Name: Shant Stepanian
Uni: sps2141
Semester: Summer 2013 CVN
Course: COMS W4115
Assignment: Final Project

BoredGame: The Language

1. An Introduction to BoredGame

BoredGame is a language designed to help people create their own games. As the tagline goes, “if you are bored from your existing board games, create a new one!”

BoredGame will specialize in specifying games based on a two-dimensional board. Among the features of the language that will help in this goal:

- Defining symbols for players and pieces
- Reading input and printing output
- Flexible input parsing to support various input formats
- Quick setup and access for the 2d board data

The rest of the documentation will cover the details of the language. Some of the conventions and style here is borrowed from the C Language Reference Manual by Ritchie

2. Tutorial

2.1 A First Example

A game typically consists of the following parts:

- Players
- Pieces
- A board, typically representable in a 2D matrix fashion
- Moves, that players can execute with pieces to modify the board and its state

The BoredGame language has constructs that help with each of these:

- Players and Pieces can be represented using the “enum” construct, which basically denotes select symbols to represent something in a game.
    - For example, in Monopoly, the Piece enum would be represented using the values of Car, Dog, Thimble, Shoe, and so on
- Boards can be represented using the “board” construct, where you can draw out your board representation in the programming language itself.
- Moves can be represented by reading in data from the user, and then using typical programming language constructs like functions and conditional statements to use that information to control the game

As a quick tutorial, let’s try to play Tic-Tac-Toe.

First, what concepts do we have here?

- Players - playerX and playerO
- Pieces on the board - X, O, and B (for blank)
- The board: an example

  X O B
  B X O
  X B B

  (playerO is in trouble!)

So how do we represent this in our program? Here is a start!

```
enum Player
	playerX, playerO
end enum
enum Piece
	X, O, B
end enum
function int main()
	board<Piece> myboard = {
		B B B
```

```
            B B B
            B B B
        }
        Player turn = playerX

        printLine("Hello " @ turn)
        string mymove = readLine()

        dosomething(myboard, mymove)
        // Now let's calculate!
        return 0
end function

function int dosomething(board<Piece> myboard, string move)
        if move ~= "([1-3])-([1-3])" -> int a, int b then
                myboard[a, b] = X
        end if
        return 0
end function
```

Let's go through some points:

- Define your enums using the "enum" declaration. Note that the declaration includes the comma-separated values that you'd like to represent.
- The main function is the entrypoint to your program. It must be defined.
- Note a couple of the programming conventions:
    - Blocks of code, such as enum declarations, function declarations, and if statements, typically end in "end" statements (e.g. end if, end function), so that it is clear when something starts and when something ends
    - Statements are separated by newlines. So a contiguous block of a statement must be laid out on a single line
- We define the board and that it would have "Piece"s as its components using the syntax with the braces {}. As should be clear above, we create a 3x3 board here
- We can print and read text using the printLine and readLine functions, respectively. These are provided for you, the client.
- For printing strings, we can conc**AT**enate strings with other types to create text dynamically using the at @ symbol.
- You can call other functions and have them do something, as shown in the dosomething example above
- Note the regular expression syntax in the first like of "dosomething" - via the use of parenthesis, it says that we will capture the text that matches what is inside of it and place it inside the "int a" and "int b" variables. Thus, it is easy to read texts and extract the data that we need.
- Each function should end in a return statement to indicate the result.

That is all for the tutorial - do dig deeper into the Language Reference Manual to get a fuller flavor of what you can do with the language, and see a larger example in the appendix with a 2-player

implementation of the game of checkers!

2.2 How to Compile and Run

To setup and compile your program:

- Create a file containing your code with the extension .game, e.g. checkers.game
- cd into the directory where your file resides
- Then run the compile command:
    - ./boredgame.out -f yourgamename
    - Note - you should not specify the .game extension here
- If the compilation is successful, the game binary is created, which you can run:
    - ./yourgamename.out

3. Language Reference Manual

returning a method early and having code after will not be marked as an error, just ignored. at least until we get a working compiler

undeclared vars give runtime exceptions, not compile-time checked yet
reserved functions, including the generated ones
vars cannot share names of classes
no preference of and over or. use parenthesis to make it clear
no nulls
stmts can change the exec env; expressions just return

3.1 Lexical Convetions

3.1.1 Whitespace handling

Spaces, tabs, and comments will be ignored from a logical perspective and are simply there to separate tokens.

Newlines (i.e. \r\n or \n) are not ignored by the language. They are used to separate statements and constructs. This will be detailed in later sections, but in summary:

- They are used to separate the beginning and end declaration of block statements
- They separate statements within blocks.
    - See examples of the above two below
- Extra newlines can be added between function/enum declarations or within function declarations as extra whitespace

```
while 1 == a do \newline-enforced-before-and-after-this-block-declaration
        stmt1 \newline
        \newline
        \newline
        stmt2 \newline
end while \newline-enforced-before-and-after-this-block-declaration
```

For the reference manual below, the text \newline will be used to denote areas where newlines must be used to demarcate regions of code.

3.1.2 Comments

Comments can be marked in two ways:

1) For a single-line via two backslashes, e.g.

```
my code line1    // my comment
my code line2
```

The commented text consists of everything to the right of the // including the // itself, up until the end of that line. Hence, the "// my comment" is the commented code above

2) The characters /* introduce a comment, which terminates with the characters */

```
This can /* also extend over
multiple lines,
like this */
```

3.1.3 Identifiers

Identifiers will consist of alphanumeric characters (the letters a-z, A-Z, and numbers 0-9). The first character of an identifier must be a letter. Identifiers are case sensitive; e.g. the identifier "iden1" is different from the identifier "IDEN1"

3.1.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

int	if	do	function
string	then	while	return
boolean	elseif	for	
board	else	true	
enum	end	false	

3.1.5 Numeric literals

Integer literals will be as a consecutive sequence of digits, e.g. 12345 or 00123

There are no floating point numbers or fractions. Any operations leading to fractions (e.g. division) will have the fractional part truncated

Negative numbers are not represented as a part of the numeric literal. You must use an operator for that, e.g. 0 - 9

3.1.6 String literals

Strings literals will start with a double quote and continue until the next double quote found on the

same line. e.g.

"12345" => results in a string 12345

"" => results in a blank string

A string cannot span lines in the source code. If it does, this will result in a compile error

Escape characters that are compatible in C can work in BoredGame as well (see http://en.wikipedia.org/wiki/Escape_sequences_in_C), except for escaping double-quotes \". It is currently not possible to escape or print quotes in BoredGame

3.2 Data Types

The language will support the following data types

3.2.1 int

Definition:

int objects in the language represent numeric integer values
In memory, they will be represented as 32-bit signed integers

3.2.2 string

Definition:

string objects represent a sequence of characters

3.2.3 boolean

Definition:

boolean objects represent either *true* or *false* in boolean logic

3.2.4 enum

Definition:

The *enum* keyword is used to define a set of identifiers that can describe the values of a particular domain

For example:

- In a deck of cards, a suit of a card can have the values diamond club heart spade
- The colors of a rainbow can have the values red orange yellow green blue indigo violet

An enum can be used in the language when the game developer would like to constrain chosen variable values to a particular type. Hence, each enum declaration would define a new type in the program.

The identifiers used for each enum value are of a global scope; hence, they cannot be used anywhere else, whether in other enums or as variable names

Note the usage of the newlines to segregate the declaration lines from the enum lines. We also cannot have extra blank lines here - hence, each enum declaration essentially comes out to be 3 lines.

BNF Syntax:

enum_decl:
    enum ID \newline
        formal_list \newline
    end enum \newline

where

formal_list:
    | ID
    | formal_list COMMA ID

Examples:

enum Suit
    diamond, club, heart, spade
end enum

3.2.5 board<T>

Definition:

The board type represents a 2d array that would represent the board structure of a game. This type can be parameterized by specifying another *enum* type inside the brackets when declaring the variable. The idea is to be able to “graphically” define the board structure of a game in the program using the representative symbols of the game that the developer decides on

The board is defined using the syntax as follows:

board<T> board = { \newline
    id11 id12 id13 … \newline
    id21 id22 id23 … \newline
    … \newline
} \newline

where

- The board declaration starts with { and ends with }, with lines in between for the pieces on the board
- id11 id12 … are of the type T, and T is a reference to an enum value.

- Each row has the same amount of identifiers, thus resulting in an n-by-m array
- Each row is split via a newline, and we cannot have any blank lines within the braces
- At least 1 row and 1 column must be defined in the board

A board object exposes some information via the following operators and identifiers. These examples use *boardvar* as the variable identifier

- boardvar.rowlength => (read-only) returns an integer representing the number of rows on the board
- boardvar.collength => (read-only) returns an integer representing the number of columns on the board
- boardvar[r,c] => (read and assign) accesses the element at the r-th row and c-th column of the board. These accesses are 1-indexed, not 0-indexed, as to fit more naturally towards the notations that board games use.

BNF:

board_literal:
    { \newline
        id_array
    }

where:
id_list:
    | ID
    | id_list ID
id_array:
    | id_list
    | id_array \newline id_list

Example: Using Chess

```
enum ChessPiece
    o, p, r, n, b, q, k, P, R, N, B, Q, K
end enum

board<ChessPiece> myboard = {
    r n b q k b n r
    p p p p p p p p
    o o o o o o o o
    o o o o o o o o
    o o o o o o o o
    o o o o o o o o
    P P P P P P P P
    R N B Q K B N R
    }
```

3.3 Expressions

An expression represents some combination of tokens that will denote an object

Here are the various ways to form them, in order of precedence (highest to lowest)

3.3.1 Primitive Literals

Specifying a literal for ints, strings, and booleans, as mentioned in sections 3.1-3.3, will result in an expression for that given type

3.3.2 Board Literals

Specifying the literal for the board, as mentioned in section 3.5, will result in an expression for that type

3.3.3 Identifiers

Specifying an identifier will result in an expression of that type. The way to declare identifiers and their types will be discussed in section 5

3.3.4 Parenthesis

An expression can be surrounded by parenthesis to help give a finer control to the developer over precedence of operations (and to override the defaults)

(*expression*)

3.3.5 Bracket operator [,]

*expr*1 [*expr*2 , *expr*3]

When used in conjunction with a board identifier (e.g. boardvar[row,col]), this will return the element
expr1 must be of type board
expr2 and expr3 must be of type int

3.3.6 dot operator

expr . *id*

When used in conjunction with a board identifier (e.g. boardvar.collength), this will return the value of the property associated with that board variable
Currently, only expressions of type board are supported for the expr value

3.3.7 binary operator

These operators are used in expressions of the following form:

expression binop *expression*

For a given pair of expressions in a binary operation, the order of evaluation is always left to right. For certain cases outlined below, the right-hand expression may not need to be evaluated.

Within the class of binary operators, we have the following precedences (highest to lowest):

- * /
    - For multiplication and division
    - Expects int expressions as inputs and returns an int value w/ the given result. As mentioned earlier, for a division operation resulting in a fractional value, the fractional part would be truncated
    - Left-associative
- + -
    - For addition and subtraction
    - Expects int expressions as inputs and returns an int value w/ the given result
    - Left-associative
- < > <= >= == !=
    - These are the comparison operators
    - These operators will return a boolean value indicating the value of the comparison
    - The left-hand and right-hand operators must have the same type
    - The expected input expression type depends on the operator:
        - < > <= >= must have int expressions
        - == != must have int, string, boolean, enum expressions
- &&
    - Boolean “and” operator
    - Expects boolean expressions as input; returns a boolean expression of value true if both are true, and false otherwise
        - The right-hand expression will only be evaluated if the left-hand expression is true, as otherwise, we already are guaranteed that the expression as a whole will be false
    - Left-associative
- ||
    - Boolean “or” operator
    - Expects boolean expressions as input; returns a boolean expression of value true if at least one of them is true, and false otherwise
        - The right-hand expression will only be evaluated if the left-hand expression is false, as otherwise, we already are guaranteed that the expression as a whole will be true
    - Left-associative
- @

- String concatenation operator
- Returns a string expression
- One of the expressions involved must be a string. The other expression can be either a string, int, boolean, or enum. (i.e. no boards are allowed to be concatenated)
- Left-associative

3.3.8 Function Calls

Functions, in addition to returning values to be used for expressions, can be called as standalone statements. This is for cases where the function has a side effect, e.g. printing out some values to the console

call:
    identifier(args$_{opt}$)
actuals_opt:
    | <blank>
    | *actuals_list*
actuals_list:
    | *expr*
    | *actuals_list*,*expr*

The arguments of the function call will be evaluated in a left-to-right order. The count and types of the arguments in the call must match the count and types of the arguments in the function declaration.

3.4 Statements

A statement represents the most granular unit of work to be executed in a program. Statements themselves don't return values (as expressions do) and can modify the environment of the program (e.g. adding symbols to the symbol table)

The following are the kinds of statements that can be defined in BoredGame.

3.4.1 Variable Declaration

To declare a variable, use the form:

vdecl:
    type_id id \newline

type_id:
    | *id*
    | *id < type_id >*

where type-specifier represents a type (e.g. int, string, boolean, custom enum type, or a board type w/ the enum subtype specified, e.g. board<myenum>)

Example:

int myvar
board<myenum> myboard

3.4.2 Variable Assignment

To assign a value to a variable, use the form:

identifier = *expression* *newline*

The type of the returned expression must equal the type of the identifier

Example:

myvar = 1

3.4.3 Variable Declaration and Assignment

This combines both the variable declaration and the expression.

vdecl = *expr* \newline
(vdecl was defined earlier)

The type of the returned expression must equal the type of the variable

Example:

int myvar = 1

3.4.4 Return Statements

A return statement indicates the end of the execution flow within a function and the value to be returned to the caller. The type of the expression must match the type defined in the function declaration (more on the function declaration structure in section 6)

return *expression* *newline*

3.4.5 if/elseif/else Conditionals

Regular If/else statements:

If/else statements will follow the pattern below.

