COMP-520 Final Report

Group 4

Jacob Beard <jbeard4@cs.mcgill.ca>
Wisam AlAbed <wisam.alabed@mail.mcgill.ca>

Abstract

Table of Contents

Introduction

Clarifications

Mandatory return in functions. Issue a warning. What happens if not all holes are plugged? Issue an error. What happens if plug specifies holes which do not exist? Issue an error. Initialization of variables before use. Contrary to many languages, examples indicate that variables may be intialized in WIG before use. We therefore initialize them to default variables.

Restrictions

We do not support complex set operations on tuples, like those demonstrated in tupleTorture.wig. The reason for this was simply that we ran out of time, and prioritized other difficult tasks, like show statements inside of functions, above this task. The implication of this restriction is that we are not able to handle tupleTorture.wig. However, all other basic examples work, as well as the 2009 benchmarks.

Extensions

We have not made any extensions to our version of WIG. The reason for this was that our primary goal was to successfully compile the existing basic wig examples, as well as the 2009 benchmarks. Our implementation matches these benchmarks, but does not implement additional language features.

Implementation Status

Our primary goal was to successfully scan, parse, weed, type check, and working code from the existing basic wig examples, as well as the 2009 benchmarks, and we have succeeded in this.

Automated tests have been written in JUnit to test scanning, parsing, weeding, and type check. We have manually verified the benchmarks for code generation.

