GAQ

GENERATOR OF ADAPTIVE QUESTIONNAIRES

Esther Kundin, Ayla Brayer

August 16, 2011

Contents

Introduction

As everyone who has visited a doctor recently knows, appointments to see doctors can take a long time. Filling out the questionnaires can take a while too. As doctors' offices are increasingly computerized, it is conceivable that diagnostic tools can be generated to hone in on diagnoses based on answers to simple questions about a patients' symptoms. Filling out secure adaptive diagnostic questionnaires, that ask questions based on your previous answers, would help doctors save time during visits and hopefully get you out of the office faster. A language that would require no prior programming experience and would be intuitive to use would be used to generate adaptive questionnaires that specialists can use to pre-diagnose patients before they even get into the office. The language should be simple enough that doctors or even their assistants should be able to use it to tailor questions to their clientele easily.

Similar questionnaires can be used by car mechanics to diagnose car trouble. Yet another application for an adaptive questionnaire generator would be to customer-satisfaction surveys or even phone menu systems. All of them would benefit from a more adaptive approach. Rather than forcing people to answer reams of unrelated questions, adaptive questionnaires can adapt to user inputs and have users answer fewer but more targeted questions. This would make the taking of surveys less tedious, which would in turn get more people to take them.

The Language

GAQ has a simple and intuitive design to insure that users with little or no programming experience will find it easy to use. The program has three logical sections – the variable declarations, the questions of the survey, and the statements to be executed upon completion. Each of these sections is optional. Each program will be stored in one file. There is no concept of including separate translation units in separate files and linking them together at this time, although that would be a useful feature to add in a future edition.

In order to simulate the interactive adaptability of the question/answer session of a survey, the questions are designed as a tree, with a root question that will be posed, and the children are the possible further questions that will be asked based on the state of the program, including the answers received from the user as well as the values of other variables.

Tutorial

Each GAQ program consists of one text file, saved with the .gaq extension. Each program can have up to 3 sections, the variable declarations, the questions, and the footer of the program that has statements.

Hello World

Here is our first hello-world program:

/* This is a comment, comments do not nest and end like this: */

print(“hello world!”); /* note how statements end in a semicolon */

How to Build a GAQ Program

To build this program, we save it as helloworld.gaq and we run:

./gaq < helloworld.gaq > helloworld.cpp

g++ -o helloworld helloworld.cpp

./helloworld

The output will be

hello world!

We have provided a sample bash script to make the 2-step compilation easier, though the header may need to be modified to have the path to bash on your machine :

./gaq_compile helloworld

./helloworld

Variable Declarations

Now that we have our sample program working, let’s add some variables and print them out:

/*

we can have ints, floats, strings, and bools as variables.

Variables must be initialized when declared

*/

int a = 1;

```
float b = 1.5;

bool c = true;

print (“The value of a is $a.”);

print (“The value of b is $b.”);

print (“The value of c is $c.”);
```

The output of this program will be:

The value of a is 1.

The value of b is 1.5.

The value of c is 1.

Questions

Now, to add a question to our program:

```
/*This program has one question */

"This is my first questions.  How are you feeling?" string /* the response will be parsed as a string */

[response == "well"] {

   print ("I am glad you are well.");

}

[response == "ill"] {

   print ("I am sorry you are not feeling well.");

}

[default] /* all other answers will go here */ {

   print("I only know about ill and well, so let's try this again.");

   repeat; /* This will cause the enclosing question, our only one, to repeat */

}

/* Here we put the program footer which will always be hit once the questions are done.

*/

print ("Now we're done.");
```

Here are some runs of the program:

$ ./test2

This is my first questions. How are you feeling?

well

I am glad you are well.

Now we're done.

$ ./test2

This is my first questions. How are you feeling?

ill

I am sorry you are not feeling well.

Now we're done.

$ ./test2

This is my first questions. How are you feeling?

weird

I only know about ill and well, so let's try this again.

This is my first questions. How are you feeling?

well

I am glad you are well.

Now we're done.

A More Complete Example

Now, here is a final program that shows most of the features of the language and would be something used in a doctor's office to screen patients:

/* This is the sample program */

float fever_low = 95.0;

float fever_high = 100.0;

float your_fever = 0.0;

"Are you ill?" string

```
    [response == "yes"] {
      "Do you have a fever?" string
      [response=="yes"] {
        "Is your fever between $fever_low and $fever_high?" string
        [response=="yes"] {
          print ("The fever range was $fever_low - $fever_high");
          "What is your temperature?" float
          [response>fever_high | response <fever_low] {repeat;}
          [default] {
            print ("We will tell the doctor that you are sick with a fever of
$response.");
          }
        }
        [default] { /* here we'll try to raise the threshold till we get the right range */
              fever_low = fever_low+5.0; fever_high = fever_high+5.0; repeat;

        }
      }
      [response=="no"] {
        "What are your other symptoms?" string
        [default] {
          print("We will tell the doctor that you are sick with symptoms of $response.");
        }
      }
    }
    [response=="no"] {
      "Are you here for a checkup?" string
```

```
    [response == "yes"] {
      print("We will tell the doctor that you're here for a checkup.");
    }
    [default] {
      print("Please talk to the secretary.");
    }
  }
```

Some sample runs are as follows:

```
$ ./test3
Are you ill?
yes
Do you have a fever?
yes
Is your fever between 95 and 100?
no
Is your fever between 100 and 105?
yes
The fever range was 100 - 105
What is your temperature?
104
We will tell the doctor that you are sick with a fever of 104.
$ ./test3
Are you ill?
yes
Do you have a fever?
no
```

What are your other symptoms?

nausea

We will tell the doctor that you are sick with symptoms of nausea.

$ ./test3

Are you ill?

no

Are you here for a checkup?

yes

We will tell the doctor that you're here for a checkup.

$ ./test3

Are you ill?

no

Are you here for a checkup?

no

Please talk to the secretary.

The Power of GAQ

GAQ is a complete programming language which can be used to compute any algorithm (though that is not what its original design was for). As an example, we have provided a sample program that computes gcd.

```
int a = 0;
int b = 0;
"Enter one positive number" int
[response < 1] {
  repeat;
}
[default] {
 a = response;
 "Enter another positive number" int
 [response < 1] {
   repeat;
 }
 [default] {
   b = response;
   print ("Finding the gcd of $a and $b.");
   "Calculating gcd. Enter anything." string
   [a == 0] {
```

```
      print("GCD is $b.");
    }
    [b == 0] {
      print ("GCD is $a.");
    }
    [a > b] {

      a = a -b;
      print ("A is now $a, B is still $b");
      repeat;
    }
    [default] {
      b = b - a;
      print ("A is still $a, B is now $b");
      repeat;
    }
  }
}
```

Language Reference Manual

1 Lexical Conventions

1.1 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. White space, as described below, is ignored except as it separates tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants. If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

1.2 Comments

The characters /* introduce a comment, which terminates with the characters '*/'. Comments do not nest, and they do not occur within string literals.

1.3 Identifiers

An identifier is a sequence of letters, digits, and the underscore, beginning with a letter. Identifier names are case-sensitive and may have any length.

1.4 Keywords

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

int	*float*	*bool*	*string*	*true*
false	*default*	*repeat*	*response*	*print*

1.5 Constants

There are several kinds of constants. Each has a datatype.

- Integer-constants
- String Literals
- Floating-constants
- Boolean-constants

1.5.1 Integer Constants

An integer constant consists of a series of digits, taken to be decimal. Negative integer will be preceded by a – character and wrapped in parenthesis, i.e. (-42). Its type is an *int*.