- The elseifs and else clauses are optional.

- We can have a list of elseifs
- The expression after the if keyword and elseif keywords must return a boolean

if_else:
    if *expression* then *newline*
        statement-list
    elseifs
    else
    end if \newline
elseifs:
    | <blank>
    | elseif *expression* then *newline*
        statement-list
else:
    | <blank>
    | else *newline*
        statement-list

In terms of execution:

- The expression of the if clause is evaluated. If it is true, then the associated statement-list is executed, and we skip through the rest of the if/else block
- Otherwise, we iterate through the expressions in the elseifs until we hit one that is true. At that point, the associated statement-list is executed, and we skip through the rest of the if/else block.
- If none evaluate to true, then we evaluate the statement-list associated with the else clause. If none exist, then we continue to the next statement after the if/else statement

Example:

```
if 1 == 0 then
    dosomething()
elseif 1 == 1 then
    dosomething()
end if
```

Regular Expression Clauses:

A special kind of expression is allowed within the conditional clauses for if statements. These are the regexp expressions, and they are not allowed anywhere else othern than within an if statement

The syntax is as follows:

regexp_match:
    expr ~= expr -> vdecl list

Both expressions here must be of type string. The first expression represents the string to check, and the second represents the regular expression to match against. To the right of the arrow are variable declarations that will be mapped to the clauses found in the regular expression.

The processing will go as follows:

- Regexp syntax is the one that is used from the C regex.h library - http://www.regular-expressions.info/posixbrackets.html
- If the regular expression matches, then the groups of the matched expression will be searched. Otherwise, the expression returns false and the code continues to the next part of the if/else
- The compiler will attempt to bind the variables declared in *vdecl list* with the groups found in left-to-right order.
    - Conversions will be done from strings to int or enums or strings, depending on the type of variable declared
- If the binding to the variable is not successful, whether from having differing variable list lists or mismatched types, then the statement will still return false and proceed to the next part of the if/else

3.4.6 Loops

while loops and for loops will be supported to facilitate repeated invocations of code

while loops:

while *expression* do \newline
    statement-list
end while \newline

The expression must return a boolean value.
Order of execution:

1. The *expression* will be evaluated.
2. If true is returned, the statement-list will be executed, after which we go back to step 1
3. Otherwise, we will exit from the while-loop

for loops:

for *var-assignment*; *expression; expression* do \newline
    statement-list
end for \newline

The for-loop executes as follows.

- Step 1: The first var-assignment is executed.

- Step 2: The expression is evaluated, and must return a boolean value
    - If the value is false, the loop exits
    - If the value is true, the statement-list is executed, and the second expression is invoked. Then repeat step 2

3.5 Programs

The above elements will be combined to form a program. Here are the details on what the final program will look like.

3.5.1 Overall Structure

The program will consist of a series of either function definitions or enum definitions

- We have specified enum definitions earlier; function definitions will be specified in the next section
- Functions must each have distinct names
- There must be 1 function defined with the name “main” and with no arguments. This will be the entry point to the program
- The function names, the enum type names, and the enum value names are all identifiers that will have global scope in the program. No local variables can use these same names
- The order in which the functions or enums are defined does not matter, i.e. a function can call another function that is defined later in the code

3.5.2 Function Definition

Functions are the containers for statements that will be executed, i.e. we have defined individual statements in the earlier section to define what can be done, and a function is the structure that organizes the flow of execution of these statements

The structure is as follows

function *type-specifier identifier* (*type-argument-list*)
    statement-list
end function

Notes:

- The type-argument-list is optional. The parenthesis are still needed in that case
- All variables declared within a function will have scope within the entire function
    - This includes variables declared in the type-argument-list and in standalone var-declaration statements
- The flow of control of a function must return a value using the “return” statement

- If it is detected at runtime that the end of the function has been reached without a return statement, then an error will be raised and the program will halt execution

Example:

```
function int dosomething ( int a1 )
    int b1 = a1 + 5
    print(b1)
    return 0
end function
```

3.6 Built-in Functions

The program provides some built-in functions to help w/ the application development. Invoking these would involve the same syntax for calling as defined in Section 3.3.8

- int print(String str) - prints out the text in str to stdout without a newline appended
- int printLine(String str) - prints out the text in str to stdout with a newline appended
- String readLine() - reads a line from stdin (terminated by the enter-key) and returns the value to the string

The following methods are generated for each enum that is defined (for these examples, we will refer to the enum name as enumname

- int printBoard_enumname(board<enumname> myboard) - prints the row x col contents of the board to stdout
- int printRow_enumname(board<enumname> myboard, int n) - prints the nth row of the board to stdout. This is used in case we want a finer grain of what to print out
- enumname parseEnum_enumname(string) - returns the enum whose id ref matches the input string

3.7 Formal Definition

4. Project Plan

Timelines and Schedule

As I am a CVN student, I worked alone on this project, and so there were no considerations in

terms of splitting up work or scheduling across team members.

At a high level, I carried out my work in the following manner:

- Start of semester
    - Project Proposal
- Mid-semester
    - Language Reference Manual draft
- End of semester (approximately a month before due date)
    - Development environment setup and microc example setup
    - Parser implementation for basic features
        - functions, enums, variables, math operations, if-statements, printing output, using existing C-like conventions for white-space and statement separators
    - Test harness implementation for parser
    - Compiler implementation for basic features
    - Parser/Compiler/Test (PCT) implementation for advanced features
        - board literals and operations, built-in functions for boards, regular expression handling
    - Test harness improvements to allow for input *and* output verification in code
    - PCT implementation for final cleanups
        - newlines for line separators, for/while loops, checkers example, and bug fixes found upon final testing

Development Environment

Operating System: Ubuntu 13.04
OCaml Version: 3.12.1 (from Ubuntu repositories)
Build: using ocamlopt as the code to execute system commands required this
Development IDE: Eclipse with the OCaml Development Tools feature

- http://ocamldt.free.fr/
- IDE was *not* used for compilation - I still relied on command-line and make for that
- IDE helped with:
    - Syntax highlighting
    - Automatic code-formatting and indentation
        - e.g. during typing and hitting the enter key, the code being edited would be indented and aligned with reference to its surrounding code
        - This was very helpful as I did not know the OCaml formatting conventions, and it was nice for the IDE to decide this for me
    - Window/file management

Coding Conventions for OCaml: Used the IDE's suggestions
Version Control: Git + Dropbox

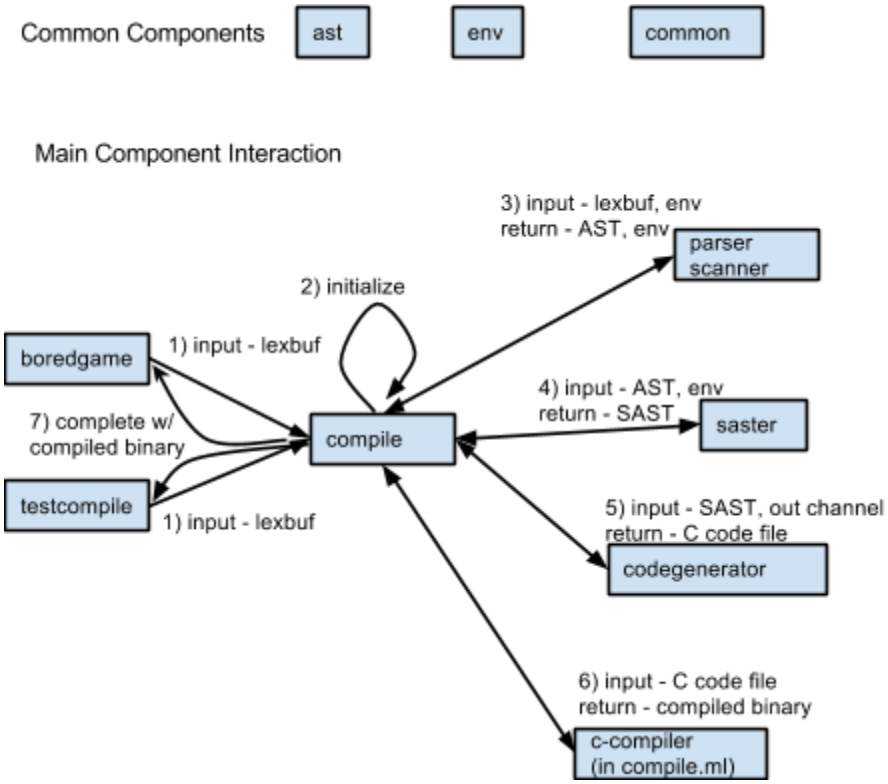
- Used Git via Eclipse to commit to a local Git repository so that changes can be committed and viewed easily

- Git repository was backed by Dropbox in case my local machine failed (I had Dropbox on my machine anyway, so it was easy to leverage)

5. Design

5.1 Architecture

5.1.1 Component Diagram



5.1.2 Component And Interaction Details

The diagram above shows the components required to go from some input buffer to a compiled binary. Each component name above corresponds to a file in the provided source code of the project.

Here are the details

Common and Core Components

The core component that ties everything together is compile. This takes in the input from the user and handles calling the various other components to get its work done.

We should also highlight the common components used by most of the modules:

- ast - Contains the types used to represent the abstract syntax tree (AST) and the semantically-analyzed syntax tree (SAST). More detail on the SAST in the sections below
- env - Contains the code to access the symbol and compiler environment used to compile the program.
    - This includes:
        - The symbol table
        - The function and type metadata
        - The current function name that is being processed
        - Lookup methods to facilitate accessing and adding to the environment
    - Note that the symbol table is just a simple map.
        - Originally, I had planned on having nested symbol tables as to represent the different scopes that arise when creating blocks in the code
        - However, the functional nature of OCaml allowed me to achieve the same effect without needing a child symbol table. More details on this in the code
- common - Contains some general utility methods

Part 1) Reading the Input

In this part, we show the compile component reading its input as a lexbuf from its clients - this is done via the compile_program_from_* methods. This is done generically as we may have different sources of input

We used 3 different sources during development via these 2 components:

- boredgame component
    - This takes in -f <programname> or -c <programname> as args. A binary called <programname>.out will result
    - -f reads in a file <programname>.game as the input
    - -c reads in the input from stdin
- testcompile component
    - This was used for unit testing the application.
    - Here, the input is a string that is written directly inside the test code

Part 2) Initialization

Here, we do some basic setup like creating the initial environment and adding some built-in function definitions (e.g. for printLine) to the env

Part 3) Parsing the Lexbuf into an AST and environment

Using parser.mly and scanner.mll, we parse the lexbuf into the AST.
After this step, we will enrich the environment to add the enum and function information, as well

as the built-in functions for the enums (e.g. printBoard_*)

Part 4) Converting the AST to a SAST

Now that we have the AST, we need to do the semantic analysis using the saster component. Tasks will include:

- Checking that expressions used by statements and other expressions have the correct type
- Doing symbol lookups and storing the symbols to the symbol tables
- Assigning the SAST-ed expression a machine address for the code generation (will expound on this in the “Runtime Environment” section

We will throw CompileExceptions here in case we detect any violations of the rules set forth in the LRM.

Part 5) Generating the C code from the SAST

With the SAST defined, we can now generate the C code. Note that we won’t need the environment here, as we’ve already saved the relevant pieces of information from it.

Generally, the print_sast_expr method will create the code to calculate values for the expression if the original machine code itself did not suffice (e.g. we don’t need any extra calculations for string or integer literals), whereas print_sast_stmt takes care of the various control statements.

These methods take in the out variable, which is used by fprintf to write the value. The only current input so far is the file channel to write out to a temp C file.

We will also throw CompileExceptions here in case we detect something amiss that we could not detect before without duplication of code.

Part 6) Calling the C compile to create the binary

We directly invoke gcc to do the compilation to the binary. We will throw a CriticalCompileException if the compile fails, as ideally we should catch all potential compile errors during steps 4 and 5. Note that this won’t mean we’d guard fully against runtime exceptions, but at the very least, our code should compile in C.

Part 7) Done

And that is it!

5.2 Runtime Environment

Here are some details on the C code generation.

5.2.1 Code Generation Strategy using register-based-like IR

For this project, we decided to compile to machine code by first translating our input code to C, and then compiling the C code. We could not do a simple translation directly from the AST, as some of the constructs we tried to model could not be taken care of by an equivalent number of lines in C (e.g. representing the board construct in BoredGame took many lines in C to create the struct and fill in the arguments, or representing multiple if conditions and condition clauses).

As a result, we had to denote a memory address to store the value of an expression, and given that address, the code generator can use that value to write out the code. As we had to define a C variable, we also needed to define the type, which is why sast_expr in the code also stores the expected type of the expression.

Hence, we had a choice between stack-based IR and register-based IR. We went with register-based IR as it was easier to translate from our code to a 3-address based code in this manner. One of the problems w/ register-based IR though is that we are limited in the number of registers. However, as we are representing this in C, we can think of local variables as registers (i.e. temporary storage of variables that the function can use), and as we can name variables as we choose, we essentially have a very number of “registers” to work with! Hence, we describe this as “register-based--*like*” IR.

5.2.2 Type System w/ Generics/Templates

We mentioned earlier that the environment component stored program metadata such as types.

Of particular note is that we store not just basic types like int, string, and boolean, but genericized or templated types (depending on if you like the Java description or the C++ description) like board<T> and enum<T>. This was required as we wanted to form behaviors on certain categories of types, while allowing parts of it to be parameterized. For example, the board concepts works generally as a 2D array representation of some information, but depending on the variable declaration, we wanted to limit it to only a certain subtype of references to use.

