Home Construction Modeling Language (HCML)
Final Project Report

Joe Janik – UIN: jj2543 – Email: jj2543@columbia.edu
W4115 Programming Languages and Translators
Final Report Deliverable - Fall 2010 Semester

Table of Contents

1 Introduction:

The objective of this project is to create a programming language where I get to reinforce material learned throughout the semester regarding Lexical Analysis, Parsing, Data Structures, Scope, and Ocaml functional programming.

The project consists of creating a language hereby called Home Construction Modeling Language (HCML) in which a user can model home construction design plans and build a catalog of supplies needed to complete work. Design plans for room configurations will be programmed and compiled where the build parameters are checked for house type and allocated square footage. Then a listing of supplies will be generated for purchasing and consumption of raw materials for best use will be calculated.

Being a first-time home buyer and then rehabbing the home, I've ran into occasions where proper planning of the space layout would have saved me time, money, and frustration. This in turn was my motivation for choosing to implement HCML. Modeling portions of the construction project (such as wall framing and drywall) using HCML will aid in visualizing the layout of the working space and generating the correct amount of supplies needed to complete the job (number of 2x4s and sheets of drywall for a 10x20ft wall section).

The HCML feature set enough to model room dimensions, perform calculations for build supplies based on project definitions, and output patterns for modeling the room objects. Concepts and commands will be utilized in a human friendly manner allowing your “average Joe” to be a Pro HCML Carpenter.

2 Language Tutorial:

The goal of HCML was simply to represent building materials during construction of projects and perform actions pertaining to bookkeeping and building walls of varying length and cutting the drywall needed for the respective wall.

2.1 Features

At a high level, HCML has the following features on which programs can be built aid in your next construction project.

Comments are implemented in a similar fashion to the C programming language. Block comments are enclosed with "/*" at the start of the comment and "*/" at the end of the comment.

Language flow control and layout is implemented using the standard grammatical symbols:

"**;**" Semi-colon to mark the end of a statement
"**,**" Comma to separate arguments within a function
"**()**" Left and Right parentheses for grouping of arguments
"**{ }**" Left and Right Curly braces for grouping block statements with their bodies

Iteration, loops, and evaluation statements are modeled off the C programming language implementations and consist of the following:

- "if else" for evaluation rules for use with comparators
- "for" performing a body of code a set number of times based on the evaluation statement
- "while" similar to "for" but with a differing evaluation statement format

Reserved data types

- String - Strings of characters
- Int - Integer numbers
- Float - Floating point numbers

Operators

- " + " - for addition
- " - " - for subtraction
- " / " - for division
- " * " - for multiplication

The operator precedence and associativity are the same as in the C programming language.

Comparators such as " == " double equal sign for comparing equality, " > " greater-than-sign and " < " for evaluating data ranges are used in HCML.

- " > " - for greater-than comparisons
- " < " - for less-than comparisons
- " == " - for an equation comparison
- " >= " - for greater-than-or-equal-to comparisons
- " <=" - for less-than-or-equal-to comparisons

The next examples sections provides short examples of how to declare variables, create functions and call then, which will aid an aspiring HCML programmer to start coding and along with using their hammer!

2.2 Examples

Declaring variables is achieved simply by stating its type followed by the variable name.

```
int height;
float length;
string myWall;
```

These variables can also be initialized at the same time as they are declared as follows:

```
int height = 60;
float length = 48;
string myWall = “Hello World”;
```

Now for a more interesting use of HCML, we'll build up a wall and calculate how many 2x4 studs are needed for the wall as well as sheets of drywall.

```
float calc2x4x12 (float length){
        float countw = 0.0;
        countw = (length / 16.0) + 4.0;
        return countw;
}

float calcDrywall (float length, float height){
        float countd = 0.0;
        countd = (length / 96.0) + (height /48.0);
        return countd;
}

int main{
        float x = 192.0;
        float y = 144.0;

        float qty2x4x12 = calc2x4x12(x);
        print(“Number of 2x4x12 studs: ”);
        print(qty2x4x12);

        float qtydrywall = calcDrywall(x,y);
        print(“Number of Drywall sheets: ”);
        print(qtydrywall);
}
```

This example illustrates the flexibility of HCML and allows the programmer to describe objects typically found in a construction project and set attributes such as height and length. Then functions can be declared to act on the objects and perform calculations to aid the contractor in gathering quantities and pieces needed for completion of that particular aspect of the construction job.

3 Language Reference Manual:

This manual describes the Home Construction Modeling Language (HCML), motivation for its creation, features of the language, and details the syntax and semantics for proper use in coding software in HCML.

3.1 Lexical Convention:

All HCML programs are in the ASCII character set and stored in a text file with ".hcml" as the file extension.

```
echo "int x = 5;" > my_program.hcml
echo "int y = 1;" >> my_program.hcml
cat my_program.hcml
        int x = 5;
        int y = 1;
```

3.1.1 Tokens:

Input tokens are characterized as one of the following: keywords, literals, operators, and identifiers. White spaces such as the space character, tabs, and returns are considered value don't cares and not taken into account when read in from the input line – they are ignored by the compiler. White spaces are only used as separators in HCML as to distinguish between keywords, literals, operators, and identifiers.

