The Green-Marl Language Specification

May 21, 2021
<table>
<thead>
<tr>
<th>Version</th>
<th>Modifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>initial concepts</td>
</tr>
<tr>
<td>0.2</td>
<td>collection types and their initialization, distinction of function and procedure, reduction assignment</td>
</tr>
<tr>
<td>0.3</td>
<td>a version that matches ASPLOS paper</td>
</tr>
<tr>
<td>0.4</td>
<td>properties of collections, collections of collections, map type, implicit graph binding, new set operations,</td>
</tr>
<tr>
<td>0.5</td>
<td>fix error in syntax definition, string type</td>
</tr>
<tr>
<td>0.6</td>
<td>change keywords to lower-case</td>
</tr>
<tr>
<td>0.6.1</td>
<td>minor improvements in writing</td>
</tr>
<tr>
<td>0.6.2</td>
<td>remove @-syntax for reduction (deferred assignment can still use @ syntax), read-only input arguments, remove local and uGraph, date/time type, edge/node/static built-ins, placeholder in group assignments</td>
</tr>
<tr>
<td>0.6.3</td>
<td>add NIL to grammar definition, fix typos, add combined variable declaration/initialization to grammar, add avg reduction,</td>
</tr>
<tr>
<td>0.7.0</td>
<td>vector type, small fixes</td>
</tr>
<tr>
<td>0.7.1</td>
<td>add date-literal to grammar definition, fix typos, add proc call to statements switched to dashes instead of underscores in grammar sorts, switched to grouping to not use same syntax as optionals in grammar</td>
</tr>
<tr>
<td>0.7.2</td>
<td>combined expr and bool-expr in grammar added node/edgeSequence alternative spelling for types added clear function to all collections changes numNodes/Edges to return long instead of int</td>
</tr>
<tr>
<td>0.7.3</td>
<td>print and println statements</td>
</tr>
<tr>
<td>0.7.4</td>
<td>BFS specific functions add pickRandom function add more node functions add escaping of placeholders</td>
</tr>
<tr>
<td>0.7.5</td>
<td>nodeFilter and edgeFilter types</td>
</tr>
<tr>
<td>0.7.6</td>
<td>add nested element access</td>
</tr>
<tr>
<td>0.7.7</td>
<td>remove date/time type, date-literal</td>
</tr>
<tr>
<td>Version</td>
<td>Authors</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0.1</td>
<td>Sungpack Hong (Stanford University)</td>
</tr>
<tr>
<td>0.2</td>
<td>Sungpack Hong (Stanford University)</td>
</tr>
<tr>
<td>0.3</td>
<td>Sungpack Hong (Stanford University)</td>
</tr>
<tr>
<td>0.4</td>
<td>Martin Sevenich (TU Munich / Oracle Labs)</td>
</tr>
<tr>
<td>0.5</td>
<td>Martin Sevenich (Oracle Labs)</td>
</tr>
<tr>
<td>0.6</td>
<td>Martin Sevenich (Oracle Labs)</td>
</tr>
<tr>
<td>0.6.1</td>
<td>Martin Sevenich (Oracle Labs)</td>
</tr>
<tr>
<td>0.6.2</td>
<td>Martin Sevenich (Oracle Labs)</td>
</tr>
<tr>
<td>0.6.3</td>
<td>Martin Sevenich (Oracle Labs)</td>
</tr>
<tr>
<td>0.7.0</td>
<td>Martin Sevenich (Oracle Labs)</td>
</tr>
<tr>
<td>0.7.1</td>
<td>Martin Sevenich (Oracle Labs), Jeff Smits (TU Delft / Oracle Labs)</td>
</tr>
<tr>
<td>0.7.2</td>
<td>Martin Sevenich (Oracle Labs), Jeff Smits (TU Delft / Oracle Labs)</td>
</tr>
<tr>
<td>0.7.3</td>
<td>Martin Sevenich (Oracle Labs)</td>
</tr>
<tr>
<td>0.7.4</td>
<td>Martin Sevenich (Oracle Labs)</td>
</tr>
<tr>
<td>0.7.5</td>
<td>Martin Sevenich (Oracle Labs), Daniel Lehmann (TU Darmstadt / Oracle Labs)</td>
</tr>
<tr>
<td>0.7.6</td>
<td>Martin Sevenich (Oracle Labs), Martijn Dwars (Oracle Labs)</td>
</tr>
</tbody>
</table>
## Contents

1 Introduction 7
   1.1 The Green-Marl Language and its Purpose 7
   1.2 A Glimpse of Green-Marl 8
   1.3 What Green-Marl is not 9
   1.4 Key Features of Green-Marl Language 10
      1.4.1 Features for Graph Data Processing 10
      1.4.2 Features for Parallel and Heterogeneous Execution 11
      1.4.3 Other Features 11

2 Syntax and Lexemes 12
   2.1 Lexemes 12
      2.1.1 Identifier (user-defined name) 12
      2.1.2 Literals 12
      2.1.3 Comments 12
   2.2 Green-Marl Syntax 12

3 Language Entities 17
   3.1 Procedures 17
      3.1.1 Entry Procedures 17
      3.1.2 Arguments and Return Values 17
      3.1.3 Aliases in Arguments 18
   3.2 Variables 18
      3.2.1 Scoping Rules 18
      3.2.2 Initial Values 19
      3.2.3 Iterators 19
      3.2.4 Semantic of Variable Assignment 20
   3.3 Statements and Expressions 20

4 Type System 21
   4.1 Overview 21
   4.2 Primitive Types 21
      4.2.1 Numeric Types 21
      4.2.2 Explicit Type Conversions and Coercions 22
      4.2.3 Boolean Type 22
      4.2.4 String Type 22
   4.3 Vector Types 23
      4.3.1 Overview 23
      4.3.2 Operator Semantics 24
      4.3.3 Restrictions 25
   4.4 Graph Type 25
      4.4.1 Range Words and Built-in Functions 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5 Node and Edge Types</td>
<td>27</td>
</tr>
<tr>
<td>4.5.1 Range Words and Built-in Functions</td>
<td>27</td>
</tr>
<tr>
<td>4.6 Property Types</td>
<td>29</td>
</tr>
<tr>
<td>4.7 Collection Types</td>
<td>31</td>
</tr>
<tr>
<td>4.7.1 Built-in Operations Collection Types</td>
<td>32</td>
</tr>
<tr>
<td>4.7.2 Iteration on Collection Types</td>
<td>34</td>
</tr>
<tr>
<td>4.8 Collections of Collections</td>
<td>36</td>
</tr>
<tr>
<td>4.9 Map Types</td>
<td>36</td>
</tr>
<tr>
<td>4.9.1 Built-in Operations</td>
<td>37</td>
</tr>
<tr>
<td>4.9.2 Behaviour in Parallel Semantics</td>
<td>37</td>
</tr>
<tr>
<td>4.10 Filter Types</td>
<td>38</td>
</tr>
<tr>
<td>4.11 Implicit Graph Binding</td>
<td>39</td>
</tr>
<tr>
<td>4.12 Variance in Local Method Calls</td>
<td>39</td>
</tr>
<tr>
<td>5 Parallel Execution and Consistency</td>
<td>41</td>
</tr>
<tr>
<td>5.1 Parallel Regions</td>
<td>41</td>
</tr>
<tr>
<td>5.1.1 Nested Parallelism</td>
<td>42</td>
</tr>
<tr>
<td>5.2 Memory Consistency in Green-Marl</td>
<td>42</td>
</tr>
<tr>
<td>5.2.1 Sequential Memory Consistency</td>
<td>42</td>
</tr>
<tr>
<td>5.2.2 Parallel Memory Consistency</td>
<td>43</td>
</tr>
<tr>
<td>5.2.3 Bulk-Synchronous Memory Consistency</td>
<td>46</td>
</tr>
<tr>
<td>5.3 Determinism Under Parallel Memory Consistency</td>
<td>46</td>
</tr>
<tr>
<td>5.3.1 Reductions</td>
<td>47</td>
</tr>
<tr>
<td>5.3.2 Deferred Assignments</td>
<td>50</td>
</tr>
<tr>
<td>5.3.3 Visibility in Nested Parallel Regions</td>
<td>50</td>
</tr>
<tr>
<td>5.4 Operations on Collection Types under Parallel Consistency</td>
<td>52</td>
</tr>
<tr>
<td>5.5 Implicit Parallel Context (Syntactic Sugar)</td>
<td>55</td>
</tr>
<tr>
<td>5.6 Notes for Green-Marl Compilers</td>
<td>55</td>
</tr>
<tr>
<td>5.6.1 Static Compiler Analysis for Data-Race Detection</td>
<td>55</td>
</tr>
<tr>
<td>5.6.2 Selective Parallel Execution of Parallel Regions</td>
<td>56</td>
</tr>
<tr>
<td>5.6.3 Implementation of Collection Types</td>
<td>58</td>
</tr>
<tr>
<td>6 Expressions and Statements: Details</td>
<td>59</td>
</tr>
<tr>
<td>6.1 Expressions</td>
<td>59</td>
</tr>
<tr>
<td>6.1.1 Operations on Numeric Types</td>
<td>59</td>
</tr>
<tr>
<td>6.1.2 Operations on Boolean Type</td>
<td>59</td>
</tr>
<tr>
<td>6.1.3 Conditional Operator</td>
<td>59</td>
</tr>
<tr>
<td>6.1.4 Reduction Operations</td>
<td>60</td>
</tr>
<tr>
<td>6.1.5 Operator Precedence</td>
<td>61</td>
</tr>
<tr>
<td>6.2 Assignment States</td>
<td>61</td>
</tr>
<tr>
<td>6.2.1 LHS and RHS</td>
<td>61</td>
</tr>
<tr>
<td>6.2.2 Reduction Assignments and Deferred Assignments</td>
<td>62</td>
</tr>
<tr>
<td>6.3 Control Statements</td>
<td>62</td>
</tr>
<tr>
<td>6.3.1 for and foreach Iteration</td>
<td>62</td>
</tr>
</tbody>
</table>
6.3.2 DFS and BFS Traversal ................................................. 63  
6.3.3 Other Control Structures ........................................... 66  
6.4 Built-in Functions and Operations ................................. 67  
6.4.1 `print` and `println` ................................................. 67

7 Interaction with Application Code ..................................... 70  
7.1 Overview ...................................................................... 70
7.2 Embedding Foreign Syntax ............................................ 70  
7.2.1 Foreign Types .......................................................... 71  
7.2.2 Foreign Expressions ................................................... 71
7.2.3 Foreign Statements .................................................... 72
7.2.4 Foreign Functions ....................................................... 72
7.2.5 Target Header Include ............................................... 72

8 Green-Marl Code Examples .................................................. 74

9 Ideas for Future Versions ..................................................... 77  
9.1 Multi-Set ..................................................................... 77
9.2 Multiple File Compilation ............................................... 77
9.3 Continue, Break ........................................................... 77
9.4 Named Continue Break ................................................ 77
9.5 Named While Loop ....................................................... 78
9.6 Acknowledged Conflicting Writes ................................. 78
9.7 Visibility Control for Collective Types ............................ 78
9.8 Neighborhood Marker ................................................... 78
9.9 Print Statement ............................................................. 79
9.10 Error Statement ........................................................... 79
1 Introduction

1.1 The Green-Marl Language and its Purpose

Green-Marl[^1] is a domain specific language (DSL) designed for easy development of graph-data processing programs. The language is also specially intended to exploit modern parallel computing environments such as multi-core and heterogeneous computers. The main idea is that let the user describe his/her algorithm concisely with the high-level language constructs of Green-Marl but let a compiler transform it into the equivalent, efficient low-level source codes for the target execution environment; for instance, CUDA code for GPU execution. In this approach, the final executable can be obtained by compiling the generated low-level source codes with an existing low-level compiler. The following figure illustrates this idea:

![Figure 1: Suggested Usage of the Green-Marl DSL](image)

The above approach provides the following benefits:

- Green-Marl enables the users to describe their own algorithm in an intuitive and concise way without considering low-level details of programming language or machine architecture. However, the language is designed such that a Green-Marl compiler can still generate equivalent but high-performing low-level code out of such high-level algorithmic description.

- A Green-Marl compiler can translate a single Green-Marl program description into the equivalents of various low-level programming languages, each targeting different...

[^1]: Green-Marl is a transliteration of two Korean words: 그린말 (depicted language)
(parallel or heterogeneous) computing environment. Green-Marl is designed in a way that such translation can be easily done.

- During the translation, Green-Marl compiler can apply high-level optimizations which may not be possible by conventional low-level language compilers. This is because the Green-Marl compiler has precise knowledge about the semantics of user’s algorithm, due to the power of high-level language constructs in Green-Marl.

### 1.2 A Glimpse of Green-Marl

The next example shows a Green-Marl program for ‘betweenness centrality’ computation algorithm[3].

**Listing 1: Betweenness Centrality computation in Green-Marl**

```green
procedure compute_bc (G: graph ; // G is an (Directed) Graph
   bc: nodeProperty<float>(G)){ // bc is float value associated with each
     // node of G
      G.bc = 0; // Initialize bc for every node in G
     foreach (r: G.nodes ) { // Outer loop: iterate every node in G
       nodeProperty<float>(G) delta; // Declare new node properties,
       nodeProperty<float>(G) sigma; // sigma and delta
        G.delta = 0; // Initialize sigma/delta for all nodes
        G.sigma = 0;
        r.sigma = 1; // Set sigma value for node r.
       inBFS (k: G.nodes from r) { // Traverse nodes in BFS order from r,
         k.s = sum (t: k.upNbrs ) { // and compute each node's sigma from
           t.sigma }; // BFS predecessors
       }
       inReverse (k!=r) { // Now traverse in reverse-BFS order
         k.delta = sum (t: k.downNbrs ) { // and compute delta from its
           k.sigma / t.sigma * (1 + t.delta) }; // BFS children
         k.bc += k.delta; // delta is accumulated into BC over r-loop.
       }
    }
}
```

Note that the original paper[3] described same algorithm in a very different manner; the original version explicitly uses queues and lists in the description. However, it is not clear how to parallelize the algorithm or how to port the algorithm for GPU from such description. To the contrary, the Green-Marl equivalent as shown in the previous code example uses high-level constructs which make the algorithm description very concise. For example, the Green-Marl description uses `inBFS` construct of which the semantics is to visit every node in breadth-first search order but to visit all the nodes having same BFS level in parallel. Consequently, the above Green-Marl description suggests a clear way to the compiler as well as to the readers about how the algorithm would be executed in parallel.

Section[8] provides more examples of graph algorithms written in Green-Marl.
1.3 What Green-Marl is not

Green-Marl does not intend to convert a naturally sequential algorithm into a parallel one in any automatic way. To the quite contrary, Green-Marl expects that the users are well aware of the parallel execution regions in their algorithms and those regions are accurately described with the language constructs. Nevertheless, the Green-Marl language constructs make it very intuitive for the user to describe the algorithm with exposing parallel regions, while the compiler can still exploit such parallelism efficiently out of the description.

Green-Marl is not a general-purpose language. It is a domain-specific language specifically tailored for graph data analysis. Therefore, it has a limited set of syntax, data structures and expressions which are suitable for graph data analysis but may not be so for other purposes. For example, Green-Marl would not be very convenient for, say, dynamic HTML page generation.

