CRAWL Language Reference Manual

Rajesh Venkataraman
rv2187@columbia.edu

Amoghavarsha Ramappa
ar2645@columbia.edu

Carter Adams
carteradams@gmail.com

1 Lexical Conventions

A CRAWL program is consists of a single translation unit stored in a file. The file is written using the ASCII character set.

1.1. Comments

CRAWL comments begin with a `#` character at the beginning of the line and are terminated at the end of the line.

1.2. Whitespace

CRAWL is a free form language. Whitespace is ignored unless bounded by quotes (`"`) on either side.

1.3. Tokens

Tokens fall into five major categories: identifiers, keywords, constants, operators and separators.

1.3.1 Identifiers

Identifiers begin with a letter or an underscore (`_`) and are followed by any sequence of letters, digits or underscores. Two characters are considered equal if their ASCII values are equal. Two identifiers are considered equal if all their characters match.

1.3.2 Keywords

The following identifiers are reserved as keywords and using them in any CRAWL program as a regular identifier will result in an error:

```
int         float     edge      graph
string      while     if        else
foreach     in        sizeof    where
```

1.3.3 Constants

Constants provide the CRAWL programmer to conveniently initialise each of the supported primitives.

Integer Constants An Integer constant consists of an optional plus ('+') sign or minus ('-') sign followed by a string of decimal digits [0-9].

Floating point Constants A floating point constant is defined similar to the definition in the 'C' language, that is, it consists of an optional plus ('+') sign or minus ('-') sign followed by an integer part of one or more digits. This has to be followed by a decimal point or an exponent sign. The decimal point might be followed by more digits. The exponent is always followed by a positive or negative integer.

String Constants String constants consist of a sequence of characters surrounded by double quotes. The quotes are not considered part of the string and the '\' character is used to generate escape sequences. If the constant has to contain the '\' character literally, one has to use the '\\' sequence. The following escape sequences are recognised by CRAWL:

```
\n          newline
\t          The horizontal tab
```

1.3.4 Operators

Arithmetic operators The following arithmetic operators are supported by CRAWL: '+', '-', '*', '/' and '%'. Their meanings are respectively those of addition, subtraction, multiplication, division and remainder after division. The '-' operator can also be used as a unary operator to indicate the negativity of a number. The binary operators' operands must have have the same type. '+=', '-=', '*=', '/=' and '%=' are used to mean the same thing as they do in the 'C' language. The associativities of these operators also follow those of the 'C' language.

Collection operators CRAWL supports the '<>' operator to define collections. The '<' sign followed by a sequence of comma (',') separated values, terminated by a '>' sign is construed to be a collection. The '[]' operator is also supported to randomly access collection's elements. Hence, in order to access the i^{th} element of a collection C, one might use C[i].

Boolean operators The following boolean operations are supported in CRAWL: `<, <=, >, >=, ==, !` which respectively mean less than , less than or equal to , greater than , greater than or equal to , equal to and the negation of the boolean operation. In addition CRAWL supports the `&&` and `||` operators to mean boolean and and boolean or.

Sizeof The sizeof operator, when used with strings returns their length and with integers and floats returns the value. Behaviour of this operator when applied to Edges, Graphs and Collections is explained later.

Precedence The precedence and associativities of the operators in CRAWL are identical to their counterparts in 'C '. To override this, one might use parentheses (`()`).

1.3.5 Separators

`,` and `;` are used as separators in CRAWL. The `;` separator indicates end of executable statement.

1.3.6 Scope

The scope of an identifier begins at immediately after its definition and ends at the end of the block. Blocks are delimited using the '`{`' and '`}`' separators.

2 Types

2.1. Integers, Floats and Strings

Integers and Double precision floating point numbers are the only two numeric types supported by CRAWL. Strings are sequences of ASCII characters.

2.2. Collections

CRAWL supports a basic collection type, not very different from the array type in 'C'. It provides random access of elements and the elements are ordered in the collection according to the order that they were inserted. While creating collections, the size of the collection has to be specified. Collections support the `+` operator which allows the programmer to add a new element to the collection. The `sizeof` operator can be used to get the number of elements in the collection.

2.3. Edges

In order to describe the connections in a graph, CRAWL supports the edge data type. An edge is composed of three parameters, namely, the 2 nodes that it connects and a weight associated with the two edges. Edges literals are defined as a comma separated list of these 3 elements delimited by parentheses. For example, `(1, 2, "red")` defines an edge between nodes 1 and 2 with weight red . The`->` operator applied to an edge returns the node that it is incident on and the `<-` returns the node that it is incident from. On applying the `sizeof` operator to an edge, the weight is returned.

