Haskell Computer Algebra System (HCAS) Final Report

Rob Tougher (rt2301@columbia.edu)

December 15, 2007

Contents

1 Introduction

For the class project I implemented a simple computer algebra system, HCAS. HCAS is a purely functional programming language that provides a set of basic operations for constructing and manipulating algebraic expressions. Simply put, you can think of HCAS as **a subset of Haskell, plus support for computer algebra.**

HCAS has three main features:

- **Purely Functional Language.** HCAS is a purely functional subset of Haskell. There are functions, recursion, lists, strings, pattern matches for function arguments, etc. No variables, sequencing of operations, or other items from imperative programming languages.
- **Construction of Mathematical Expressions.** HCAS allows you to construct mathematical expressions using an intuitive syntax. You are able to define mathematical expressions inline. That is, if you write the expression $x + y - z$, this is automatically constructed as a math expression.
- **Navigation of Mathematical Expressions.** HCAS uses the concept of pattern matching in function arguments to allow you to navigate mathematical expressions. Consider the following simple function:

```
printType (left*right) =
        "Multiplication"

printType (left+right) =
        "Addition"
```

  This function has two definitions, one for addition and one for multiplication. The version that gets executed at runtime is chosen based on the expression argument that you pass in. So if you call the function as "printType(x*y+z)", the version for addition will be called, because addition binds the loosest. In the body of the function "left" refers to "x*y", and "right" refers to "z".

2 Language Tutorial

2.1 Command Line Usage

One command line utility is shipped with this project: *hcasi*, the HCAS interpreter. This utility uses standard input for the HCAS input and standard output for the program output:

```
$ ./hcasi < myscript.hcas > out.txt
```

2.2 Hello World!

Every good language tutorial contains a few "Hello World!" programs. The following examples get you started. You can run these using the following syntax:

```
$ echo "main = ..." | ./hcasi
```

The simplest script is the following:

```
main = "Hello World!"
```

An HCAS script contains one or more function declarations. One of the functions must be named "main" and have no arguments. When HCAS interprets your script, it finds the main function, evaluates it, and then returns the result to standard output. In the above example, the main function returns the string "Hello World!'

HCAS allows you to perform basic math calculations like many other languages:

```
main = 3+7*4
```

Here, the script will return 31. All of the typical mathematical operations are supported, including addition, subtraction, multiplication, division, and exponentiation.

HCAS also allows you to use lists:

```
main = [1,2,3,4,5]
```

In the above program, main returns a list of numbers.

Like many of the current popular languages, HCAS allows you to create user-defined functions. These functions can have zero or more arguments:

```
main = foo(5)
foo(x) = x + 7.5
```

Here, main will return the number 12.5.

You are allowed to create multiple definitions for a function. You can give each of these definitions different "patterns" for its arguments. HCAS chooses the function to call based on the arguments you pass to the call. This is called "pattern matching":

```
main = foo(x+y)
foo(left+right) = "This is addition"
foo(left-right) = "This is subtraction"
foo(x) = "This is neither addition nor subtraction"
```

In the above example main will return "This is addition", because the addition expression "x+y" is passed as the first argument to foo.

Besides math patterns, you can also use list patterns:

```
main = head([1,2,3,4,5])
head(x:xs) = x
```

In the above example, "x" refers to the first item of the list, and "xs" refers to the rest of the list. The head function returns the first item of the list, so our main function will return the number 1.

You can also give literal values as patterns:

```
main = foo("hello")
foo("hello") = "Hi!"
foo("goodbye") = "Bye!"
```

Here, the call to foo will match the function declaration that has the string "hello", which returns the string "Hi!". Any literal values can be used in function declarations, including numbers, strings, and lists.

The following is a simple example script that calculates the derivative for a math expression:

```
main = derivative(3*x^2+2*x)

derivative(a+b) = derivative(a) + derivative(b)
derivative(a-b) = derivative(a) - derivative(b)
derivative(c*x^e) = c*e*simplify(x^(e-1))
derivative(c*x) = c
derivative(x) = 0
```

```
simplify(x^1) = x
simplify(x^0) = 1
simplify(x+0) = x
simplify(0+x) = x
simplify(x+y) = simplify(x) + simplify(y)
simplify(x-y) = simplify(x) - simplify(y)
simplify(x) = x
```

If you run this with HCAS, it should print out $6 * x + 2$.

3 Language Manual

3.1 Introduction

HCAS is a purely functional programming language that allows for the manipulation of mathematical expressions. This section describes the language.

3.2 Top-Level Structure

The top level of the HCAS grammar is the *program*. A program is contained in a single source file. A program has one or more *functions*, as shown by the following grammar:

```
program:
    function-list

function-list:
    function
    function function-list
```

Functions are separated in the file by one or more characters of whitespace. Whitespace can either be a single space, carriage return, tab, or comment. Functions are typically declared on separate lines, though they do not need to be.

A function declares a reusable parameterized *expression*. It contains zero or more input expressions and a single output expression:

```
function:
    identifier '(' expression-list ')' '=' expression

expression-list:
    expression
    expression ',' expression-list
```

The *identifier* is the function's name and is how other expressions can call this function (more on identifiers later). The expression-list specifies the list of input expressions for the function, and are available to be used in the body of the output expression. The parentheses and expression-list are optional if no input expressions are needed for the function.

One (and only one) of the functions in a program must be named **main**. When a program is run, the main function is evaluated. Its return value is the return value for the program. If the program does not contain a main element the interpreter will not execute the program and instead print out an error message. The main function should not contain any input expressions.

As mentioned previously, comments are considered whitespace. Comments are started with the open brace character '{' and terminated with the close brace character '}'.

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

```
True False let in
```

3.3 Expressions

3.3.1 Data Types

An expression in HCAS has a value. An expression's value can be one of three main types: math expression, list expression, or boolean expression. The type of an expression is implicitly determined at runtime based on the contents of the expression. For example, if an expression consists of a single string literal, it is considered to be a list expression (a list of character values).

The following is the basic grammar for expressions:

```
expression:
    expression-atom expression-tail
```

```
expression-atom:
    identifier
    string-atom
    number-atom
    etc...

expression-tail:
    ‘+’ expression
    ‘-’ expression
    etc...
```

An expression-atom represents the smallest unit of an expression, like a number, variable, string, or boolean constant. These atoms are combined into larger expressions using expression-tails. Tails contain an operator and a right-side expression. In an expression, an expression-tail is optional. That is, you can have only a single expression-atom in your expression.

The operators in this section are listed in the order of their precedence. For example, the multiplication operator ‘*’ appears before the addition operator ‘+’, which implies that the multiplication operator binds more tightly than the addition operator. With the exception of the exponentiation operator, all operators are left-to-right.

3.3.2 Expression Atoms

identifier An identifier is a sequence of letters, digits, and underscores. The first character must be a letter.

An identifier's type is determined at runtime based on its scope:

- If the identifier is contained inside of a function, and that function has an argument with that same name, the identifier will refer to the value of the argument:

  ```
  foo (x) = x + 3
  ```

  Here, x refers to the function's argument.

- Otherwise, if there is a function declaration with that name, it will refer to a function call. You can optionally specify input expressions in parentheses if the function requires input expressions. Here's an example of a function call:

  ```
  foo = "Hello"
  bar = foo ++ " world!"
  ```

  Here, foo is a function call.

- Otherwise, the identifier represents a variable in a math expression. An example:

```
foo = x + y
```

  Here, x and y are both math variables.

character A character represents a single printable character, and is contained in single quotes:

```
'c'
```

string A string represents a list of characters and begins and terminates with a double quote '"'. You cannot embed double quotes in a string. The following is a string example:

```
"Hello world"
```

A string is semantically equal to a list of characters. Lists are introduced later in this section.

number A number represents a numeric value. It can contain a sequence of digits, followed by a decimal, followed by another sequence of digits. You can omit the first sequence of digits, or the decimal and second sequence, but not both. the following are all valid numbers:

```
4
4.5
.5
1.05
```

boolean The identifiers **True** and **False** are reserved words and represent the boolean values of true and false.

'[' expression-list ']' A list is an array of zero or more items. It consists of an open square bracket '[', followed by zero or more occurrences of an expression separated by a comma, followed by a closing square bracket ']'. The following is a list of three numbers:

```
[1, 2, 3]
```

‘(’ expression ‘)’ You can use parentheses to group mathematical expressions:

```
x*(y+z)
```

‘-’ expression A subtraction operator ‘-’ preceding a math expression will negate the value of that expression.

3.3.3 Boolean Operators

The following binary operators are used for boolean expressions. They evaluate to either true or false.

expression ‘==’ expression This expression evaluates to true if the expression on the right is equal to the expression on the left. For string expressions the strings must contain the same characters; for number expressions the numbers must contain the same value; for boolean expressions the booleans must contain the same value; and for math expressions the expressions must have the same math contents (operators and atoms).

expression ‘>’ expression This expression evaluates to true if the left-hand expression is greater than the right-hand expression. This is valid for strings and numbers, but not for lists or math expressions.

expression ‘<’ expression Represents less than.

expression ‘&&’ expression This expression evaluates to true if both expressions evaluate to true. Both expressions need to be boolean expressions. If one is another type, an error will be returned.

expression ‘||’ expression This expression evaluates to true if at least one of the expressions evaluate to true. Both expressions need to be boolean expressions. If one is another type, an error will be returned.

3.3.4 Math Operators

The following operators can be used in math expressions. The allowed atoms for these expressions are numbers and identifiers. Identifiers can refer to either math variables, function calls, or input expressions.

If both sides of a binary math operator are numbers, the operation will be automatically evaluated. Otherwise, a math expression will be implicitly created.

expression '=' expression Defines an equation. Can be useful for implementing routines for solving simultaneous equations.

expression '^' expression Exponentiation.

expression '*' expression Multiplication.

expression '/' expression Division.

expression '+' expression Addition.

expression '−' expression Subtraction.