In the compile code, this was represented via the type_id type, which was either a SimpleTypeId or a GenericTypeId (which itself referred to its subtype). Generally, most comparisons of type behavior were done on a string comparison of either the main type name or the full genericized type name, which possibly would not scale well for more complex uses, but which sufficed for our model so far.

5.2.3 Memory Usage and Array Representation

As we were dealing with C code, we had to guard against improper memory leaks. As to keep

things simple for the users of the language, we looked to keep everything on the stack so that clients did not have to keep track of memory or worry about garbage collection.

This meant a few trade-offs:

- The 2D array for the boards was actually represented as a 1D array in C, as it proved difficult to manipulate the elements of a 2D array on the stack with the way I was code generating.
- Strings were given a max length (defined as 1024) and the full char array for it was always allocated on the stack. With more time to develop, we could have been smarter to do proper mallocs and callocs, but given the time and the relatively small game programs to be written in the first iteration of the language, I deferred on this optimization to start.
- One area where we do still have a memory leak to be resolved is reading in of input from users. I could not get this to work directly off stack data - this will have to be a TODO item

5.2.4 Regular Expressions

The regular expression feature in BoredGame is implemented under the hood in C using the regex.h library. We were able to work this on the stack as well, using the strategy of fixed size char arrays mentioned above.

5.3 Error Recovery

Error detection for the BoredGame language is handled in the various components mentioned earlier (mostly in the saster and codegenerator components), but the recovery and error display mechanisms are currently rather light.

The main features added for this is to be able to show the location of the error code and the surrounding text in case. See the scanner component and the parse_program_from_lexbuf method in the compile component for more information.

6. Test Plan

6.1 Test Case Setup and Automation

The main structure of the tests was to pass in an example program and then to verify its output. Its output could have been:

- For positive-result tests - the text that the compiled binary would print out upon execution
- For negative-result tests - just the expectation that a CompileException was raised. If not, an error would be thrown by the tests

The setup and execution of the tests were done directly inside the OCaml code using a separate testcompile.ml module. I preferred doing this in code as it resembled Java JUnit tests that I am used to, and there was one less hop from code compilation to test execution, thus saving time

as multiple iterations of tests would occur. Another subtle benefit is that some of the early iterations of the test harness code were reusable as components in the regular code base.

The test harness would stop upon any test failure. Hence, running tests to ensure that the code still worked proved to be easily done in two steps:

- Running “make testcompile” to build the test
- Running “./testcompile.out”

6.2 Test Case Strategy and Examples

In terms of the strategy, I looked to test out both positive and negative use cases (e.g. cases where the feature in question worked fine, and cases where a parsing or compiling or semantic exception took place, and I expected the compiler to detect it).

I looked to have sets of tests for each of the features of the language as mentioned in the “Project Plan: Timelines and Schedule” section. Some of the tests would happen to combine some trivial cases, but generally, I tried to keep them separate.

Here are some quick examples showing both positive and negative test cases. The full test suite is

```
test_progress_program_expect_exception "testBadMainWithArgs" "
function int main(int arg)
     return 0
end function
";


test_progress_program_expect_exception "testMismatchedFunctionAssignType" "
function int main()
     boolean myvar
     myvar = func2()
     return 0
end function
function int func2()
     return 1
end function
";

test_progress_program_compare "testBasicFunctionCallAndAssign" "
function int main()
     /// testing out comments work
     int myvar = func2()
     printLine(\"myvar=\" @ myvar)
     return 0
end function
function int func2()
```

```
        return 1
    end function
    " "myvar=1";

    test_progress_program_compare "testBasicStringConcat" "
    function int main()
        printLine(\"abc\" @ 123)
        string mystring
        mystring = \"abc\"
        printLine(mystring)
        mystring = \"abc\" @ \"do re mi\"
        printLine(mystring)
        printLine(\"abc\" @ \"do re mi\")
        mystring = \"abc\" @ 123
        printLine(mystring)
        return 0
    end function
    " "abc123
abc
abcdo re mi
abcdo re mi
abc123";
```

7. Thoughts on the Project

7.1 Ideas Learned

The purpose and techniques behind 3-address code generation were rammed home during this project! My first try at the code generation started off as a translation effort, but as I started to implement more complex features, I found that I could not do it easily (very prescient call by the professor to have me do it in C). Hence, I switched to the register-like IR code generation, and it helped the design of the compiler a lot.

The semantic analysis and code generation were also very interesting to do. I had not had the chance to do anything like it before in my projects at school and work, so it was enjoyable to do here, e.g. to work out the tree for the syntax and to know what kind of validations to put in

It also helps to have a good understanding of your functional domain when creating your language, as then you would know which features will be useful. In addition, your initial ideas may look good on paper initially, but after trying out some code, you may find reasons to adapt. For my change, I made such a change when I started off with a switch statement to work w/ the regular expression feature, and I decided to avoid the switch and just make the regular expressions a standard part of the if/else statement, which was more flexible. In addition, I had more ideas on stuff I can add to the program only after completing the core language implementation and then trying to implement the fully working Checkers example.

7.2 Lessons Learned / Advice to Other Teams

Use version control (e.g. Git, Subversion) to track your code. It is very helpful to be able to commit your code, and then be able to compare against your last committed code after making some changes.

Having an IDE like Eclipse helped immensely for the formatting support and version control integration, but I would not suggest to try to actually build with it. Using the command line and make file does the job.

OCaml can have a steep learning curve to start, so get used to some pain at first. One negative of not having a mature IDE support is that you cannot do things like refactor or diagnose problems easily. It can be very frustrating at first to get a cryptic OCaml error message and not know what to do with it. It maybe took a few days of coding in OCaml before I finally got the hang of it and was able to spend more time writing working code instead of figuring out what the error message that the OCaml compiler emitted meant.

Start early! The compressed summer schedule did not help things, but it would have been better for me to start coding earlier, especially to get over the initial OCaml learning curve.

Set goals for what you want to get out of the project and which features to implement; get the professor's feedback if necessary. It was hard for me to come up with an idea, so I used this mainly as an exercise to get practice in creating a language and doing new things. For example, the professor suggested having my language use line-breaks to separate statements; though that may not be the design decision I'd take if I had to create a language at work, it was an opportunity for me to get more practice at working on the scanner and parser, and learning is always valuable when taking a course.

Have a short-term and long-term plan for the features to implement. While I knew overall what I wanted to implement (the long-term plan), it helped to keep track of the next few items I wanted to take care of in the short-term, so that I can focus on only those goals, instead of having my mind wander to other goals.

8. Appendix

8.1 Example Program - Checkers

```
enum PieceType
	c, C, noPieceType
end enum

enum Player
	p1, p2, noPlayer
end enum

enum Piece
	r, R, b, B, o
end enum


function int main()
	printLine("Welcome to Checkers!")

	board<Piece> myboard = {
		r o r o r o r o
		o r o r o r o r
		r o r o r o r o
		o o o o o o o o
		o o o o o o o o
		o b o b o b o b
		b o b o b o b o
		o b o b o b o b
		}

	Player turn = p1

	while true do
		Player winner = gameover(myboard, turn)
		if winner == noPlayer then
			printLine("")
			printLine("It is Player " @ turn @ "'s turn")
			printCheckersBoard(myboard)
			printLine("")
			printLine("Player " @ turn @ ", please enter your move in the format
of [a-h][1-8] [a-h][1-8], e.g. a8 b7")

			if eval_move(readLine(), myboard, turn) then
				turn = getOtherPlayer(turn)
			else
				printLine("Invalid move - please try again")
			end if
		else
```

```
                    printLine("Player " @ winner @ " has won. Congratulations!\nExiting
the program now")
                    return 0
             end if
      end while
end function


/* We customize the printing here so that we can see the board coordinates
Maybe we can add something like this as a helper method later on in
the language itself */
function int printCheckersBoard(board<Piece> myboard)
      printLine("    a b c d e f g h")
      printLine("    ---------------")
      int i
      for i = 1; i <= myboard.rowlength; i = i + 1 do
            print(i @ " | ")
            printBoardRow_Piece(myboard, i)
      end for
      return 0
end function


// We have this pattern-matching switch statement to allow for various input // moves,
e.g. for chess 0-0-0 or 0-0 for castling notation, which is
// different from the other move notations that are of the form
// ([a-h])([0-9]])-([a-h])([0-9]])
function boolean eval_move(string input, board<Piece> myboard, Player player)
      int srcrow
      int srccol
      int tgtrow
      int tgtcol

      if input ~= "([a-h])([1-8]) ([a-h])([1-8])" -> string s_srccol, int s_srcrow,
string s_tgtcol, int s_tgtrow then
            srcrow = s_srcrow
            srccol = ind_to_number(s_srccol)
            tgtrow = s_tgtrow
            tgtcol = ind_to_number(s_tgtcol)
      else
            printLine("Invalid move input format")
            return false
      end if

      printLine("Read move " @ srcrow @ srccol @ "-" @ tgtrow @ tgtcol)

      return processMove(myboard, player, srcrow, srccol, tgtrow, tgtcol)
end function
```

```
function boolean processMove(board<Piece> myboard, Player player, int srcrow, int srccol,
int tgtrow, int tgtcol)
	Piece curpiece = myboard[srcrow,srccol]
	if player != getPiecePlayer(curpiece) then
		printLine("Invalid move - player must own the piece")
		return false
	end if

	Piece targetpiece = myboard[tgtrow,tgtcol]
	if getPiecePlayer(targetpiece) != noPlayer then
		printLine("Invalid move - target must be unoccupied")
		return false
	end if

	int rowmove = srcrow - tgtrow
	int colmove = srccol - tgtcol

	if abs(rowmove) != abs(colmove) then
		printLine("Not a diagonal move")
		return false
	end if

	if getPieceType(curpiece) == c then
		if player == p1 and rowmove < 0 then
			printLine("Player " @ player @ " can only move up the board with a
small piece")
			return false
		elseif player == p2 and rowmove > 0 then
			printLine("Player " @ player @ " can only move down the board with a
small piece")
			return false
		end if
	end	if

	if abs(rowmove) == 0 then
		printLine("Small piece can only move 1 space diagonally. A move of " @
abs(rowmove) @ " was tried")
		return false
	elseif abs(rowmove) == 1 then
		myboard[srcrow, srccol] = o

		if (tgtrow == 1 or tgtrow == 8) and getPieceType(curpiece) == c then
			myboard[tgtrow, tgtcol] = promote(curpiece)
		else
			myboard[tgtrow, tgtcol] = curpiece
		end if
		return true
	elseif abs(rowmove) == 2 then
		int intermRow = (srcrow + tgtrow) / 2
		int intermCol = (srccol + tgtcol) / 2
```

```
            if getPiecePlayer(myboard[intermRow,intermCol]) != getOtherPlayer(player)
then
                  printLine("Invalid move - to move 2 spaces, must jump over the
opponent's piece")
                  return false
            end if

            myboard[intermRow, intermCol] = o
            myboard[srcrow, srccol] = o

            if (tgtrow == 1 or tgtrow == 8) and getPieceType(curpiece) == c then
                  myboard[tgtrow, tgtcol] = promote(curpiece)
            else
                  myboard[tgtrow, tgtcol] = curpiece
            end if
            return true
      else
            printLine("Pieces cannot move more than 2 spaces diagonally. A move of " @
abs(rowmove) @ " was tried")
            return false
      end if
end function


function int abs(int num)
      if num > 0 then
            return num
      else
            return 0 - num
      end if
end function


// Would have preferred a more succinct way to represent this in the language,
// e.g. some kind of mapping syntax (x => X), but de-scoping this for now
function Player getPiecePlayer(Piece piece)
      if piece == b or piece == B then
            return p1
      elseif piece == r or piece == R then
            return p2
      else
            return noPlayer
      end if
end function


function Player getOtherPlayer(Player player)
      if player == p1 then
            return p2
```

```
	elseif player == p2 then
		return p1
	else
		fail("Invalid case - player must be p1 or p2")
	end if
end function


// Would have preferred a more succinct way to represent this in the language,
// e.g. some kind of mapping syntax (x => X), but de-scoping this for now
function PieceType getPieceType(Piece piece)
	if piece == b or piece == r then
		return c
	elseif piece == B or piece == R then
		return C
	else
		return noPieceType
	end if
end function


function int ind_to_number(string input)
	if input == "a" then
		return 1
	elseif input == "b" then
		return 2
	elseif input == "c" then
		return 3
	elseif input == "d" then
		return 4
	elseif input == "e" then
		return 5
	elseif input == "f" then
		return 6
	elseif input == "g" then
		return 7
	elseif input == "h" then
		return 8
	end if

	fail("This condition should not reach")
end function


function Player gameover(board<Piece> myboard, Player player)
	int i
	int j
	int p1score = 0
	int p2score = 0
```

```
	for i = 1; i <= myboard.rowlength; i = i + 1 do
		for j = 1; j <= myboard.collength; j = j + 1 do
			Piece piece = myboard[i,j]
			if (getPiecePlayer(piece) == p1) then
				p1score = p1score + 1
			elseif (getPiecePlayer(piece) == p2) then
				p2score = p2score + 2
			end if
		end for
	end for

	printLine("p1 " @ p1score @ " p2 " @ p2score)
	if p1score == 0 and p2score == 0 then
		fail("Invalid state - both scores cannot be zero")
	elseif p1score == 0 then
		return p1
	elseif p2score == 0 then
		return p2
	else
		return noPlayer
	end if
end function


function Piece promote(Piece p)
	if p == r then
		return R
	elseif p == b then
		return B
	else
		return p
	end if
end function
```