Parsing And Abstract Syntax Trees

The Grammar

```
Package wig;

/****************************************
* this is almost fully taken from joos.sablecc
****************************************/
Helpers

 all =                   [0..0xffff];
 letter =                [['a'..'z'] + ['A'..'Z']];
 digit =                ['0'..'9'];
 letter_or_digit =      letter | digit;
 letter_or_digit_or_us = letter_or_digit | '_';
 letter_or_us =          letter | '_';
 nonzero_digit =        ['1'..'9'];
 octal_digit =          ['0'..'7'];
 lf =                   10;
 cr =      13;
 sp =                   32;
 ht =                   9;
 line_terminator =      lf;
 octal_escape =         '\' octal_digit octal_digit octal_digit;
 esc_sequence =         '\b' | '\t' | '\n' | '\f' | '\r' |
                        '\"' | '\' ''' | '\\' | octal_escape;

 ident =     letter_or_us letter_or_digit_or_us*;
 blank =     sp|ht|line_terminator;
 blank_set =    [sp+[ht+line_terminator]];

States
 wig,
 html,
 hole,
 starting_new_tag,
 in_input_tag,
  in_input_tag_attr_expr,
 in_input_tag_attr_rhs,
 in_input_tag_attr_rhs_value,
 in_input_tag_attr_rhs_no_quotes_value,
 in_input_tag_attr_rhs_hole_value,
 in_input_tag_attr_rhs_hole_rt_bracket,
 in_any_other_tag,
  in_any_other_tag_attr_expr,
 in_any_other_tag_attr_rhs,
 in_any_other_tag_attr_rhs_value,
 in_any_other_tag_attr_rhs_no_quotes_value,
 in_any_other_tag_attr_rhs_hole_value,
 in_any_other_tag_attr_rhs_hole_rt_bracket,
 in_end_tag;

Tokens

 eol_comment =          '//' [all - line_terminator]* line_terminator;

 /*******************************************************************
 * Blanks should work the same everywhere except in certain states
 * related to parsing attributes because a space may need to be
 * parsed as a delimiter if an attribute does not have quotes.
 *******************************************************************/
 {wig,
 html,
 hole,
 starting_new_tag,
 in_input_tag,
 in_any_other_tag,
 in_end_tag} blanks =                blank+;


 /*********************************************************************
 * Basic stuff in the top-level wig scope.
 *********************************************************************/


    {wig} schema = 'schema';
    {wig} session = 'session';
    {wig} show = 'show';
    {wig} exit = 'exit';
    {wig} return = 'return';
    {wig} if = 'if';
    {wig} else = 'else';
    {wig} while = 'while';
    {wig} plug = 'plug';
    {wig} receive = 'receive';
    {wig} int = 'int';
    {wig} bool = 'bool';
    {wig} string = 'string';
    {wig} void = 'void';
    {wig} tuple = 'tuple';
    {wig} true = 'true';
    {wig} false = 'false';

    {wig} l_brace = '{';
    {wig} r_brace = '}';
    {wig} assign = '=';
    {wig} semicolon = ';';

    {wig} lt = '<';
    {wig} gt = '>';
    {wig} l_par = '(';
    {wig} r_par = ')';
    {wig} l_bracket = '[';
    {wig} r_bracket = ']';
    {wig} comma = ',';
    {wig} keep = '\+';
    {wig} remove = '\-';
    {wig} join = '<<';
    {wig} eq = '==';
    {wig} neq = '!=';
    {wig} lteq = '<=';
    {wig} gteq = '>=';
    {wig} not = '!';
    {wig} minus = '-';
    {wig} plus = '+';
    {wig} mult = '*';
    {wig} div = '/';
    {wig} mod = '%';
    {wig} and = '&&';
    {wig} or = '||';
    {wig} dot = '.';

    {wig} service = 'service';
    {wig} const = 'const';
    {wig} html = 'html';

 {wig} identifier =            ident;
 {wig} intconst =              '0' | (nonzero_digit digit*);
 {wig} stringconst =           '"' ([all - '"']*  '\"'*)* '"';

 /*********************************************************************
 * HTML scope
 *********************************************************************/

    {wig->html} html_tag_start = '<html>';
    {html->wig} html_tag_end = '</html>';

    {html} html_text = [all -  [ '<' + '>' ]]*;


 /*********************************************************************
 * Holes
 *********************************************************************/
    {html->hole} lt_bracket = '<[';
    {hole} hole_identifier = ident;
    {hole->html} gt_bracket = ']>';

 /*********************************************************************
 * Tags
 * There are two distinct types: input and everything else
 * input parses attributes of the form name="value" and name=value
 * All other tags parse attributes as well.
 * Note also that input does not currently lex name or type attributes
 * as special cases, but this logic would be trivial to add if
 *   it is deemed useful.
 *********************************************************************/

 /*********************************************************************
 * Handle input attribute
 *********************************************************************/

  {starting_new_tag->in_input_tag} input_tag = 'input';

  {in_input_tag->in_input_tag_attr_expr} input_tag_attr_lhs = ident;

 {in_input_tag_attr_expr->in_input_tag_attr_rhs}
  input_tag_assign = blank_set* '=' blank_set*;

 {in_input_tag_attr_rhs->in_input_tag_attr_rhs_value}
  input_tag_attr_left_quote = '"';


  {in_input_tag_attr_rhs_value} input_tag_attr_rhs_value = [all - '"']*;
  {in_input_tag_attr_rhs_value->in_input_tag} input_tag_attr_right_quote = '"';

 {in_input_tag_attr_rhs->in_input_tag_attr_rhs_hole_value}
  input_tag_attr_rhs_hole_lt_bracket = '<[';

 {in_input_tag_attr_rhs_hole_value->in_input_tag_attr_rhs_hole_rt_bracket}
  input_tag_attr_rhs_hole_value  = ident;

 {in_input_tag_attr_rhs_hole_rt_bracket->in_input_tag}
  input_tag_attr_rhs_hole_gt_bracket = ']>';

 {in_input_tag_attr_rhs->in_input_tag_attr_rhs_no_quotes_value}
  input_tag_attr_rhs_no_quotes_value =
   [all-['"' + [ blank_set + [ '>' + '<' ]]]]*;


 {in_input_tag_attr_rhs_no_quotes_value->in_input_tag}
  input_tag_blank_delimiter = blank_set+;


 /*****************************************************************
 * Any other tags attribute
 *****************************************************************/


  {html->starting_new_tag} html_lt = '<';

  {starting_new_tag->in_any_other_tag} any_other_tag = ident;

 {in_any_other_tag->in_any_other_tag_attr_expr}
  any_other_tag_attr_lhs = ident;

 {in_any_other_tag_attr_expr->in_any_other_tag_attr_rhs}
  any_other_tag_assign = blank_set* '=' blank_set*;


 {in_any_other_tag_attr_rhs->in_any_other_tag_attr_rhs_hole_value}
  any_other_tag_attr_rhs_hole_lt_bracket = '<[';

 {in_any_other_tag_attr_rhs_hole_value->
  in_any_other_tag_attr_rhs_hole_rt_bracket}
   any_other_tag_attr_rhs_hole_value  = ident;

 {in_any_other_tag_attr_rhs_hole_rt_bracket->in_any_other_tag}
  any_other_tag_attr_rhs_hole_gt_bracket  = ']>';


 {in_any_other_tag_attr_rhs->in_any_other_tag_attr_rhs_value}
  any_other_tag_attr_left_quote = '"';

 {in_any_other_tag_attr_rhs_value}
   any_other_tag_attr_rhs_value = [all - '"']*;

 {in_any_other_tag_attr_rhs_value->in_any_other_tag}
```

```
  any_other_tag_attr_right_quote = '"';


 {in_any_other_tag_attr_rhs->in_any_other_tag_attr_rhs_no_quotes_value}
  any_other_tag_attr_rhs_no_quotes_value =
   [all-['"' + [ blank_set + [ '>' + '<' ]]]]*;

 {in_any_other_tag_attr_rhs_no_quotes_value->in_any_other_tag}
  any_other_tag_blank_delimiter = blank_set+;



 /*********************************************************************
 * Finish the start tag
 *********************************************************************/

  {in_any_other_tag->html,
   in_any_other_tag_attr_rhs->html,
   in_any_other_tag_attr_rhs_no_quotes_value->html,
   in_input_tag->html,
   in_input_tag_attr_rhs->html,
   in_input_tag_attr_rhs_no_quotes_value->html,
  in_input_tag_attr_expr->html} html_forward_slash_gt = '/>';

  {in_any_other_tag->html,
   in_any_other_tag_attr_rhs->html,
   in_any_other_tag_attr_rhs_no_quotes_value->html,
   in_any_other_tag_attr_expr->html,
   in_input_tag->html,
   in_input_tag_attr_rhs->html,
   in_input_tag_attr_rhs_no_quotes_value->html,
   in_input_tag_attr_expr->html} html_gt = '>';

 /*********************************************************************
 * Rules for close tags
 *********************************************************************/
  {html->in_end_tag} html_end_tag_lt_forward_slash = '</';
  {in_end_tag} html_end_tag = ident;
  {in_end_tag->html} html_end_tag_gt = '>';



/***************************************
* also inspired by joos.sablecc
***************************************/
Ignored Tokens
 blanks,
 eol_comment;

/***************************************
* inspired by original wig grammar
***************************************/
```

```
Productions
 service =
  T.service l_brace P.html* P.schema* variable*
   function* P.session* r_brace;

    html =
  const T.html identifier assign html_tag_start
   htmlbody* html_tag_end semicolon;

    htmlbody =
  {tag_start} html_lt any_other_tag
   any_other_tag_attr* html_gt |
  {tag_start_forward_slash_end} html_lt any_other_tag
   any_other_tag_attr* html_forward_slash_gt |
  {tag_end}   html_end_tag_lt_forward_slash
   html_end_tag html_end_tag_gt |
  {hole}      lt_bracket hole_identifier gt_bracket |
       {whatever}  html_text |
       //{meta}      meta |
       {input}     html_lt input_tag inputattr* html_gt |
  {input_start_forward_slash_end}     html_lt
   input_tag inputattr* html_forward_slash_gt  |
  {input_close_tag} html_lt input_tag
   inputattr* html_forward_slash_gt;

 any_other_tag_attr =
  {attr_with_quotes}      any_other_tag_attr_lhs
   any_other_tag_assign any_other_tag_attr_left_quote
   any_other_tag_attr_rhs_value? any_other_tag_attr_right_quote |
  {attr_without_quotes}   any_other_tag_attr_lhs
   any_other_tag_assign? any_other_tag_attr_rhs_no_quotes_value?
   any_other_tag_blank_delimiter* |
  {attr_with_hole}     any_other_tag_attr_lhs any_other_tag_assign?
   any_other_tag_attr_rhs_hole_lt_bracket any_other_tag_attr_rhs_hole_value
   any_other_tag_attr_rhs_hole_gt_bracket  ;

    inputattr =
  {attr_with_quotes}      input_tag_attr_lhs input_tag_assign
   input_tag_attr_left_quote input_tag_attr_rhs_value?
   input_tag_attr_right_quote |
  {attr_without_quotes}   input_tag_attr_lhs
   input_tag_assign? input_tag_attr_rhs_no_quotes_value?
   input_tag_blank_delimiter* |
  {attr_with_hole}     input_tag_attr_lhs input_tag_assign?
   input_tag_attr_rhs_hole_lt_bracket input_tag_attr_rhs_hole_value
   input_tag_attr_rhs_hole_gt_bracket  ;

    schema =
        T.schema identifier l_brace field* r_brace;

    field =
        simpletype identifier semicolon;

    variable =
        P.type identifiers semicolon;

    identifiers =
        {one}  identifier |
        {many} identifiers comma identifier;

    simpletype =
        {int}    int |
        {bool}   bool |
        {string} string |
        {void}   void;

    type =
        {simple} simpletype |
        {tuple}  tuple identifier;

    function =
        P.type identifier l_par arguments? r_par compoundstm;

    arguments =
        {one}  argument |
        {many} arguments comma argument;

    argument =
        P.type identifier;

    session =
        T.session identifier l_par r_par compoundstm;

    stm =
        {no}     semicolon |
        {show}   show document P.receive? semicolon |
        {exit}   exit document semicolon |
        {return} return semicolon |
        {retexp} return exp semicolon |
        {if}     if l_par exp r_par stm |
  {ifelse} if l_par exp r_par
   [then_stm]:stm_no_short_if else [else_stm]:stm |
        {while}  while l_par exp r_par stm |
        {comp}   compoundstm |
        {exp}    exp semicolon;

    stm_no_short_if =
        {no}     semicolon |
        {show}   show document P.receive? semicolon |
        {exit}   exit document semicolon |
        {return} return semicolon |
        {retexp} return exp semicolon |
  {ifelse} if l_par exp r_par
   [then_stm]:stm_no_short_if else [else_stm]:stm_no_short_if |
        {while}  while l_par exp r_par stm_no_short_if |
        {comp}   compoundstm |
        {exp}    exp semicolon;
    document =
        {id}   identifier |
        {plug} T.plug? identifier l_bracket plugs r_bracket;

    receive =
       T.receive l_bracket inputs  r_bracket;

    compoundstm =
        l_brace variable* stm* r_brace;

    plugs =
        {one}  P.plug |
        {many} P.plugs comma P.plug;

    plug =
        identifier assign exp;

    inputs =
        {one} P.input |
        {many} P.inputs comma P.input;

    input =
        lvalue assign identifier;

    exp =
        {assign}  lvalue assign or_exp |
  {assign_assign}  [first_lvalue]:lvalue
   [first_assign]:assign [second_lvalue]:lvalue
   [second_assign]:assign or_exp |
        {default} or_exp;

    or_exp =
        {or}      [left]:or_exp or [right]:and_exp |
        {default} and_exp;

    and_exp =
        {and}     [left]:and_exp and [right]:cmp_exp |
        {default} cmp_exp;

    cmp_exp =
        {eq}      [left]:add_exp eq [right]:add_exp |
        {neq}     [left]:add_exp neq [right]:add_exp |
        {lt}      [left]:add_exp lt [right]:add_exp |
        {gt}      [left]:add_exp gt [right]:add_exp |
        {lteq}    [left]:add_exp lteq [right]:add_exp |
        {gteq}    [left]:add_exp gteq [right]:add_exp |
        {default} add_exp;

    add_exp =
        {plus}    [left]:add_exp plus [right]:mult_exp |
        {minus}   [left]:add_exp minus [right]:mult_exp |
        {default} mult_exp;

    mult_exp =
        {mult}    [left]:mult_exp mult [right]:join_exp |
        {div}     [left]:mult_exp div [right]:join_exp |
        {mod}     [left]:mult_exp mod [right]:join_exp |
        {default} join_exp;

    join_exp =
        {join}    [left]:tuple_exp join [right]:join_exp |
        {default} tuple_exp;

    tuple_exp =
        {keep}        tuple_exp keep identifier |
        {remove}      tuple_exp remove identifier |
        {keep_many}   tuple_exp keep l_par identifiers r_par |
        {remove_many} tuple_exp remove l_par identifiers r_par |
        {default}     unary_exp;

    unary_exp =
        {not}     not base_exp |
        {neg}     minus base_exp |
        {default} base_exp;

    base_exp =
        {lvalue} lvalue |
        {call}   identifier l_par exps? r_par |
        {int}    intconst |
        {true}   true |
        {false}  false |
        {string} stringconst |
        {tuple}  tuple l_brace fieldvalues? r_brace |
        {paren}  l_par exp r_par;

    exps =
        {one}  exp |
        {many} exps comma exp;

    lvalue =
        {simple}    identifier |
        {qualified} [left]:identifier dot [right]:identifier;

    fieldvalues =
        {one}  fieldvalue |
        {many} fieldvalues comma fieldvalue;

    fieldvalue =
        identifier assign exp;
```