1.5.2 String Literals

A string literal, also called as string constant, is a sequence of characters surrounded by double quotes. It is of type string and initialized with the given characters. The literals are immutable and two identical string literals are not necessarily equal. The type is *string*.

1.5.3 Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, and e or E, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (but not both) may be missing. Negative floating constant will be preceded by a '–' character and wrapped in parenthesis i.e. (-1.0). The type is *float*.

1.5.4 Boolean constants

There are 2 boolean constants that are reference via keywords – *true* and *false*. Their type is *bool*.

1.6 Whitespace

Whitespace is defined as the ASCII space, horizontal tab and form feed characters, as well as line terminators and comments.

2 Syntax Notation

The syntax for specifying a full GAQ program is as follows:

program:

declarations$_{opt}$ questions$_{opt}$ footer$_{opt}$

Thus, an empty program is an acceptable, though admittedly useless, program.

2.1 Meaning of Identifiers

Identifiers are used to refer to objects, also called variables. A variable is a location in storage, and its interpretation depends on the type of the variable. The valid types for variables are *int*, *float*, *bool*, and *string*. Variables must be assigned an initial value at declaration time.

2.2 Variable Declarations

Variable declarations are only allowed at the beginning of the program. Once the next section, the question specification, begins, no further declarations are allowed. Each identifier can only be used once. Once an identifier is declared as being of a given type, it may not be re-declared.

Variable declarations are of the form

variable-declarations:

type identifier = constant _expression;

where the expression needs to be a constant of the same type as the variable type. The valid types for variable declarations are *int*, *float*, *bool*, and *string*.

2.3 Expressions

An expression can be either a constant, or it can be an expression based on other variables and binary operators applied to them.

expression:

constant
variable
assignment
expression operator expression

One type of expression is an assignment, where an expression is assigned to a variable in the format of

assignment:

identifier = expr;

2.3 Operators

The following operators are available for use in expressions. Their use is the same as in expressions in the C language. One difference, however, is that the types of the expressions on either side of the operators must match, there will be no implicit conversion done. Thus, comparing and *int* and a *float* is illegal.

Relational operators: < = , < , >= , > , == , != - all of these apply to *int*s and *floats*. Only == and != can be used also with *bool*s and *string*s.

Logical operators: & (and), | (or) - apply only to *bool*, *float* and *int*. The outcome of a logical operator is of type *bool*.

Mathematical binary operators: + , -, *, / - apply only to *float* or *int.*

2.4 Question Syntax

The questions in a GAQ program are specified in a nested fashion.
Each question that will be asked to the user is represented as a string followed by the type that it will be cast to. The response to each question will be stored in the implicitly-declared *response* variable of type *type*. Each answer will be able to access that variable. When a new nested question is asked, a new *response* variable will be in the scope, shadowing the one available for the previous question. The scope of the *response* variable will begin after the response type declaration and will be available for the rest of the question, except when a nested question is in scope. Reference to the *response* variable out of scope is not permitted, with the exception of referring to it inside a print statement, in which case the variable *response* will be set to the empty string. For discussions about how casting is done from the string response into the *response* variable, see the section on casting below.

question:

string-literal type answer

2.5 Strings

A string is a sequence of characters surrounded by double quotes. It is of type *string* and initialized with the given characters. The string can have references to variables in it which will be evaluated at runtime. To reference a variable from within a string, the variable identifier will be preceded by $ and go until the end of the variable name, which is to say, until it hits a character that is not a letter, digit, or underscore. To print the actual character of $ and not have it interpreted as referring to a variable, it can be preceded by a backslash. Thus, the question of "$hello" would print the contents of the variable identified by hello. Whereas the question of "\ $hello" would print the string of characters: $hello. In order to print non-string variables inside questions, they will implicitly be cast into strings. See the casting section below for the exact definitions.

2.6 Answer Syntax

Each question is followed by a series of answers where each answer consists of expressions enclosed in square braces followed by a block of code in curly braces. The expressions in the square braces cannot be empty. The keyword default, which is a catchall, can be substituted for the expression. When evaluating the answer to a question, each of the expressions in square braces will be evaluated in the order they were declared, and the first one to evaluate to true will have its answer block run. Thus, once a default is seen, any other answers seen afterwards will be ignored.

answer:

[expression] {answer-block}$_{opt}$ answer

[default] {answer-block}$_{opt}$

2.7 Answer Block

Each block of code in the answer-block will consist of a list of answer expressions. An answer block can be empty, in the case of the final answer block that will be a leaf in the tree.

answer-block :

answer-expression;$_{opt}$ answer-block

2.8 Statement

Expressions can take two forms. They can either be a variable identifier assigned an expression, such as hello = 5+2; or a print output statement, described below.

statement:

identifier = expression;
print (string);

2.9 Answer Expression

Expressions that are allowed inside the answer-block can take three forms. They can either be a list of statements, a nested question, or they can be the keyword repeat. Repeat must be the last expression in the Answer Expression. Only one question per Answer Block is allowed.

answer-expression:

statement answer-expression
question
repeat;

2.10 Program Footer

The program footer will be run after all questions are evaluated. This will be the place where further variable assignments can be done and output operations will be performed. It consists of a list of statements. The program footer is optional.

footer:

statement$_{opt}$ footer

3 Output

There is one way to print output in a GAQ program.

print (string);

This syntax will print the string inside of the parenthesis to stdout. The format of the string is the same as in the question syntax and will do variable substitution before printing the results. This form of printing can be done anywhere in the program where statements are allowed – inside the answer block or in the footer.

4 Casting

Implicit casting is done in two places. When parsing an answer from a user, the string typed in will be cast to the response type. Also, when printing out the contents of variables inside a print statement, variables will be implicitly cast to strings. No explicit casting by the programmer is allowed.

Casting to string – when printing out variables

- *bool* will be cast to the strings “0” for false and “1” for true.
- *int* will be cast to a string representation of an int, with an optional ‘– ‘ for negative numbers, followed by a series of digits.
- *float* will be displayed based on the default representation used by cout on your system with the c++ compiler that is used to build the program.

Casting to an int, bool, or float when parsing the user’s response

Casting will be done based on the implementation of cin for the given type of the c++ compiler that you are using. If the attempt to stream the response into an object of the given type fails, then the question will be repeated. Thus, if the program specified that the response should be an *int* and the user enters “three”, the program will continue asking the question over again until the user enters the response in the correct format.

Project Plan

To execute this project, we split the work into front-end, and back-end. First, we wrote the representation of the abstract syntax tree of the program, based on the LRM. Then, we worked independently on getting the front-end and backend working enough to parse the sample hello-world program above. For the front end, we wrote the lexer and parser, while for the backend, we focused on converting the AST to a second AST to represent the C++ program we convert into, as well as output that second AST into valid C++. We then hooked the two parts together and got the entire hello-world program working end-to-end. From there, we proceeded to add each language feature, one at a time, to both the front and back ends, writing a regression test for each one. As new features were added, the old regression tests were run, and were updated if necessary. Then, after getting the core of the language working, we added some more complicated test programs, and then added error-handling to ensure that we catch misuses of the language, adding tests for each negative case to ensure that the compiler flags it correctly.

Project Timeline

Milestone	Date
Proposal	June 8
LRM	June 29
AST designed	July 18
Svn Setup	July 20
Hello World	July 29
All Language Features Complete with Regression Tests	August 7
Error Handling Complete	August 12
Final Report Complete	August 16

Software Development Environment

Since we had 2 developers working on the same code, we needed to agree on an architecture as well as version control system to use. We used a unix-environment, Ayla working on a Mac with OSX, Esther working on a Windows box running Cygwin. An online SVN repository was set up using GoogleCode. The only tools required were an implementation of the ocaml suite, bash, and g++.

