McGill University
School of Computer Science
COMP 520

# Wig Compiler Report

Report No. 2011-d

Ismail Badawi
Carla Morawicz

November 29, 2011

# Contents

# 1 Introduction

## 1.1 Clarifications

Several aspects of the WIG language are unclear or ambiguous, many of them of minor importance. The big issues mainly concern the type-correctness of various language constructs. Chief amongst these issues is the behavior of tuples and schemas; when are two tuples equivalent, do tuples necessarily need to adhere to a declared schema, what exactly are the semantics of the three tuple operations, and so on.

The `show`, `exit`, `plug` and `receive` constructs can also be treated with varying levels of strictness (and the provided compilers do implement them in different ways). What types is a hole allowed to be plugged with (only strings, or can the other types be coerced to strings)? What types can be received from `show` or `exit` statement? Must all holes be plugged? What happens if a hole is plugged twice? Or if we try to plug a missing hole?

Our general approach has been very permissive. For tuples, we consider structural equivalence; two tuples are type-equivalent (and can be assigned to each other) if every field that appears in one appears in the other, and with the same type. In particular, schema names, along with the order in which the fields are given, are ignored.

When it comes to templating, we usually opt for silent failures. Holes can be plugged twice; we use the rightmost value. Gaps can be omitted, empty strings are used instead. A hole can be plugged by any type, including tuples – we provide a suitable string representation. Tuples cannot be received, however – parsing tuples from an HTML input would not be out of the question, but doesn't seem useful from the end user's perspective. We don't allow plugging missing holes; there doesn't appear to be any situation in which this is useful.

## 1.2 Restrictions

Our version of WIG has one restriction; `show` and `exit` statements are not allowed inside functions. In practice, this probably would not have terribly difficult to implement, considering we're able to do it for sessions (there are just a couple of big picture complications, chiefly relating to how function arguments are handled), but we didn't plan for it from the beginning, and once we got down to it it didn't seem all that important, considering exactly zero of the past benchmarks make use of this feature.

## 1.3 Extensions

Aside from minor syntactic sugar like standard increments and `for` loops, we haven't made any extensions to our version of WIG.

## 1.4 Implementation Status

Our WIG implementation is feature-complete; it is able to generate working WIG services for nearly all of the past benchmarks. Notably, it is able to compile our benchmark (`hangman.wig`), which previously only `lkwig` could compile.

# 2 Parsing and Abstract Syntax Trees

## 2.1 The Grammar

Our `bison` grammar follows.