8.2 testcompile code / test case examples

```
open Ast
open Printf
open Compile
;;
module StringMap = Map.Make(String);;

(* Obtained from:
http://stackoverflow.com/questions/2214970/collecting-the-output-of-an-external-command-u
sing-ocaml *)
let syscall_and_collect_output = fun command ->
  let chan = Unix.open_process_in command in
  let res = ref ([] : string list) in
```

```
  let rec process_otl_aux () =
    let e = input_line chan in
    res := e::!res;
    process_otl_aux() in
  try process_otl_aux ()
  with End_of_file ->
    let stat = Unix.close_process_in chan
    in (String.concat "\n" (List.rev !res)), stat;
;;


(* TODO delete this method - no longer needed *)
let test_progress_program program_name program_text =
  printf "Test case (%s) START\n" program_name;
  compile_program_from_string program_name program_text;
  printf "Test case (%s) SUCCESS\n" program_name;
;;

let test_progress_program_compare program_name program_text expected_result =
  printf "Test case (%s) START\n" program_name;
  compile_program_from_string program_name program_text;
  let program_output, _ = syscall_and_collect_output ("./" ^ program_name ^ ".out") in
  if program_output = expected_result
  then printf "Test case (%s) SUCCESS\n" program_name
  else
    (printf "Expected program output:\n[[[%s]]]\nbut was:\n[[[%s]]]\n" expected_result
program_output;
     raise(Failure("Mismatched expectation and actual output. See previous messages"))
    );
;;

let test_progress_program_expect_exception program_name program_text =
  try
    printf "Test case (%s) START\n" program_name;
    compile_program_from_string program_name program_text;
    raise (Failure ("Expecting to throw an exception in this case for " ^ program_name))
  with CompileException(_) -> printf "Test case (%s) SUCCESS: successfully caught the
expected exception!\n" program_name;
;;


let _ =
  printf "Starting Test Suite:\n";

  test_progress_program_expect_exception "testenum-duplicatevar" "
  enum myenum
       a, b, c, c
  end enum
  function int main()
  return 0
```

```
end function
";
test_progress_program_expect_exception "testenum-duplicatevar2" "
enum myenum
     a, b, dupe
end enum
enum myenum2
     dupe, e, f
end enum
function int main()
return 0
end function
";
test_progress_program_expect_exception "testenum-enum-var-conflict" "
enum myenum
     a, b, c, myenum2
end enum
enum myenum2
     d, e, f
end enum
function int main()
return 0
end function
";

test_progress_program_expect_exception "testMismatchedAssignment" "
enum myenum
     a, b, c
end enum
function int main()
  myenum e1
     e1 = 6
     return 0
end function
";


test_progress_program_expect_exception "testBadIfCondition" "
function int main()
     if 1 + 2 then
     end if
     return 0
end function
";

test_progress_program_expect_exception "testMismatchedReturnType" "
function int main()
     return true
end function
";
```

```
test_progress_program_expect_exception "testDupeFuncs" "
function int main()
     return 0
end function
function int main(int a)
     return 0
end function
";

test_progress_program_expect_exception "testMissingMain" "
function int notAMain()
     return 0
end function
";

test_progress_program_expect_exception "testBadMainWithArgs" "
function int main(int arg)
     return 0
end function
";


test_progress_program_expect_exception "testMismatchedFunctionAssignType" "
function int main()
     boolean myvar
     myvar = func2()
     return 0
end function
function int func2()
     return 1
end function
";

test_progress_program_compare "testBasicFunctionCallAndAssign" "
function int main()
     /// testing out comments work
     int myvar = func2()
     printLine(\"myvar=\" @ myvar)
     return 0
end function
function int func2()
     return 1
end function
" "myvar=1";

test_progress_program_compare "testBasicStringConcat" "
function int main()
     printLine(\"abc\" @ 123)
     string mystring
```

```
      mystring = \"abc\"
      printLine(mystring)
      mystring = \"abc\" @ \"do re mi\"
      printLine(mystring)
      printLine(\"abc\" @ \"do re mi\")
      mystring = \"abc\" @ 123
      printLine(mystring)
      return 0
  end function
  " "abc123
abc
abcdo re mi
abcdo re mi
abc123";

  test_progress_program_expect_exception "testBadConcatWrongTypes1" "
  function int main()
    boolean me = true
      printLine(me @ 123)
  end function
  ";

  test_progress_program_expect_exception "testBadConcatWrongTypes2" "
  function int main()
      printLine(123 @ 123)
  end function
  ";

  test_progress_program_expect_exception "testBadFunctionCallArgsMissing1" "
  function int main()
      me()
  end function

  function int me(int a)
      return 0
  end function
  ";

  test_progress_program_expect_exception "testBadFunctionCallArgsMissing2" "
  function int main()
      me(2)
  end function

  function int me()
      return 0
  end function
  ";

  test_progress_program_expect_exception "testBadFunctionCallArgsWrongType" "
  function int main()
```

```
        me(true)
    end function

    function int me(int a)
        return 0
    end function
    ";

    test_progress_program_compare "testOperatorAssociativity" "
    function int main()
        int myint = (6 + 14) * 20
        int myint2 = 6 + (14 * 20)
        int myint3 = 6 + 14 * 20
        printLine(\"test operator associativity \" @ myint @ \"-\" @ myint2 @ \"-\" @
myint3)
        return 0
    end function
    " "test operator associativity 400-286-286";

    test_progress_program_expect_exception "testOpTypeMismatch" "
    function int main()
        boolean comp = true and 1
        return 0
    end function
    ";



    test_progress_program_compare "testAndOr" "
    function int main()
        if andTrue(0) then
               printLine(\"Basic case1 success\")
        else
               printLine(\"Basic case1 failed\")
        end if
        if andFalse(0) then
               printLine(\"Basic case2 failed\")
        else
               printLine(\"Basic case2 success\")
        end if

        if andTrue(0) and andTrue(1) and andTrue(2) then
               printLine(\"case3 success\")
        else
               printLine(\"case3 failed\")
        end if

        if andTrue(0) and andTrue(1) and andFalse(2) then
               printLine(\"case4 failed\")
        else
```

```
	printLine(\"case4 success\")
end if

if andFalse(0) and andTrue(1) and andTrue(2) then
	printLine(\"case4.1 - verify lazy evalution failed\")
else
	printLine(\"case4.1 - verify lazy evalution success\")
end if

if orFalse(0) or orFalse(1) or orTrue(2) then
	printLine(\"case5 success\")
else
	printLine(\"case5 failed\")
end if

if orTrue(0) or orFalse(1) or orFalse(2) then
	printLine(\"case5.1 - verify lazy evalution success\")
else
	printLine(\"case5.1 - verify lazy evalution failed\")
end if

if orFalse(0) or orFalse(1) or orFalse(2) then
	printLine(\"case6 failed\")
else
	printLine(\"case6 success\")
end if

if orFalse(0) or andTrue(1) and andTrue(2) then
	printLine(\"case 7 ambiguous true\")
else
	printLine(\"case 7 ambiguous false\")
end if

if (orFalse(0) or andTrue(1)) and andTrue(2) then
	printLine(\"case 7.1 success\")
else
	printLine(\"case 7.1 failed\")
end if

if orFalse(0) or (andTrue(1) and andTrue(2)) then
	printLine(\"case 7.2 success\")
else
	printLine(\"case 7.2 failed\")
end if

if andTrue(0) and andTrue(1) or orFalse(2) then
	printLine(\"case 8 ambiguous true\")
else
	printLine(\"case 8 ambiguous false\")
end if
```

```
      if (andTrue(0) and andTrue(1)) or orFalse(2) then
            printLine(\"case 8.1 success\")
      else
            printLine(\"case 8.1 failed\")
      end if

      if andTrue(0) and (andTrue(1) or orFalse(2)) then
            printLine(\"case 8.2 success\")
      else
            printLine(\"case 8.2 failed\")
      end if

      return 0
  end function

  function boolean andTrue(int a)
      printLine(\"andTrue\" @ a)
      return true
  end function
  function boolean andFalse(int a)
      printLine(\"andFalse\" @ a)
      return false
  end function
  function boolean orTrue(int a)
      printLine(\"orTrue\" @ a)
      return true
  end function
  function boolean orFalse(int a)
      printLine(\"orFalse\" @ a)
      return false
  end function
      " "andTrue0
Basic case1 success
andFalse0
Basic case2 success
andTrue0
andTrue1
andTrue2
case3 success
andTrue0
andTrue1
andFalse2
case4 success
andFalse0
case4.1 - verify lazy evalution success
orFalse0
orFalse1
orTrue2
case5 success
```

```
orTrue0
case5.1 - verify lazy evalution success
orFalse0
orFalse1
orFalse2
case6 success
orFalse0
andTrue1
andTrue2
case 7 ambiguous true
orFalse0
andTrue1
andTrue2
case 7.1 success
orFalse0
andTrue1
andTrue2
case 7.2 success
andTrue0
andTrue1
case 8 ambiguous true
andTrue0
andTrue1
case 8.1 success
andTrue0
andTrue1
case 8.2 success";


  test_progress_program_compare "testIfElse" "
  function int main()
       /* Ensure that only the first valid path is taken for these tests
       (i.e. do not process subsequent paths even if the condition is successful */
       if 1 == 1 then
              printLine (\"Case 1 is good!\")
       elseif 1 == 1 then
              printLine(\"Don't print this!\")
       elseif 1 == 1 then
              printLine(\"Don't print this!\")
       else
              printLine(\"Don't print this!\")
       end if

       if false then
              printLine(\"Don't print this!\")
       elseif true then
              printLine (\"Case 2 is good!\")
       elseif true then
              printLine(\"Don't print this!\")
       else
```

```
            printLine(\"Don't print this!\")
        end if

        if 1 == 555 then
            printLine(\"Don't print this!\")
        elseif 1 == 555 then
            printLine(\"Don't print this!\")
        elseif 1 == 1 then
            printLine (\"Case 3 is good!\")
        else
            printLine(\"Don't print this!\")
        end if

        if 1 == 555 then
            printLine(\"Don't print this!\")
        elseif 1 == 555 then
            printLine(\"Don't print this!\")
        elseif 1 == 55 then
            printLine(\"Don't print this!\")
        else
            printLine (\"Case 4 is good!\")
        end if

        return 0
    end function
    " "Case 1 is good!
Case 2 is good!
Case 3 is good!
Case 4 is good!";

    test_progress_program_expect_exception "testBoardWrongRowColCount" "
    enum myenum
        a, b, c
    end enum
    enum myenum2
        d, e, f
    end enum
    function int main()
        board<myenum> myboard
        myboard = {
            a a a a
            b b
            c c c c
        }
        /* todo validate the enum types */
        board<myenum> myotherboard
    end function
    ";
```

```
  test_progress_program_compare "testRegexp" "
  enum myenum
       aa, bb, cc
  end enum
  function int main()
    string input = \"___ abc123def ___\"

    if input ~= \"[a-z]*([0-9]+)([a-z]*)\" -> string inputA, string inputB then
       printLine(\"#1 I made it in with these values \" @ inputA @ \"-\" @ inputB)
       else
              printLine(\"I did not make it \")
    end if

    if input ~= \"[0-9]+([a-z]*)\" -> string inputA then
       printLine(\"#2 My string value is \" @ inputA)
       else
              printLine(\"I did not make it \")
    end if

    if input ~= \"[5-9]+([a-z]*)\" -> string inputA then
              printLine(\"Should not get to this point \")
       elseif input ~= \"([0-9]+)([a-z]*)\" -> int numberA, string inputB then
       printLine(\"#3 Made it with this value \" @ (numberA + 100) @ \"-\" @ inputB)
    end if

    return 0
  end function

  " "#1 I made it in with these values 123-def
#2 My string value is def
#3 Made it with this value 223-def";


    test_progress_program_compare "testLogic" "
  enum myenum
       aa, bb, cc
  end enum
  function int main()
    int int1 = 5
    string str1 = \"abcd\"
       printLine(\"equality int-\" @ (int1 == 5))
       printLine(\"equality int false-\" @ (int1 == 6))
       printLine(\"inequality int-\" @ (int1 != 6))
       printLine(\"equality string const-\" @ (str1 == \"abcd\"))
       printLine(\"equality string const false-\" @ (str1 == \"zzzzabcd\"))
       printLine(\"equality string dynamic-\" @ (str1 == (\"ab\" @ \"cd\")))
       printLine(\"inequality string dynamic-\" @ (str1 != (\"zzzab\" @ \"cd\")))
       return 0
  end function
```

```
  " "equality int-true
equality int false-false
inequality int-true
equality string const-true
equality string const false-false
equality string dynamic-true
inequality string dynamic-true";



  test_progress_program_compare "testWhile" "
  function int main()
    int i = 0
    while i < 5 do
       int sameattrname = 999
       print(i @ \"-\" @ sameattrname @ \" \")
       i = i + 1
    end while

       i = 0
    while i < 0 do
       print(\"This should never get called\")
    end while

       printLine(\"!\")

       i = 0
    while i < 3 do
              int j
              j = 0
       while j < 2 do
              print(\"j\")
              j = j + 1
       end while

       /* show that the variable scope is kept within the block */
       int sameattrname
       sameattrname = 333
       print(i @ \"-\" @ sameattrname @ \" \")
       i = i + 1
    end while

       return 0
  end function
  " "0-999 1-999 2-999 3-999 4-999 !
jj0-333 jj1-333 jj2-333 ";

  test_progress_program_compare "testFor" "
  function int main()
```

```
      int i
    for i = 0; i < 5; i = i + 1 do
      int sameattrname = 999
      print(i @ \"-\" @ sameattrname @ \" \")
    end for
    for i = 0; i < 0; i = i + 1 do
      print(\"This should never get called\")
    end for

      printLine(\"!\")

    for i = 0; i < 3; i = i + 1 do
      int j
          for j = 0; j < 2; j = j + 1 do
            print(\"j\")
      end for

      /* show that the variable scope is kept within the block */
      int sameattrname
      sameattrname = 333
      print(i @ \"-\" @ sameattrname @ \" \")
    end for

      return 0
  end function
  " "0-999 1-999 2-999 3-999 4-999 !
jj0-333 jj1-333 jj2-333 ";

  test_progress_program_expect_exception "testLoopsNestedScopeVarFail" "
  function int main()
    int i = 0
    int j = 0
    while i < 5 do
      int sameattrname
      sameattrname = 999
      while j > 5 do
             int sameattrname /* this should throw a compile exception */
             sameattrname = 998
      end while
    end while
      return 0
  end function
  ";

  test_progress_program_compare "testBoardDiffSize" "
  enum Piece
      r, b, B, R, o
  end enum

  function int main()
```

```
      board<Piece> myboard2 = {
            r o r o r o r o r
            o r o r o r o r o
            r o r o r o r o r
            o o o o o o o o o
            o o o o o o o o o
            o b o b o b o b o
            b o b o b o b o b
            o b o b o b o b o
            }
      printBoard_Piece(myboard2)
      return 0
  end function
  " "r o r o r o r o r
o r o r o r o r o
r o r o r o r o r
o o o o o o o o o
o o o o o o o o o
o b o b o b o b o
b o b o b o b o b
o b o b o b o b o ";


  test_progress_program_compare "testBoardAndFuncCalls" "
  enum myenum
      aa, bb, cc
  end enum
  enum myenum2
      dd, ee, ff
  end enum

  function int main()
      board<myenum> myboard = {
            aa aa aa bb
            bb bb bb cc
            cc cc cc aa
      }
      printLine(\"Board 1!\")
      printBoard_myenum(myboard)

      board<myenum2> myotherboard = {
            dd dd dd ee
            ee ee ee ff
            ff ff ff dd
      }
      printLine(\"Board 2!\")
      printBoardWithMyFunction(myotherboard, 5, 6)

      myenum testvar = myboard[1, 1]
      printLine(\"Board has \" @ myboard.rowlength @ \" rows and \" @ myboard.collength
```

```
@ \" cols\")
       printLine(\"Hello my enum value is \" @ testvar @ \". That's it!\")
       printLine(\"Hello my enum value is \" @ myboard[1, 1] @ \". That's it!\")
       myboard[1, 1] = bb
       printLine(\"Hello my enum value is \" @ myboard[1, 1] @ \". That's it!\")

       myenum2 myreadenum = parseEnum_myenum2(\"ee\")
       printLine(\"I just read in \" @ myreadenum)

       return 0
  end function
  function int printBoardWithMyFunction(board<myenum2> myboard, int var1, int var2)
       printLine(\"My new function with \" @ var1 @ \" and \" @ var2)
       printBoard_myenum2(myboard)
       return 0
  end function
  " "Board 1!
aa aa aa bb
bb bb bb cc
cc cc cc aa
Board 2!
My new function with 5 and 6
dd dd dd ee
ee ee ee ff
ff ff ff dd
Board has 3 rows and 4 cols
Hello my enum value is aa. That's it!
Hello my enum value is aa. That's it!
Hello my enum value is bb. That's it!
I just read in ee";

  printf "Finished all test cases - we have a success!\n";
;;
```