Nevertheless, a Green-Marl program can still interact with other programs that are written in other languages. Note that a Green-Marl program is expected to be compiled into another programming language, i.e. source-to-source translation. Therefore a user can isolate codes for graph analysis into the modules in his/her application, (re-)write only such modules in Green-Marl and compile into the target language, such as C++. The generated target language code can be linked with the rest of the application as if it is a normal hand-written target language code. The following figure illustrates this idea. Please refer to Section 7 for details about how Green-Marl program can interact with other part of the application.

![Figure 2: Interaction of Green-Marl Program with User Application](image)

---

A Green-Marl compiler is recommended to make its output fairly human-readable.
1.4 Key Features of Green-Marl Language

1.4.1 Features for Graph Data Processing

Since it is a DSL designed for graph data processing, Green-Marl has several language-level constructs for that purpose.

- **Built-in Data Types:** It contains built-in data types for graphs such as Graph, Node, and Edge. Therefore the Green-Marl compiler has precise knowledge about the semantics of operations on those structures, and exploits such knowledge in compiler optimization. In the following code, for example, the compiler can freely transform the code from the upper form into the lower form and vice versa. Note that depending on the parallelization strategy, one code can perform better than the other.

  Listing 2: Unoptimized Green-Marl Code

```plaintext
foreach (t: G.nodes) { // for every node t
    // Send t to every out-neighbor and reduce at there
    foreach (s: t.outNbrs) {
        s.bar += t.foo;
    }
}
```

Listing 3: Example Compiler Optimization in Green-Marl

```plaintext
foreach (t: G.nodes) { // for every node t
    // Read foo from its in-neighbors and reduce at t
    foreach (s: t.inNbrs) {
        t.bar += s.foo;
    }
}
```

- **Graph Iteration Methods:** Green-Marl provides several iteration methods that visit the whole nodes/edges of the graph in natural ways. All of these methods have similar syntactic form but different well-defined semantics. Examples are foreach and inBFS syntax shown in Listing. See Section 6.3 for more about graph iteration methods.

- **Node / Edge Properties:** In Green-Marl, the nodes and edges of a graph can be associated with arbitrary data: e.g. cost of a node or capacity of an edge. Such an associated data is referred as a node (edge) property. Green-Marl allows the user to declare and use properties dynamically just like normal variables, instead of defining them as fields of a class. See Section 4.6 for more about the properties.

3Suppose that the compiler has chosen to parallelize only the outer loop. Then, the reduction in the lower code can be implemented with normal read-and-write, instead of expensive atomic add instructions.
1.4.2 Features for Parallel and Heterogeneous Execution

In addition, Green-Marl provides several language constructs for the purpose of parallel and heterogeneous execution.

- **Parallel Constructs:** Green-Marl provides two kinds of parallel constructs; parallel iterations and reduction operation. Each parallel-iteration defines a parallel execution region, which can be nested in any depth. However, the compiler has freedom to apply parallelism on those iterations selectively. See Section 5 for detailed discussion.

- **Consistency Model:** Green-Marl features a weaker consistency model (Section 5.2.2) for the purpose of efficient parallelization and heterogeneous execution. Specifically, it does not assume sequential consistency inside parallel execution region at all - any writes inside a parallel execution is not guaranteed to be visible (or not visible) to the concurrent parallel executions. However, reductions (Section 5.3.1) can be used to enforce deterministic result under this consistency model. Also the language supports bulk-synchronous consistency model (Section 5.2.3).

1.4.3 Other Features

- **Statically Typed:** Green-Marl is a statically typed language. In other words, every type is identified by the compiler at compile time.
2 Syntax and Lexemes

The basic syntactic form of Green-Marl resembles that of C. The whole syntax rules can be found in the box at the end of this section. The semantic details of the syntax can be found in Section 6, while this section focuses on lexical rules.

The source code of Green-Marl uses the ASCII character set only. (A future extension may support Unicode, at least for string literals).

2.1 Lexemes

All lexemes of Green-Marl are composed of printable ASCII characters.

2.1.1 Identifier (user-defined name)

A valid identifier is a sequence of alphanumeric characters or underscores, except reserved words. An identifier cannot begin with number or underscore (except it is an underscore only - see Section 5.5). Uppercase and lowercase letters are differentiated.

2.1.2 Literals

- Integer literals: Green-Marl only uses decimal integer literals, i.e., there is no Hex or Oct numbers. A valid integer literal is either a decimal value in the valid integer value range, \( +\text{INF} \) or \( -\text{INF} \). (See discussion about Integer Type in Section 4.2.1 for the valid integer value range). Also note that \( +\text{INF} \) and \( -\text{INF} \) is a single lexeme, i.e., there is no space between +/- and INF.

- Floating Point Literals: floating point literals are <numbers>.<numbers> which can be optionally preceded by (+/-). In other words, it is like C but no exponential notation, nor trailing precision character. All the floating point literals are assumed to be typed as Double but it may be automatically coerced into Float or Int. (See discussion about Coercion)

- String Literals: string literals are ASCII characters inside double quotation marks. C escape character rules are applied as same. Unicode string is not (yet) supported in Green-Marl.

- Boolean Literals: a Boolean literal is either True or False.

2.1.3 Comments

Comment rules are like C: Any characters inside /* */ are considered as block comments. Double slash // makes any following characters into single line comments.

2.2 Green-Marl Syntax

Detailed explanations about syntactic elements can be found in Section 6.
• **Sorts** are printed italic.

• **Literals** are printed in fixed-width font and **reserved words** are highlighted in blue.

• 0 or more repetition is denoted with `{ }`, 1 or more repetition with `{ }+.

• Optional syntax is surrounded by `[ ]`.

• The `|` is used as a choice operator. Grouping choices is done with `( )`.

```plaintext
unit ::= { toplevel }+

toplevel ::= proc-def

proc-def ::= proc-header statement-block

proc-header ::= proc-keyword name ( [ arg-list ] [ ; outarg-list ] ) [ : return-type ]

proc-keyword ::= procedure | proc

arg-list ::= name : type-prop { , name : type-prop }*

outarg-list ::= arg-list

type-prop ::= type | property

type ::= prim-type | graph-type | vector-type | node-edge-type

| graph-collection-type | collection-type | filter-type | map-type

return-type ::= prim-type | node-edge-type

prim-type ::= int | long | float | double | bool

| string

vector-type ::= vect < type > [ expr ]

| iVect [ expr ]

| lVect [ expr ]

| fVect [ expr ]

| dVect [ expr ]

graph-type ::= graph | dGraph

dnode-edge-type ::= node [ ( name ) ]

| edge [ ( name ) ]

graph-collection-type ::= ( nodeSet | N_S ) [ ( name ) ]

| ( edgeSet | E_S ) [ ( name ) ]

| ( nodeOrder | N_O ) [ ( name ) ]

| ( edgeOrder | E_O ) [ ( name ) ]

| ( nodeSeq | nodeSequence | N_Q ) [ ( name ) ]

| ( edgeSeq | edgeSequence | E_Q ) [ ( name ) ]

filter-type ::= nodeFilter [ ( name ) ]

| edgeFilter [ ( name ) ]

collection-type ::= collection < graph-collection-type > [ ( name ) ]

map-type ::= map < prim-type , prim-type >

property ::= node-property < prim-type > [ ( name ) ]

| edge-property < prim-type > [ ( name ) ]
```

13
node-property < vector-type > [ ( name ) ]
edge-property < vector-type > [ ( name ) ]
node-property < graph-collection-type > [ ( name ) ]
edge-property < graph-collection-type > [ ( name ) ]

node-property ::= nodeProperty | nodeProp | N_P
edge-property ::= edgeProperty | edgeProp | E_P
name ::= identifier
name-comma-list ::= name { , name }*
statement ::= statement-block
              | decl-statement
              | normal-assignment
              | defer-assignment
              | reduction-assignment
              | for-iteration
              | foreach-iteration
              | bfs-iteration
              | dfs-iteration
              | if-then-else
              | while-statement
              | do-while-statement
              | built-in-call-statement
              | proc-call-statement
statement-block ::= { { statement }* }
decl-statement ::= type-prop name-comma-list ;
                 | type-prop name = expr ;
normal-assignment ::= lhs = rhs ;
defer-assignment ::= lhs <= rhs ;
reduction-assignment ::= rhs reduce-assign rhs ;
                       | rhs-opt-list reduce-assign2 rhs-opt-list ;
                       | rhs-opt-list2 reduce-assign2 rhs-opt-list2 ;
reduce-assign ::= += | *= | &= | |=
reduce-assign2 ::= max= | min=
lhs-opt-list ::= lhs [ < lhs { , lhs }* > ]
rhs-opt-list ::= rhs [ < rhs { , rhs }* > ]

lhs-opt-list2 ::= < lhs [ ; rhs { , rhs }* ] >
rhs-opt-list2 ::= < rhs [ ; rhs { , rhs }* ] >

lhs ::= name | name . name | element-access
rhs ::= expr
literal ::= integer-literal | float-literal | boolean-literal | string-literal
nil ::= NIL
expr ::= literal
       | nil
       | name | name . name
<table>
<thead>
<tr>
<th>built-in-call</th>
</tr>
</thead>
<tbody>
<tr>
<td>( expr )</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>- expr</td>
</tr>
<tr>
<td>! expr</td>
</tr>
<tr>
<td>reduce-expr</td>
</tr>
<tr>
<td>expr biop expr</td>
</tr>
<tr>
<td>expr ? expr : expr</td>
</tr>
<tr>
<td>element-access</td>
</tr>
<tr>
<td>biop ::= *</td>
</tr>
</tbody>
</table>
| comp-op ::= == | < | > | <= | >= | !=
| element-access ::= element-access [ expr ]
| ::= name |
| reduce-expr ::= reduce-type iterator-bound [ filter ] ( expr )
| reduct-type ::= sum | product | avg | max | min | any | all
| iterator-bound ::= ( name : name [ - ] , range-word1 )
| | ( name : name , range-word2 ) |
| filter ::= ( expr )
| range-word1 ::= nodes | edges | items
| range-word2 ::= nbrs | inNbrs | outNbrs | upNbrs |
| | downNbrs | inEdges |
| | outEdges | upEdges | downEdges |
| for-iteration ::= for iterator-bound [ filter ] statement
| foreach-iteration ::= foreach iterator-bound [ filter ] statement
| bfs-iteration ::= inBFS bfs-dfs-header statement [ bfs-reverse-iteration ]
| dfs-iteration ::= inDFS bfs-dfs-header statement [ dfs-reverse-iteration ]
| bfs-dfs-header ::= iterator-bound2 [ filter ] [ navigator ]
| bfs-reverse-iteration ::= inReverse [ filter ] statement
| dfs-reverse-iteration ::= inPost [ filter ] statement
| navigator ::= [ expr ]
| iterator-bound2 ::= ( name : name [ - ] . nodes ( from | ; ) name )
| if-then-else ::= if ( expr ) statement
| | if ( expr ) statement else statement |
| while-statement ::= while ( expr ) statement
| do-while-statement ::= do statement while ( expr ) ;
| built-in-call-statement ::= built-in-call ;
| built-in-call ::= built-in-name ( [ rhs-comma-list ] )
| | name . built-in-name ( [ rhs-comma-list ] )
| rhs-comma-list ::= rhs { , rhs }*
| proc-call-statement ::= proc-call ;
| proc-call ::= call-name ( [ rhs-comma-list ] [ ; lhs-comma-list ] )
| lhs-comma-list ::= lhs-ignore { , lhs-ignore }*
lhs-ignore ::= lhs | #
3 Language Entities

3.1 Procedures

3.1.1 Entry Procedures

The top-level entities of Green-Marl are procedure definitions. There are two types of procedures in Green-Marl: entry procedures and local procedures. An entry procedure begins with the keyword `procedure` or `proc`, while a local procedure with `local`.

A procedure is the entry point of Green-Marl program, branched from the user application. In other words, this is what is called by the application (see Section 7 for the interaction between user-application and Green-Marl program). The entry procedure should be called only in `virtually sequential context`. That is, at the time when the procedure is invoked by the application, there should be no other concurrent execution context which may potentially modify any values that are reachable through the arguments of the procedure. However, a Green-Marl compiler may further require `true sequential context`; i.e., there is actually no other concurrent execution and therefore the compiler can safely assume that all the hardware resource (e.g. GPU) are fully available.

A local procedure can be called from an entry procedure or from another local procedure. Local procedures can be called inside a parallel region (see Section 5.1) as well. A compiler must do inter-procedural analysis including such local procedure calls, when it validates parallel consistency semantics (see Section 5.2.2).

Listing 4: Green-Marl Procedure

```plaintext
procedure get_max_nbr_dist ( // Entry function
  G: graph, S: nodeSet(G), X: nodeProperty<int>(G), Y: nodeProperty<int>(G)) : int {
  int max_dist = 0;
  foreach (s: S.items) { // for each node s in set S
    foreach (n: s_nbrs) { // for each neighbors of node s
      // do in parallel
      max_dist max= s.X + n.Y; // compute distance and reduce by maximum
    }
  }
  return max_dist;
}
```

3.1.2 Arguments and Return Values

A Green-Marl procedure has two kinds of arguments: input arguments and output arguments. As the names suggest, input arguments are arguments that are given to the procedure, while output arguments produced by the procedure.

Basically, the output arguments are to support multi-valued return (see Listing 5 below). Therefore, users should assign values to the output arguments before the procedure returns. A compiler can give a warning if an output argument may not be defined before procedure return. In that case the value is undefined after the procedure is returned. An output argument can be read inside the procedure, once assigned. The read value is
undefined otherwise. When invoking a local procedure that has output parameters, all output arguments must be variable references. A compiler should produce an error if a local procedure is invoked where the argument that is provided as output parameter is not a variable reference.

Listing 5: Output arguments

```haskell
proc get_max_min (a, b: int ; m: int): int { // m is an output argument
    if (a < b) {
        m = a;
        return b;
    }
    else {
        m = b;
        return a;
    }
}
```

Input arguments are used for input only - they are immutable. That means input arguments can only be read but not overwritten or in the case of maps, properties etc. the content cannot be altered. Output arguments are used for input and output. They may be read and modified. The caller of a Green-Marl procedure is supposed to provide initial values for output arguments.

Return types can only be primitive types or `node` or `edge`. Complex types like properties or collections have to be returned as output arguments. The compiler will throw an error if not all code paths return a value if a return type is specified.

### 3.1.3 Aliases in Arguments

It is not allowed in Green-Marl to have aliases in procedure arguments. In other words, all the input arguments that are passed by references (i.e. graph, property, collection) should be distinct with each other.

Note that there is no way that a Green-Marl compiler can enforce this non-alias requirement at the call-site in the application code (in a different language) where a Green-Marl entry procedure is initially entered. Thus, it is up to the users who should make sure that there is no alias in the arguments when a Green-Marl entry function is invoked from application-side. Green-Marl compiler simply assumes each argument in the entry function is distinct.

### 3.2 Variables

#### 3.2.1 Scoping Rules

Every variable in Green-Marl program has a lexical, static scope. In addition, a variable name cannot be shadowed by another variable inside a nested scope.

Listing 6: Scope of Green-Marl variables

```haskell
// scope of argument names is limited to the procedure
procedure foo(G: graph, x,y: int, A: nodeProp<int>(G)) {
```
int z = 1; // variables are local
if (x > 0) {
    int k = y + 3;
    int z = 5; // Error! Cannot shadow the definition of z
}
foreach (n: G.nbrs) {
    int w = n.A + 3; // w is private to each instance of n-loop.
}

3.2.2 Initial Values

When a variable of a primitive type (Section 4.2) or node/edge type (Section 4.5) is declared, its initial value is undefined. Therefore such a variable should be initialized before it is used. The compiler may give a warning otherwise.