---

```
%right '=' tPLUSEQ tMINEQ tMULTEQ tDIVEQ tMODEQ tANDEQ tOREQ
%left tOR
%left tAND
%nonassoc tEQ tLEQ tGEQ tNEQ '<' '>'
%left '+' '-'
%left '*' '/' '%'
%left tKEEP tDISCARD
%right tCOMBINE
%right '!' tUMINUS tINC tDEC

%%

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

htmls : html
        | htmls html
;

html : tCONST tHTML tIDENTIFIER '=' tHTMLOPEN nehtmlbodies tHTMLCLOSE ';'
     | tCONST tHTML tIDENTIFIER '=' tHTMLOPEN tHTMLCLOSE ';'
;

nehtmlbodies : htmlbody
             | nehtmlbodies htmlbody
;

htmlbody : '<' tIDENTIFIER attributes '>'
         | '<' tIDENTIFIER attributes tCLOSINGTAG
         | tOPENINGTAG tIDENTIFIER '>'
         | tOPENINGGAP tIDENTIFIER tCLOSINGGAP
         | tWHATEVER
         | tMETA
         | '<' tINPUT inputattrs '>'
         | '<' tSELECT inputattrs '>' nehtmlbodies tOPENINGTAG tSELECT '>'
         | '<' tSELECT inputattrs '>' tOPENINGTAG tSELECT '>'
;

inputattrs : inputattr
           | inputattrs inputattr
;

inputattr : tNAME '=' attr
          | tTYPE '=' inputtype
```

```
          | attribute
;

inputtype : tTEXT
          | tRADIO

attributes : /* empty */
           | neattributes
;

neattributes : attribute
             | neattributes attribute
;

attribute : attr
          | attr '=' attr
;

attr : tIDENTIFIER
     | tSTRINGCONST
;

schemas : /* empty */
        | neschemas
;

neschemas : schema
          | neschemas schema
;

schema : tSCHEMA tIDENTIFIER '{' fields '}'
;

fields : /* empty */
       | nefields
;

nefields : field
         | nefields field
;

field : simpletype tIDENTIFIER ';'
;

nevariables : variable
            | nevariables variable
;

variable : type identifiers ';'
;

identifiers : tIDENTIFIER
            | identifiers ',' tIDENTIFIER
```

```
;

simpletype : tINT
           | tBOOL
           | tSTRING
           | tVOID
;

type : simpletype
     | tTUPLE tIDENTIFIER
;

functions : /* empty */
          | nefunctions
;

nefunctions : function
            | nefunctions function
;

function : type tIDENTIFIER '(' arguments ')' compoundstm
;

arguments : /* empty */
          | nearguments
;
nearguments : argument
            | nearguments ',' argument
;

argument : type tIDENTIFIER
;

sessions : session
         | sessions session
;

session : tSESSION tIDENTIFIER '(' ')' compoundstm
;

stms : /* empty */
     | nestms
;
nestms : stm
       | nestms stm
;
stm : ';'
    | tSHOW document receive ';'
    | tEXIT document ';'
    | tRETURN ';'
    | tRETURN exp ';'
    | tIF '(' exp ')' stm
    | tIF '(' exp ')' stmnoshortif tELSE stm
```

```
    | tWHILE '(' exp ')' stm
    | tFOR '(' optionalexp ';' optionalexp ';' optionalexp ')' stm
    | tUNTIL '(' exp ')' stm
    | compoundstm
    | exp ';'
;

stmnoshortif : ';'
    | tSHOW document receive ';'
    | tEXIT document ';'
    | tRETURN ';'
    | tRETURN exp ';'
    | tIF '(' exp ')' stmnoshortif tELSE stmnoshortif
    | tWHILE '(' exp ')' stmnoshortif
    | tUNTIL '(' exp ')' stmnoshortif
    | tFOR '(' optionalexp ';' optionalexp ';' optionalexp ')' stmnoshortif
    | compoundstm
    | exp ';'
;

document : tIDENTIFIER
         | tPLUG tIDENTIFIER '[' plugs ']'
;
receive : /* empty */
        | tRECEIVE '[' inputs ']'
;

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

plugs : plug
      | plugs ',' plug
;

plug : tIDENTIFIER '=' exp
;

inputs : /* empty */
       | neinputs
;

neinputs : input
         | neinputs ',' input
;

input : lvalue '=' tIDENTIFIER
;

exp : lvalue
    | lvalue '=' exp
    | lvalue tPLUSEQ exp
    | lvalue tMINEQ exp
```

```
    | lvalue tMULTEQ exp
    | lvalue tDIVEQ exp
    | lvalue tMODEQ exp
    | lvalue tANDEQ exp
    | lvalue tOREQ exp
    | tINC lvalue
    | tDEC lvalue
    | exp tEQ exp
    | exp tNEQ exp
    | exp '<' exp
    | exp '>' exp
    | exp tLEQ exp
    | exp tGEQ exp
    | '!' exp
    | '-' exp %prec tUMINUS
    | exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | exp '%' exp
    | exp tAND exp
    | exp tOR exp
    | exp tCOMBINE exp
    | exp tKEEP tIDENTIFIER
    | exp tKEEP '(' identifiers ')'
    | exp tDISCARD tIDENTIFIER
    | exp tDISCARD '(' identifiers ')'
    | tIDENTIFIER '(' exps ')'
    | tINTCONST
    | tBOOLCONST
    | tSTRINGCONST
    | tTUPLE '{' fieldvalues '}'
    | '(' exp ')'
;

optionalexp : /* empty */
            | exp

exps : /* empty */
     | neexps
;

neexps : exp
       | neexps ',' exp
;

lvalue : tIDENTIFIER
       | tIDENTIFIER '.' tIDENTIFIER
;

fieldvalues : /* empty */
            | nefieldvalues
;
```

```
nefieldvalues : fieldvalue
              | fieldvalues ',' fieldvalue
;

fieldvalue : tIDENTIFIER '=' exp
;
```

---

## 2.2 Using the `flex` or `SableCC` Tool

Our scanner includes the following start conditions:

- WIG – The initial state, where all WIG keywords are reserved.
- HTML – Entered after `<html>` is scanned and left when `</html>` is scanned. We handle meta tags properly so long as they do not end with a -; that is, `<!-- is fine -->`, but `<!-- isn't --->`, for simplicity; correctly recognizing the second would require a more complicated action in the same vein as the one for block comments, but we also want in this case to store the content of the tag, and that makes things less straightforward.
- HTMLTAG – Entered from HTML when a `<` or `</` is scanned, and left when `>` or `/>` is scanned. Here `input` and `select` are reserved words.
- HTMLINPUT – Entered from HTMLTAG if the tag's name is `input`.
- TYPERHS – Entered from HTMLINPUT when `type=` is scanned. In this state only `"text"`, `text`, `"radio"` and `radio` can be scanned. This sequence of states (HTMLTAG, HTMLINPUT, TYPERHS) is needed because other tags (notably, `<link>`) may have a `type` attribute that takes other values.
- HTMLRHS – Entered from HTMLTAG when a `=` is scanned, and left after either a string constant is scanned, or an identifier. "Identifiers" here can start with a number, so that e.g. `size=20` is valid.
- INPUTRHS – same as HTMLRHS, but knows to return to HTMLINPUT instead of HTMLTAG.
- GAP – Entered from HTML when `<[` is scanned and left when `]>` is scanned. (We played around with the provided compilers and found that gaps are usually not recognized within tags.) Here nothing is reserved.