Using the flex or SableCC Tool

We implemented a scanner with multiple states and correct notion of lexical scoping. The most difficult part of developing the grammar was attempting to parse HTML. Correctly parsing HTML is known to be a hard problem in general. One reason for the creation of XHTML was to bring sane parsing rules to

HTML. It is for this reason that we were encountering parser exceptions right up until the last project, as we continually tested against new benchmarks.

In general, the problem of parsing HTML is difficult because of the many different forms that attributes and tags can take. any HTML tag may end with a ">" or "/>". It may or may not have a closign tag. Attributes may or may not have right-hand-side values. When they do have values, they may or may not be enclosed in quotes.

This can lead to interesting requirements for the lexer. On repercussion of this is in the way whitespace is handled. In our WIG grammar, whitespace is universally ignored, except when lexing attributes, due to that fact that because an attribute right-hand side value may or may not be enclosed in quotes, a space is needed to separate it from other attributes. It is thus necessary to tokenize whitespace while parsing attributes. This particular example motivated a heavily state-oriented approach to our WIG grammar.

Wig HTML constants add additional complexity by allowing attribute right-hand-side values to contain wig holes, as seen in the example poll.wig. Finally, input tags must be treated as distinct entities from other tags so as to later allow verification of receive parameters in show statements.

In order to handle the complexity of tokenizing and parsing html, we made liberal use of states (start conditions). This allowed us to define special tokenizing rules depending on whether we were in the top-level wig scope, or inside of an HTML statement, or inside of an attribute, or in its right-hand-side value.

Using the bison or SableCC Tool

We reused a lot of the code and productions from the original wig grammar provided on the website. One way our grammar differs from the original is that we allow for a service with nothing inside so `service{}` is accepted by our parser. We also modified the production rules for service, html, htmlBody and inputattr to account for lexical scoping introduced as part of our scanner. Finally, meta tags are ignored by the parser, and are filtered during the lexing stage.

Abstract Syntax Trees

We use the Java Feflection API to enable declarative filtering. We provide a global data structure which contains a set of tokens which we wish to filter. Only one method providing a definition of default behaviour is then needed in order to convert our CST to an AST. The AST structure will be identical to the CST, with all unnecessary tokens removed.

Figure 1. Declarative Definiton of Set of Tokens to Filter when Converting WIG CST to AST

```
private Set<Class<? extends Token>> classesToFilter =
 new HashSet<Class<? extends Token>>(){{
  add(TSemicolon.class);
  add(TService.class);
  add(TLBrace.class);
  add(TRBrace.class);
  add(TConst.class);
  add(THtml.class);
  add(TAssign.class);
  add(THtmlTagStart.class);
  add(THtmlTagEnd.class);
  add(THtmlLt.class);
  add(THtmlGt.class);
  add(THtmlEndTagLtForwardSlash.class);
  add(THtmlEndTagGt.class);
  add(TLtBracket.class);
  add(TGtBracket.class);
  add(TInputTag.class);
  add(TInputTagAssign.class);
  add(TInputTagAttrLeftQuote.class);
  add(TInputTagAttrRightQuote.class);
  add(TInputTagBlankDelimiter.class);
  add(TComma.class);
  add(TLPar.class);
  add(TRPar.class);
  add(TShow.class);
  add(TExit.class);
  add(TReturn.class);
  add(TIf.class);
  add(TElse.class);
  add(TWhile.class);
  add(TLBracket.class);
  add(TRBracket.class);
  add(TAssign.class);
}};
```

Figure 2. defaultOut Method using Java Reflection to Filter Unwanted Tokens Based on their Class

```
public void defaultOut(Node node) {
 Class nodeClass = node.getClass();
 for( Method m : nodeClass.getMethods()){
  Class returnType = m.getReturnType();

  if(returnType.equals(LinkedList.class)){
   ParameterizedType genericReturnType =
    (ParameterizedType) m.getGenericReturnType();
   Type[] types = genericReturnType.getActualTypeArguments();
   if(types.length > 0){
    Type type = types[0];
    ifListTypeIsFilteredRemoveNodes(m, type.getClass(), node);
   }
  }else{
   ifTypeIsFilteredRemoveNode(m, returnType, node);
  }

 }
}
```

Desugaring

We do not resolve any syntactic sugar during parsing.

Weeding

We weed unwanted parse trees using SableCC's provided depthFirstAdapter to traverse the tree and perform appropriate checks. We do this because we feel at this stage there are certain parse trees that will create trouble in later phases, so it is best to handle them at an early stage, issue errors and warning, and potentially modify the AST to a less problematic form.

We weed out programs that have division by zero and programs that have functions missing a return statement. We have tested them on the benchmarks weed.wig and weed2.wig found in wig/tests.

When our weeder encounters a problem it may either issue a warning by printing to stderror, or, if it deems the error fatal, it may throw a runtime exception. For division by zero, it will throw a runtime exception, and for missing return, it will issue a warning. Additionally, when encountering a missing return, the weeder will insert an AReturnStm into the AST, to allow our code generation algorithms to correctly function later on.

The weeder that filters functions without return statements uses two passes to resolve missing return statements.

Testing

We have set up automated unit test cases that run through basic wig examples, the 2009 benchmarks, and the factorial example, in order to test our implementation. Everything works and the test cases are all successful.

In addition, we have created two special test cases: weed.wig, which contains a division by zero error, and weed2.wig, which contains a function missing a return statement. These test cases are in the folder src/wig/tests of our project. Both test cases succeed.

We also have a unit test which tests to make sure that `pretty(parse(pretty(parse(X)))) = pretty(parse(X))`.

Symbol Tables

We used the strategies discussed in class and in the course notes to develop our symbol table. This means that we used a cactus stack which was woven together with our AST, which is to say that, the nodes in our AST were mapped to nodes in the cactus stack, and the nodes in the cactus stack contained symbol tables, whose entries mapped to nodes in the AST.

Scope Rules

Variables may be declared inside of services and compound statements. Compound statements may be used directly inside of sessions, and because a compound statement is also a statement, it may be used inside of other statements.

Sessions and compound statements both create new scopes. Scope rules are as follows: two variables with the same name may not be declared inside the same scope. However, two variables with the same name may be declared if they are in different scopes. If the scopes are hierarchically nested, then we would say that the variable in the inner scope is *shadowing* the variable in the outer scope.

Additionally, we chose to allow both schema and const html entities define new scopes. For const html, each hole is considered to be a scoped variable name of type Hole, and for the schema, each entity is considered to be a scoped entity of type int, string, or bool. Constructing a scope for each of these entities greatly simplified analysis later.