Project Log

The basic work-breakdown was that the front-end work was done by Ayla, the backend work by Esther with some language features implemented by Ayla, the error handling by Ayla, and the report by Esther. Of course, this is a rough sketch of the work, and both of us looked over and contributed to every part of the project.

Milestone	Date	Owner
Ast.ml completed	July 18	Ayla
SVN setup	July 20	Esther
astc.ml completed	July 21	Esther
Initial lexer/parser implementation	July 27	Ayla
Initial backend setup	July 28	Esther
Hello World complete	July 29	Esther
Global environment for variable declaration storage	July 31	Esther
Support for questions in front-end	August 2	Ayla
Printing of strings and variables	August 3	Esther
Backend of expressions and statements	August 5	Esther
Full front-end support	August 6	Ayla
Full back-end support	August 7	Esther
Error handling in declarations	August 7	Ayla
Type and scope checking	August 11	Ayla
Project writeup	August 16	Esther

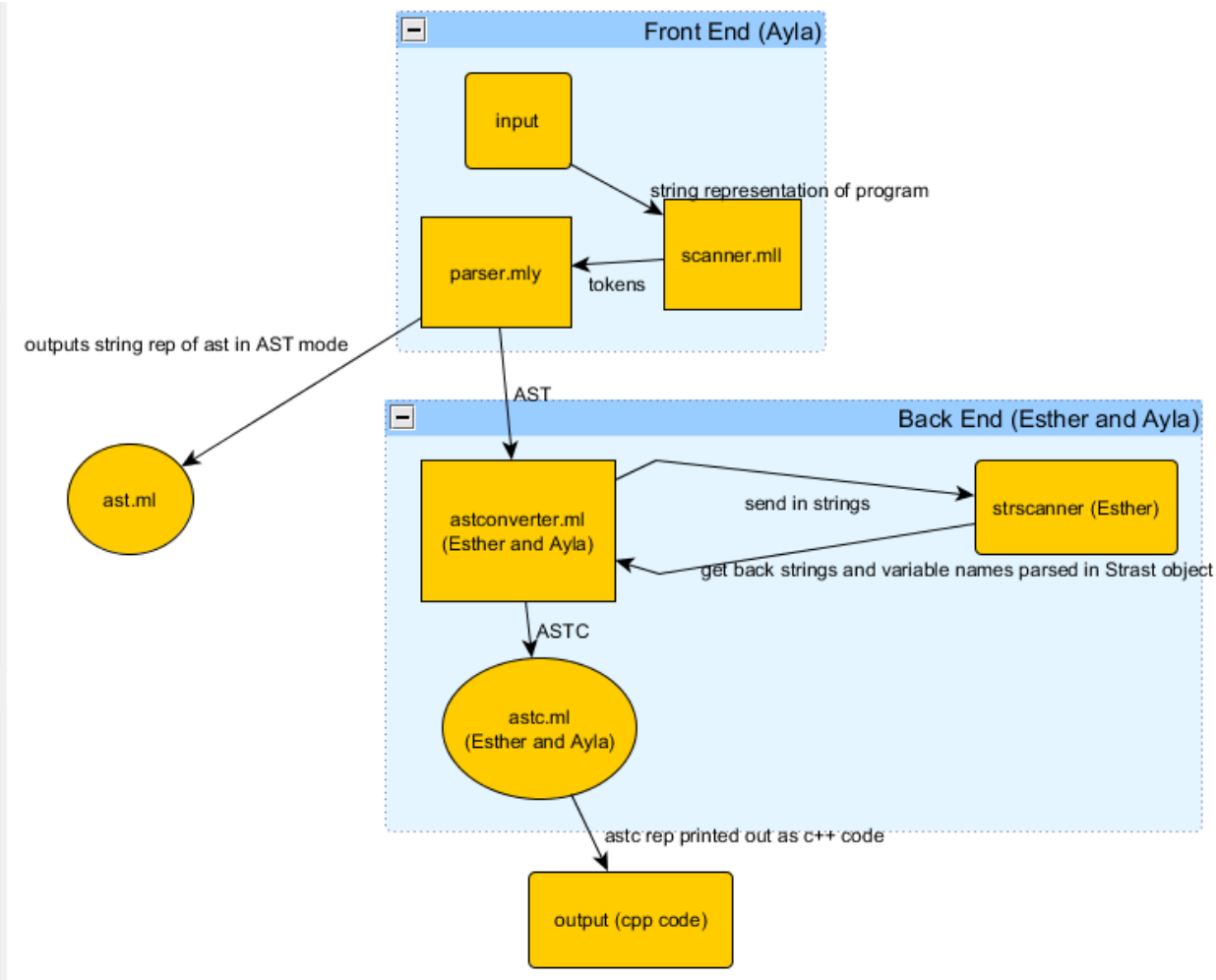
Style Guide

To make a more uniform coding convention and the code easier to read, we have used the following conventions:

- Tabs are not used, but are set to be 3 spaces wide. This ensures that the code looks the same no matter which editor is being used. There are settings for vi and emacs to set the tab width, consult the documentation.
- Nested functions are indented once
- Lines should not exceed 80 characters

- If lines need to be split, they can be split when a function call is made, and its parameters will appear on the following line indented one tab width past the start of the function call
- Comments exceeding one line should both open and close at the beginning of a line
- When matching, each possible match should be on a separate line and they should be aligned
- Each function or variable declaration should be preceded by a comment describing its purpose

Architectural Design



Interfaces

The main interfaces used are the ast.ml and astc.ml. ast.ml represents the structure of the input and is the output of the front-end lexer and parser and is the input of the backend translator. The ast reflects the structure of the gaq program code. The backend, in its turn, converts the ast representation into the astc.ml representation which is a representation of the structure of the c++ code that will need to be output. The backend uses the astconverter to convert from the ast objects into corresponding astc objects. The backend also uses its own small lexer to parse strings to determine where there are tokens that refer to variables. A small strast is then used to represent the ordered list of string and variable name objects, which will be parsed into the astc appropriately as well. The full astc tree representation is then in turn printed out in string format where the string representation is a valid C++ code.

Test Plan

As mentioned above, the testing for this project was done in stages. As support for each feature was added, a corresponding test program was written for it, and compiled. As other features were added, the old programs were recompiled and compared with the old output to see if anything changed, and all changes were reported. When the changes were legitimate, such as when the design of the string output was slightly modified, all old output was updated and retested. Since our compilation is a two-step process, where we first translate a GAQ program into c++, and then compile the c++, the testing also involved two steps. As each feature was added, a sample test program was translated into c++. The output was then compiled into binary and run, with the results tested by hand. Once we were satisfied with the results, the cpp file was committed to svn and all future regression tests were only translated and compared against the golden copy. If the c++ representation was changed for any files, then the testing by hand needed to be redone with a new golden copy of c++ saved, and then future regression tests were run against those copies. The regression tests were run via a shell script that would run each test in a test_cases directory through the gaq compiler and do a diff with the output vs the golden copy, and report which tests succeeded, and which tests failed. The tests were chosen by having one test implement each feature in the language that was described in the ast. As each feature was implemented, a test case was added for it by the implementer.