Some fairly subtle bugs were found to have been caused by our initially naive use of start conditions. For example, we had a case where

```
<input name="test" type="text">
```

would parse properly, but

```
<input type="text" name="test">
```

would cause a syntax error. Our flex scanner contains a fair amount of code duplication as a result, as some rules appear several times in different states, the only difference being what state they begin in their associated action. There doubtless exist nicer ways of dealing with this sort of thing.

## 2.3 Using the `bison` or `SableCC` Tool

The provided start grammar included 273 shift/reduce conflicts. Most of them (that is, all but one) were resolved by specifying operator precedences using `%left`, `%right` and `%nonassoc` directives. One was caused by the "dangling else" problem discussed in the readings; taking a cue from the JOOS compiler, we introduced a `stmnoshortif` production for the `then` part of an `if ... else` statement.

## 2.4 Abstract Syntax Trees

Our abstract syntax trees are composed of 18 node types, namely `SERVICE`, `SESSION`, `HTML`, `HTMLBODY`, `ATTRIBUTE`, `PLUG`, `RECEIVE`, `INPUT`, `SCHEMA`, `ID`, `VARIABLE`, `FUNCTION`, `TYPE`, `STATEMENT`, `DOCUMENT`, `EXP`, `ARGUMENT`, and `FIELDVALUE`, all mapping straightforwardly to elements of the WIG language.

## 2.5 Desugaring

We provide the following constructs, implemented as syntactic sugar:

- `for` and `until` loops, which are transformed into equivalent while loops
- The compound assignment operators `+=`, `-=`, `*=`, `/=`, `%=`, `&&=` and `||=`, which are transformed into equivalent assignments
- Standard prefix increment and decrement operators, which are transformed into equivalent assignments (we don't provide the postfix versions; typically those change the value and return the old one, and there isn't really a way to do that in WIG, so implementing them as syntactic sugar would be difficult, to say the least).

Each of these corresponds to an extra token recognized by the scanner, and one or two extra productions in the parser. For instance, `until` loops are implemented as simply as:

```
stm : ...
    | tUNTIL '(' exp ')' stm
      { $$ = makeSTATEMENTwhile(makeEXPnot($3), $5); }
      ...
```

## 2.6 Weeding

The following cases are weeded out (straightforwardly, by traversing the AST):

- variables of type `void`, in any context (i.e. as arguments to functions, as global variables, as local variables in sessions or functions, and inside schemas.)
- sessions that don't always `exit`
- `void` functions where something is returned
- non `void` functions where nothing is returned
- `<input>` tags without `name` or `type` attributes
- `<select>` tags without a `name` attribute
- `return` statements in sessions
- `exit` and `show` statements in functions

There are a few cases that we considered weeding but ultimately decided not to (in keeping with our general permissive approach):

- empty schema declarations, e.g. `schema S {}`. This is never useful, but we don't really the harm in it.
- `<input>` tags with more than one name, or more than one type. This doesn't really cause trouble because the order in which we traverse the AST is always the same, so even though there's more than one name or type, it's actually well defined which one we'll use.
- `<select>` tags with more than one name, similarly

## 2.7 Testing

The validity of the scanning and parsing phase is verified by checking whether `pretty(parse(x)) == pretty(parse(pretty(parse(x))))` holds for every WIG service x. This is obviously undecidable, so in practice we just tried for many many WIG services. A representative sample is included in the `group-d/wig/tests` directory, and this property is checked for these during the build process.

# 3 Symbol Tables

## 3.1 Scope Rules

There is one global scope; in particular, functions, HTMLs, schemas, global variables and sessions all reside in the same namespace, so their names may not clash (`wigismo`'s error messages might be slightly misleading in this respect). New scopes are introduced by blocks, which can occur anywhere, bringing with them variable declarations. Variable names in nested scopes shadow those in outer scopes. A variable name may not be reused inside a single scope.

## 3.2 Symbol Data

Each `SYMBOL` just keeps a pointer to the relevant AST node. From there we can get at any information we deem relevant, such as the line number on which the symbol appears, its type if it's a variable or an argument, its return type if it's a function, and so on.

## 3.3 Algorithm

We implement the cactus stack structure for our symbol tables as seen in class. There is a global symbol table for the whole service (`mst`, declared in `symbol.h`); in addition, whenever a block statement is encountered (that is, whenever a new scope is entered, because blocks are the only way to introduce new scopes in WIG), the node for that statement keeps a pointer to its local symbol table, which keeps a pointer to the symbol table of the next outer scope. First, we insert all html literals, schemas, global variables, functions and sessions into the `mst`. Then we traverse the AST, keeping track of which scope we're in. When a new scope is entered, all of its local variable declarations are inserted into the table for that statement (`blockS.table`), and in so doing are checked for uniqueness. Whenever an identifier is used, we check the current scope and work our way up to the `mst` looking for a match, reporting an error if there isn't one.