Symbol Data

We use three primary data structures: an AST, a Symbol Table, and a Cactus Stack Symbol Tree, which is a tree of nodes containing Symbol Tables. They are woven together in the following manner: Each AST node which defines a scope (a service, or compound statement), is mapped to a node in the Cactus Stack. The node in the Cactus stack contains a Symbol Table, which maps variable names to their types, and to their location in the original AST.

Algorithm

Constructing the CactusStackSymbolTree

We use only one pass to construct the CactusStackSymbolTree. In this pass, two main things happen. First, each time an entity is visited that defines a scope (a service, a compound statement, a schema, or a const html), that entity creates a new GenericTreeNode containing a new SymbolTable, and pushes it on the CactusStackSymbolTree. In this way, entities that create scope are able to construct a data structure that captures the notion of nested scope, in the manner discussed in class.

A "current" scope is also kept through the duration of the pass. Each time an entity creates a new scope, the "current" scope is set to the new scope. When exiting the scoped entity, the scope is popped from the CactusStackSymbolTree. To retrieve the current scope, one simple needs to call `peek()` on the CactusStackSymbolTree.

The second important thing that occurs during this pass is that all entities are placed in the SymbolTable instance associated with the current scope.

Performing Analysis

After the Cactus Stack Symbol Tree data structure had been populated, analysis was performed, and the weaving of the AST nodes to SymbolTables and back was often essential. The most complex example of this was the checking of tuples against their associated schemas when referencing a particular property. In order to perform this analysis, for a given AQualifiedLvalue, the name of the tuple would be retrieved. Next, the CactusStackSymbolTree would be traversed vertically, searching each scope until the tuple's declaration is found in some scope's symbol table. The reference in the symbol table would contain a backwards link which would then be used to retrieve the AST node. This node would then be used to retrieve the schema name. A second vertical traversal of the CactusStackSymbolTree would then be performed until the schema declaration would be found in a SymbolTable. This table would again contain a backwards link which would return a Node in the AST. Finally, the fields would be consulted to find one that matches the field name which the original tuple had been attempting to reference. This particular analysis was implemented in the class `TupleFieldExistenceCheckerDFA`.

Based upon the above example, one may also understand the reasoning behind keeping fields to be kept in their own nested scope, created when visiting the schema. Otherwise, a variable could be declared in the same scope as the schema declaration, and would then have direct access to the schema's declared fields, which would not adequately capture the desired behaviour.

This same method also used to check the use of plugs with holes in html const entities.

Forms of Analysis Performed

We were at this stage able to perform various forms of code analysis using the symbol table, without explicitly attempting to perform any type checking. At this stage, we checked for the following exceptions:

- Repeat declaration of same variable in global and local scope.
- Repeat declaration of same formal parameter.
- Repeat declaration of same method.
- Collisions between formal parameters and local variable declaration.
- Call to a method that does not exist.
- Not all the holes are plugged in const html.
- Not all plugs exist in const html.
- Variable never declared.
- Accessing properties on tuples that are not declared in the associated schema

We perform multiple passes on the AST in order to build up our symbol table and perform these analyses:

- first pass works on the global scope it builds up the symbol table
- second pass checks to see if all identifiers have been declared
- third pass checks to see if all plugs have corresponding holes
- fourth pass checks to see if all holes have been plugged

- fifth pass checks to see that all tuples do not access properties that are not declared in their associated schemas

In each of these, a subclass of java.lang.RuntimeException is thrown. An informative error message is printed with the line number and position of the token that caused the exception.

Testing

We have set up automated unit test cases that run through basic wig examples, the 2009 benchmarks, and the factorial example, in order to test our implementation. Everything works and the test cases are all successful.

In addition, we have special tests written to test each of the above possible exceptions with simple representative examples. These tests are encapsulated in the JUnit test class `TestASTToSymbolTableConverter`. This these examples include tests for both success and for failure.

Type Checking

Our type checking uses almost exclusively the strategies, algorithms, and data structures that were discussed in class and in the course notes. We also used the Joosc compiler SableCC source code as inspiration.

Major Data Structures

The main data structure was the Hashtable that mapped nodes to types. We chose to implement this using a regular Java Hashtable<Node, Token>. The keys of the Hashtable were Nodes in our ast, and the values were Token types representing types(TBool, TString, TVoid, TTuple, TInt)

The second major data structure was the Hashtable that mapped method names to a list of formal parameters. We chose to implement this using a regular Java Hashtable<String, List<PArgument>>. The keys of the hashtable were string identifiers representing method names, and the values were a list of formal parameters of that method.

The third major data structure was the Hashtable that mapped method names to return types. We chose to implement this using a regular Java Hashtable<String, Token>. The keys of the Hashtable were string identifiers representing method names, and the values the return type of that method.

The fourth major data structure was the Hashtable that mapped Schema names to a list of Strings. We chose to implement this using a regular Java Hashtable<String, List<String>>. The keys of the Hashtable were string identifiers representing schema names, and the values were a list of Strings representing the tuple variables that where defined to be of that schema type.

Types

We have determined that the types that are relevant for WIG programs are the following: Int, Bool, String, Void, Tuple.

Type Rules

1. A Function must return a value whose type is equivalent to the type of the function as it was declared.

   a. if a function is declared to be of type Void it must return void

   b. if a function is declared to be of type int it must return an int

   c. if a function is declared to be of type String it must return a string

   d. if a function is declared to be of type bool it must return a bool

   e. if a function is declared to be of type tuple it must return a tuple

2. The expression(Condition) of an if statement or if/else statement must be of type bool.

3. In an assignment expression the type of the left-hand side value(lValue) must correspond to the type of the right-hand side(rValue)

   a. if an lvalue is of type int the rValue must be of type int

   b. if an lvalue is of type string the rValue must be of type string

   c. if an lvalue is of type bool the rValue must be of type bool

   d. if an lvalue is of type tuple the rValue must be of type tuple

4. The left hand side and right hand side of an OR expression must both be of type bool. The resulting evaluation of the OR expression yields an expression of type bool.

5. The left hand side and right hand side of an AND expression must both be of type bool. The resulting evaluation of the AND expression yields an expression of type bool.

6. The left hand side and right hand side of an EQUALS(==) expression must both be of the same type. The resulting evaluation of the EQUALS expression yields an expression of type bool.

7. The left hand side and right hand side of a NOTEQUALS(!=) expression must both be of the same type. The resulting evaluation of the NOTEQUALS expression yields an expression of type bool.

8. The left hand side and right hand side of a GREATERTHAN(>) expression must both be of type int. The resulting evaluation of the GREATERTHAN expression yields an expression of type bool.

9. The left hand side and right hand side of a GREATERTHANEQUALS(>=) expression must both be of type int. The resulting evaluation of the GREATERTHANEQUALS expression yields an expression of type bool.

10. The left hand side and right hand side of a LESSTHAN expression must both be of type int. The resulting evaluation of the LESSTHAN expression yields an expression of type bool.

11. The left hand side and right hand side of a LESSTHANEQUAL expression must both be of type int. The resulting evaluation of the LESSTHANEQUALS expression yields an expression of type bool.

12. The left hand side and right hand side of a DIVISION(/) expression must both be of the type int. The resulting evaluation of the DIVISION expression yields an expression of type int.

13. The left hand side and right hand side of a MOD(%) expression must both be of the type int. The resulting evaluation of the MOD expression yields an expression of type int.

14. The left hand side and right hand side of a MULTIPLICATION(*) expression must both be of the type int. The resulting evaluation of the MULTIPLICATION expression yields an expression of type int.

15. The left hand side and right hand side of a SUBTRACTION(-) expression must both be of the type int. The resulting evaluation of the SUBTRACTION expression yields an expression of type int.