3.1.2 Comments:

Program code that is commentary will start with the character pattern /* and close with */. This commenting style is the same as the one from in the C programming language, where any characters in between /* and */ are ignored by the compiler. Comments should not occur within string literals and nested comments are not supported.

```
/* this is a valid single line comment in HCML */
/* this is a valid multiple line
 * comment in HCML considered a comment block
 */
Invalid uses of the comment identifier:
/* this is a invalid comment /* because of the nested comment identifier
*/
"this is a string with the comment open /* and close */ identifiers embedded, the have no affect as comments here"
```

3.1.3 Identifiers:

Identifiers are a sequence of letters and numbers started by a letter. Identifiers are case sensitive and are used to signify a character pattern definition in HCML such as variables and function names.

Identifier → any sequence of Letters ['a' - 'z' 'A' – 'Z']
followed by any number ['0' - '9']

3.1.4 Keywords:

The following identifiers are reserved in HCML as keywords (having specific predefined meaning) and can not be used in any other context within HCML code.

int	float	string	false
true	if	else	while
for	return		

3.1.5 Constants:

There will be three types of constants within HCML. They are integer, character, and floating number constants.

3.1.6 Integer Constants:

Integer constants will be a sequence of digits that represent a whole number.

Integer constants defined by any digit or sequence of digits
['0' – '9']*
Examples of integers are: 3, 54, 1012, 643671, 0

3.1.7 Character Constants:

Character constants will be an ASCII alphanumeric letter consisting of the following set:

Character constants defined by any single letter of either upper or lower case
letters ['a' -'z'] or ['A' - 'Z']
Examples of characters are: a, x, F, D

Blank space and tab characters will be ignored until the end of line character " \n " is detected.

3.1.8 Floating Constants:

Floating constants will be a sequence of digits that represent a fractional number, they will have the keyword double.

Floating constants defined by any digit either preceded or followed by a dot character '.' representing a fractional number.

['0' – '9'] . ['0' – '9']

Examples of floating constants are:

1.25, 0.54, 101.2, 64.3, .671, 0.1

3.1.9 String Literals:

String literals are a sequence of characters followed by any combination of characters or integers.

String literals defined by starting with an opening quote, any character sequence, and then followed by a closing quote.

['a' – 'z'] ['A' – 'Z'] ['0' – '9']

Examples of string literals are:

This is a string literal example!

2tabs followed by text

“1 + 2 = 3 is a string literal because of the quote enclosure”

3.2 Meaning of Identifiers:

Identifiers will begin with a character and can be followed by any sequence of characters or digits. Identifiers can not start with a digit.

Identifiers defined by starting with a character:

['a' – 'z'] ['A' – 'Z']

```
int id1 = 5;      /* correct usage of an identifier as a variable names */
string MyName = “John Doe”;
```

Boolean data types are used within HCML as identifier keywords “true” and “false”. The values of each are integer and inverse of each other, either 0 or 1.

Boolean types true and false map in the following way:

```
true = 1;
false = 0;
```

3.3 Expressions:

HCML provides support for building expressions with a common set of arithmetic and comparator operators.

3.3.1 Primary Expressions:

Primary expressions consist of identifiers, strings, floating and/or integer numbers, characters, and any sequence making up a declaration of the variable or re-assignment. All identifiers, string literals, and constant expressions are primary expressions.

Primary Expression defined for declaration and re-assignment:

```
int variableX = 100;
variableX = 90;
string myExpression = hello world;
```

3.3.2 Postfix Expressions:

Postfix expressions consist function calls in HCML. The dot '.' operator separators the caller from the function to be called.

3.3.3 Function Calls:

Function calls are implemented as the following in HCML.

Definition for calling functions:

```
identifier ( args are optional, dependent on called func)
build(x, y);
generate();
```

3.3.4 Multiplicative Operators:

The multiplicative operators of multiplication and division are implemented in HCML. Operator precedence is from left to right when both operators are present in the expression.

The operator for multiplication is the character ' * ' and results in the product of the first and second operands.

Examples of multiplication operator usage:

```
int x = 3; int y = 5;
x * y ;  /* results in 15 */
x * y *x;        /* results in 45, operations left to right */
x * (y * x);     /* results in 45, with 2nd and 3rd operand
                    computed first. */
```

The operator for division is the character ' / ' and results in the quotient of the first operand by the second.

Examples of division operator usage:

```
int x = 2; int y = 4;
y / x;   /* results in 2 */
y / x / x;        /* results in 1, operations left to right */
y / (y / x);      /* results in 2, with 2nd and 3rd operand
                     computed first. */
```

3.3.5 Additive Operators:

The additive operators of addition and subtraction are implemented in HCML. Operator precedence is from left to right when both operators are present in the expression.

The operator for addition is the character ' + ' and results in the sum of the first and second operands.

Examples of addition operator usage:

```
int x = 5; int y = 7;
x + y;  /* results in 12 */
x + y + x;        /* results in 17, operations left to right */
x + (y + x);      /* results in 17, with 2nd and 3rd operand
                     computed first. */
```

The operator for subtraction is the character ' - ' and results in the difference of the first operand by the second.