## 3.4 Testing

The symbol table phase was tested by constructing a minimal WIG service for each possible sort of error, and running wigismo on each of them, checking that an appropriate error message is printed, and compilation is terminated. These test cases all live in `group-d/wig/tests/symbol`, and cover the following situations:

- Calling undeclared functions
- Using the same name for different arguments to a function
- Using the same name for local variables in the same scope
- Using the same name for two global entities
- Plugging an undeclared hole
- Showing or exiting with an undeclared HTML
- Receiving an undeclared input
- Receiving an input into an undeclared variable or undeclared field of a tuple
- Declaring a tuple with an undeclared schema
- Assigning to an undeclared variable, or undeclared field of a tuple
- Using undeclared variables or undeclared fields of tuples in expressions

Further, wigismo's pretty printer can be instructed to dump the contents of the symbol table whenever a new scope is entered. Each new scope is marked by an extra level of indentation. This allows us to compare the results of the symbol table phase against a manual construction. For instance, the following (rather contrived) service:

```
service {
    const html Test = <html>Test</html>;
    schema S {
        int x;
    }
    int x;
    int test(bool x) {
        int y;
        y = 10;
        while (y > 0) {
            string x;
            y = y - 1;
        }
        return 0;
    }
    session Main() {
        tuple S x;
        x.x = test(true);
        exit Test;
    }
}
```

produces the following pretty-printer output:

```
service
{
    ================================================
    =============|Symbols in this Scope|=============
    ================================================
     S: schema
     x: int variable
     test: function returning int
     Test: html
     Main: session
    ================================================

    const html Test = <html>Test</html>;

    schema S {
        int x;
    }

    int x;
    int test(bool x)
    {
        ================================================
        =============|Symbols in this Scope|=============
        ================================================
         S: schema
```

```
         x: int variable
         test: function returning int
         Test: html
         Main: session
             x: bool argument
             y: int variable
        ================================================

        int y;
        y = 10;
        while (y > 0)
        {
            ================================================
            =============|Symbols in this Scope|=============
            ================================================
             S: schema
             x: int variable
             test: function returning int
             Test: html
             Main: session
                 x: bool argument
                 y: int variable
                     x: string variable
            ================================================

            string x;
            y = (y - 1);
        }
        return 0;
    }

    session Main()
    {
        ================================================
        =============|Symbols in this Scope|=============
        ================================================
         S: schema
         x: int variable
         test: function returning int
         Test: html
         Main: session
             x: tuple S variable
        ================================================

        tuple S x;
        x.x = test(true);
        exit Test;
    }
}
```

# 4 Type Checking

## 4.1 Types

The WIG language has support for three primitive types, namely `int`, `bool`, and `string`. It also has tuples, which are collections of key-value pairs adhering to a `schema`.

## 4.2 Type Rules

- A `service` is type correct if the sessions and functions within are type correct.
- A session or function is type correct if its body is type correct.
- The skip statement (`;`) is always type correct.
- A sequence `s1, s2` of statements is type correct if `s1` and `s2` are both type correct.
- The expressions plugged into a `show` statement must type check; the variables received from a `show` must not be of a tuple type.
- The expressions plugged into an `exit` statement must type check.
- A `return` statement is type correct if the type of the expression returned is compatible with the return type of the function. (Return statements in sessions do not pass the weeder.)
- Block statements are type correct if their body is type correct.
- An `if` statement is type correct if the condition expression is of type `bool` and the body is type correct.
- An `if ... else` statement is type correct if the condition expression is of type `bool` and both the then part and the else part of the statement are type correct.
- A `while` statement is type correct if the condition expression is of type `bool` and the body is type correct.
- An expression statement is type correct if the expression is type correct.
- The left and right operands of the arithmetic opreators `-`, `*`, `/`, and `%` must be of type `int`; the result is of type `int`.
- For addition, the two operands must be of type `string` or `int`. If either or both are `string`s, the result is a `string`, otherwise it's an `int`.
- The type of the expression on the right hand side of a `=` must either be of the same type or a compatible type as the identifier on the left hand side; the result is the type that is evaluated from the right-hand side expression.
- The left and right operatands of the boolean operators `&&` and `||` must be of type `bool`; the result is of type `bool`.
- The left and right operands of the relation operators `<`, `>`, `<=`, `>=` must be of type `int`; the result is of type `int`.
- For the equality operators `==` and `!=`, the types of the left and right operands must be the same or compatible, where compatibility in the case of tuples refers to structural equivalence – that is, every variable that appears in one of the schemas also appears in the other schema with the same type – the result is of type `bool`.
- For the unary minus operator, the expression being negated must have type `int`; the result is of type `int`.