8.3 Full Source Code Listings

```
(********************* ast.ml *********************)
open Printf

exception CompileException of string;;
exception ParseException of string;;
exception CriticalCompileException of string;;



type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | Concat |
And | Or
;;

type type_id =
```

```
  | SimpleTypeId of string
  | GenericTypeId of string * type_id
;;

type var_decl = { vtype : type_id; vname : string; }
;;

type expr =
  IntLiteral of int
  | StringLiteral of string
  | BooleanLiteral of bool
  | BoardLiteral of string list list
  | Id of string
  | Binop of expr * op * expr
  | MatrixOp of expr * expr * expr
  | ClassOp of expr * string
  | Assign of expr * expr
  | Call of string * expr list
  | RegexpMatcher of expr * expr * var_decl list  (* both expressions should evaluate to
strings *)
;;

type stmt =
  Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * (expr * stmt) list * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | VarDecl of var_decl
  | VarDeclAndAssign of var_decl * expr
  | NoOpStmt
;;

type func_decl =
  {rettype : type_id;
   fname : string;
   params : var_decl list;
   body : stmt list; (* TODO make the body a stmt so that we can limit it to be a block
early on *)
  }
;;

type enum_decl = {
    ename : string;
    ids : string list;
  }
;;
```

```
type program = enum_decl list * func_decl list
;;



(* Corresponds to a reference to be used in the 3-address code generation
   The code_name would represent some snippet of C (whether a variable name
   or a literal) that can directly be plugged in anywhere
 *)
type reference =
  {ref_type : type_id;
   code_name : string;
  }
;;


(***********
 * Semantically-analyzed syntax tree (i.e. SAST data structures)
 *)
type sast_expr = { e : sast_expr_detail; etype : type_id; eref : string }
and sast_expr_detail =
  SastIntLiteral of int
  | SastStringLiteral of string
  | SastBooleanLiteral of bool
  | SastBoardLiteral of int * int * sast_expr list list
  | SastId of string
  | SastMatrixOp of sast_expr * sast_expr * sast_expr
  | SastBinop of sast_expr * op * sast_expr
  | SastClassOp of sast_expr * string
  | SastAssign of sast_expr
  | SastCall of string * sast_expr list
  | SastRegexpMatcher of sast_expr * sast_expr * var_decl list * reference list  (* both
expressions should evaluate to strings *)
;;


type sast_stmt =
  SastBlock of sast_stmt list
  | SastExpr of sast_expr
  | SastReturn of sast_expr
  | SastIf of (sast_expr * sast_stmt) list * sast_stmt
  | SastFor of sast_expr * sast_expr * sast_expr * sast_stmt
  | SastWhile of sast_expr * sast_stmt
  | SastVarDecl of string * reference
  | SastVarDeclAndAssign of string * reference * sast_expr
  | SastNoOpStmt
;;
```

```
(******************** boredgame.ml ********************)
type action = Console | File

let _ =
  let action, filename =
    if Array.length Sys.argv > 1
    then
      (let input = List.assoc Sys.argv.(1) [ ("-c", Console); ("-f", File) ] in
       input, Sys.argv.(2)
      )
    else raise(Failure("Pass in -c for console or -f fileName for the file")) in
  match action with
    | Console ->
      let lexbuf = Lexing.from_channel stdin in
      Compile.compile_program_from_lexbuf filename lexbuf
    | File ->
      let lexbuf = Lexing.from_channel (open_in (filename ^ ".game")) in
      Compile.compile_program_from_lexbuf filename lexbuf
;;



(******************** codegenerator.ml ********************)
open Ast;;
open Printf;;
open Common;;


let type_to_codetype = function
  | SimpleTypeId(simp_type) ->
    (match simp_type with
      | "boolean" -> "bool"
      | "string" -> "const char*"
      | any -> any
    )
  | GenericTypeId(gentype, paramtype) ->
    (match gentype with
      | "board" -> "struct board"
      | "enum" -> "const char*"
      | _ -> raise(CompileException("No other generic type allowed other than board or
enum: " ^ gentype))
    )
;;


let op_to_string = function
  | Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
```

```
  | Div -> "/"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | Or -> "||"
  | And -> "&&"
  | Concat -> raise(CompileException("concat is not implemented via this methodology"))
;;



(* emits the format value to use for the printf function on the C side *)
let type_to_print_format_for_concat sexpr =
  if type_is_of_type "int" sexpr.etype then "%d", sexpr.eref
  else if type_is_of_type "string" sexpr.etype then "%s", sexpr.eref
  else if type_is_of_type "enum" sexpr.etype then "%s", sexpr.eref
  else if type_is_of_type "boolean" sexpr.etype then "%s", (sprintf "(%s ? \"true\" :
\"false\")" sexpr.eref)
  else raise(CompileException("No other type than string or int supported for
concatenation:" ^ (type_id_to_string sexpr.etype)));
;;



(*
   print out the 3-address code to generate the values for a particular expression
   You will note that some of the blocks won't do anything. This is because the eref
value itself already
   has the full informatio to represent the value (e.g. the literal in code, or the
variable name)

   For the others, it is the responsibility of this method to emit the code to set the
value for the
   sexpr.eref value that is passed into here
 *)
let rec print_sast_expr out sexpr =
  match sexpr.e with
    | SastIntLiteral(l) -> ();
    | SastStringLiteral(l) -> ();
    | SastBooleanLiteral(l) -> ();
    | SastBoardLiteral(numrows, numcols, board_elems) ->
      fprintf out "const char* %sArray[%d];\n" sexpr.eref (numrows * numcols);
      for row = 0 to numrows - 1 do
        for col = 0 to numcols - 1 do
          let elem = List.nth (List.nth board_elems row) col in
          fprintf out "%sArray[%d] = %s;\n" sexpr.eref (calc_matrix_to_array_index row
col numcols) elem.eref;
```

```
        done
      done;
      fprintf out "struct board %s;\n" sexpr.eref;
      fprintf out "%s.boardarray = %sArray;\n" sexpr.eref sexpr.eref;
      fprintf out "%s.rowlength = %d;\n" sexpr.eref numrows;
      fprintf out "%s.collength = %d;\n" sexpr.eref numcols;
    | SastId(id) -> ();
    | SastMatrixOp(board, row, col) -> ()  (* the code earlier already did the 3-address
code encompassing *)
    | SastClassOp (classexpr, member_name) ->
      if not (type_is_of_type "board" classexpr.etype)
      then raise(CompileException("Dot operator can only be used on classes (e.g.
board)"));
      (match member_name with
        | "rowlength" -> fprintf out "%s %s = %s.rowlength;\n" (type_to_codetype
sexpr.etype) sexpr.eref classexpr.eref;
        | "collength" -> fprintf out "%s %s = %s.collength;\n" (type_to_codetype
sexpr.etype) sexpr.eref classexpr.eref;
        | _ -> raise(CompileException("No other field names supported for the board
class"))
      )
    | SastRegexpMatcher(input, regexp, vars, var_refs) ->
      print_sast_expr out input;
      print_sast_expr out regexp;
      fprintf out "bool %s;\n" sexpr.eref;
      fprintf out "char %s_groups[MAX_MATCHING_GROUPS][MAX_STRING_SIZE];\n" sexpr.eref;
      List.iter
        (fun var_ref -> match (type_to_base_type var_ref.ref_type) with
          | "string" -> fprintf out "char* %s; // local regexp var\n" var_ref.code_name;
          | "int" -> fprintf out "int %s; // local regexp var\n" var_ref.code_name;
          | "enum" -> fprintf out "%s %s; // local regexp var\n" (type_to_codetype
var_ref.ref_type) var_ref.code_name;
          | _ -> raise(CompileException("Do not support binding regexp params to any
other type than string, int, enums"));
        ) var_refs;
      fprintf out "int %s_val = compare_regexp(%s, %s, %s_groups);\n" sexpr.eref
input.eref regexp.eref sexpr.eref;
      fprintf out "if (%s_val > 0) {\n" sexpr.eref;
      fprintf out "%s = true;\n" sexpr.eref;
      for refnum = 0 to (List.length var_refs) - 1 do
        let var_ref = List.nth var_refs refnum in
        match (type_to_base_type var_ref.ref_type) with
          | "string" -> fprintf out "%s = %s_groups[%d + 1];\n" var_ref.code_name
sexpr.eref refnum;
          | "int" -> fprintf out "%s = atoi(%s_groups[%d + 1]);\n" var_ref.code_name
sexpr.eref refnum;
          | "enum" -> fprintf out "%s = parseEnum_%s(%s_groups[%d + 1]);\n"
var_ref.code_name (type_id_to_string (type_to_generic_subtype var_ref.ref_type))
sexpr.eref refnum;
          | _ -> raise(CompileException("Do not support binding regexp params to any
```

```
other type than string, int, enums"));
      done;
      fprintf out "} else if (%s_val == 0) {\n" sexpr.eref;
      fprintf out "%s = false;\n" sexpr.eref;
      fprintf out "} else {\n";
      fprintf out "printf(\"Regexp did not compile. This is a runtime exception\");
exit(-1);\n";
      fprintf out "}\n";
    | SastBinop(e1, op, e2) ->
      print_sast_expr out e1;
      (match op with
        | Concat ->
          (print_sast_expr out e2;
           fprintf out "char %s[1001];\n" sexpr.eref;
           let e1printformat, e1val = type_to_print_format_for_concat e1 in
           let e2printformat, e2val = type_to_print_format_for_concat e2 in
           fprintf out "snprintf(%s, 1000, \"%s%s\", %s, %s);\n" sexpr.eref e1printformat
e2printformat e1val e2val
          );
        | Equal ->
          print_sast_expr out e2;
          if type_is_of_type "string" e1.etype
          then
            (fprintf out "int %sCmpVal = strcmp(%s, %s);\n" sexpr.eref e1.eref e2.eref;
             fprintf out "bool %s = %sCmpVal == 0;\n" sexpr.eref sexpr.eref;
            )
          else fprintf out "%s %s = %s %s %s;\n" (type_to_codetype sexpr.etype)
sexpr.eref e1.eref (op_to_string op) e2.eref;
        | Neq ->
          print_sast_expr out e2;
          if type_is_of_type "string" e1.etype
          then
            (fprintf out "int %sCmpVal = strcmp(%s, %s);\n" sexpr.eref e1.eref e2.eref;
             fprintf out "bool %s = %sCmpVal != 0;\n" sexpr.eref sexpr.eref;
            )
          else fprintf out "%s %s = %s %s %s;\n" (type_to_codetype sexpr.etype)
sexpr.eref e1.eref (op_to_string op) e2.eref;
        | And ->
          (* For And and Or, we take care to do lazy-evaluation, i.e. to ensure that we
only evaluate the code
             generated for the second expression if it is warranted. For And, we break
off early if
             the first arg is false, and for Or, we break off early if it is true *)
          fprintf out "%s %s;\n" (type_to_codetype sexpr.etype) sexpr.eref;
          fprintf out "if (%s == false) { %s = %s; } " e1.eref sexpr.eref e1.eref;
          fprintf out "else {\n";
          print_sast_expr out e2;
          fprintf out "%s = %s %s %s;\n" sexpr.eref e1.eref (op_to_string op) e2.eref;
          fprintf out "}\n";
        | Or ->
```

```
          fprintf out "%s %s;\n" (type_to_codetype sexpr.etype) sexpr.eref;
          fprintf out "if (%s == true) { %s = %s; } " e1.eref sexpr.eref e1.eref;
          fprintf out "else {\n";
          print_sast_expr out e2;
          fprintf out "%s = %s %s %s;\n" sexpr.eref e1.eref (op_to_string op) e2.eref;
          fprintf out "}\n";
        | _ ->
          print_sast_expr out e2;
          fprintf out "%s %s = %s %s %s;\n" (type_to_codetype sexpr.etype) sexpr.eref
e1.eref (op_to_string op) e2.eref;
      )
    | SastAssign(e) ->
      print_sast_expr out e;
      fprintf out "%s = %s;\n" sexpr.eref e.eref;
    | SastCall (id, exprs) ->
      let args = String.concat ", " (List.map (fun e -> e.eref) exprs) in
      List.iter (print_sast_expr out) exprs;
      fprintf out "%s %s = %s(%s);\n" (type_to_codetype sexpr.etype) sexpr.eref id args;
;;


let rec print_sast_stmt out input_stmt = match input_stmt with
  | SastNoOpStmt -> ()
  | SastBlock(stmts) ->
    fprintf out "{\n";
    List.iter (print_sast_stmt out) stmts;
    fprintf out "}\n";
  | SastVarDecl(vname, idref) ->
    fprintf out "%s %s;  // for var %s\n" (type_to_codetype idref.ref_type)
idref.code_name vname;
  | SastVarDeclAndAssign(vname, idref, expr) ->
    print_sast_expr out expr;
    fprintf out "%s %s = %s;  // for var %s\n" (type_to_codetype idref.ref_type)
idref.code_name expr.eref vname;
  | SastExpr(e) ->
    print_sast_expr out e;
  | SastReturn(e) ->
    print_sast_expr out e;
    fprintf out "return %s;\n" e.eref;
