Wig Report

COMP 520

Group F

Fall 2011

Andrew Bodzay (260329399)
Patrick Desmarais (260329253)
Iain Macdonald (260270134)

Presented to: Kamal Al-Marhubi

Introduction

Clarifications

The WIG language definition does not precisely define whether what we called gaps and input names should be unique inside **html constants**. Gaps were also referred to holes in the some milestone; they refer to variables enclose in those kinds of brackets: “<[“, “]>”. Input names are basically the ‘name’ attribute value present in each input html element. The following rules were applied for those two WIG elements:

- The type-checking phase throws an error on any html constant with two gaps of the same name, and accepts html constants with repeated input names.
- The symbol phase throws an error on any unmatched gap or input names respectively referred to in the plug or receive sub-clauses of any show or exit statement.

To simplify the tracking of gaps in the html constants, the symbol phase verifications and the writing of the CGI script, we chose to keep gaps unique inside each html constants. Since many WIG benchmark programs contained the same input names multiple times inside their html constants, we decided to allow their repetition.

The WIG language definition also does not precisely define type equivalence, and particularly, operations involving tuple constructs. Operations involving tuples include assignments, join, keep, and drop. The language definition does not specify which actions on tuples should be accepted based on their schema type. The equivalence between two tuples follows structural equivalence, without taking into account the order of fields inside those tuples. This type of equivalence allows the programmer to use the join, keep and drop operations to construct a new tuple and assign it to an existing tuple defined by some schema without worrying about the order of fields. However, two tuples with fields of the same name, but of different types, cannot be assigned to each other. There is a strict equivalence between simple types. The following definitions of the tuple operations were applied in the type-checking phase:

- The join operation operates by doing a union operation on fields, where the value of any field on the left tuple that is also found in the right tuple gets assigned the value on the right tuple’s field. Two tuples with fields of the same name, but different types throws an error.
- The keep operation removes every field except those found in the corresponding list of identifiers.
- Conversely to the keep operation, the drop operation removes every field in the corresponding list of identifiers.

In addition to put ourselves in the place of the wig programmer, the major decisions to clarify the WIG language took into account the easiness to translate to a CGI language such as Python, the one chosen for this project.

Restrictions

We restrict the use of show or exits in functions. The motivation for this is that searching for shows through a function within a session has the potential to become quite complicated, and managing the breaks in the program execution would be problematic if attempting to do this. This may have some implication on the way code will have to be written, as all the output to the browser must be done from within sessions themselves, possibly preventing a programmer from being able to reuse code for frequently used shows. While this is an unfortunate side effect, it shouldn't be impossible to work around, and should not prevent a programmer from being able to accomplish any tasks that he could have accomplished without this restriction.

The scanner does not handle gaps inside html tags. This issue was not caught in the scanparse week due to limited testing. After extending and automating tests with more benchmark programs, the syntax errors involving gaps in html tags were caught. Since these gaps may have had repercussions also on their possible presence at the place of input names, this issue was not resolved in the compiler. Since the majority of benchmarks worked, we decided to work on more important aspects of the compiler in later steps of the implementation.

Extensions

The WIG script shows a page to the user if no session was specified. This features makes WIG programmers life easier for testing and debugging. Hence, programmers can write their WIG program and try all sessions without specifying all sessions uniquely in the url of their browser. This idea was taken from another compiler names lkwig.

In addition to this debugging feature, the Python script handles the access to the global file and storing of global information differently than a normal open-write-close sequence of operation. A locking scheme is used to protect users from modifying the global information concurrently. This required additional Python lines of code in the output CGI script, which are added at the emit phase of the compiler.

Implementation Status

Every phase has been implemented, and the compiler emits a CGI script in Python. The compiler follows the WIG language definition, with the clarifications, restrictions and extensions mentioned above.

The code generation works for a reasonable number of WIG programs found in the benchmark list of the course. However, the code generation is not perfect. For example, there are still some minor issues to translating tuples into Python without overriding other variables values. Aside from small discrepancies, the tiny.wig program is an example of how the different phases of the project have been correctly implemented.

Parsing and Abstract Syntax Trees

The Grammar

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

htmls : html
      | htmls html

html : tCONST tHTML tIDENTIFIER '=' tHTMLTAG nehtmlbodies
tHTMLENDTAG ';'
      | tCONST tHTML tIDENTIFIER '=' tHTMLTAG tHTMLENDTAG ';'

nehtmlbodies : htmlbody
           | nehtmlbodies htmlbody

htmlbody : tag
      | gap
      | tWHATEVER
      | tMETA metacontents tMETAEND

metacontents : /* empty */
           | nemetacontents

nemetacontents : tMETACONTENT
                | nemetacontents tMETACONTENT

tag : '<' tIDENTIFIER attributes '>'
      | tHTMLSTARTENDTAG tIDENTIFIER attributes '>'
      | '<' tHTMLINPUT attributes '>'
      | tHTMLSTARTENDTAG tHTMLINPUT attributes '>'
      | '<' tHTMLSELECT attributes '>'
      | tHTMLSTARTENDTAG tHTMLSELECT attributes '>'

gap : tHTMLSTARTGAP tIDENTIFIER tHTMLENDGAP

attributes : /* empty */
           | neattributes

neattributes : attribute
           | neattributes attribute

attribute : tIDENTIFIER '=' attrvalue
            | '"' tSTRINGCONST '"' '=' attrvalue
            | tHTMLINPUTATTRNAME '=' attrvalue
```

```
            | tHTMLINPUTATTRTYPE '=' attrvalue
            | tHTMLINPUTATTRTYPETEXT '=' attrvalue

attrvalue : tIDENTIFIER
          | tINTCONST
          | '"' tINTCONST '"'
          | '"' tSTRINGCONST '"'
          | tHTMLINPUTATTRTYPETEXT
          | tHTMLINPUTATTRTYPERADIO       ;

schemas : /* empty */
     | neschemas

neschemas : schema
          | neschemas schema

schema : tSCHEMA tIDENTIFIER '{' fields '}'

fields : /* empty */
     | nefields

nefields : field
     | nefields field

field : simpletype idlist ';'

nevariables : variable
          | nevariables variable

variable : type idlist ';'

idlist : tIDENTIFIER
     | idlist ',' tIDENTIFIER

simpletype : tINT
          | tBOOL
          | tSTRING
          | tVOID

type : simpletype
     | tTUPLE tIDENTIFIER

functions : /* empty */
          | nefunctions

nefunctions : function
          | nefunctions function

function : type tIDENTIFIER '(' parameters ')' '{' statements '}'
```

```
parameters : /* empty */
           | neparameters

neparameters : parameter
           | neparameters ',' parameter

parameter : type tIDENTIFIER

sessions : session
      | sessions session

session : tSESSION tIDENTIFIER '(' ')' '{' statements '}'

statements : /* empty */
           | nestatements

nestatements : statement
           | nestatements statement

statement : simplestatement
           | variable
           | ifthenstatement
           | ifthenelsestatement
           | whilestatement

simplestatement : ';'
                | showstatement
                | '{' statements '}'
                | expressionstatement
                | returnstatement

showstatement : tSHOW document receive ';'
           | tSHOW document ';'
           | tEXIT document ';'