Here is a sample test program for testing the printing of int variables:

```
int a = 5;

print("a=$a.");
```

The corresponding C++ program is:

```
#include <iostream>

#include <string>

#include <sstream>

using namespace std;

 int main() {

std::string response;

bool _break = false;

int a = 5;

std::cout << "a" << "=" << a << "." << std::endl;

}
```

Another test program was used to ensure that the add operator worked.

```
int y = 100;

"question one" int

[response < y + 2] {

  repeat;

}
```

The corresponding c++ code is:

```
 #include <iostream>

#include <string>

#include <sstream>

using namespace std;

 int main() {

std::string response;

std::string thisresponse = "";

bool _break = false;

int y = 100;

while (!_break) {

std::cout << "q" << "u" << "e" << "s" << "t" << "i" << "o" << "n" << "
" << "o"

<< "n" << "e" << std::endl;

std::getline(std::cin, response);

int thisresponse;

std::istringstream ss(response);

if(!(ss >> thisresponse)) {

std::cout << "Invalid input." << std::endl;

continue;

}
```

```
if ((thisresponse)  <  ((y)  +  (2))) {
continue;

}
else {
break;

}

}

}
```

The regression testing script code was this:

```
#!/bin/bash

for file in `ls *gaq`
do
 pathfilename=${file%.*}
 echo "Running test $pathfilename"
 # echo $pathfilename
 ../gaq -c < $file > ${pathfilename}.new.cpp
 if [ $? -ne 0 ]; then
    echo "Compilation failed! Test $pathfilename failed!"
    continue
 fi
```

```
 diff ${pathfilename}.new.cpp ${pathfilename}.cpp > $
{pathfilename}.diff

 if [ $? -ne 0 ]; then

    echo "Output of compilation is different, Test $pathfilename
failed!"

    continue

 fi

 echo "Test $pathfilename passed!"

done
```

A second set of tests was constructed to check error conditions. Any rule that was added to the code that threw a Failure exception had a corresponding test case added for it. These negative test cases were added to make sure exceptions were thrown for programs that might get through the parser but are not allowed by the language. Most of these were constructed by Ayla who worked on the error handling. One such example is redeclaring variables. The sample code follows:

```
string g = "string1";

int g = 56;
```

Lessons Learned

There were many unexpected issues faced by our team throughout the project. One of the first issues encountered was just the complexity of learning a new language, especially the first functional language either of us used. A lot of time was spent on just figuring out how to use the language. Another unexpected stumbling block was the complexity of getting a correct scanner and parser to work. This took much more time than originally anticipated. Also, the complexity of typed variables was underestimated, as well as the complexity of our language as a whole. Every aspect of your language – in fact, every line you add to the LRM – adds hours of coding and checking inside the compiler.

Our advice for future teams includes starting early and getting a basic scanner and parser for your language worked out as soon as possible. Also, leaving an extra week for testing and error handling is a must, we are very happy that we did so. Working through some extra example programs in O'Caml before embarking on the compiler would probably have been very useful before starting as well.

Appendix

scanner.mll

```
{ open Parser }

rule token = parse
   [' ' '\t' '\r' '\n'] { token lexbuf }
|  "/*"               { comment lexbuf }
|  '('                { LPAREN }
|  ')'                { RPAREN }
|  '['                { RSBRACE }
|  ']'                { LSBRACE }
|  '{'                { LBRACE }
|  '}'                { RBRACE }
|  ';'                { SEMI }
|  '+'                { PLUS }
|  '*'                { TIMES }
|  '='                { ASSIGN }
|  "=="          { EQ }
|  "!="               { NEQ }
|  "<="               { LEQ }
|  ">="               { GEQ }
|  '<'                { LT }
|  '>'                { GT }
|  '-'                { MINUS }
|  '/'                { DIVIDE }
|  '&'                { AND }
|  '|'                { OR }
|  "default"     { DEFAULT }
|  "repeat"           { REPEAT }
|  "print"       { PRINT }
|  "int"         { INT }
|  "float"       { FLOAT }
|  "string"           { STRING }
|  "bool"        { BOOL }
|  "true"        { TRUE(true) }
|  "false"     { FALSE(false) }
|  ['0'-'9']+ as n    { LITERAL(int_of_string n) } (*integers*)
(*floating points*)
|  (['0'-'9']+ ?)'.'['0'-'9']+('e' ('+'|'-')?['0'-'9']+)? as lxm
{FLOATPT(float_of_string lxm) }
|  ['0'-'9']+('.')(['0'-'9']+ ?)('e' ('+'|'-')?['0'-'9']+)?  as lxm
{FLOATPT(float_of_string lxm) }
```

```
|  ['0'-'9']+('e' ('+'|'-')?['0'-'9']+) as lxm {FLOATPT
(float_of_string lxm) }
|  ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
(*identifiers*)
|  '\"'[^'\"']*'\"'   as lxm { STR(lxm) } (* strings *)
|  eof          { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
char)) }

(*ignore comments*)
and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

parser.mly

```
%{ open Ast %}


%token LPAREN RPAREN RSBRACE LSBRACE LBRACE RBRACE SEMI PLUS TIMES
ASSIGN EQ
%token NEQ LEQ GEQ LT GT MINUS DIVIDE AND OR DEFAULT REPEAT RTFILE
%token RTSTDOUT PRINT INT FLOAT STRING BOOL TRUE FALSE EOF
%token <int> LITERAL
%token <bool> TRUE FALSE
%token <string> ID
%token <string> STR
%token <float> FLOATPT

%token <Ast.exprn> EXP

/*precedence rules*/
%nonassoc NOQUESTION
%right  ASSIGN
%left  AND OR
%left  EQ NEQ
%left  LT GT LEQ GEQ
%left  PLUS MINUS
%left  TIMES DIVIDE
%left BOOL INT STRING FLOAT

%start program
%type <Ast.program> program

%%
```

```
/*
a GAQ program consists of variable declaration followed by
questions followed by statements. we enforce this order here
*/
program:
/*nothing*/                    {([],[],[])}
| program var_decleration { let (v, q, e) =
      $1 in match q with
                [] -> (match e with
                       [] -> ($2 :: v, q, e)
                       |h::t ->
                          raise (Failure("declaration out of order")))
            |head::tail -> raise (Failure("out of order"))}
| program question           { let (v, q, e) =
      $1 in match e with
             []->  (v, $2 :: q , e)
                 |h::t -> raise (Failure("question out of order"))}
| program statement       { let (v, q, e) = $1 in (v, q, $2 :: e) }



/*
parse variable declaration. We already enforce correct types at
this point
*/
var_decleration:
INT ID ASSIGN LITERAL SEMI {IntDeclaration ("int", $2, $4)}
| INT ID ASSIGN LPAREN MINUS LITERAL RPAREN SEMI {IntDeclaration
("int", $2, -1*$6)}
| FLOAT ID  ASSIGN LPAREN MINUS FLOATPT RPAREN SEMI { FloatDeclaration
("float", $2, -1.0*.$6) }
| FLOAT ID  ASSIGN FLOATPT SEMI { FloatDeclaration ("float", $2, $4) }
| STRING ID ASSIGN STR SEMI { StringDeclaration ("string", $2, $4) }
| BOOL ID ASSIGN TRUE SEMI { BoolDeclaration ("bool", $2, true) }
| BOOL ID ASSIGN FALSE SEMI { BoolDeclaration ("bool", $2, false) }

/*expressions*/
exprn:
  LITERAL          { Literal ($1) }
| ID               { Id ($1) }
| STR              { Str ($1) }
| FLOATPT          { Float ($1) }
| TRUE             { Bool(true) }
| FALSE             { Bool(false) }
/* binary operations */
| exprn EQ exprn    { Binop($1, Equal, $3) }
| exprn NEQ exprn   { Binop($1, Neq, $3) }
| exprn LT exprn    { Binop($1, Less, $3) }
```

