AGRAJAG

A GRAph JArGon

Final Report

by:

Zachary Salzbank (zis2102)
Erica Sponsler (es3094)
Nate Weiss (ndw2114)

1 Introduction

The purpose of AGRAJAG is to simplify the process of connecting different objects together, and manipulating them through code. Our language is based on C, but with the capability to easily express tree and graph data structures. We do this by creating a new data type called Node which can hold a value and references to objects that are related to it, also referred to as children. The Node type can have up to ten children. This language is optimal for representing graphs and trees, as well as processing the data contained by these structures. Graphs and trees are very important data structures to both computer scientists and mathematicians. Graph theory and combinatorics use these structures to help solve complex problems and increase the efficiency of many programs.

2 Language Tutorial

AGRAJAG's syntax is very similar to C's syntax. The major differences are:

- Instead of a `main` method, you need to have a `root` method
- No `include` statements
- No direct access to pointers
- No `for` loops
- No type casting
- If something is a character, it can only be added to other characters, and will always be treated as a character. The same goes for ints and booleans.
- Variable declaration and instantiation must be in separate statements and at the beginning of each block. (Between `{` and `}`).
- No strings or arrays - these can be simulated through nodes
- Only types are `int`, `boolean`, `char`, and `Node<Type>`
- Built-in `print` function
- Nodes:
    - Nodes have a value and children
    - Nodes are declared by `Node<type> n; n = <expression of type>;`
    - `Node<Node<Node<Node<...>>>>` is allowed, as long as it eventually terminates in a base type of `int`, `boolean` or `char`.
    - A node's child must have the same type as the parent node
    - Given a node n, n's value is accessed by `n.value`
    - Given a node n, n's i'th child (indexed starting at 0) is accessed by `n[i]`

Apart from these differences, writing a program in AGRAJAG is very similar to C. A short example using a `Node<int>` is below:

Program	Result

`void root()`
`{`
`  Node<int> x;`
`  x = <1>;`
`  x.value = x.value + 42;`
`  print(x.value);`
`}`
	43

You can also have a node with children. To create a child of a node, you must simply instantiate the child of an already instantiated node. For example, the following short program will create a child node whose value is 2, and then set that value to 42.

Program	Result

`void root()`
`{`
`  Node<int> x;`
`  x = <67>;`
`  x[0] = <2>;`
`  x[0].value = 42;`
`  print(x.value);`
`  print(x[0].value);`
`}`
	67
	42

3 Language Reference Manual

3.1 Lexical conventions

3.1.1 Comments

A comment begins when the character sequence `/*` is encountered. The comment ends when the character sequence `*/` is encountered. Comments cannot be nested.

3.1.2 Identifiers

An identifier can be comprised of any combination of alphabetic characters (upper and lower case), integers, and the underscore '_' character. Identifiers cannot begin with an integer or an underscore. Identifiers are case sensitive, for instance `helloWorld` and `HelloWorld` would not be equivalent.

The maximum length of an identifier is 16 characters. Anything longer than 16 characters will result in an error. The minimum length will be 1 character. Anything less than 1 character will result in an error.

3.1.3 Keywords

The following words are reserved as keywords and cannot be used as identifiers:

- **`int`**
- **`char`**
- **`boolean`**

- **`Node`**
- **`void`**
- **`null`**
- **`return`**
- **`if`**
- **`else`**
- **`while`**
- **`true`**
- **`false`**

3.1.4 Constants

The following types of constants are supported in AGRAJAG:

- **Integer constants**
  Integers are any sequence of digits not enclosed within single or double quotes. Digits in any other context (such as in an identifier name) are not considered constants.

  The maximum value of an Integer constant is 2147483647.
  The minimum value of an Integer constant is -2147483648
- **Character constants**
  A character constant is a single character contained within single quotes. A single quote character is denoted by \' (backslash + single quote), and a backslash character is denoted by \\ (two backslashes).

3.2 Syntax Notation

This reference manual uses the following syntax notation:

- Terminals: Indicated by **`bold Courier New`** typeface.
- Non-terminals: Indicated by regular `Courier New` typeface.
- Any terminal or non-terminal followed by the subscript $_{opt}$ notation, is not required and can be omitted.

3.3 Data Type

3.3.1 Node

- A node will consist of a value and a list of pointers to that node's successors. Nodes can have at most 10 successors.
- A node's value can be the value of any expression including another node, as long as the types match.
- A node is inherently directed, as not all nodes that are connected point to each other. A node's successors are not necessarily 'aware' that they are a child to the first node. However, if the user wishes to have an undirected graph, they must simply ensure that for every node `N`, all successors of `N` include `N` as a successor.

3.3.2 Properties of Nodes

- The value of each node will be mutable and obtainable through operations on that node.

    - For a node **`x`**, the value will be obtained by **`x.value`**
- The list of successors will also be mutable and obtainable through operations on that node.
- For a node **`x`**, the nth successor of this node will be obtained by `x[n]` where **`n`** is any non-negative integer up to and including 9.
- Nodes can be assigned the **`null`** value.

3.3.3 Intended Use of Nodes

Mainly to build trees and graphs, but other data structures can be created by using Nodes. An example is a string, which could be represented by a collection of nodes with character values.

3.4 Expressions

Expressions return values. This section lists the available expressions. Precedence of expressions is the same as the order of sections listed below. All expressions within a section have the same precedence.

3.4.1 Primary Expressions

Primary expressions are the first to be evaluated. Leftmost expressions are evaluated first.

3.4.1.1 identifier

An identifier is any previously declared variable, as long as it has been declared according to the rules specified in the Declarations section below.

3.4.1.2 constant

Any boolean (**`true`** or **`false`**), integer, or character. The type returned for each type of constant is as follows:

- boolean: **`boolean`**
- integer: **`int`**
- character: **`char`**

3.4.1.3 (expression)

Parentheses are used to alter precedence. Expressions within parentheses will be evaluated as primary expressions, even if the expressions contained within the parentheses are a lower precedence than the surrounding operations. The returned type and value evaluates to the same as `expression`.

3.4.1.4 function-name (expression-list opt)

Evaluates the expressions described in `function-name`, optionally passing values via `expression-list`. `expression-list` can be a single expression, or a comma-separated list of expressions. The declaration of `function-name` must return a value (cannot be a **`void`** function declaration). Calling **`void`** functions is covered under `statements`.

3.4.2 Operators

3.4.2.1 Unary Operators

Unary operators modify the result of a single expression. Rightmost unary operators are evaluated first.

3.4.2.1.1 `– expression`

The result is the negated value of `expression`, with the same type. `expression` must be **`int`**.

3.4.2.1.2 **`! expression`**

The result is the logical opposite of `expression`. `expression` must be of type **`boolean`** and the return type is **`boolean`**.

3.4.2.2 Multiplicative Operators

Leftmost multiplicative operators are evaluated first.

3.4.2.2.1 `expression * expression`

Multiplies the first `expression` by the second `expression`. The type of both expressions must be the same. The allowed type for `expression` is **`int`**.

3.4.2.2.2 `expression / expression`

Divides the first `expression` by the second `expression`. The type restrictions for multiplication apply to division as well. Since division is always integer division, the result will be the highest integer value less than or equal to the quotient.

3.4.2.3 Additive Operators

Leftmost additive operators are evaluated first.

3.4.2.3.1 `expression + expression`

Adds the first `expression` to the second `expression`. Allowed types for `expression` are **`int`** or **`char`**, but both `expressions` must be of the same type.

3.4.2.3.2 `expression - expression`

Subtracts the second `expression` from the first `expression`. The type restrictions for addition apply to subtraction as well.

3.4.2.4 Relational Operators

Relational operators return a **`boolean`** type. Allowed types for `expression` are **`int`** or **`char`**, but both `expressions` must be of the same type. The following relational operators are available:

- `expression` **<** `expression` (less than)
- `expression` **>** `expression` (greater than)
- `expression` **<=** `expression` (less than or equal to)
- `expression` **>=** `expression` (greater than or equal to)

3.4.2.5 *Equality Operators*

Equality operators return a **`boolean`** type. The type restrictions for relational operators apply to equality operators as well. `expression` can also be a **`boolean`** type. The following equality operators are available:

- `expression` **==** `expression` (equal to)
- `expression` **!=** `expression` (not equal to)

3.4.2.6 *expression && expression*

The **`boolean and`** operator (**`&&`**) returns **`true`** if both expressions are true. Otherwise, it returns **`false`**. Both expressions must be of **`boolean`** type.

3.4.2.7 *expression || expression*

The **`boolean or`** operator (**`||`**) returns **`true`** if either expression is true. Otherwise, it returns **`false`**. Both expressions must be of **`boolean`** type.

3.4.2.8 *Assignment Operator*

An assignment has the following form:

- `identifier` **=** `expression or`
- `node-identifier.value = expression`
- `node-identifier[expression] = expression`
    - Note: the `expression` within the brackets must evaluate to an `int`

The rightmost assignment will occur first. Operands must have the same type. After the leftmost assignment occurs, the value of `expression` is returned.

3.5 Declarations

Declarations are used in AGRAJAG to specify variables and functions. When declaring a variable the type and name must be specified. The types available are:

- **`int`**
- **`char`**
- **`boolean`**
- **`Node<Type>`**
    - when using a `Node`, the contained type specified can be any of the above types, including `Node`
    -

Each variable must have its own type specified, and must be a separate statement from other declarations. After a variable declaration has been made wherever the variable appears in the program, the value associated with the variable will be used. A variable declaration has three parts: type, variable name, statement end. For instance, declaring a variable called `number` of type `int` would look like this:

```
int number;
```

Functions can be declared to return any of the types listed above, or the `void` type, and can take as parameters any values and/or variables of the types listed above. Function declarations can be made anywhere in the program, before or after the function(s) that call them, but cannot be nested within other function definitions. Declarations of functions have four parts: return type, function name, parameter list, function body. Return type specifies the type of value the function returns. Function names can be any valid identifier that is not a reserved keyword. Parameter lists can have up to eight parameters with any mixture of the above types. A function body consists of a set of statements enclosed in braces ('{', '}'). These statements can be any valid AGRAJAG program, but must return a value of the same type as the specified return type. The form of a function declaration is:

```
type function_name(parameter-list) {sequence of statements return
statement}
```

where *`parameter-list`* is a comma separated list of zero or more variables and their associated types. For instance:

```
type parameter1, type parameter2, type parameter3
```

The `root` function is the entry point to the program. It will be the first function executed, and has the return type `void`.