Examples of division operator usage:

```
int x = 1; int y = 6;
y – x;  /* results in 5 */
y - x – x;        /* results in 4, operations left to right */
y - (x – x);      /* results in 6, with 2nd and 3rd operand
                     computed first. */
```

3.3.6 Operator Precedence

Operator precedence is from left to right when multiple operators are present in the expression. The PMDAS rule is in effect for operation when all operators are present in the expression.

Parenthesis → Multiplication → Division → Addition → Subtraction

Examples of operator precedence:

```
int x = 2; int y = 5;
x * y + y / y;   /* results in  11*/
y - x * x;        /* results in 1 */
y * (x + x);      /* results in 20, with 2nd and 3rd operand
                     computed first. */
```

3.3.7 Relational and Equality Operators:

Relational operators are provided in HCML. The operators are used to compare 2 operands and result in a Boolean value of either true or false.

3.3.8 Equality operators:

An identifier with " == " double equal sign signifies the equality operator. It returns true when the left operand is equal to the right operand.

Examples of equality operator:

```
int x = 2; int y = 5; int z = 2;
x == y;          /* evaluates to false */
x == z;          /* evaluates to true */
```

An identifier with "!=" bang equal sign signifies the "not equal to" operator. It returns true when the left operand is not equal to the right operand.

Examples of not equal to operator:

```
int x = 2; int y = 5; int z = 2;
x != y;          /* evaluates to true */
x != z;          /* evaluates to false */
```

3.3.9 Greater-than, Greater-than-or-equal-to operators:

An identifier with " > " greater-than character signifies the greater-than operator. It returns true when the left operand is greater-than the right operand.

Examples of greater-than operator:

```
int x = 2; int y = 5; int z = 2;
y > x;           /* evaluates to true */
x > z;           /* evaluates to false */
```

An identifier with " >= " greater-than-or-equal-to character signifies the greater-than-or-equal-to operator. It returns true when the left operand is greater-than or equal to the right operand.

Examples of greater-than-or-equal-to operator:

```
int x = 2; int y = 5; int z = 2;
x >= y;          /* evaluates to false */
x >= z;          /* evaluates to true */
```

3.3.10 Less-than, Less-than-or-equal-to operators:

An identifier with " < " less-than character signifies the less-than operator. It returns true when the left operand is less-than the right operand.

Examples of less-than operator:

```
int x = 2; int y = 5; int z = 2;
y < x;          /* evaluates to false */
z < y;          /* evaluates to true */
```

An identifier with " <= " less-than-or-equal-to character signifies the less-than-or-equal-to operator. It returns true when the left operand is less-than or equal to the right operand.

Examples of less-than-or-equal-to operator:

```
int x = 2; int y = 5; int z = 2;
x <= y;         /* evaluates to true */
x <= z;         /* evaluates to true */
```

3.3.11 Logical AND Operator:

Logical AND operator ties expressions together in a conditional statement. In order for a logical AND operation to evaluate to true, all expressions within the common conditional statement need to evaluate to either all true or false. The double ampersand "&&" is the identifier for the logical AND operation.

Examples of the logical AND operator:

```
int x = 2; int y = 5; int z = 2;
(x == z) && (y > x);  /* evaluates to true */
(x == z) && (y < x);  /* false due to 2nd expression */
```

3.3.12 Logical OR Operator:

Logical OR operator excludes expressions apart in a conditional statement. In order for a logical OR operation to evaluate to true, only one expression within the common conditional statement needs to evaluate to either all true. The double pipe " || " is the identifier for the logical OR operation.

Examples of the logical OR operator:

```
int x = 2; int y = 5; int z = 2;
(x == z) || (y < x);    /* evaluates true due to 1st expression */
(x > z) || (y < x);     /* evaluates false , neither are true */
```

3.3.13 Conditional Operator:

Conditional operations are supported in HCML with the if and else identifiers. A conditional statement can be built using the “if” identifier followed by the “else” when two direction paths are present.

Examples of conditional operators where statement1 is executed if the conditional expression evaluates to true:

```
if (expression)
        statement1;
else
        statement2;
```

3.3.14 Assignment Expressions:

The equal sign ' = ' is used as an assignment operator for assigning values to variables. Cascading common type variables is possible with the comma operator and assignment variable to assign the same value to each variable of that line.

Examples of assignment operator:

```
int x = 5;        /* initialize variable to value 5 */
x = 25;           /* reassign value of x to 25 */
int y,z = 0;      /* initialize variables y and z to value 0 */
```

3.3.15 Comma Operator:

The comma ' , ' operator is used as a separator between expressions and statements. Cascading common type variables is possible with the comma operator. Another use of the comma operator is separating the conditional expressions within a conditional statement.

Examples of comma operator:

```
int x, y, z;      /* initialize common type variables */
for(expression1, expression2, expression3)
build(10, 20);
```

3.4 Declarations:

3.4.1 Meaning of Declarators:

Declarations are used for declaring the use of an identifier with a type. The declaration binds the identifier with the type given in the expression.

Examples of a Declaration:

```
int x;      /* declare variable x as type integer */
String myString;        /* declaring a string varaible */
double y;       /* declare variable y as type float */
```

3.4.2 Initialization:

Initialization of variables is achieved by using the assignment operator within an expression. It binds the value on the right of the equal sign with the variable on the left of the equal sign. Initialization is set as default to the null value if no value is given for a variable when first declared.