- For the negation operator `!`, the expression being negated must type `bool`; the result is of type `bool`.
- The types of the arguments of a function call must correspond to the types declared in the function's signature; the type of the expression is the return type of the function.
- Any integer constant (matching the pattern `0|[1-9][0-9]*`) has type `int`.
- The literals `true` and `false` are of type `bool`.
- Any double quoted string is of type `string`.
- Tuple literals have a tuple type – their schema is defined implicitly by the types of their field values, assuming they type check.
- For the keep and discard operators (`\+` and `\-`), the left operand must be of type tuple; each identifier appearing on the right hand side must appear in its schema; the result is tuple type with a new schema generated automatically.
- For the combine operator (`<<`), the left and right operands must be of type tuple, the common variables in their schemas must have the same type, and the result is a tuple whose schema is composed of the union of their variables.

## 4.3 Algorithm

Type checking is done by traversing the AST and storing the computed type of each expression in the `TYPE` field of the corresponding `EXP` AST node. This process is largely straightforward; to constants and literals we can immediately associate a type, for identifiers we simply read the information gathered during the symbol table phase, and from there the type of each expression can be obtained in accordance with the type rules. Once each expression has been annotated with its type, we have enough information to enforce the type rules for statements, functions, sessions, and finally the entire service.

## 4.4 Testing

As in the symbol table phase, a suite of minimal WIG services that provoke every possible error message is provided. These live in `group-d/wig/tests/type`, and cover the following situations:

- Combining incompatible tuples
- Discarding or keeping a missing field from a tuple
- Using anything but a boolean variable in the condition of an `if` or `while` statement
- Comparing non-comparable types
- Assigning to a variable the result of an expression whose type it is not assignable from
- Receiving tuples
- Returning from a function an expression whose type is not compatible with the function's return type
- Using the arithmetic operators (except `+`) with anything but integers
- Using the relational operators with anything but booleans
- Calling functions with incompatible argument types

Also as in the symbol phase, wigismo's pretty-printer was enhanced to annotate every expression with its type. Anonymous tuples are given unique generated schema names. The service above produces the following output:

```
service
{
    const html Test = <html>Test</html>;

    schema S {
        int x;
    }

    int x;
    int test(bool x)
    {
        int y;
        (y = (10 : int) : int);
        while (((y : int) > (0 : int) : bool))
        {
            string x;
            (y = ((y : int) - (1 : int) : int) : int);
        }
        return (0 : int);
    }

    session Main()
    {
        tuple S x;
        (x.x = (test((true : bool)) : int) : int);
        exit Test;
    }
}
```

# 5 Resource Computation

## 5.1 Resources

We assign unique integer identifiers to every `if`, `if ... else`, and `while` statement; these are used to generate appropriate function names (as described in the code generation section) and are analogous to labels for `goto` statements in jasmin.

We also assign unique integer ids to every variable. There isn't a clean way to introduce arbitrary blocks in python, so scoping can be awkward. Even if we opt for an ugly solution (e.g. use `if True:` to introduce blocks), we still have to deal with the odd control flow induced by `show` statements. Making every variable unique in this way makes saving local state very easy – we can just have a big dictionary of local variables, since every key is unique (in fact, we could dispense with the variable names in the generated code entirely, but we keep them to make it easier to read.)

## 5.2 Algorithm

We maintain a global counter, and traverse the AST. Whenever an `if`, `if ... else`, or `while` statement is encountered, we increment the counter and assign it to the statement.

Assigning ids to variables is done during the parsing phase; an incrementing id is assigned whenever a `VARIABLE` node is created.

## 5.3 Testing

There isn't very much to test here. This phase is necessary for the code generation phase; verifying that this latter phase works is sufficient.

# 6 Code Generation

## 6.1 Strategy

`wigismo` generates python code. All of the WIG language constructs, with the notable exception of `show` and `exit`, have equivalents in python. A `while` loop is a `while` loop, a `||` operator is an `or` operator, and so on, but the semantics of `show` makes for less than straightforward control flow. The big picture is as follows:

- Every service defines two variables: `g`, an instance of `wigismo.Store`, which stores global variables, and `l`, a dictionary which stores local variables. Variables are all given unique integer ids, so this one dictionary is sufficient (that is, scoping is not an issue). No initialization happens; every access to a variable, local or global, is wrapped in a call to `get`, with a suitable default value based on type.
- HTMLs become functions which print the contents of the HTML. Holes are the arguments to the function, and are inserted into the output using string interpolation. `show` and `exit` statements partly correspond to just calling these functions, passing any values plugged as arguments. (python supports both default arguments and keyword arguments; holes are all given a default value of `''`, and the HTML functions are always called with every argument named, so in this way holes can be omitted easily.)
- Schemas don't appear in the generated code at all.
- Functions map to ordinary python functions. Since we don't allow `show` or `exit` statements in functions, this is straightforward.
- Sessions become sets of functions. Since python doesn't support the `goto` statement, every session is split into several functions, one for every possible entry or jump point (the start of a loop, after a loop, after an `if`, the start of an `else`, after an `else`, and after a `show`). This is necessary because regular control flow is often impossible to apply (e.g. what if we have a `show` statement inside a loop?)
- When a `show` statement is encountered, the locals dictionary is written to a file, along with the name of the function to call when this session is resumed (this function is called using reflection). If any values are to be received, this happens at the beginning of that function.
- In some cases we have to be careful about generating valid code; for instance, python contains many reserved keywords which aren't reserved in WIG (a common one is `pass`), so we append underscores to function argument names and hole names to avoid conflicts. HTML literals end up in python strings, so some characters (backslashes, single quotes, newlines) have to be escaped.