Similarly, when a property (Section 4.6) is declared, the property value at each node (edge) is undeclared.

However, when a collection-type (Section 4.7) variable is declared, an empty collection is automatically created.

Graph-type variable (Section 4.4) cannot be declared inside a Green-Marl procedure: it should be only passed through an input argument to the entry procedure.

Listing 7: Initial Values

```
proc foo (G: graph, n: node(G)) {
    int z; // prim-type variable z is declared
    int y = z; // value of z is undefined. A compiler may give a warning.
    nodeProp<int>(G) A; // property type variable A is declared
    Z = n.A; // value of A for each node in graph G is undefined
    nodeSet(G) S; // collection type variable S is declared.
    int s = S.size(); // s is 0. S is an empty-set
    graph G2; // Error! A graph cannot be declared inside.
}
```

3.2.3 Iterators

Iterator is a special kind of variable that is inherently defined by iteration sentences (Section 6.3.1) or reduction expressions (Section 6.1.4). The scope of an iterator is the body sentence (body expression) and filters attached to it (see Listing 8).

Iterators are read-only and thus cannot be written.

Listing 8: Iterator Scope Example

```
proc foo(G: graph; A: nodeProp<int>(G)) {
    node(G) m;
    // m is an iterator for the foreach statement
```
3.2.4 Semantic of Variable Assignment

In Green-Marl, assignment is defined as copying values of RHS into LHS; any expression can serve as RHS. LHS should be either a variable or a property access (Section 6.2.1). LHS and RHS should be type compatible. (See Section 4.2.2 for type compatibility and coercion rules.)

Graph type and Property type variables are read-only and cannot be assigned. See Section 5.5 for syntactic sugar for copying and initializing property values.

Assignment of collection or map types means to make a copy of the contents of RHS collection into LHS one. The original contents in LHS collection are lost (Section 4.7).

```
Listing 9: Assignment Example
proc foo (G: graph , A: nodeProp<int>(G), b: int , S: nodeSet(G)) {
    nodeProp<int>(G) B = A; // Error! Property itself is read-only.
    int c = b; // okay
    nodeSet(G) P = S; // make a copy of S.
}
```

3.3 Statements and Expressions

Unlike to C-based languages, Green-Marl strictly distinguishes statements from expressions: statements can have side-effects while expressions cannot. Side-effect means mutating the content of memory that is accessible after the sentence/expression is executed.

A reason for such restriction is to facilitate compiler’s optimization of expressions. For example, in the following code a compiler may transform the computation of z as in the commented line, based on the fact that the procedure is expensive to compute but is free of side-effects.

```
Listing 10: Example of possible optimization of a procedure call
proc example(b: int) {
    int z = some_expensive_function() * b;
    // == > The compiler may transform the above expression as below
    // int z = (b==0)? 0 : some_expensive_function() * b;
}
```
4 Type System

4.1 Overview

Green-Marl has a very simple type system. All the types in Green-Marl are intrinsic; i.e., there is no notion of user defined type. There is no notion of inheritance, either. The compiler can therefore simply determine the type of each variable or expression, statically.

4.2 Primitive Types

4.2.1 Numeric Types

There are four subtypes in numeric: int, long, float, and double.

Green-Marl ints are integer values of range [-2,147,483,647  2,147,483,646]. Please note that valid range is composed of all 4 byte binary numbers but 0x80000000 and 0x7FFFFFFF.

Similarly, Green-Marl longs are integer values of range [-9,223,372,036,854,775,807  9,223,372,036,854,775,806], Again, the valid range is composed of all 8 byte binary numbers but 0xFFFFFFFFFFFFFFFF and 0x7FFFFFFFFFFFFFFF.

Floating points (float and double) in Green-Marl are intentionally defined loosely for the purpose of wider portability. The only enforcement in Green-Marl is that float uses at least 4 bytes and that double uses at least 8 bytes for data representation, which roughly corresponds to single precision and double precision format in IEEE 754 standard.

A Green-Marl compiler should map each type into a floating point type of the target programming language with matching size: e.g. float/double in C or Cuda.

Section 6.1.1 covers operators defined for numeric-type values.

Two special numeric values are +INF and -INF which are compatible to any numeric type. However, it gives an undefined result, doing any numeric operation other than comparison to +INF/-INF (Code 11).

Listing 11: Example use of +INF

```plaintext
int z = +INF;
int x = 0;
bool b = (x < z); //==> Result is true
bool c = (x < (z+1)); //==> Result is undefined because z+1 is undefined
int w = -1 * +INF; //==> Result is undefined. -INF != -1 * +INF
// INF is useful, when one gets the minimum of certain values.
// Suppose G: graph, A: nodeProp<int>(G)
int y = +INF
foreach (n: G.nodes) {
    y min = n.A; // compute minimum among n.A
}
```

4 The extreme values are reserved for +/- INF.
4.2.2 Explicit Type Conversions and Coercions

When a numeric operator is applied to two numeric expressions, both expressions should have an exactly same type. Similarly when a numeric-type RHS is assigned into a numeric-type LHS, the two operands should have an exactly same type.

The user, however, can explicitly change the type of a numeric expression; Green-Marl syntax of explicit for type conversion is similar to that of C. Also, in certain cases, the compiler inserts type conversion automatically in place of the user, i.e., coercion occurs. Table 2 summarizes coercion rules in Green-Marl. See Listing 12 for example.

<table>
<thead>
<tr>
<th>RHS</th>
<th>LHS</th>
<th>Coercion</th>
<th>Compiler Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>long</td>
<td>int → long</td>
<td>implicit conversion</td>
</tr>
<tr>
<td>int</td>
<td>float</td>
<td>int → float</td>
<td>implicit conversion with warning</td>
</tr>
<tr>
<td>long</td>
<td>float</td>
<td>long → float</td>
<td>implicit conversion with warning</td>
</tr>
<tr>
<td>long</td>
<td>double</td>
<td>long → double</td>
<td>implicit conversion with warning</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
<td>float → double</td>
<td>implicit conversion</td>
</tr>
</tbody>
</table>

Listing 12: Coercion and Explicit Type Conversion Example

```java
int i = 3;
float f = 0.1;
long l = 10;
double d = 0.2;

long l2 = i; // coercion (Int -> Long)
int i2 = l;  // type error
int i3 = (int) l; // Okay, explicit type conversion
float f2 = i; // coercion (Int -> Float); compiler gives a warning
float f3 = (float) i; // No warning.
```

4.2.3 Boolean Type

Boolean type variables can have only one of two values: true and false.

4.2.4 String Type

String type variables contain text and are immutable. String literals are a sequence of ASCII characters surrounded by double quotation marks. Control sequences are allowed and are handled just like C strings. Double quotation marks and backslashes have to be escaped with a backslash.

Listing 13: String Escaping Example

```java
string s1 = "Hello World"; // Hello World
string s2 = "The world says \"Hello\""; // The world says "Hello"
string s3 = "\\\"\\\"\"; // "
```

22
Strings support the normal comparison operators (==, <, >, <=, >=), where \( s1 == s2 \) is equal to \( s1.equals(s2, true) \). The comparison operators use lexicographical ordering. See Listing 14 for a few examples. Strings provide several built-in methods:

<table>
<thead>
<tr>
<th>Signature</th>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>int</td>
<td>returns the number of characters in the string.</td>
</tr>
<tr>
<td>contains(string)</td>
<td>bool</td>
<td>returns true if the argument string is a substring (or equal) to the string, false otherwise.</td>
</tr>
<tr>
<td>beginsWith(string)</td>
<td>bool</td>
<td>returns true if the string starts with the argument string, false otherwise.</td>
</tr>
<tr>
<td>endsWith(string)</td>
<td>bool</td>
<td>returns true if the string ends with the argument string, false otherwise.</td>
</tr>
<tr>
<td>equals(string, bool)</td>
<td>bool</td>
<td>returns true, if the string has the same length as the argument string and if all characters are equal, false otherwise. The comparison is case sensitive if the bool argument is true, otherwise it is case-insensitive.</td>
</tr>
</tbody>
</table>

Listing 14: String Type Example

```java
// declaration & initialization
String s1 = "Hello World";
String s2 = "Green-Marl";

// assignment
s1 = s2;

// built-ins
int l = s1.length();
bool b1 = s1.beginsWith("Hello");
bool b2 = s2.contains("Green");

// comparison
bool b3 = s1 < s2;
bool b4 = s2 == s;
```

4.3 Vector Types
4.3.1 Overview
The Green-Marl type system supports vectors (i.e. one-dimensional matrices) for numeric types. A vector does have a fixed size, called dimension. The dimension specifies the amount of entries in the vector. Each entry can be accessed using an index in the range \([0, \text{dimension})\). Using an index outside of this range will result in undefined behavior. Note that the type of a vector variable is defined by the combination of the content
type as well as the dimension - two vectors with the same content type but different dimensions are considered to be of different types.

### Table 4: Vector Types

<table>
<thead>
<tr>
<th>Content Type</th>
<th>Vector Type</th>
<th>Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>vect&lt;int&gt;</td>
<td>iVect</td>
</tr>
<tr>
<td>long</td>
<td>vect&lt;long&gt;</td>
<td>lVect</td>
</tr>
<tr>
<td>float</td>
<td>vect&lt;float&gt;</td>
<td>fVect</td>
</tr>
<tr>
<td>double</td>
<td>vect&lt;double&gt;</td>
<td>dVect</td>
</tr>
</tbody>
</table>

#### Listing 15: Vector Example

```cpp
// declare new double vector of dimension 3
vect<double>[3] v1;

// set a single value in v1
v1[2] = 0.73;

// set all values in v1 to 0.5
v1 = 0.5;
```

### 4.3.2 Operator Semantics

The following operators are defined for vector types: + - * /

Note that some operators are only defined for two vectors, while others are only defined for a vector and a numeric scalar, or both. Operations involving two vectors are only defined if both vectors have the same dimension. A vector returned by such an operation will have the same dimension as the/both input vectors. See Table 5 for all operators and types and their semantics.

### Table 5: Arithmetic Operators on Vectors
<table>
<thead>
<tr>
<th>Operator</th>
<th>Types</th>
<th>Result Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>vector &amp; vector</td>
<td>vector</td>
<td>( x = v + w \iff x[i] = v[i] + w[i], 0 \leq i &lt; \text{dim}(v) )</td>
</tr>
<tr>
<td>-</td>
<td>vector &amp; vector</td>
<td>vector</td>
<td>( x = v - w \iff x[i] = v[i] - w[i], 0 \leq i &lt; \text{dim}(v) )</td>
</tr>
<tr>
<td>*</td>
<td>vector &amp; numeric</td>
<td>vector</td>
<td>( x = v \times s \iff x[i] = s \times v[i], 0 \leq i &lt; \text{dim}(v) )</td>
</tr>
<tr>
<td>/</td>
<td>vector &amp; numeric</td>
<td>vector</td>
<td>( x = v / s \iff x[i] = v[i] / s, 0 \leq i &lt; \text{dim}(v) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>numeric &amp; vector</td>
<td>vector</td>
</tr>
</tbody>
</table>

Testing for equality (==) and inequality (!=) is defined on vector types. Note that testing for smaller(-equal) or greater(-equal) is not defined for vector types. Comparing two vectors is only defined if both vectors have the same dimension. Table 6 defines the semantics of the comparison operators.

**Table 6: Comparison Operators on Vectors**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>( v == w \iff \forall i; 0 \leq i &lt; \deg(v) : v[i] == w[i] )</td>
</tr>
<tr>
<td>!=</td>
<td>( v! = w \iff \neg(v == w) )</td>
</tr>
</tbody>
</table>

### 4.3.3 Restrictions

Green-Marl restricts the expressions that can be used to specify the dimension of a vector variable. Only expressions of type `int` are allowed and the value must be constant during the runtime of the procedure.

### 4.4 Graph Type

Green-Marl has a type for directed graphs: `dGraph`. `graph` is a synonym for `dGraph`. There is no separate type for multi-graph: any graph is assumed to be a multi-graph. The following figure illustrates the difference between directed graph, undirected graph and a multi-graph.

Every Graph in Green-Marl is read-only. In other words, the graph is immutable. It is also not allowed to declare a graph-type variable inside a Green-Marl procedure. Instead, graphs should be handed as input arguments to the entry procedure.

**Listing 16: Example Use of Graph Type**

```cpp
procedure foo(G1, G2: graph) { // graph can be an input argument
graph G3; // Error - graph cannot be defined
```
In current usage model of Green-Marl, creation and modification of the graph itself must be done outside Green-Marl language - i.e. they should be done in the application side with the target language. Such a design decision is based on the following rationale:

- Green-Marl is designed for the analysis of graph data, rather than graph manipulation.

- In many applications, graph modification is less frequent than graph analysis. Also in many cases, analysis routines work on a specific snapshot of the graph so that it is safe to assume the graph is immutable during the analysis. (i.e. can work on a snapshot)

- Green-Marl is designed for portable execution, while some graph processing systems (such as Pregel [6]) provides very unique methods for loading graph data. Thus, it would be more convenient for users to work directly on the system API, in such cases.

Nevertheless, it is our future plan to develop a sister language (or a language extension) to Green-Marl, which is more apt to creating and modifying graph instances but less optimized for graph analysis.

### 4.4.1 Range Words and Built-in Functions

Range word is a syntax that delineates the range of for-iteration and foreach (Section 6.3.1) iteration. The following table summarizes range words available to graph types.

<table>
<thead>
<tr>
<th>Source Type</th>
<th>Range Word</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>dGraph</td>
<td>nodes</td>
<td>every node in the graph</td>
</tr>
<tr>
<td></td>
<td>edges</td>
<td>every edge in the graph</td>
</tr>
</tbody>
</table>

The below table summarizes built-in functions (Table 8) for graph types. A compiler implementation may add other built-in functions.

<table>
<thead>
<tr>
<th>Source Type</th>
<th>Signature</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>dGraph</td>
<td>numNodes():long</td>
<td>number of nodes in the graph</td>
</tr>
<tr>
<td></td>
<td>numEdges():long</td>
<td>number of nodes in the graph</td>
</tr>
<tr>
<td></td>
<td>pickRandom():node</td>
<td>returns a random node from the graph, or NIL if the graph does not have nodes</td>
</tr>
</tbody>
</table>
4.5 Node and Edge Types

`node` and `edge` are data types for fundamental components of graphs. Note that `node` type and `edge` type are not compatible to integer types.

Nodes and edges can be compared for equality (`==, !=`) and for ordering (`<, >, <=, >=`), though the ordering is undefined. In Green-Marl, `node` and `edge` type variables are always bound to a graph instance. In the following code, for example, node `n` belongs to graph `G1` and `m` to `G2`. Therefore it is an error to compare those two nodes, because they belong to different graphs. If only one graph is present in a procedure, the explicit binding can be omitted since all nodes are implicitly bound to this one graph. For more details can be found in Section 4.11.

