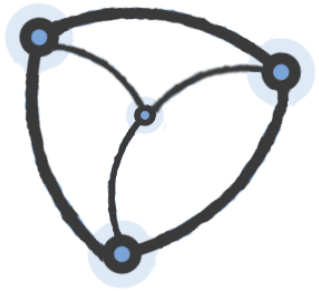

Graphene

Language Reference Manual

COMS W4115 Programming Languages and Translators

Adith Tekur (at2904)
Pooja Prakash(pk2451)
Sarah Panda (sp3206)
Sumiran Shah (srs2222)
Neha Rastogi (nr2477)

Team 8

March 26, 2014

Contents

1 Introduction

In this manual, we describe the various features of Graphene that can be used in defining and manipulating graphs. We present an overview of the lexical conventions, syntax of the language and the grammar representing Graphene.

2 Lexical Conventions

Like for every language, Graphene is composed of lexemes following patterns represented by tokens. Apart from whitespaces (that represent spaces, tabs and newlines) and comments, our language has the tokens - identifiers, keywords, constants, operators and separators.

2.1 Comments

Graphene provides support for both single and multi-line comments. Single-line comments begin with "//". Multi-line comments on the other hand bengin with "/*" and and with "*/". An example is as follows.

```
// This is a single-line comment

/* This is a...
 multi-line comment */
```

Note that one multi-line comment cannot be nested within another multi-line comment.

2.2 Identifiers

An identifier represents a unique entity within a program. Graphene follows the naming convention specified below.

1. An identifier must begin with a '$'
2. Following the '$', an identifier can be any sequence of alphabets, digits or underscores.

Graphene is case-sensitive.

2.3 Keywords

The following keywords convey special meaning to Graphene and hence should not be used elsewhere.

```
if                              while
else                            Graph
for                             Node
```

```
Edge                                    def
has                                     True
connects                                False
lambda
```

2.4 Reserved

The following characters are reserved for use in the grammar.

```
+           )           ==          &&          $
-           {           !=          ||          ->
*           }           <           !           <->
/           [           >           .           '
%           ]           >=          ,           "
(           =           <=          ;
```

3 Types

3.1 Basic Types

Graphene handles type dynamically. Basic data-types recognized and handled approriately are:

1. int - This is used to define integer numbers. Data of the form $[0-9]*$ will be handled as type *int*.
2. float - This is used to define floating-point numbers. $[0-9]+.[0-9]+$ translates to type $float$.
3. boolean - Values 'True' or 'False' will define *boolean* type.
4. string - This is used to represent strings (sequence of characters enclosed within "" or ' '). There is no char type. Instead, it is handled as a string made of 1 character. This eliminates the need to handle ' and " differently.

All the basic data-types - int, float, char and boolean can also be represented in Graphene as a string (sequence of characters enclosed within "").

The type of the variable is determined dynamically on the basis of the the characters enclosed and is handled accordingly.

The following is one such example.

```
integerVar = 70;                    //integerVar is handled as an integer
floatVar = 50.5;                    //floatVar is handled as a floating point
boolVar = True;                     //boolVar is handled as boolean.
stringVar = "graphene";             //stringVar is handled as a string
```

3.2 Derived Types

The following are the derived data-types supported by Graphene.

1. Graph - This is used used to define graphs.
2. Node - This is used to define a node in terms of user-defined properties representing the node. Graphene assigns a unique id to every node that is created. In other words, the id*s* of the nodes are globally unique.
3. Edge - This is used to define an edge in terms of user-defined properties representing the edge. Like in nodes, Graphene ensures that all edges have globally unique identifiers.
4. Arrays - Graphene supports arrays to represent multiple entities of the same type. An element within an array can be accessed using its index (Ex. arr[0] retrieves the first element of the array arr).

3.3 Escape Sequences

The following escape sequences are supported by Graphene.

`\n`	`newline`
`\t`	`tab`
`\\`	`backslash`
`\'`	`single quote`
`\"`	`double quote`

3.4 Boolean Constants

The boolean constants supported by Graphene are True and False.

4 Scope

4.1 Block Scope

Variables declared in a block are valid only for that block. A block can be a part of the function definition.

4.2 Function Scope

Variables declared in a function are valid only within that function. Once the function returns, the variables declared within the function are no longer visible.

4.3 Global Scope

Variables declared outside the functions are valid for all the functions in that program.