## 6.2 Code Templates

Python has significant whitespace; blocks are marked by levels of indentation. This makes certain ideas in the template difficult to show (as some templates reset the indentation level). In what follows, take leading whitespace to mean "some unspecified but nonzero level of indentation". Also, assume every snippet resides in a session `S`, and `N1` and `N2` stand for the unique ids computed during the resource computation phase, and `N` is an integer id, incremented whenever code for a `show` statement is generated. `D` is a type appropriate default value – `0` for `int`, `False` for `bool`, `''` for tuples. `l` and `g` refer to the locals dictionary and `wigismo.Store` object mentioned above.

We omit trivial templates (e.g. `E1 - E2` → `E1 - E2`).

| Construct | Template |
|---|---|
| `x` | if `x` is a global variable<br>`g.get('x_N', D)`<br>if `x` is a local variable<br>`l.get('x_N', D)`<br>if `x` is a function argument<br>`x` |
| `x = E` | if `x` is a global variable<br>`g.set('x_N', E)`<br>if `x` is a local variable<br>`wigismo.set(l, 'x_N', E)`<br>if `x` is a function argument<br>`x = E` |
| `E1 + E2` | if `E1` and `E2` have the same type<br>`E1 + E2`<br>otherwise<br>`str(E1) + str(E2)` |
| `E1 \+ (f1, ..., fn)` | `wigismo.tuple_keep(E1, f1, ..., fn)` |
| `E1 \- (f1, ..., fn)` | `wigismo.tuple_discard(E1, f1, ..., fn)` |
| `E1 << E2` | `wigismo.tuple_combine(E1, E2)` |

| Construct | Template |
|---|---|
| `exit plug H [g1=x1,...,gn=xn];` | `wigismo.output(sessionid,`<br>`  lambda: output_H(g1_=x1, ..., gn_=xn, exit=True))`<br>`sys.exit(0)` |
| `show plug H [g1=x1,...,gn=xn]`<br>`  receive [r1=i1,...,rn=xn]` | `wigismo.output(sessionid,`<br>`  lambda: output_H(g1_=x1, ..., gn_=xn))`<br>`state = wigismo.Store(sessionid)`<br>`state.set('locals', l)`<br>`state.set('start', 'session_S_show_N')`<br>`sys.exit(0)`<br><br>`def session_S_show_N(sessionid):`<br>`  # assign wigismo.get_field('i1', t1) to r1`<br>`  # where t1 is the type of r1`<br>`  # using the appropriate assignment template` |
| `if (C)`<br>`  B` | if `B` has no `show` statement:<br>`if C:`<br>`  B`<br><br>otherwise<br>`if not (C):`<br>`  session_S_N1(sessionid)`<br>`B`<br>`session_S_N1(sessionid)`<br><br>`def session_S_N1(sessionid):`<br>`  # set indent` |

<table>
<tr>
<td>

```
if (C)
    B1
else
    B2
```

</td>
<td>
if B1 and B2 have no show statements

```
    if C:
        B1
    else:
        B2
```

otherwise

```
        if not (C):
            session_S_N1(sessionid)
        B1
        session_S_N2(sessionid)

    def session_S_N1(sessionid):
        B2
        session_S_N2(sessionid)

    def session_S_N2(sessionid):
        # set indent
```

</td>
</tr>
<tr>
<td>

```
while (C)
    B
```

</td>
<td>
if B has no show statements

```
    while C:
        B
```

otherwise

```
        if C:
            session_S_N1(sessionid)
        session_S_N2(sessionid)

    def session_S_N1(sessionid):
        B
        if C:
            session_S_N1(sessionid)
        else:
            session_S_N2(sessionid)

    def session_S_N2(sessionid):
        # set indent
```

</td>
</tr>
</table>

## 6.3 Algorithm

The algorithm for generating code is again a largely straightforward traversal of the AST. A global variable keeps track of the current indentation level, and we also have tables of strings that map, for instance, WIG operators and types to their python equivalents (so that `string` becomes `str`, `||` becomes `or`, and so on), as well WIG types to suitable default values (so that `int` maps to `0`, for example). Then we go through and simply print out the appropriate template for every statement, expression, function, and so on.