```
| exprn LEQ exprn   { Binop($1, Leq, $3) }
| exprn GT exprn    { Binop($1, Greater, $3) }
| exprn GEQ exprn   { Binop($1, Geq, $3) }
| exprn PLUS exprn  { Binop($1, Add, $3) }
| exprn MINUS exprn { Binop($1, Sub, $3) }
| exprn TIMES exprn { Binop($1, Mult, $3) }
| exprn DIVIDE exprn { Binop($1, Div, $3) }
| exprn AND exprn   { Binop($1, And, $3) }
| exprn OR exprn    { Binop($1, Or, $3) }
/* assigmnet */
| ID ASSIGN exprn { Assign($1, $3) }
/*
negative nums: to save a headache later in the compiler, we filter out
the forbidden negation types already here. Not very elegant, but
that's life
*/
| LPAREN MINUS LITERAL RPAREN {IntNegation($3)}
| LPAREN MINUS FLOATPT RPAREN {FloatNegation($3)}
| LPAREN MINUS ID RPAREN      {IdNegation($3)}
| LPAREN PLUS FLOATPT RPAREN  {Float ($3)}
| LPAREN PLUS LITERAL RPAREN  {Literal ($3)}
| LPAREN PLUS ID RPAREN       {Id ($3)}
| LPAREN exprn RPAREN         {Exprn($2)}

/* variable types */
vartype:
  INT     {"int"}
| STRING  {"string"}
| BOOL       {"bool"}
| FLOAT   {"float"}

/*statement list*/
statement_list:
                                    { [] }
| statement_list statement { $2 :: $1 }

/*
a statement can be a print statement or an
expression followed by a semicolon
*/
statement:
   exprn SEMI                       { Expr($1) }
|  PRINT LPAREN STR RPAREN SEMI    { Print($3) }

/*question*/
question:
  STR vartype answer_list   {$1, $2, List.rev $3}
```

```
/*answer*/
answer:
  RSBRACE exprn LSBRACE LBRACE answer_block RBRACE {$2, $5}
| RSBRACE DEFAULT LSBRACE LBRACE answer_block RBRACE {Bool(true), $5}

/*answer list*/
answer_list:
                                  { [] }
 |answer_list answer { $2 :: $1 }

answer_block:
/*no nested question*/
   statement_list %prec NOQUESTION  {Block(List.rev $1)}
/*repeat statementmust applear at the end od the answer block*/
|  statement_list REPEAT SEMI       {Repeat(List.rev $1)}
/* nested question */
|  statement_list question          {AnswBlock(List.rev $1, $2)}
```

ast.ml

```
(* binary operators *)

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

(* expressions *)
type exprn =
    Literal of int
  | Str of string
  | Id of string
  | Float of float
  | Bool of bool
  | Assign of string * exprn
  | Binop of exprn * op * exprn
  | IntNegation of int
  | FloatNegation of float
  | IdNegation of string
  | Exprn of exprn
  | Noexpr

(* the type of a variable is stored as a string *)
type vartype = string

(*
a statement can be either a print or expression.  since we don't have
any function calls in our langage, it was easier to represent the
print directly in the tree rather than make it a special case in the
```

```
interpretation layer
*)
type statement =
    Print of string
   | Expr of exprn

(* we have different types of variable declarations *)
type var_decleration =
    IntDeclaration of string * string * int
|   FloatDeclaration of string * string * float
|   StringDeclaration of string * string * string
|   BoolDeclaration of string * string * bool

(* statement lists are used in the answer block or footer *)
type statement_list = statement list

(*
Since questions and answers are nested, need to be declated
with "and" so that they can refer to eatch other
*)
type question = string * vartype * answer_list
and answer = exprn * answer_block
and answer_block =
    Block of statement_list
   |AnswBlock of statement_list * question
   |Repeat of statement_list
and answer_list = answer list

(* Top-level program *)
type program = var_decleration list * question list * statement list


(*********************************************
 Below are all the print function
    for debugging/display of the ast
*********************************************)

(* print an expression *)
let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Id(s) -> s
  | Str (s) -> s
  | Float (s) -> string_of_float s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
        Add -> "+"
      | Sub -> "-"
      | Mult -> "*"
```

```
      | Div -> "/"
      | Equal -> "=="
      | Neq -> "!="
      | And -> "&"
      | Or -> "|"
      | Less -> "<"
      | Leq -> "<="
      | Greater -> ">"
      | Geq -> ">=") ^ " " ^
      string_of_expr e2
  | Bool (s) -> string_of_bool s
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | IntNegation (i) -> "(-" ^ string_of_int i ^ ")"
  | FloatNegation (i) -> "(-" ^ string_of_float i ^ ")"
  | IdNegation (i) -> "(-" ^ i ^ ")"
  | Exprn (e) -> string_of_expr e
  | Noexpr -> ""


(* print variable decleration *)
let string_of_var_decleration = function
   IntDeclaration (t,i,v) -> t ^ " " ^ i ^ " " ^ " = " ^
                             string_of_int v ^ ";\n"
|  StringDeclaration (t,i,v) -> t ^ " " ^ i ^ " " ^ " = " ^ v ^ ";\n"
|  FloatDeclaration (t,i,v) -> t ^ " " ^ i ^ " " ^ " = " ^
                               string_of_float v ^ ";\n"
|  BoolDeclaration  (t,i,v) ->  t ^ " " ^ i ^ " " ^ " = " ^
                                string_of_bool v ^ ";\n"


(* print a statement *)
let rec string_of_statement = function
   Print (str) -> "print" ^ "(" ^ str ^ ");\n"
|  Expr (e) -> string_of_expr e ^ ";\n"
(* print an answer *)
and string_of_answer = function
    (e, ans_block) -> "condition: if(" ^ string_of_expr e ^ ")\n" ^
                      string_of_answer_block ans_block
(* print a question *)
and string_of_question =  function
     (str, tp, ans) ->  "" ^ str ^ "(" ^ tp ^ ")\n" ^
                           String.concat "\n" (List.map
string_of_answer ans)
(* print an answer_block *)
and string_of_answer_block = function
     Block (sl) ->
        "" ^ String.concat "" (List.map string_of_statement sl)
   | AnswBlock (sl, q) ->
      String.concat "\n" (List.map string_of_statement sl) ^
```

```
      "\n" ^ string_of_question q
   | Repeat (sl) ->
      String.concat "\n" (List.map string_of_statement sl) ^
      "Repeat;\n"

(* print a qaq program devided to 3 parts *)
let string_of_program (vars , qs, statem ) =
  "DECLERATIONS:\n" ^ String.concat "" (
     List.map string_of_var_decleration (List.rev vars) )
     ^ "\n" ^
  "QUESTIONS:\n" ^ String.concat "\n" (
     List.map string_of_question (List.rev qs)) ^ "\n" ^
  "STATEMENTS:\n" ^String.concat "" (
     List.map string_of_statement (List.rev statem) ) ^ "\n"
```

astc.mll

```
(* c++ binary operators *)
type cop = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater
         | Geq | And | Or

(* c++ expressions *)
type cexpr =
    CLiteral of string
  | CVar of string
  | CStr of string
  | CBincop of cexpr * cop * cexpr
  | CAssign of string * cexpr
  | CNegate of string
  | CNegateId of string
  | CNoexpr