3.3.5 List Operators

The following operators are used for lists and strings.

expression '++' expression Represents concatenation. You can concatenate strings or lists. The following are a few examples:

```
"Hello " ++ "world"
[1,2,3] ++ [4,5,6]
```

expression ':' expression A colon represents a pattern match for lists. This operator can be used **only** in the arguments of a function declaration. Take the following example:

```
foo (x:xs) = foo(xs) ++ [x]
```

Here, x refers to the first item of a list, and xs refers to the rest of the items. This technique will work for both strings and lists.

3.3.6 Miscellaneous Expressions

let-in The let-in expression allows you to define a set of functions that are local to an expression. Take the following example:

```
foo =
    let
        x = 3
        y = 5
    in
        x + y
```

Here, x and y are functions local to the foo function.

3.4 Functions

A function provides a reusable parameterized expression. As shown in a previous section, the following is the grammar for a function:

```
function:
    identifier '(' input-expression-list ')' '=' expression

input-expression-list:
    expression
    expression ',' input-expression-list
```

The input expression-list declares zero or more input expressions that can be used in the output expression.

You can declare several functions with the same name, but with different input expression-lists. The version of the function chosen at runtime depends on the input of the caller. This is defined as *pattern matching* and works as follows:

- If you specify a literal as a function argument, that literal must be matched exactly. So if you declare a function with a single argument of "hello", that function will be chosen only when a caller specifies the string value "hello" in the function call. This works for strings, characters, numbers, and lists.
- If a function argument contains a list pattern match (using the colon ':' operator), the variables in the pattern match will refer to elements of the list. For example:

  ```
  foo (x:xs) = foo(xs) ++ [x]
  ```

  Here, x refers to the first item of a list, and xs refers to the rest of the items. This technique will work for both strings and lists.

  You can specify multiple items in a list pattern match. So if you declare an argument as 'x:y:z', x will refer to the first item in the list, y will refer to the second item in the list, and z will refer to the rest of the list.
- If a function argument contains a math expression, that math expression will be matched. Take the following example:

  ```
  foo (left+right) = bar(left) + bar(right)
  ```

  In this example, left refers to the left side of the addition expression, and right refers to the right side of the addition expression.
- If a function argument contains a single identifier, it refers to the value passed in by the caller, similar to function arguments in imperative languages.

3.4.1 Function Precedence

A function call may match multiple function declarations. In this situation the following rules are used to determine which declaration to use for the call:

- If multiple matching functions are in the same block scope (i.e. at the file level or let-block level), the declaration that appears closest to the beginning (top) of the block will be used.
- If one function appears in a deeper block, that function is used first. So if you define a function "foo" in a let-block, and another function "foo" appears at the file level, the function declaration inside the let-block will be used.

3.4.2 Applicative-Order Argument Evaluation

When a function call is being evaluated, its arguments are evaluated first. This is referred to applicative-order argument evaluation. The arguments are evaluated from left-to-right.

This is a departure from Haskell, which uses normal-order evaluation. In Haskell, arguments are not evaluated until they are needed within the body of the function, and are evaluated only if needed.

3.5 Syntax Summary

```
program:
    function-list

function-list:
    function
    function function-list

function:
    identifier ‘(’expression-list ‘)’ ‘=’ expression

expression-list:
    expression
    expression ‘,’ expression-list

expression:
    expression-atom expression-tail

expression-atom:
    identifier
    identifier ‘(’ expression-list ‘)’
    string-atom
    number-atom
    boolean-atom
    list-atom
    let-in

expression-tail:
    ‘==’ expression
    ‘>’ expression
    ‘<’ expression
    ‘&&’ expression
    ‘||’ expression
    ‘=’ expression
    ‘^’ expression
    ‘*’ expression
    ‘/’ expression
```

```
    ‘+’ expression
    ‘-’ expression
    ‘++’ expression
    ‘:’ expression

let-in:
    ‘let’ function-list ‘in’ expression

string-atom:
    ‘"’ letters ‘"’

number-atom:
    digits ‘.’ digits

boolean-atom:
    ‘True’
    ‘False’

list-atom:
    ‘[’ expression-list ‘]’

character-atom:
    single-quote character single-quote
```

4 Project Plan

For this software project I used an informal software development life cycle (SDLC). I used three phases in the SDLC: requirements gathering, technical design, and implementation.

During the requirements gathering phase I authored the Proposal and Language Reference Manual. These documents specified the functionality that I was intending to include in my interpreter. During the technical design phase I came up with a simple architecture for the software and chose the development tools I would be using. During the implementation phase I coded the interpreter.

At the beginning of this class I came up with some rough dates regarding milestones in the project:

- Sep 25th – proposal complete.
- Oct 18th – LRM and front-end parser complete.
- Nov 14th – first pass of interpreter complete.
- Nov 31st – code complete.

- Dec 16th – final report complete.

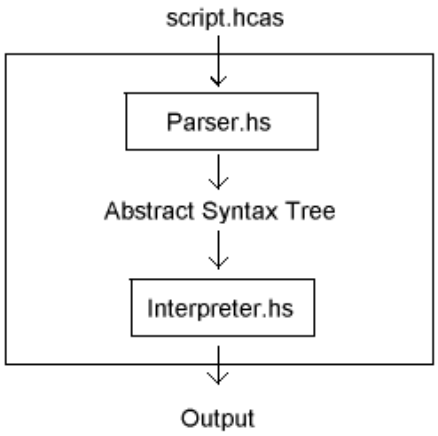
I'm on schedule to complete these milestones, and I'm actually a little bit ahead of schedule.

I used the following tools and technologies during the project:

- Haskell - a functional programming language
- Glasgow Haskell Compiler (6.6.1) - GHC is a popular Haskell compiler and interpreter.
- Parsec - Parsec is a parsing library written in Haskell that allows you to perform top-down parsing.
- vim - for editing source files.
- subversion - for source code control. All source files were under source code control, including document deliverables (proposal, LRM, etc).

5 Architectural Design

The following block diagram shows the major components of my interpreter:



In the above diagram, the process is kicked off by passing the contents of an HCAS script to the interpreter. Here, the HCAS script is named "script.hcas".

The contents of the HCAS script are loaded as a string and then passed to the first major module in the project, "Parser.hs". This Haskell module uses the Parsec parsing library to

parse the string. While parsing the string, this module produces an abstract syntax tree (AST). The AST is represented as a set of Haskell types and is the final output of this module.

The AST is then passed to the "Interpreter.hs" module, which interprets the abstract syntax tree and returns an evaluated AST as output, which is immediately translated into a string and printed to standard output.

6 Test Plan

6.1 Example Scripts

6.1.1 Basics

The following table lists example scripts with their output.

HCAS	output
main = 7	7
main = 7 + 8	15
main = 5 * 6	30
main = 5 * 8 + 4	44
main = x	x
main = x + y	x + y
main = x + y y = 7	x + 7
main = x + y x = 8 y = 7	15
main = rev(”Hello”) rev (x:xs) = rev (xs) ++ [x] rev ([]) = []	”olleH”
main = terms(a+b-c+d*e) terms (l+r) = terms(l) ++ terms(r) terms (l-r) = terms(l) ++ n(terms(r)) terms(x)=[x] n(x:[])=[-x] n(x:xs)=n(xs)++[-x]	[a, b, -c, d*e]

6.1.2 Calculus Derivatives

The following script shows how derivatives can be calculated. It currently returns a list of boolean values, [True, True]. This means that the derivative function is working as expected.

```
main = [derivative(3*x^2+2*x) == 6*x+2,
    simplify(derivative(7*x^3 + 4*x^2 + 6*x + 7)) == 21*x^2 + 8*x + 6]

derivative(a+b) = derivative(a) + derivative(b)
derivative(a-b) = derivative(a) - derivative(b)
derivative(c*x^e) = c*e*simplify(x^(e-1))
derivative(c*x) = c
derivative(x) = 0

simplify(x^1) = x
simplify(x^0) = 1
simplify(x+0) = x
simplify(0+x) = x
simplify(x+y) = simplify(x) + simplify(y)
simplify(x-y) = simplify(x) - simplify(y)
simplify(x) = x
```

6.1.3 Simplification

The following script shows how to write a simplification function.

```
main = [simplify(3*x*2) == 6*x,
        simplify(6*2*x*y*2) == 24*x*y,
        simplify(x*8*2) == 16*x]


{
    Simplifies an expression, multiplying all constants.
}
simplify(a*b) =
    let
        t = terms(a*b)
        nt = numTerms(t)
        pnt = product(nt)
        vt = varTerms(t)
    in
        product([pnt] ++ vt)
simplify(x) = x
```

```
{
    Returns a list of all multiplication terms for the expression.
}
terms(a*b) = terms(a) ++ terms(b)
terms(a) = [a]


{
    Returns the numeric terms for the specified list.
}
numTerms(x:xs) = numTerms(x, isNumber(x)) ++ numTerms(xs)
numTerms(x, True) = [x]
numTerms(x, False) = []
numTerms([]) = []



{
    Returns the variables in the specified term list.
}
varTerms(x:xs) = varTerms(x, isNumber(x)) ++ varTerms(xs)
varTerms(x, True) = []
varTerms(x, False) = [x]
varTerms([]) = []


{
    Product of list elements.
}
product(x:y:xs) = product([x*y] ++ xs)
product(x:[]) = x
product(x:y) = x * y


{
    Returns true if the specified arg is a number.
    Here we're relying on HCAS' functionality of implicitly
    multiplying numeric constants.
}
isNumber(x) = checkNum(x+1)
checkNum(x+1) = False { Not a number, + was preserved. }
checkNum(x) = True { It's a number. }
```

6.2 Unit Tests

I used unit tests for testing the interpreter. My unit tests were broken into three major groups:

- Parser Tests – the first set of tests were centered around the parsing module. I created a set of simple HCAS test scripts, and then for each script I wrote a Haskell unit test that parsed the contents of the script and then tested the resulting AST to make sure it was parsed correctly.
- Interpreter Tests – the second set of tests centered around the interpreter module. These tests were written in HCAS itself, and not Haskell. Each unit test consisted of an HCAS script that executed some code and returned a result. Every HCAS script in this group needed to return either a boolean True value or a list of boolean True values. If any of the scripts returned something else (False, math expressions, etc), the test would fail.
- Commandline Tests – the third set of tests focused on the commandline utility. Each test consisted of an HCAS script and a text file that contained the expected output. Each script would be executed, and the output of the script would be tested against the expected output.

