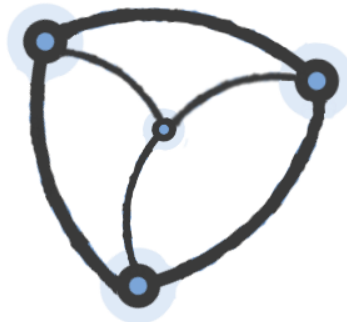

Graphene

Language Reference Manual

COMS W4115 Programming Languages and Translators

Adith Tekur (at2904), Neha Rastogi (nr2477), Sarah Panda (sp3206)
Sumiran Shah (srs2222), Pooja Prakash(pk2451)
Team 8

March 24, 2014

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 to 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 in 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          Edge
else        contains
for         connects
while       lambda
Graph       def
Node
```

2.4 Reserved

The following characters are reserved for use in the grammar.

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

2.5 Types

2.6 Basic Types

The following are other basic data-types supported by Graphene.

1. int - This is used to define integer numbers.
2. float - This is used to define floating-point numbers.
3. boolean - This is used to define boolean values and can take values True or False.
4. string - This is used to represent strings (sequence of characters enclosed within ””).
5. Graph - This is used used to define a graphs.
6. Node - This is used to define a node in terms of user-defined properties representing the node.
7. Edge - Like Node, this is used to define an edge in terms of user-defined properties representing the edge.

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.

```
integerVal = "70";              //integerVal is handled as an integer
floatVal = "50.5";              //floatVal is handled as a floating point
boolVal = "True";               //boolVal is handled as boolean.
charVal = "c";                  //charVal is handled as a character
stringVal = "graphene";         //stringVal is handled as a string
```

2.7 Derived Types

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).

2.8 Escape Sequences

The following escape sequences are supported by Graphene.

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

2.9 Boolean Constants

The boolean constants supported by Graphene are True and False.

3 Scope

3.1 Block Scope

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

3.2 Function Scope

Variables declared in a function are valid only within that function and it goes out of scope when the function returns.

3.3 Global Scope

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

3.4 Lexical Scope

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

3.5 Linkage Scope

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

4 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
```

argument list includes lambda as a possible first argument.
argument set does not.

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

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


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

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
    epsilon


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 "contains" keylist

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

keylist:
    $id
    $id "," keylist

graph:
```

```
    "Graph" $id "contains" 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 ")"
    epsilon

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

constant:
    NUMCONST
    CHARCONST
    STRINGCONST
    true
    false
```

5 Grammar Explanation

5.1 Lambda Functions

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

A lambda function can be used by a user to specify the logic on the basis of which certain functions 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.

5.2 User-defined Functions

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

User-defined functions are identified by the keyword "def". These functions 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 arguments that is passed. Like in lambda functions, Graphene supports multiple return values.

5.3 Nodes and Edges

5.3.1 Declaration

```
node:
    $id "contains" 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 "contains" keyword implicates that the node should contain values for at least the specified set of keys. An example is as follows.

```
edge:
    $id "connects" $id ',' $id "contains" 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 node, the edges can also have key-value pairs and is specified similar to that of nodes.

5.3.2 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 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.

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

```
Person contains name, school;          //Declaring a node Person
Person 2 = ("Jacob", "Columbia Univ"); //Assigning values to keys of
                                       //Person for node 2
```

5.4 Graphs

```
graph:
    "Graph" $id "contains" 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 can be "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 in case of directed graphs. Connector "$->$" refers to directed edge in a single direction while "$<->$" refers to directed edges in both directions.