vars["__return_value"] = 0
__returned_from_fn = False
```

```
def __call_fn(fn_name):
     global __vars
     call_stack_copy = copy.deepcopy(__vars["__call_stack"])
     del __vars["__call_stack"]
     old_vars = copy.deepcopy(__vars)
     __vars["__call_stack"] = call_stack_copy
     __vars["__call_stack"].append({"name":fn_name,"next_logic":1,
"old_vars":old_vars})

def __set_fn_logic(n):
     global __vars
     __vars["__call_stack"][-1]["next_logic"] = n

def __return_from_fn(return_value):
     global __returned_from_fn
     global __vars
     call_stack_copy = copy.deepcopy(__vars["__call_stack"])
     __vars = __vars["__call_stack"][-1]["old_vars"]
     call_stack_copy.pop()
     __vars["__call_stack"] = call_stack_copy
     __returned_from_fn = True
     __vars["__return_value"] = return_value

def __continue_stack_execution():
     while True:
          if __vars["__call_stack"]:
               fn_name = __vars["__call_stack"][-1]["name"]
               fn_ln = __vars["__call_stack"][-1]["next_logic"]
               globals()["__logic_fn_"+fn_name+"_"+str(fn_ln)]()
          else:
               break
          if not __returned_from_fn:
               break

def __save_session_Contribute():
     session_file = "Contribute$"+str(__sid)
     open(session_file, 'w').close()
     session_vars = dict((k, __vars[k]) for k in __vars if k not in
__global_vars)
     with open(session_file, "w") as f:
          pickle.dump(session_vars, f)
          pickle.dump(__next_logic, f)
     return

def __init_session_Contribute():
     global __sid
     global __next_logic
     __sid = str(uuid.uuid4())
```

```
     __next_logic = 1
     __save_session_Contribute()
     __logic_session_Contribute_1()

def __load_session_Contribute(session_id):
     global __vars
     global __next_logic
     global __sid
     __sid = session_id
     with open("Contribute$"+str(__sid), "r") as f:
          session_vars = pickle.load(f)
          __next_logic = pickle.load(f)
          __vars = dict(__vars.items() + session_vars.items())
     if __vars["__exited_from"] != -1:
          __next_logic = __vars["__exited_from"]
          __save_session_Contribute()
          globals()
["__logic_session_Contribute_"+str(__vars["__exited_from"])]()
          return
     __continue_stack_execution()
     globals()["__logic_session_Contribute_"+str(__next_logic)]()

def __session_Contribute():
     sid = __cgi_input.getvalue("sid", "")
     if sid == "":
          __init_session_Contribute()
     else:
          __load_session_Contribute(sid)

def __logic_session_Contribute_1():
     global __vars
     global __next_logic
     __vars["i_18_8"] = 87
     print(__layout(__Welcome({})))
     __next_logic = 2
     __save_session_Contribute()
def __logic_session_Contribute_2():
     global __vars
     global __next_logic
     print(__layout(__Pledge({})))
     __next_logic = 3
     __save_session_Contribute()
def __logic_session_Contribute_3():
     global __vars
     global __next_logic
     __vars["i_18_8"] = int(__cgi_input.getvalue("contribution"))
     __next_logic = 4
     __save_session_Contribute()
```

```
     __logic_session_Contribute_4()
def __logic_session_Contribute_4():
     global __vars
     global __next_logic
     __vars["amount_15_7"] = (__vars["amount_15_7"] +
__vars["i_18_8"])
     print(__layout(__Total({'total':__vars["amount_15_7"]})))
     __vars["__exited_from"] = 4
     __next_logic = 5
     __save_session_Contribute()
def __logic_session_Contribute_5():
     global __vars
     global __next_logic
     __next_logic = 4
     __save_session_Contribute()
     __logic_session_Contribute_4()
print "Content-type: text/html"
print
__load_global_vars()
if __session == "Contribute":
     __session_Contribute()
else:
     print __layout("Please select one of the following sessions:
Contribute")
__save_global_vars()
```

Table 15: Compiled tiny.wig example

As can be seen from the generated code, it works but there is a lot of room for optimizations.

6.6 Testing

For this part, I compiled the benchmark scripts in the `tests/codegen/` folder. I manually tested them and all of them worked including the `factorial.wig`. The only bug I have is in the game.wig, the winning guess number needs to be entered twice before it's accepted.

7 Availability and Group Dynamics

7.1 Manual

Services can be compiled with the following command:

```
fwig -c -o [output_filename] [input_filename]
```

Table 16: Example of compilation

You can also run fwig without any arguments to display a help message. Once it's compiled, you can set the appropriate permissions and run the .py file from cgi-bin.

7.2 Demo Site

http://cgi.cs.mcgill.ca/~ftufek/cgi-bin/

The following demos are available:

001.py	004.py	counter.py	gameTuple.py	talk.py
002.py	calculator.py	factorial.py	legomen.py	tiny.py
003.py	chat.py	game.py	riddles.py	tupletorture.py

Table 17: Available demos

In order to run them, you can just append their names at the end of the URL. If you want to see the code behind them, they are available in the tests/codegen folder.

7.3 Division of Group Duties

N/A

8 Conclusions and Future Work

8.1 Conclusions

So this report explaind the different phases involved in my implementation of a WIG compiler. I discussed about parsing, symbol table generation, typechecking and code generation.
I learned different things like flex and bison. I also learned a lot about C++ which I wasn't very familiar with at the beginning of the project.
Before that project, I always thought that the compiler always generates assembly code and I learned that it wasn't the case, that it was more general than that.
I learned a lot about how to debug different kind of code. For example debugging my simulated function call stack in the python cgi script was very instructive.

8.2 Future Work

I think a standard library with different modules could be very useful. For example there could be a function named random() which would return a random number, it wouldn't even require an import. It could be provided by doing a translation from the “WIG standard library” into the very comprehensive Python's standard library.
I think having a way to share code could be useful too. We would have a script that only defined functions and scripts could include that file before declaring the service.
My compiler has a lot of room for optimization. The generated code is very primitive. For example the tuple operations could be placed in a base class from which every Schema derives. I should also find a way to divide the session logic into lesser chunks by using a smarter analysis. Also the code for saving and loading session files and global file could be combined into a single function, right now it generates 3 functions for each session (load_session, save_session, init_session) and 2 functions for the global file (load_global_variables, save_global_variables).

8.3 Course Improvements

I think writing a compiler for WIG is very instructive, but at the end it still remains that WIG isn't really a modern or widely used language. I think it would be very interesting if we did compile an actual popular language. For example, Ruby on Rails and Django are widely used frameworks for doing web development. Their main drawback is their performance. So we could write a compiler that will take RoR or Django project as input and will output Java Servlet's (which are very fast). If successful, this project could have a real world usage. Of course, I realize that it's a very difficult task and probably impossible too, so we would limit ourselves to basics like we did for the bigwig language.

8.4 Goodbye

I think compilers are very interesting piece of software and that's why I took this class, but I don't have any plans for now to do research on compilers.