3.6 Statements

There are several allowable statements in AGRAJAG. A sequence of statements will be executed in the order that they appear in the program. The statements that are recognized by AGRAJAG are:

- **Expression statement:** A single expression followed by the end-statement character, ';'.
- **Conditional statement:** An expression that evaluates a boolean expression, and will execute the appropriate statements based on the result. It has the form:

  ```
  if (boolean expression) then {sequence of statements} else
  {sequence of statements_opt}
  ```

  There is no form of conditional without an else section. If the programmer wishes there to be no action in the else case they may write a statement that has the form:

  ```
  if (boolean expression) then {sequence of statements} else {}
  ```

- **While statement:** An expression that evaluates a sequence of expressions based on the value of a boolean expression. The form of a while statement is:

  ```
  while (boolean expression) {sequence of statements}
  ```

  The sequence of statements will be executed until the boolean expression evaluates to false. The boolean expression will be evaluated before each execution of the loop statements.

- **Return statement:** A statement that specifies what value a function should return. A return statement has the form:

  ```
  return (expression_opt);
  ```

  The expression inside the return statement will be evaluated, and the value will then be returned to the calling function. The type of the evaluated expression must match the return type of the function in which the return statement appears. If the expression is omitted, then no value will be returned. In this case, the type of the function should be `void`.

- **Calling a function** with a return type of **`void`** is a statement as well, because all expressions must evaluate to a value, but statements do not have this restriction. A function call to a void function has the form:

  ```
  function-name(parameter-list_opt);
  ```

3.7 Scope Rules

A lexical block begins with a '`{`' and ends with a '`}`'. The scope of any variable in a program will be the lexical block that it is defined in, after the point at which it is defined. If blocks are nested, the scope of the variables within the outer block extend into the inner block. To avoid ambiguity, no overlapping names can be used for identifiers; within any scope, there will only be one identifier with a certain name.

AGRAJAG allows for global variables, which are variables defined outside the scope of any one function, and will be in the scope of all functions. Nested functions are not allowed. Therefore, all functions have global scope. To avoid ambiguity, function identifiers must be unique.

3.8 Compilation and Output

Compilation on the file '`program.ag`' is performed by running the command:

```
./agrajag < program.ag
```

The program output will be displayed on the standard output of the terminal that the compiler was run on. The built in `print` function is used to output data. Any **`int, char or boolean`** argument passed to the `print` function will be output.

3.9 Syntax Summary

3.9.1 Expressions

primary
- expression
! expression
expression binop expression
lvalue assignop expression

3.9.2 Primary

constant
(expression)
primary (expression-list$_{opt}$)
lvalue

3.9.3 lvalue

identifier
node [primary]
node.Value

3.9.4 binop

+	-		
/	*		
<	>	<=	>=
!=	==		
&&			
||			

3.9.5 assignop

=

3.9.6 statement

expression ;
while (expression) { statement-list }
if (expression) { statement-list } else { statement-list$_{opt}$ }
return (expression);

3.9.7 statement-list

statement
statement statement-list

3.9.8 expression-list

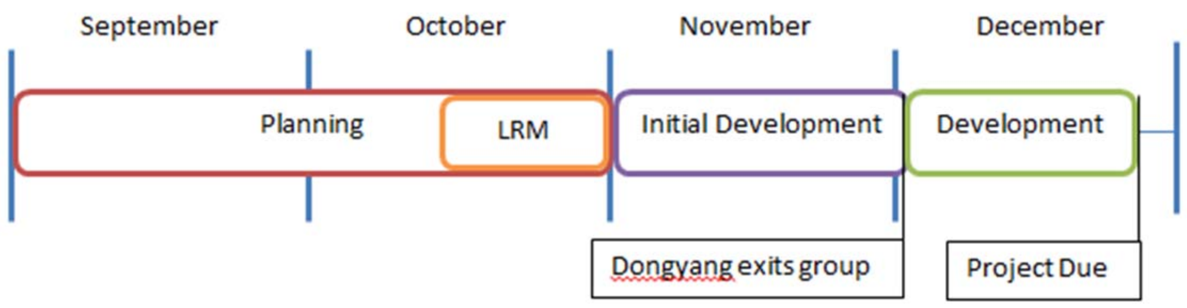
expression
expression,expression-list

4 Project Plan

4.1 Processes

Our team met weekly to help ensure that we were headed in the right direction and making progress. We spent the weekly meetings planning our strategy for long and short term development. Our specifications were outlined in the LRM, and as we determined a need for them to change we would keep the document up-to-date so all team members could reference the most up-to-date specifications. Development was done mainly individually or in pairs, and weekly meetings were used to catch the team up on what progress was made by individuals over the week. Testing was done as features were implemented, and our automated test suites were kept up to date by individuals as changes were made necessary.

4.2 Project Timeline



4.3 Roles and Responsibilities

The project was split between the three group members. Zachary was responsible for semantic analysis as well as the interpreter. Erica was primarily responsible for the production of bytecode from the semantically correct AST, as well as executing the bytecode. Nate took responsibility for the parser, scanner, and provided significant assistance with bytecode compilation and execution.

4.4 Development Environment

We used OCaml and OCamllex for the various portions of the compiler/interpreter. We used VIM and Emacs as our primary editors.

4.5 Project Log

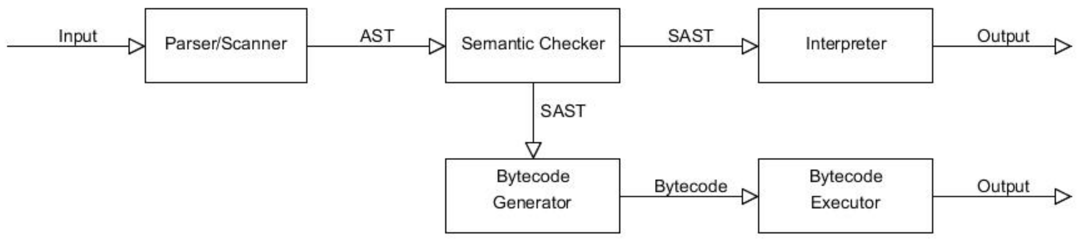
Date	Activity
9/12	Team formed
9/20	First Meeting ● Discussed ideas for languages ● Decided to make a language that includes nodes and supports graphs/trees. ● Wrote proposal draft ● Assigned sample code writing to each group member for the week.
9/27	Second Meeting ● Reviewed sample code progress ● Finished writing final draft of proposal
9/28	Proposal Due
10/4	No meeting because proposal feedback not received yet
10/5	Proposal feedback received
10/11	Third Meeting ● Began work on LRM ● Assigned work to be completed on the LRM to each group member ● Dongyang absent - email sent to catch up on meeting topics
10/17	Email sent out as a meeting reminder to all group members

10/18	Fourth Meeting ● Continued work on LRM ● Resolved issues and questions encountered by individuals during the writing of the LRM ● Dongyang absent
10/23	Fifth Meeting ● Continued work on LRM ● Assigned additional work to ensure completeness over the coming week. ● Dongyang absent
10/24	Confirmation email sent to change meeting time for the coming week.
10/25	No Meeting due to time conflicts
10/26	Email sent to Dongyang to confirm his participation in the group and to ensure meeting attendance was made a priority.
10/27	Email received from Dongyang confirming plans to attend meeting.
10/27	Sixth Meeting ● Finished LRM
10/31	LRM Due
11/1	No meeting due to need for LRM feedback in order to move forward with the project.
11/8	No meeting due to election holiday and need for LRM feedback in order to move forward with the project.
11/10	Email sent to TA requesting an estimate for when we would receive LRM feedback.
11/13	LRM Feedback received
11/15	Seventh Meeting ● Discussed next-steps ● Assigned work to review microc, conduct research, and assess work required for type implementation and nodes. ● Broke coding/research into four sections and assigned responsibilities: ○ Dongyang - Update Scanner/Parser ○ Zachary - Begin semantic analysis research/coding ○ Nathaniel - Research Java bytecode to see if it’s worth translating our language into Java bytecode. ○ Erica - Review microc bytecode, translate a few programs, see what would need to change in compile.ml/bytecode.ml
11/22	Eighth Meeting ● Reviewed progress: ○ Parser/Scanner - no progress ○ Semantic analysis - progress made creating a SAST, or Semantically correct AST. ○ Java bytecode research - probably feasible but likely more complicated than editing microc code. ○ Review of bytecode - Example program translated into bytecode by hand. Several additional commands identified as necessary. ● Reassigned some responsibilities: ○ Nathaniel - assist with bytecode/compile modifications
11/29	Ninth Meeting ● Reviewed progress ● Decided not much else could be accomplished without parser/scanner completed

	● Dongyang absent ● Assigned Parser/Scanner to Nathaniel
12/1	Email sent to Dongyang letting him go from the group
12/6	Tenth Meeting ● Discussed progress on scanner/parser ○ Nodes supported ○ Types supported ● Identified a few remaining commands yet to be implemented in parser/scanner ● After review of parser/scanner, made plans to continue work on semantics and bytecode
12/13	No meeting due to time conflicts
12/14	Eleventh Meeting ● Discussed progress ○ Worked collectively on compile/bytecode/execute ○ Worked on interpret.ml ● Initial interpreter working on most test cases.
12/17	Interpreter improved, working with more test cases
12/18	Mostly working compile.ml and execute.ml completed
12/19	Twelfth Meeting ● Discussed a few remaining modifications ● Reviewed presentation
12/20	Presented project. Received feedback about node storage in compile/execute
12/20-12/21	Node storage improved in compile/execute
12/22	Report completed/turned in

5 Architectural Design

AGRAJAG can compile a program into bytecode and execute it, or run the program without creating this intermediate representation.



The AGRAJAG compiler consists of five different functionality blocks:

- **Parser/Scanner:** Accepts the program as input and generates an abstract syntax tree (AST). Programs written in accordance with the syntax of the language will be accepted by this state, but that does not mean they are valid programs.

- **Semantic Checker:** Accepts the AST from the previous stage and ensures it complies with the semantic rules dictated in the language reference manual. If the program is not semantically correct, an error message will be returned to the user stating the problem encountered. Semantically correct code in the form of a Semantic AST (SAST) can be passed to the interpreter or the bytecode generator stage.
- **Interpreter:** Runs the code from the SAST, starting with the root function. Code is not compiled into specific instructions and does not model the operation of an actual processor, but has the same output as the executed bytecode.
- **Bytecode Generator:** Accepts the code from the SAST and generates a stack-based bytecode representation of the code. This code can be output to the user or run in the bytecode executor.
- **Bytecode Executor:** Runs the bytecode generated in the previous stage. The output from the program is reported back to the user.

The Parser/Scanner and the Bytecode Generator/Executor were developed by Erica and Nate. The Semantic Checker and Interpreter were developed by Zachary.