Examples of Initializations:

```
int x = 0;
String myString; = Building a wall;
char  c;          /* initialized to null */
```

3.4.3 Function Declarators:

Function declarations are used for declaring the type of a value returned by a function and provide the block for it's use. The declaration binds all variables to the block with the only exception being the return value of the function. Here main and calc-area are functions with there respective bodies.

Examples of a function declarations:

```
int main{
        int x = 5;
        int y = 6;
        int area = calc_wall(x,y);
        string statement  = area of the wall is ;
        print(statement, area);
        return 0;
}

int calc-area(int x, int y){
        return (x * y);
}
```

3.5 Statements:

3.5.1 Expression Statement:

Expression statements perform evaluations on left and right operands. Expressions are defined as a left operand and a right operand that are operated on by an operator. The operations that can be expressed by expressions are: addition, subtraction, multiplication, division, all the comparisons (greater-than, greater-than-or-equal-to, less-than, less-than-or-equal-to, etc...), logical operations of AND and OR, assignments, and equalities.

Examples of expressions:

```
int x, y, z = 5;
x = x + y + z;
(x >= y) || (y > z);
```

3.5.2 Iteration Statements:

Iteration statements are supported in HCML using the while and for loops. The while loop evaluates the conditional statement within the parentheses and performs the block if true. An incremental expression is needed within the block to modify the conditional value to prevent an endless loop.

Examples of a while loop:

```
while(conditional){
        expression1;
        expression2;
        conditional_increment_expression;
}
while(int x < 5){
        statement1;
        statement2;
        x = x + 1;
}
```

The for loop evaluates the conditional statement within the parentheses and performs the block if true. The control of the loop is also defined within the parentheses and listed as the expression to change the value of the conditional to prevent an endless loop.

Examples of a for loop:

```
for(statement, conditional, expression){
        expression1;
        expression2;
}
```

3.6 Scope:

Scope in HCML is given to block statements and any variable declared within the block is visible to all expressions within the same block. Once a block has been closed by the right curly brace symbol " } " all declarations and stored values are not visible within the next block.

3.7 Preprocessing:

Language flow control and layout will be implemented using the standard grammatical symbols: semi-colon, left and right parentheses, and left and right curly braces. Scope will be determined by the use of the left and right curly braces, where the left curly brace opens a block statement and the right curly brace closes the block statement.

3.7.1 Line Statements:

Line control is achieved by using the semi-colon character " ; " to close a line after a statement, declaration, or expression. Carryover to a new line is implemented simply by carrying over the text to the next line with indentation.

Examples of line control:

```
int x;      /* line closure with ; character*/
int x, t, my_long_variable_name, id1,
        id2, id3, id4;          /* line carry over */
```

3.7.2 Block statements:

Block statements are achieved by using the Left "{ " and Right " }" Curly braces to enclose blocks of declarations, expressions, and statements. Scope is only reserved in blocks.

Examples of Block control:

```
int main{
        declaration1;
        statement1;
        return 0;
}
int run{
        declaration2;
        statement2;
        return 0;
}
```

Declarations 1 and 2 have no relation to each other, they are out of scope due to belong to different blocks. Same is true for statements 1 and 2.

3.8 Grammar:

Declaration:

declarator = initializer

type-specifier declaration

Function definition:

declaration - identifier – statement

Type-specifier: one of

int, float, string

Statement:

expression-statement

iteration-statement

Conditional-Statement:

if (expression) *statements*

if (expression) *statements* else *statements*

Iteration-Statement:

while (expression) *statements*

for (expression, expression, expression) *statements*

Expression:

assignment-expression

expression, assignment-expression

Logical-OR-Expression:

expression || expression

Logical-AND-Expression:

expression && expression

Equality-Expression:

expression == expression - or - expression != expression

Additive-Expression:

expression + expression - or - expression – expression

Multiplicative-Expression:

expression * expression - or - expression / expression

Postfix-Expression:

expression . identifier

Constant:

integer-constant

character-constant

floating-constant

4 Project Plan:

4.1 Project Process, Roles, and Responsibilities

For this project, I was the only member contributing due to being a CVN student and required to work independently. This allowed for both good and bad within the project where the good was communication and direction of the project was mine alone however the bad was that division of labor was not an option and all project work was mine to take and perform to.

4.2 Programming Style

The programming style used for this project conformed to the style used in the class examples and microc language. Comments were included where needed to describe the function or types that were being declared for the particular component. Spacing of two was used for the left as the indentation and parentheses were used to explicitly call attention to how evaluations or matches were defined.

4.3 Project Time-Line

The project time-line is closely related to the project milestones defined by the deliverable due dates (Project Proposal, LRM, and Final Report) and follows below:

Dates week of (start - end)	Project Deliverable or Source Component
September 8 – September 29	Project Proposal white paper
October 4 – November 3	Language Reference Manual
November 8 – November 24	Scanner, Parser, and AST
November 28 – December 8	Interpreter and Testing
December 20 – December 22	Project Final Report

4.4 Project Log

Unfortunately I didn't create a repo (git log would have been nice to provide the details here – I used git extensively in my W4118 OS class this semester) so tracking changes and logs are based on notes I created in my notebook or on stickies.