2.4. Graphs

Graphs in CRAWL are defined just as a pair, namely that of the number of nodes and a collection of edges. Graph literals are of the form `((num_of_nodes,`

`edge_collection))` . Graphs also support the binary `->` operator, which returns a collection of edges going out from the given node. For example, `g->3` returns a collection of edges going out from node number 3. Nodes are numbered from `0` to `n - 1`, where `n` is the number of nodes in the Graph. They also support the `<-` operator which returns a collection of edges incident on the specified node. Using the `sizeof` keyword, one can get the number of nodes in the Graph.

3 Control Structures Loops and branching constructs

foreach – in - where

`foreach` is a keyword used to iterate over a collection. Used in conjunction with the `in` operator, it allows for obtaining the next element from the collection being iterated over. The `where` keyword can be used to further limit the execution of this loop. For example, to find the sum of the elements in an integer collection where the value is greater than or equal to 2, one might write:

```
int num = 0;
int coll[10] = <1, 2, 3>;
foreach ( num in coll where num >= 2)
  sum += num;
```

while

`while` is a looping construct. It tests a condition at the beginning of each iteration and executes the statements in the loop if the condition evaluates to be `true`. The statements associated with the loop must be enclosed in braces.

if

`if` is a keyword used for conditional execution. The statements associated with the if conditional are executed once if the condition evaluates to be `true`.

else

When used in conjunction with the `if` conditional, the `else` block is executed whenever the condition in the `if` conditional evaluates to false.

4 Grammar

Program → *toplevel-stmts*

toplevel-stmts → *toplevel-stmt toplevel-stmts* | *toplevel-stmt*

toplevel-stmt → *function-definition* | *stmts* | *ε*

stmts →*stmt stmts* | *stmt*

stmt → *definition* | *executable-stmt* | *expression* | *ε*

executable-stmt → *non-semi-stmt* | *semi-stmt*

semi-stmt → *expression ';'* | *return-stmt ';'* | *ε ';'*

return-stmt → ***return*** *expression*

non-semi-stmt → *if-stmt* | *while-stmt* | *foreach-stmt*

if-stmt → ***if*** *'(' expression ')' '{' stmts '}'* | ***if*** *'(' expression ')' '{' stmts '}'* ***else*** *'{' stmts '}'*

while-stmt → ***while*** *'(' expression ')' '{' stmts '}'*

foreach-stmt → ***foreach*** *'(' <id>* ***in*** *expression ')' '{' stmts '}'*

| ***foreach*** *'(' <id>* ***in*** *expression* ***where*** *expression ')' '{' stmts '}'*

expression → *'(' expression ')'* | *assignment-expression* | *expression binop expression* | *unop-prefix expression* | *expression unop-postfix* | *function-call* |

graph-expression | *edge-expression* | *collection-expression* | *access-expression* | *<id>* | *<constant>*

binop → *'+'* | *'-'* | *'*'* | *'/'* | *'%'* | *'->'* | *'<-'* | *'=='* | *'<='* | *'>='* | *'<'* | *'>'* | *'!='* | *'&&'* | *'||'*

assignment-expression → *lvalue assignment-op expression*

lvalue → *<id>*

assignment-op → *'='* | *'+='* | *'-='* | *'*='* | *'/='* | *'%='*

unop-prefix → *'!'* | *'-'* | ***sizeof***

unop-postfix → *'->'* | *'<-'*

definition → *type <id> '=' expression ';'* | *type <id> '[' expression ']' = expression ';'* | *type <id> '[]' = expression ';'*

function-definition → *type <id> '(' parameters ')' '{' stmts '}'*

parameters → *parameter-list* | *ε*

parameter-list → *type <id> ',' parameter-list* | *type <id>*

function-call → *<id> '(' call-parameters ')'*

expressions → *ε* | *expression-list*

expression-list → *expression ',' expression-list* | *expression*

graph-expression → *'((' expression, expression '))'*

edge-expression → *'(' expression, expression, expression ')'* | *'(' expression, expression ')'*

collection-expression → *'<' expressions '>'*

access-expression → *<id> '[' expression-list ']'*

type → ***int*** | ***float*** | ***string*** | ***edge*** | ***graph***