Listing 17: Example Use of Node Type

```c
// graph G1, G2
// n is a node of G1; m is a node of G2.
proc foo(G1, G2: graph, n: node(G1), m: node(G2)) {
    bool b = (n==m); // error - n and m belongs to different graphs
}
```

4.5.1 Range Words and Built-in Functions

Range word is a syntax that delineates the range of `for`-iteration and `foreach` iteration (Section 6.3.1). Table 8 summarizes range words available to `node` type and `edge` type; the meaning of each range word is illustrated in Table 7. There are few things to be noticed:

- `in/outNbrs` are for directed graphs; `inNbrs` requires use of reverse-edges in a directed graph. `nbrs` is a synonym to `outNbrs` when used for directed graphs.
- `in/outNbrs` gives a multi-set of neighborhood nodes because a graph is assumed to be a multi-graph.
- `up/downNbrs` are only defined during BFS traversal (Section 6.3.2). `upNbrs` are (in-) `Nbrs` that are closer to the BFS root node than the current node, in hop distance. Conversely, `downNbrs` are (out-) `Nbrs` that are farther from the BFS root node. Note that there are neighborhood nodes that do not belong to either `upNbrs` or `downNbrs`: e.g. node `m` in Figure 3.

Table 9: Range words of Node Type

---

5 A compiler can still represent nodes and edges as long or integer in implementation.
Figure 3: The Illustration of Semantics of Range Words of Node Type

OutNbrs = {a, b, b}  
OutEdges = {i, j, k}

InNbrs = {a, b, b}  
InEdges = {i, j, k}

Nbrs = {a, b, b}  
Edges = {i, j, k}

The following table summarizes the built-in functions (Section 6.4) of the node and edge type. A compiler implementation may add other built-in functions.

<table>
<thead>
<tr>
<th>Source Type</th>
<th>Range Word</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>nbs, outNbrs</td>
<td>nbs is a synonym for outNbrs</td>
</tr>
<tr>
<td></td>
<td>edges, outEdges</td>
<td>edges is a synonym for outEdges</td>
</tr>
<tr>
<td></td>
<td>inNbrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inEdges</td>
<td></td>
</tr>
<tr>
<td></td>
<td>upNbrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>upEdges</td>
<td>only available during BFS traverse</td>
</tr>
<tr>
<td></td>
<td>downNbrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>downEdges</td>
<td></td>
</tr>
</tbody>
</table>

The following table summarizes the built-in functions (Section 6.4) of the node and edge type. A compiler implementation may add other built-in functions.

Table 10: Built-in Functions of Node Types
<table>
<thead>
<tr>
<th>Source Type</th>
<th>Signature</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>numNbrs() : long</td>
<td>size of the out-neighbor multi-set</td>
</tr>
<tr>
<td>node</td>
<td>numOutNbrs() : long</td>
<td>degree() : long outDegree() : long</td>
</tr>
<tr>
<td>node</td>
<td>numInNbrs() : long</td>
<td>size of the in-neighbor multi-set</td>
</tr>
<tr>
<td></td>
<td>inDegree() : long</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hasEdgeTo(node) : bool</td>
<td>true if the node has an edge to the argument node</td>
</tr>
<tr>
<td></td>
<td>hasEdgeFrom(node) : bool</td>
<td>true if the argument node has an edge to the node</td>
</tr>
<tr>
<td></td>
<td>pickRandomOutNbr() : node</td>
<td>returns a random out-neighbor. If the node has no out-neighbors, the behavior is undefined.</td>
</tr>
<tr>
<td></td>
<td>pickRandomInNbr() : node</td>
<td>returns a random in-neighbor. If the node has no in-neighbors, the behavior is undefined.</td>
</tr>
<tr>
<td></td>
<td>pickRandomInOutNbr() : node</td>
<td>returns a random out- or in-neighbor. If the node has neither out- nor in-neighbors, the behavior is undefined.</td>
</tr>
<tr>
<td>edge</td>
<td>fromNode() : node</td>
<td>source node of the directed edge</td>
</tr>
<tr>
<td></td>
<td>toNode() : node</td>
<td>destination node of the directed edge</td>
</tr>
<tr>
<td>neighbor iterator</td>
<td>fromEdge() : edge</td>
<td>gives an edge that reaches to the corresponding iterator</td>
</tr>
<tr>
<td></td>
<td>toEdge() : edge</td>
<td></td>
</tr>
</tbody>
</table>

NIL is a special value for node/edge type. NIL can be assigned to any node/edge type variable, regardless of graph binding. The result of accessing node/edge property from the NIL node/edge is undefined.

Listing 18: NIL value for node/edge type

```
graph G1, G2;
nodeProp<int>(G) A;
node(G1) n;
node(G2) m;
    n = NIL; // okay
    m = NIL; // okay
    n = m;    // Error
int x = n.A; // undefined
```

### 4.6 Property Types

Although Green-Marl does not support user-defined types, the user can still associate nodes and edges with any number of primitive-, vector- or collection-type data; such an association can be made even dynamically. In Green-Marl, such data associated with nodes (edges) are referred as node (edge) property. At the moment, properties cannot contain map types.
• property declaration syntax is as follows:
  `nodeProp<target_type>(graph_name) property_name`

• properties can be declared just like normal variable and has static scope (Section 3.2.1); see 4.6.

• properties access syntax is as follows: `source_name.property_name`

• in above syntax, the source must be bound to the same graph that the property belongs to; see 4.6.

• every element of a property declaration is initialized with a default value based on the type of its member.

Listing 19: Static Scoping Rule of Property Declaration

```plaintext
procedure foo (G: graph, A: nodeProp<int>(G)) {
    nodeProp<int>(G) B; // this property is alive only inside the procedure
    foreach (n: G.nodes) {
        nodeProp<int>(G) C; // this property is private to each instance of n-loop
        foreach (k: G.nodes) {
            k.C = k.A * n.A + 1; // property access
        }
        n.B = max(k: G.nodes) (k.C > 0){k.A}
    }
}
```

Listing 20: Type Rule of Property Declaration

```plaintext
procedure foo (G1, G2: graph; A: nodeProp<int>(G1)) {
    node(G2) a;
    a.A = 0; // error - a is not a node belong to G1.
    edge(G1) b;
    b.A = 0; // error - b is an edge of G1, not a node.
    node(G1) c;
    bool z = c.A; // error - target type of c.A is int; cannot be assigned to bool
}
```

Followings are additional notes for property types.

• Being syntax sugars, `nodeProp` and `N_P` is a synonym to `nodeProperty`; `edgeProp` and `E_P` to `edgeProperty`.

• When a property is used as a procedure argument, it is passed by reference. Aliases between references are not allowed. (See Section 3.1.2 and 3.1.3)

The following applies to collections as types of properties:

• When being created, each element of the property is set to an empty collection
<table>
<thead>
<tr>
<th>Collection Name</th>
<th>Ordered-ness</th>
<th>Uniqueness</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodeSet</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>edgeSet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nodeOrder</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>edgeOrder</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nodeSeq</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>edgeSeq</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Collections in Green-Marl

- For the elements of the property apply copy-semantics: When assigning values to an entry, a copy of the collection is stored instead of a reference.

- When reading an entry from the property, a copy of the collection is returned.

### 4.7 Collection Types

There are six collection types in Green-Marl. More specifically, three collections are defined for nodes and the other three for edges. Collections in Green-Marl are summarized in Table 4. Noticeably, the difference between each collection type comes from orderedness and uniqueness. For example, elements in a set are un-ordered and unique, while elements in a sequence are ordered and can be repeated. Collections that are given as input arguments in a procedure are immutable, i.e. no values can be added or removed.

Following are notes for collection types.

- Collections are bound to a graph, similar to Property Types. (Section 4.6)
- When a collection is first declared, it becomes automatically an empty collection. In other words, there required no separate initialization. (Section 3.2.2)
- When a collection is used as a procedure argument, it is passed by reference. Aliases between references are not allowed. (Section 3.1.2 and 3.1.3)
- When a collection is used as an argument for a collection-built-in, both collections have to be bound to the same graph instance.
- The semantics of an assignment in collection type is to create a copy.
- Comparison operator between collections is not defined.

The following code example shows declaration and assignment rules of collection types.

Listing 21: Declaration and Assignment of Collection Types

```plaintext
// Assume a, b, c are distinct node. (They don’t have to be by the syntax)
procedure foo(G1, G2: graph, a, b, c: node(G1), d: node(G2)) {
  nodeSet(G1) S1; // S1 is a node set of graph G1. S is an empty set here.
```
N_S(G1) S2;  // N_S is a synonym to nodeSet
N_S(G2) S3;

S1.add(a);  // S1 becomes {a}
S1.add(a);  // S1 still {a} (a is repeated)
S1.add(d);  // Error - d does not belong to G1

S3 = S1;  // Error - S3 does not belong to G1
S2 = S1;  // Copy S2 into S1
S1.add(b);  // S1 becomes {a, b}
bool cond = (S2.has(b));  // cond is False;

// Order and Sequence
nodeOrder(G1) O1;
nodeSeq(G1) Q1;

O1.push(a);
O1.push(b);
O1.push(c);  // O1 becomes {a, b, c}
O1.push(c);  // O1 still is {a, b, c}

Q1.push(a);
Q1.push(b);
Q1.push(c);
Q1.push(c);  // Q1 is {a, b, c, c}

4.7.1 Built-in Operations Collection Types

Table 5 and Table 6 summarize built-in operations of Collection Types in Green-Marl. Note that these operators are not mathematical operators that create a new set. Rather they modify the current set.

Notice that in Table 5 and Table 6 there is a column named ‘Type of Operation’, which denotes the type of the operation. For example, has() and size() are classified as check operations, while push() and add() are classified as append operations. These classifications determine which operations can be applied together when operations are applied under parallel consistency. See Section 5.4 for further discussion.

The following code example shows how ordering is preserved by built-in operations.

Listing 22: Ordered-ness of Elements After Operations on an Order

// Assume a, b, c, d are distinct node. (They don’t have to, by the syntax)
procedure foo(G: graph, a, b, c, d: node(G)) {

    // Order and Sequence
    nodeOrder(G) O1, O2, O3;

    O1.push(a);  // O1 becomes {a}
    O1.push(a);  // O1 is still {a}
    O1.push(b);  // O1 becomes {a, b}
    O1.pushFront(c);  // O1 becomes {c, a, b}

    node(G) x = O1.pop();  // O1 becomes {a, b}, x is c
}

32
<table>
<thead>
<tr>
<th>Operation</th>
<th>Type of Operation</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>has(node) : bool</code></td>
<td>check</td>
<td>returns true if the set contains the node</td>
</tr>
<tr>
<td><code>size() : int</code></td>
<td>check</td>
<td>returns the number of elements in the current set</td>
</tr>
<tr>
<td><code>add(node)</code></td>
<td>append</td>
<td>add a node to the set</td>
</tr>
<tr>
<td><code>add(N_S)</code></td>
<td>append</td>
<td>add all the nodes from the argument set to the target set</td>
</tr>
<tr>
<td><code>remove(node)</code></td>
<td>remove</td>
<td>remove the given node from the set, if the node is in the current set</td>
</tr>
<tr>
<td><code>remove(N_S)</code></td>
<td>remove</td>
<td>remove all the nodes in the argument set from the current set</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>remove</td>
<td>remove all nodes in the current set</td>
</tr>
<tr>
<td><code>addAll(N_S)</code></td>
<td>append</td>
<td>adds all nodes in the argument set to the target set</td>
</tr>
<tr>
<td><code>retainOnly(N_S)</code></td>
<td>remove</td>
<td>removes all nodes from the current set that are not in the argument set</td>
</tr>
<tr>
<td><code>removeAll(N_S)</code></td>
<td>remove</td>
<td>removes all nodes from the current set that are in the argument set</td>
</tr>
<tr>
<td><code>isSubsetOf(N_S) : bool</code></td>
<td>check</td>
<td>returns true if every element of the current set is also element of the argument set</td>
</tr>
</tbody>
</table>

Figure 5: Built-in Operations of nodeSet (edgeSet)
<table>
<thead>
<tr>
<th>Operation</th>
<th>Class</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>has(node) : bool</td>
<td>check</td>
<td>returns true if the collection contains the node</td>
</tr>
<tr>
<td>size() : int</td>
<td>check</td>
<td>returns the number of elements in the current collection</td>
</tr>
<tr>
<td>front() : node</td>
<td>check</td>
<td>returns the front node of the order</td>
</tr>
<tr>
<td>back() : node</td>
<td>check</td>
<td>returns the back node of the order</td>
</tr>
<tr>
<td>pushBack(node)</td>
<td>append</td>
<td>add the node at the back of the order. push is a synonym for pushBack</td>
</tr>
<tr>
<td>push(node)</td>
<td>append</td>
<td>add all elements in argument collection at the back of the order</td>
</tr>
<tr>
<td>pushFront(node)</td>
<td>append</td>
<td>add the argument node to the front of the order</td>
</tr>
<tr>
<td>pushFront(N_O)</td>
<td>append</td>
<td>add all elements in argument collection at the front of the order</td>
</tr>
<tr>
<td>popFront() : node</td>
<td>remove</td>
<td>removes a node at the front of the order</td>
</tr>
<tr>
<td>popBack() : node</td>
<td>remove</td>
<td>removes a node at the back of the order. pop is a synonym for popBack</td>
</tr>
<tr>
<td>pop() : node</td>
<td>remove</td>
<td>remove all nodes in the current set</td>
</tr>
<tr>
<td>clear()</td>
<td>remove</td>
<td>remove all nodes in the current set</td>
</tr>
</tbody>
</table>

Figure 6: Built-in Operations of nodeOrder (edgeOrder). The same operations are defined for nodeSeq (edgeSeq)

```plaintext
node(G) y = O1.popBack(); // O1 becomes {a}, y is b
O1.push(b); // O1 becomes {a,b} again
O2.push(c); O2.push(d); // O2 becomes {c,d}
O1.push(O2); // O2 becomes {a,b, c,d}
O1.popBack(); O1.popBack(); // O1 becomes {a,b} again
O1.pushFront(O2); // O1 becomes {c,d, a,b}
O1.pop(); O1.pop(); O1.push(c)// O1 becomes {a,b,c}, O2 is {c,d}
O1.pushFront(O2); // O1 becomes {d, a,b,c}
```

### 4.7.2 Iteration on Collection Types

Green-Marl allows iterating over collection types using `for` and `foreach` (Section 6.3). The only range word defined for all of the collection types is `items`, which stands for to iterate all the elements in the collection.

Let us consider for-iteration first, which adopts Sequential Consistency (Section 5.2.1). Iteration order over unordered collection is undefined and can be non-deterministic. For ordered collection, iteration order follows the natural order of the collection; or the reverse of such order (using ^ symbol, see Code 4.7.2).
Listing 23: Sequential Iteration Over Collections

```c
procedure foo(G: graph, S: N_S(G), O: N_O(G), Q: N_Q(G)) {
    // Assume
    // S = {a, b, c}
    // O = {a, b, c}
    // Q = {a, b, c, b}
    for(s: S.items) {
        // S can be visited in any order. {a,b,c} or {b,c,a} or {c,b,a}
        // However no element is repeated
    }
    for(s: O.items) {
        // O is visited as in the defined order: {a, b, c}
    }
    for(s: Q^.items) {
        // ^ is a special symbol to iterate the sequence in reverse order
        // Thus, iteration order would be: {b, c, b, a}
    }
}
```

Now consider `foreach`-iteration, which adopts Parallel Consistency (Section 5.2.2). In this case, every element in the collection is iterated at the same time, conceptually. Therefore order information is lost; an order becomes same to a set and a sequence to a multi-set. (See Code 22)

In both types of iterations (i.e. `for` and `foreach`) on any type of collection, it is an error to mutate the collection that is being iterated. A compiler can demote the error into a warning; and the behavior in such cases can be defined by the compiler (not by the language).