Date	Project Deliverable or Source Component
Wed, September 29, 2010	Project Proposal white paper submitted
Wed, November 03, 2010	Language Reference Manual submitted
November 8 – December 8 with time off for midterm prep	Review class Ocaml calc microc examples, start code of scanner, parser, ast, and interpreter.
December 8 – December 22 with time off for final prep	Modified test script file from microc example and developed hcml test cases, produced final report

4.5 Project Tools and Environment

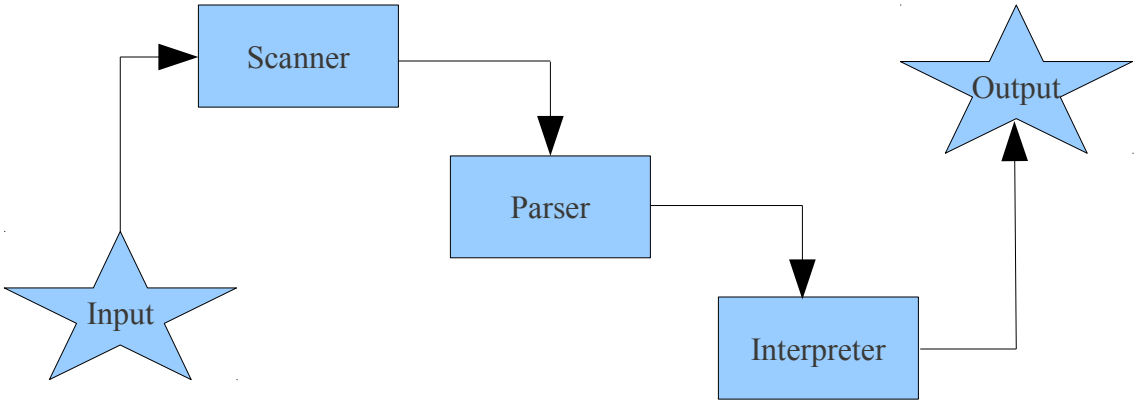
I worked primarily on a x86 desktop PC loaded with: Linux distribution Ubuntu 10.04 LTS (the Lucid Lynx – updated to Linux Kernel version 2.6.32-26-generic-pae), Objective Caml apt-package version 3.11.2 (this includes toplevel, lexer and parser generators, and compiler), gedit 2.30.3 (for editing source files), GNU make version 3.81, and Open Office version 3.2.

5 Architectural Design:

HCML is an interpreted language so there are no back-end components such as a compiler or intermediary byte-code interpreter and translator. Being a CVN student and not part of a team for this project, each component was implemented by myself.

5.1 Architectural Block Diagram

The block diagram for the HCML translator is given below showing the major components of the system and the way in which the components interact.



5.2 Interfaces and Components

The interfaces and flow can be seen from the block diagram above where the arrows show which components interact with the next. Component dependencies exist in that the scanner component depends and opens the parser component. In turn, the parser component depends and opens the AST component for data types/structures where they are listed and build from by the interpreter. The input is a HCML source file for a program. It is provided to the scanner as its input and starts the translation from HCML source to output.

5.2.1 Scanner

The Scanner (scanner.mll) is implemented using the Ocamllex tool with lex syntax containing regular expressions for the HCML language tokens. The scanner uses the input source file and tokenizes the input feed while removing white spaces, tabs, end-of-lines, and comments.

5.2.2 Parser

The Parser (parser.mly) is implemented using the Ocamlyacc tool with yacc syntax containing yacc directives, grammar, and action rules for the HCML language. The parser uses the tokens passed in by the scanner component as its input.

5.2.3 Interpreter

The Interpreter (interpret.ml) is implemented using the Ocaml functional programming language. It builds the inputted program and evaluates the instructions/expressions based on the parser generated abstract syntax tree. The interpreter walks the abstract syntax tree, performs type checking, then creates symbol tables for both local and global variables, and evaluates expressions associated with them.

6 Test Plan:

Testing started early on in the project when adding features to HCML such as data types and function types. Making sure that the front end was working correctly by scanning the inputted text and then parsing to get the program into a structure. I first started to manually type in the inputs using the HCML interpreter. I would type in both valid and invalid statements to make sure they were picked up correctly such that the valid statements would be accepted and the invalid statements would throw an exception.

Soon after I started manually testing, I started to digest the microc language and test-suite provided by Professor Edwards. I liked the automated testing it had and tried to modify it for my project along with the Makefile procedure. I was able to adopt the make system however the testing broke with the directory structure and was unable to find the comparison correct output files to compare to. So the nice test suite pass/fail printouts and automation was not used however I did modify the test cases for HCML use. I was able to test with these modified version by single input into the HCML interpreter using the " < " input redirection operator in the Unix shell environment. Example: ./hcml < test-if.hcml

6.1 Test cases

test-addint.hcml	Test case to evaluate an expression involving addition of two integers.
test-addfloat.hcml	Test case to evaluate an expression involving addition of two float values.
test-mathint.hcml	Test case to evaluate order of math operations of integer values.
test-mathfloat.hcml	Test case to evaluate order of math operations of float values.
test-fib.hcml	Test case to evaluate Fibonacci numbers of data set 0-5.
test-for.hcml	Test case to run through a for loop.
test-vardec.hcml	Test case in which variables of each type (int, float, string) were declared.
test-varassign.hcml	Test case in which each type variable was declared and initialized.
test-gcd.hcml	Test case to evaluate the greatest common divisor.
test-ifesle.hcml	Test case to toggle between if else cases.
test-while.hcml	Test case to run through a while loop.
test-example.hcml	Test case from Section 2 Language Tutorial example.