  | SastIf(conds, s2) ->
    let iflabel = get_next_tmp_id () in
    fprintf out "// Start of if-else block for %s\n" iflabel;
    List.iter
      (fun (expr, stmt) ->
        print_sast_expr out expr;
        fprintf out "if (%s) {" expr.eref;
        print_sast_stmt out stmt;
       fprintf out "goto %s;\n" iflabel;
        fprintf out "}\n";
      ) conds;
```

```
    print_sast_stmt out s2;
    fprintf out "%s:;\n" iflabel;
    fprintf out "// End of if-else block for %s\n" iflabel;
  | SastWhile(sexpr, sast_stmt) ->
    (* The order of operations of the while loop is modeled similarly to what was shown
in class,
       where the condition statement was placed "after" the body of the while in the
generated code,
       so that we have one less jump/goto method to execute *)
    let label = get_next_tmp_id () in
    fprintf out "goto %sCond;\n" label;
    fprintf out "%sStart:;\n" label;
    print_sast_stmt out sast_stmt;
    fprintf out "%sCond:;\n" label;
    print_sast_expr out sexpr;
    fprintf out "if (%s) {" sexpr.eref;
    fprintf out "goto %sStart;\n" label;
    fprintf out "}\n";
  | SastFor(e1, e2, e3, s) ->
    (* See the comment in the SastWhile block around how the condition/body of the for is
organized.
       We do it in a similar way here *)
    let label = get_next_tmp_id () in
    print_sast_expr out e1;
    fprintf out "goto %sCond;\n" label;
    fprintf out "%sStart:;\n" label;
    print_sast_stmt out s;
    print_sast_expr out e3;
    fprintf out "%sCond:;\n" label;
    print_sast_expr out e2;
    fprintf out "if (%s) {" e2.eref;
    fprintf out "goto %sStart;\n" label;
    fprintf out "}\n";
;;



(******************** common.ml *********************)
open Ast;;
open Printf;;


(* Stored as an array so that it can be mutable *)
let sast_counter = [| 0 |];;

(* This is to get a new symbol for use in the 3-address code generation,
   essentially the equivalent of getting new registers to store addresses in

   We use an array to store this so that we have a globally-accessed variable for
   this. It proved to be inconvenient at first to put this as a field on environment
```

```
   and so the global variable proved to be flexible. We can change this one day if
   we choose
 *)
let get_next_tmp_id () =
  sast_counter.(0) <- (sast_counter.(0) + 1);
  "tmp" ^ (string_of_int sast_counter.(0));
;;


(************
 * Type-helper methods
 *)
let type_type = SimpleTypeId("type");;
let function_type = SimpleTypeId("function");;
let string_type = SimpleTypeId("string");;
let int_type = SimpleTypeId("int");;
let boolean_type = SimpleTypeId("boolean");;

let rec type_id_to_string = function
  | SimpleTypeId(id) -> id;
  | GenericTypeId(gentype,paramtype) -> sprintf "%s<%s>" gentype (type_id_to_string
paramtype);
;;

let type_to_base_type t = match t with
  | SimpleTypeId(id) -> id;
  | GenericTypeId(gentype, _) -> gentype
;;

let type_is_of_type t2name t1 = (type_to_base_type t1) = t2name;
;;

let rec type_to_generic_subtype = function
  | SimpleTypeId(id) -> raise(CompileException("Cannot access generic subtype of
SimpleType"));
  | GenericTypeId(gentype,paramtype) -> paramtype;
;;




(****
 * The two methods for calculationg the array index.
 * One is for Ocaml to the code, and one is for the c code itself
 * We keep them both here so that folks can easily see the logic in both places
 *
 * Note - we store as a regulary array in C, and not a 2D array, as it proved to be much
harder
 * to get 2d arrays to work on the stack in C.
 * *****)
```

```
let calc_matrix_to_array_index row col numcols =
  row * numcols + col
;;

(* We use the array reference directly for thee reference, as opposed to generating a new
address via get_next_tmp_id,
   as I could not come up w/ a 3-address code representation for array references that
worked for
   both retrieval and storage. This way works out. In any case, the indexes themselves
are used in a 3-address-like manner
 *)
let calc_matrix_to_array_index_for_c_code board row col =
  (* Use %s - 1 here as we index matrices in the code by 1-index, whereas C does it
0-indexed *)
  sprintf "%s.boardarray[(%s - 1) * %s.collength + (%s - 1)]" board.eref row.eref
board.eref col.eref
;;



(******************** compile.ml *********************)
open Ast;;
open Printf;;
open Env;;
open Common;;
open Saster;;
open Codegenerator;;

module StringMap = Map.Make(String);;


let initialize_env_with_globals enums funcs =
  let env = Env.create_new () in
  (* register some default types *)
  let env = Env.add_type env "int" int_type in
  let env = Env.add_type env "string" string_type in
  let env = Env.add_type env "boolean" boolean_type in
  (* register some default functions *)
  let env = Env.add_func env {fname = "print"; rettype = int_type; params =
[{vtype=string_type; vname="str"}]; body = []} in
  let env = Env.add_func env {fname = "printLine"; rettype = int_type; params =
[{vtype=string_type; vname="str"}]; body = []} in
  let env = Env.add_func env {fname = "fail"; rettype = int_type; params =
[{vtype=string_type; vname="str"}]; body = []} in
  let env = Env.add_func env {fname = "readLine"; rettype = string_type; params = [];
body = []} in
  (* register the functions from our own code *)
  let env = (List.fold_left Env.add_enum) env enums in
  let env = (List.fold_left Env.add_func) env funcs in
  env
```

```
;;


let process_enum out env enum =
  List.iter
    (fun id ->
      let idref = (Env.lookup_symbol env id) in
      fprintf out "const char *%s = \"%s\";\n" idref.code_name id
    ) enum.ids
;;


let func_signature_for_prototype func =
  let var_strs = List.map (fun var -> sprintf "%s %s" (type_to_codetype var.vtype)
var.vname ) func.params in
  sprintf "%s %s(%s)" (type_to_codetype func.rettype) func.fname (String.concat ", "
var_strs);
;;


let func_signature env func =
  let var_strs = List.map
    (fun var ->
      let idref = Env.lookup_symbol env var.vname in
      sprintf "%s %s" (type_to_codetype var.vtype) idref.code_name
    ) func.params in
  sprintf "%s %s(%s)" (type_to_codetype func.rettype) func.fname (String.concat ", "
var_strs);
;;

let process_function out env func =
  let env = Env.assign_cur_func env func.fname in
  let env = List.fold_left
    (fun env param -> Env.add_var_decl env param)
      env func.params in
  fprintf out "\n";
  fprintf out "%s\n" (func_signature env func);

  (* Now we emit the code for the functions. We will have special handling here for those
functions that
     have c-code already defined
   *)
  match func.fname with
    | "print" -> fprintf out "{ printf (\"%s\", %s); return 0; } // special function\n"
"%s" (Env.lookup_symbol env "str").code_name
    | "printLine" -> fprintf out "{ printf (\"%s\\n\", %s); return 0; } // special
function\n" "%s" (Env.lookup_symbol env "str").code_name
    | "fail" -> fprintf out "{ printf (\"%s\\n\", %s); printf (\"Exiting due to
failure\"); exit(-1);  return -1;} // special function\n" "%s" (Env.lookup_symbol env
"str").code_name
```

```
    | "readLine" -> fprintf out "{
        char *myline;
        myline = (char *) malloc(MAX_STRING_SIZE + 1);
        size_t nbytes = MAX_STRING_SIZE;
        getline(&myline, &nbytes, stdin);
        return myline;
        // TODO handle memory cleanup
    } // special function\n"
    | fname ->
      match func.body with
        | [block] ->
          (match block with
            | Block(stmts) ->
              let sast_block, env = to_sast_stmt env block in
              fprintf out "{\n";
              print_sast_stmt out sast_block;
              fprintf out "printf(\"Runtime exception - no return defined in this path of
the code\"); exit(-2);}\n\n";
            | _ -> raise (CompileException ("Function must only start with a block"))
          )
        | _ -> raise (CompileException ("Function must only start with a block"))
;;


let process_program program_name env enums funcs =
  let funcs = (List.map snd (StringMap.bindings env.func_map)) in
  let c_file_prefix = "zzzz_"^program_name in
  let c_file = c_file_prefix ^ ".c" in
  let out_file = program_name ^ ".out" in
  let out = open_out c_file in

  (* First, write out the contents of the code to a temporary file *)
  fprintf out "#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <regex.h>
#include <string.h>
#define MAX_STRING_SIZE 1024
#define MAX_MATCHING_GROUPS 5

struct board {
       const char** boardarray;
    int rowlength;
    int collength;
};

// credits on this code go to: http://stackoverflow.com/a/11864144
int compare_regexp (const char* source, const char* regexString, char
groups[][MAX_STRING_SIZE]) {
       regex_t regexCompiled;
```

```
	regmatch_t groupArray[MAX_MATCHING_GROUPS];
	int foundGroups = 0;

	if (regcomp(&regexCompiled, regexString, REG_EXTENDED)) {
		// Could not compile regular expression
		return -1;
	};

	if (regexec(&regexCompiled, source, MAX_MATCHING_GROUPS, groupArray, 0) == 0) {
		unsigned int g = 0;
		for (g = 0; g < MAX_MATCHING_GROUPS; g++) {
			if (groupArray[g].rm_so == (size_t)-1)
			break;  // No more groups

			foundGroups++;

			char sourceCopy[strlen(source) + 1];
			strcpy(sourceCopy, source);
			sourceCopy[groupArray[g].rm_eo] = 0;
			strncpy(groups[g], sourceCopy + groupArray[g].rm_so, 1024);
		}
	}
	regfree(&regexCompiled);

	if ( foundGroups > 0) {
		foundGroups--;  // this is because one of the groups in the method is the
full matched string itself
		// if no groups were specified in the regexp, then this still returns 1.
Hence, we force it to 0,
		// as no actual groups were found
	}
	return foundGroups;
}
";

  List.iter (fun func -> fprintf out "%s;\n" (func_signature_for_prototype func)) funcs;
  fprintf out "\n";
  List.iter (fun enum -> process_enum out env enum) enums;
  fprintf out "\n";
  List.iter (fun func -> process_function out env func) funcs;
  close_out out;

  (* Now compile that C code to its final binary output *)
  (* Note that we throw a CriticalCompileException here if we fail. We segregate this
from the regular
     CompileException, as a compilation failure in C code implies that we missed out on
some checks
     here in the Ocaml code, and ideally our Ocaml code should always generate valid C
code (and if it
     can't, it should throw the error before even trying to compile)
```

```
   *)
  let result = Unix.system (sprintf "/usr/bin/gcc %s -o %s" c_file out_file) in
  match result with
    | Unix.WEXITED(exit_code) ->
      (match exit_code with
        | 0 -> printf "Compile executed successfully for program %s\n" program_name;
        | _ -> raise(CriticalCompileException(sprintf "Compile of c-code failed and
returned with exit code %d" exit_code));
      )
    | _ -> ignore(raise(CriticalCompileException("Unexpected process status")));
  ();
;;


(* Here, we generate some functions specific for each enum
   So far, we have:
   1) parseEnum_<enumName> to return an enum value given a string
   2) printing out a board of enums
   3) printing out a row of enums on a board *)
let get_enum_functions enum =
  let signature = sprintf "function %s parseEnum_%s(string str)\n" enum.ename enum.ename
in
  let body = List.fold_left
    (fun l id ->
      let ifbody = sprintf "if str == \"%s\" then\nreturn %s\nend if\n" id id in
      ifbody :: l
    ) [] enum.ids in
  let body_string = String.concat "\n" body in
  let end_sig = "end function\n" in
  let printBoardFunc = sprintf "function int printBoard_%s(board<%s> myboard)
       int row
       for row = 1; row <= myboard.rowlength; row = row + 1 do
              printBoardRow_%s(myboard, row)
       end for
       return 0
  end function

function int printBoardRow_%s(board<%s> myboard, int row)
       int col
       for col = 1; col <= myboard.collength; col = col + 1 do
              print(myboard[row, col] @ \" \")
       end for
       printLine(\"\")
       string dummy = \"\" /* This is only here to work around Eclipse syntax
highlighting issue.... */
       return 0
  end function
  " enum.ename enum.ename enum.ename enum.ename enum.ename in

  signature ^ body_string ^ end_sig ^ printBoardFunc
```

```
;;

let parse_program_from_lexbuf lexbuf =
  let enums, funcs =
    try Parser.program Scanner.token lexbuf