Listing 24: Parallel Iteration Over Collections and Mutation During Iteration

```c
// G: graph, a,b,c: node(G)
// S:nodeSet(G) = {a,b,c}
// O:nodeOrder(G)= {b,a,c}
// Q:nodeSeq(G) = {c,c,b,a}

// The following two iterations are same, since every element of
// the collection is iterated, concurrently.
// (i.e. loses ordering when doing parallel iteration)
foreach(s: S.items) {...}
foreach(o: O.items) {...}

// Modification during iteration is an error,
// even for sequential consistency
for(s: Q.items) {
    ...
    Q.pushFront(a); // Error - Modifying Q while being iterated.
}
```
4.8 Collections of Collections

Green-Marl provides a data type that can store elements of a collection type. This collection of collections type (short: CC) offers the same operations as a Sequence, but has some different properties. Note that all elements in the collection of collection have to be of the same type (e.g. only nodeSet).

- adding a collection to a CC will store a copy of the collection
- retrieving a collection from a CC will return the actual collection
- when iterating over the elements of a CC, the iterator points to the element in the collection. So changing the content of the iterator also changes the content of the element (compare Code 4.8).

Listing 25: Declaration and Usage of a Collection of Collection Type

```cpp
collection<nodeSet> c; // declares a collection of nodeSets
nodeSet set;
c. pushBack(set); // stores a copy of set in the collection
nodeOrder order;
c. pushBack(order); // type error, order is not a nodeSet
```

Listing 26: Receiving Copies or References to Elements

```cpp
collection<N_S> coll;
N_S set;
coll. pushFront(set);
N_S set2 = coll.front();
set2.add(n); // the set in coll does NOT contain n, because set2 is a copy!
for(s: coll.items) {
    s.add(n); // the set in coll now contains n because s is a reference.
}
```

4.9 Map Types

Green-Marl provides a map data type that maps keys to values. Maps are statically typed and the type of a map’s keys and values are part of its type. The following types are allowed as key: int, long, string, vertex, and edge. The following types are allowed as value: bool, int, long, float, double, string, vertex, edge, set, sequence, map, and vect. A map cannot contain duplicate keys and each key in the map is bound to exactly one value.

Unlike collections and properties, maps are not bound to a graph. Maps allow inserting and reading values with a certain key. If a key already exists in the map, a new insert will override the old value. When retrieving a value from the map for a key that does not exist, the map returns a default value. The default value depends on the type of the values in the map (see Table 4.8). The default value can also be the result of some built-in
<table>
<thead>
<tr>
<th>Value Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>false</td>
</tr>
<tr>
<td>int / long</td>
<td>0</td>
</tr>
<tr>
<td>float / double</td>
<td>0.0</td>
</tr>
<tr>
<td>string</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>node / edge</td>
<td>NIL</td>
</tr>
<tr>
<td>set</td>
<td>empty set</td>
</tr>
<tr>
<td>sequence</td>
<td>empty sequence</td>
</tr>
<tr>
<td>map</td>
<td>empty map</td>
</tr>
<tr>
<td>vect</td>
<td>empty vector</td>
</tr>
</tbody>
</table>

Figure 7: Default Values for Different Types of Map-Values

operations (see Section 4.9.1). When performing a reduction on an entry of a map with a value that has not been set before, it is assumed to be the default value.

Listing 27: Declaration of Map type and Reading and Writing Entries

```java
map<int, double> map1; // map declaration with Int as key-type and Double as value-type
map1[5] = 1.0; // map key 5 to value 1.0
double d = map1[5]; // read the value mapped to key 5

map<node(G), int> map2;
node(G) n;
map2[n] = 15; // n now mapped to 15
node(G2) n2;
map2[n2] = 25; // Error: key-type and n2 are bound to different graphs
```

4.9.1 Built-in Operations

Table 8 shows the different built-in operations on maps and their default behavior in case the map is empty. \( K \) and \( V \) refer to the type of the key/value of the map.

4.9.2 Behaviour in Parallel Semantics

**Writes:** If two writes are conflicting, that means they run in parallel and write to the same key, it is not deterministic which write will be visible afterwards, but at least one of the writes will be visible.

**Read-Writes:** If any of the built-in operations is called on a map while writing to it in parallel, the result of the operation is undefined.

**Reductions:** Reductions on map-entries are deterministic. Therefore it can be assumed, that there is no race-condition in the following code.
Listing 28: Parallel Iteration Over a Map Type

```cpp
map<node(G), int> map;
foreach (n: G.nodes) {
    map[n]++;
}
```

4.10 Filter Types

Green-Marl provides a `nodeFilter` and an `edgeFilter` type. They represent a black-box representation of a predicate (that is, a function returning boolean) on nodes or edges. Both types provide only a single operation, namely to evaluate this filter on a given node (or edge, respectively). The `evaluate` operation returns a boolean value.

Listing 29: Node and Edge Filter evaluate Usage

```cpp
procedure foo(G: graph, fn: nodeFilter(G), fe: edgeFilter(G)) {
    node(G) n = ...; // given a node from somewhere
    if (fn.evaluate(n)) { // we can evaluate a nodeFilter on it
        println("node filter on node {} evaluated to true", n);
    }
    edge(G) e = ...; // same goes for edges
    if (fe.evaluate(e)) {
        println("edge filter on edge {} evaluated to true", e);
    }
}
```

Since the `evaluate` operation returns a `bool` it can be used anywhere boolean expressions are allowed. In particular, this includes the filter expression part of iterators like `foreach` or reduction statements and the navigator expression for `inBFS` and `inDFS`.

Listing 30: Filter Evaluation in Iteration or Reduction

```cpp
// filter expression of iterators
foreach (n: G.nodes)(filter.evaluate(n)) {
    ...
}
// navigator for BFS or DFS traversal
inBFS (n: G.nodes from root)[filter.evaluate(n)] {
    ...
}
// reduction
int filteredSum = sum(n: G.nodes)(filter.evaluate(n)){n.A + n.B};
```

Note that filters are strictly read-only inside Green-Marl code and there is no mechanism to create a filter inside a Green-Marl procedure. In particular, one cannot declare variables with filter type, assign to variables with filter type, or declare procedure output arguments with filter type. Thus, the only way to obtain a filter in Green-Marl is as an input argument to a procedure.

Listing 31: Filters are immutable in Green-Marl

```cpp
// the only way to declare a name with filter type: input argument
```
The exact creation and execution mechanism for filters is implementation dependent. However, it is guaranteed that the invocation of a filter (by calling `evaluate`) is free of side-effects.

### 4.11 Implicit Graph Binding

Usually nodes, edges as well as properties, collections and filters have to be bound to a graph explicitly. But if there is only one graph instance defined in a context, it is possible to use implicit binding, since the compiler can figure out the correct binding by itself. Using explicit binding together with implicit binding within one procedure is possible.

**Listing 32: Implicit and Explicit Graph Binding**

```plaintext
procedure implicit(G: graph) {
  node n; // implicit binding possible
  node(G) n2; // explicit binding still possible
}
```

### 4.12 Variance in Local Method Calls

Local procedures in Green-Marl are covariant in their input arguments and invariant in their output arguments. This means that you can call a local procedure with an input argument whose type is a subtype of the corresponding parameter. However, the output arguments need to be of the same type as the corresponding parameter.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Semantics</th>
<th>Default Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>size() : int</td>
<td>returns the number of keys in the map</td>
<td>returns 0</td>
</tr>
<tr>
<td>clear()</td>
<td>deletes all keys and values from the map</td>
<td>no op</td>
</tr>
<tr>
<td>remove(K)</td>
<td>deletes the key from the map so that the default value will be returned for the key and hasKey will return false</td>
<td>no op</td>
</tr>
<tr>
<td>hasKey(K) : bool</td>
<td>returns true if the map contains a mapping for the specified key. Otherwise false</td>
<td>returns false</td>
</tr>
<tr>
<td>hasMaxValue(K) : bool</td>
<td>returns true if the specified key is mapped to the largest value in the map. Otherwise false</td>
<td>returns false</td>
</tr>
<tr>
<td>hasMinValue(K) : bool</td>
<td>returns true if the specified key is mapped to the smallest value in the map. Otherwise false</td>
<td>returns false</td>
</tr>
<tr>
<td>getMaxKey() : K</td>
<td>returns the key that is mapped to the highest value in the map. If there is more than one key mapped to the highest value, it is indeterminate which one is returned</td>
<td>undefined</td>
</tr>
<tr>
<td>getMinKey() : K</td>
<td>returns the key that is mapped to the smallest value in the map. If there is more than one key mapped to the highest value, it is indeterminate which one is returned</td>
<td>undefined</td>
</tr>
<tr>
<td>getMaxValue() : V</td>
<td>returns the largest value in the map</td>
<td>returns the default value</td>
</tr>
<tr>
<td>getMinValue() : V</td>
<td>returns the smallest value in the map</td>
<td>returns the default value</td>
</tr>
</tbody>
</table>

Figure 8: Built-in Operations of map
foreach iterate every element in the given range (e.g. nodes in a graph, elements in a set) in parallel | Section 6.3.1

inBFS traverse the graph in breadth-first search order. Visit every node of the same BFS level (i.e. nodes having the same hop distance from the root node) in parallel | Section 6.3.2

Table 11: Green-Marl Syntax for a Parallel Execution Region

5 Parallel Execution and Consistency

5.1 Parallel Regions

Green-Marl is designed to provide intuitive ways in exploiting data parallelism of graph algorithm. The basic idea is similar to that of OpenMP: the user describes parallel regions with simple language constructs such as `foreach`, while the compiler and the runtime handles details of parallel execution such as thread creation or job scheduling.

The parallel execution model adopted in Green-Marl is fork-join style (or more accurately split-merge style) parallelism. That is, the execution becomes parallel at the beginning of a parallel region; at the end of the region, all those parallel executions are merged and the execution becomes sequential again. In other words, all the concurrent executions of the parallel region are synchronized at the end of parallel region.

Listing 33: Parallel Region Example

```
int z = 0;    // Sequential Execution

foreach (n: G.nodes){ // beginning for parallel execution region
    // all the instances of the loop-iteration happen in parallel
    z += n.A;
} // End of parallel execution region:
    // all the parallel executions of the above region are merged
    // before continue to the next statement

int k = z + 1; // Sequential Execution resumes
```

Conceptually, the execution of a parallel region is maximally parallelized. For instance, in the above code example, the parallel region is executed concurrently for every node \( n \) in the graph \( G \). Therefore, no ordering is guaranteed between instances of this parallel execution region. However, this full parallelization is only a conceptual tool that does not enforce any regulation in implementation – the implementation of compiler/runtime is free to choose how many threads are actually utilized for the parallel region.

A parallel region, however, is bound to a memory consistency that is different from sequential execution. Section 5.2 provides more detailed discussions about memory consistencies in Green-Marl.

The following table summarizes the syntax for parallel execution region in Green-Marl.

Green-Marl basically expects the user to mark parallel execution region with above language constructs; however there are also ways to suggest parallel execution in an implicit way (See Section 5.5).
5.1.1 Nested Parallelism

Green-Marl allows nesting of parallel regions in any depth. See Listing 34 as an example. Note that in the case of nested parallel regions, we can refer a specific parallel region (i.e. a specific loop) using the iterator name of the loop.

Listing 34: Nested Parallel Execution Region

```java
foreach (n: G. nodes) { // outmost loop (n-loop)
    int x = 0;
    foreach (t: n. nbrs) { // (t-loop)
        foreach (r: t. nbrs) { // innermost loop (r-loop)
            if (r.A > t.A) x += r.A;
        }
    }
    n.B = x*2;
}
```

Each parallel region is fork-joined (or split-merged) independently. As an example, consider Listing 34 again, where the outermost loop (n-loop) are concurrently executed. Among all those concurrent execution instances of the n-loop, let us first focus on a single specific one; we denote it as an n-instance. However, this specific n-instance is further split at the beginning of a nested parallel region (t-loop) at line 3. Similarly, each t-instance is further split into multiple concurrent r-instances at line 4. Merging of concurrent execution instances happen in reverse order. That is, all the concurrent r-instances from the same t-instance are merged before the t-instance reaches to line 7. Similarly all t-instances from the same n-instance are merged before line 8.

Note that all the concurrent execution instances are independent. Therefore, in the previous example, even when all the t-instances from a single n-instance have been merged at line 8, there can be other t-instances concurrently executing line 4 – those t-instances are originated from other n-instances. The following figure illustrates this concept:

5.2 Memory Consistency in Green-Marl

5.2.1 Sequential Memory Consistency

The basic memory consistency model of Green-Marl is sequential memory consistency. That is, the effect of a memory update is visible to any sentence that comes after in program order. For example, while (or do-while) loop (Section 6.3.3) adopts sequential memory consistency, just like in C. for loops (Section 6.3.1) adopt sequential memory consistency, as well. Note that, though, there is no requirement of any specific ordering when performing for-iteration on a set or a multi-set; the iteration order can be even non-deterministic (Section 4.7.2). Nevertheless, no matter what order the set is iterated, sequential memory consistency is always be preserved. In other words, the execution of a for-loop is always serialize-able. See below code example.

Listing 35: Sequential Consistency Example
int x = 1; // Write to x is visible in the next sentence
int z = x + 1;

// for-iteration on the set of nodes in G:
// it is not defined what order those nodes are iterated.
// However, there must be at least one sequential order.
for (n: G.nodes) {
    x = x + n.A;
}

DFS-order graph traversal (Section 6.3.2) also adopts sequential consistency.

5.2.2 Parallel Memory Consistency

On the other hand, parallel regions (Section 5.1) adopt a different memory consistency model. This model assumes that the parallel region is being executed concurrently and that there are natural data races between concurrent execution instances. Therefore this model does NOT guarantee anything about the visibility of writes which are made by concurrent executions. That is to say, a write made by one instance of a parallel region may or may not be visible to other instances of the parallel region.

The model guarantees the following things.

- Self-visibility: A write by an execution instance is always visible to the current instance later in program order, unless the write is overwritten by another concurrent instance.

- Eventual visibility: At the end of the parallel region, when all the concurrent execu-
tions are merged, every write made by concurrent execution instance of the region becomes visible.

As an example, let’s think about the case of the following bipartite graph (Figure 5.2.2) and apply it to the following code example (Listing 36):

Listing 36: Parallel Consistency Example I

```c
// Color, Value is N.P<int>(G)
// Initialize values of all nodes as 0
G.value = 0;

// Parallel Execution
foreach (n: G.nodes) {
    // Blue node changes its 'value' from 0 -> 1
    if (n.Color == BLUE)
        n.value = 1;
    // Red node sums up 'value' of its neighboring blue nodes.
    // However value change of blue nodes may or may not be visible.
    else
        n.value = sum (t: n.nbrs) {t.value};
}
```

The result is non-deterministic: at the end of the parallel region (i.e. after n-loop), value of red nodes can be 0, 1, or 2. However, value of blue nodes is guaranteed to be 1.

Write-Write data race is the case when multiple concurrent writes is being written to the same variable (or property location). In such a case, at the end of the parallel region, one (and only one) of those writes becomes effective; however, it is non-deterministic which of those writes will be the effective one. As an example, the following code results in a non-deterministic value of x, when applied to the code in Listing 5.2.2 x can be either blue or red.

Listing 37: Parallel Consistency Example II
Read-Write data race is the case when a variable (or a property access) is concurrently read and write at the same time. In this case, a concurrent read may or may not see the result of a write from another concurrent instance.