The full test suite contained 144 unit tests. I created a Haskell module, "AllTests.hs", which contained all of the unit testing code.

7 Lessons Learned

Fortunately I started coding my interpreter early on, which gave me a head start and allowed me to complete my interpreter a month early. Starting as early as possible is my best advice to future teams.

If I were to do this project over again, I would probably try to create a compiler that emitted assembly code. I created an interpreter, so I missed out on all of the fun stuff related to Intermediate Representation and Code Generation. I think that the features of my language also would have made this a bit difficult (math expression pattern matching, etc).

Overall, I'm very happy with how my project went.

8 Appendix - Code Listing

8.1 AST.hs

```
module AST where

{-|
```

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        The type declarations.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-}
data Block =            Block [Statement]
data Statement =        Function String [Expression] Expression
data Expression =
                        -- Strings and lists.
                        List [Expression]
                        | Concat Expression Expression
                        | ListPattern [Expression]
                        | CharValue Char

                        -- Function-related items
                        | Call String [Expression]
                        | Let Block Expression

                        -- Math expressions
                        | Variable String
                        | MathVariable String
                        | Number Float
                        | Addition Expression Expression
                        | Multiplication Expression Expression
                        | Subtraction Expression Expression
                        | Division Expression Expression
                        | Parens Expression
                        | Exponent Expression Expression
                        | Equals Expression Expression
                        | Negation Expression

                        -- Logical expressions
                        | BoolEquals Expression Expression
                        | BoolGreaterThan Expression Expression
                        | BoolLessThan Expression Expression
                        | BoolAnd Expression Expression
                        | BoolOr Expression Expression
                        | BoolVal Bool

                        -- Interpreter error
                        | Err String

{-|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        The Eq instances.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-}
instance Eq Block where
        (Block a1) == (Block a2) = (a1 == a2)

instance Eq Statement where
        (Function a1 b1 c1) == (Function a2 b2 c2) = (a1 == a2 && b1 == b2 && c1 == c2)

instance Eq Expression where
        (Concat l1 r1) == (Concat l2 r2) = (l1 == l2 && r1 == r2)
        (Variable s1) == (Variable s2) = (s1 == s2)
        (MathVariable s1) == (MathVariable s2) = (s1 == s2)
        (Call a1 b1) == (Call a2 b2) = (a1 == a2 && b1 == b2)
        (Let a1 b1) == (Let a2 b2) = (a1 == a2 && b1 == b2)
        (List a1) == (List a2) = (a1 == a2)
        (ListPattern a1) == (ListPattern a2) = (a1 == a2)
        (Number a1) == (Number a2) = (a1 == a2)
        (Addition a1 b1) == (Addition a2 b2) = (a1 == a2 && b1 == b2)
        (Multiplication a1 b1) == (Multiplication a2 b2) = (a1 == a2 && b1 == b2)
        (Subtraction a1 b1) == (Subtraction a2 b2) = (a1 == a2 && b1 == b2)
        (Division a1 b1) == (Division a2 b2) = (a1 == a2 && b1 == b2)
```

```
        (Parens a1) == (Parens a2) = (a1 == a2)
        (Exponent a1 b1) == (Exponent a2 b2) = (a1 == a2 && b1 == b2)
        (Equals a1 b1) == (Equals a2 b2) = (a1 == a2 && b1 == b2)
        (BoolEquals a1 b1) == (BoolEquals a2 b2) = (a1 == a2 && b1 == b2)
        (BoolGreaterThan a1 b1) == (BoolGreaterThan a2 b2) = (a1 == a2 && b1 == b2)
        (BoolLessThan a1 b1) == (BoolLessThan a2 b2) = (a1 == a2 && b1 == b2)
        (BoolAnd a1 b1) == (BoolAnd a2 b2) = (a1 == a2 && b1 == b2)
        (BoolOr a1 b1) == (BoolOr a2 b2) = (a1 == a2 && b1 == b2)
        (Negation a1) == (Negation a2) = (a1 == a2)
        (CharValue a1) == (CharValue a2) = (a1 == a2)
        (BoolVal a1) == (BoolVal a2) = (a1 == a2)

{-|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        The Show instances.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-}
instance Show Statement where
        show st = (showHaskell st)

instance Show Block where
        show b = (showHaskell b)

instance Show Expression where
        show expr = (showHaskell expr)

{-|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        The ShowHaskell class and instances.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-}
class ShowHaskell a where
        showHaskell :: a -> String

instance ShowHaskell Expression where
        showHaskell (Concat l r) = "(Concat " ++ (showHaskell l) ++ " " ++ (showHaskell r) ++ ")"
        showHaskell (Number a) = "(Number " ++ (showHaskell a) ++ ")"
        showHaskell (Call a b) = "(Call \"" ++ a ++ "\" [" ++ (showHaskell b) ++ "])"
        showHaskell (Let a b) = "(Let " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (ListPattern a) = "(ListPattern [" ++ (showHaskell a) ++ "])"
        showHaskell (Addition a b) = "(Addition " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (Multiplication a b) = "(Multiplication " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (Variable s) = "(Variable \"" ++ s ++ "\")"
        showHaskell (MathVariable s) = "(MathVariable '" ++ s ++ "')"
        showHaskell (Subtraction a b) = "(Subtraction " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (Division a b) = "(Division " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (Parens a) = "(Parens " ++ (showHaskell a) ++ ")"
        showHaskell (Exponent a b) = "(Exponent " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (Equals a b) = "(Equals " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (BoolEquals a b) = "(BoolEquals " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (BoolGreaterThan a b) = "(BoolGreaterThan " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (BoolLessThan a b) = "(BoolLessThan " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (BoolAnd a b) = "(BoolAnd " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (BoolOr a b) = "(BoolOr " ++ (showHaskell a) ++ " " ++ (showHaskell b) ++ ")"
        showHaskell (BoolVal b) = "(BoolVal " ++ (showHaskell b) ++ ")"
        showHaskell (Err s) = "(Err \"" ++ s ++ "\")"
        showHaskell (Negation e) = "(Negation " ++ (showHaskell e) ++ ")"
        showHaskell (CharValue e) = "(CharValue '" ++ [e] ++ "')"
        showHaskell (List a) = "(List [" ++ (showHaskell a) ++ "])"

instance ShowHaskell Block where
        showHaskell (Block statements) = "(Block [" ++ (showHaskell statements) ++ "])"

instance ShowHaskell Float where
        showHaskell f = (show f)
```

```
instance (ShowHaskell a) => ShowHaskell [a] where
        showHaskell (x:[]) = (showHaskell x)
        showHaskell (x:xs) = (showHaskell x) ++ "," ++ (showHaskell xs)
        showHaskell [] = ""

instance ShowHaskell Bool where
        showHaskell True = "True"
        showHaskell False = "False"

instance ShowHaskell Statement where
        showHaskell (Function name args expr) = "(Function \"" ++ name ++ "\" [" ++ (showHaskell args) ++ "] " ++ (showHaskell e


{-|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        The ShowHCAS class and instances.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-}
class ShowHCAS a where
        showHCAS :: a -> String

instance ShowHCAS Expression where
        showHCAS (Number n) = (numberString n)
        showHCAS (Multiplication a b) = (showHCAS a) ++ "*" ++ (showHCAS b)
        showHCAS (Division a b) = (showHCAS a) ++ "/" ++ (showHCAS b)
        showHCAS (Addition a b) = (showHCAS a) ++ "+" ++ (showHCAS b)
        showHCAS (Subtraction a b) = (showHCAS a) ++ "-" ++ (showHCAS b)
        showHCAS (Exponent a b) = (showHCAS a) ++ "^" ++ (showHCAS b)
        showHCAS (Parens inner) = "(" ++ (showHCAS inner) ++ ")"
        showHCAS (Equals a b) = (showHCAS a) ++ "=" ++ (showHCAS b)
        showHCAS (Negation inner) = "-" ++ (showHCAS inner)
        showHCAS (MathVariable s) = s
        showHCAS (Err s) = "ERROR: " ++ s
        showHCAS (CharValue c) = "'" ++ [c] ++ "'"
        showHCAS (BoolVal True) = "True"
        showHCAS (BoolVal False) = "False"
        showHCAS (List l)       | (length l) > 0 && (isString l) = "\"" ++ (toString l) ++ "\""
                                | otherwise = "[" ++ (showHCAS l) ++ "]"
                where
                        isString ((CharValue c):xs) = isString xs
                        isString [] = True
                        isString _ = False
                        toString ((CharValue c):xs) = [c] ++ (toString xs)
                        toString [] = []
instance (ShowHCAS a) => ShowHCAS [a] where
        showHCAS (x:[]) = (showHCAS x)
        showHCAS (x:xs) = (showHCAS x) ++ "," ++ (showHCAS xs)
        showHCAS [] = ""

instance ShowHCAS Bool where
        showHCAS True = "True"
        showHCAS False = "False"

instance ShowHCAS Statement where
        showHCAS (Function name [] expr) = name ++ " = " ++ (showHCAS expr)
        showHCAS (Function name args expr) = name ++ "(" ++ (showHCAS args) ++ ") = " ++ (showHCAS expr)

instance ShowHCAS Block where
        showHCAS (Block statements) = (showHCASSpaceDelimited statements)

showHCASSpaceDelimited :: (ShowHCAS a) => [a] -> String
showHCASSpaceDelimited (x:[]) = (showHCAS x)
showHCASSpaceDelimited (x:xs) = (showHCAS x) ++ "\n" ++ (showHCAS xs)
showHCASSpaceDelimited [] = ""
```