16. The left hand side and right hand side of an ADDITION(+) expression must both be of the type int for integer addition to occur. The resulting evaluation of the ADDITION expression yields an expression of type int. If one of the sides is a string then string concatenation will occur and the resulting evaluation of the

17. The base expression side of an UNARYMINUS(-) expression must be of the type int. The resulting evaluation of the UNARYMINUS expression yields an expression of type int.

18. The base expression side of a LOGICALNOT(!) expression must both be of the type bool. The resulting evaluation of the LOGICALNOT expression yields an expression of type bool.

19. The left hand side and right hand side of a JOIN(>>) expression must both be of the type tuple. The resulting evaluation of the JOIN expression yields an expression of type tuple.

20. The left hand side of a KEEP(+/) expression must be of the type tuple. The resulting evaluation of the KEEP expression yields an expression of type tuple.

21. The left hand side of a REMOVE(-/) expression must be of the type tuple. The resulting evaluation of the REMOVE expression yields an expression of type tuple.

22. The type of a string constant base expression is a string.

23. The type of a int constant base expression is an int.

24. The type of a false constant expression is a bool.

25. The type of a true constant expression is a bool.

26. During a method call the arguments list is checked against the formal parameters of the method We check to make sure that the correct number of arguments is passed to the method We check to make sure that the corresponding types of each argument matches with the type of the formal parameter

Algorithm

Describe your algorithm for checking type rules.

The algorithm uses one pass on the ast using SableCC's depthfirstadapter. We keep a hashtable that maps nodes to types(our types in this case are represented as Token objects either(TInt, TString, Tbool, TVoid, or TTuple). We use the symbol table when needed to lookup types of id's. This is relevant when we hit a simpleLValue node and need to store the type of the id in our hashtable. We then apply the type rules mentioned up and do appropriate checks when he hit each rule in our walk of the AST.

Our implementation reuses similar strategies to those employed when performing analysis in the Symbol Table milestone. So, for example, in the Symbol Table milestone, when checking tuples against their associated schemas in order to check that a particular referenced property has actually been implemented in its schema declaration, one would traverse the CactusStackSymbolTree vertically, searching each scope until the tuple's declaration is found in some scope's symbol table. The reference in the symbol table would then contain a backwards link that would be used to retrieve the AST node. This node would then used to retrieve the schema name. A second vertical traversal of the CactusStackSymbolTree would then performed until the schema declaration is found in a SymbolTable. This table would then contains a backwards link to the AST, which would return the Node in the AST. Finally, the fields are consulted to find one that matches the field name the original tuple was attempting to reference.

Likewise, in the Type Checker, when attempting to check a tuple field's type, a similar retrieval mechanism would be employed. One would traverse the CactusStackSymbolTree vertically, find the associated schema declaration, follow the backwards link to retrieve the AST node of the tuple declaration, which would be used to fetch the schema name. A second vertical traversal of the CactusStackSymbolTree would performed to to retrieve the schema declaration, which would contain a backwards link to the AST, which would allow one to retrieve the individual fields in the Schema node's scope symbol table. The symbol table could then be consulted together with the fields' identifiers to to retrieve type information. The method of traversal is the same as in the Symbol Table example; it is primarily the operation that changes.

Testing

We have tested this phase by including unit tests that test for every possible type error that occurs and ensures an exception is thrown by our type checker. Our type checker works for everything except tuples. We only handle very simple cases of tuples related to qualified values. Tuples are handled very primitively the only thing we handle for tuples are qualified lvalues for example given a tuple 't1' we will do type checking for 't1.a = 10' to check that the field 'a' of the tuple 't1' is of type INT

We do not handle checking the right hand side of a keep and/or remove tuple operations We do not handle initializing a tuple, for example `t1 = { a = 10, b = true, s = "hello" }`. We do not handle checking to see if the join of two tuples makes sense semantically

for some of the exceptions thrown I am not able to precisely say at what line and what character position the error occurs because our AST is strips all the tokens like parenthesis and equals signs ect which we would have used to get line and character positions.

Code Generation

Strategy

Early on in the project, we identified that the most fundamental challenge presented by the WIG language was that posed by the `show` statement. `Show` displays information to the user and potentially reads information back. In the WIG language, it is used in a synchronous fashion, just like a `printf` or `scanf` in the C language. However, to actually achieve this effect with a web application involves handling multiple requests: first an HTTP GET to display the web page, and then an HTTP POST to receive the user data. On the server, this involves two separate function invocations (`doGet` followed by `doPost`). When the first function exits, and the user is presented with a form, the state of all local variables are lost, and `show` may be followed by statements that further depend on the state of these local variables. Thus, the challenge presented by the WIG language is to suspend execution when it encounters a `show` during an HTTP GET, and to resume execution upon receiving user data via an HTTP POST, while still preserving local variable state state. The WIG language thus abstracts out the asynchronous nature of the HTTP protocol, and presents the developer with a straightforward synchronous API for interacting with the user.

Our strategy went through two iterations. We first thought to use Continuations, which allow one to do precisely what is described above: suspend execution at an arbitrary point, and then resume execution at that point with all local variables and even the call stack intact. Another way to state this is that a continuation allows one to save the local variable state as well as the call stack, and return at any time to the frame at which the continuation was taken. We chose to target the Mozilla Rhino implementation of the JavaScript language, as it supports Continuations. Unfortunately, this turned out to be technically infeasible, as Continuations may only be used by Rhino in its interpreted mode, and it was necessary to compile our scripts to Java bytecode so that they could be deployed to a servlet container.

Rather than use Continuations, we instead elected to transform our WIG program to the "continuation-passing style". In this style of programming, the program never returns, but instead always passes around a function closure representing the next step of execution.

In our transformation, when we wish to suspend execution, we return a function closure representing the next step of execution, and store it in a variable. Closures capture the state of all local variables visible in the closure's scope. The saved closure is then invoked when receiving a request from a user, which effectively allows us to resume the thread of execution with the state of all local variables intact. An example of this will be described below.

Code Templates

We did not use templates, but instead programmed against a JavaScript DOM, using the JSDT library. We therefore present representative transformations here. Because JavaScript has a basic C-style syntax, most expressions in JavaScript are the same as in WIG, and so are excluded here.

Session

Figure 3. WIG Session

```
session Contribute() {
   //statements inside the session
}
```

Figure 4. JavaScript Session

```
var $Contribute_continuation = null;

function $Contribute($http) {
    if (!$Contribute_continuation) {
        $Contribute_continuation = $get_new_Contribute_continuation($http);
    }
    $Contribute_continuation = $Contribute_continuation($http);
}

function $get_new_Contribute_continuation() {
    return function ($http) {
    //statements inside the session
    };
}
```

Show

Figure 5. WIG Show

```
show plug Again[correction = "lower"] receive[guess = guess];
//statements after the show
```

Figure 6. JavaScript Show

```
$show($http, $Again, {
 $correction: "lower"
});
return function ($http) {
 //statements after the show
};
```

if statement

Figure 7. WIG if statement

```
if (condition) {
 //statements inside the if
}
//statements after the if
```

Figure 8. JavaScript if statement

```
function $if_2($http) {
 if (condition) {
  //statements inside the if
  return $FSAT_if_2($http);  //injected at innermost scope
 } else {
     return $FSAT_if_2($http);
 }
}

function $FSAT_if_2($http) {
 //statements after the if
}

return $if_2($http);
```

(explain mandatory else block)

ifelse Statement

Figure 9. WIG ifelse Statement

```
if (condition){
 //statements inside the if
}
else{
 //statements inside the else
}
//statements after the if/else block
```

Figure 10. JavaScript ifelse Statement

```
function $if_1($http) {
 if (condition) {
  //statements inside the if
  return $FSAT_if_1($http);  //injected at innermost scope
 } else {
  //statements inside the else
  return $FSAT_if_1($http);  //injected at innermost scope
 }
}

function $FSAT_if_1($http) {
  //statements after the if/else block
}

return $if_1($http);
```

While Statement

Figure 11. WIG While Statement

```
while (condition){
 //statements inside the while
}
//statements after the while
```

Figure 12. JavaScript While Statement

```
function $while_1($http) {
 if (condition) {
  //statements inside the while
  return $while_1($http);  //injected at innermost scope
 } else {
  return $FSAT_while_1($http);
 }
}

function $FSAT_while_1($http) {
 //statements after the while
}
return $while_1($http);
```

Functions

Declaration

Figure 13. WIG Function Declaration

```
int calculate(int pin1, int pin2){
 //function statements

 return(return_value);
}
```

Figure 14. JavaScript Function Declaration

```
function $calculate( $http, $continuation, $pin1, $pin2){
 //function statements

 $continuation($http,$return_value);
}
```

Invocation

Figure 15. WIG Function Invocation

```
calculate(ppin1, ppin2);
//statements after function invocation
```

Figure 16. JavaScript Function Invocation

```
function $calculate_1($http, $return_value_calculate_1) {
 //statements after function invocation
}
return $calculate($http, $calculate_1, $ppin1, $ppin2);
```

Algorithms

We used two primary algorithms for implementing the above transformations: one to handle statements, and one to handle function invocation expressions inside statements.

