Home Construction Modeling Language (HCML)
Language Reference Manual

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

Table of Contents

1 Introduction:

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.

2 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;
```

2.1 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.

2.2 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"
```

2.3 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']

2.4 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.

null	double	end	build
true	if	string	floor
false	while	construct	window
for	return	position	door
else	print	wall	generate
int	char	ceiling	demo

2.5 Constants:

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

2.5.1 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

2.5.2 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.

2.5.3 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

2.6 String Literals:

String literals are a sequence of characters followed by any combination of characters or integers. String literals will be enclosed by a quotation pair “ ”.

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 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

4 Expressions:

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

4.1 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”;
```

4.2 Postfix Expressions:

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

4.2.1 Function Calls:

Function calls are implemented as postfix expressions in HCML.

Definition for calling functions:

```
expression . identifier
wall.build();
project.generate();
```

4.3 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. */
```

4.4 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. */
```

4.5 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. */
```

4.6 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.

4.6.1 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 */
```

4.6.2 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 */
```

4.6.3 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 */
```

4.7 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 */
```

4.8 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 */
```

4.9 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;
```

4.10 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 */
```

4.11 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)
wall.build(10, 20);
```

5 Declarations:

5.1 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 */
```

5.2 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 */
```

5.3 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.

Examples of a function declarations:

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

int calc_area(int x, int y){
        int area = x * y;
        return area;
}
```

6 Statements:

6.1 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)
```

6.2 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;
        }
```

7 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.

8 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.

8.1 Line Control:

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 using the " / " forward-slash character to carry forward the line.

Examples of line control:

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

8.2 Block Control:

Block control is 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:

```
main{
        declaration1;
        statement1;
}
run{
        declaration2;
        statement2;
}
```

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.

9 Grammar:

Declaration:

declarator = initializer

type-specifier declaration

Function definition:

declaration - identifier – statement

Type-specifier: one of

char, int, double

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

expression != expression

Additive-Expression:

expression + expression

expression – expression

Multiplicative-Expression:

expression * expression

expression / expression

Postfix-Expression:

expression . identifier

Constant:

integer-constant

character-constant

floating-constant