(* output contains either strings or identifiers of variables *)
type atom =
    CId of string
  | CString of string

(* A list of atoms that will be passed to std::cout *)
type cout =
    CStrings of atom list

(* a c++ statement *)

(* cstmt definition *)
type cstmt =
```

```
    CBlock of cstmt list
  | CExpr of cexpr
  | CReturn of cexpr
  | CWhile of cstmt
  | CIf of cexpr * cstmt
  | CElse of cstmt
  | Cout of cout
  | CIn of string
  | Continue
  | Break
  | BreakAll

(* The toplevel representation of a c++ program *)
type cprogram = cstmt

(************************************************************
  Symbol Table
************************************************************)
(* need a stringmap to keep track of the declared variables *)
module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
    (* type of each variable *)
    variable_types : string StringMap.t;
    (* value as string of each variable*)
    variable_values: string StringMap.t;
    (* response type in the current scope *)
    response_type: string;
    (* string value of the response in current scope *)
    response_value: string;
  }

(************************************************************
  Printing functions for all types that will be used to output
  the c++ program
************************************************************)
(* prints out the variables as declarations in the C++ code *)
let print_env env =
     StringMap.fold
     (fun key value data ->
        (Printf.sprintf "%s %s = %s;\n"  value key
        (StringMap.find key env.variable_values) ) ^ data)
      env.variable_types ""



(* defines the top-level function for printing a cprogram
   based on the variables already parsed *)
```

```
let string_of_cprogram = function
    (statements, myenv) ->
let string_of_cop = function
    Add -> " + "
  | Sub -> " - "
  | Mult ->  " * "
  | Div -> " / "
  | Equal -> " == "
  | Neq -> " != "
  | Less -> " < "
  | Leq -> " <= "
  | Greater -> " > "
  | Geq -> " >= "
  | And -> " && "
  | Or -> " || "
in

(*
function get_variable
looks up var name in myenv, return var name if valid
*)
let get_variable var =
   if ((String.compare var "response") == 0 ) then
      "thisresponse"
   else
     (try StringMap.find var myenv.variable_values;
     var with Not_found ->
         raise (Failure ("undeclared variable " ^ var)))
in
(* convert cexpr to a c++ string *)
let rec string_of_cexpr = function
    CLiteral (s) -> s
  | CVar (s) -> get_variable s
  | CStr(s) -> "std::string (" ^ s ^ ")"
  | CBincop(e, o, e2) ->
        "(" ^ string_of_cexpr(e) ^ ") " ^ string_of_cop(o) ^
        " (" ^string_of_cexpr(e2) ^ ")"
  | CAssign (s, e) ->
     (get_variable s) ^ " = (" ^ string_of_cexpr(e) ^ ")"
  | CNegate (b) -> "-" ^ b
  | CNegateId (id) -> let vbl = get_variable id in "-" ^ vbl
  | CNoexpr ->  ""
in
(*
this is the part that prints out the types parsed in from strscanner
*)
let string_of_printstring = function
   Strast.Variable(s) ->
      get_variable (String.sub s 1 (String.length(s)-1))
```

```
 | Strast.Str(s) -> "\"" ^ s ^ "\""
in
(* here is were we print things out to cout *)
let string_of_atom = function
    CId(s) -> StringMap.find s myenv.variable_values
(* here's the tricky part where we find the $<id> embedded in srings
*)
  | CString(s) -> let lexbuf = Lexing.from_string s in
     Strscanner.toklist.contents <- [];
     Strscanner.tokenize lexbuf;
     String.concat " << " (List.map string_of_printstring (
     List.rev Strscanner.toklist.contents) )
in
let string_of_cout = function
  CStrings(s) -> "std::cout << " ^
  String.concat " << " (List.map string_of_atom s) ^ " << std::endl;"
in
(* c++ code for all kinds of statements *)
let rec string_of_cstmt = function
    Cout(s) -> string_of_cout(s)
  | CBlock(s) -> String.concat "\n" (List.map string_of_cstmt s)
  | CExpr(e) -> string_of_cexpr(e) ^ ";"
  | CReturn(e) -> "return (" ^ string_of_cexpr(e) ^ ");\n"
  | CWhile(s) -> "while (!_break) {\n" ^
                      string_of_cstmt(s)  ^ "\n}\n"
  | CIf (e, s) -> "if (" ^ string_of_cexpr(e) ^ ") {\n" ^
       string_of_cstmt(s) ^ "\n}\n"
  | CElse (s) -> "else {\n" ^ string_of_cstmt(s) ^ "\n}\n"
  | CIn(t) -> if((String.compare t "string") == 0) then
                ("std::getline(std::cin, response);\n" ^
"std::string" ^
                 " thisresponse(response);\n" )
              else (ignore(myenv.response_type = t);
                  "std::getline(std::cin, response);\n" ^ t ^
                  " thisresponse;\n" ^
                  "std::istringstream ss(response);\n " ^
                  "if(!(ss >> thisresponse)) {\n" ^
                  "std::cout << \"Invalid input.\" << std::endl; \n "
^
                  "continue;\n}\n")
  | Continue -> "continue;\n"
  | Break -> "break;\n"
    (* use this to break out of all questions. *)
  | BreakAll -> "_break = true;\n"
in
(*
here is the main body of compile that uses all the above declared
functions
*)
```

```
        "#include <iostream>\n#include <string>\n" ^
        "#include <sstream>\nusing namespace std;\n int main() {\n" ^
        "std::string response;\n" ^
        "std::string thisresponse = \"\";\n" ^
        "bool _break = false;\n" ^
        (print_env myenv) ^
        string_of_cstmt (statements) ^
        "\n}\n"
```

actconverter.ml

```
(* Esther Kundin & Ayla Brayer

   This program has all of the functions that convert from the objects
in Ast to the objects in Astc *)
open Ast
open Astc

(************** Global environment - reference will be set in main
func ****)
let globalEnv = ref {Astc.variable_types = StringMap.empty;
                     Astc.variable_values = StringMap.empty;
                     Astc.response_type = "string";
                     Astc.response_value = "garbage" }


(*let get_variable_type var =
   if ((String.compare var "response") == 0) then
globalEnv.contents.response_type else
     (try StringMap.find var globalEnv.contents.variable_types
      with Not_found -> raise (Failure ("undeclared variable " ^
var))) *)
(*************************************************************************)


(******** error / type checking functions ***************)

(*
function get_vbl_type - returns variable type (as string)
param var - variable name (string)
param t   - type of the in scope response variable,
               if called outside a question "out_of_scope" is expected
```

```
*)
let get_vbl_type var t =
   if ((String.compare var "response") == 0) then
      if ((String.compare t "out_of_scope") == 0) then
          raise (Failure ("Accessing 'response' out of scope"))
      else
          t
   else (* raise exception if the variable was not found *)
      (try StringMap.find var globalEnv.contents.variable_types with
Not_found
       -> raise (Failure ("undeclared variable " ^ var)))


(*
function get_expression_type - returns the type of a given expressions
while
         checking the validity of the expression
param e - Ast.exprn
param t - type of the in scope response variable,
            if called outside a question "out_of_scope" is expected
*)
let rec get_expression_type e t = match e with
    Literal(i) ->"int"
   |Bool(s) -> "bool"
   |Float(s) ->"float"
   |Id(id) -> let rt = get_vbl_type id t in rt
   |Noexpr -> raise (Failure ("missing expression"))
   |Str(s) ->"string"
   |Exprn(e) ->  let rt = get_expression_type e t in rt
   |IntNegation (i) -> "int"
   |FloatNegation (f) -> "float"
   |IdNegation (id) -> (* negation is only valid on int or float type
*)
      let tp = get_vbl_type id t in
      if ((String.compare tp "float") == 0 || (String.compare tp
"int") == 0) then
         tp
      else
         raise (Failure ("illigal negation (allowed only on int or
float type)"))
    |Assign (id,e2) ->
       if ((String.compare id "response") == 0) then
          raise (Failure ("illigal assignemt to 'response' variable"))
       else
          let rhs = get_vbl_type id t in
          let lhs = get_expression_type e2 t in
          if ((String.compare rhs lhs) == 0) then
             rhs
          else
```