```
numberString n  | (take 2 (reverse s)) == "0." = (take ((length s) - 2) s)
                | otherwise = s
        where
                s = (show n)
```

8.2 Parser.hs

```
module Parser where

import AST
import Text.ParserCombinators.Parsec


{-|
        Parses an entire file.
-}
file :: Parser Block
file =
                do {
                        whitespace;
                        ss <- (sepBy1 (function) whitespace);
                        return (Block ss);
                }

{-|
        Eats zero or more occurences of whitespace.
-}
whitespace :: Parser ()
whitespace =
                do {
                        many singleWhitespace;
                        return ();
                }
singleWhitespace :: Parser ()
singleWhitespace =
                do {
                        tab <|> space <|> newline <|> hcasComment;
                        return ();
                }
atLeastOneWhitespace :: Parser ()
atLeastOneWhitespace =
                do {
                        many1 singleWhitespace;
                        return ();
                }
hcasComment =
                do {
                        try(string "{");
                        manyTill anyChar (try (string "}"));
                        return ('c');
                }

{-|
        Parses a function definition.
-}
function :: Parser Statement
function =
                do {
                        ident <- identifier;
                        whitespace;
                        do {
                                args <- callParens;
                                whitespace;
                                (functionExpression ident args);
                        }
                        <|>
                        (functionExpression ident []);
```

```
                }


functionExpression :: String -> [Expression] -> (Parser Statement)
functionExpression ident args =
                do {
                        char '=';
                        whitespace;
                        rv <- expression;
                        (buildFunction ident (isReserved ident) args rv);
                }


buildFunction :: String -> Bool -> [Expression] -> Expression -> (Parser Statement)
buildFunction ident True args rv =
                do { (fail ("The string '" ++ ident ++ "' is a reserved word and cannot be used for a function's name.")); }
buildFunction ident False args rv =
                do {
                        return (Function ident args rv);
                }



{-|
        Returns true if the string is a reserved identifier.
-}
isReserved "let" = True
isReserved "in" = True
isReserved "True" = True
isReserved "False" = True
isReserved _ = False



{-|
        Parses an expression.
-}
expression = exprBoolAnd

exprBoolAnd :: (Parser Expression)
exprBoolAnd = do { left <- exprBool; whitespace; (exprBoolAnd' left); }
exprBoolAnd' :: Expression -> (Parser Expression)
exprBoolAnd' left = do { string "&&"; whitespace; right <- exprBool; (exprBoolAnd' (BoolAnd left right)); }
                <|> do { string "||"; whitespace; right <- exprBool; (exprBoolAnd' (BoolOr left right)); }
                <|> return left;

exprBool :: (Parser Expression)
exprBool = do { left <- exprFunc; whitespace; (exprBool' left); }
exprBool' :: Expression -> (Parser Expression)
exprBool' left = do { try(string "=="); whitespace; right <- exprFunc; (exprBool' (BoolEquals left right)); }
                <|> do { string ">"; whitespace; right <- exprFunc; (exprBool' (BoolGreaterThan left right)); }
                <|> do { string "<"; whitespace; right <- exprFunc; (exprBool' (BoolLessThan left right)); }
                <|> return left;


exprFunc :: (Parser Expression)
exprFunc = do { left <- exprAdd; whitespace; (exprFunc' left); }
exprFunc' :: Expression -> (Parser Expression)
exprFunc' left = do { try(string "=="); whitespace; right <- exprFunc; (exprBool' (BoolEquals left right)); }
                <|> do { string "="; whitespace; right <- exprAdd; (exprFunc' (Equals left right)); }
                <|> return left;

exprAdd :: (Parser Expression)
exprAdd = do { left <- exprMult; whitespace; (exprAdd' left); }
exprAdd' :: Expression -> (Parser Expression)
```

```
exprAdd' left = do { try(string "++"); whitespace; right <- exprMult; (exprAdd' (Concat left right)); }
                <|> do { string "+"; whitespace; right <- exprMult; (exprAdd' (Addition left right)); }
                <|> do { string "-"; whitespace; right <- exprMult; (exprAdd' (Subtraction left right)); }
                <|> return left;

exprMult :: (Parser Expression)
exprMult = do { left <- exprNegate; whitespace; (exprMult' left); }
exprMult' :: Expression -> (Parser Expression)
exprMult' left = do { string "*"; whitespace; right <- exprNegate; (exprMult' (Multiplication left right)); }
                <|> do { string "/"; whitespace; right <- exprNegate; (exprMult' (Division left right)); }
                <|> return left;

exprNegate :: (Parser Expression)
exprNegate = do { string "-"; whitespace; right <- exprExp; return (Negation right); }
                <|> (exprExp);

exprExp :: (Parser Expression)
exprExp = do { left <- exprListPattern; whitespace; (exprExp' left); }
exprExp' :: Expression -> (Parser Expression)
exprExp' (Exponent left right) = do { string "^"; whitespace; right' <- exprListPattern; (exprExp' (Exponent left (Exponent righ
                <|> return (Exponent left right);
exprExp' left = do { string "^"; whitespace; right <- exprListPattern; (exprExp' (Exponent left right)); }
                <|> return left;


exprListPattern :: (Parser Expression)
exprListPattern = do { left <- exprAtom; whitespace; (exprListPattern' [left]); }
exprListPattern' :: [Expression] -> (Parser Expression)
exprListPattern' (left:[]) = do { string ":"; whitespace; right <- exprAtom; (exprListPattern' ([left] ++ [right])); }
                <|> return (left);
exprListPattern' (x:xs) = do { string ":"; whitespace; right <- exprAtom; (exprListPattern' ([x] ++ xs ++ [right])); }
                <|> return (ListPattern ([x] ++ xs));


exprAtom :: (Parser Expression)
exprAtom =
                do { letAtom; }
                <|> do { boolAtom; }
                <|> do { identifierAtom; }
                <|> do { stringAtom; }
                <|> do { listAtom; }
                <|> do { numberAtom;}
                <|> do { parensAtom; }
                <|> do { charAtom; }


{-|
        Parses a string value.
-}
stringAtom :: Parser Expression
stringAtom =
                do {
                        char '"';
                        s <- (manyTill anyChar (char '"'));
                        return (makeStr s);
                }


makeStr :: String -> Expression
makeStr s =
                (List (strExpr s))
        where
                strExpr (x:xs) = [(CharValue x)] ++ (strExpr xs)
                strExpr [] = []
```

```
charAtom :: (Parser Expression)
charAtom =
                do {
                        char '\'';
                        c <- anyChar;
                        if c == '\'' then
                                do { fail "A character literal must contain a single non-quote character.";}
                                else
                                        do {
                                                char '\'';
                                                return (CharValue c);
                                        }
                }



{-|
        Parses an identifier, which represents either a variable or
        a function's name.
-}
identifier :: Parser String
identifier =
                do {
                        c <- letter;
                        cs <- many (identifierChar);
                        return (c:cs);
                }

identifierChar =
                do {
                        (alphaNum <|> char '_');
                }


{-|
        Parses the arguments of a function call.
-}
callParens :: Parser [Expression]
callParens =
                do {
                        char '(';
                        whitespace;
                        rvs <- sepBy expression listSeparator;
                        char ')';
                        return rvs;
                }


{-|
        Parses a list.
-}
listAtom :: Parser Expression
listAtom =
                do {
                        char '[';
                        whitespace;
                        rvs <- sepBy expression listSeparator;
                        char ']';
                        return (List rvs)
                }

listSeparator :: Parser ()
listSeparator =
                do {
```

```
                        whitespace;
                        (char ',');
                        whitespace;
                }



{-|
        Parses an identifier atom (either an identifier or a function call).
-}
identifierAtom :: (Parser Expression)
identifierAtom =
                do {
                        ident <- identifier;
                        whitespace;
                        do {
                                p <- callParens;
                                (buildCall ident (isReserved ident) p);
                        }

                        <|>
                        (buildVariable ident (isReserved ident));
                }



buildCall :: String -> Bool -> [Expression] -> (Parser Expression)
buildCall ident True p = do { (fail ("The string '" ++ ident ++ "' is a reserved word and cannot be used for a function call nam
buildCall ident False p =
                do {
                        return (Call ident p);
                }


buildVariable :: String -> Bool -> (Parser Expression)
buildVariable ident True = do { (fail ("The string '" ++ ident ++ "' is a reserved word and cannot be used for a variable name."
buildVariable ident False =
                do {
                        return (Variable ident);
                }


boolAtom :: (Parser Expression)
boolAtom =
                do {
                        try(trueBegin);
                        return (BoolVal True);
                }
                <|>
                do {
                        try(falseBegin);
                        return (BoolVal False);
                }

trueBegin = do { string "True"; notFollowedBy identifierChar; }
falseBegin = do { string "False"; notFollowedBy identifierChar; }


{-|
        Parses a let expression.
-}
letAtom :: (Parser Expression)
letAtom =
                do {
                        try(letAtomBegin);
```

```
                        whitespace;
                        firstFunc <- letAtomParse;
                        whitespace;
                        otherFuncs <- (manyTill letAtomParse letAtomEnd);
                        whitespace;
                        rv <- expression;
                        whitespace;
                        return (Let (Block ([firstFunc] ++ otherFuncs)) rv);
                }

letAtomBegin =
                do {
                        string "let";
                        (notFollowedBy identifierChar);
                }

letAtomParse :: (Parser Statement)
letAtomParse =
                do {
                        fd <- function;
                        whitespace;
                        return fd;
                }

letAtomEnd =
                do {
                        string "in";
                        notFollowedBy identifierChar;
                }