## 6.4 Runtime System

Since `wigismo` generates python code, we have access to a very large standard library. The functionality we need has been encapsulated into the `wigismo.py` module, which provides the following facilities:

- the `random_string` function generates random strings, for use in generating session identifiers (using python's `random` module)
- the `get_field` function handles parsing and retrieving CGI fields, and coercing them to appropriate types (using python's `cgi` module)
- the `output` function wraps the output HTML in a form if necessary (a neat use of higher order functions)
- the `Store` class provides thread-safe storage of key-value pairs to a file, for use in maintaining global state, and dumping local state following a `show` statement
- the `tuple_keep`, `tuple_discard` and `tuple_combine` functions implement the three tuple operations
- While assignments in python can be chained (e.g. `x = y = 4`), a statement like `x = (y = 2) + 3` raises a `SyntaxError`; that is, assignments aren't really expressions in python. To get around this, the `set` function performs the assignment and returns the value assigned.

## 6.5 Sample Code

```
#!/usr/bin/python
import sys
import wigismo
import cgitb
cgitb.enable()

g = wigismo.Store('tiny_globals.pck')
l = {}

def output_Welcome():
    sys.stdout.write(' ')
    sys.stdout.write('<body >')
    sys.stdout.write('\n    Welcome!\n  ')
    sys.stdout.write('</body>')
    sys.stdout.write(' ')

def output_Pledge():
    sys.stdout.write(' ')
    sys.stdout.write('<body >')
    sys.stdout.write('\n    How much do you want to contribute?\n    ')
    sys.stdout.write('<input name="contribution" type="text" size="4" >')
    sys.stdout.write('\n  ')
    sys.stdout.write('</body>')
    sys.stdout.write(' ')

def output_Total(total_=''):
    sys.stdout.write(' ')
    sys.stdout.write('<body >')
    sys.stdout.write('\n    The total is now ')
    sys.stdout.write('%s' % total_)
```

```
    sys.stdout.write('.\n  ')
    sys.stdout.write('</body>')
    sys.stdout.write(' ')

def session_Contribute(sessionid):
    wigismo.set(l, 'i_2', 87)
    wigismo.output(sessionid, lambda: output_Welcome())
    state = wigismo.Store(sessionid)
    state.set('locals', l)
    state.set('start', 'session_Contribute_show_0')
    sys.exit(0)

def session_Contribute_show_0(sessionid):
    wigismo.output(sessionid, lambda: output_Pledge())
    state = wigismo.Store(sessionid)
    state.set('locals', l)
    state.set('start', 'session_Contribute_show_1')
    sys.exit(0)
    pass

def session_Contribute_show_1(sessionid):
    l['i_2'] = wigismo.get_field('contribution', int)
    g.set('amount_1', (g.get('amount_1', 0) + l.get('i_2', 0)))
    wigismo.output(sessionid, lambda: output_Total(total=g.get('amount_1', 0)), exit=True)
    sys.exit(0)
    pass

def wigismo_restart():
    state = wigismo.Store(wigismo.sessionid)
    l.update(state.get('locals'))
    globals()[state.get('start')](wigismo.sessionid)

sessions = ['Contribute']
for session in sessions:
    if wigismo.sessionid == session:
        globals()['session_%s' % session]('%s$%s' % (session, wigismo.random_string(20)))
        sys.exit(0)
    elif wigismo.sessionid.startswith('%s$' % session):
        wigismo_restart();
        sys.exit(0)

print 'Content-type: text/html'
print
print '<title>Illegal request</title>'
print '<p>You entered an invalid session name.</p>'
print '<p>Try one of these:</p>'
print '<ul>'
for session in sessions:
    print '<li><a href="?%s">%s</a></li>' % (session, session)
print '</ul>'
sys.exit(0)
```

## 6.6 Testing

The code generation phase was tested by trying to compile and run several services (past benchmarks, and services in the `public_html/wig/examples` directory). Notably, `wigismo` was able to successfully compile our benchmark `hangman.wig` service. We also compared its output to our manual construction of the `08rps.wig` service during the CGI vs WIG milestone. Many somewhat interesting issues surfaced while trying out various benchmarks:

- In python, blocks are marked by levels of indentation. A common pattern in WIG services (more so than in other languages, because WIG doesn't have arrays) is to have large if-else-if structures, like so:

  ```
  if (c1) {
      ...
  }
  else if (c2) {
      ...
  }
  else if (c3) {
      ...
  }
  ```

  There's nothing special about `else if`; in the AST, we just have an `if ... else` statement whose `else` part happens to be another `if` or `if ... else` statement. As such, we were generating code like this:

  ```
  if c1:
      ...
  else:
      if c2:
          ...
      else:
          if c3:
              ...
  ```

  The indentation level kept climbing. This might not seem like such a big deal, except that, as it turns out, python has a limit to the number of indented blocks one can have; this limit is hardcoded somewhere to be 100. Some services (notably, `2008/wig/group-4/phish.wig`) broke this limit, causing the generated code to raise an `IndentationError: too many indentation levels.`.

