Haskell Computer Algebra System (HCAS) Language Reference Manual

Rob Tougher (rt2301@columbia.edu)

October 16, 2007

1 Introduction

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

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 $\{-|$ and terminated with $-\}$. They can span multiple lines.

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

```
main True False let in
```

3 Expressions

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.

3.2 Expression Atoms

3.2.1 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.

3.2.2 character

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

```
'c'
```

3.2.3 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.

3.2.4 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
```

3.2.5 boolean

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

3.2.6 '[' 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]
```

3.2.7 '(' expression ')'

You can use parentheses to group mathematical expressions:

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

3.2.8 '-' expression

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

3.3 Boolean Operators

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

3.3.1 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).

3.3.2 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.

3.3.3 expression '<' expression

Represents less than.

3.3.4 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.

3.3.5 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.

3.4 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.

3.4.1 expression '=' expression

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

3.4.2 expression '^' expression

Exponentiation.

3.4.3 expression '*' expression

Multiplication.

3.4.4 expression '/' expression

Division.

3.4.5 expression '+' expression

Addition.

3.4.6 expression '−' expression

Subtraction.

3.5 List Operators

The following operators are used for lists and strings.

3.5.1 expression '++' expression

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

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

3.5.2 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.

3.6 Miscellaneous Expressions

3.6.1 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.

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.

5 Examples

The following are a few examples of the HCAS language:

```
-- ------------------------------------------------
-- Should print "Addition"
-- ------------------------------------------------
main =
    printType (x*y+z)

printType (left*right) =
    "Multiplication"

printType (left+right) =
    "Addition"



-- ------------------------------------------------
-- Should result in (m*a + m*b - m*c)
-- ------------------------------------------------
main =
    distribute(m , a+b-c)

distribute (m, left+right) =
    distribute (m, left) + distribute (m, right)

distribute (m, left-right) =
    distribute (m, left) - distribute (m, right)

distribute (m, p) =
    m*p


-- ------------------------------------------------
-- Should result in "desreveR"
-- ------------------------------------------------
main =
    reverse ("Reversed")

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

6 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
```