{-|
        Parses parens for a math expression.
-}
parensAtom :: Parser Expression
parensAtom =
                do {
                        char '(';
                        whitespace;
                        rv <- expression;
                        char ')';
                        return (Parens rv);
                }


{-|
        Parses a number literal. Can be either an integer
        or float.
-}
numberAtom :: Parser Expression
numberAtom =    do {    do {    before <- (many1 digit);
                                do { (numberAfter (read before)); }
                                <|> return (Number (fromIntegral (read before)));
                        }
                        <|> do { (numberAfter 0); }
                }

numberAfter :: Int -> (Parser Expression)
numberAfter before =
                do {    char '.';
                        after <- (many1 digit);
                        return (Number ((fromIntegral before) + ((read after) / (10.0 ^ (length after)))));
                }
```

8.3 Interpreter.hs

```
module Interpreter where

import AST

{-|
        Interprets a file. Executes the main function and returns
        the result.
-}
interpretFile :: Block -> Expression
interpretFile (Block statements) =
        case (mainFunction statements) of
                (Left err) -> err
                (Right (Function _ _ mainFunc)) -> interpret [(Block statements)] mainFunc

mainFunction (x:xs) =
        if (isMain x) then
                (Right x)
                else
                        (mainFunction xs)

mainFunction [] =
        (Left (Err "Could not find main function in file."))

nonMainFunctions (x:xs) =
        if (isMain x) then
                (nonMainFunctions xs)
                else
                        [x] ++ (nonMainFunctions xs)
nonMainFunctions [] = []

isMain (Function "main" _ _) = True
isMain (Function _ _ _) = False


{-|
        Interprets an expression, given the block list
        as its current "context". Each block represents a set
        of statements available to the interpreter at a point
        in time.

        The block list is a stack that grows at the beginning, so
        the newest entry exists at the first position.
-}

interpret :: [Block] -> Expression -> Expression

interpret _ (BoolVal b) =
        (BoolVal b)

interpret _ (Number n) =
        (Number n)

interpret _ (CharValue c) =
        (CharValue c)

interpret blocks (Let block expr) =
        (interpret ([block] ++ blocks) expr)

interpret blocks (List inner) =
                (List (interpretList inner))
```

```
        where
                interpretList (x:xs) = [(interpret blocks x)] ++ (interpretList xs)
                interpretList [] = []

interpret blocks (Parens inner) =
                (interpretParens inner')
        where
                inner' = (interpret blocks inner)
                interpretParens (Err s) = (Err s)
                interpretParens (List s) = (List s)
                interpretParens (Number n) = (Number n)
                interpretParens (BoolVal b) = (BoolVal b)
                interpretParens inner'' = (Parens inner'')

interpret blocks (BoolEquals left right) =
                (boolEquals left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                boolEquals (Err s) _ = (Err s)
                boolEquals _ (Err s) = (Err s)
                boolEquals (Multiplication a1 b1) (Multiplication a2 b2) =
                        (BoolVal (((boolEquals a1 a2) == (BoolVal True)) && ((boolEquals b1 b2) == (BoolVal True))))
                boolEquals (Addition a1 b1) (Addition a2 b2) =
                        (BoolVal (((boolEquals a1 a2) == (BoolVal True)) && ((boolEquals b1 b2) == (BoolVal True))))
                boolEquals (Subtraction a1 b1) (Subtraction a2 b2) =
                        (BoolVal (((boolEquals a1 a2) == (BoolVal True)) && ((boolEquals b1 b2) == (BoolVal True))))
                boolEquals (Division a1 b1) (Division a2 b2) =
                        (BoolVal (((boolEquals a1 a2) == (BoolVal True)) && ((boolEquals b1 b2) == (BoolVal True))))
                boolEquals (Exponent a1 b1) (Exponent a2 b2) =
                        (BoolVal (((boolEquals a1 a2) == (BoolVal True)) && ((boolEquals b1 b2) == (BoolVal True))))
                boolEquals (Parens a1) (Parens a2) =
                        (BoolVal (((boolEquals a1 a2) == (BoolVal True))))
                boolEquals (Negation a1) (Negation a2) =
                        (BoolVal (((boolEquals a1 a2) == (BoolVal True))))
                boolEquals (Number n1) (Number n2) = (BoolVal (n1 == n2))
                boolEquals (List l1) (List l2) = (BoolVal (l1 == l2))
                boolEquals (BoolVal b1) (BoolVal b2) = (BoolVal (b1 == b2))
                boolEquals (MathVariable v1) (MathVariable v2) = (BoolVal (v1 == v2))

                boolEquals left'' right'' = (Err ("The left value '" ++ (showHCAS left'') ++ "' and the right value '" ++ (showH


interpret blocks (BoolGreaterThan left right) =
                (boolGreaterThan left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                boolGreaterThan (Number n1) (Number n2) = (BoolVal (n1 > n2))
                boolGreaterThan left'' right'' = (Err ("The left value '" ++ (showHCAS left'') ++ "' and the right value '" ++ (

interpret blocks (BoolLessThan left right) =
                (boolLessThan left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                boolLessThan (Number n1) (Number n2) = (BoolVal (n1 < n2))
                boolLessThan left'' right'' = (Err ("The left value '" ++ (showHCAS left'') ++ "' and the right value '" ++ (sho

interpret blocks (BoolAnd left right) =
                (boolAnd left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                boolAnd (BoolVal b1) (BoolVal b2) = (BoolVal (b1 && b2))
```

```
                boolAnd left'' right'' = (Err ("The left value '" ++ (showHCAS left'') ++ "' and the right value '" ++ (showHCAS

interpret blocks (BoolOr left right) =
                (boolOr left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                boolOr (BoolVal b1) (BoolVal b2) = (BoolVal (b1 || b2))
                boolOr left'' right'' = (Err ("The left value '" ++ (showHCAS left'') ++ "' and the right value '" ++ (showHCAS

interpret blocks (Addition left right) =
                (addition left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                addition (Number n1) (Number n2) = (Number (n1 + n2))
                addition left'' right'' = (Addition left'' right'')

interpret blocks (Subtraction left right) =
                (subtraction left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                subtraction (Number n1) (Number n2) = (Number (n1 - n2))
                subtraction left'' right'' = (Subtraction left'' right'')

interpret blocks (Multiplication left right) =
                (multiplication left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                multiplication (Number n1) (Number n2) = (Number (n1 * n2))
                multiplication left'' right'' = (Multiplication left'' right'')

interpret blocks (Division left right) =
                (division left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                division (Number n1) (Number 0) = (Err "You cannot divide by zero.")
                division (Number n1) (Number n2) = (Number (n1 / n2))
                division left'' right'' = (Division left'' right'')

interpret blocks (Exponent left right) =
                (exponent' left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                exponent' (Number n1) (Number n2) = (Number (n1 ** n2))
                exponent' left'' right'' = (Exponent left'' right'')

interpret blocks (Concat left right) =
                (cat left' right')
        where
                left' = (interpret blocks left)
                right' = (interpret blocks right)
                cat (Err s) _ = (Err s)
                cat _ (Err s) = (Err s)
                cat (List s1) (List s2) = (List (s1 ++ s2))
                cat left'' right'' = (Err ("The left value '" ++ (showHCAS left'') ++ "' and the right value '" ++ (showHCAS rig

interpret blocks (Equals left right) =
                (Equals left' right')
        where
                left' = (interpret blocks left)
```

```
                right' = (interpret blocks right)

interpret blocks (Negation inner) =
                (negation inner')
        where
                inner' = (interpret blocks inner)
                negation (Negation inner'') = inner''
                negation (Number n) = (Number (0-n))
                negation inner'' = (Negation inner'')

interpret blocks (Variable name) =
        case (findFunction blocks name []) of
                (Just (Function _ _ expression)) -> (interpret (findBlocks blocks name []) expression)
                (Nothing) -> (MathVariable name)

interpret _ (MathVariable name) = (MathVariable name)

interpret blocks (Call callName callArgs) =
                (interpretFunction (findFunction blocks callName callArgs'))
        where
                interpretFunction (Just (Function _ functionArgs functionExpression)) =
                                (interpret blocks' functionExpression)
                        where
                                blocks' = [(Block (createStatements callArgs' functionArgs))] ++ (findBlocks blocks callName cal

                interpretFunction (Nothing) =
                        (Err ("Cannot find function with name '" ++ callName ++ "' and args (" ++ (showHCAS callArgs') ++ ").")

                callArgs' = (interpretArgs callArgs)

                interpretArgs (x:xs) = [(interpret blocks x)] ++ (interpretArgs xs)
                interpretArgs [] = []

{-|
        Finds the function declaration for the given call, looking
        through the specified block list.
-}
findFunction :: [Block] -> String -> [Expression] -> Maybe Statement
findFunction (block:blocks) callName callArgs =
        case (findFunctionInBlock block callName callArgs) of
                (Nothing) -> (findFunction blocks callName callArgs)
                (Just f) -> (Just f)
findFunction [] _ _ = Nothing


{-|
        Similar to findFunction above, but just for a single block.
-}
findFunctionInBlock :: Block -> String -> [Expression] -> Maybe Statement
findFunctionInBlock (Block ((Function functionName functionArgs functionExpression):statements)) callName callArgs =
        case (argumentsMatch callArgs functionArgs) && (functionName == callName) of
                True -> (Just (Function functionName functionArgs functionExpression))
                False -> (findFunctionInBlock (Block statements) callName callArgs)
findFunctionInBlock (Block []) _ _ = Nothing


findBlocks :: [Block] -> String -> [Expression] -> [Block]
findBlocks (block:blocks) callName callArgs =
        case (findFunctionInBlock block callName callArgs) of
                (Nothing) -> (findBlocks blocks callName callArgs)
                (Just f) -> ([block] ++ blocks)
findBlocks [] _ _ = []