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

plugs : plug
      | plugs ',' plug

plug : tIDENTIFIER '=' expression

receive : tRECEIVE '[' inputs ']'

inputs : /* empty */
      | neinputs
```

```
neinputs : input
     | neinputs ',' input

input : tIDENTIFIER '=' tIDENTIFIER
     | tIDENTIFIER '.' tIDENTIFIER '=' tIDENTIFIER

ifthenstatement : tIF '(' expression ')' statement

ifthenelsestatement : tIF '(' expression ')' statementnoshortif tELSE
statement

statementnoshortif : simplestatement
                | ifthenelsestatementnoshortif
                | whilestatementnoshortif

ifthenelsestatementnoshortif : tIF '(' expression ')'
          statementnoshortif tELSE statementnoshortif

whilestatement : tWHILE '(' expression ')' statement

whilestatementnoshortif : tWHILE '(' expression ')' statementnoshortif


expressionstatement : statementexpression ';'

statementexpression : assignment
                | functioninvocation

returnstatement : tRETURN returnexpression ';'

returnexpression : /* empty */
                | expression

assignment : tIDENTIFIER '=' expression
           | tupleinvocation '=' expression

functioninvocation : tIDENTIFIER '(' arguments ')'

arguments : /* empty */
           | nearguments

nearguments : expression
           | nearguments ',' expression

expression : orexpression
           | assignment
```

```
orexpression : andexpression
           | orexpression tOR andexpression

andexpression : eqexpression
           | andexpression tAND eqexpression

eqexpression : relexpression
           | eqexpression tEQ relexpression
           | eqexpression tNEQ relexpression

relexpression : addexpression
           | relexpression '<' addexpression
           | relexpression '>' addexpression
           | relexpression tLTEQ addexpression
           | relexpression tGTEQ addexpression

addexpression : multexpression
           | addexpression '+' multexpression
           | addexpression '-' multexpression

multexpression : tupleopexpression
                | multexpression '*' tupleopexpression
                | multexpression '/' tupleopexpression
                | multexpression '%' tupleopexpression

tupleopexpression : unaryexpression
                | tupleopexpression tJOIN unaryexpression
                | tupleopexpression tKEEP tIDENTIFIER
                | tupleopexpression tKEEP '(' idlist ')'
                | tupleopexpression tDROP tIDENTIFIER
                | tupleopexpression tDROP '(' idlist ')'

unaryexpression : '-' unaryexpression
                | unaryexpressionnotminus

unaryexpressionnotminus : '!' unaryexpression
                      | tIDENTIFIER
                      | primaryexpression

primaryexpression : literal
                | '(' expression ')'
                | functioninvocation
                | tuplecreation
                | tupleinvocation

tuplecreation : tTUPLE '{' fieldvalues '}'

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

```
nefieldvalues : fieldvalue
           | nefieldvalues ',' fieldvalue

fieldvalue : tIDENTIFIER '=' expression

tupleinvocation : tuplee '.' tIDENTIFIER

tuplee : tIDENTIFIER
      | tuplecreation
      | functioninvocation

literal :tINTCONST
       | tTRUE
       | tFALSE
       | tSTRINGCONST
```

Using the flex tool

The flex implementation makes extensive use of the start conditions to handle several problems. In the case of html constants, some additional keywords needed to be caught inside html tags. For example, “select” and “input” as well as gaps constructs had to be taken separately from ‘whatever’ elements. For example, in particular cases, the double-quote and equal characters helped delimitate strings of interest in html tags, as well as the identification of every identifier present in html tags such as ‘input’ and ‘select’.

The start conditions helped create two main different types of grammar. The lexical analysis of html constants and their assigned strings of tags was differentiated from other WIG constructs involving like those in functions and sessions. Within the html markup itself, start conditions were also used upon the opening of an html tag to account for the identification of ‘input’, ‘select’ and other identifiers. Other start conditions for html include gaps and html comments to restrict their use according to the clarifications and restrictions mentioned in the previous section.

Html gaps and comments are not accepted inside html tags. Html gaps were later discovered to not be handled properly inside the lexer with a few benchmark programs. When putting comments in html tags on regular web pages, even Safari and Chrome browsers did not parse the html document ignoring those html comments present inside tags. Therefore, we let the tokens definition without the possibility of having those html comments or gaps inside tags, due to the complexity of adding another start condition inside an html tag. As we should have probably refactored the start conditions, the time for this phase of the compiler in the first week was not enough and we preferred to put more effort in later phases of the compiler.

We also used start conditions to perform comments which span multiple lines, starting

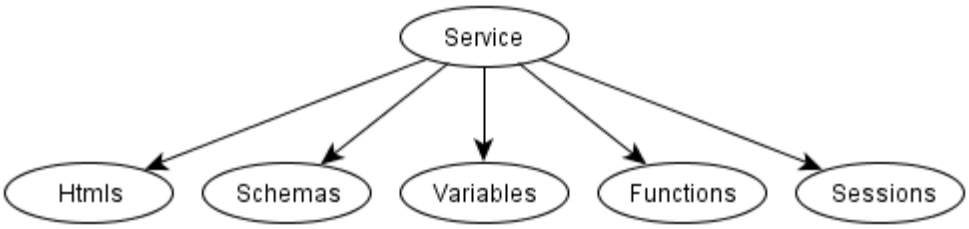
comment parsing on the reception of a '/*', and exiting comment mode upon the reception of a '*/'. These comments were simply swapped out from the compiled code; hence, start conditions allowed to not take into account any keyword encountered between those four characters. However, the number of lines needed to match up to report any compiler error, so it involves more than one lexical rule.

Using the Bison Tool

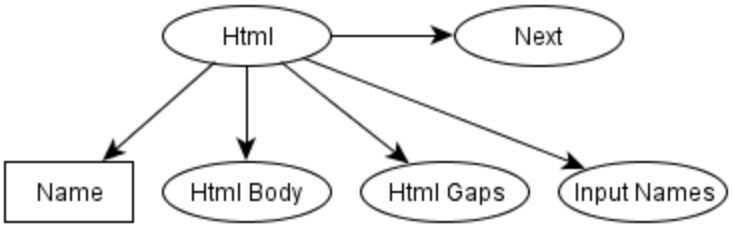
An interesting point in the bison implementation of the parser comes with tuple invocations. The compiler needed to account for the possibility of having a function call, an identifier or an anonymous tuple on the left hand side of those invocation expressions. In order to keep the grammar as concise as possible, we decided to accept a larger number of target for this invocation that would be checked later by the type-checker.

Another interesting aspect comes from the precedence of operations. We assumed that the precedence of regular operations of addition and multiplication held, but we needed to account for tuple operations as well. In this sense, we made tuple operations have a higher precedence than multiplication operations. To apply this precedence of operations, a similar template as the one defined for Java and JOOS was used.

Abstract Syntax Trees

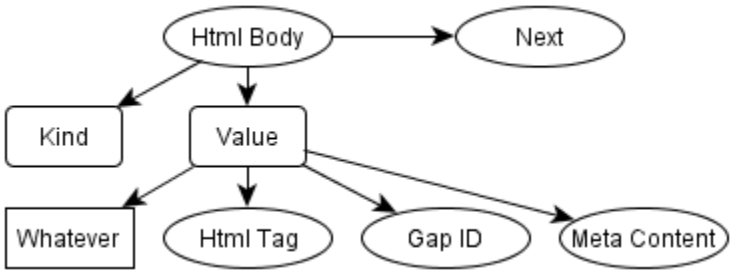


The service is divided into five components: htmls, schemas, variables, functions and sessions. The grammar is strict on the presence of the service and at least one html and one session. These last two components must be present or the parser will reject the WIG program.