```
            raise (Failure ("illigal " ^ rhs ^ " to " ^
            lhs ^ " assignemt"))
    |Binop (e1, o, e2) ->
       let rhs = get_expression_type e1 t in
       let lhs = get_expression_type e2 t in match o with
          (* mathematical operators *)
          Ast.Add | Ast.Mult | Ast.Sub | Ast.Div ->
(*
for the above operations to be legal, sub-expr must be of same type
*)
             if ((String.compare rhs lhs) == 0) then
    (* the above operations are not legal on strings & booleans *)
               if (((String.compare rhs "string") == 0 ||
                    (String.compare rhs "bool") == 0 )) then
                       raise (Failure (
                       "illigal operation preformed on " ^ rhs))
                 else
                    rhs
             else
               raise (Failure (
               "illigal " ^ "rhs" ^ " on " ^ "lhs" ^ " operation"))
            (* comparison operators *)
    |Ast.Less | Ast.Leq | Ast.Greater | Ast.Geq ->
(*
for the above operations to be legal, sub-expr must be of same type
*)
        if ((String.compare rhs lhs) == 0) then
       (* the above operations are not legal on strings & booleans *)
           if (((String.compare rhs "string") == 0 ||
                 (String.compare rhs "bool") == 0 )) then
             raise (Failure (
                  "illigal operation preformed on " ^ rhs))
           else
             "bool" (* return bool *)
         else
            raise (Failure (
            "illigal "^" rhs "^" on "^" lhs " ^ "operation"))
      |Ast.Equal | Ast.Neq -> (*expressions must be of same type*)
             if ((String.compare rhs lhs) == 0) then
                rhs
             else
                raise (Failure (
                "" ^ rhs ^ " cannot be comapred with " ^ lhs))
       (* logical operators *)
      |Ast.And | Ast.Or ->
       (*logical operators cannot be applied on strings*)
          if (((String.compare rhs "string") == 0 ||
               (String.compare lhs "string") == 0 )) then
             raise (Failure ("&&/|| applied to type string"))
```

```
          else
              "bool" (* return type of logical expressions is bool *)



(*
function - check_answer_expression
  The answer expression may evaluate to a boolean,
  int or float but not a string. Also checks that the
  expression is in accordance with the type rules
param t - type of the in scope response variable,
             if called outside a question "out_of_scope" is expected
param a  - Ast.answer
*)
let rec check_answer_expression t = function
   (expn, ab) -> match expn with
      Literal(i) -> t (* stand alone int is ok *)
      |Bool(s) ->  t  (* stand alone bool is ok *)
      |Float(s) -> t  (* stand alone float is ok *)
      |Id(id) -> let tp = get_vbl_type id t in
         if ((String.compare tp "string") == 0) then
            raise (Failure ("string type is not allowed in answer
expression"))
         else
            t
      |Exprn(e) -> let rt = get_expression_type e t in
         if ((String.compare rt "string") == 0) then
            raise (Failure
            ("string type is not allowed as answer expression"))
         else
            t
      |Noexpr -> raise (Failure ("missing answer-expression"))
      |Str(s) -> raise (Failure (
         "string is not allowed as an answer expression"))
      |Assign(id, ex) -> let tp = get_expression_type ex t in
         if ((String.compare tp "string") == 0) then
            raise (Failure (
            "string type is not allowed as answer expression"))
         else
            tp
      |Binop (e1, o, e2) -> let tp = get_expression_type expn t in tp
      |IntNegation (i) -> t
      |FloatNegation (f) -> t
      |IdNegation (id) -> let tp = get_vbl_type id t in
         if ((String.compare tp "float") == 0 ||
             (String.compare tp "int") == 0) then
               tp
         else
```

```
            raise (Failure (
            "illigal negation (allowed only on int or float type)"))


(*
function - check_types - check validity of the answer's expressions
param t - type of the in scope response variable,
             if called outside a question "out_of_scope" is expected
param answerlist - Ast.answer_list
*)
let check_types = function
    (t, answerlist) ->
       ignore (List.fold_left check_answer_expression t answerlist)




(**** Main conversion functions that convert from ast to astc *****)



(*
function cop_of_op - Ast to Astc conversion
*)
let cop_of_op = function
    Ast.Add -> Astc.Add
  | Ast.Sub -> Astc.Sub
  | Ast.Mult -> Astc.Mult
  | Ast.Div -> Astc.Div
  | Ast.Equal -> Astc.Equal
  | Ast.Neq -> Astc.Neq
  | Ast.Less -> Astc.Less
  | Ast.Leq -> Astc.Leq
  | Ast.Greater -> Astc.Greater
  | Ast.Geq -> Astc.Geq
  | Ast.And -> Astc.And
  | Ast.Or -> Astc.Or


(*
function cexpr_of_expr - converts expresions to Astc type
*)
let rec cexpr_of_expr = function
    Literal(i) -> Astc.CLiteral (string_of_int i)
  | Id(s) -> Astc.CVar (s)
  | Str(s) -> Astc.CStr(s)
  | Float(s) -> Astc.CLiteral(string_of_float s)
  | Bool (s) -> Astc.CLiteral(string_of_bool(s))
```

```
  (*For assignments, need to make sure that we're not assigning to
response*)
  | Assign (id, e) ->
       Astc.CAssign(id, (cexpr_of_expr e))
  | Binop (e1, o, e2) ->
       Astc.CBincop (cexpr_of_expr e1, cop_of_op o, cexpr_of_expr e2)
  | IntNegation (i)   -> Astc.CNegate (string_of_int i)
  | FloatNegation (f) -> Astc.CNegate (string_of_float f)
  | IdNegation (id)   -> Astc.CNegateId id
  | Exprn(e) -> cexpr_of_expr(e)
  | Noexpr -> Astc.CNoexpr


(*
function validate_statements - make sure
    expressions are error free
param sl - Ast.statement_list
param t  - type of the in scope response variable,
            if called outside a question "out_of_scope" is expected

*)
let rec validate_statements sl t = match sl with
    [] ->  ""
    | hd::tl -> match hd with
                   Ast.Print(s) -> validate_statements tl t
                  |Ast.Expr (e) -> ignore (get_expression_type e t);
                                   validate_statements tl t


(* function cstmt_of_statement - convert statement to cstatement *)
let cstmt_of_statement = function
   Ast.Print(s) -> Astc.Cout (Astc.CStrings [ Astc.CString (s) ] )
 | Ast.Expr (e) -> Astc.CExpr (cexpr_of_expr e)


(*
function cstmt_of_answer_block - convert Ast.answer_block to
   Astc.cstmt
*)
let rec cstmt_of_answer_block ab t = match ab with
    Ast.Block (sl) -> ignore (validate_statements sl t);
       ((List.map cstmt_of_statement sl) @ [Astc.BreakAll] )
   | Ast.AnswBlock (sl, q) ->  ignore (validate_statements sl t);
       ((List.map cstmt_of_statement sl) @
       ([cstmt_of_question q]) @ [Astc.Break] )
   | Ast.Repeat (sl) -> ignore (validate_statements sl t);
       ((List.map cstmt_of_statement sl) @ [Astc.Continue])
(*
function cstmt_of_question - convert Ast.question to
```