Listing 38: Parallel Consistency Example III

```java
// Color is nodeProp<int>(G)
int x = 0;
foreach (n: G.nodes) {
    x = n.Color; // multiple writes to a single location
}
// One of those writes become visible at the end of parallel loop.
// x can be either Blue or Red.

// concurrent reads from the same variable
if (x != n.Color) {
    // The execution can be come into this branch,
    // because a write from another concurrent execution may have altered
    // value of x between line 3 and line 4.
    ...
}
```

Moreover, under parallel memory consistency, there is no guarantee that multiple writes would be visible in the same order to every concurrent execution instance. (In other words, total store order is not guaranteed.) As an example, suppose the following code is applied to the previous graph instance (Figure 5.2.2). It is possible that some node end up with value having LEFT_FIRST, while others having RIGHT_FIRST.

Listing 39: Parallel Consistency Example IV

```java
// flag, value is nothing but nodeProp<int>
// node(G): node_1 and node_3 are the blue nodes In the figure
// LEFT_FIRST, RIGHT_FIRST, SAME -> integer constants
G.value = 0; // Initially, all nodes have value 0
foreach (n: G.nodes) {
    // Node 1 and 3 update their value to 1
    if ((n == node_1) || (n == node_3))
        n.value = 1;
    // Other nodes monitor values of node number 1 and 3.
    else {
        // Check if the value has been changed at node 1 and node 3.
        bool left_changed = (node_1.value == 1);
        bool right_changed = (node_3.value == 1);
        // Check which one of them has been changed first.
    }
```

45
Fundamentally, Green-Marl’s parallel memory consistency assumes that concurrent reads and writes inevitably incur data-races and thus becomes non-deterministic. The designers of Green-Marl believe that such non-determinism is a fundamental nature of parallel graph processing. Nevertheless, Green-Marl provides methods which can enforce certain determinism out of such non-deterministic concurrent execution. Section 5.3 discusses such mechanisms in detail.

Note that all the code examples above (Listing 36 - Listing 39) have non-determinism induced by the data-races. A compiler should detect such non-determinism and report it to the user. See Section 5.6.1 for details.

5.2.3 Bulk-Synchronous Memory Consistency

Green-Marl supports Bulk-Synchronous memory consistency [1] as well through deferred assignment statement (Section 5.3.2). In Bulk-Synchronous memory consistency, it is guaranteed that a write to a memory location is not visible to every concurrent execution instance, even to the current one that has made the write, until the (specified) synchronization point. At the synchronization point, on the other hand, all the updates made inside loop become visible at once. The synchronization point is the end of binding loop, which is usually the current loop; however, the user can explicitly specify synchronization point in the case of nested parallel regions. See Section 5.3.3.

Listing 40: Bulk-Synchronous Consistency Example

```
// A is a nodeProp<int>(G)
foreach (n: G. nodes ) {
  // Writing of n.A is deferred until the end of the binding loop, i.e. n-loop
  n.A <= sum (t: n. nbrs ) {
    // Thus, reading of t.A always gives the unmodified value. t.A)
  }
  // at the end of binding-loop, all the writes to A become visible at once
}
```

5.3 Determinism Under Parallel Memory Consistency

As mentioned in Section 5.2.2, parallel memory consistency allows data races among concurrent writes and concurrent reads. However, Green-Marl provides certain mechanisms that enforce deterministic results out of such non-deterministic executions. This section discusses such mechanisms in detail.
5.3.1 Reductions

Reduction is the most important determinism-enforcing mechanism in Green-Marl. In a general sense, reduction is a mathematical mechanism that computes a single representative value out of a set of values in a deterministic way. For example, summation is a reduction which adds up all the values in a collection.

In Green-Marl, reduction can take one of two different forms: Assignment form and Expression form. Let us consider Assignment form first.

The following code is an example of reduction in Green-Marl. The \( += \) symbol at line 5 stands for reduction by addition: at the end of the loop \( x \) will contain the sum of property \( A \) for all nodes in the Graph. Note that \( += \) symbol at line 5 is not the same to reading and writing of the same variable \( y \) at line 6 – such a reading and writing results in a non-deterministic value of \( y \) at the end of n-loop.

```c
nodeProp<int>(G) A;
int x = 0;
int y = 0;
foreach (n: G.nodes) {
  x += n.A; // Reduction by addition
  y = y + n.A; // Not a reduction. The result is non-deterministic
}
```

The expression form of a reduction is a syntactic sugar that is functionally equivalent to the same reduction in assignment form - it simply allows more convenient (or intuitive) code writing. See the following code, as an example of summation (i.e. reduction by addition) in expression form.

```c
// A is a nodeProp<int>
int x;
x = sum (n:G.nodes) {n.A};
// The above expression is equivalent to below
x = 0;
foreach (n:G.nodes)
x += n.A;
```

However, the semantics of assignment form is, in fact, slightly different from corresponding expression form, because the expression form also implies automatic initialization of the target location. On the contrary, the user has to provide explicit initialization for assignment form. The following table summarizes how two forms of reduction can be switched from one to another.

```c
target = reduce_operator (iterator: range) {body_expression}
```

```c
```

Listing 41: Reduction by Addition (Assignment Form)

Listing 42: Reduction by Addition (Expression Form)

Listing 43: Reductions in Expression Form

Listing 44: Reductions in Assignment Form
<table>
<thead>
<tr>
<th>Reduce Assign</th>
<th>Reduce Operator</th>
<th>Auto Initializer</th>
<th>Semantics</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>sum</td>
<td>0</td>
<td>addition</td>
<td>Numeric</td>
</tr>
<tr>
<td>*=</td>
<td>product</td>
<td>1</td>
<td>multiplication</td>
<td>Numeric</td>
</tr>
<tr>
<td>max=</td>
<td>max</td>
<td>-INF</td>
<td>maximum</td>
<td>Numeric</td>
</tr>
<tr>
<td>min=</td>
<td>min</td>
<td>+INF</td>
<td>minimum</td>
<td>Numeric</td>
</tr>
<tr>
<td>++</td>
<td>count</td>
<td>0</td>
<td>syntactic sugar for += 1</td>
<td>Numeric</td>
</tr>
<tr>
<td>N/A</td>
<td>avg</td>
<td>0</td>
<td>arithmetic mean; syntax sugar for sum / count</td>
<td>Numeric</td>
</tr>
<tr>
<td>&amp;&amp;=</td>
<td>all</td>
<td>true</td>
<td>logical ‘and’</td>
<td>Boolean</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=</td>
<td>any</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 12: Reductions in Green-Marl

```java
// A,B is a nodeProp<int>
target = automatic_initializer
foreach (iterator : range) {
   target reduce_assign body_expression
}
```

The following table summarizes all the reductions defined in Green-Marl, both in assignment form and reduction form:

A target location of reduction can be either a scalar variable or a property. The following code shows how a reduction can be applied to a property.

Listing 45: Reduction to Property

```java
// A,B is a nodeProp<int>
foreach (n: G.nodes)
   foreach (t: n nbrs)
      t.A += n.B; // The entire property A becomes the target of reduction
```

An interesting feature of Green-Marl reduction is to augment ‘reduction by minimum/-maximum’ with ‘argmin/argmax expressions’. In other words, the user can obtain not only the maximum value of the body expression but also sub-expressions which maximize the body expression, or argmax. Line 7 of Listing 46 shows an example of this feature. Here, the maximum of the body expression n.A * 2 is stored into x. However, this maximum value is stored along with two other expressions (n and n.A) into two other variables i and a, at the same time: it is guaranteed that the values stored into i and a come from the same execution instance that stored x. Also note that Line 9-11 does not achieve the same determinism – y and j might be written from two distinct n-instances.

Listing 46: Max and Argmax

```java
// A,B is a nodeProp<int>
int x,y; x= 0; y=0; // variable to store max
```
<table>
<thead>
<tr>
<th>Reduce Operator</th>
<th>Auto Initializer</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>vector, all entries = 0</td>
<td>same as expression type</td>
</tr>
<tr>
<td>avg</td>
<td>vector, all entries = 0</td>
<td>vect&lt;double&gt;</td>
</tr>
</tbody>
</table>

Table 13: Auto Initializer for Reductions on Vector Properties

```
3 int a,b; // variable to store argmax
4 node(G) i,j; // variable to store argmax
5 foreach (n: G. nodes) {
6   // compute maximum of n.A * 2 as well as the arguments maximize it
7   x <i, a> max= n.A * 2 <n, n.A>;
8   // The result of following sentences (y,j,b) are non-deterministic
9   if (n.A*2 > y) {
10      y = n.A*2; j = n; b = n.B;
11   }
12 } // i,a are guaranteed to be the values that have saved with maximum x
13 // But there is no such guarantee for j and b with y
```

Variables that are being reduced should not be read or written otherwise. A compiler should detect such an access and give an error, as in the following code example. However, the compiler may provide an option to demote such errors into warnings; the semantics of such access (i.e. reading from or writing to a reduction-target location) is undefined by the language specification, i.e. a compiler may declare its own semantics for this.

Listing 47: Example of Errors on Reduction Variables

```
// A,B is a nodeProp<int>
int x = 0;
int y;
foreach (n: G. nodes){
 x += n.A; // x is a reduction-target by addition.
 y = x + 1; // error. x cannot be read because it is being reduced
 if (y > 100)
  x = 100; // error. x cannot be written because it is being reduced
 x *= 1; // error. x is already being reduced with a different operator
}
```

Reductions are defined on properties of vector type too. The semantics do not change, except that the auto initializer and the return value of the reductions will be of vector type. See Table 13 for more information. The return type of the reduction will have the same dimension as the expression inside the reduction. Please note that max, min are not defined on vectors since there is no defined ordering.

Listing 48: Example of Reductions on Vector Type Properties

```
// A,B is a nodeProp<int>
int x = 0;
int y;
foreach (n: G. nodes){
 x += n.A; // x is a reduction-target by addition.
 y = x + 1; // error. x cannot be read because it is being reduced
 if (y > 100)
  x = 100; // error. x cannot be written because it is being reduced
 x *= 1; // error. x is already being reduced with a different operator
}
```

6For example, a compiler may define the semantics of reading a reduction-target variable as to give the partial reduction result.
5.3.2 Deferred Assignments

Green-Marl supports Bulk Synchronous consistency (Section 5.2.2) through deferred assignments. Syntax-wise, deferred assignment is quite similar to reduction in assignment from. (See Listing 38 in Section 5.2.2) The only noticeable difference is the use of <= symbol in place of reduction assignment symbols such as +=.

Unlike to reduction, however, a variable (or a property) can be read inside a loop, even though it is currently being assigned with deference. In this case, a read of the variable always gives the unmodified value since the writes to the variable is not effective yet but deferred to the end of binding loop (Section 5.2.2).

The variable, which is being defer-assigned, should not be written otherwise; see Listing 49 below for example.

Listing 49: Bulk-Synchronous Consistency Example

```plaintext
// A,B is a nodeProp <int>
foreach (n: G.nodes) {
    // Writing of n.A is deferred until the end of the current loop
    n.A <= sum (t: n.nbrs)  // Reading of t.A therefore gives the unmodified value
        {t.A};
    // error, property A is being defer-assigned during n-loop.
    // It cannot be written otherwise.
    n.A = n.B + 1;
}
```

5.3.3 Visibility in Nested Parallel Regions

The visibility of writes in nested parallel regions (Section 5.1.1) is defined in a recursive manner. We explain this with the following code example (Listing 50). Inside a nested loop, at line 7, reading of y is non-deterministic because of the concurrent writes to y at line 9. However once t-loop is finished, reading of y becomes deterministic at line 12. On the other hand, reading of x at line 13 is still non-deterministic, since there are still concurrent writes inside this outer loop at line 15.

Listing 50: Parallel Consistency Example I

```plaintext
int x = 0;
foreach (n: G.nodes) {
    int y = 0;
    // Nested Loop.
    // Every t writes to the same y.
}```
foreach (t: n.nbrs) {
    int z1 = y; // reading of y is non-deterministic (because of line 7)
    int z2 = x; // reading of x is non-deterministic (because of line 8, 12)
    y = 1;
    x = 1;
}
int z1 = y; // reading of y is deterministic
int z2 = x; // reading of x is non-deterministic (because of line 13)
y = 2;
x = 2;
}

The binding loop of deferred assignment (Section 5.3.2) might be ambiguous in case of nested loops, in which case a compiler must try finding an appropriate binding loop in place of the user. The appropriate binding loop means a nested loop that does not induce any further data race or double-bound error, if the loop is chosen as the binding loop. Still, a compiler may fail to find such a loop. In such case, the compiler should raise an error. See Listing 51 for examples.

Listing 51: Example Cases of Automatic Detection of Binding Loops

// G: graph
// A, B: N_P<int>(G)
// case 1
foreach (s: G.nodes) {
    // obviously, this reduction is bound to s-loop.
    // Compiler should find it.
    s.A += 1
}
// case 2
foreach(s: G.nodes){
    foreach(t: s.nbrs){
        // Should be bound to s-loop. Otherwise induces data-race.
        // A naive compiler implementation may not find it, however.
        t.A += s.B;
    }
}
// case 3
foreach(s: G.nodes){
    foreach(t: s.nbrs){
        // Can be bound to either t-loop or s-loop.
        s.A += t.B;
    }
}
// case 3 + case 2
foreach(s: G.nodes){
    foreach(t: s.nbrs){
        // Now, this should be bound to s-loop because of another reduction
        // inside the following t2-loop. A compiler may fail to find the
        // correct binding loop.
        s.A += t.B;
    }
}
5.4 Operations on Collection Types under Parallel Consistency

Section 4.7 has discussed the semantics of operations defined on collection types under sequential consistency. This section clarifies those semantics under parallel consistency (Section 5.2.2).

In case of ‘append’ class, the order between elements that are appended under parallel consistency is non-deterministic. However, (1) every appended element becomes visible at the end of the parallel region and (2) the ordering with respect to initial content of the collection is preserved. See the following code example.

**Listing 52: Append under Parallel Consistency**

```java
// Assume a,b,c,d,e are distinct node.
nodeOrder(G) O1, O2, O3;
nodeSet(G) S, S1;
// Assume S = {a,b,c,d}. S1 ={} O1 ={}
foreach (s: S.items) {
    // Concurrent addition guarantees no ordering
    S1.add(s);
    O1.push(s);
}
// S1 is {a,b,c,d} => order does not matter
// O1 can be in any order: {a,b,c,d},{b,d,a,c} ...

// Assume S = {a,b,c,d}. O2 ={e}
foreach (s: S.items) {
    // However, ordering is preserved, with respect to the initial data
    if ((s==a) | (s==b)) O2.pushBack(s);
    else O2.pushFront(s);
}
// O2 can be {(a,b},e ,(c,d)},
// i.e. a and b comes before e (in any order); c and d after d.