{-|
```

```
        Returns true if the specified call arguments match
        the function arguments.
-}
argumentsMatch (callArg:callArgs) (functionArg:functionArgs)
        | (argumentMatches callArg functionArg) == False = False
        | (length callArgs) /= (length functionArgs) = False
        | otherwise = (argumentsMatch callArgs functionArgs)
argumentsMatch [] [] = True
argumentsMatch _ _ = False


{-|
        Returns true if the given call argument matches
        the function argument.
-}
argumentMatches _ (Variable n) = True
argumentMatches (Addition a1 b1) (Addition a2 b2) = (argumentMatches a1 a2) && (argumentMatches b1 b2)
argumentMatches (Subtraction a1 b1) (Subtraction a2 b2) = (argumentMatches a1 a2) && (argumentMatches b1 b2)
argumentMatches (Multiplication a1 b1) (Multiplication a2 b2) = (argumentMatches a1 a2) && (argumentMatches b1 b2)
argumentMatches (Division a1 b1) (Division a2 b2) = (argumentMatches a1 a2) && (argumentMatches b1 b2)
argumentMatches (Exponent a1 b1) (Exponent a2 b2) = (argumentMatches a1 a2) && (argumentMatches b1 b2)
argumentMatches (Parens a1) (Parens a2) = (argumentMatches a1 a2)
argumentMatches (Negation a1) (Negation a2) = (argumentMatches a1 a2)
argumentMatches (Equals a1 b1) (Equals a2 b2) = (argumentMatches a1 a2) && (argumentMatches b1 b2)
argumentMatches (Number n1) (Negation n2) = n1 < 0
argumentMatches (CharValue c1) (CharValue c2) = c1 == c2
argumentMatches (Number n1) (Number n2) = n1 == n2
argumentMatches (List s1) (List s2) = s1 == s2
argumentMatches (List l) (ListPattern ((List p):ps))
                | (length l) >= (length p) = (argumentMatches (List lhead) (List p)) && (argumentMatches (List ltail) (ListPatte
                | otherwise = False
        where
                lhead = (take (length p) l)
                ltail = (drop (length p) l)
argumentMatches (List (l:[])) (ListPattern (p:ps:[])) =
        (argumentMatches l p)
argumentMatches (List (l:ls)) (ListPattern (p:[])) =
        (argumentMatches (List ([l]++ls)) p)
argumentMatches (List (l:ls)) (ListPattern (p:ps)) =
        (argumentMatches l p) && (argumentMatches (List ls) (ListPattern ps))
argumentMatches _ _ = False


{-|
        Creates a set of statements for the call argument and
        function argument. The basic idea here is that new function
        declarations are added to the "statements" context for all
        arguments in the current call. These declarations are available
        for the duration of the call, and then are popped off the
        stack (implicitly, when the call unwinds).
-}
createStatement callArg (Variable functionArg) =
        [(Function functionArg [] callArg)]
createStatement (Addition a1 b1) (Addition a2 b2) =
        (createStatement a1 a2) ++ (createStatement b1 b2)
createStatement (Subtraction a1 b1) (Subtraction a2 b2) =
        (createStatement a1 a2) ++ (createStatement b1 b2)
createStatement (Multiplication a1 b1) (Multiplication a2 b2) =
        (createStatement a1 a2) ++ (createStatement b1 b2)
createStatement (Division a1 b1) (Division a2 b2) =
        (createStatement a1 a2) ++ (createStatement b1 b2)
createStatement (Exponent a1 b1) (Exponent a2 b2) =
        (createStatement a1 a2) ++ (createStatement b1 b2)
createStatement (Parens a1) (Parens a2) =
        (createStatement a1 a2)
```

```
createStatement (Negation a1) (Negation a2) =
        (createStatement a1 a2)
createStatement (Equals a1 b1) (Equals a2 b2) =
        (createStatement a1 a2) ++ (createStatement b1 b2)
createStatement (Number n1) (Negation (Number n2)) =
        []
createStatement (Number n1) (Negation (Variable v1)) =
        (createStatement (Number (0-n1)) (Variable v1))
createStatement (Number n1) (Number n2) =
        []
createStatement (List (a1:as)) (List (b1:bs)) =
        (createStatement a1 b1) ++ (createStatement (List as) (List bs))
createStatement (List l) (ListPattern ((List p):ps)) =
                (createStatement (List ltail) (ListPattern ps))
        where
                ltail = (drop (length p) l)
createStatement (List (l:[])) (ListPattern (p:(Variable v):[])) =
        (createStatement l p) ++ [(Function v [] (List []))]
createStatement (List (l:[])) (ListPattern (p:(List []):[])) =
        (createStatement l p)
createStatement (List (l:ls)) (ListPattern (p:[])) =
        (createStatement (List ([l]++ls)) p)
createStatement (List (l:ls)) (ListPattern (p:ps)) =
        (createStatement l p) ++ (createStatement (List ls) (ListPattern ps))



{-|
        Creates a set of statements for the given call and function arguments.
-}
createStatements (callArg:callArgs) (functionArg:functionArgs) =
        (createStatement callArg functionArg) ++ (createStatements callArgs functionArgs)
createStatements [] [] = []
```

8.4 MainInterpreter.hs

```
module Main where

import AST
import Parser
import Interpreter
import Text.ParserCombinators.Parsec

main =
        do {
                script <- getContents;
                case (parse file "" script) of
                        (Right parsed) ->
                                do {
                                        interpreted <- return (interpretFile parsed);
                                        putStrLn (showHCAS interpreted);
                                }
                        (Left err) ->
                                do {
                                        putStrLn (show err);
                                }
        }
```

8.5 AllTests.hs

```
import Test.HUnit
import AST
import Parser
import Text.ParserCombinators.Parsec
import HW1
import System.Directory
import Interpreter
import System


{-|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        Main entry point. Calls all of the different tests.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-}

main =
        do {
                interpreterFiles <- (getDirectoryContents "testinput/interpreter/");
                commandlineFiles <- (getDirectoryContents "testinput/commandline/");
                runTestTT (TestList(allTests ++ (makeInterpreterTests interpreterFiles) ++ (makeCommandlineTests commandlineFile
        }




{-|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        Commandline tests. Each script has a corresponding "expected"
        file which contains the expected output. This expected output
        should be matched exactly.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-}

makeCommandlineTests (".":xs) = makeCommandlineTests xs
makeCommandlineTests ("..":xs) = makeCommandlineTests xs
makeCommandlineTests (".svn":xs) = makeCommandlineTests xs
makeCommandlineTests (x:xs) = (makeCommandlineTest (take 5 (reverse x)) (take ((length x) - 5) x)) ++ (makeCommandlineTests xs)
makeCommandlineTests [] = []

makeCommandlineTest "sach." x = [(hcasiCommandlineTest x)] --, (hcashCommandlineTest x)]
makeCommandlineTest _ _ = []


hcasiCommandlineTest fileName = TestCase (
                        do {
                                script <- readFile ("testinput/commandline/" ++ fileName ++ ".hcas");
                                (system ("bin/hcasi/hcasi < testinput/commandline/" ++ fileName ++ ".hcas > bin/testoutput/comma
                                expected <- readFile("testinput/commandline/" ++ fileName ++ "_expected.txt");
                                actual <- readFile("bin/testoutput/commandline/" ++ fileName ++ "_actual_hcasi.txt");
                                assertEqual fileName expected actual;
                        }
                )


hcashCommandlineTest fileName = TestCase (
                        do {
                                (system "cp Interpreter.hs bin/testoutput/commandline/");
                                (system "cp AST.hs bin/testoutput/commandline/");
                                (system ("bin/hcash/hcash < testinput/commandline/" ++ fileName ++ ".hcas > bin/testoutput/comma
                                (system ("ghc --make bin/testoutput/commandline/" ++ fileName ++ ".hs"));
```

```
                                (system ("bin/testoutput/commandline/" ++ fileName ++ " > bin/testoutput/commandline/" ++ fileNa
                                expected <- readFile("testinput/commandline/" ++ fileName ++ "_expected.txt");
                                actual <- readFile("bin/testoutput/commandline/" ++ fileName ++ "_actual_hcash.txt");
                                assertEqual fileName expected actual;
                        }
                )



{-|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        Interpreter tests. All scripts must return a boolean value of True.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-}


makeInterpreterTests (".":xs) = makeInterpreterTests xs
makeInterpreterTests ("..":xs) = makeInterpreterTests xs
makeInterpreterTests (".svn":xs) = makeInterpreterTests xs
makeInterpreterTests (x:xs) = (makeInterpreterTest (take 5 (reverse x)) (take ((length x) - 5) x)) ++ (makeInterpreterTests xs)
makeInterpreterTests [] = []

makeInterpreterTest "sach." x = [(hcasiInterpreterTest x)] --, (hcashInterpreterTest x)]
makeInterpreterTest _ _ = []



hcasiInterpreterTest fileName = TestCase (
                        do {
                                s <- readFile ("testinput/interpreter/" ++ fileName ++ ".hcas");