6 Test Plan

6.1 Representative Source Code and Target Code

6.1.1 Hello World!

Source Code

```
void root()
{
    Node<char> hi;
    hi = <'h'>;
    hi[0] = <'e'>;
    hi[1] = <'l'>;
    hi[2] = <'l'>;
    hi[3] = <'o'>;
    hi[4] = <' '>;
    hi[5] = <' '>;
    hi[6] = <'w'>;
    hi[7] = <'o'>;
    hi[8] = <'r'>;
    hi[9] = <'l'>;
    hi[10] = <'d'>;
    hi[11] = <' '>;
    hi[12] = <'!'>;

    print_string(hi, 14);
}

void print_string(Node<char> n, int len)
{
    int i;
    i = 0;
    print(n.value);
    while (i < len - 1)
    {
       print(n[i].value);
       i = i + 1;
    }
}
```

Byte Code

```
0 global         35 LitI 1        72 Lfp           109 Cnd          146 Sth          183 LitI 0
variables        36 Sfp           73 LitI 1        110 LitI 1       147 Drp 4        184 Rts 1
0 Jsr 52         37 Drp 4         74 Sth           111 LitI 37      148 LitC 32      185 Ent 0
1 Hlt            38 Rsp           75 Drp 4         112 Lfp          149 Cnd          186 LitI 0
2 Ent 1          39 LitI 1        76 LitC 108      113 LitI 6       150 LitI 1       187 Ssp
3 LitI 0         40 LitI 7        77 Cnd           114 Sth          151 LitI 37      188 Rsp
4 Ssp            41 Lfp           78 LitI 1        115 Drp 4        152 Lfp          189 LitI 0
5 LitI 0         42 LitI -3       79 LitI 37       116 LitC 111     153 LitI 11      190 Rts 1
6 LitI 1         43 LitI 7        80 Lfp           117 Cnd          154 Sth
7 Sfp            44 Lfp           81 LitI 2        118 LitI 1       155 Drp 4
8 Drp 4          45 LitI 1        82 Sth           119 LitI 37      156 LitC 33
9 LitI -2        46 Sub           83 Drp 4         120 Lfp          157 Cnd
10 LitI 37       47 Lt            84 LitC 111      121 LitI 7       158 LitI 1
11 Lfp           48 Bne -31       85 Cnd           122 Sth          159 LitI 37
12 LitI -1       49 Rsp           86 LitI 1        123 Drp 4        160 Lfp
13 Ldh           50 LitI 0        87 LitI 37       124 LitC 114     161 LitI 12
14 Jsr -3        51 Rts 2         88 Lfp           125 Cnd          162 Sth
15 Drp 1         52 Ent 1         89 LitI 3        126 LitI 1       163 Drp 4
16 Bra 23        53 LitI 0        90 Sth           127 LitI 37      164 LitI 14
17 LitI 0        54 Ssp           91 Drp 4         128 Lfp          165 LitI 1
18 Ssp           55 LitC 104      92 LitC 32       129 LitI 8       166 LitI 7
19 LitI -2       56 Cnd           93 Cnd           130 Sth          167 Lfp
20 LitI 37       57 LitI 1        94 LitI 1        131 Drp 4        168 Jsr 2
21 Lfp           58 Sfp           95 LitI 37       132 LitC 108     169 Drp 1
22 LitI 1        59 Drp 4         96 Lfp           133 Cnd          170 Rsp
23 LitI 7        60 LitC 101      97 LitI 4        134 LitI 1       171 LitI 0
24 Lfp           61 Cnd           98 Sth           135 LitI 37      172 Rts 0
25 Ldh           62 LitI 1        99 Drp 4         136 Lfp          173 Ent 0
26 LitI -1       63 LitI 37       100 LitC 32      137 LitI 9       174 LitI 0
27 Ldh           64 Lfp           101 Cnd          138 Sth          175 Ssp
28 Jsr -3        65 LitI 0        102 LitI 1       139 Drp 4        176 Rsp
29 Drp 1         66 Sth           103 LitI 37      140 LitC 100     177 LitI 0
30 LitI 1        67 Drp 4         104 Lfp          141 Cnd          178 Rts 1
31 LitI 7        68 LitC 108      105 LitI 5       142 LitI 1       179 Ent 0
32 Lfp           69 Cnd           106 Sth          143 LitI 37      180 LitI 0
33 LitI 1        70 LitI 1        107 Drp 4        144 Lfp          181 Ssp
34 Add           71 LitI 37       108 LitC 119     145 LitI 10      182 Rsp
```

6.1.2 Binary Search

Source Code

```
void root(){
    Node<int> treeRoot;
    Node<int> result;
    treeRoot = <5>;
    treeRoot[0] = <3>;
    treeRoot[1] = <7>;
    treeRoot[0][0] = <2>;
    treeRoot[0][1] = <4>;
    treeRoot[1][0] = <6>;
    treeRoot[1][1] = <8>;

    result = binSearch(treeRoot, 4);
    if(result == null){
       print(false);
    } else {
        print(true);
    }
}

Node<int> binSearch ( Node<int> sNode, int searchFor ) {
      while (sNode != null) {
            if (searchFor < sNode.value) {
                  sNode = sNode[0];
            } else {
                  if(searchFor > sNode.value) {
                         sNode = sNode[1];
                  } else {
                         return sNode;
                  }
            }
      }
      return null;
}
```

Byte Code

```
0 global         41 Beq 13        84 LitI 2        127 Drp 4        170 Ssp
variables        42 LitI 0        85 LitI 37       128 LitI 8       171 Rsp
0 Jsr 74         43 Ssp           86 Lfp           129 Cnd          172 LitI 0
1 Hlt            44 LitI -2       87 LitI 0        130 LitI 2       173 Rts 1
2 Ent 0          45 LitI 37       88 Sth           131 LitI 37      174 Ent 0
3 LitI 0         46 Lfp           89 Drp 4         132 Lfp          175 LitI 0
4 Ssp            47 LitI 1        90 LitI 7        133 LitI 1       176 Ssp
5 Bra 58         48 Ldh           91 Cnd           134 Ldh          177 Rsp
6 LitI 0         49 LitI -2       92 LitI 2        135 LitI 1       178 LitI 0
7 Ssp            50 Sfp           93 LitI 37       136 Sth          179 Rts 1
8 LitI -3        51 Drp 4         94 Lfp           137 Drp 4        180 Ent 0
9 LitI 7         52 Rsp           95 LitI 1        138 LitI 4       181 LitI 0
10 Lfp           53 Bra 8         96 Sth           139 LitI 2       182 Ssp
11 LitI -2       54 LitI 0        97 Drp 4         140 LitI 7       183 Rsp
12 LitI 37       55 Ssp           98 LitI 2        141 Lfp          184 LitI 0
13 Lfp           56 LitI -2       99 Cnd           142 Jsr 2        185 Rts 1
14 LitI -1       57 LitI 7        100 LitI 2       143 LitI 1
15 Ldh           58 Lfp           101 LitI 37      144 Sfp
16 Lt            59 Rts 2         102 Lfp          145 Drp 4
17 Beq 13        60 Rsp           103 LitI 0       146 LitI 1
18 LitI 0        61 Rsp           104 Ldh          147 LitI 7
19 Ssp           62 Rsp           105 LitI 0       148 Lfp
20 LitI -2       63 LitI -2       106 Sth          149 LitNull
21 LitI 37       64 LitI 7        107 Drp 4        150 Eql
22 Lfp           65 Lfp           108 LitI 4       151 Beq 8
23 LitI 0        66 LitNull       109 Cnd          152 LitI 0
24 Ldh           67 Neq           110 LitI 2       153 Ssp
25 LitI -2       68 Bne -62       111 LitI 37      154 LitB false
26 Sfp           69 LitNull       112 Lfp          155 Jsr -2
27 Drp 4         70 Rts 2         113 LitI 0       156 Drp 1
28 Rsp           71 Rsp           114 Ldh          157 Rsp
29 Bra 33        72 LitI 0        115 LitI 1       158 Bra 7
30 LitI 0        73 Rts 2         116 Sth          159 LitI 0
31 Ssp           74 Ent 2         117 Drp 4        160 Ssp
32 LitI -3       75 LitI 0        118 LitI 6       161 LitB true
33 LitI 7        76 Ssp           119 Cnd          162 Jsr -2
34 Lfp           77 LitI 5        120 LitI 2       163 Drp 1
35 LitI -2       78 Cnd           121 LitI 37      164 Rsp
36 LitI 37       79 LitI 2        122 Lfp          165 Rsp
37 Lfp           80 Sfp           123 LitI 1       166 LitI 0
38 LitI -1       81 Drp 4         124 Ldh          167 Rts 0
39 Ldh           82 LitI 3        125 LitI 0       168 Ent 0
40 Gt            83 Cnd           126 Sth          169 LitI 0
```

6.2 Test Suites

Our tests are broken up into two main groups: tests and semantic tests. Tests (located in the `tests` folder) are used to test the functionality of out code. Each of these tests should pass. Semantic tests are used to ensure that code that does not pass the semantics required by the LRM do not get through to the compilation stage. All these tests should fail.

6.3 Test Case Selection

Most of the test cases were chosen to test node functionality and to ensure that it matched the specifications as stated in our LRM. We ensured that Nodes could be declared, initialized, and manipulated properly in every context (e.g. within a function, within a block, in the context of other nodes, etc.).

6.4 Test Automation

We used several automated test suites to test our translator. There is an all-encompassing shell script to run every test case we have, and ensure that the expected result is achieved. We also developed subset scripts that ran a particular class of tests. For example, there is a suite of semantic tests that exercise the semantic analysis, ensuring that all the illegal semantic errors are properly caught. There are also suites of test cases to test every aspect of Node functionality, from declaring nodes of various types to accessing and setting children and values of nodes. These test suites were extremely helpful when developing the bytecode compilation module.

6.5 Responsibilities

Team members were generally responsible for creating test cases for the functionality which they implemented. However, many of the test cases created by one person to test one aspect of the program were useful for testing other facets of the translator. For instance, many of the test cases created specifically for the interpreter were also useful for testing the bytecode execution, and vice versa. We also were all responsible for checking specifications in the Language Reference Manual and ensuring that all specifications had appropriate corresponding test cases. Zachary was especially instrumental in keeping the Language Reference Manual and test cases up-to-date.

7 Lessons Learned

Throughout the course of the project we all learned a lot, not just about compilers, but also about organizing a project where we knew the end goal but weren't sure exactly what it would take to get there. Our collective advice to future teams is to look at the project reports by previous teams, talk to people, and try to get a grasp on what the project entails. In retrospect the project seems relatively straightforward, but at the beginning we didn't always know where we were headed. The best thing to do would probably be to get some perspective early on. It would have helped a lot to know how to get to a completed project from the beginning.