4.4 Lexical Scope

A variable's scope is based only on its position within the textual corpus of the code. Variable names in the same scope must be unique.

4.5 Linkage Scope

External library function or constants can be accessed through their identifiers and are shared by the entire program.

5 Grammar

```
function-dec:
    function
    lambda

lambda:
    'lambda' lambda-arguments compound-statement return-arguments ';'

function:
    'def' function-arguments $ID compound-statement return-arguments ';'

function-arguments:
    lambda-arguments

argument-list:
    'lambda' $ID
    'lambda' $ID ',' argument-set


lambda-arguments:
    '(' argument-set ')' '=>'

argument-set:
    type $ID
    type $ID ',' argument-set

return-arguments:
    '=>' '(' return-set ')'

            ε

return-set:
    $ID
    $ID ',' return-set

statement-list:
    statement
    statement statement-list
```

```
statement:
    expression-statement
    selection-statement
    iteration-statement
    class-method-expression

compound-statement:
    '{' '}'
    '{' statement-list '}'

expression-statement:
    ';'
    expression ';'

expression:
    assignment-expression;

assignment-expression:
    type $ID '=' type '.new()'
    type $ID '=' assignment-expression
    logical-OR-expression

logical-OR-expression:
    logical-AND-expression
    logical-OR-expression '' logical-AND-expression

logical-AND-expression:
    equality-expression
    logical-AND-expression '&&' equality-expression

equality-expression:
    relational-expression
    equality-expression '==' relational-expression
    equality-expression '!=' relational-expression

relational-expression:
    additive-expression
    relational-expression '<' additive-expression
    relational-expression '>' additive-expression
    relational-expression '<=' additive-expression
    relational-expression '>=' additive-expression

additive-expression:
    multiplicative-expression
    additive-expression '+' multiplicative-expression
    additive-expression '-' multiplicative expression
```

```
multiplicative-expression:
    primary-expression
    multiplicative-expression '*' primary-expression
    multiplicative-expression '/' primary-expression

type:
    primitive-type
    array-type

primitive-type:
    'Node'
    'Edge'
    'Graph'

array-type:
    primitive-type '[]'
    $ID '[]'

program:
    declaration-list

declaration-list:
    declaration-list declaration
    declaration

declaration:
    var-dec
    function-dec
    statement

statement:
    expression-stmt
    compound-stmt
    selection-stmt
    iteration-stmt
    return-stmt
    break-stmt

statement-list:
    statement-list statement
    statement

expression-stmt:
    expression ;
    ;
```

```
selection-stmt:
    if ( simple-expression ) statement
    if ( simple-expression ) statement else statement

iteration-stmt:
    while ( simple-expression ) statement
    foreach ( mutable in simple- expression ) statement

return-stmt:
    return ;
    return expression ;

break-stmt:
    break ;

compound-stmt:
    { local-declarations statement-list }

function-dec:
    type-specifier $id ( params ) statement
    $id ( params ) statement

params:
    param-list

            ε


param-list:
    param-list ; param-type-list
    param-type-list

param-type-list:
    type-specifier param-id-list

param-id-list:
    param-id-list , param-id
    param-id

param-id:
    $ID
    constant

var-dec:
    node
    graph
    edge
    list
```

```
node:
    $id "has" keylist

edge:
    $id "connects" $id ',' $id "has" key-list

keylist:
    $id
    $id "," keylist

graph:
    "Graph" $id "has" graph-type "{" edge-list "}"

graphtype:
    "u"
    "d"

edge-list:
    $id connector $id
    $id connector $id "," edge-list

connector:
    "->"
    "<->"

list:
    "List" $id

node:
    "Node" initializer

edge:
    "Edge" initializer

initializer:
    $id $id "=(" kv-list ")"

kv-list:
    kv
    kv "," kv-list

kv:
    $id
    $id ":" $id
```

```
multiple-return:
    "(" id-list ")" "=" call

id-list:
    $id
    $id "," id-list

expression:
    mutable "=" expression
    mutable "+=" expression
    mutable "-=" expression
    mutable "++"
    mutable "--"
    simple-expression

multiple-return-expr:
    mutable-list "=" call

simple-expression:
    simple-expression "or" and-expression
    and-expression

and-expression:
    and-expression "and" unary-rel-expression
    unary-rel-expression

unary-rel-expression:
    "not" unary-rel-expression
    rel-expression

rel-expression:
    sum-expression relop sum-expression
    sum-expression

relop:
    "<="
    "<"
    ">"
    ">="
    "=="
    "!="

sum-expression:
    sum-expression sumop term
    term
```

```
sumop:
    "+"
    "-"

term:
    term mulop unary-expression
    unary-expression

mulop:
    "*"
    "/"
    "%"

unary-expression:
    unaryop unary-expression
    factor

unaryop:
    "-"
    "*"

factor:
    immutable
    mutable

mutable:
    ID
    ID "[" expression "]"

immutable:
    ( expression )
    call
    constant

call:
    $id "." ucall
    ucall

ucall:
    $id args
    $id args "." ucall

args:
    "(" arg-list ")"
```