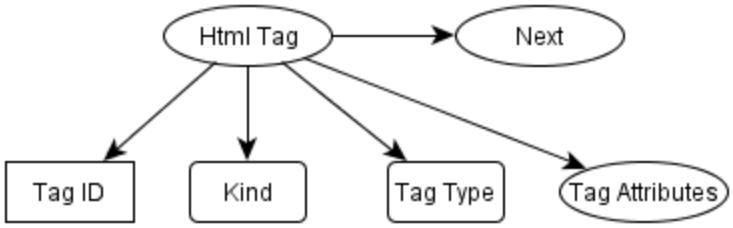


After parsing, the html node contains a pointer the the next html node in sequence, if any exists, as well as it's name and a node for the body of the html. During the symbol table phase, symbol nodes are added for the gaps in the html, as well as the names of any input tags.

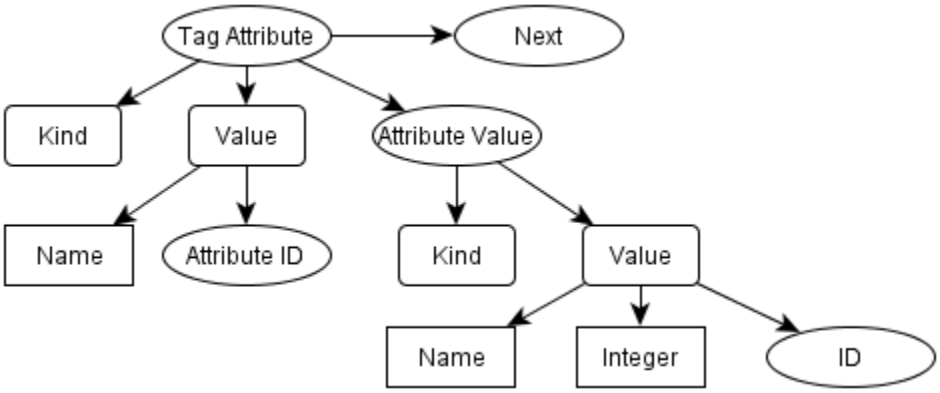
The reasoning behind the latter two is that such knowledge makes it easy to ensure that compiled code does not attempt to plug any invalid gaps or receive any invalid input.



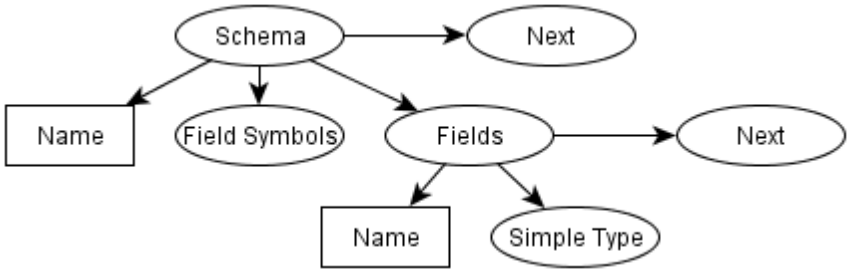
Within the html body node is a kind, which confers knowledge of the type of html body node that is being encountered. Many of the nodes use this type of identification to separate between different varieties of the same node. In this case, it is used to identify the type of html body node, and ensure we perform the correct operations on it. The actual contents of the node themselves are contained in the value field. The possible types of html body node are whatever, html tag, gap and meta content. A whatever node contains a string with the information associated with the node. An html tag body nodes and metacontent body nodes have pointers to their corresponding nodes. A gap node has a pointer to the identifier corresponding to the gap, and a metacontent node has a pointer to the metacontent node. Metacontent nodes are straightforward, containing a pointer to the next metacontent node as well as a string representing the metacontent itself.



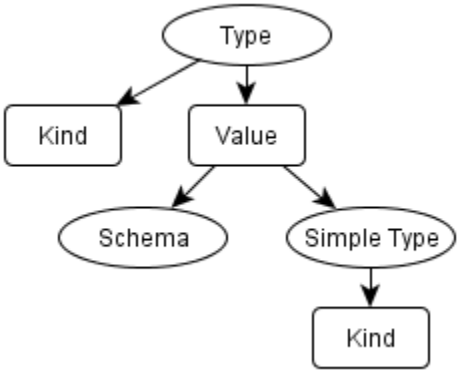
There are 3 different kinds of tags from our perspective, input tags, select tags, and all other tags. The reason we care about input and select tags in particular is that they contain information which could later be passed to a WIG program. The tag type for each tag corresponds to whether the tag is a start tag or an end tag. For input and select tags, the attributes are stored in a tag attribute node.



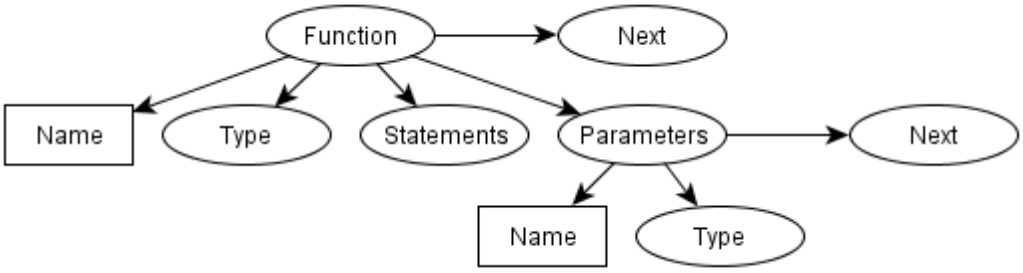
There are four types of tag attribute nodes, name attributes, type attributes, other attributes (string) and other attributes (ID). For the other attributes, their information is stored within either the name or attribute ID sections respectively. The value of an attribute is stored in an attribute value node. The value node is divided into five types, text, radio, other string values, other integer values, and other ID values, all of which have their information stored in their respective places.



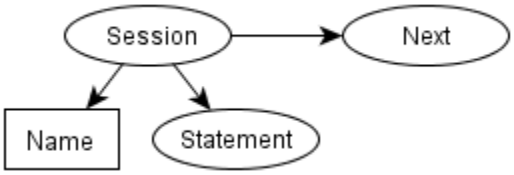
Schema nodes have a string containing their name, as well as a field node containing the schema's fields. The field node holds the name of the field as well as the simple type that should be stored in the field. The schema node also has a field symbol, which is used during the symbol and type checking phases.



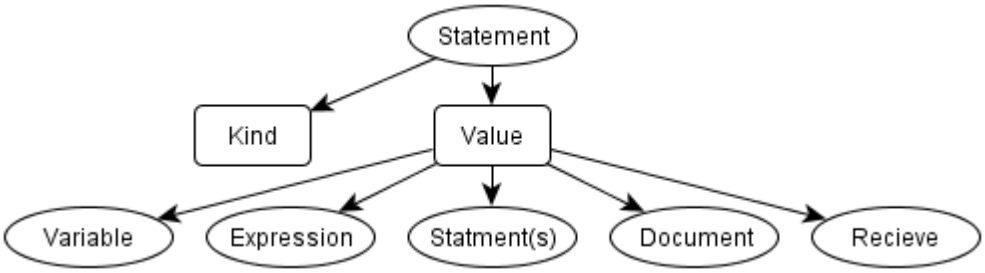
Type nodes come in two varieties. The first is those corresponding to simple types, which contain a pointer to a simple type node, which further specifies the type as being one of the simple boolean, integer or string types. The second type node variety is that which corresponds to a schema. In this case, the node contains the schema in question.