The biggest obstacle we encountered in doing the project had more to do with team member participation than the project itself. Our group started out with four group members, and we tried to divide the work reasonably between ourselves. However, any work assigned to our fourth group member didn't end up getting done. He didn't show up to over half the team meetings, and none of the work assigned to him was actually completed until we decided to let him go from the group. Unfortunately his portion of the project had been the parser and scanner. This meant that progress was practically halted as we were expecting him to complete his work. Once he was no longer on the project we immediately set to work on the parser and scanner, and we progressed steadily throughout the rest of the project, completing our compiler on schedule in spite of the setback.

In light of this experience the first lesson learned was that it is more important to worry about the group as a whole than an individual member of the team. It might have helped us to cut our losses earlier instead of giving our last group member the benefit of the doubt too many times.

The second lesson learned was that it pays to look ahead at how we plan to implement something and make sure it is possible before assuming that it will work. We decided to implement types, and were going to store our data as bytes to take advantage of the different type sizes. Unfortunately we learned later that it would not be possible to store our integer, character, and boolean data objects as a byte array in OCaml. By the time we realized this, we had already implemented an infrastructure to accommodate the size of differently typed objects.

The last lesson learned was that writing code in OCaml cannot be approached like writing a program in other languages that we were more familiar with. Large chunks of code could not be written and then tested because figuring out where the type errors occurred would be very time consuming. Rather, changing one or two lines of code at a time and then recompiling was a much better development style. This allowed for easier debugging during development.

8 Appendix

Please see the attached pages for source code.

```
OBJS = ast.cmo sast.cmo semantics.cmo parser.cmo scanner.cmo interpret.cmo bytecode.cmo
compile.cmo execute.cmo agrajag.cmo

TESTS = \
arith1 \
arith2 \
fib \
for1 \
func1 \
func2 \
func3 \
gcd \
global1 \
hello \
if1 \
if2 \
if3 \
if4 \
ops1 \
var1 \
while1

TARFILES = Makefile testall.sh scanner.mll parser.mly \
    ast.ml bytecode.ml interpret.ml compile.ml execute.ml agrajag.ml \
    $(TESTS:%=tests/test-%.ag) \
    $(TESTS:%=tests/test-%.out)

agrajag : $(OBJS)
    ocamlc -g -o agrajag $(OBJS)

.PHONY : test
test : agrajag testall.sh
    ./testall.sh

scanner.ml : scanner.mll
    ocamllex scanner.mll

parser.ml parser.mli : parser.mly
    ocamlyacc -v parser.mly

%.cmo : %.ml
    ocamlc -g -c $<

%.cmi : %.mli
    ocamlc -g -c $<

agrajag.tar.gz : $(TARFILES)
    cd .. && tar czf agrajag/agrajag.tar.gz $(TARFILES:%=agrajag/%)

.PHONY : clean
clean :
    rm -f agrajag parser.ml parser.mli scanner.ml testall.log \
    *.cmo *.cmi *.out *.diff *.output *.parsed
```

```
# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
bytecode.cmo: ast.cmo
bytecode.cmx: ast.cmx
compile.cmo: sast.cmo bytecode.cmo ast.cmo
compile.cmx: sast.cmx bytecode.cmx ast.cmx
execute.cmo: bytecode.cmo ast.cmo
execute.cmx: bytecode.cmx ast.cmx
interpret.cmo: sast.cmo ast.cmo
interpret.cmx: sast.cmx ast.cmx
agrajag2.cmo: scanner.cmo parser.cmi compile.cmo bytecode.cmo ast.cmo
agrajag2.cmx: scanner.cmx parser.cmx compile.cmx bytecode.cmx ast.cmx
agrajag.cmo: semantics.cmo scanner.cmo sast.cmo parser.cmi interpret.cmo execute.cmo
compile.cmo bytecode.cmo ast.cmo
agrajag.cmx: semantics.cmx scanner.cmx sast.cmx parser.cmx interpret.cmx execute.cmx
compile.cmx bytecode.cmo ast.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
sast.cmo: ast.cmo
sast.cmx: ast.cmx
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
semantics.cmo: sast.cmo ast.cmo
semantics.cmx: sast.cmx ast.cmx
parser.cmi: ast.cmo
```

```
type action = Ast | Interpret | Semantics | Bytecode | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);
              ("-i", Interpret);
                        ("-s", Semantics);
              ("-b", Bytecode);
              ("-c", Compile); ]
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  let semantic_prog = Semantics.translate program in
  match action with
    Ast -> let listing = Ast.string_of_program program
           in print_string listing
  | Semantics -> print_endline (Sast.string_of_sast semantic_prog)
  | Interpret -> ignore (Interpret.run semantic_prog)
  | Bytecode -> let listing =
      Bytecode.string_of_prog (Compile.translate semantic_prog)
    in print_endline listing
  | Compile -> Execute.execute_prog (Compile.translate semantic_prog)
```

```
(* Writtent by Zachary Salzbank, Erica Sponsler and Nate Weiss *)

{ open Parser }

let char_regex = [^ '\\' '\'']
let id = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*

rule token = parse
  [' ' '\t' '\r' '\n']           { token lexbuf } (* Whitespace *)
| "/*"                           { comment lexbuf }           (* Comments *)
| '('                            { LPAREN }
| ')'                            { RPAREN }
| '{'                            { LBRACE }
| '}'                            { RBRACE }
| '['                            { LBRACKET }
| ']'                            { RBRACKET }
| ';'                            { SEMI }
| ','                            { COMMA }
| '+'                            { PLUS }
| '-'                            { MINUS }
| '*'                            { TIMES }
| '/'                            { DIVIDE }
| '='                            { ASSIGN }
| '!'                            { BANG }
| "&&"                           { BOOLAND }
| "||"                           { BOOLOR }
| "=="                           { EQ }
| "!="                           { NEQ }
| '<'                            { LT }
| "<="                           { LEQ }
| ">"                            { GT }
| ">="                           { GEQ }
| "if"                           { IF }
| "else"                         { ELSE }
| "while"                        { WHILE }
| "break"                        { BREAK }
| "return"                       { RETURN }
| "int"                          { INT }
| "char"                         { CHAR }
| "boolean"                      { BOOLEAN }
| "Node"                         { NODE }
| ".value"                       { VALUEOF }
| "void"                         { VOID }
| "true"                         { BOOLEAN_LITERAL(true) }
| "false"                        { BOOLEAN_LITERAL(false) }
| "null"                         { NULL_LITERAL }
| "'\\''"                        { CHAR_LITERAL('\'') }
| "'\\\\'"                       { CHAR_LITERAL('\\') }
| "'" (char_regex as c) "'"      { CHAR_LITERAL(c) }
| ['0'-'9']+ as lxm              { INTEGER_LITERAL(int_of_string lxm) }
| id as lxm                      { ID(lxm) }
| eof                            { EOF }
| _ as char                      { raise (Failure("illegal character " ^ Char.escaped char)) }
```

```
and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

```ocaml
(* Written by Zach Salzbank, Erica Sponsler and Nate Weiss *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | BoolAnd | BoolOr

type constant =
    Integer of int
  | Character of char
  | Boolean of bool
  | Null

type obj_type =
    IntType
  | CharType
  | BooleanType
  | VoidType
  | NodeType of obj_type
  | NullType

type l_value =
    Id of string
  | Unop of l_value * unary_op

and unary_op = Child of expr | ValueOf

and expr =
    Literal of constant
  | LValue of l_value
  | Node of expr
  | Binop of expr * op * expr
  | Assign of l_value * expr
  | Call of string * expr list
  | Neg of expr
  | Bang of expr
  | Noexpr

type v_decl = {
    vname: string;
    vtype: obj_type;
    vdefault: expr option;
}

type stmt =
    Block of v_decl list * stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | While of expr * stmt

type func_decl = {
    ftype: obj_type;
    fname : string;
    formals : v_decl list;
    locals : v_decl list;
    body : stmt list;
  }

type program = v_decl list * func_decl list

let string_of_char = function
    '\'' -> "\\'"
  | '\\' -> "\\\\"
  | _ as c   -> String.make 1 c

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | BoolAnd -> "&&"
  | BoolOr -> "||"

let rec string_of_const = function
    Integer(l) -> string_of_int l
  | Character(c) -> "'" ^ string_of_char c ^ "'"
  | Boolean(b) -> if b then "true" else "false"
  | Null -> "null"

let rec string_of_unop = function
    ValueOf -> ".value"
  | Child(e) -> "[" ^ string_of_expr e ^ "]"

and string_of_lval = function
    Id(s) -> s
  | Unop (l, uo) -> string_of_lval l ^ string_of_unop uo

and string_of_expr = function
    Literal(c) -> (match c with
        Integer(l) -> string_of_int l
      | Character(c) -> "'" ^ string_of_char c ^ "'"
      | Boolean(b) -> if b then "true" else "false"
      | Null -> "null"
    )
  | LValue(l) -> string_of_lval l
  | Node(e) -> "<" ^ string_of_expr e ^ ">"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Assign(l, e) -> string_of_lval l ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Neg(e) -> "-" ^ string_of_expr e
  | Bang(e) -> "!" ^ string_of_expr e
  | Noexpr -> ""
```