                                case (parse file "" s) of
                                        (Right parsed) ->
                                                do {
                                                        res <- return (interpretFile parsed);
                                                        case (passed res) of
                                                                True -> do { assertEqual fileName True True; }
                                                                False -> do { putStrLn ""; putStrLn "Error!"; putStrLn ("filenam
                                                }
                                        (Left err) ->
                                                do {
                                                        putStrLn "";
                                                        putStrLn (fileName ++ ": " ++ (show err));
                                                        putStrLn "";
                                                }
                        }
                )


hcashInterpreterTest fileName = TestCase (
                        do {
                                (system "cp Interpreter.hs bin/testoutput/interpreter/");
                                (system "cp AST.hs bin/testoutput/interpreter/");
                                (system ("bin/hcash/hcash < testinput/interpreter/" ++ fileName ++ ".hcas > bin/testoutput/inter
                                (system ("ghc --make bin/testoutput/interpreter/" ++ fileName ++ ".hs"));
                                (system ("bin/testoutput/interpreter/" ++ fileName ++ " > bin/testoutput/interpreter/" ++ fileNa
                                actual <- readFile("bin/testoutput/interpreter/" ++ fileName ++ "_actual_hcash.txt");
                                case (parse exprAtom "" actual) of
                                        (Right res) ->
                                                do {
                                                        case (passed res) of
                                                                True -> do { assertEqual fileName True True; }
                                                                False -> do { putStrLn ""; putStrLn "Hcash Error!"; putStrLn ("f
                                                }
```

```
                                        (Left err) ->
                                                do {
                                                        putStrLn "";
                                                        putStrLn (fileName ++ ": " ++ (show err));
                                                        putStrLn "";
                                                        assertEqual fileName True False;
                                                }
                        }
                )



passed :: Expression -> Bool
passed (BoolVal False) = False;
passed (BoolVal True) = True;
passed (List ((BoolVal False):xs)) = False;
passed (List ((BoolVal True):xs)) = (passed (List xs));
passed (List []) = True;
passed _ = False;




{-|
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        The following are hardcoded Haskell tests.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
-}



testFoo = TestCase (do
                        x <- return 3;
                        assertEqual "testFoo" 3 x;
                )



testStringValue = TestCase (
                                do
                                        h <- return (CharValue 'H');
                                        e <- return (CharValue 'e');
                                        l <- return (CharValue 'l');
                                        o <- return (CharValue 'o');
                                        expected <- return (List [h, e, l, l, o]);
                                        (Right actual) <- return (parse stringAtom "" "\"Hello\"");
                                        assertEqual "testStringValue" expected actual;
                        )


testIdentifier1 = TestCase (
                                do {
                                        expected <- return "Helloworld__";
                                        (Right actual) <- return (parse identifier "" "Helloworld__");
                                        assertEqual "testIdentifier1" expected actual;
                                }
                        )

testIdentifier2 = TestCase (
                                do {
                                        expected <- return "var1_hello";
                                        (Right actual) <- return (parse identifier "" "var1_hello");
                                        assertEqual "testIdentifier2" expected actual;
                                }
```

```
                        )

testIdentifier3 = TestCase (
                                do {
                                        (Left actual) <- return (parse identifier "" "_bad");
                                        assertEqual "testIdentifer3" "" "";
                                }
                        )


testStringConcat1 = TestCase (
                                do {
                                        left <- return (makeStr "Hello");
                                        right <- return (makeStr "World");
                                        expected <- return (Concat left right);
                                        (Right actual) <- return (parse expression "" "\"Hello\"++\"World\"");
                                        assertEqual "testStringConcat1" expected actual;
                                }
                        )

testStringConcatTwoItems = TestCase (
                                do {
                                        s1 <- return (makeStr "One");
                                        s2 <- return (makeStr "Two");
                                        s3 <- return (makeStr "Three");
                                        expected <- return (Concat (Concat s1 s2) s3);
                                        (Right actual) <- return (parse expression "" "\"One\"++\"Two\"++\"Three\"");
                                        assertEqual "testStringConcatTwoItems" expected actual;
                                }
                        )


testConcatThreeItems = TestCase (
                                do {
                                        s1 <- return (makeStr "One");
                                        s2 <- return (makeStr "Two");
                                        s3 <- return (makeStr "Three");
                                        s4 <- return (makeStr "Four");
                                        expected <- return (Concat (Concat (Concat s1 s2) s3) s4);
                                        (Right actual) <- return (parse expression "" "\"One\" ++ \"Two\" ++ \"Three\" ++ \"Four
                                        assertEqual "testConcatThreeItems" expected actual;
                                        (Right actual2) <- return (parse expression "" "\"One\" ++ \"Two\"  ++\"Three\" ++ \"Fou
                                        assertEqual "testConcatThreeItems" expected actual2;
                                }
                        )


testCallNoArgs = TestCase (
                                do {
                                        expected <- return (Call "hello" []);
                                        (Right actual) <- return (parse expression "" "hello()");
                                        assertEqual "testCallNoArgs" expected actual;
                                }
                        )


testCallOneArg = TestCase (
                                do {
                                        expected <- return (Call "foo" [(makeStr "bar")]);
                                        (Right actual) <- return (parse expression "" "foo(\"bar\")");
                                        assertEqual "testCallOneArg" expected actual;
                                }
                        )

testCallEmbeddedCall = TestCase (
```

```
                                do {
                                        expected <- return (Call "foo" [(Call "bar" [])]);
                                        (Right actual) <- return (parse expression "" "foo(bar( ))");
                                        assertEqual "testCallEmbeddedCall" expected actual;
                                }
                        )

testCallEmbeddedCall2 = TestCase (
                                do {
                                        expected <- return (Call "foo" [(Call "bar" [])]);
                                        (Right actual) <- return (parse expression "" "foo ( bar( ) )");
                                        assertEqual "testCallEmbeddedCall" expected actual;
                                }
                        )

testListEmpty = TestCase (
                                do {
                                        expected <- return (List []);
                                        (Right actual) <- return (parse expression "" "[]");
                                        assertEqual "testListEmpty" expected actual;
                                }
                        )

testListEmptyWithSpace = TestCase (
                                do {
                                        expected <- return (List []);
                                        (Right actual) <- return (parse expression "" "[   ]");
                                        assertEqual "testListEmptyWithSpace" expected actual;
                                }
                        )

testListOneItem = TestCase (
                                do {
                                        expected <- return (List [(makeStr "foo")]);
                                        (Right actual) <- return (parse expression "" "[\"foo\"]");
                                        assertEqual "testListOneItem" expected actual;
                                }
                        )

testListTwoItems = TestCase (
                                do {
                                        expected <- return (List [(makeStr "foo"), (makeStr "bar")]);
                                        (Right actual) <- return (parse expression "" "[ \"foo\" , \"bar\" ]");
                                        assertEqual "testListTwoItems" expected actual;
                                }
                        )


testFunctionSimple = TestCase (
                                do {
                                        expected <- return (Function "main" [] (makeStr "Hello"));
                                        (Right actual) <- return (parse function "" "main = \"Hello\"");
                                        assertEqual "testFunctionSimple" expected actual;
                                }
                        )


testFuncNum = TestCase (
                                do {
                                        expected <- return (Function "main" [] (Number 37.65));
                                        (Right actual) <- return (parse function "" "main = 37.65");
                                        assertEqual "testFuncNum" expected actual;
                                }
                        )
```

```
testNum1 = TestCase (
                        do {
                                expected <- return (Number 1.0);
                                (Right actual) <- return (parse numberAtom "" "1");
                                assertEqual "testNum1" expected actual;
                        }
                )

testNum2 = TestCase (
                        do {
                                expected <- return (Number 1.3);
                                (Right actual) <- return (parse numberAtom "" "1.3");
                                assertEqual "testNum2" expected actual;
                        }
                )

testNum3 = TestCase (
                        do {
                                expected <- return (Number 1.3);
                                (Right actual) <- return (parse numberAtom "" "1.30");
                                assertEqual "testNum3" expected actual;
                        }
                )

testNum4 = TestCase (
                        do {
                                expected <- return (Number 1.3);
                                (Right actual) <- return (parse numberAtom "" "1.30000");
                                assertEqual "testNum4" expected actual;
                        }
                )


testNum5 = TestCase (
                        do {
                                expected <- return (Number 2.045);
                                (Right actual) <- return (parse numberAtom "" "2.045");
                                assertEqual "testNum5" expected actual;
                        }
                )


testNum6 = TestCase (
                        do {
                                expected <- return (Number 0.045);
                                (Right actual) <- return (parse numberAtom "" "0.045");
                                assertEqual "testNum6" expected actual;
                        }
                )


testNum7 = TestCase (
                        do {
                                expected <- return (Number 0.045);
                                (Right actual) <- return (parse numberAtom "" ".045");
                                assertEqual "testNum7" expected actual;
                        }
                )



testNum8 = TestCase (
                        do {
                                expected <- return (Number 203.045);
                                (Right actual) <- return (parse numberAtom "" "0203.045");
```

```
                                assertEqual "testNum8" expected actual;
                        }
                )


testNum9 = TestCase (
                        do {
                                (Left actual) <- return (parse numberAtom "" "");
                                assertEqual "testNum9" "" "";
                        }
                )



testInOneFunc = TestCase (
                        do {
                                s <- readFile "testinput/testInOneFunc.hcas";
                                expected <- return (Block [(Function "main" [] (makeStr "Hello world"))]);
                                (Right actual) <- return (parse file "" s);
                                assertEqual "testInOneFunc" expected actual;
                        }
                )

testInTwoFuncs = TestCase (
                        do {
                                s <- readFile "testinput/testInTwoFuncs.hcas";
                                expected <- return (Block [(Function "foo" [] (makeStr "foo")), (Function "bar" [] (make
                                (Right actual) <- return (parse file "" s);
                                assertEqual "testInTwoFuncs" expected actual;
                        }
                )

testInBadString = TestCase (
                        do {
                                s <- readFile "testinput/testInBadString.hcas";
                                (Left actual) <- return (parse file "" s);
                                assertEqual "testInBadString" "" "";
                        }
                )

oneArg = TestCase (
                        do {
                                s <- readFile "testinput/oneArg.hcas";
                                expected <- return (Block [(Function "foo" [(Variable "a")] (makeStr "Hello"))]);
                                (Right actual) <- return (parse file "" s);
                                assertEqual "oneArg" expected actual;
                        }
                )

addSimple = TestCase (
                        do {
                                s <- readFile "testinput/addSimple.hcas";
                                expected <- return (Block [(Function "main" [] (Addition (Variable "x") (Variable "y")))
                                (Right actual) <- return (parse file "" s);
                                assertEqual "addSimple" expected actual;
                        }
                )
```