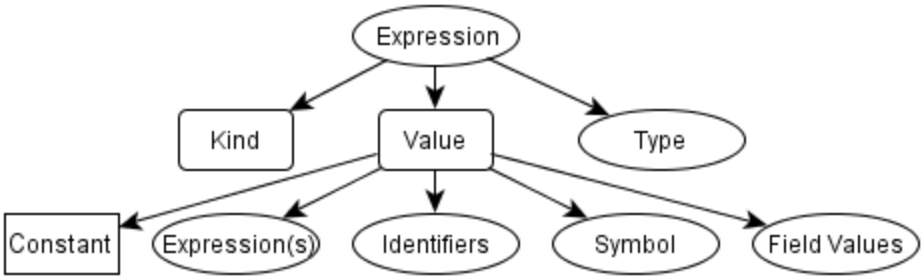
Function nodes have store the function name as a string, the return type within a type node, the function's statements within a statement node, and the function parameters within a parameter node. A parameter node contains the parameter's name and it's type.



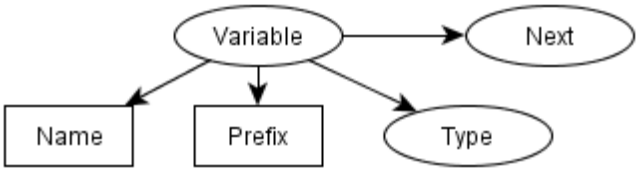
A session node simply holds the name of the session and the statements to be executed within the session.



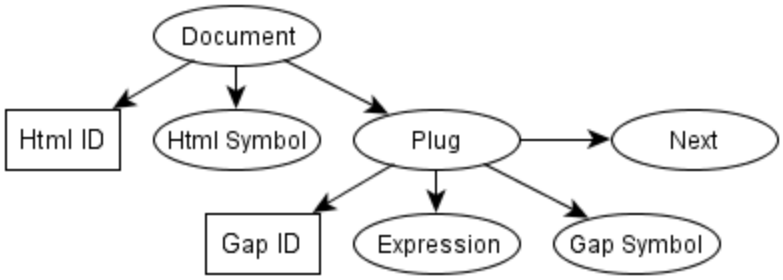
There are several kinds of statement nodes. A variable statement node contains a variable node. A block statement node contains another statement node for the statements within the block. A sequence statement node contains two statement nodes. Sequence statement nodes are used to have several statements in sequence instead of the next pointer that many of the other nodes in the AST have. Expression statement nodes have contains an expression node. Return statement nodes have an expression node for their return value. Show statement nodes have document nodes corresponding to the html document they show as well as a receive node for the inputs they are expected to receive. Exit statement nodes contain only a document node corresponding to the html to be shown on exit. If statement nodes contain an expression node for the if expression, and the statement to be executed if it evaluates to true. If-Else statement nodes contain the expression node to be resolved, as well as two statement nodes to be executed depending on the result. A while statement contains an expression which states the conditions for the loop, as well as a statement to be executed within the loop.



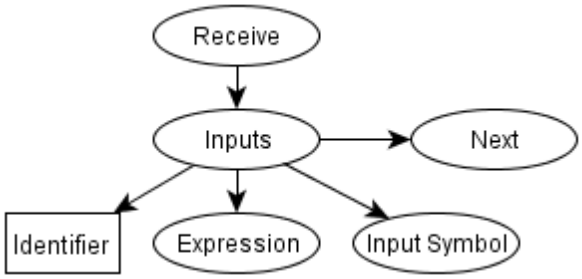
Like statement nodes, there are also a wide variety of expression nodes. Many of these types of nodes are binary expression nodes, such as multiplication, checking equality and logical operations. These binary expression nodes have as their value two expressions representing their operands. Unary expression nodes, such as that for logical not, have a single expression as their value, corresponding to the expression the operator is to be applied to. Tuple creation expressions contain a field value node, corresponding to the field values of the tuple. Tuple invocation expressions have an expression corresponding to the tuple to be invoked, as well as a string representing the identifier to invoke. Drop and keep expressions have two items, the first an expression representing the tuple the operation should be performed on, the second a list of identifiers corresponding to the items to be dropped or kept. Join operations have two expressions corresponding to the two tuples to be joined. The constant nodes for strings, integers, and booleans contain the value of the constant. Nodes for identifier expressions contain a string with their name, a string with a prefix created in the symbol table phase and used during the emission phase to ensure that scope rules are followed. It also contains a symbol created during the symbol table phase which in turn contains the variable declaration of the identifier in question, which is used during the type checking phase. Nodes for function calls have a string for their name, an argument node containing the various arguments, (essentially a list of expressions), and a symbol created during the symbol table phase which contains the function's declaration. The latter being used to check that the correct arguments are passed and that the return type is correct. In addition, all expression nodes also contain a type node which indicates the type of the expression, filled in during the type-checking phase of compilation.



Variable nodes contain the name of the variable being declared and a type node corresponding to it's type. There is also a prefix which is evaluated during the symbol table phase. This prefix is used during the emit phase to ensure that the compiler obeys the proper scope rules by ensuring that variables do not overlap with one another when they are not supposed to.



The document node, used for shows and exits, has within it a string identifier of the html being called, and a plug node for any plugs that are filled. It also contains an html symbol of the html it refers to, which is added during the symbol table phase. Each plug node has a string for the identifier to be plugged, an expression to plug with, and a gap symbol, which is filled out and used during the symbol table phase to ensure that the gap identifiers used by the plug actually exist in the html in question.



Receive nodes have input nodes within them, which detail the identifiers and expressions associated with the receive statement. There is also an input symbol used during the symbol table phase to ensure that the inputs listed can actually be received from the html in question.

In addition to these, we also have symbol nodes, which are created during the symbol table phase of compilation. All of these nodes contain either an html, a gap, an input tag name, a schema, a schema field, a function, a parameter, a session, or a variable. All such nodes also have a name, referring to the name of the item in question. These symbols are used to figure out what identifiers are referencing and to ensure that the same identifier is not used to mean more than one thing at the same time.

On top of the features mentioned, most nodes in the AST also have a line number, at which the node's content appears in the code to be compiled. This is used when reporting error messages during compilation, making said messages much more useful to the user.

Weeding

There are three parse trees which we weed. The first is any parse tree that contains an expression which is divided by a constant zero. The reason for this being that this would clearly interrupt normal progression of the code. This weeding is carried out by searching the parse tree for expressions, and upon the encounter of a division expression, we simply check that the

divisor is not zero, and if it is, an error is returned, informing the user of this division by zero.

The second parse tree which gets weeded is any tree which contains a session with no show or exit statement. Since the purpose of wig is to output forms to the user, a session without a show or exit should simply not exist. We disallow this by weeding parse trees containing sessions with no show or exit nodes. This is done by simply traversing the tree until we encounter a session node, and proceed to check all the nodes beneath it to ensure that at least one of those nodes is either a show or an exit.