```
let rec string_of_obj_type t = match t with
    IntType -> "int"
  | CharType -> "char"
  | BooleanType -> "boolean"
  | VoidType -> "void"
  | NodeType(s) -> "Node<" ^ string_of_obj_type s ^ ">"
  | NullType -> "null"

let string_of_vdecl id = match id.vdefault with
    None -> string_of_obj_type id.vtype ^ " " ^ id.vname ^ ";\n"
  | Some(d) -> string_of_obj_type id.vtype ^ " " ^ id.vname ^ " = " ^ string_of_expr d ^ ";\n"

let rec string_of_stmt = function
    Block(vars, stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
       String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([], [])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_fdecl fdecl =
  string_of_obj_type fdecl.ftype ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_vdecl
  fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

```
/* Written by Zachary Salzbank, Erican Sponsler and Nate Weiss */

%{ open Ast %}

%token INT CHAR BOOLEAN VOID
%token NODE
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA LBRACKET RBRACKET
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token BOOLAND BOOLOR BANG
%token VALUEOF
%token RETURN IF ELSE WHILE BREAK
%token NULL_LITERAL
%token <bool> BOOLEAN_LITERAL
%token <char> CHAR_LITERAL
%token <int> INTEGER_LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc BANG
%right ASSIGN
%left EQ NEQ
%left BOOLAND BOOLOR
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left VALUEOF

%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [] }
 | program vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   func_decl LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
   { { ftype = snd $1;
         fname = fst $1;
     formals = $3;
     locals = List.rev $6;
     body = List.rev $7 } }

obj_type:
    INT                         { IntType }
  | CHAR                        { CharType }
  | BOOLEAN                     { BooleanType }
  | NODE LT obj_type GT         { NodeType($3) }
```

```
obj_decl:
    obj_type ID      { ($2, $1) }

func_decl:
    VOID ID   { ($2, VoidType) } /* Need this because can't declare anything but function as
    void */
  | obj_decl  { $1 }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    obj_decl                   { [{ vname = fst $1; vtype = snd $1; vdefault = None; }] }
  | formal_list COMMA obj_decl { ({ vname = fst $3; vtype = snd $3; vdefault = None; }) :: $1 }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
    obj_decl SEMI { { vname = fst $1; vtype = snd $1; vdefault = None; } }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | RETURN SEMI { Return(Noexpr) }
  | LBRACE vdecl_list stmt_list RBRACE { Block(List.rev $2, List.rev $3) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([], [])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

literals:
    INTEGER_LITERAL                            { Integer($1) }
  | CHAR_LITERAL                               { Character($1) }
  | BOOLEAN_LITERAL                            { Boolean($1) }
  | NULL_LITERAL                               { Null }

l_value:
    ID                                         { Id($1) }
  | l_value VALUEOF                            { Unop($1, ValueOf) }
  | l_value LBRACKET expr RBRACKET             { Unop($1, Child($3)) }

expr:
    literals                                   { Literal($1) }
  | LT expr GT                                 { Node($2) }
  | l_value                                    { LValue($1) }
  | MINUS expr                                 { Neg($2) }
  | BANG expr                                  { Bang($2) }
  | expr BOOLAND expr                          { Binop($1, BoolAnd, $3) }
  | expr BOOLOR expr                                { Binop($1, BoolOr, $3) }
  | expr PLUS   expr                                { Binop($1, Add,   $3) }
  | expr MINUS  expr                                { Binop($1, Sub,   $3) }
  | expr TIMES  expr                                { Binop($1, Mult,  $3) }
  | expr DIVIDE expr                                { Binop($1, Div,   $3) }
  | expr EQ     expr                                { Binop($1, Equal, $3) }
  | expr NEQ    expr                                { Binop($1, Neq,   $3) }
  | expr LT     expr                                { Binop($1, Less,  $3) }
  | expr LEQ    expr                                { Binop($1, Leq,   $3) }
  | expr GT     expr                                { Binop($1, Greater,  $3) }
  | expr GEQ    expr                                { Binop($1, Geq,   $3) }
  | l_value ASSIGN expr                             { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN                    { Call($1, $3) }
  | LPAREN expr RPAREN                              { $2 }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

```
(* Written by Zachary Salzbank *)

open Ast

type simple_expr =
    Literal of Ast.constant
  | LValue of l_value
  | Node of simple_expr
  | Binop of expr * Ast.op * expr
  | Assign of l_value * expr
  | Call of func_decl * expr list
  | Neg of expr
  | Bang of expr
  | Noexpr

and unary_op = Child of expr | ValueOf

and simple_l_value =
    Id of v_decl
  | Unop of l_value * unary_op

and stmt =
    Block of v_decl list * stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | While of expr * stmt

and expr = simple_expr * Ast.obj_type

and l_value = simple_l_value * Ast.obj_type

and v_decl = {
  vvname: string;
  vvtype: Ast.obj_type;
  vvdefault: expr option;
}

and func_decl = {
  fftype: Ast.obj_type;
  ffname : string;
  fformals : v_decl list;
  flocals : v_decl list;
  fbody : stmt list;
  parsed: bool;
}

type program = v_decl list * func_decl list

let string_of_sast_v v =
  "Variable: " ^ v.vvname ^ " = " ^ string_of_obj_type v.vvtype ^ "\n"

let string_of_sast_f f =
  "Function: " ^ f.ffname ^ " = " ^ string_of_obj_type f.fftype ^
  String.concat "\n\t" ("\nFormals: " :: (List.map string_of_sast_v f.fformals)) ^
  String.concat "\n\t" ("\nLocals: " :: (List.map string_of_sast_v f.flocals))


let string_of_sast (vars, funcs) =
  String.concat "" (List.map string_of_sast_v vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sast_f funcs)
```

```ocaml
(* Written by Zachary Salzbank *)

open Ast
open Sast

type symbol_table = {
  parent : symbol_table option;
  variables : Sast.v_decl list;
  functions: Sast.func_decl list;
}

type trans_env = {
  scope : symbol_table;
}

let rec find_variable (scope : symbol_table) name =
  try
    List.find (fun v -> v.vvname = name) scope.variables
  with Not_found ->
    match scope.parent with
    Some(parent) -> find_variable parent name
    | _ -> raise (Failure("variable " ^ name ^ " not defined"))

let var_exists scope name =
    try
      let _ = find_variable scope name
      in true
    with Failure(_) ->
      false

let integer_check i =
  if (i > 32767 || i < -32768) then
    raise (Failure("invalid value for integer"))
  else
    i

let id_check name =
  if String.length name > 16 then
    raise (Failure("identifiers must be 16 characters or less"))
  else
    name

let rec find_function (scope : symbol_table) name =
  try
    List.find (fun f -> f.ffname = name) scope.functions
  with Not_found ->
    match scope.parent with
    Some(parent) -> find_function parent name
    | _ -> raise (Failure("function " ^ name ^ " not defined"))

let rec find_print (scope : symbol_table) t =
  try
    List.find (fun f -> (f.ffname = "print" && (List.hd f.fformals).vvtype = t)) scope.functions
  with Not_found ->
```

```
    match scope.parent with
    Some(parent) -> find_print parent t
    | _ -> raise (Failure("function print(" ^ string_of_obj_type t ^ ") not defined"))

let func_exists scope name =
    List.exists (fun f -> f.ffname = name) scope.functions

let assign_allowed lt rt = match lt with
    NodeType(t) -> (lt = rt) || (rt = NullType)
  | _ -> lt = rt

let rec can_assign lt rval =
  let (_, rt) = rval in
    if assign_allowed lt rt then
      rval
    else
      raise (Failure("type " ^ string_of_obj_type rt ^ " cannot be put into type " ^
      string_of_obj_type lt))

let inner_type t =
  match t with
      NodeType(it) -> it
    | _ -> raise (Failure("accessor cannot be used on " ^ string_of_obj_type t))


let is_node = function
    NodeType(_) -> true
  | _ -> false

let can_op lval op rval =
  let (_, lt) = lval
  and (_, rt) = rval in
  let type_match = (lt = rt) in
  let int_or_char = (lt = IntType || lt = CharType) in
  let node = ((is_node lt) && rt == NullType) || (lt == NullType && (is_node
  rt)) || (type_match && (is_node lt)) in
  let result = match op with
      Ast.Add     -> (type_match && int_or_char), lt
    | Ast.Sub     -> (type_match && int_or_char), lt
    | Ast.Mult    -> (type_match && lt = IntType), lt
    | Ast.Div     -> (type_match && lt = IntType), lt
    | Ast.Equal   -> (type_match && (int_or_char || lt = BooleanType)) || node, BooleanType
    | Ast.Neq     -> (type_match && (int_or_char || lt = BooleanType)) || node, BooleanType
    | Ast.Less    -> (type_match && int_or_char), BooleanType
    | Ast.Leq     -> (type_match && int_or_char), BooleanType
    | Ast.Greater -> (type_match && int_or_char), BooleanType
    | Ast.Geq     -> (type_match && int_or_char), BooleanType
    | Ast.BoolAnd -> (type_match && lt == BooleanType), BooleanType
    | Ast.BoolOr  -> (type_match && lt == BooleanType), BooleanType
  in if fst result then
    snd result
  else
    raise (Failure("operator " ^ string_of_op op ^ " cannot be used on types " ^
      string_of_obj_type lt ^ " and " ^ string_of_obj_type rt))
```