```
   Astc.cstmt
*)
and cstmt_of_question = function
  (q, t, answers) -> check_types (t, answers); Astc.CWhile (
     Astc.CBlock [ Astc.Cout (Astc.CStrings [ Astc.CString  q ]);
      Astc.CIn t; cstmt_of_answer_list answers t] )

(* function cstmt_of_answer - convert Ast.answer to Astc.cstmt*)
and cstmt_of_answer ans t = match ans with
    (e, ab) -> Astc.CIf (
       cexpr_of_expr(e), (Astc.CBlock(cstmt_of_answer_block ab
t)))
(*
function cstmt_of_answer_list - convert Ast.answer_list to Astc.cstmt
*)
and cstmt_of_answer_list answ t = (* function*) match answ with
    [] ->  Astc.Break
  | hd::tl -> Astc.CBlock [ cstmt_of_answer hd t; Astc.CElse
(cstmt_of_answer_list tl t)]


(**********************************************************************)


(*
function cprogram_of_program
Top level conversion function that will be called from compile.ml,
converst from Ast.program to Astc.program
*)
let cprogram_of_program  = function
   (questions, statements, myenv) -> globalEnv.contents <- myenv;
   ignore(validate_statements statements "out_of_scope");
   Astc.CBlock ( (List.map cstmt_of_question questions) @
   (List.map cstmt_of_statement statements))
```

strast.ml

```
(*
This module describes the parts of a question string - regular
string and references to variables
*)
type printstring =
  Str of string
| Variable of string

type printstrings = Pstrings of printstring list
```

strscanner.mll

```
{

  open Strast
  (*
   here is a list where we keep each token - a Variable
   or a Str, we need a ref so that we can change it otherwise,
   it's immutable
   *)
  let toklist = ref []
}


let identchar = ['A'-'Z' 'a'-'z' '_' '0'-'9']
let tok = '$' identchar+
rule tokenize = parse
 (* deal with escaped $ *)
  | '\\' '$'
     { toklist.contents <- List.append [Str("$")] toklist.contents;
       tokenize lexbuf }
 (* deal with escaped quote *)
  | '\\' '"'
     { toklist.contents <- List.append [Str("\"")] toklist.contents;
     tokenize lexbuf }
 (* deal with other slashes *)
  | '\\'
     { toklist.contents <-
          List.append [Str("\\\\")] toklist.contents;
```

```
       tokenize lexbuf }
 (* ignore quotes *)
  | '"'    { tokenize lexbuf }
 (* ignore newline inside a string (not an explicit newline) *)
  | '\n'    {tokenize lexbuf}
 (* deal with variables referenced with a $ *)
  | tok
      { toklist.contents <-
      List.append [Variable(Lexing.lexeme lexbuf)] toklist.contents;
      tokenize lexbuf }
 (* here is everything else *)
  | _
      { toklist.contents <-
      List.append [Str(Lexing.lexeme lexbuf)] toklist.contents;
      tokenize lexbuf }
  | eof         { () }



(****
the following was used for testing just this module
{
let string_of_printstring = function
   Variable(s) -> print_string ("variable: "); print_endline(s)
 | Str(s) -> print_endline(s)

  let main () =
    let lexbuf = Lexing.from_string "this is a $token test \\$token2 "
in
    tokenize lexbuf;
    Printf.printf "tokens: \n" ;
    List.iter string_of_printstring toklist.contents

  let _ = Printexc.print main ()
}
****)
```

compile.ml

```
open Astconverter

module StringMap = Map.Make(String)


(*
function add_type - takes a var_decl and adds the type
                    to a types map
*)
let add_type types = function
    Ast.IntDeclaration (t, name, v) ->
       if (StringMap.mem name types) then
          raise (Failure ("multiple decleration of variable: " ^
name))
       else
          StringMap.add name t types
  | Ast.FloatDeclaration (t, name, v) ->
     if (StringMap.mem name types) then
        raise (Failure ("multiple decleration of variable: " ^ name))
     else
        StringMap.add name t types
  | Ast.StringDeclaration (t, name, v) ->
     if (StringMap.mem name types) then
        raise (Failure ("multiple decleration of variable: " ^ name))
     else
        StringMap.add name t types
  | Ast.BoolDeclaration (t, name, v) ->
     if (StringMap.mem name types) then
        raise (Failure ("multiple decleration of variable: " ^ name))
     else
        StringMap.add name t types

(*
function add_values - takes a var_decl and adds the value
 to a values map
*)
let add_values values = function
    Ast.IntDeclaration (t, name, v) ->
        StringMap.add name (string_of_int v) values
  | Ast.FloatDeclaration (t, name, v) ->
    StringMap.add name (string_of_float v) values
  | Ast.StringDeclaration (t, name, v) ->
    StringMap.add name v values
  | Ast.BoolDeclaration (t, name, v) ->
    StringMap.add name (string_of_bool v) values
```

```
(*
function check_name -
   here we hard-code all the keywords to check
   variable names against
   All we actually need here is to check for the
   'response' name, all others are already taken care of
   by the parser. With that said, it works and it's too
   late to chance it.
*)
let check_name name = if
   ((String.compare name "default") == 0) or
   ((String.compare name "repeat") == 0) or
   ((String.compare name "response") == 0) or
   ((String.compare name "print") == 0) or
   ((String.compare name "int") == 0) or
   ((String.compare name "float") == 0) or
   ((String.compare name "bool") == 0) or
   ((String.compare name "string") == 0) or
   ((String.compare name "true") == 0) or
   ((String.compare name "false") == 0)  then
   raise (Failure ("Cannot use a keyword as a variable name: " ^
name))
   else ()

(* checks the names of variables to make sure it isn't using keywords
*)
let check_variable_names = function
    Ast.IntDeclaration (t, name, v) -> check_name name
  | Ast.FloatDeclaration (t, name, v) -> check_name name
  | Ast.StringDeclaration (t, name, v) -> check_name name
  | Ast.BoolDeclaration (t, name, v) -> check_name name

(* function compile - compiles Ast.program *)
let compile (var_decl, questions, statements) =
    List.iter check_variable_names var_decl;
    (* get a list of all types declared *)
    let types = List.fold_left add_type StringMap.empty var_decl in
    (* get a list of all values declared *)
    let values = List.fold_left add_values StringMap.empty var_decl
    in
    (* create a scope that has all the types and values for all
    variable names *)
    let myenv = {Astc.variable_types = types;
                 Astc.variable_values = values;
                 Astc.response_type = "string";
                 Astc.response_value = "garbage" }
    in
```

```
    (* for debugging, print them out print_env myenv *)
      let ccode = Astc.string_of_cprogram (
         Astconverter.cprogram_of_program(List.rev questions,
                                          List.rev statements,
                                          myenv),
         myenv)
       in print_string ccode
```

gaq.ml

```
open Compile


type action = Ast | Compile

(* main compilation program *)
let _ =
   let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [("-a", Ast);
                             ("-c", Compile)]

    else Compile in

    let lexbuf = Lexing.from_channel stdin in
    let program = Parser.program Scanner.token lexbuf in

    match action with
         (* print ast *)
         Ast -> let listing = Ast.string_of_program program
                     in print_string listing
           (* compile *)
       | Compile ->  Compile.compile (program)
```