    with Parsing.Parse_error ->
      let region_text =
        let start_index = max (lexbuf.Lexing.lex_start_pos - 20) 0 in
        let length =
          if start_index + 40 > String.length lexbuf.Lexing.lex_buffer
          then (String.length lexbuf.Lexing.lex_buffer) - start_index
          else 40
         in
        String.sub lexbuf.Lexing.lex_buffer start_index length
      in
      (* need a separate printf from the raise-ParseException call, as Ocaml seems to cut
off the error message when shown via the raise call *)
      printf "Found parsing error at character #%d on char %s in region\n[%s]\n"
        lexbuf.Lexing.lex_start_pos
        (Char.escaped (String.get lexbuf.Lexing.lex_buffer lexbuf.Lexing.lex_start_pos))
        region_text
      ;
      raise(ParseException("Found parsing error - see previous log message"))
  in
  enums, funcs
;;

let parse_program_from_string program_text =
  parse_program_from_lexbuf (Lexing.from_string program_text)
;;

let compile_program_from_lexbuf program_name lexbuf =
  let enums, funcs = parse_program_from_lexbuf lexbuf in
  let enumfuncs = List.fold_left
    (fun l enum ->
      let _ , funcs = parse_program_from_string (get_enum_functions enum) in
      funcs @ l
    ) [] enums in
  let funcs = funcs @ enumfuncs in
  let env = initialize_env_with_globals enums funcs in
  let main_method =
    try Env.lookup_function env "main"
    with Not_found -> raise(CompileException("Program must define a \"main\" method"))
  in
  if List.length main_method.params <> 0
  then raise(CompileException("\"main\" methods must not have any args defined"));
  process_program program_name env enums funcs;
;;

let compile_program_from_string program_name program_text =
```

```
  compile_program_from_lexbuf program_name (Lexing.from_string program_text);
;;



(********************* env.ml *********************)
open Ast;;
open Printf;;
open Common;;

module StringMap = Map.Make(String);;

(****************
 * Root type definitions for the environment
 ***************)

type symbol_table = { symbol_map : reference StringMap.t }
;;

type env =
  {symbols : symbol_table;  (* store all symbols in this map for quick access, incl.
symbols for function names *)
   func_map : func_decl StringMap.t;  (* map for the function metadata *)
   type_map : type_id StringMap.t;  (* map for the type metadata *)
   cur_func_name : string;  (* The current function that is being processed for this
environment. Used during the code generation *)
  }
;;

let create_new () =
  { func_map = StringMap.empty; type_map = StringMap.empty; symbols = { symbol_map =
StringMap.empty }; cur_func_name = ""; }
;;


(****************
 * Lookup methods to search the environment
 ***************)

let lookup_function env id =
  StringMap.find id env.func_map
;;

let lookup_type env id =
  StringMap.find id env.type_map
;;

let lookup_symbol env id =
  let rec lookup_symbol_table tab id =
    if StringMap.mem id tab.symbol_map
```

```
    then StringMap.find id tab.symbol_map
    else raise(CompileException(sprintf "Could not find id reference [%s] inside code for
function [%s]" id env.cur_func_name))
  in lookup_symbol_table env.symbols id
;;


(****************
 * Methods to facilitate type lookups to resolve type names to a unified
 * form (e.g. converting enums classes like say MyEnum to enum<MyEnum>
 ***************)

let rec resolve_type_id env typeid = match typeid with
  | SimpleTypeId(t) -> lookup_type env t
  | GenericTypeId(gentype, paramtype) ->
    (* This is to handle the case where the type is already property resolved, e.g.
       board<enum<myenum2>> should not resolve to board<enum<enum<myenum2>>>
       This won't scale well for "pure" generics, but for our limited case, this should
work
     *)
    let resolved_type = (resolve_type_id env paramtype) in
    if (type_id_to_string typeid) = (type_id_to_string resolved_type)
    then typeid
    else GenericTypeId(gentype, resolved_type);
;;

let resolve_var_decl env v =
  { vtype = resolve_type_id env v.vtype; vname = v.vname }
;;


(****************
 * Processing-specific methods to modify some state of the environment
 * while functions are being processed
 ***************)

let assign_cur_func env func_name=
  { func_map = env.func_map; type_map = env.type_map; symbols = env.symbols;
cur_func_name = func_name }
;;


(****************
 * Methods to add symbols and other information to the environment
 ***************)

(* The core method to add symbols to an environment *)
let add_symbol env symbol p_ref_type =
  let add_symbol_to_table table symbol p_ref_type =
    if StringMap.mem symbol table.symbol_map
```

```
    then raise (CompileException ("Symbol already exists: " ^ symbol))
    else {symbol_map = StringMap.add symbol { ref_type=p_ref_type; code_name="var" ^
get_next_tmp_id () } table.symbol_map;}
  in
  {func_map = env.func_map;
   type_map = env.type_map;
   symbols = add_symbol_to_table env.symbols symbol p_ref_type;
   cur_func_name = env.cur_func_name;
  }
;;

let add_var_decl env v =
  let v = resolve_var_decl env v in
  add_symbol env v.vname v.vtype
;;

let add_type env typename typeval =
  let env = add_symbol env typename type_type in
  {func_map = env.func_map;
   type_map = StringMap.add typename typeval env.type_map;
   symbols = env.symbols;
   cur_func_name = env.cur_func_name;
  }
;;


(*
   For enums, we register the enum itself as a type class, and all the enum
   elements inside of it as symbols in the global environment
 *)
let add_enum env enum =
  let enum_type = GenericTypeId("enum", (SimpleTypeId(enum.ename))) in
  let env = add_type env enum.ename enum_type in
  let env = List.fold_left (fun env enumid -> add_symbol env enumid enum_type) env
enum.ids in
  env
;;

(*
   We always resolve the type information prior to adding something to the
   environment, as this makes the code generation logic easier to handle
 *)
let add_func env func =
  let func =
    {rettype = resolve_type_id env func.rettype;
     fname = func.fname;
     params = List.map (fun p -> resolve_var_decl env p) func.params;
     body = func.body} in
  let env = add_symbol env func.fname function_type in
  {func_map = StringMap.add func.fname func env.func_map;
```

```
   type_map = env.type_map;
   symbols = env.symbols;
   cur_func_name = env.cur_func_name;
  }
;;
```

```
(********************* Makefile.ml *********************)
PARSER_FILES = ast.ml parser.ml scanner.ml
PARSER_OBJS = ast.cmo parser.cmo scanner.cmo

COMPILE_FILES = ${PARSER_FILES} common.ml env.ml saster.ml codegenerator.ml compile.ml

COMMON_FILES = unix.cmxa str.cmxa


# NOTE - we need to rely on the OBJs as ocamlyacc and lex need the objects
# created to compile
# We then still need to compile from the native sources when compiling
# via ocamlopt

boredgame : $(PARSER_OBJS)
	ocamlopt -o boredgame.out ${COMMON_FILES} ${COMPILE_FILES} boredgame.ml

testcompile : $(PARSER_OBJS)
	ocamlopt -o testcompile.out ${COMMON_FILES} ${COMPILE_FILES} testcompile.ml

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

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

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

.PHONY : clean
clean :
	rm -f parser.ml parser.mli scanner.ml testall.log \
	*.cmo *.cmi *.out *.diff *.c *.o *.cmx

# Generated by ocamldep -native *.ml *.mli
ast.cmo:
ast.cmx:
boredgame.cmo: compile.cmx
boredgame.cmx: compile.cmx
```

```
codegenerator.cmo: env.cmx common.cmx ast.cmx
codegenerator.cmx: env.cmx common.cmx ast.cmx
common.cmo: ast.cmx
common.cmx: ast.cmx
compile.cmo: scanner.cmx parser.cmi env.cmx ast.cmx
compile.cmx: scanner.cmx parser.cmx env.cmx ast.cmx
env.cmo: common.cmx ast.cmx
env.cmx: common.cmx ast.cmx
parser.cmo: ast.cmx parser.cmi
parser.cmx: ast.cmx parser.cmi
saster.cmo: env.cmx common.cmx ast.cmx
saster.cmx: env.cmx common.cmx ast.cmx
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
testcompile.cmo: compile.cmx ast.cmx
testcompile.cmx: compile.cmx ast.cmx
parser.cmi: ast.cmx



(********************* parser.mly *********************)
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA NEWLINE
%token DOT
%token ARROW MATCH
%token LBRACKET RBRACKET
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token AT
%token AND OR
%token RETURN IF THEN ELSE ELSEIF FOR WHILE END DO
%token FUNCTION ENUM
%token <int> LITERAL
%token <string> STRINGLITERAL
%token <string> ID
%token <bool> BOOLEANLITERAL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%nonassoc ARROW MATCH
%nonassoc COMMA
%right ASSIGN
%left OR AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left AT
%left LBRACKET
%left DOT
```

```
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program
%type <Ast.enum_decl> enum_decl

%%

/* A general note to the instructor on the coding of the rules below, and specifically on
concatenating lists:
  I explicitly am choosing to concatenate lists using the list concatenation @, instead
of a new head to another list
  and then reversing the list when setting it to its final home

  Though the concatenation is not tail-recursive and so less-efficient processing-wise,
it makes the code
  easier to understand and debug, as I don't have to debug potential issues about the
order of the list that
  results.

  We can change this if performance is an issue (but per the notes in class, we don't
have to worry about performance
  too much here
  */


program:
  /* nothing */ { [], [] }
  | program enum_decl { (fst $1 @ [$2] ), snd $1 }
  | program fdecl { fst $1, snd $1 @ [$2] }
  | program NEWLINE { $1 }
;

/*Note - this causes a shift-reduce conflict. We cannot avoid it w/ yacc as per various
commentary
  on the web, such as this:
http://stackoverflow.com/questions/17114634/yacc-shift-reduce-conflict-when-parse-c-templ
ate-arguments
  */
type_id:
  | ID               { SimpleTypeId($1) }
  | ID LT type_id GT { GenericTypeId($1, $3) }
;

fdecl:
  | FUNCTION type_id ID LPAREN vdecl_list RPAREN NEWLINE stmt_list END FUNCTION NEWLINE
    { { rettype = $2; fname = $3; params = $5; body = [Block($8)]} }
;
```

```
enum_decl:
  | ENUM ID NEWLINE formal_list NEWLINE END ENUM NEWLINE
    { { ename = $2; ids = $4 } }
;

vdecl:
  | type_id ID { { vtype = $1; vname = $2; } }
;

formal_list:
  | ID                    { [$1] }
  | formal_list COMMA ID { $1 @ [$3] }
;

vdecl_list:
  /* nothing */    { [] }
  | vdecl     { [$1] }
  | vdecl_list COMMA vdecl { $1 @ [$3] }
;

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

stmt:
  | expr NEWLINE { Expr($1) }
  | RETURN expr NEWLINE { Return($2) }
  | IF expr THEN NEWLINE stmt_list elseifs %prec NOELSE END IF NEWLINE { If($2,
Block($5), $6, Block([])) }
  | IF expr THEN NEWLINE stmt_list elseifs ELSE NEWLINE stmt_list END IF { If($2,
Block($5), $6, Block($9)) }
  | FOR expr SEMI expr SEMI expr DO NEWLINE stmt_list END FOR NEWLINE
    { For($2, $4, $6, Block($9)) }
  | WHILE expr DO NEWLINE stmt_list END WHILE NEWLINE { While($2, Block($5)) }
  | vdecl NEWLINE { VarDecl($1) }
  | vdecl ASSIGN expr NEWLINE { VarDeclAndAssign($1, $3) }
  | NEWLINE { NoOpStmt }
;

elseifs:
  /* nothing */ { [] }
  | elseifs ELSEIF expr THEN NEWLINE stmt_list { $1 @ [($3, Block($6))] }
;

id_list:
  | ID          { [$1] }
  | id_list ID { $1 @ [$2]  }
;
```

```
id_array:
  | id_list                  { [$1] }
  | id_array NEWLINE id_list { $1 @ [$3] }
;

expr:
  | LITERAL          { IntLiteral($1) }
  | BOOLEANLITERAL    { BooleanLiteral($1) }
  | STRINGLITERAL    { StringLiteral($1) }
  | LBRACE NEWLINE id_array NEWLINE RBRACE    { BoardLiteral($3) }
  | ID                      { Id($1) }
  | expr LBRACKET expr COMMA expr RBRACKET { MatrixOp($1, $3, $5) }
  | 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 AND expr { Binop($1, And,   $3) }
  | expr OR expr { Binop($1, Or,   $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) }
  | expr AT    expr { Binop($1, Concat,   $3) }
  | expr DOT    ID { ClassOp($1, $3) }
  | expr ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }
  | expr MATCH expr ARROW vdecl_list { RegexpMatcher($1, $3, $5) }
;

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { $1 }
;

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