```
let translate (globals, funcs) =
  let rec trans_lval env = function
      Ast.Id(n) -> let vdecl = (find_variable env.scope n) in
                   Sast.Id(vdecl), vdecl.vvtype
    | Ast.Unop(lval, op) -> let l, t = trans_lval env lval in
                            let inner = inner_type t in
                            let newt = match op with
                                Ast.Child(_) -> t
                              | Ast.ValueOf  -> inner
                            in Sast.Unop((l, t), trans_unop env op), newt
  and trans_unop env = function
      Ast.Child(e) -> let e, t = (trans_expr env e) in
                      if (t == IntType) then
                        Sast.Child(e, t)
                      else
                        raise (Failure("index must be of type int"))
    | Ast.ValueOf  -> Sast.ValueOf
  and trans_expr env = function
      Ast.Literal(l) -> (match l with
          Integer(i) -> Literal(Integer(integer_check i)), IntType
        | Character(c) -> Literal(Character(c)), CharType
        | Boolean(b) -> Literal(Boolean(b)), BooleanType
        | Null -> Literal(Null), NullType
      )
    | Ast.Node(e) ->
        let e, t = trans_expr env e
        in Sast.Node(e), NodeType(t)
    | Ast.LValue(l) ->
        let lv, t = trans_lval env l
        in Sast.LValue(lv, t), t
    | Ast.Binop(e1, op, e2) ->
        let e1 = trans_expr env e1
        and e2 = trans_expr env e2
        in let rtype = can_op e1 op e2 in
        Sast.Binop(e1, op, e2), rtype
    | Ast.Call(n, a) ->
        let args =
          List.map (fun s -> (trans_expr env s)) a in
        let fdecl = if n = "print" then
          (find_print env.scope (snd (List.hd args)))
        else
          (find_function env.scope n)
        in let types =
          List.rev (List.map (fun v -> v.vvtype) (List.rev fdecl.fformals)) in
        let checked_args = try
            List.map2 can_assign types args
          with Invalid_argument(x) ->
            raise (Failure("invalid number of arguments")) in
        Sast.Call(fdecl, checked_args), fdecl.fftype
    | Ast.Assign(lv, e) ->
        let lval, t = (trans_lval env lv) in
        let aval = (trans_expr env e) in
        Sast.Assign((lval, t), (can_assign t aval)), t
    | Ast.Neg(e) ->
        let e, t = (trans_expr env e) in
        if t = IntType then
          Sast.Neg(e, t), t
        else
          raise (Failure("cannot negate type " ^ string_of_obj_type t))
    | Ast.Bang(e) ->
        let e, t = (trans_expr env e) in
        if t = BooleanType then
          Sast.Bang(e, t), t
        else
          raise (Failure("cannot get logical opposite of type " ^ string_of_obj_type t))
    | Ast.Noexpr ->
        Sast.Noexpr, VoidType
  in let add_local env v =
    let evalue = match (var_exists env.scope (id_check v.vname)) with
        true -> raise (Failure("redeclaration of " ^ v.vname))
      | false -> match v.vdefault with
                      None -> None
                    | Some(e) -> Some(can_assign v.vtype (trans_expr env e))
    in let new_v = {
            vvname = v.vname;
            vvtype = v.vtype;
            vvdefault = evalue;
      }
    in let vars = new_v :: env.scope.variables
    in let scope' = {env.scope with variables = vars}
    in {(*env with*) scope = scope'}
  in let rec trans_stmt env = function
      Ast.Block(v, s) ->
        let scope' = {parent = Some(env.scope); variables = []; functions = []}
        in let env' = {(*env with*) scope = scope'}
        in let block_env = List.fold_left add_local env' (List.rev v)
        in let s' = List.map (fun s -> trans_stmt block_env s) s
        in Sast.Block(block_env.scope.variables, s')
    | Ast.Expr(e) ->
        Sast.Expr(trans_expr env e)
    | Ast.Return(e) ->
        Sast.Return(trans_expr env e)
    | Ast.If (e, s1, s2) ->
        let e' = trans_expr env e
        in Sast.If(can_assign BooleanType e', trans_stmt env s1, trans_stmt env s2)
    | Ast.While (e, s) ->
        let e' = trans_expr env e
        in Sast.While(can_assign BooleanType e', trans_stmt env s)
  in let add_func env f =
    let new_f = match ((var_exists env.scope f.fname) || (func_exists env.scope f.fname)) with
      true -> raise (Failure("redeclaration of " ^ f.fname))
    | false ->
        let scope' = {parent = Some(env.scope); variables = []; functions = []}
        in let env' = {(*env with*) scope = scope'}
        in let env' = List.fold_left add_local env' (List.rev f.formals)
        in {
          fftype = f.ftype;
          ffname = id_check f.fname;
          fformals = env'.scope.variables;
          flocals = [];
          fbody = [];
          parsed = false;
        }
    in let funcs = new_f :: env.scope.functions
    in let scope' = {env.scope with functions = funcs}
    in {(*env with*) scope = scope'}
  in let trans_func env (f : Ast.func_decl) =
    let sf = find_function env.scope (f.fname)
    in let functions' = List.filter (fun f -> f.ffname != sf.ffname) env.scope.functions
    in let scope' = {parent = Some(env.scope); variables = sf.fformals; functions = []}
    in let env' = {(*env with*) scope = scope'}
    in let formals' = env'.scope.variables
    in let env' = List.fold_left add_local env' (f.locals)
    in let remove v =
      not (List.exists (fun fv -> fv.vvname = v.vvname) formals')
    in let locals' = List.filter remove env'.scope.variables
    in let body' = List.map (fun f -> trans_stmt env' f) (f.body)
    in let new_f = {
      sf with
      fformals = formals';
      flocals = locals';
      fbody = body';
      parsed = true;
    }
    in let funcs = new_f :: functions'
    in let scope' = {env.scope with functions = funcs}
    in {(*env with*) scope = scope'}
  in let validate_func f =
    let is_return = function
        Sast.Return(e) -> true
      | _ -> false
    in let valid_return = function
        Sast.Return(e) -> if assign_allowed f.fftype (snd e) then
                            true
                          else
                             raise (Failure(  f.ffname ^ " must return type " ^
                              string_of_obj_type f.fftype ^
                              ", not " ^ string_of_obj_type (snd e)
                             ))
      | _ -> false
    in let returns = List.filter is_return f.fbody
    in let _ = List.for_all valid_return returns
    in let return_count = List.length returns
    in if (return_count = 0 && f.fftype != VoidType) then
      raise (Failure(f.ffname ^ " must have a return type of " ^ string_of_obj_type f.fftype))
    else if List.length f.fformals > 8 then
      raise (Failure(f.ffname ^ " must have less than 8 formals"))
    else
      f
  in let make_print t =
    {
      fftype = VoidType;
      ffname = "print";
      fformals = [{
        vvname = "val";
        vvtype = t;
        vvdefault = None;
      }];
      flocals = [];
      fbody = [];
      parsed = false;
    }
  in let global_scope = {
    parent = None;
    variables = [];
    functions = List.map make_print [IntType; CharType; BooleanType];
  }
  in let genv = {
    scope = global_scope;
  }
  in let genv = List.fold_left add_local genv (List.rev globals)
  in let genv = List.fold_left add_func genv (List.rev funcs)
  in let genv = List.fold_left trans_func genv (List.rev funcs)
  in if func_exists genv.scope "root" then
    (genv.scope.variables, List.map validate_func genv.scope.functions)
  else
    raise (Failure("no root function defined"))
```

```
(* Written by Zachary Salzbank *)

open Ast
open Sast

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

type node = {
  value: int;
  child: int array;
}

type environment = {
  locals: int NameMap.t;
  globals: int NameMap.t;
  nodes: node array;
}

exception ReturnException of int * environment

let node_check i =
  if i < 0 then
    raise (Failure("node does not exist yet"))
  else
    i

let get_child a i =
  let cur_len = Array.length a
  in if i < cur_len then
    a, Array.get a i
  else
    let a = Array.append a (Array.make (i - cur_len + 1)  (-1))
    in a, Array.get a i

let array_replace a i x =
  let left = Array.sub a 0 i
  in let right = Array.sub a (i+1) ((Array.length a)-i-1)
  in let nodes = Array.append left (Array.make 1 x)
  in Array.append nodes right

(* Main entry point: run a program *)

let run (vars, funcs) =
  (* Put function declarations in a symbol table *)
  let func_decls = List.fold_left
    (fun funcs fdecl -> NameMap.add fdecl.ffname fdecl funcs)
    NameMap.empty
    funcs

  in let rec l_value (env:environment) = function
        Id(var) ->
```

```
          if NameMap.mem var.vvname env.locals then
            (NameMap.find var.vvname env.locals), env
          else
            (NameMap.find var.vvname env.globals), env
      | Unop(lv, op) ->
          let lvi, env = l_value env (fst lv)
          in let rnode = Array.get env.nodes (node_check lvi)
          in match op with
              Child(e) ->
                let v, env = eval env e
                in let children, child = get_child rnode.child (node_check v)
                in let rnode' = {rnode with child = children}
                in let nodes = array_replace env.nodes lvi rnode'
                in let env' = {env with nodes = nodes}
                in child, env'
            | ValueOf  ->
                rnode.value, env
  and eval (env:environment) e =
   (* Evaluate an expression and return (value, updated environment) *)
      let e, t = e in match e with
        Literal(c) -> (match c with
            Integer(i) -> i
          | Character(ch) -> Char.code ch
          | Boolean(b) -> if b then 1 else 0
          | Null -> -1
          ), env
      | Noexpr -> 1, env (* must be non-zero for the for loop predicate *)
      | Binop(e1, op, e2) ->
          let v1, env = eval env e1 in
          let v2, env = eval env e2 in
          let b1 = if (v1 == 1) then true else false in
          let b2 = if (v2 == 1) then true else false in
          let boolean i = if i then 1 else 0 in
          (match op with
              Add -> v1 + v2
            | Sub -> v1 - v2
            | Mult -> v1 * v2
            | Div -> v1 / v2
            | Equal -> boolean (v1 = v2)
            | Neq -> boolean (v1 != v2)
            | Less -> boolean (v1 < v2)
            | Leq -> boolean (v1 <= v2)
            | Greater -> boolean (v1 > v2)
            | Geq -> boolean (v1 >= v2)
            | BoolAnd -> boolean (b1 && b2)
            | BoolOr -> boolean (b1 || b2)
          ), env
      | Assign(lval, e) ->
          let v, env = eval env e
          in (match fst lval with
              Id(var) ->
                if NameMap.mem var.vvname env.locals then(
                  let locals = NameMap.add var.vvname v env.locals in
                  v, {env with locals = locals;}
```

```
               ) else
                 let globals = NameMap.add var.vvname v env.globals
                 in v, {env with globals = globals;}
           | Unop(lv, op) ->
               let lvi, env = l_value env (fst lv)
               in let rnode = Array.get env.nodes (node_check lvi)
               in let n, env = (match op with
                   Child(e) ->
                     let iv, env = eval env e
                     in let a, cv = get_child rnode.child iv
                     in let a = array_replace a iv v
                     in {rnode with child = a}, env
                 | ValueOf  ->
                     {rnode with value = v}, env)
               in let nodes = array_replace env.nodes lvi n
               in let env = {env with nodes = nodes;}
               in v, env)
      | Neg(e) ->
          let v, env = eval env e
          in (v * -1), env
      | Bang(e) ->
          let v, env = eval env e
          in let value = if v = 1 then 0 else 1
          in value, env
      | Node(e) ->
          let nvalue, env = eval env (e, NullType)
          in let n = {
            value = nvalue;
            child = Array.make 10 (-1);
          }
          in let index = Array.length env.nodes in
          let nodes = Array.append env.nodes (Array.make 1 n)
          in let env = {env with nodes = nodes;}
          in index, env
      | LValue(l) -> l_value env (fst l)
      | Call(f, actuals) ->
          if f.ffname = "print" then
            let fml = List.hd f.fformals
            in let t = fml.vvtype
            in let v, env = eval env (List.hd actuals)
            in let s = match t with
                IntType -> string_of_int v
              | BooleanType -> if v==1 then "true" else "false"
              | CharType -> string_of_char (Char.chr v)
              | _ -> raise (Failure("Invalid print call"))
            in print_endline s;
            0, env
          else
            let fdecl = NameMap.find f.ffname func_decls
            in
            let actuals, env = List.fold_left
              (fun (actuals, env) actual ->
                let v, env = eval env actual in
                v :: actuals, env)
          ([], env)
          (List.rev actuals)
      in try
        let env' = call fdecl actuals env
        in 0, {env' with locals = env.locals;}
      with ReturnException(v, env') ->
        let env = {env' with locals = env.locals;}
        in v, env
and add_local env vdecl =
  let dv = match vdecl.vvtype with
      NodeType(_) -> -1
    | _ -> 0
  in let value, env = match vdecl.vvdefault with
      Some(x) -> eval env x
    | None -> dv, env
  in
  let locals = NameMap.add vdecl.vvname value env.locals
  in let env = {env with locals = locals;}
  in
  env
and exec env = function
    Block(vars, stmts) ->
      let env' = List.fold_left add_local env vars
      in let env' = List.fold_left exec env' stmts
      in let locals = NameMap.fold
        (fun lname lval lst -> if NameMap.mem lname env.locals then
            NameMap.add lname lval lst
          else
            lst
        )
        env'.locals
        NameMap.empty
      in {env' with locals = locals};
  | Expr(e) ->
      let _, env = eval env e in env
  | If(e, s1, s2) ->
    let v, env = eval env e in
    exec env (if v != 0 then s1 else s2)
  | While(e, s) ->
    let rec loop env =
      let v, env = eval env e in
        if v != 0 then loop (exec env s) else env
    in loop env
  | Return(e) ->
    let v, env = eval env e in
    raise (ReturnException(v, env))