  The somewhat ugly fix to this was to explicitly check, when generating code for an `if ... else` statement, whether the else part was itself an `if` or `if ... else` statement, and if so writing `el` instead of `else:\n`, and setting a special flag so that the `else` part knows not to indent itself, resulting in code like:

  ```
  if c1:
      ...
  elif c2:
      ...
  elif c3:
      ...
  ```

- Blocks in python must contain at least one statement, otherwise an `IndentationError` error is raised. A snippet like this:

```
if (some_condition) {}
nextStatement();
```

generated code like this:

```
if some_condition:
nextStatement()
```

Less contrived examples involve reaching the end of a block after coming back from a `show` statement; this resulted in empty functions being generated. The (again, somewhat ugly) fix to this was to explicitly end every block with a `pass` statement, resulting for the above in code like this:

```
if some_condition:
    pass
nextStatement()
```

# 7 Availability and Group Dynamics

## 7.1 Manual

Running `wigismo` on the input WIG service (`example.wig`, say) generates a python file (`example.py`). This file has to be moved to whatever directory CGI scripts are served from, and its permissions have to be changed to `755`. The generated file depends on a python module we wrote called `wigismo`; there could be trouble when it tries to import it, depending on the environment. The simplest way to make it work is to copy the `wigismo.py` file into the `cgi-bin` directory, or into some directory on the python search path. Note that setting the `$PYTHONPATH` environment variable is not reliable; CGI scripts typically don't run under the same environment as the user. An ad-hoc solution would be to alter the search path (in the python file, after compiling), by adding this line just below `import sys`:

```
sys.path.insert(0, '/absolute/path/to/wigismo.py')
```

## 7.2 Demo Site

Demo services available at `http://www.cs.mcgill.ca/~ibadaw/wigismo/`.

## 7.3 Division of Group Duties

The majority of the work was done through pair programming; that is, both team members sitting on one computer, alternately typing. There are several advantages to this. For one thing, both members of the team work on every part of the compiler; this helps reduce any discrepancy in the experience gained, and it also makes oversights and such less common, since every piece of code has two pairs of eyes looking at it. Moreover, since we have "joint ownership" of the code, there is no blame to be assigned when something breaks. Having your partner next to you also makes you less likely to procrastinate.

Of course, this approach is not necessarily perfect. For instance, it requires that everything be worked on together – if one member is overly busy, no progress can be made. It's also hard to quantify – svn commits, for instance, may not accurately reflect who actually committed what (and there is no concept of a "joint commit" or anything like that). It can also take longer – in some cases, if the problem is well understood and it's just a matter of tedious typing, it would definitely be faster to have two programmers working independently.

# 8 Conclusions and Future Work

## 8.1 Conclusions

First, provide a brief summary of the report. Next, describe the main things that you learned in the course of this work, and attempt to draw new conclusions, even if they are just "soft" experience-based ones. Detail the things you learned that you did not expect to learn.

## 8.2 Future Work

The WIG language is very restrictive; it is practically impossible to write something worthwhile in it. In particular, WIG needs:

- Support for arrays, or indeed any useful data structure. A majority of the benchmarks end up using some large number of variables (or generating a whole bunch of code) to emulate this anyway. Depending on the target language, this might not even be so difficult to implement; one could think of transforming arrays in WIG into lists in python, for instance.
- Some way to access operating system facilities. Reading and writing files would be useful, for instance, or accessing databases, or making HTTP requests. Most general purpose languages provide some way to do this; it would just be a matter of generating the appropriate code.
- Better string manipulation. As it stands strings in WIG can't even be indexed; the only supported operation is concatenation. Strong support for string manipulation goes a long way towards making a language usable, and again, it would simply be a matter of delegating to the target language.

## 8.3 Course Improvements

On the whole, COMP 520 is a very fun and well structured course. The workload is nontrivial, but the project is split up into clear, well-defined milestones, and making the source for the JOOS compiler available provides ample guidance – learning by example really is quite effective. The only downside is the WIG language itself; it might have been an interesting concept back in the late 90s, but as it stands, it's awfully dated and obsolete. Writing anything of value with it is difficult. Modern web frameworks are both easier to use and more powerful. As such, completing the project is less satisfying than it could be, since it's likely `wigismo` will never be used again. Of course, the same could be said of any sort of coursework; these projects are undertaken mainly for their instructive value, and little else. With this concept, though – building a compiler for a DSL – there is such potential for building something useful, and it seems silly to have students write 4500-line CGI script generators instead!

(One simple idea would be to flip the course around; provide the source for a WIG compiler, and have students write a JOOS compiler. Generating code that can actually run on a JVM is certainly more interesting than CGI!)

## 8.4 Goodbye

Ismail thinks compilers are fun and interesting and is taking COMP 621 next semester, and applying for a Master's degree in this area. Carla's passions lie elsewhere.