Lastly, we also weed any parse tree containing a function that has a show or exit statement within it. The motivation for this is that it causes difficulty to have to halt the program midway through execution of a function. Doing so would require that we save the state of not only the locals in the function, but also save the location from which the function was called in the session or function that called it, as well as all the local variables in any functions that called them. This gets complicated very quickly, and would result in programs having to be split up into very inefficient parts, resulting in some very messy code at the end. Thus, we deemed it to be a simpler option to disallow it completely.

Testing

The scanparse phase was first tested by creating simple programs and using classic ones from the benchmark to see if the correct programs were accepted. Tested programs all passed. More test cases including those of html constants were lead to make sure that gaps and input names were correctly mapped to their specific AST constructs. The weed phase was tested by creating simple programs for each item that needed to be weeded. As the cases being weeded are not very complicated, only a few tests were needed. To test that no proper programs are wrongly rejected, the weeder was tested against the benchmark programs. In all tested cases, the scanner and parser, followed by the weeder, successfully reject programs which are incorrect for whatever reason, and accept programs that do not have any errors.

Symbol Tables

Scope Rules

The way scope is defined in our compiler is pretty straightforward, due to the simplicity of the Wig language. First, before any variable can be used, it must be declared. This declaration sticks with the variable throughout a given level of scope, and if another variable is declared using the same identifier within the same level, the program will not compile. When a new scope is entered, a program may create new declarations using identifiers which have already been used in higher levels, with the declaration in the closest scope being the one which will be referenced through the use of an identifier. A new level of scope can be entered in one of two ways, either by creating a block statement using braces, or by entering a function declaration. In the case of a block statement, all knowledge of variables from higher levels of scope is retained and can be modified as one would expect, however new declarations can be used to temporarily use an identifier to reference a different variable. When the scope is left, this

variable is forgotten, and the use of the identifier is restored to it's declaration in the higher scope. When using a function call, all declarations and variables from within the function or session from which the function was called are forgotten higher scopes are forgotten. Thus, a function has no access to the variables used by the session or function which invoked it in the first place.

One particularly contentious issue in the development of our symbol table was the use of schemas. It was debated whether the identifiers held within a tuple should be checked to match an existing schema when a declaration is encountered during the building of the symbol table, as well as whether identifiers should be checked for joining and keeping operations. It was decided that these operations amount to essentially checking the type of a given tuple, and therefore belong in the type-checking phase of the compilation.

Symbol Data

The symbol table stores information about functions, parameters, sessions, htmls, gaps, html input names, schemas, fields and variables as symbols. We are storing the name of each symbol, for unique identification within a scope, and for computing the hash of the symbol to map to a location in the symbol table, which is implemented as a hash table. This hash is computed as the shifted sum of the characters' ASCII codes. We also remember the symbol kind, which must be one of the symbol types defined above. These are stored in an enum structure within the symbol table. The final piece of information is a pointer from the symbol table entry to the location where the symbol is defined within the abstract syntax tree. This eased coordinated navigation of the abstract syntax tree and symbol table for future milestones involved in writing our compiler.

Algorithm

The symbol table construction implemented by our compiler is done in two phases. The first gathers information about the symbols defined, without verifying their usage, only checking that a symbol is not defined multiple times. This primary pass makes it possible to use functions within other functions, regardless of their order in the code. This phase does not delve into sessions or functions, and constructs the symbol table from scratch, filling entries with newly created structures. The second phase, which is referred to as the implementation phase, checks the usage of symbols, and raises and prints errors that it detects.

Both phases navigate the tree as we would expect. In the first phase, when an html is encountered, it's corresponding symbol is added to the symbol table. Whenever anything is added to the symbol table, we check to make sure that no other symbols at the same level share a name with the symbol being added. If they do, an error is returned. Within the html, we check for gaps and inputs, ensuring that there none of the names used for gaps overlap with those of other gaps, and likewise for input names. When a schema is encountered, it is added

to the symbol table, and we proceed to check it's fields. These fields are then checked against one another, to ensure that there are no two fields of the same name. Similarly, when functions and sessions are encountered, they are checked, and if their names don't conflict with anything, are added to the symbol table.

In the second phase, we begin to verify uses of variable names, checking within functions and sessions to ensure everything obeys scope rules, and that we don't have any undeclared identifiers. When a variable declaration is encountered, we check that it doesn't share a name with anything else within the same scope. We also check that the variable is either of a simple type or of a type belonging to a declared schema, returning an error otherwise. We also save a prefix to the declaration, using the symbol table level to help ensure emitted code follows scope rules. For functions, a fresh level of scope is added to the symbol table, and the parameters and statements of the function are checked. Sessions are similar, but there are no parameters to bother with. For parameters, we add them to the symbol table, ensuring that they are of an existing type (as was done for variables). For variable statements, we handle the variables as before. Block statements require we add a level of scope, and continue with the statements inside. For show exist show/exit statements we check that they refer to an existing html by searching upwards through the symbol table, returning an error if they does not. If there are plugs to be filled, these are searched for in the symbol table as well, ensuring that all gaps that the plug is attempting to fill exist, and that no gaps go unplugged, returning an error in either case. If we are receiving from some inputs, this is checked as well in the symbol table to ensure that all the inputs the code wants to receive exist in the html in question. For all other statements, we continue navigating the tree as expected. When an expression is encountered, if it is an identifier, we go upwards through the symbol table until we find the variable declaration or parameter for it, and add the corresponding symbol, as well as the determined prefix associated with the declaration to the expression node. For function calls, we likewise ensure that the function exists in the symbol table.

Testing

This phase was tested against several benchmarks. Automated tests were executed on a suite of Wig programs made available in the public_html folder of the course. By adding the generated output of our Wig programs involving any symbol error, tests were further executed to see if the corresponding output remained the same at the end of every compilation of the compiler.

Type Checking

The type checking phase of the compiler is responsible for ensuring that all expressions and statements in the program are type correct. That is, that the types of all assignments, function calls, and other uses of symbols respect a set of rules which govern the usage of these

symbols. These rules are mostly predetermined based on conventions adopted from other languages, and from syntactic and semantic rules in the WIG language. There are, however, a few cases for which we had to devise what typing rules to adopt, the most notable of these cases was the usage of schemas and tuples.

Types

There are 3 simple types supported by our language:

- Integers, which store an integer value
- Booleans, which store either true or false
- Strings, which store a string of characters

Our language also allows for a more complex schema and tuple type. Schemas and tuples in the WIG language are somewhat similar to structures in the c language. A schema is a definition of a type which, when instantiated is referred to as a tuple. Any schema consist of one or more fields, each of a primitive type (int, string, boolean). The biggest difference between WIG tuples and c structures is the join, keep, and drop operators, which permit modification in the code of the fields in a tuple to create a new tuple with a different set of fields.

Type Rules

- The right hand side of an assignment operator (=) must have the same type as the target for assignment on the left hand side.

- Operands to the boolean operators equal (==), not equal (!=) must have the same type. The result is a boolean.

- Operands to the boolean operators greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=) must both be integers and the result of this operation is a boolean.

- The operand to the boolean not operator (!) must be a boolean and the result is a boolean.

- The operand to the unary minus operator (-) must be an integer, and the result is an integer.

- Operands to addition (+), subtraction (-), division (/), multiplication (*), and modulo (%) operators must both be integers, and the result is an integer.

- Operands to the boolean and (&&), and or (||) operators must both be booleans, and the result is a boolean.