(* Invoke a function and return an updated environment *)
and call fdecl actuals (env : environment) =
  (* Enter the function: bind actual values to formal arguments *)
  let locals =
    List.fold_left2
        (fun locals formal actual -> NameMap.add formal actual locals)
        NameMap.empty
        (List.map (fun v -> v.vvname) fdecl.fformals)
        actuals
  in let env = {env with locals = locals}
  (* Initialize local variables *)
  in let env = List.fold_left
    add_local
    env
    (List.rev fdecl.flocals)
  in
  (* Execute each statement in sequence, return updated environment *)
  List.fold_left exec env fdecl.fbody

(* Run a program: initialize environment with globals, find and run "root" *)
in let env = {
  globals = NameMap.empty;
  locals = NameMap.empty;
  nodes = Array.make 0 {
    value = 0;
    child = Array.make 10 (-1);
  };
}
in let env = List.fold_left
    (fun env vdecl ->
      let value, env = match vdecl.vvdefault with
          Some(x) -> eval env x
        | None -> 0, env
      in let globals = NameMap.add vdecl.vvname value env.globals
      in {env with globals = globals;}
    )
    env vars
in try
  call (NameMap.find "root" func_decls) [] env
with ReturnException(v, env) ->
  ignore(print_endline("Exited with code " ^ string_of_int v));
  env
```

```
(* Written by Erica Sponsler and Nate Weiss *)

type bstmt =
    LitI of int   (* Push a literal *)
  | LitC of char  (* Push a character *)
  | LitB of bool  (* Push a boolean *)
  | LitNull       (* Push null *)
  | Drp of int    (* Drop a certain number of bytes from the stack *)
  | Bin of Ast.op (* Perform arithmetic on top of stack *)
  | Lod           (* Fetch global variable *)
  | Ldh           (* Fetch from the heap *)
  | Str           (* Store global variable *)
  | Sth           (* Store value to the heap *)
  | Lfp           (* Load frame pointer relative *)
  | Sfp           (* Store frame pointer relative *)
  | Cnd          (* Create a node on the node-heap with value at top of stack *)
  | Jsr of int    (* Call function by absolute address *)
  | Ssp
  | Rsp
  | Ent of int    (* Push FP, FP -> SP, SP += i *)
  | Rts of int    (* Restore FP, SP, consume formals, push result *)
  | Beq of int    (* Branch relative if top-of-stack is zero *)
  | Bne of int    (* Branch relative if top-of-stack is non-zero *)
  | Bra of int    (* Branch relative *)
  | Hlt           (* Terminate *)

type prog = {
    num_globals : int;   (* Number of global variables *)
    text : bstmt array; (* Code for all the functions *)
  }

let string_of_stmt = function
    LitI(i) -> "LitI " ^ string_of_int i
  | LitC(c) -> "LitC " ^ string_of_int (int_of_char c)
  | LitB(b) -> "LitB " ^ string_of_bool b
  | LitNull -> "LitNull"
  | Drp(i) -> "Drp " ^ string_of_int i
  | Bin(Ast.Add) -> "Add"
  | Bin(Ast.Sub) -> "Sub"
  | Bin(Ast.Mult) -> "Mul"
  | Bin(Ast.Div) -> "Div"
  | Bin(Ast.Equal) -> "Eql"
  | Bin(Ast.Neq) -> "Neq"
  | Bin(Ast.Less) -> "Lt"
  | Bin(Ast.Leq) -> "Leq"
  | Bin(Ast.Greater) -> "Gt"
  | Bin(Ast.Geq) -> "Geq"
  | Bin(Ast.BoolAnd) -> "And"
  | Bin(Ast.BoolOr) -> "Or"
  | Lod -> "Lod"
  | Ldh -> "Ldh"
  | Str -> "Str"
  | Lfp -> "Lfp"
  | Sfp -> "Sfp"
  | Sth -> "Sth"
  | Cnd -> "Cnd"
  | Jsr(i) -> "Jsr " ^ string_of_int i
  | Ssp -> "Ssp"
  | Rsp -> "Rsp"
  | Ent(i) -> "Ent " ^ string_of_int i
  | Rts(i) -> "Rts " ^ string_of_int i
  | Bne(i) -> "Bne " ^ string_of_int i
  | Beq(i) -> "Beq " ^ string_of_int i
  | Bra(i) -> "Bra " ^ string_of_int i
  | Hlt    -> "Hlt"

let string_of_prog p =
  string_of_int p.num_globals ^ " global variables\n" ^
  let funca = Array.mapi
      (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
  in String.concat "\n" (Array.to_list funca)
```

```
(* Written by Erica Sponsler and Nate Weiss *)

open Sast
open Bytecode

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
    function_index : (int * Ast.obj_type) StringMap.t; (* Index for each function *)
    global_index   : (int * Ast.obj_type) StringMap.t; (* "Address" for global variables *)
    node_heap_index : (int * Ast.obj_type) StringMap.t; (* "Address" for node storage *)
    local_index    : (int * Ast.obj_type) StringMap.t; (* FP offset for args, locals *)
    number_of_locals : int; (* number of locals *)
  }

type glh = Glob | Loc | Heap

let size_of = function
    Ast.IntType ->  4
  | Ast.CharType -> 1
  | Ast.BooleanType -> 1
  | Ast.VoidType -> 1
  | Ast.NodeType(_) -> 4
  | Ast.NullType -> 4

(* val enum : int -> 'a list -> (int * 'a) list *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

let snd = function
    (x, y) -> y

let fst = function
    (x, y) -> x

let inFunc = false

let rec remove_print = function
    [] -> [];
  | hd::tl -> if((compare "print" hd.ffname) == 0) then (remove_print tl) else (hd :: (
  remove_print tl))

let rec find_base_type env = function
    Id s ->
      (try let id_pair  = (StringMap.find s.vvname env.local_index) in
      (snd id_pair)
          with Not_found -> try let id_glob_pair = (StringMap.find s.vvname env.global_index) in
      (snd id_glob_pair)
          with Not_found -> raise (Failure ("undeclared variable " ^ s.vvname)))
  | Unop(l,op) -> (find_base_type env (fst l))

let rec find_this_type base_type = function
```

```
    Id s -> (match base_type with
      Ast.NodeType(t) -> t
    | _ -> raise (Failure "base_type is not a node in find_this_type Id"))
  | Unop(l,op) -> (match base_type with
      Ast.NodeType(t) -> (match op with
    ValueOf -> (find_this_type t (fst l))
      | Child(exp) -> (find_this_type base_type (fst l)))
    | _ -> raise (Failure "base_type is not a node in find_this_type Id"))

let find_type env l =
    (find_this_type (find_base_type env l) l)

let heap_glob_or_loc env = function
    Id s ->
       (try (ignore (StringMap.find s.vvname env.local_index));
      Loc
          with Not_found -> try (ignore (StringMap.find s.vvname env.global_index));
      Glob
          with Not_found -> raise (Failure ("undeclared variable " ^ s.vvname)))
  | Unop(l, op) -> Heap

(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)
let string_map_pairs map pairs =
  List.fold_left (fun m (i, (n, t)) -> StringMap.add n (i, t) m) map pairs

(** Translate a program in SAST form into a bytecode program.  Throw an
    exception if something is wrong, e.g., a reference to an unknown
    variable or function *)
let translate (globals, functions) =

  (* Allocate "addresses" for each global variable *)
  let global_indexes = string_map_pairs StringMap.empty (enum 1 0 (List.map (fun v -> (v.vvname,
   v.vvtype)) globals)) in

  (* Assign indexes to function names; built-in "print" is special *)
  let built_in_functions = StringMap.add "print" (-1, Ast.IntType) StringMap.empty in
  let function_indexes = string_map_pairs built_in_functions
      (enum 1 1 (List.map (fun f -> (f.ffname, f.fftype)) (remove_print functions))) in

  (* Translate a function in SAST form into a list of bytecode statements *)
  let translate env fdecl =
    (* Bookkeeping: FP offsets for locals and arguments *)
    let num_formals = List.length fdecl.fformals
    and num_locals = List.length fdecl.flocals
    and local_offsets = enum 1 1 (List.map (fun v -> (v.vvname, v.vvtype)) fdecl.flocals)
    and formal_offsets = enum (-1) (-2) (List.map (fun v -> (v.vvname, v.vvtype))
    fdecl.fformals) in
    let env = { env with local_index = string_map_pairs
          StringMap.empty (local_offsets @ formal_offsets); number_of_locals = num_locals } in
    let loc_env = env in
    let rec l_value_helper lenv = function
        Id s ->
      (try let id_pair  = (StringMap.find s.vvname lenv.local_index) in
      [LitI (fst id_pair)] @ [LitI 37] @ [Lfp]
```

```
        with Not_found -> try let id_glob_pair = (StringMap.find s.vvname lenv.global_index) in
    [LitI (fst id_glob_pair)] @ [LitI 56] @ [Lod]
        with Not_found -> raise (Failure ("undeclared variable " ^ s.vvname)))
    | Unop(l, op) -> (match op with
    ValueOf -> (l_value_helper lenv (fst l)) @ [LitI (-1)] @ [Ldh]
  | Child(exp) -> (l_value_helper lenv (fst l)) @ (simple_expr lenv (fst exp)) @ [Ldh])

  and

  l_value lenv lval from = (match lval with
    Id s ->
      (try let id_pair  = (StringMap.find s.vvname lenv.local_index) in
       [LitI (fst id_pair)]
      with Not_found -> try let id_glob_pair = (StringMap.find s.vvname lenv.global_index) in
       [LitI (fst id_glob_pair)]
         with Not_found -> raise (Failure ("undeclared variable " ^ s.vvname)))
      | Unop(l, op) -> (match op with
      ValueOf -> (l_value_helper lenv (fst l)) @ [LitI (-1)]
    | Child(exp) -> (l_value_helper lenv (fst l)) @ (simple_expr lenv (fst exp))))

  and

  simple_expr lenv = function
  Literal c -> (match c with
         Ast.Integer(i) -> [LitI i]
       | Ast.Character(ch) -> [ LitC ch ]
       | Ast.Boolean(b) -> [ LitB b ]
       | Ast.Null -> [ LitNull ]
         )
    | Binop (e1, op, e2) -> (simple_expr lenv (fst e1)) @ (simple_expr lenv (fst e2)) @ [Bin
    op]
    | Assign (l, e) -> (simple_expr lenv (fst e)) @ (l_value lenv (fst l) (Assign(l,e))) @ (
    match (heap_glob_or_loc lenv (fst l)) with
    Glob -> [Str]
  | Loc -> [Sfp]
  | Heap -> [Sth])

    | Call (fname, actuals) -> (try
    (List.concat (List.map (fun a -> (simple_expr lenv (fst a))) (List.rev actuals))) @ (match
     fname.ffname with
      "print" -> (match (List.hd fname.fformals).vvtype with
        Ast.IntType -> [Jsr (-1)]
      | Ast.BooleanType -> [Jsr (-2)]
      | Ast.CharType -> [Jsr (-3)]
      | _ -> raise (Failure ("Cannot print this type.")))
    | _ -> [Jsr (fst (StringMap.find fname.ffname env.function_index)) ])
      with Not_found -> raise (Failure ("undefined function " ^ fname.ffname)))
    | Neg(e) -> [LitI 0] @ (simple_expr lenv (fst e)) @ [Bin Ast.Sub]
    | Bang(e) -> [LitI 1] @ (simple_expr lenv (fst e)) @ [Bin Ast.Sub]
    | Node(e) -> (simple_expr lenv e) @ [Cnd]
    | LValue(l) -> (l_value lenv (fst l) (LValue(l))) @ (match (heap_glob_or_loc lenv (fst l))
     with
    Glob -> [LitI 8] @ [Lod]
  | Loc -> [LitI 7] @ [Lfp]
```