(********************* saster.ml *********************)
open Ast;;
open Printf;;
open Env;;
open Common;;
```

```
let op_expected_type = function
  | Add -> int_type
  | Sub -> int_type
  | Mult -> int_type
  | Div -> int_type
  | Equal -> boolean_type
  | Neq -> boolean_type
  | Less -> boolean_type
  | Leq -> boolean_type
  | Greater -> boolean_type
  | Geq -> boolean_type
  | Or -> boolean_type
  | And -> boolean_type
  | Concat -> string_type
;;


(*
   Convert the incoming expression to a semantically-analyzed expression.
   Tack on the autogenerated symbol reference if needed
 *)
let rec to_sast_expr env expr = match expr with
  | IntLiteral(l) ->
    (* Note - this can be represented as a direct literal in C; hence, no need for the
get_next_tmp_id *)
    {e = SastIntLiteral(l); etype = int_type; eref = string_of_int l};
  | BooleanLiteral(l) ->
    (* Note - this can be represented as a direct literal in C; hence, no need for the
get_next_tmp_id *)
    {e = SastBooleanLiteral(l); etype = boolean_type; eref = string_of_bool l};
  | StringLiteral(l) ->
    (* Note - this can be represented as a direct literal in C; hence, no need for the
get_next_tmp_id *)
    {e = SastStringLiteral(l); etype = string_type; eref = sprintf "\"%s\"" l};
  | Id(id) ->
    let idref = (Env.lookup_symbol env id) in
    {e = SastId(id); etype = idref.ref_type; eref = idref.code_name};
  | BoardLiteral(array) ->
    let numrows = List.length array in
    if numrows = 0 then raise(CompileException("Board must have >0 rows defined"));
    let numcols = List.length (List.hd array) in
    if numcols = 0 then raise(CompileException("Board must have >0 columns defined"));
    let idref = Env.lookup_symbol env (List.hd (List.hd array)) in
    let boardtype = idref.ref_type in
    if List.for_all (fun l -> List.length l = numcols) array <> true then
raise(CompileException("All rows of a board must have the same number of columns"));

    let board_elems =
```

```
      (List.map (fun elems -> List.map
        (fun id ->
          let valexpr = to_sast_expr env (Id(id)) in
          if not(type_is_of_type "enum" valexpr.etype) then raise(CompileException("Only
enum types are allowed in a board"));
          valexpr
        ) elems) array) in
    (* The board literal is too complex to represent as a standalone literal in C, as we
did for int, boolean, string
       Hence, we will use get_next_tmp_id here for this *)
    {e = SastBoardLiteral(numrows, numcols, board_elems); etype = GenericTypeId("board",
boardtype); eref = get_next_tmp_id () };
  | Assign(expr, e) ->
    let exprtype, exprref =
      (match expr with
        | Id(id) -> let idref = Env.lookup_symbol env id in (idref.ref_type,
idref.code_name)
        | MatrixOp(boardexp, rowexp, colexp) -> let sast_expr = to_sast_expr env expr in
(sast_expr.etype, sast_expr.eref)
        | _ -> raise(CompileException("not yet ipmlemented for others"))
      )
    in
    let rhs = to_sast_expr env e in
    if rhs.etype = exprtype
    then {e = SastAssign(rhs); etype = rhs.etype; eref = exprref}
    else raise(CompileException(sprintf "args do not have matching types: LHS[%s] vs.
RHS[%s]" (type_id_to_string exprtype) (type_id_to_string rhs.etype)));
  | MatrixOp(boardexp, rowexp, colexp) ->
    let board = to_sast_expr env boardexp in
    let row = to_sast_expr env rowexp in
    let col = to_sast_expr env colexp in
    if not (type_is_of_type "board" board.etype) then raise(CompileException("lhs in
matrix operation must be of type board; instead it was " ^ (type_id_to_string
board.etype)));
    if not (type_is_of_type "int" row.etype) then raise(CompileException("index operands
in matrix operation must be of type int; instead it was " ^ (type_id_to_string
row.etype)));
    if not (type_is_of_type "int" col.etype) then raise(CompileException("index operands
in matrix operation must be of type int; instead it was " ^ (type_id_to_string
col.etype)));
    let arrayref = calc_matrix_to_array_index_for_c_code board row col in
    {e = SastMatrixOp(board, row, col); etype = type_to_generic_subtype board.etype; eref
= arrayref}
  | ClassOp(expr, memberName) ->
    let sexpr = to_sast_expr env expr in
    if not(type_is_of_type "board" sexpr.etype) then raise(CompileException("Dot operator
can only be used on classes (e.g. board)"));
    let return_type = match memberName with
      | "rowlength" -> int_type
      | "collength" -> int_type
```

```
      | _ -> raise(CompileException("No other args than rowlength and collength are
supported"))
    in
    {e = SastClassOp(sexpr, memberName); etype = return_type; eref = get_next_tmp_id ()}
  | RegexpMatcher(inputstr, regexpstr, vars) ->
    let input = to_sast_expr env inputstr in
    let regexp = to_sast_expr env regexpstr in
    if input.etype <> string_type then raise(CompileException("Input to regexp matching
operation must be a string"));
    if regexp.etype <> string_type then raise(CompileException("Regexp in regexp matching
operation must be a string"));
    (* the actual symbols (i.e. the 4th arg to SastRegexpMatcher) will be fixed later in
the If logic inside to_sast_stmt *)
    {e = SastRegexpMatcher(input, regexp, vars, []); etype = boolean_type; eref =
get_next_tmp_id ()}
  | Binop(e1, op, e2) ->
    let sexp1 = to_sast_expr env e1 in
    let tmpid = get_next_tmp_id () in
    let sexp2 = to_sast_expr env e2 in
    (match op with
      | Concat ->
        if sexp1.etype <> string_type && sexp2.etype <> string_type
        then raise(CompileException("One of the types in a concatenation must be a
string"));
      | _ ->
        if sexp1.etype <> sexp2.etype
        then raise(CompileException("Both sides of Binop need to evaluate to the same
type"));
    );
    {e = SastBinop(sexp1, op, sexp2); etype = op_expected_type op; eref = tmpid}
  | Call (id, exprs) ->
    let idref = Env.lookup_symbol env id in
    if idref.ref_type <> function_type
    then raise(CompileException("Attempting to make a function call on a non-function " ^
id));
    let sexprs = List.map (fun expr -> to_sast_expr env expr) exprs in
    let func = Env.lookup_function env id in
    let numfuncargs = List.length func.params in
    let numcallargs = List.length sexprs in
    if numcallargs <> numfuncargs
    then raise(CompileException(sprintf "Function mismatch in terms of # params passed
in. Func [%s] expects %d args, only received %d" func.fname numfuncargs numcallargs ));
    List.iter2
      (fun fparam cexpr ->
        let ftype = type_id_to_string fparam.vtype in
        let ctype = type_id_to_string cexpr.etype in
        if ftype <> ctype then raise(CompileException(sprintf "Mismatch in function types
to call of %s for var %s [func-%s vs. call-%s]" func.fname fparam.vname ftype ctype));
      )
      func.params sexprs;
```

```
    {e = SastCall(id, sexprs); etype = func.rettype; eref = get_next_tmp_id ()}
;;


let rec to_sast_stmt env = function
  | NoOpStmt -> SastNoOpStmt, env
  | Block(stmts) ->
    (* Note - for the statements in a block, we do want the environment changes from one
statement to carry
       over to the next. Hence, we use fold_left here *)
    let sast_stmts, env = List.fold_left
      (fun (sast_stmts, env) stmt ->
        let new_sast_stmt, new_env = to_sast_stmt env stmt in
        sast_stmts @ [new_sast_stmt], new_env
      ) ([], env) stmts in
    SastBlock(sast_stmts), env;
  | VarDecl(v) ->
    let env = Env.add_var_decl env v in
    let idref = Env.lookup_symbol env v.vname in
    SastVarDecl(v.vname, idref), env;
  | VarDeclAndAssign(v, expr) ->
    let env = Env.add_var_decl env v in
    let idref = Env.lookup_symbol env v.vname in
    let sexpr = to_sast_expr env expr in
    SastVarDeclAndAssign(v.vname, idref, sexpr), env;
  | Expr(e) ->
    (
    let sexpr = to_sast_expr env e in
    match e with
      | Assign(id, e) -> SastExpr(sexpr), env;
      | Call (id, exprs) -> SastExpr(sexpr), env;
      | _ -> raise(CompileException("No other expression type is allowed as a
statement"));
    )
  | Return(e) ->
    let sexpr = to_sast_expr env e in
    let func = Env.lookup_function env env.cur_func_name in
    if sexpr.etype = func.rettype
    then SastReturn(sexpr), env
    else raise(CompileException(sprintf "Return statement value [%s] in function [%s]
does not match the return type of the function [%s]" (type_id_to_string sexpr.etype)
func.fname (type_id_to_string func.rettype)))
  | If(e, s1, elseifs, s2) ->
    let conds = (e, s1) :: elseifs in
    let sast_conds = List.fold_left
      (fun list (expr, stmt) ->
        let sexpr = to_sast_expr env expr in
        if sexpr.etype <> boolean_type then raise (CompileException ("Clause of the
if-statement must have a boolean value"));
        let sexpr, newenv = match sexpr.e with
```

```
          | SastRegexpMatcher(input, regexp, vars, _) ->
            let newenv = List.fold_left (fun env var -> Env.add_var_decl env var) env
vars in
             let newrefs = List.map (fun var -> Env.lookup_symbol newenv var.vname) vars
in
            let newsexpr = { e = SastRegexpMatcher(input, regexp, vars, newrefs); etype =
sexpr.etype; eref = sexpr.eref } in
            newsexpr, newenv;
          | _ -> sexpr, env
            (* no need to carry the environment over, as it will be in a block and any
newly-created variables would lose scope *)
        in
        let sast_stmt, _ = to_sast_stmt newenv stmt in
        list @ [(sexpr, sast_stmt)];
      ) [] conds in
    let sast_stmt2, newenv = to_sast_stmt env s2 in
    (SastIf(sast_conds, sast_stmt2), env)
  | While(expr, stmt) ->
    let sexpr = to_sast_expr env expr in
    (* no need to carry the environment over, as it will be in a block and any
newly-created variables would lose scope *)
    let sast_stmt, _ = to_sast_stmt env stmt in
    SastWhile(sexpr, sast_stmt), env
  | For(e1, e2, e3, s) ->
    let sexpr1 = to_sast_expr env e1 in
    let sexpr2 = to_sast_expr env e2 in
    let sexpr3 = to_sast_expr env e3 in
    let sast_stmt, _ = to_sast_stmt env s in
    if sexpr2.etype <> boolean_type then raise (CompileException ("2nd clause of the
for-loop declaration must return a boolean"));
    SastFor(sexpr1, sexpr2, sexpr3, sast_stmt), env;
;;



(********************* scanner.mll *********************)
{ open Parser }

rule token = parse
  | [' ' '\t'] { token lexbuf } (* Whitespace *)
  | '\r'?'\n' { NEWLINE } (* Note - newlines are NOT just whitespace in the boredgame
language *)
  | "/*"     { comment lexbuf }            (* Comments *)
  | "//"     { comment_singleline lexbuf } (* Comments *)
  | '('      { LPAREN }
  | ')'      { RPAREN }
  | '{'      { LBRACE }
  | '}'      { RBRACE }
  | '.'      { DOT }
  | '['      { LBRACKET }
```

```
  | ']'      { RBRACKET }
  | ';'      { SEMI }
  | ','      { COMMA }
  | '+'      { PLUS }
  | '-'      { MINUS }
  | '*'      { TIMES }
  | '/'      { DIVIDE }
  | '@'      { AT  }
  | '='      { ASSIGN }
  | "=="     { EQ }
  | "!="     { NEQ }
  | '<'      { LT }
  | "<="     { LEQ }
  | ">"      { GT }
  | ">="     { GEQ }
  | "and"     { AND }
  | "or"     { OR }
  | "->"     { ARROW }
  | "~="     { MATCH }
  | "if"     { IF }
  | "then"     { THEN }
  | "else"   { ELSE }
  | "elseif" { ELSEIF }
  | "for"    { FOR }
  | "do"  { DO }
  | "while"  { WHILE }
  | "return" { RETURN }
  | "end"    { END }
  | "function"    { FUNCTION }
  | "enum"    { ENUM }
  | "true"     { BOOLEANLITERAL(true) }
  | "false"     { BOOLEANLITERAL(false) }
  | ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
  | '\"'[^'\"']*'\"' as lxm { STRINGLITERAL(String.sub lxm 1 ((String.length lxm) - 2)) }
  | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
  | eof { EOF }
  | _ as char { raise (Failure(Printf.sprintf "illegal character %s at %d near %s"
                          (Char.escaped char)
                          lexbuf.Lexing.lex_start_pos
                          (String.sub lexbuf.Lexing.lex_buffer
(lexbuf.Lexing.lex_start_pos - 5) 10)
                               )) }
and comment = parse
  | "*/" { token lexbuf }
  | _    { comment lexbuf }
and comment_singleline = parse
  | '\r'?'\n' { token lexbuf }
  | _    { comment_singleline lexbuf }
```