// Assume S = {a,b}. O1 ={a,b}, O2 = {c,d}, O3 = {}
foreach (s: S.items) {
    // Addition of collection is not atomic
    if (s==a) O3.push(O1);
    else if (s==b) O3.push(O2);
}
// O3 can be end up with {a,c,b,d}. (i.e. push(O1) can push(O3) can be interleaved.
```

As for ‘remove’ class, the semantics is different for unordered collection and ordered collection.
For unordered collections (i.e. set), the semantics of concurrent removal is the eventual one; all the removed element is surely removed at the end of the loop.

For ordered collections (i.e. order, sequence), the semantics of concurrent removal is undefined.

The case of ordered collection may not be from what is normally expected; however, note that it essentially enforces (unordered but) sequential consistency if each parallel `pop()` is guaranteed to obtain a distinct element from the collection. Therefore it remains as undefined by the language specification. Nevertheless, a compiler may refine the undefined semantics of removal operation under parallel consistency as unordered sequential consistency.

### Listing 53: Remove under Parallel Consistency

```c
// Assume a,b,c,d,e are distinct node.
nodeSeq(G) Q1,Q2;
nodeSet(G) S, S1;

// Assume S = {a,b,c,d}. S1 ={a,b,c}
foreach (s1: S1.items) {
    S.remove(s1); // Concurrent removal is eventual.
}
// S = {d} here for sure

// Assume S1 ={a,b,c,d}, Q1 = {a,b}, Q2 = {}
foreach (s1: S1.items) {
    node(G) n = Q1.pop(); // Error - concurrent removal is undefined for ordered collection
    if (n != NIL ) Q2.push(n);
}
// If we do not regard concurrent pop() as an error under parallel consistency,
// Q2 can possibly be {a,a,a,b}, i.e., same element can be popped multiple times.
// A compiler may have an option to enforce serialization on is concurrent pop case,
// so that Q2 can only be {a,b} or {b,a}
```

The semantics of copy (by-assignment) under parallel consistency is eventual and atomic. See the following code for explanation.

### Listing 54: Copy under Parallel Consistency

```c
nodeSeq(G) Q1,Q2;
nodeSet(G) S, S1, S2;

// Assume S = {e}, S1 = {a,b}, S2 = {c,d}
foreach (s: G.nodes) {
    if (s.Color == Blue ) S = S1; // Concurrent removal is eventual
    else S = S2;
}
// S = {a,b} or {c,d} here for sure
```
Table 14: Reductions in Green-Marl

<table>
<thead>
<tr>
<th></th>
<th>Lookup</th>
<th>Append</th>
<th>Remove</th>
<th>Assign</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lookup</td>
<td>Okay</td>
<td>Warning (non-deterministic)</td>
<td>Warning (non-deterministic)</td>
<td>Warning (non-deterministic)</td>
</tr>
<tr>
<td>Append</td>
<td></td>
<td>Okay</td>
<td>Error (undefined)</td>
<td>Error (undefined)</td>
</tr>
<tr>
<td>Remove</td>
<td></td>
<td></td>
<td>Set: Okay, Order/ Seq: Error (undefined)</td>
<td>Error (undefined)</td>
</tr>
<tr>
<td>Assign</td>
<td></td>
<td></td>
<td></td>
<td>Warning (atomic, non-deterministic)</td>
</tr>
</tbody>
</table>

Green-Marl does not encourage users to mix up heterogeneous classes of operation on a single collection under parallel consistency – e.g., adding elements into a collection while removing elements from the same collection concurrently. In such cases, either the result is non-deterministic or undefined. Table 5.4 summarizes the semantics and compiler action when two different classes of operations are performed on a collection.

Let us explain Table 5.4 a little bit more. First, the results of look-up operations are non-deterministic when they are mixed up with other operations in a parallel region - e.g. calling `size()` of a collection while the collection is concurrently growing. Compiler should warn the user in such cases.

Second, it is an error to mix up two different operation classes other than lookup: the semantics is not defined for such a case and it should be treated as an error. In essence, a collection in Green-Marl is expected either to grow or to shrink in a parallel execution region, but not both. However, a compiler may give an option to define the semantics of such mixed concurrent operations as having unordered sequential consistency. See the following code for example.

Listing 55: Append under Parallel Consistency

```c
// Suppose G has five nodes {a,b,c,d,e} and Color[a,c,e]= Blue
nodeSeq(G) Q1;
nodeOrder(G) O1;
// Assume Q1 = {a,a,b,b,c,d,e}. O1 = {a,b,c}
foreach (q: Q1.items) {
    // Concurrent addition guarantees no ordering
    if (q.color == Blue) O1.add(q);
    else if O1.remove(q);
}
// What is a right semantics, if multiple values are added and removed to a collection at the exact same time?
// Green-Marl language specifies this as undefined and thus erroneous.
// However a G-M compiler can give an option to turn off above error.
// In such case, the compiler assumes 'unordered sequential consistency'
```

54
5.5 Implicit Parallel Context (Syntactic Sugar)

In order to make it easy to exploit natural data-parallelism in graph algorithms, Green-Marl provides a few syntactic sugars that can even further simplify writing parallel regions.

Reduction in assignment form (Section 5.3.1) is one example of such syntax sugars. Section 5.3.1 explained that a reduction in expression form is completely identical to its corresponding assignment form combined with automatic initialization.

The other one is collective or group assignment. In Green-Marl, one can assign property values for a group of nodes (edges) in a simple way, using an underscore as placeholder in the group assignment syntax. See the following code snippet for example.

Listing 56: Example of Collective Assignments

```cpp
nodeProp<int> A, B;
setNode(G) S;

// ... 
G.A = 0; // For every node in G, set its A value as 0
S.A = 2; // For every node in Set S, set its B value as 0
G.B = _.A + 1; // For every node in G, set its A value as its B value plus 1
G.A = _.degree(); // Sets the value of A for each node to the node's degree

// The above three statements are equivalent to the statements below
for (n: G.nodes) n.A = 0;
for (n: S.items) n.A = 2;
for (n: G.nodes) n.B = n.A + 2;
```

The underscore binds to the variable on the left-hand-side of the assignment. Note that the use of the placeholder ( _. ) is only allowed in group assignments and only in the right-hand-side of the assignment. Also placeholders may only be used for graph type variables.

As can be seen in the previous code example, group assignment is identical to For-iteration combined with simple assignment. However, it is strongly recommended that the compiler should try automatically parallelizing such For-iterations that came from group assignments; in many cases, they are embarrassingly parallel and it is very easy to check if so.

5.6 Notes for Green-Marl Compilers

5.6.1 Static Compiler Analysis for Data-Race Detection

Throughout Section 5.6.1 it has been discussed that a Green-Marl compiler should analyze the code and detect (potential) data-races in parallel regions and should give errors or
Table 15: Default Compiler Actions for Detected Data Race

<table>
<thead>
<tr>
<th></th>
<th>Read</th>
<th>Write</th>
<th>Reduce</th>
<th>Deferred-Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Okay</td>
<td>Warning</td>
<td>Error</td>
<td>Okay</td>
</tr>
<tr>
<td>Write</td>
<td></td>
<td>Warning</td>
<td>Error</td>
<td></td>
</tr>
<tr>
<td>Reduce</td>
<td></td>
<td></td>
<td>Okay, if same op Error otherwise</td>
<td>Error</td>
</tr>
<tr>
<td>Deferred-Write</td>
<td></td>
<td></td>
<td></td>
<td>Warning</td>
</tr>
</tbody>
</table>

Table 5.4 summarizes default compiler actions for each data race type. However, the compiler can provide an option to demote Reduce-Read or Reduce-Write error into warning.

5.6.2 Selective Parallel Execution of Parallel Regions

By its design, Green-Marl drives the user to expose all the parallel regions in his/her algorithm so that the compiler or runtime can exploit parallelism inside those regions.

However, it is not necessary for all the parallel regions to be executed actually in parallel at the runtime. To the contrary, it is recommended for the compiler or the runtime executing only some of those regions in parallel, selectively, in order that the overhead of parallelization would not exceed performance benefit from it. Consider the
following code example (Listing 58). Although the original user description is a nested parallel region (line 1-5), the compiler can still decide to parallelize the outer loop only. In such case, the compiler is able to replace the reduction (line 3) with normal read and write (line 9) – parallel reduction typically is implemented using atomic instructions which take 2-3x times slower than normal read and write instructions.

Listing 58: Selection of Parallel Execution

```java
foreach (s: G.nodes) { // outer loop (s-loop)
    foreach (t: s.nbrs) { // inner loop (t-loop)
        s.A += t.B;
    }
}
// The compiler may choose to parallelize only the outer loop
foreach (s: G.nodes) { // outer loop (s-loop) -> parallel
    for (t: s.nbrs) { // inner loop (t-loop) -> sequential
        s.A = s.A + t.B // reduction can be replaced with normal read & write
    }
}
```

Likewise, the compiler or runtime may choose to execute sequential regions in parallel, as long as they are sure to deliver sequential consistency as in the original program. For example, let us consider Listing 59. The first loop (line 2-4) can be safely transformed into a parallel loop (i.e. `foreach`) since `nodeSet` guarantees uniqueness of elements and property A is only accessed through s. On the other hand, the second loop (line 9-11) cannot be parallelized naively; there is be a read-write data race between writing to property A (via n.A) and reading of property A (via t.A). However, the compiler may still (optionally) generate a parallel execution code for the loop that delivers (unordered) sequential consistency by using reader-writer locks on nodes or transactional memory.

Listing 59: Parallel Execution of Sequential Loops

```java
// The compiler can safely change following for into foreach
nodeSet S;
for (s: S.nodes) {
    s.A = s.A +1 ;
}
// Blind parallel execution does not guarantee sequential consistency.
// A compiler may parallelize it with unordered sequential consistency,
// using locks or transactional memory.
for (n: G.nodes) {
    n.A = sum(t:s.nbrs){t.A};
}
```

7For example, if the target system is a multi-core CPU which features only several processors, parallelizing the outer loop would be enough to consume all the CPU and memory bandwidth of the system – further parallelizing nested loops would be pure overhead.
5.6.3 Implementation of Collection Types

Green-Marl language puts no constraint on the implementation of Collection types (Section 4.7) as long as its behavior under sequential consistency (Section 4.7) and parallel behavior (Section 5.4) is guaranteed.

The compiler is encouraged to use any best implementation of collections available for the target system - e.g. Multi-core CPU or GPU or distributed environments. The compiler may also exploit the fact that the size of the set is bounded; for example the maximum size of node set of a graph is the number of the nodes in the graph, at most. Thus a compiler may use the bitmap representation for the set, for instance.

The compiler may use any different implementation for internal collection objects, other than 1-1 mapping specification to the target language. For example consider following code example – the implementation of S1 and S2 can be different.

Listing 60: Different Implementations for Internal Collections

```java
// Say nodeSet is mapped to a Queue by compiler specification. // S1 should be implemented as a Queue.
procedure foo(G: graph, S1: nodeSet(G), A: N_P<int>(G)) {
    nodeSet(G) S2; // However, S2 can be implemented as anything.
    foreach (s: S1. items )(s.A > 0) {
        S2. add(s);
    }
    while (some_condition(G)) {
        int m; node (G) am;
        m = +INF;
        foreach (s: S2. items ) {
            m =m <am> min = s.A <s>;
        }
        S2. remove(am); // S2 is always removed by some min-value.
    }
    // S2 might be implemented as a heap.
}
```
6 Expressions and Statements: Details

6.1 Expressions

In Green-Marl, expressions are strictly differentiated from sentences: expressions are always side-effect free. Therefore a compiler can always reorder computation of sub-expressions or apply short-circuits, safely.

Green-Marl syntax has strictly different positions for sentences and expressions. Expressions are placed at (1) RHS of assignments (Section 6.2), (2) in place of input-arguments of procedure call sites, (3) conditional parts of if and (do-)while statements (Section 6.3.3), (4) filter and navigators in for/foreach iteration (Section 6.3.1) and DFS/BFS traversal (Section 6.3.2). In the last two positions, the expression should be boolean-typed.

Every expression is typed. Operations are defined only between compatible types. See Section 4.2.2 for related discussion.

6.1.1 Operations on Numeric Types

There are two different kinds of operations defined for numeric types: Arithmetic and Comparison. Arithmetic binary operators are +, -, *, /, and % (only for int, long, float and double). Comparison operators are <, >, ==, and !=. The semantics of those operators are same to C. There is also the unary operator – as in C. Finally, there is a parenthesis-like operator | |, which gives the absolute value of a numeric-type expression inside.

Listing 61: Examples of || Operator

```c
// z becomes 3.5, y becomes 3
float z = | -3.5 |;
int y = | -3 |;
```

6.1.2 Operations on Boolean Type

There are two different kinds of operations defined for numeric types: Logical and Comparison. Logical binary operators are && and ||. Comparison operators are == and !=. Also there is unary operator !. The semantics of those operators are same to C.

6.1.3 Conditional Operator

Listing 62: Examples of Conditional Operator

```c
Bool_expr ? expr1 : expr2
```

Green-Marl also provides Ternary conditional operator as in C. The semantics is if the value of Boolean expression is true, the value of the ternary expression is expr1; expr2 otherwise. The type of expr1 and expr2 should be the same.
6.1.4 Reduction Operations

Reductions are discussed in Section 5.3.1 where reduction in Green-Marl can take any of assignment form or operation form. Reductions in operation form take the following syntax:

\[
\text{reduction_op (iterator_name : source_name . range_word) ( (filter_expr) )} \{ \text{body_expression} \}
\]

Iterator (Section 3.2.3) is a read-only variable that points the current item of the iteration range. Source variable and range word together defines the range of iteration. The source variable can be of type \text{graph} (Section 4.4), \text{node}, \text{edge} (Section 4.5.1), or any collection (Section 4.7.2), while each type can be followed by different range words. Note that the type of an iterator is automatically determined by the type of source and range word.

The semantics of reduction operation is to compute body expression for every iterator value in the range and apply reduction operation for all of those computed values. Table 12 in Section 5.3.1 summarizes all the reduction operations in Green-Marl. Reduction expression can be optionally accompanied with Boolean-typed Filter expression. With each iterator value, the filter expression is computed before computing the body expression; body expression is not considered in reduction if the result of the filter expression is false. See the following code snippet as an example.

Listing 62: Examples of Count Reduction

```c
// Reduction by addition.
// summation: for all nodes of G, such that n.Color == Blue, n.A + n.B
```

Note that being free of side-effect, reduction operations can be always computed under parallel consistency (Section 5.3.1).

If the range of reduction is an empty set, or every element is filtered out the result of reduction operation is its initialization value in Table 12 in Section 5.3.1. See the following code example.

Listing 63: Examples of Empty Range

```c
nodeSet(G) S; // S is an empty set
int z2 = sum(n:S.items){n.A +n.B}; // z2 becomes 0
int z3 = max(n:S.items){n.A +n.B}; // z2 becomes -INF
```

Reduction operations can be nested. See the following code example:

Listing 64: Nested Reduction

```c
// Compute Sum of maximum of some expression.
// Note that the 'scope' of n reaches inside the nested expression.
int z2 = sum (n:G.nodes){n.A + max(m:n.nbrs){m.B + n.B}}
```

\text{count} is a syntactic sugar for \text{sum}(.)

60
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Parenthesis</td>
<td>Left to Right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reduction Operations</td>
<td>Reduction Operations</td>
<td>Left to Right</td>
</tr>
<tr>
<td>-</td>
<td>Unary Minus</td>
<td>Left to Right</td>
</tr>
<tr>
<td>!</td>
<td>Logical Negation</td>
<td>Left to Right</td>
</tr>
<tr>
<td>(type)</td>
<td>Type Cast</td>
<td>Left to Right</td>
</tr>
<tr>
<td>* / %</td>
<td>Multiplication/Division/Modulo</td>
<td>Left to Right</td>
</tr>
<tr>
<td>+ -</td>
<td>Addition/Subtraction</td>
<td>Left to Right</td>
</tr>
<tr>
<td>&gt; &gt;= &lt; &lt;=</td>
<td>Numeric Comparison</td>
<td>Left to Right</td>
</tr>
<tr>
<td>== !=</td>
<td>Numeric/Logical Comparison</td>
<td>Left to Right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical And</td>
<td>Left to Right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>Ternary Operator</td>
<td>Left to Right</td>
</tr>
</tbody>
</table>

Table 16: Operator Precedence in Green-Marl

Listing 65: Examples of Count Reduction