```
   | Heap -> (*[LitI 4] @*) [Ldh])
     | Noexpr -> []

   in let rec stmt lenv = function
   Block (vars, sl) -> if (inFunc)
   then ((ignore (inFunc = false)); (List.concat (List.map (fun a -> (stmt lenv a)) sl)))
   else let new_env = {lenv with local_index = (string_map_pairs lenv.local_index (enum 1 (lenv
   .number_of_locals + 1) (List.map (fun v -> (v.vvname, v.vvtype)) vars))); number_of_locals =
    lenv.number_of_locals + (List.length vars) } in [LitI (List.length vars)] @ [Ssp] @ List.
   concat (List.map (fun a -> (stmt new_env a)) sl) @ [Rsp]
     | Expr e       -> (simple_expr lenv (fst e)) @ [Drp (size_of (snd e))]
     | Return e     -> (simple_expr lenv (fst e)) @ [Rts num_formals]
     | If (p, t, f) -> let t' = (stmt lenv t) and f' = (stmt lenv f) in
   (simple_expr lenv (fst p)) @ [Beq(2 + List.length t')] @
   t' @ [Bra(1 + List.length f')] @ f'
     | While (e, b) ->
     let b' = (stmt lenv b) and e' = (simple_expr lenv (fst e)) in
     [Bra (1+ List.length b')] @ b' @ e' @
     [Bne (-(List.length b' + List.length e'))]

   in ((ignore (inFunc = true));[Ent num_locals] @      (* Entry: allocate space for locals *)
   (stmt loc_env (Block([](*fdecl.fformals*), fdecl.fbody))) @  (* Body *)
   [LitI 0; Rts num_formals])   (* Default = return 0 *)

 in let env = { function_index = function_indexes;
        global_index = global_indexes;
        node_heap_index = StringMap.empty;
        local_index = StringMap.empty;
            number_of_locals = 0} in

 (* Code executed to start the program: Jsr main; halt *)
 let entry_function = try
   [Jsr (fst (StringMap.find "root" function_indexes)); Hlt]
 with Not_found -> raise (Failure ("no \"main\" function - sincerely, compile.ml"))
 in

 (* Compile the functions *)
 let func_bodies = entry_function :: List.map (translate env) functions in

 (* Calculate function entry points by adding their lengths *)
 let (fun_offset_list, _) = List.fold_left
     (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0) func_bodies in
 let func_offset = Array.of_list (List.rev fun_offset_list) in

 { num_globals = List.length globals;
   (* Concatenate the compiled functions and replace the function
      indexes in Jsr statements with PC values *)
   text = Array.of_list (List.map (function
   Jsr i when i > 0 -> Jsr func_offset.(i)
     | _ as s -> s) (List.concat func_bodies))
 }
```

```
(* Written by Erica Sponsler and Nate Weiss *)

open Ast
open Bytecode

(* Stack layout just after "Ent":

              <-- SP
   Local n
   ...
   Local 0
   Saved FP   <-- FP
   Saved PC
   Arg 0
   ...
   Arg n *)
module NodeMap = Map.Make(String)

type node = {
  value: int;
  children: int list;
  }

type env = {
    stack: int array;
    node_heap: (node) NodeMap.t;
    globals: int array;
    saved_sp: int array;
  }

let rec modify_ith_element new_val i target list =
  if i==target then (match list with
    [] -> new_val :: []
  | hd::tl -> new_val :: tl)
      else (match list with
    hd::tl -> hd :: (modify_ith_element new_val (i+1) target tl)
      | _ -> raise (Failure ("Invalid List")))

let execute_prog prog =

  let rec exec fp sp hp sc pc env = match prog.text.(pc) with
(*    Lit i  -> stack.(sp) <- i ; exec fp (sp+1) (pc+1)*)
    LitI i -> env.stack.(sp) <- i ; exec fp (sp+1) hp sc (pc+1) env
  | LitC c -> env.stack.(sp) <- (int_of_char c) ; exec fp (sp+1) hp sc (pc+1) env
  | LitB b -> env.stack.(sp) <- if b then 1 else 0 ; exec fp (sp+1) hp sc (pc+1) env
  | LitNull -> env.stack.(sp) <- (-1) ; exec fp (sp+1) hp sc (pc+1) env
(*  | Drp -> exec fp (sp-1) (pc+1) *)
  | Drp i -> exec fp (sp-1) hp sc (pc+1) env
  | Bin op -> let op1 = env.stack.(sp-2) and op2 = env.stack.(sp-1) in
      env.stack.(sp-2) <- (let boolean i = if i then 1 else 0 in
      let b1 = (op1 == 1) in
      let b2 = (op2 == 1) in
      match op with
    Add     -> op1 + op2
```

```
     | Sub     -> op1 - op2
     | Mult    -> op1 * op2
     | Div     -> op1 / op2
     | Equal   -> boolean (op1 =  op2)
     | Neq     -> boolean (op1 != op2)
     | Less    -> boolean (op1 <  op2)
     | Leq     -> boolean (op1 <= op2)
     | Greater -> boolean (op1 >  op2)
     | Geq     -> boolean (op1 >= op2)
     | BoolAnd -> boolean (b1 && b2)
     | BoolOr  -> boolean (b1 || b2)) ;
    exec fp (sp-1) hp sc (pc+1) env
  | Lod -> env.stack.(sp-2) <- env.globals.(env.stack.(sp-2)) ; exec fp (sp-1) hp sc (pc+1) env
  | Str ->  env.globals.(env.stack.(sp-1)) <- env.stack.(sp-2) ; exec fp (sp-1) hp sc (pc+1) env
  | Lfp -> env.stack.(sp-2) <- env.stack.(fp+env.stack.(sp-2)) ; exec fp (sp-1) hp sc (pc+1) env
  | Sfp -> env.stack.(fp+env.stack.(sp-1)) <- env.stack.(sp-2) ; exec fp (sp-1) hp sc (pc+1) env
  | Ldh -> (try let node_val = (NodeMap.find (string_of_int env.stack.(sp-2)) env.node_heap) in
  (match env.stack.(sp-1) with
    (-1) -> env.stack.(sp-2) <- node_val.value ; exec fp (sp-1) hp sc (pc+1) env
   | _ -> env.stack.(sp-2) <- (List.nth node_val.children env.stack.(sp-1)) ; exec fp (sp-1) hp
    sc (pc+1) env) with Not_found -> raise (Failure ("Failed to find node in Ldh")))
  | Sth ->  (try let node_val = (NodeMap.find (string_of_int env.stack.(sp-2)) env.node_heap) in
   (match env.stack.(sp-1) with
    (-1) -> let new_heap = (NodeMap.add (string_of_int env.stack.(sp-2)) {node_val with value
    = env.stack.(sp-3)} env.node_heap) in exec fp (sp-2) hp sc (pc+1) {env with node_heap =
    new_heap}
   | _ -> let new_node = {node_val with children = (modify_ith_element env.stack.(sp-3) 0 env.
  stack.(sp-1) node_val.children)} in let new_heap = (NodeMap.add (string_of_int env.stack.(sp
  -2)) new_node env.node_heap) in exec fp (sp-2) hp sc (pc+1) {env with node_heap = new_heap})
   with Not_found -> raise (Failure ("Failed to find node in Sth")))
  | Cnd -> let new_heap = (NodeMap.add (string_of_int hp) {value = env.stack.(sp-1); children =
  []} env.node_heap) in
  env.stack.(sp-1) <- hp ; exec fp sp (hp+1) sc (pc+1) {env with node_heap = new_heap}
  | Jsr(-1) -> print_endline (string_of_int env.stack.(sp-1)) ; exec fp sp hp sc (pc+1) env
  | Jsr(-2) -> print_endline (if (env.stack.(sp-1) == 1) then "true" else "false") ; exec fp sp
  hp sc (pc+1) env
  | Jsr(-3) -> print_endline (Char.escaped (char_of_int env.stack.(sp-1))) ; exec fp sp hp sc (
  pc+1) env
  | Jsr i   -> env.stack.(sp)   <- pc + 1       ; exec fp (sp+1) hp sc i env
  | Ssp -> env.saved_sp.(sc) <- sp               ; exec fp (sp + env.stack.(sp-1) - 1) hp (sc+1)
   (pc+1) env
  | Rsp ->  let new_sp = env.saved_sp.(sc-1) in exec fp new_sp hp (sc-1) (pc+1) env
  | Ent i   -> env.stack.(sp)   <- fp           ; exec sp (sp+i+1) hp sc (pc+1) env
  | Rts i   -> let new_fp = env.stack.(fp) and new_pc = env.stack.(fp-1) in
             env.stack.(fp-i-1) <- env.stack.(sp-1) ; exec new_fp (fp-i) hp sc new_pc env
  | Beq i   -> exec fp (sp-1) hp sc (pc + if env.stack.(sp-1) =  0 then i else 1) env
  | Bne i   -> exec fp (sp-1) hp sc (pc + if env.stack.(sp-1) != 0 then i else 1) env
  | Bra i   -> exec fp sp hp sc (pc+i) env
  | Hlt     -> ()

  in let env = { stack = Array.make 1024 (-1);
        node_heap = NodeMap.empty;
        globals = Array.make prog.num_globals (-1);
        saved_sp = Array.make 1024 (-1) }
  in exec 0 0 0 0 0 env
```