```
addThree = TestCase (
                                do {
                                        s <- readFile "testinput/addThree.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Addition (Addition x y) z))]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "addThree" expected actual;
                                }
                        )


multOne = TestCase (
                                do {
                                        s <- readFile "testinput/multOne.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        expected <- return (Block [(Function "main" [] (Multiplication x y) )]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "multOne" expected actual;
                                }
                        )



multAdd = TestCase (
                                do {
                                        s <- readFile "testinput/multAdd.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Addition (Multiplication x y) z) )]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "multAdd" expected actual;
                                }
                        )



addMult = TestCase (
                                do {
                                        s <- readFile "testinput/addMult.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Addition x (Multiplication y z)) )]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "addMult" expected actual;
                                }
                        )

multTwo = TestCase (
                                do {
                                        s <- readFile "testinput/multTwo.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Multiplication (Multiplication x y) z) )
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "multTwo" expected actual;
                                }
                        )
```

```
addSubtract = TestCase (
                                do {
                                        s <- readFile "testinput/addSubtract.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Subtraction (Addition x y) z) )]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "addSubtract" expected actual;
                                }
                        )

addDivide = TestCase (
                                do {
                                        s <- readFile "testinput/addDivide.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Addition x (Division y z)) )]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "addDivide" expected actual;
                                }
                        )


parensSimple = TestCase (
                                do {
                                        s <- readFile "testinput/parensSimple.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Parens x) )]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "parensSimple" expected actual;
                                }
                        )


exponentSimple = TestCase (
                                do {
                                        s <- readFile "testinput/exponentSimple.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Exponent x y) )]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "exponentSimple" expected actual;
                                }
                        )



exponentAdd = TestCase (
                                do {
                                        s <- readFile "testinput/exponentAdd.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Addition x (Exponent y z)))]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "exponentAdd" expected actual;
                                }
                        )
```

```
exponentAddSecond = TestCase (
                                do {
                                        s <- readFile "testinput/exponentAddSecond.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Addition (Exponent x y) z))]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "exponentAddSecond" expected actual;
                                }
                        )


equalsAddition = TestCase (
                                do {
                                        s <- readFile "testinput/equalsAddition.hcas";
                                        a <- return (Variable "a");
                                        b <- return (Variable "b");
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        expected <- return (Block [(Function "main" [] (Equals (Addition a b) (Addition x y)))])
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "equalsAddition" expected actual;
                                }
                        )



equalsExponent = TestCase (
                                do {
                                        s <- readFile "testinput/equalsAddition.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (Equals x (Exponent y z)))]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "equalsAddition" expected actual;
                                }
                        )




boolEquals = TestCase (
                                do {
                                        s <- readFile "testinput/boolEquals.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [] (BoolEquals x y ))]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "boolEquals" expected actual;
                                }
                        )



boolGreaterThan = TestCase (
                                do {
                                        s <- readFile "testinput/boolGreaterThan.hcas";
                                        x <- return (Variable "x");
```

```
                y <- return (Variable "y");
                z <- return (Variable "z");
                expected <- return (Block [(Function "main" [] (BoolGreaterThan x y ))]);
                (Right actual) <- return (parse file "" s);
                assertEqual "boolGreaterThan" expected actual;
        }
    )


boolLessThan = TestCase (
        do {
                s <- readFile "testinput/boolLessThan.hcas";
                x <- return (Variable "x");
                y <- return (Variable "y");
                z <- return (Variable "z");
                expected <- return (Block [(Function "main" [] (BoolLessThan x y ))]);
                (Right actual) <- return (parse file "" s);
                assertEqual "boolLessThan" expected actual;
        }
    )


boolAnd = TestCase (
        do {
                s <- readFile "testinput/boolAnd.hcas";
                x <- return (Variable "x");
                y <- return (Variable "y");
                z <- return (Variable "z");
                expected <- return (Block [(Function "main" [] (BoolAnd x y ))]);
                (Right actual) <- return (parse file "" s);
                assertEqual "boolAnd" expected actual;
        }
    )


boolOr = TestCase (
        do {
                s <- readFile "testinput/boolOr.hcas";
                x <- return (Variable "x");
                y <- return (Variable "y");
                z <- return (Variable "z");
                expected <- return (Block [(Function "main" [] (BoolOr x y ))]);
                (Right actual) <- return (parse file "" s);
                assertEqual "boolOr" expected actual;
        }
    )

listPattern = TestCase (
        do {
                s <- readFile "testinput/listPattern.hcas";
                x <- return (Variable "x");
                y <- return (Variable "y");
                z <- return (Variable "z");
                expected <- return (Block [(Function "main" [(ListPattern [x,y])] (makeStr "Hello"))]);
                (Right actual) <- return (parse file "" s);
                assertEqual "listPattern" expected actual;
        }
    )
```

```
listPatternThreeItems = TestCase (
                                do {
                                        s <- readFile "testinput/listPatternThreeItems.hcas";
                                        x <- return (Variable "x");
                                        y <- return (Variable "y");
                                        z <- return (Variable "z");
                                        expected <- return (Block [(Function "main" [(ListPattern [x,y,z])] (makeStr "Hello"))])
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "listPatternThreeItems" expected actual;
                                }
                        )


reservedLet = TestCase (
                                do {
                                        s <- readFile "testinput/reservedLet.hcas";
                                        (Left actual) <- return (parse file "" s);
                                        assertEqual "reservedLet" "" "";
                                }
                        )


letSimple = TestCase (
                                do {
                                        s <- readFile "testinput/letSimple.hcas";
                                        vx <- return (Variable "x");
                                        vy <- return (Variable "y");
                                        fx <- return (Function "x" [] (Number 3));
                                        fy <- return (Function "y" [] (Number 4));
                                        expected <- return (Block [(Function "main" [] (Let (Block [fx, fy]) (Addition vx vy)))])
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "letSimple" expected actual;
                                }
                        )



letOneLine = TestCase (
                                do {
                                        s <- readFile "testinput/letOneLine.hcas";
                                        vx <- return (Variable "x");
                                        vy <- return (Variable "y");
                                        fx <- return (Function "x" [] (Number 3));
                                        fy <- return (Function "y" [] (Number 4));
                                        expected <- return (Block [(Function "main" [] (Let (Block [fx, fy]) (Addition vx vy)))])
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "letOneLine" expected actual;
                                }
                        )


negation = TestCase (
                                do {
                                        s <- readFile "testinput/negation.hcas";
                                        expected <- return (Block [(Function "main" [] (Negation (Variable "x")))]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "negation" expected actual;
                                }
                        )


testCfloat = TestCase (
                                do {
```

```
                                        expected1 <- return (1.0);
                                        (Right actual1) <- return (parse cfloat "" "1.");
                                        assertEqual "testCfloat" expected1 actual1;

                                        expected2 <- return (1.2);
                                        (Right actual2) <- return (parse cfloat "" "1.2");
                                        assertEqual "testCfloat" expected2 actual2;

                                        expected3 <- return (12);
                                        (Right actual3) <- return (parse cfloat "" "1.2e1");
                                        assertEqual "testCfloat" expected3 actual3;

                                        expected4 <- return (10);
                                        (Right actual4) <- return (parse cfloat "" "100e-1");
                                        assertEqual "testCfloat" expected4 actual4;

                                        expected5 <- return (0.05);
                                        (Right actual5) <- return (parse cfloat "" ".05");
                                        assertEqual "testCfloat" expected5 actual5;

                                        expected6 <- return (100000);
                                        (Right actual6) <- return (parse cfloat "" "1e5");
                                        assertEqual "testCfloat" expected6 actual6;

                                        expected7 <- return (0.1);
                                        (Right actual7) <- return (parse cfloat "" ".001e2");
                                        assertEqual "testCfloat" expected7 actual7;
                                }
                        )


variableWithFalse = TestCase (
                                do {
                                        s <- readFile "testinput/variableWithFalse.hcas";
                                        expected <- return (Block [(Function "main" [] (Variable "Falsedude"))]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "variableWithFalse" expected actual;
                                }
                        )



charValue = TestCase (
                                do {
                                        s <- readFile "testinput/char.hcas";
                                        expected <- return (Block [(Function "main" [] (CharValue 'c'))]);
                                        (Right actual) <- return (parse file "" s);
                                        assertEqual "charValue" expected actual;
                                }
                        )







allTests = [testFoo,
                        testStringValue,
                        testIdentifier1,
                        testIdentifier2,
                        testIdentifier3,
                        testStringConcat1,
                        testStringConcatTwoItems,
```

```
testConcatThreeItems,
testCallNoArgs,
testCallOneArg,
testCallEmbeddedCall,
testCallEmbeddedCall2,
testListEmpty,
testListEmptyWithSpace,
testListOneItem,
testListTwoItems,
testFunctionSimple,
testFuncNum,
testNum1,
testNum2,
testNum3,
testNum4,
testNum5,
testNum6,
testNum7,
testNum8,
testNum9,
testInOneFunc,
testInTwoFuncs,
testInBadString,
oneArg,
addSimple,
addThree,
multOne,
multAdd,
addMult,
multTwo,
addSubtract,
addDivide,
parensSimple,
exponentSimple,
exponentAdd,
exponentAddSecond,
equalsAddition,
boolEquals,
boolGreaterThan,
boolLessThan,
boolAnd,
boolOr,
listPattern,
listPatternThreeItems,
reservedLet,
letSimple,
letOneLine,
testCfloat,
variableWithFalse,
negation,
charValue]
```