- Operation is a tuple, for which a schema must exist. If the resulting tuple from a join operation does not have a schema which matches that type, a type error is thrown.

- The left hand side operator to the keep operator (\+) must be a tuple. However, this tuple does not need to have a preexisting schema. The right hand side for this operator is a list of identifiers which must all be fields in the tuple passed on the left hand side. The result is a tuple of the type containing only the fields of the tuple on the left hand side with identifiers listed in the identifier list.

- The left hand side operator to the drop operator (\-) must be a tuple. However, this tuple does not need to have a preexisting schema. The right hand side for this operator is a list of identifiers which must all be fields in the tuple passed on the left hand side. The result is a tuple of the type containing only the fields of the tuple on the left hand side without identifiers listed in the identifier list.

- Integer constants have the integer type.

- The true expression has the boolean type. Similarly, the false expression has the boolean type.

- String constants have the string type.

- Declared tuples have the schema type which matches their declaration. If no such schema exists, the expression is considered type incorrect.

- Function return expressions must return an expression which has the same type as the function is declared to return. If the function is declared to return void, the return expression should simply be "return;".

- If expressions should switch on a boolean expression, and may contain anything within the block. I.e. if (<boolean exp>) stm [else stm].

- While expressions should also switch on a boolean expression, and may contain anything wichin the block. I.e. while (<boolean exp>) stm.

- The tuple de-referencing operator, "identifier.identifier" requires that the left hand side be a tuple, and the right hand side a field within that tuple. If this is not the case, a type error is raised.

- Function calls require that all arguemnts to the function match the arguments listed in the function definition, in the same order. That is, the arguments provided must match the required arguments in type and order exactly.

Type checking of tuples

There was some debate regarding how tuples should be defined. We originally had

decided to require all usage of tuples to have a matching schema. This meant that any instantiated tuple must have had matching schema, which would have had a few effects. Firstly, any variable which was to be created as a tuple must have had matching schema type. Secondly, all join, keep, and drop operations on an existing tuple must result in a new tuple which matches another schema. We decided later to drop this requirement, however, as it did not make our implementation any easier, and constrained the number of "correct" programs which would type check. Removing this restriction allows for a single statement to perform an operation which previously would have required several lines of code.

Algorithm

To type check, we simply navigate the tree until we encounter an expression that needs to be type checked. We then set the types from the bottom up, such that the lowest expressions, such as constants and identifiers, which already know their types, pass their type upwards to make sure it fits in with the expectations of the expression from in which it resides. If it does not, a type error is thrown. If it does match expectations, the type of the higher expression is set using the information contained within it, and type checking propagates upwards in this fashion.

We elected to test for structural equivalence in our tuple type comparisons. This structural equivalence test involves testing that the two tuples, or the tuple and the schema have the same set of fields, with the same names and the same types, without necessarily the same order. In other words, two tuples, or a tuple and a schema are identical if and only if the fields within the tuple have the same names and the same types within the tuple or schema. Testing equivalence of the other types available in WIG is much simpler, as the only other permissible types are the primitives int, string, and boolean. The equivalence test for these primitive types is simply if they are the same or not, casting between types is not permitted.

Testing

This phase was tested by creating several WIG test programs, many of which were created with unresolved type errors. Nearly 400 cases were created with the intent of provoking every error message possible in any conceivable way possible. To check the type correct programs were not rejected, a few simple, correct WIG programs were written and used along with milestones from previous years. In all tested cases where we expected the program to fail in the type checking phase, it did, and all type correct programs did not have issues in the type checking phase, so it's safe to say that the type checker works as expected.

Code Generation

Strategy

Our compiler emits code for the Python programming language. It relies on Python's

CGI libraries to perform the tasks outlined in the WIG specification. One of the main difficulties in translating WIG to Python is in translating an asynchronous user-interactive program into several synchronous, deterministic program executions. The main issues in this translation are the saving and restoring of variable state within the program, and halting execution at a particular point and resuming at the very next instruction.

The strategy we used to solve the asynchronous issue in WIG was to break the WIG program into atomic blocks, where each block defined a deterministic, non-reentrant piece of code. Much like the basic blocks seen in static analysis, these blocks guarantee that once the first statement has been executed, the only way to complete the block is by executing the last statement. Additionally, we split the program in such a way that all code which returns HTML to the user, or requests input from the user falls at the end of a function. The reason we did this is because Python has no goto statement, and dividing the WIG program into functions in this way was simpler than developing a large if-elif-else tree of execution. Furthermore, the restriction that show and exit statements may not be used in functions ensures that we may treat each WIG function as one Python function, and not worry about execution flow through complex execution trees.

To save and restore global variable state, we used a text file uniquely identified by the name of the program running. Thus, when a user access the WIG service, global data can be retrieved for this program. Additionally, each WIG program has its own set of data, and this data is shared across all sessions of the program, exactly as is specified in the WIG language.

Saving and restoring local variable state proved to be a bit more complex than for global state for a few reasons. First, each user must have their own local variables which only their WIG session has access to. Second, because we divided WIG functions into several different Python functions, we need to worry about sharing the WIG local variables among the Python functions. We implemented the saving and restoring of locals in much the same way as the globals, with a file identified by the name of the program, and the session id. We created Python global variables for all WIG local variables, so that these variables may be easily shared across the Python program. We do not need to worry about scope overwriting in this case, as this is taken care of by the symbol table, and any local which has the same name across multiple scope definitions should be re-initialized at the beginning of each scope.

Code Templates

One of the novel features of the WIG language is tuples. Supporting tuples in Python required a bit of work, as there is no clear type which should be used for tuples. We implemented WIG tuples as Python dictionaries, with the tuple variables mapping to Python dictionary keys, and the tuple values mapping to Python dictionary values. The join operation is thus as simple as joining both dictionaries element-by-element, using list iterators in Python. The keep operation is as simple as creating a new dictionary and only adding certain values from the original dictionary, and the drop operation is as simple as creating a new dictionary and omitting certain values from the original dictionary as the values are copied to the new dictionary. Using these templates, we can easily implement WIG tuples using Python.

Another complex translation involved changing the WIG HTML strings into Python HTML

strings. We used the Python string formatting operation, '%' and dictionaries to plug values into the HTML strings. This obeys our rule about only plugging when the exact parameters are matched. We had to print the HTML strings carefully from the abstract syntax tree, in such a way that the original text was not altered. This involved effectively walking through the tree created for HTML text, and printing out the text in the appropriate order, and also adding some background session handling code, such as embedding the session identifier in the HTML, for return with each request.

We were also careful in maintaining type information in stored global and local information. Were we to simply write out key, value pairs of variables to the file, it would be impossible to recover the type information from the file, for a '0' could be interpreted as a string "0," or the integer 0. To solve this problem, we emitted and read a three-tuple to and from the file, which contained the variable name, a type identifier: i for integer, s for string, b for boolean, and d for dictionary (or tuple), and the value of the variable. This enabled the save and restore functions to easily translate from Python datatypes to strings for writing to files, and back to Python datatypes.

Algorithm

The algorithm for generating Python code from the WIG program relies heavily on dividing WIG sessions into several different blocks, and interleaving the execution of these blocks in such a way that the Python program executes as the WIG program should. This algorithm works by traversing linear or branched execution flows, watching for shows and exits, and breaking into sub-functions whenever one is seen. In the special case of while loops, a different approach is taken to ensure the while loop executes appropriately as specified in the WIG language. While loops are split so that there is a function to evaluate the while loop criteria, and then functions to handle the interior of the loop, just as a linear or branched execution flow.