ε

```
arg-list:
    expression "," arg-list
    expression

constant:
    NUMCONST
    CHARCONST
    STRINGCONST
    True
    False
```

6 Grammar Explanation

6.1 Lambda Functions

```
lambda:
    'lambda' lambda-arguments compound-statement return-arguments ';'
```

A lambda function, defined by beginning with the keyword "lambda", can be used by a user to specify the logic on the basis of which functions utilizing lambdas can perform manipulations. For example, clustering functions will accept a lambda function as their first argument that specifies the heuristic for grouping of nodes into clusters. The keyword lambda identifies such a function. Like regular functions, lambda functions takes in zero or more comma separated list of arguments. A comma separated list of arguments can be returned. Built-in functions will specify the signature of the lambda they except. A mismatch will result in a compile error.

6.2 User-defined Functions

```
function:
     'def' function-arguments $ID compound-statement return-arguments ';'
```

User-defined functions, beginning with the keyword "def", are structurally similar to lamba functions except that the first argument may be a lambda function. Note that in the presence of a lambda function, it has to always be the first argument that is passed. Like in lambda functions, Graphene supports multiple return values.

A user-defined function can accept a lambda as an optional first argument. A lambda however, cannot accept another lambda as an argument.

6.3 Function Calls

```
call:
    $id "." ucall
    ucall
```

A function call consists of the name of the function followed by its arguments. The return value of this function can in turn call another function, providing for function chaining.

6.4 Handling Multiple Return Arguments

```
multiple-return:
    "(" id-list ")" "=" call
```

Graphene provides out-of-the-box support for multiple return arguments. A list of variables can then form the LHS for a function call that returns multiple values. In the event of mismatch of number of arguments, an error is generated.

6.5 Nodes and Edges

6.5.1 Declaration

```
node:
    $id "has" keylist
```

Graphene allows users to define properties that represent a node. These properties are typically key-value pairs. The user can define a universal set of keys for a particular node type. The "has" keyword implies that the node should contain values for at least the specified set of keys.

```
edge:
    $id "connects" $id ',' $id "has" key-list
```

The keyword "connects" is used to specify the the two nodes the edge is used to connect. In the above production rule, the first $id is the id of the edge. The two $ids following the "connects" keyword are the ids of the nodes. Also, like nodes, edges can also have key-value pairs and are specified similar to the key-value pairs of nodes.

6.5.2 Initialization

```
initializer:
    $id $id "=(" kv-list ")"
```

The above production rule is used to initialize both nodes and edges by assigning values to the keys of their corresponding types. The first $id represents the type of the node or the edge as defined in the declaration part. The second $id is the unique id of the node or the edge. kv-list represents the set of values to be assigned to the keys defined. Also, if a user wants to define additional keys that are not defined, (s)he may do so by using the format "key":"value" where key is the new key and value is its corresponding value. But values *must* be provided for the declared set of keys.

An example of the declaration and initialization of a node is as follows.

```
Person has name, school;          //Declaring a node-type Person
Person 2 = ("Jacob", "CU", "age":"20");
                                  //Assigning values to keys of
                                  //Person for node 2
```

6.6 Graphs

6.6.1 Declaration & Initialization

```
graph:
    "Graph" $id "has" graph-type "{" edge-list "}"
```

A variable of graph data type can be declared using the edge list and a keyword which represents whether the graph is directed or not. graph-type is "u" for undirected graphs and "d" for directed graphs. Edge list contains the list of all edges along with the connector which represents the direction of the edges. Connector "$->$" refers to a directed edge in a single direction and "$<->$" refers to directed edges in both directions. The latter also represents the edge in an undirected graph.