6.2 Sample Test Runs

Example run of test-example.hcml:

janikj@ubuntu-desktop:~/Desktop/hcml$ ./hcml < test-example.hcml

Output:

Number of 2x4x12 studs: 16

Number of Drywall sheets: 5

janikj@ubuntu-desktop:~/Desktop/hcml$ ./hcml < test-mathfloat.hcml

Output:

7 Lessons Learned:

Overall I found this project challenging in that everything I was learning was new material for me and needed to be implemented to make the final project successful. From Lexical analysis for generating tokens, parsing the tokens to create structures representative of the program, and then to evaluating the expressions...the details of these processes are mechanical and this project was a great way to reinforce the material learned from the lectures. I was never exposed to a functional programming language and there was a learning curve associated with Ocaml. I watched the introduction to Ocaml lecture multiple times and paid closer attention to lecture material if it had direct references and examples implemented in Ocaml.

Particular to the project, I started working on the scanner and parser based on the lecture information and needed to use the references listed in section 9 for specifics about the lex and yacc syntaxes. It would have been beneficial to consult the books and chapters related to lex/yacc before jumping into the scanner and parser. Additionally it was nice to have examples like calculator and microc although would be nice to get the information released at the same time as the intro to Ocaml lecture. The Ocaml compiler had a nice and useful feature that called out missed match cases and gave an example of a valid case. This was useful when debugging the interpreter.

I felt as though the homework assignments for PLT were few and far between and focused more on test-type material instead of directly applicable to the project. Only homework 1 had Ocaml material reinforcing the introduction Ocaml lecture. If there was more homework spread out throughout the semester dealing specifically with the project, I think it'd be better to digest the project in smaller amounts. Something like have more formal deliverables for the project such as the different components (scanner, parser, ast, etc...) with specific due dates and have the grading feedback for each to build upon for the next component. During this same semester, I also took the W4118 OS course and we had a 2 week development cycle for each assignment. There was constant reinforcement of course material with each coding assignment. The PLT project has it all wrapped up in this one huge coding deliverable at the end...just think it'd be more digestible and manageable broken down into smaller chucks spread throughout the semester.

Advice for future students would be to start looking and digesting Ocaml code early on in the semester. Get a hold of the calc and microc examples and understand how the components work to create the final output. Allow plenty of time to understanding the Ocaml language and syntax before jumping into any project code. Unfortunately I didn't have enough time to complete all the planned functionality from proposal to finish (such as bin packing to minimize waste when drywalling a room by using a best fit algorithm).

8 Appendix:

8.1 scanner.mll

```
(* Header *)
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
(* opening comment *)
| "/*"    { comment lexbuf }
(* control flow *)
| '('     { LEFTPR }
| ')'     { RGHTPR }
| '{'     { LEFTBR }
| '}'     { RGHTBR }
| ';'     { SEMI }
| ','     { COMMA }
| '='     { ASSIGN }
(* Integer operators *)
| '+'     { IPLUS }
| '-'     { IMINUS }
| '*'     { ITIMES }
| '/'     { IDIVIDE }
(* Float operators *)
| "+"     { FPLUS }
| "-"     { FMINUS }
| "*"     { FTIMES }
| "/"     { FDIVIDE }
(* comparisons *)
| "=="    { EQ }
| "!="    { NEQ }
| '<'     { LT }
| "<="    { LEQ }
| ">"     { GT }
| ">="    { GEQ }
| “||”     { OR }
```

```
| “&&” { AND }
(* branches/loops *)
| "if"    { IF }
| "else" { ELSE }
| "for"  { FOR }
| "while"        { WHILE }
| "return"       { RETURN }
(* variables *)
| "int" { INT }
| "float" { FL }
| "bool" { BOOL }
| "string" { STRING }
| ['0'-'9']+ as lxm { INTEGER(int_of_string lxm) }
| ['0'-'9']+ ['.'] ['0'-'9']* ('e' ['-' '+']? ['0'-'9']+)? as lxm { FLOAT (float_of_string lxm)}
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { IDENT(lxm) }
| eof    { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
(* closing comment *)
 "*/"    { token lexbuf }
| _      { comment lexbuf }

(* Trailer *)
```