This Python sub-functioning system is then tied together using a dispatcher dictionary, which enables calling of functions using their string names, by indexing functions with their string names in the dictionary. This not only allows simple resuming of execution flow from a paused state, by maintaining a list of functions to be executed, but it also allows functions to be combined in ways which more accurately represent a non-linear WIG execution flow, in cases such as branching and looping.

Runtime System

The runtime system used by the generated code is the Python environment. We focus on Python 2.5, as that is the version available on linux.cs.mcgill.ca machines, which run Debian 5. We rely heavily on the CGI libraries available in Python to allow the program to function as a web interface program, rather than a simple Python script.

Sample Code

The following is the emitted code for tiny.wig:

```
#!/usr/bin/python

import sys
import cgi
import cgitb
import random
import string
import os
import fcntl

cgitb.enable()

# Helper variables and setup
session_key_str = 'session_key'
session_key = None
sessionid_str = 'sessionid'
sessionid = None
sessionid_length = 10
session_statefile = './tiny.wig_state/statefile'
global_statefilename = './tiny.wig_state/globals'
global_statefile = None
local_statefilename = './tiny.wig_state/'
local_statefile = None
local_breakpointsfilename = './tiny.wig_state/breakpoints'
session_exec = []
globals = {}
output_header_str = 'Content-type: text/html\n\n'
form = cgi.FieldStorage()
os.system('mkdir -p ./tiny.wig_state/')
logfile = open('./tiny.wig_state/log', 'a')


# Prints out 'page' with corr. 'gaps' if any
def show_page(page, gaps):
    gaps['url'] = 'http://' + str(os.getenv('HTTP_HOST')) +
str(os.getenv('SCRIPT_NAME'))
    gaps['session_key'] = session_key
    gaps['sessionid'] = sessionid

    print output_header_str
    print page % gaps
    return

# Returns the join of two dictionaries
def tuple_join(tup1,tup2):
    for k in tup1.keys():
        if tup2.has_key(k): continue
        else: tup2[k] = tup1[k]
    return tup2

# Returns the keep on a dictionary given a list
def tuple_keep(t,l):
```

```
    newt = dict()
    for k in l:
        newt[k] = t[k]
    return newt

# Returns the drop on a dictionary given a list
def tuple_drop(t,l):
    newt = dict()
    for k in t.keys():
        if k in l: continue
        else: newt[k] = t[k]
    return newt

# HTML
# For displaying error messages
Error = """<html><body>
Error: %(error_message)s
</body> </html>"""

# For displaying end of session messages
End = """<html><body>
<p>Your session has ended.
</body> </html>"""

# For displaying new services
raw_scriptname = 'http://' + str(os.getenv('HTTP_HOST')) +
str(os.getenv('SCRIPT_NAME'))
New = """<html><body>
No session specified. Please select a session:<br />
<a href=""" + raw_scriptname + """?
session_key=ses_Contribute>Contribute</a><br />
</body></html>"""

Welcome_4 = """
<html>
<form method="GET" action="%(url)s">
 <body>
    Welcome!
  </body>
<p><input type="submit" value="continue"/>
<input name="session_key" type="hidden" value="%(session_key)s"/>
<input name="sessionid" type="hidden" value="%(sessionid)s"/>
</form></html>
"""

Pledge_9 = """
<html>
<form method="GET" action="%(url)s">
 <body>
    How much do you want to contribute?
    <input name="contribution"  type="text"  size="4">
  </body>
```

```
<p><input type="submit" value="continue"/>
<input name="session_key" type="hidden" value="%(session_key)s"/>
<input name="sessionid" type="hidden" value="%(sessionid)s"/>
</form></html>
"""

Total_13 = """
<html>
<form method="GET" action="%(url)s">
 <body>
    The total is now %(total)s.
  </body>
<p><input type="submit" value="continue"/>
<input name="session_key" type="hidden" value="%(session_key)s"/>
<input name="sessionid" type="hidden" value="%(sessionid)s"/>
</form></html>
"""

globals['p1_amount'] = 0
globals['p2_i'] = 0

def storeglobalstate():
    global globals
    global global_statefile
    global_statefile.seek(0)
    global_statefile.write('p1_amount i ' + str(globals['p1_amount'])
+ '\n')
    global_statefile.truncate()
    fcntl.lockf(global_statefile, fcntl.LOCK_UN)
    global_statefile.close()

def storelocalstate():
    global globals
    global local_statefile
    local_statefile.seek(0)
    local_statefile.write('p2_i i ' + str(globals['p2_i']) + '\n')
    local_statefile.truncate()
    local_statefile.close()

def loadglobalstate():
    global globals
    global global_statefile
    global global_statefilename
    try:
        global_statefile = open(global_statefilename, 'r+')
        fcntl.lockf(global_statefile, fcntl.LOCK_EX)
    except IOError:
        global_statefile = open(global_statefilename, 'w')
        fcntl.lockf(global_statefile, fcntl.LOCK_EX)
        return
    for line in global_statefile.readlines():
        var = line.split(' ')[0]
```

```
        typ = line.split(' ')[1]
        val = line.split(' ')[2]
        if var == 'p1_amount':
            if typ == 'i':
                globals['p1_amount'] = int(val)
            elif typ == 's':
                globals['p1_amount'] = str(val)
            elif typ == 'b':
                globals['p1_amount'] = bool(val)
            elif typ == 'v':
                globals['p1_amount'] = int(val)
            elif typ == 'd':
                globals['p1_amount'] = dict(val)

def loadlocalstate():
    global local_statefile
    global local_statefilename
    global globals
    try:
        local_statefile  = open(local_statefilename, 'r+')
    except IOError:
        local_statefile = open(local_statefile, 'w')
        return
    for line in local_statefile.readlines():
        var = line.split(' ')[0]
        typ = line.split(' ')[1]
        val = line.split(' ')[2]
        if var == 'p2_i':
            if typ == 'i':
                globals['p2_i'] = int(val)
            elif typ == 's':
                globals['p2_i'] = str(val)
            elif typ == 'b':
                globals['p2_i'] = bool(val)
            elif typ == 'v':
                globals['p2_i'] = int(val)
            elif typ == 'd':
                globals['p2_i'] = dict(val)

def storebreakpoint(bkpt):
    global local_breakpointsfilename
    f = open(local_breakpointsfilename, 'w')
    f.write(str(bkpt))
    f.close()

def loadbreakpoint():
    global local_breakpointsfilename
    try:
        f = open(local_breakpointsfilename, 'r')
    except IOError:
        return None
    bkpt_raw = f.read()
```

```
bkpt =
bkpt_raw.replace('[','').replace(']','').replace('"','').replace(' ','
').split(',')
    f.close()
    if bkpt is not None:
        return bkpt[0]
    return bkpt

# Session Contribute
def fn_Contribute2():
    # restore state
    if form.has_key('contribution'):
        try:
            globals['p2_i'] = int(form['contribution'].value);
        except ValueError:
            globals['p2_i'] = 0;

    globals['p1_amount'] = (globals['p1_amount']+globals['p2_i'])
    storebreakpoint('End')
    storeglobalstate()
    storelocalstate()
    gaps = {
        'total': (globals['p1_amount']),
    };
    show_page(Total_13, gaps);

    return 0;

def fn_Contribute1():
    storebreakpoint('fn_Contribute2')
    storeglobalstate()
    storelocalstate()
    gaps = {
        };
    show_page(Pledge_9, gaps);

    return 0;

def fn_Contribute0():
    globals['p2_i'] = 0
    globals['p2_i'] = 87
    storebreakpoint('fn_Contribute1')
    storeglobalstate()
    storelocalstate()
    gaps = {
        };
    show_page(Welcome_4, gaps);

    return 0;

# Entry point for session Contribute.
```

```
def ses_Contribute():
    # initialize breakpoint
    breakpoint = loadbreakpoint()
    if breakpoint == None:
        breakpoint = 'fn_Contribute0'
    dispatcher[breakpoint]()
    return;

# Dispatcher
# Bookends for the dispatcher dictionary, so we don't have
# to be concerned about where commas go in the dict
def f():
    return None
def g():
    return None
dispatcher = {
'f' : f,
'fn_Contribute0' : fn_Contribute0,
'fn_Contribute1' : fn_Contribute1,
'fn_Contribute2' : fn_Contribute2,
'g' : g
}


session_dispatcher = {
'f' : f,
'ses_Contribute' : ses_Contribute,
'g' : g
}


def random_sessionid():
    str = ''
    for i in range(25):

str = str + random.choice(string.ascii_uppercase +
string.ascii_lowercase + string.digits)
    return str

# Everything starts here.
def main():
    global sessionid
    global session_key
    global local_statefile
    global local_breakpointsfilename

    sessionid = form.getvalue('sessionid');
    session_key = form.getvalue('session_key');

    if sessionid is None:
        sessionid = random_sessionid()
        bp = None
```

```
        local_statefile = './tiny.wig_state/statefile' + sessionid
        local_breakpointsfilename = './tiny.wig_state/breakpoints' +
sessionid
    else:
        local_statefile = './tiny.wig_state/statefile' + sessionid
        local_breakpointsfilename = './tiny.wig_state/breakpoints' +
sessionid
        bp = loadbreakpoint()

    loadlocalstate()
    loadglobalstate()

    if bp is None:
        if session_key is None:
            show_page(New,{})
        else:
            # dispatch to the session which must end with a show or
exit to close the global and local files.
            session_dispatcher[session_key]()
            return
    elif bp == 'End':
        show_page(End,{})
    else:
        # dispatch to the session which must end with a show or exit
to close the global and local files.
        session_dispatcher[session_key]()
        return
    storelocalstate()
    storeglobalstate()
    return

main()
```