Statements

Critical to our method of code generation is the concept of "innermost scope", which we define to be the point to which the compiler will eventually append code representing the continuation, or the "next" step in execution. The continuation may be encoded as a function call to an outside function, or may simply be additional statements which are inlined.

Each statement, when visited, is associated with zero or more innermost scopes. Some statements, like `if`, `ifelse`, `while`, and `show` are guaranteed to have an innermost scope, because their transformation always requires the creation of a function closure. Other statements may or may not have an innermost scope, depending on the way in which they are used. So, for example, a compound statement will have an innermost scope if one of the statements it contains has an innermost scope. An expression statement and return expression statement will have an inner scope if it contains a function invocation expression. Finally, an empty statement and return statement will never have an innermost scope.

The continuation is not appended to the innermost scope when the node is initially visited, but is instead appended later, when a compound statement is visited. The reason for this is that SableCC does not provide generic methods for traversing trees, and, specifically, there is not a straightforward method of getting the sibling of a node. Given these constraints, we deemed that it would be simpler to defer the construction of the hierarchy of nested scopes to occur when we visit a compound statement. The algorithm we then apply is as follows, which we present here in pseudocode:

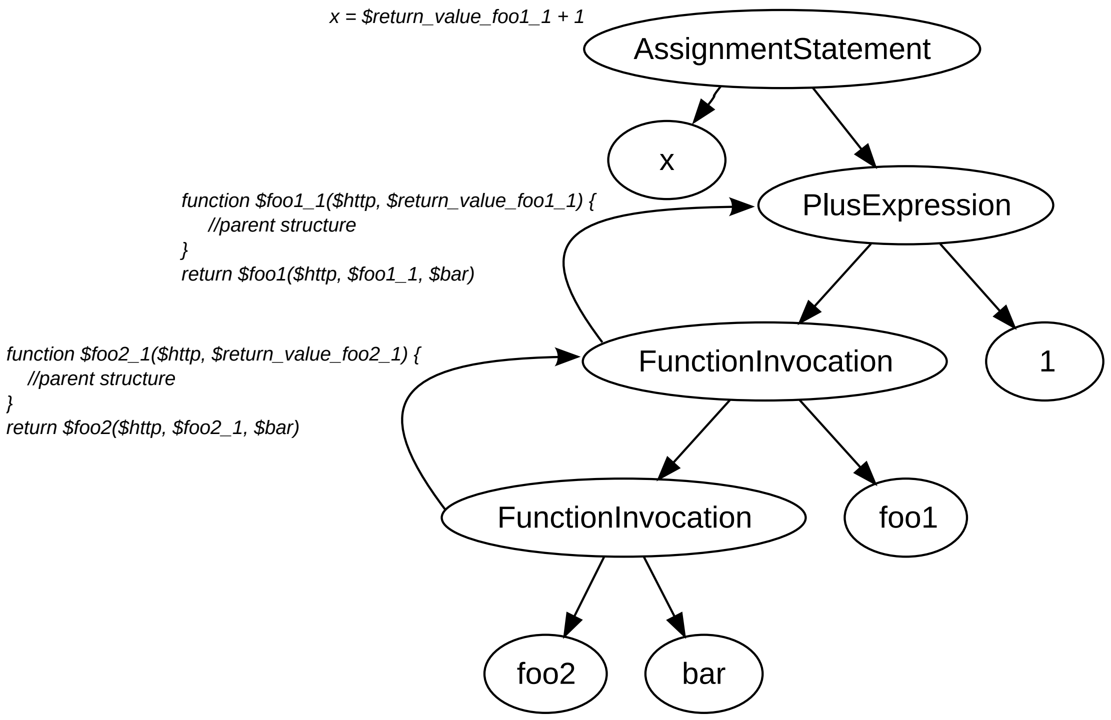
Figure 17. Compound Statement Generation Algorithm

```
List of Statements outACompoundstm(ACompoundstm node):

    toReturn = new empty List of statements

    FunctionDeclaration innermost = null;

    lastStatements = new empty List of statements

    stmList = get list of statements from node

    for each stm in reverse(stmList):

            FunctionDeclaration funDec = innermost scope of stm;

            if funDec is not null:
                if innermost is null:
                    innermost = funDec

                if lastStatements is not null:
                    append all statements in lastStatements to funDec
                    clear lastStatements

            add stm to lastStatements

    for each stm in reverse(lastStatements):
        toReturn.add(lastNode)

    set this nodes innermost scope to innermost

    return toReturn
```

Function Invocation Expressions

Dealing with function invocations, such that `show` statements could be included inside of functions, required a sophisticated algorithm. The above algorithm for statements allowed one to produce a code template associated with an innermost scope, and then to create the structure of hierarchical nested scopes when a compound statement was visited. Generating code for function invocations within statements was a very different problem. Where the recursive structure of a statement was defined in terms of its siblings, the recursive structure of a function invocation would be defined in terms of its parents. Consider the following examples:

Figure 18. Parse Tree for Expression `x=foo1(foo2(bar))+1`


x = $return_value_foo1_1 + 1
AssignmentStatement
x
PlusExpression
function $foo1_1($http, $return_value_foo1_1) {
//parent structure
}
return $foo1($http, $foo1_1, $bar)
function $foo2_1($http, $return_value_foo2_1) {
//parent structure
}
return $foo2($http, $foo2_1, $bar)
FunctionInvocation
1
FunctionInvocation
foo1
foo2
bar

In general, our strategy was to structure of our generated code for function expressions such that it would be flipped "inside-out", with code generated for foo2 enclosing code generated for foo1. This is to say that code generated for a node for expressions was defined recursively in terms of its parents, and not its children. This structuring would respect the expected order of execution, as foo2 would be evaluated before foo1. The return value for foo2 would be passed as argument foo2_return_value to continuation function foo2_1, and the return value for foo1 would be passed as argument foo1_return_value to continuation function foo1_1. Finally the original statement, with variable names foo2_return_value and foo1_return_value replacing the original function invocations, would be appended to the the innermost scope of the function created, in this case foo1_1.

Some global data structures were needed to handle the transformation:

Figure 19. Function Invocation Code Generation Data Structures

```
Stack<Node> statementStack = new Stack<Node>();

 CallGlobalStructures{
 boolean hasBeenInCallExp;
 List<ASTNode> callExpRecursiveStructure;
 FunctionDeclaration transientCallExpInnermostScope;
}

Map<Node,CallGlobalStructures> nodeToCallGlobalStructuresMap =
 new HashMap<Node,CallGlobalStructures>();
```

First, we maintain a stack of statements. Each statement is mapped to a `CallGlobalStructures` object via the `nodeToCallGlobalStructuresMap`. Finally, the CallGlobalStructures class contains:

- `hasBeenInCallExp`, a boolean that records whether a callexp had been visited in that statement the hierarchical structure of call expressions
- `callExpRecursiveStructure`, the top-level nodes of the nested structure created by visiting function invocation expressions
- `transientCallExpInnermostScope`, the innermost scope of the hierarchical structure of call expressions

The reason why we have a CallGlobalStructures object mapped to each statement node, rather than just maintain a single global CallGlobalStructures object which gets re-initialized on entering a statement, is that statements may contain other statements. Thus, a stack of nodes is needed in order to know in which scope a function call expression is being evaluated, and therefore which CallGlobalStructures object to manipulate.