8.2 parser.mly

```
%{ open Ast %}

/* declarations */

%token SEMI LEFTPR RGHTPR LEFTBR RGHTBR COMMA RETURN ASSIGN
%token IPLUS IMINUS ITIMES IDIVIDE FPLUS FMINUS FTIMES FDIVIDE
%token EQ NEQ LT LEQ GT GEQ OR AND
%token IF ELSE FOR WHILE INT FL BOOL STRING
%token <int> INTEGER
%token <float> FLOAT
```

```
%token <string> IDENT
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ OR AND
%left IPLUS IMINUS FPLUS FMINUS
%left ITIMES IDIVIDE FTIMES FDIVIDE
%nonassoc UMINUS

%start program
%type <Ast.program> program

%%
/* rules */

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

fdecl:
   vtype_list IDENT LEFTPR formals_opt RGHTPR LEFTBR vdecl_list stmt_list RGHTBR
                 { { ftype = $1;
                                fname = $2;
                     formals = $4;
                     locals = List.rev $7;
                     body = List.rev $8 } }
vtype_list:
    /* nothing */          { [] }
  | vtype_list fdecl       { $2 :: $1 }

formals_opt:
```

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

formal_list:
    IDENT                    { [$1] }
  | formal_list COMMA IDENT    { $3 :: $1 }

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

vdecl:
    INT IDENT SEMI           { $2 }
  | FL IDENT SEMI            { $2 }
  | STRING IDENT SEMI        { $2 }

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

stmt:
    expr SEMI                { Expr($1) }
  | RETURN expr SEMI         { Return($2) }
  | LEFTBR stmt_list RGHTBR  { Block(List.rev $2) }
  | IF LEFTPR expr RGHTPR stmt %prec NOELSE
                             { If($3, $5, Block([])) }
  | IF LEFTPR expr RGHTPR stmt ELSE stmt
                             { If($3, $5, $7) }
  | FOR LEFTPR expr_opt SEMI expr_opt SEMI expr_opt RGHTPR stmt
                             { For($3, $5, $7, $9) }
  | WHILE LEFTPR expr RGHTPR stmt
                             { While($3, $5) }

expr_opt:
    /* nothing */          { Noexpr }
  | expr                   { $1 }
```

```
expr:
    vtype INTEGER          { Integer($2) }
  | vtype FLOAT                  { Float ($2) }
  | vtype IDENT            { Ident($2) }
  | expr IPLUS   expr       { Binop($1, IAdd, $3) }
  | expr IMINUS  expr       { Binop($1, ISub, $3) }
  | expr ITIMES  expr       { Binop($1, IMul, $3) }
  | expr IDIVIDE expr       { Binop($1, IDiv, $3) }
  | expr FPLUS   expr       { Finop($1, FAdd, $3) }
  | expr FMINUS  expr       { Finop($1, FSub, $3) }
  | expr FTIMES  expr       { Finop($1, FMul, $3) }
  | expr FDIVIDE expr       { Finop($1, FDiv, $3) }
  | IMINUS expr %prec UMINUS  { $2 }
  | FMINUS expr %prec UMINUS  { $2 }
  | expr EQ     expr        { Cop($1, Eql, $3) }
  | expr NEQ    expr        { Cop($1, Neq, $3) }
  | expr LT     expr        { Cop($1, Lt,  $3) }
  | expr LEQ    expr        { Cop($1, Leq, $3) }
  | expr GT     expr        { Cop($1, Gt,  $3) }
  | expr GEQ    expr        { Cop($1, Geq, $3) }
  | expr OR   expr        { Cop($1, Or, $3) }
  | expr AND  expr        { Cop($1, And, $3) }
  | IDENT ASSIGN expr          { Assign($1, $3) }
  | IDENT LEFTPR actuals_opt RGHTPR { Call($1, $3) }
  | LEFTPR expr RGHTPR       { $2 }

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

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

8.3 ast.ml

```
type iop = IAdd
  | ISub
  | IMul
  | IDiv

type fop = FAdd
  | FSub
  | FMul
  | FDiv

type cop = Eql
  | Neq
  | Lt
  | Leq
  | Gt
  | Geq
  | Or
  | And

type uop = USub

type vartype = Int of int
  | Float of float
  | String of string

type expr =
    Integer of int
  | Float of float
  | Ident of string
  | Binop of expr * operator * expr
  | Finop of expr * fop * expr
  | Uop of uop * expr
  | Cop of expr * cop * expr
  | Assign of string * expr
  | Call of string * expr list
```

```
  | Noexpr

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

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

type program = string list * func_decl list

let rec string_of_expr = function
    Integer(l) -> string_of_int l
  | Float (f) -> string_of_float f
  | Ident(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
          Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/"
      | Eql -> "==" | Neq -> "!="
      | Lt -> "<" | Leq -> "<=" | Gt -> ">" | Geq -> ">=") ^ " " ^
      string_of_expr e2
  | Finop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
          FAdd -> "+" | FSub -> "-" | FMul -> "*" | FDiv -> "/") ^ " " ^
      string_of_expr e2
```

```
  | Uop(o, e2) ->
      (match o with
           USub -> "-" ) ^ " " ^
      string_of_expr e2
  | Cop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
           Eql -> "==" | Neq -> "!=" | Lt -> "<" | Leq -> "<=" | Gt -> ">"
      | Geq -> ">=" | Or -> "||" | And -> "&&") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_vdecl id =
    "int " ^ id ^ ";\n"
 (* | "float " ^ id ^ ";\n"
  | "string " ^ id ^ ";\n" *)

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
```

```
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

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

8.4 interpret.ml

```
open Ast

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

exception ReturnException of int * int NameMap.t

(* Main entry point: run a program *)

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

  (* Invoke a function and return an updated global symbol table *)
  let rec call fdecl actuals globals =