Testing

We have performed a reasonable amount of testing on the emission module. It seems to work well on the several different benchmark programs that we tested. Our testing strategy was to generated code using our compiler, and using the lkwig compiler (which generates Perl code), and then compare the execution of the two from a web browser. If for the same inputs, the program gave the same output, the code generation was considered appropriate for the WIG function.

It is worth noting that our language implementation is somewhat restrictive in its type checking. Particularly, there are many benchmark programs from previous years which are rejected by the type system. This has to do with the way we elected to implement tuple operations, and the type checking of these operations.

Availability and Group Dynamics

Manual

Our WIG compiler, as found in wig/src of the subversion repository, has an associated Makefile. Using 'make,' one can compile the compiler. Then, using the ./wigc command, or by adding the working directory to the execution path, and using the wigc command, one can compile WIG programs into python scripts.

Without any options, the compiler will read in a WIG program, and generate a Python script of the same name, with the .py extension, rather than the .wig extensions. With the generated Python file, one need only copy this file to a working CGI binary directory under a public Apache webserver directory, such as on one's SOCS account, at ~/public_html/cgi_bin. Next, the mode must be set to 700 using "chmod 700 <program>.py," and finally, the service may be run by visiting the URL of the Apache CGI binary directory, appended with the program name and the .py extension. For SOCS applications, that is http://cs.mcgill.ca/~username/cgi-bin/programname.py

Demo Site

Some demonstration programs are available at http://cs.mcgill.ca/~imacdo9/cs520.html . On this webpage, there are links to seven different WIG programs, their WIGC compiled Python scripts, for execution, and their LKWIG compiled scripts, also for execution.

Division of Group Duties

The work for the compiler was divided somewhat haphazardly, with members simply informing the rest of the group if they were to start working on a portion of the project. The result is that all members of the group worked at least a little on each part of the compiler.

One issue that came up again and again over the course of the project was the issue of how to manage 3 people working on the same small file at the same time. As most of the milestones involved working only a handful of files, coordinating work on these items was a difficult task. In many cases, an intimate understanding of the code was necessary to complete milestone objectives properly, and gaining an intimate understanding of someone else's code has to potential to be a very time-consuming task. A somewhat simple workaround for this which would have been much more efficient would have been to split up the milestones from the beginning, having all 3 members working in parallel on different portions of the compiler. This would significantly reduce the amount of time and effort required, and would have streamlined the whole process. However, considering that the main purpose of creating project was to learn the inner workings of a compiler, it is tough to regret working in the way we did, as by the end of the project, all of the members in the group ended up with at least some experience in all of

the stages of the compilation process. So despite the inefficeicy of our method of working on the project, this method was very successful in that it allowed everyone to achieve the main goal of the course.

Conclusions and Future Work

Conclusions

In this course, we not only learned the fundamentals of compiler design and implementation, but also several other important lessons. We learned how to manage large programs, such as our 8kLOC compiler, written in c. We learned the importance of testing, particular in building a compiler, where the strangest test cases are the ones most likely to provoke error messages. We learned how to work well in a group, and identify and delegate tasks between three different people, and also how to communicate effectively, specifically speaking about managing a large project such as this one. We also learned how to use SVN, and the joys of using it as a revision control system.

Expanding upon these lessons a bit, one can easily see the benefits of using a scrum-based approach to engineering large software projects, much like the approach used in this project. Additionally, the importance of testing and test automation at a higher level, specifically in the interest of identifying errors at lower layers of software, before they become harder to fix at higher layers of abstraction. Without such organizational paradigms, our compiler would not have been as successful as it was.

Compilers are very interesting programming tools. Their understanding will prove quite useful in their future in our software engineering careers and allows us to have an idea how one would have implemented a compiler for certain languages. Additionally, it will give us insight into the way these computer languages really work, at the core, and allow us to program more efficiently in these languages.

Future Work

A simple addition which would have been wise to add in retrospect would be support for floating point numbers. This would not have been too difficult to implement, requiring similar support that was given for the other three simple types. The only thing that could be different would be allowing for integers to be automatically cast into floating points. This would have been straightforward in the type checking phase, as any time when we would normally accept a floating point, we could specify that an integer could be accepted as well.

Another good idea would have been to support for loops, as they are a useful piece of syntactic sugar that programmers seem to like. This would require a bit of desugaring, which would take place in the scanning and parsing phase. It would not be too complicated to convert a for loop into it's individual parts, the variable declaration, the while loop with it's condition, and an increment within the loop. In order to allow for a variable declaration directly above the while loop, the whole construct would have to be placed within a block statement.

Course Improvements

The week of the scanner and parser should be extended or started sooner. Once the language is defined in the Flex and Bison tools, testing all the benchmarks and adjustments takes a long time.