Here is a pseudcode description of our algorithm:

Function Invocation Code Generation Algorithm

```
defaultIn(Node node):
 if node is a statement:
  statementStack.push(node);
  nodeToCallGlobalStructuresMap.put(node,new CallGlobalStructures())

defaultOut(Node node):
 if node is a statement:
  statementStack.pop()

outACallBaseExp(ACallBaseExp node):
 CallGlobalStructures structures =
  nodeToCallGlobalStructuresMap.get(this.statementStack.peek())

 structures.setHasBeenInCallExp(true)

 List<ASTNode> toReturn = new ArrayList<ASTNode>()

 //e.g. foo
 String functionName = node.getIdentifier().getText()

 //e.g. foo_1
 String nextStatementLabelName =
  getFunctionInvocationLabelName(functionName)

 //e.g. return_value_foo_1
 String nextStatementReturnValueLabelName =
  getNextStatementReturnValueLabel(nextStatementLabelName)

 //e.g. "return foo(http,foo_1);"
 returnStatement =
  createCallExpReturnStatement(functionName,
          nextStatementLabelName,
          node.params())
 //e.g. "function foo_1(http,return_value_foo_1){}"
 nextStatementFunctionDeclaration =
  createCallExpFunctionDeclaration(nextStatementLabelName,
  nextStatementReturnValueLabelName)


 if structures.getTransientCallExpInnermostScope() is null:
  //put the statements at the top level of the structure
  structures.getCallExpRecursiveStructure()
   .add(nextStatementFunctionDeclaration);
  structures.getCallExpRecursiveStructure().add(returnStatement);
 else:
  //inject the statements in the innermost scope
  List statementList = structures.getTransientCallExpInnermostScope();
  statementList.add(nextStatementFunctionDeclaration);
  statementList.add(returnStatement);

 //the newly created nextStatementFunctionDeclaration
 //will always be the new innermost scope
 structures.setTransientCallExpInnermostScope(
  nextStatementFunctionDeclaration);

 //Return the name of the next statement return value.
 //Doing so substitutes the variable for the function
 //invocation in the resulting statement.
 return createSafeSimpleName(nextStatementReturnValueLabelName);
```

Here are some further examples of how this plays out with respect to the algorithm for transforming statements.

Figure 20. Example Transformation of Expressions with If Statements, WIG 1

```
if(foo1(bar)){
 x=1;
}
```

Figure 21. Example Transformation of Expressions with If Statements, JavaScript 1

```
function $foo1_1($http, $return_value_foo1_1) {
 function $if_1($http) {
  if ($return_value_foo1_1) {
   $x = 1;
   return $FSAT_if_1($http);
  } else {
   return $FSAT_if_1($http);
  }
 }

 function $FSAT_if_1($http) {}
 return $if_1($http);
}

return $foo1($http, $foo1_1, $bar);
```

Figure 22. Example Transformation of Expressions with If Statements, WIG 2

```
if(condition){
 x=foo2(bar);
}
```

Figure 23. Example Transformation of Expressions with If Statements, JavaScript 2

```
function $if_1($http) {
 if ($condition) {
  function $foo2_1($http, $return_value_foo2_1) {
   $x = $return_value_foo2_1 + 1;
   return $FSAT_if_1($http);
  }
  return $foo2($http, $foo2_1, $bar);
 } else {
  return $FSAT_if_1($http);
 }
}

function $FSAT_if_1($http) {}

return $if_1($http);
```

Figure 24. Example Transformation of Expressions with If Statements, WIG 3

```
if(foo1(bar1){
 x=foo2(bar2) + 1;
}
```

Figure 25. Example Transformation of Expressions with If Statements, JavaScript 3

```
function $foo1_1($http, $return_value_foo1_1) {
 function $if_1($http) {
  if ($return_value_foo1_1) {
   function $foo2_2($http, $return_value_foo2_2) {
    $x = $return_value_foo2_2 + 1;
    return $FSAT_if_1($http);
   }
   return $foo2($http, $foo2_2, $bar);
  } else {
   return $FSAT_if_1($http);
  }
 }


 function $FSAT_if_1($http) {}
 return $if_1($http);
}
return $foo1($http, $foo1_1, $bar);
```

Figure 26. Example Transformation of Expressions with Show Statements, WIG

```
show plug Doc[hole = foo(bar)];
```

Figure 27. Example Transformation of Expressions with Show Statements, JavaScript

```
function $foo2_1($http, $return_value_foo2_1) {
 $show($http, $Document, {
  $hole: $return_value_foo2_1
 });
 return function ($http) {};
}
return $foo2($http, $foo2_1, $bar);
```

Miscellaneous

Naming Collisions

We implemented a simple strategy to avoid naming collision when converting between WIG and JavaScript identifiers. A strategy was necessary because some names are legal in wig, but reserved in JS. For example, some benchmarks have used "number" as a variable name, and another one used "new", both of which are reserved words in JavaScript.

Our strategy was simply to prefix a "$" to all WIG identifiers used in JavaScript. "$" can legally be included in an identifier in JavaScript (a fact that many JavaScript toolkits, including Mootools and JQuery take advantage of to provide short names for core functions), but is illegal in the default wig grammar. This allowed us to easily avoid naming collisions between our source and target domains.

Variable Shadowing

JavaScript provides excellent native support for variable shadowing, so no special strategy was required to implement this.

Runtime System

We targeted Java Servlets, using the Rhino JSC compiler. The final compiled form is a standard WAR file that may be deployed to a servlet container, such as Tomcat.

Our entire project is built with Ant, which includes tasks to parse, weed, and check one or several WIG files, generate JavaScript code from WIG, compile the JavaScript code to Java bytecode using JSC, package all compiled JavaScript classes into a WAR file, and then deploy the WAR file to a running Tomcat server. Unfortunately, we did not have enough time to add automated testing of the generated code, and instead chose to rely on manual testing. However, it would have been feasible to build a small testing framework to emulate the servlet environment, and then write tests against the generated JavaScript code.

Sample Code

Figure 28. Generated output of tiny.wig

```
loadClass("Constants");
var $Welcome = "<html> <body> \n    Welcome!\n   </body> </html>";
var $Pledge = "<html> <body> \n    How much do you want to contribute?\n\
 <input name=\"contribution\"type=\"text\"size=4 > </body> </html>";
var $Total = "<html> <body> \n    \
 The total is now <[$total]>.\n   </body> </html>";
var $amount = 0;
var $Contribute_continuation = null;


function $Contribute($http) {
    if (!$Contribute_continuation) {
        $Contribute_continuation = $get_new_Contribute_continuation($http);
    }
    $Contribute_continuation = $Contribute_continuation($http);
}


function $get_new_Contribute_continuation() {
    return function ($http) {
        var $i = 0;
        $i = 87;
        $show($http, $Welcome);
        return function ($http) {
            $show($http, $Pledge);
            return function ($http) {
                $i = $receive($http, "contribution");
                $amount = $amount + $i;
                $exit($http, $Total, {
                    $total: $amount
                });
            };
        };
    };
}
```

Limitations

Our compiled code does not handle multiple concurrent users. If there are multiple users accessing a single session, then they will make use of a single session continuation, and thus "step on" one another's session. This was a deliberate decision on our part with respect to WIG semantics. However, it would have been easy to add by leveraging the Java servlet HttpSession API.

We also do not support complex operations on tuples, as presented in the tupleTorture.wig example.