    (* Evaluate an expression and return (value, updated environment) *)
    let rec eval env = function
          Integer(i) -> i, env
      | Float(i) -> i, env
      | Noexpr -> 1, env (* must be non-zero for the for loop predicate *)
      | Ident(var) ->
```

```
        let locals, globals = env in
        if NameMap.mem var locals then
          (NameMap.find var locals), env
        else if NameMap.mem var globals then
          (NameMap.find var globals), env
        else raise (Failure ("undeclared identifier " ^ var))
  | Binop(e1, iop, e2) ->
        let v1, env = eval env e1 in
      let v2, env = eval env e2 in
        let boolean i = if i then 1 else 0 in
        (match iop with
          IAdd -> v1 + v2
        | ISub -> v1 - v2
        | IMul -> v1 * v2
        | IDiv -> v1 / v2), env
  | Finop(f1, fop, f2) ->
        let v1, env = eval env f1 in
      let v2, env = eval env f2 in
        let boolean f = if f then 1 else 0 in
        (match fop with
          FAdd -> v1 +. v2
        | FSub -> v1 -. v2
        | FMul -> v1 *. v2
        | FDiv -> v1 /. v2), env
  | Cop(e1, cop, e2) ->
        let v1, env = eval env e1 in
      let v2, env = eval env e2 in
        let boolean i = if i then 1 else 0 in
        (match cop with
        | Eql -> boolean (v1 = v2)
        | Neq -> boolean (v1 != v2)
        | Lt -> boolean (v1 < v2)
        | Leq -> boolean (v1 <= v2)
        | Gt -> boolean (v1 > v2)
        | Geq -> boolean (v1 >= v2)
        | Or -> boolean (v1 || v2)
```

```
        | And -> boolean (v1 && v2)), env
  | Assign(var, e) ->
        let v, (locals, globals) = eval env e in
        if NameMap.mem var locals then
          v, (NameMap.add var v locals, globals)
        else if NameMap.mem var globals then
          v, (locals, NameMap.add var v globals)
        else raise (Failure ("undeclared identifier " ^ var))
  | Call("print", [e]) ->
        let v, env = eval env e in
        print_endline (string_of_int v);
        0, env
  | Call(f, actuals) ->
        let fdecl =
          try NameMap.find f func_decls
          with Not_found -> raise (Failure ("undefined function " ^ f))
        in
        let actuals, env = List.fold_left
            (fun (actuals, env) actual ->
                let v, env = eval env actual in v :: actuals, env)
            ([], env) (List.rev actuals)
        in
        let (locals, globals) = env in
        try
          let globals = call fdecl actuals globals
          in 0, (locals, globals)
        with ReturnException(v, globals) -> v, (locals, globals)
in

(* Execute a statement and return an updated environment *)
let rec exec env = function
       Block(stmts) -> List.fold_left exec env stmts
  | Expr(e) -> let _, env = eval env e in env
  | If(e, s1, s2) ->
        let v, env = eval env e in
        exec env (if v != 0 then s1 else s2)
```

```
  | While(e, s) ->
        let rec loop env =
          let v, env = eval env e in
          if v != 0 then loop (exec env s) else env
        in loop env
  | For(e1, e2, e3, s) ->
        let _, env = eval env e1 in
        let rec loop env =
          let v, env = eval env e2 in
          if v != 0 then
            let _, env = eval (exec env s) e3 in
            loop env
          else
            env
        in loop env
  | Return(e) ->
        let v, (locals, globals) = eval env e in
        raise (ReturnException(v, globals))
in

(* Enter the function: bind actual values to formal arguments *)
let locals =
  try List.fold_left2
        (fun locals formal actual -> NameMap.add formal actual locals)
        NameMap.empty fdecl.formals actuals
  with Invalid_argument(_) ->
      raise (Failure ("wrong number of arguments passed to " ^ fdecl.fname))
in
(* Initialize local variables to 0 *)
let locals = List.fold_left
      (fun locals local -> NameMap.add local 0 locals) locals fdecl.locals
in
(* Execute each statement in sequence, return updated global symbol table *)
snd (List.fold_left exec (locals, globals) fdecl.body)

(* Run a program: initialize global variables to 0, find and run "main" *)
```

```
  in let globals = List.fold_left
      (fun globals vdecl -> NameMap.add vdecl 0 globals) NameMap.empty vars
  in try
    call (NameMap.find "main" func_decls) [] globals
  with Not_found -> raise (Failure ("did not find the main() function"))
```

8.5 hcml.ml

```
let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
    ignore (Interpret.run program)
```

9 References:

- HCML white paper proposal submitted: 9/29/2010
- HCML Language Reference Manual (LRM) submitted: 11/3/2010
- Microc language as jumping point for building off of and using automated test suite and make system.
- Developing Applications with Objective Caml – O'Reilly Publishing
  http://caml.inria.fr/pub/docs/oreilly-book/ocaml-ora-book.pdf
- The Objective Caml system release 3.12 Documentation and User's Manual
  http://caml.inria.fr/distrib/ocaml-3.12/ocaml-3.12-refman.pdf
- Introduction to Objective Caml by Jason Hickey
  http://www.cs.caltech.edu/courses/cs134/cs134b/book.pdf