Testing

We tested our compiler against all 2009 benchmarks, the WIG classic examples, and factorial.wig. Everything works, except for tupleTorture.wig, as we do not support operations on tuples beyond instantiation and field dereferencing. Also, todo.wig manifests unexpected behaviour, in that we are not able to add todo messages. At this stage, it is unclear why this behaviour manifests in this way.

Availability and Group Dynamics

Manual

After checking the project out of SVN, it is set up to build all of the basic WIG examples, the 2009 benchmarks, and the recursion test, and deploy it to a Tomcat servlet running at localhost:8081, with username `test`, and password `test`. Simply check out and run `ant install`. Each benchmark will then be exposed at http://localhost:8081/wig_servlet/<wig_file_name_without_wig_extension>. For example, poll.wig will be accessible at URL http://localhost:8081/wig_servlet/poll.

To install the to a different Tomcat server, simply change the properties `manager.url`, `manager.username`, and `manager.password` to reflect those of your Tomcat server.

To use a servlet container that is not Tomcat, simply build the WAR file and install it manually. To build the project's WAR file, run `ant war`.

The easiest way to add more WIG files to be compiled and deployed as servlets is to add them to `src/wig/tests`. They will be automatically detected, compiled, packaged and deployed to the server at URL http://domain/wig_servlet/<wig_file_name_without_wig_extension>.

Demo Site

The 2009 benchmarks, basic WIG examples, and factorial example are accessible on Jake's web site, at http://echo-flow.com:8080/wig_servlet/.

Division of Group Duties

Division of labor for each Milestone and Deliverable

WIG Milestone 1: Tiny example

Jake implemented the C version of the tiny example using Flex and Bison. This included an evaluator that could handle complex expressions. Jake also developed the algorithm used to minimize the number of parentheses.

Wisam implemented the the Java version of the tiny example using SableCC.

Joos deliverables: Benchmarks, and Comments and Desugaring

Jake handled comments and desugaring, and Wisam wrote the benchmark.

WIG milestone 2: CGI vs. WIG

Jake completed this assignment while Wisam was at a conference.

WIG milestone 3: scanner and parser

Jake developed the lexer, and Wisam developed the parser.

WIG milestone 4: symbol tables

Jake completed this assignment.

WIG milestone 5: type checking

Wisam completed this assignment.

JOOS deliverable: stack limits

Jake completed this assignment.

JOOS deliverable: peephole patterns

Jake and Wisam worked closely to pair program this assignment.

WIG deliverable: compiler

Jake did the following:

1. Formalized the WIG-to-JavaScript transformations.
2. Hand-compiled several benchmarks to verify that the transformations would work as expected. These hand-compiled benchmarks may be found in `group-4/wig/benchmark/RhinoServlet/src-gen`.
3. Developed the algorithms used to implement the transformations, as described in the section called “Algorithms”.
4. Implemented the above algorithms.

Wisam worked on implementing the basic transformation of wig expressions as they appear in the grammar.

WIG deliverable: report

A large part of the report was composed of previous reports written for each milestone. Jake edited the milestone reports for inclusion in the final report.

The section on code generation was original, and written by Jake. The introduction, conclusion, and section on group dynamics was written by Jake as well. It was, unfortunately, not possible to reach Wisam to collaborate with him on these sections.

Overall Group Dynamics

Jake was the stronger programmer and more comfortable with the course material conceptually, so was always in a position of leadership. We tried allocating the work in different ways for each assignment. We started out by pair-programming, but for certain assignments, this felt like a suboptimal approach. We actually had the opportunity to allocate whole assignments to individual group members based on that member's strengths or avaiblability, and this was generally more productive. However, it did not scale very well, as there were mroe assignments where it would not have been possible to Wisam to do them. For example, all of the JOOS assignments, which involved C programming, would not have been feasible for Wisam to do on his own. This motivated us to iterate on our strategy, and try and allocate the work in large, discrete chunks for these assignments. Unfortunately, this was often not the best strategy. For example, this strategy was completely inappropriate for use with the Stack Limit assignment, which required tight integration and pair-programming in order to be productive. The lesson we took out of this is that the nature of the assignment must also be taken account when selecting a strategy for allocating work. Finally, Jake handled most of the work on code generation,and the final report, with Wisam contributing with the implementation aspect.

We felt that at some stages we could have communicated better, with one another, and with Chris Pickett. In particular, the writing of the final report could have been better organized.

Conclusions and Future Work

Conclusions

Summary

In this project, we developed a compiler that would take as input a WIG file and compile it to JavaScript, which could then be compiled to Java bytecode and deployed to a Java servlet container to run as a web application. The project touched on the main aspects of compiler development: we implemented a scanner and parser using the SableCC parser generator; we created a symbol table using the output of SableCC, which we were then able to use to verify the legality of our program, both in terms of type, and other criteria; and finally, we implemented code generation to transform WIG to JavaScript, and used Rhino's JSC to compile the resulting JavaScript code to class files that could run on a Java Virtual Machine, inside a servlet container.

Lessons Learned

Many lessons were learned over the course of this project.

Jake's Take

It was interesting that some concepts used in this project came up later in the project, in a context that was not directly related. For example, a Cactus Stack data structure was used to encode the notion of nested scopes when developing the symbol table. Later on, during the code generation phase, when implementing the algorithm that would compile Function Invocation Expressions in WIG to JavaScript, a stack was also necessary, as we needed to encode the notion of statements which contain other statements which contain function expressions. The concepts from the symbol table phase very much informed this part of the code generation phase.

Using SableCC was very interesting, as, by not giving one basic tree traversal methods on Nodes (e.g., there is no `Node.getChildren()` method) it basically forces the developer to use the Visitor pattern. We found this to be extremely constraining at first, as it really does force one to think about tree manipulation algorithms in a different way, but ultimately we found it to be a very elegant approach. Certainly, it proved to be more than sufficient for all of our needs. I do wonder how well it would scale though. As it is necessary to put every Node that gets visited in a Hashtable so that it may be retrieved by its parents, it seems like it may become very expensive as programs become very large. I wonder if it would scale up to a program with millions of lines of code, for example. code.

It was interesting that it took an hour to think of and define the most complex transformation from WIG to JavaScript, that of Function Invocation Expressions, but it took eight hours to invent a precise algorithm that would actually execute the transformation using SableCC, and finally to implement it. I believe this was a difficult transformation to implement and think about because the code generated at each node relied on a recursive relationship with its parents, rather than with its children, thus turning the structure of the program "inside out". This is a difficult thing to express, but I wonder if there might not be a higher-level way of representing such a transformation than that which may be expressed by pseudocode or a general purpose programming language. I wonder if one might be able to draw ideas about this from the world of model transformation.

Wisam's Take

With a big project such as this I learned that it not as easy as it may seem to coordinate work amongst members. For me personally I realized that I have a lot of weaknesses that I must overcome in order to improve efficiency and effectiveness. The topics in this course where very new to me so I struggled to grasp the concepts quickly and effectively which meant a slow down in actual implementation. I found myself rushing a lot of times last minute trying to get things done so better time management is definitely something I need to work on. I learned a lot about compilers and the different phases involved, in particular how they are realized at the implementation level. In terms of specifics I learned a lot about parsing a grammar to an AST as well as specifics of type checking. I also became more familiar of SableCC and how it generates an AST and how to utilize it for your specific needs.

Future Work

A feature lacking from our compiler are complex tuple operations. We don't see this as a large shortcoming though, as we instead chose to focus on developing elegant transformations.

It might be interesting if WIG was extended to provide high-level language constructs to generate code which allows the browser to communicate with the server asynchronously via Asynchronous JavaScript and XML (AJAX). This could be provided by injecting appropriate JavaScript code into html const fragments. Part of the challenge would then be to develope code templates for the JavaScript code to be injected.

Goodbye

Jake plans to build a compiler that transforms Statecharts, perhaps encoded as SCXML, to JavaScript. A Statechart-to-JavaScript compiler is a prerequisite for his research in modelling the behaviour of web-based user interfaces, and so the skills acquired in this course will soon be put to good use.