```c
// The following two RHS are equivalent.
int z1 = count(n:G.nodes)(n.Color == Blue);
int z2 = sum(n:G.nodes)(n.Color == Blue){1};
```

6.1.5 Operator Precedence

summarizes the operator precedence in Green-Marl. The precedence rule of Green-Marl is almost identical to that of C language.

6.2 Assignment States

6.2.1 LHS and RHS

Green Marl assignment statements have the following form. Its semantics is to modify the content of LHS location as the value of RHS.

LHS = RHS;

The LHS can be either a single scalar variable or a property location. A property location takes the following form:

driver_name . property_name

The driver name should be either a node-type variable (or iterator) or an edge-type variable (or iterator), while the property name should be a node or edge property, correspondingly. The node (edge) should be bound to the same graph as the property is. (See Section 4.6 for the related discussions).
The RHS can be any expression composed of literal, expression, scalar variable access, property access, or procedure or built-in function calls. Property access has the same syntax as property location in LHS, but reads the value of the property location.

The procedure calls included in the RHS expression should be free of side-effect and output arguments. There are two exceptions: (1) RHS procedure can have side-effects or output arguments if it is the sole element of RHS expression. (2) Procedures with ignored output arguments can be included in any RHS expression (See Section 3.3 for examples).

The type of LHS should be exactly matched with RHS. However, the compiler applies coercions between certain numeric types (See Section 4.2.2 for details).

The visibility of assignment statement under parallel consistency is discussed in Section 5.2.2.

Collective assignment is one syntax sugar where a graph name is used in place of node or edge name in property access or property location. See Section 5.5 for further discussion.

6.2.2 Reduction Assignments and Deferred Assignments

Green-Marl has two other syntax elements which takes similar form as assignment: reduction assignment and deferred assignment. Their syntax takes the following form, while their semantics are discussed in Section 5.3. The constraints in LHS and RHS are same as normal assignment.

| LHS reduce_assign_symbol RHS ; // reduction assignment |
| LHS <= RHS           // deferred assignment |

6.3 Control Statements

6.3.1 for and foreach Iteration

for and foreach are two different iteration methods provided in Green-Marl. They have following syntactic forms:

| for (iterator_name : source_name (^) . range_word) ( (filter_expr) ) body_statement |
| foreach (iterator_name : source_name . range_word) ( (filter_expr) ) body_statement |

Iterator (Section 3.2.3) is a read-only variable that points the current item of the iteration range. Source variable and range word together defines the range of iteration. The source variable can be of type Graph (Section 4.4.1), Node, Edge (Section 4.5.1), or any collection (Section 4.7.2), while each type can be followed by different range words. Note that the type of an iterator is automatically determined by the type of source and range_word.

The semantics of for and foreach iteration is to execute body sentence for every element in the range of iteration. Both kinds of iterations can be accompanied with optional,
Boolean-typed Filter expression. At each iteration instance, the filter expression is computed before executing the body statement, and the body statement is skipped if the result of expression is false.

The difference between **for** and **foreach** iteration is that, **for**-iteration assumes sequential consistency (Section 5.2.1) while **foreach** parallel consistency (Section 5.2.2).

**for** iteration on set-type source variable is unordered, i.e. it can follow any order. On the other hands for iteration on order and sequence type source variable follows its given order. The user can also enforce to follow the reverse of the given order, by putting symbol after the source name. **foreach** iteration on order and sequence type source variable loses order information. See Section 4.7.2 for the semantics of iteration on collection types.

The following code gives examples of **for** and **foreach** iteration:

```plaintext
Listing 66: Examples of for and foreach Iteration

nodeSet(G) S;  // A set of nodes of graph G
nodeOrder(G) O;  // An order of nodes of graph G
//...

// Parallel iteration on the elements of set S.
foreach (s: S. items) {
  // type of s node(G). s is read-only.
  ...
}

// Reverse iteration on an Node Order O
// Iterate only node whose color equals to BLUE
for (o: O^. items) (o. color == BLUE) {
  ...
}
```

### 6.3.2 DFS and BFS Traversal

Green-Marl allows two fundamental graph traversal methods: Depth-First Search (DFS) order traversal and Breadth-First Search (BFS) order traversal [2].

DFS traversal has the following syntax:

```plaintext
inDFS ( iterator_name : source_name (^) . nodes [ from |;] root_name ) ( ( filter_expr ) ) ( [navigator_expr] )
{ body_statement }
( inPost ((filter_expr2)) { post_visit_statement } )
```

The header syntax of DFS traversal is similar to that of **for/foreach** iteration. However, here source must be a graph type variable and root must be a node in that graph. The semantics are to traverse the graph in depth-first order from the root node and to execute the body statement at each node. Every (reachable) node is visited once and only once. The body statement is executed whenever the node is visited for the first time and prior to visiting its neighbors, i.e. pre-order visitation. When there are multiple non-visited neighbor nodes from current node, it is not defined in which order those neighbors are
visited. For example, if the graph in Figure 11 is applied to Listing 67, the result of a DFS iteration at line 6 - 8 can be \{a,b,c,e,f,d\}.

![Figure 11: Example Undirected Graph](image)

Listing 67: Example DFS Iteration using Figure 11

```plaintext
node(G) a;
nodeOrder(G) O1, O2, O3; // An order of nodes of graph G

//... // DFS order (pre-order) traversal
inDFS (s: G.nodes from a) {
    O1.push(s);
}
// O1 can be \{a, b, c, e, f, d\}

// DFS order (post-order) traversal. ';' is a short-hand for From
inDFS (s: G.nodes ; a) {} // do nothing on pre-visit
inPost {
    O2.push(s); // do something on post-visit
}
// O1 can be \{e, f, c, d, b, a\}
```

It is also possible to visit the nodes in post DFS order, by using inPost clause, as shown in line 12 - 15 of 67.

Two optional boolean expressions can be attached to DFS traversal: a *filter expression* and a *navigator expression*. As in for iteration, the filter expression is computed at the moment the node is first visited and the body statement is not executed if the value of filter expression is *false*. The navigator expression is similarly computed at the visiting moment and if the navigator evaluates to *false* the body will not be executed. Additionally if the value of the navigator expression is false, the node’s neighbors are not further considered in the traversal. See the following code for example.

---

*more accurately, its outgoing edges*
Listing 68: Example of Filters and Navigators in DFS Traversal using the graph from Figure 11

```c
node(G) a;
nodeOrder(G) O1, O2; // An order of nodes of graph G

// DFS order traversal with filters
inDFS (s: G.nodes from a)(s.color == Blue) {
    O1.push(s);
}
// O1 can be {a, b, e, f, d}

// DFS order traversal with navigators
inDFS (s: G.nodes from a) [s.color == Blue] {
    O1.push(s);
}
// O1 is { a, b, d }
```

BFS traversal has a similar syntax to DFS traversal, as shown in the following code snippet:

```c
inBFS (iterator_name : source_name (^) . nodes [from |] root_name) ( (filter_expr) ) ( [navigator_expr] )
{ body_sentence }
( inReverse ((filter_expr2)) { reverse_visit_sentence } )
```

The semantics of BFS traversal is to traverse the graph in BFS order [2]. That means a node that has a shorter hop distance from the root node is visited before the other. Green-Marl also allows reverse BFS order traversal using the `inReverse` statement. See Listing 1 for an example.

The semantics of filter and navigator for BFS traversals are the same as those for DFS traversals.

Listing 69: Example of BFS Iteration using the graph from Figure 11

```c
node(G) a;
nodeOrder(G) O1, O2; // An order of nodes of graph G

// DFS order (pre-order) traversal
inBFS (s: G.nodes from a) {
    O1.push(s);
}
// O1 can be {a, b, c, e, d, f}

inBFS (s: G.nodes ; a) {} // do nothing on pre-visit
inReverse {
    O2.push(s); // do something on post-visit
}
// O2 can be {f, d, e, b, c, a}
```

A big difference between DFS traversal and BFS traversal is their consistency model. DFS traversal assumes sequential consistency (Section 5.2.1). On the other hand BFS traversal assumes parallel consistency (Section 5.2.2). Specifically, each node with an
### 6.3.3 Other Control Structures

The syntax and semantics of other control structures in Green-Marl are quite similar to C language.

- **if** and **if-else** statements have following syntax:

```plaintext
if (bool_expr) then_statement
if (bool_expr) then_statement
else else_statement
```

When nesting **if** and **if-else** statements it may not be clear to which if-statement an else-clause belongs (also known as *dangling else* or *if-else ambiguity*). In Green-Marl, this

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentLevel() : int</td>
<td>Returns the hop-distance from the current node to the root node from where the BFS/DFS traversal started.</td>
</tr>
<tr>
<td>node.parentNode() : node</td>
<td>Returns one predecessor of the current node. There might be multiple predecessors, in this case it is undefined which one of them is returned.</td>
</tr>
<tr>
<td>node.parentEdge() : edge</td>
<td>Returns the edge leading from a predecessor node to the current node.</td>
</tr>
</tbody>
</table>

Table 17: BFS/DFS Specific Functions

equal hop distance from the root node is visited in parallel. As an example, consider a BFS traversal is applied to the graph in Figure 11 starting from node a. In this case, node a is visited first. Then nodes b and c are visited concurrently. Finally nodes d, e and f are visited concurrently.

Green-Marl introduces three functions that are specific to BFS/DFS and that cannot be used elsewhere. See Table 17 for their signatures and semantics.

See the code below for an example that uses the graph from Figure 11.

Listing 70: BFS functions example

```plaintext
node a;
inBFS (s: G.nodes from a) {
    int level = currentBFSLevel();
    node parent = s.BFSParentNode();
    println("{} - ", level, parent);
}
// One possible output could be:
// 0 - NIL
// 1 - a
// 2 - c
// 2 - b
// 2 - c
```
problem is resolved by attaching the else-clause to the closest if-statement. Programming languages like Pascal and C follow the same convention. The following example makes the ambiguity and its resolution clear.

Listing 71: Handling of Ambiguous if-else

```plaintext
// The following If-Else is ambiguous.
if (cond1)
    if (cond2) sent2();
else sent3();

// The above If-Else is same to below
if (cond1) {
    if (cond2) {
        sent2();
    } else {
        sent3();
    }
}

// Below is different from above two
if (cond1) {
    if (cond2)
        sent2();
} else
    sent3();
```

while and do-while statements have following syntax. Their semantics is same to C-language’s. while and do-while statements adopt sequential consistency.

```plaintext
while (bool_expr)
    body_statement

do body_statement
while (bool_expr);
```

6.4 Built-in Functions and Operations

There are some types which have built-in functions and operations: Graphs (Section 4.4), Nodes and Edges (Section 4.5) and Collections (Section 4.7). Functions or check-class operations can be used as a (sub-) expression since they have no side effects. Other operations should be always used as a stand-alone sentence.

Green-Marl also defines the following ground-level built-in functions. The implementation of each function is compiler-specific. Each compiler can add more built-in functions. Note that rand() is a function, as far as the language specification is concerned.

6.4.1 print and println

Green-Marl provides two special built-in functions for printing/displaying messages: print and println. print and println have the same semantics, except that println will print a line break at the end of the message.
<table>
<thead>
<tr>
<th>Signature</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniform() : double</td>
<td>returns a random number in [0; 1)</td>
</tr>
<tr>
<td>rand(long N) : long</td>
<td>returns a random number in [0; N]</td>
</tr>
<tr>
<td>log(double) : double</td>
<td>returns the natural logarithmic value</td>
</tr>
<tr>
<td>pow(double base, double exp) : double</td>
<td>returns the power function</td>
</tr>
<tr>
<td>sqrt(double) : double</td>
<td>returns the square root value</td>
</tr>
<tr>
<td>exp(double) : double</td>
<td>returns the base-e exponential value</td>
</tr>
<tr>
<td>max(int, int) : int</td>
<td>returns the larger of the two numbers</td>
</tr>
<tr>
<td>max(long, long) : long</td>
<td></td>
</tr>
<tr>
<td>max(float, float) : float</td>
<td></td>
</tr>
<tr>
<td>max(double, double) : double</td>
<td></td>
</tr>
<tr>
<td>min(int, int) : int</td>
<td>returns the smaller of the two numbers</td>
</tr>
<tr>
<td>min(long, long) : long</td>
<td></td>
</tr>
<tr>
<td>min(float, float) : float</td>
<td></td>
</tr>
<tr>
<td>min(double, double) : double</td>
<td></td>
</tr>
</tbody>
</table>

Table 18: Ground-level Built-in Functions in Green-Marl

```java
println("Hello World"); // prints 'Hello World'
```

Note that `print` and `println` do not have a fixed number of arguments. In fact `print` needs at least one argument and `println` accepts any number of arguments. The first argument has to be of type `string`, all other arguments can be of any type. The first argument will always be interpreted as the format string. The format string can define placeholders using `{}`; each placeholder will be replaced with one of the arguments. The first placeholder in the format string will be replaced with the first argument and so on. If the number of arguments is greater than the number of placeholders, the compiler might issue a warning and the surplus arguments will be ignored. If the number of arguments is smaller than the number of placeholders, the compiler might issue a warning and the surplus placeholders will not be replaced. Placeholders can be escaped - and therefore be printed as `{}` - by placing a blackslash (`\`) in front of it.

```java
node n;
nodeProperty<int> prop;
println("{} is a node and {} is a node property", n, prop);
println("this is an escaped placeholder \{\} ");
```

See the Table [19] below for the semantics of each type when it is used as argument of `print` or `println`:

The compiler will not rearrange the `print/println` calls in a way that would alter the order of messages being emitted. A compiler might provide a special flag that changes this behavior though.

Note that when using `print` inside a parallel context, then the order becomes undefined. Consider the example below, the order of 'A's and 'B's in the resulting messages is undefined.
<table>
<thead>
<tr>
<th>Type</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>prints the value of the primitive</td>
</tr>
<tr>
<td>long</td>
<td>prints the value of the primitive</td>
</tr>
<tr>
<td>float</td>
<td>prints the value of the primitive</td>
</tr>
<tr>
<td>double</td>
<td>prints the value of the primitive</td>
</tr>
<tr>
<td>bool</td>
<td>prints ‘true’ if the value is true, ‘false’ otherwise</td>
</tr>
<tr>
<td>string</td>
<td>prints the content of the string</td>
</tr>
<tr>
<td>graph</td>
<td>prints a string that identifies the object, this might</td>
</tr>
<tr>
<td>node/edge</td>
<td>depend on the target system</td>
</tr>
<tr>
<td>nodeProperty/edgeProperty</td>
<td>prints a string that identifies the object, this might</td>
</tr>
<tr>
<td>nodeSet/edgeSet</td>
<td>depend on the target system</td>
</tr>
<tr>
<td>nodeSequence/edgeSequence</td>
<td>prints a string that identifies the object, this might</td>
</tr>
<tr>
<td>nodeOrder/edgeOrder</td>
<td>depend on the target system</td>
</tr>
<tr>
<td>nodeFilter/edgeFilter</td>
<td>prints a string that identifies the object, this might</td>
</tr>
<tr>
<td></td>
<td>depend on the target system</td>
</tr>
</tbody>
</table>

Table 19: Semantics of different argument types

```java
foreach (n: G.nodes) {
    println("A");
    println("B");
}
```
7 Interaction with Application Code

7.1 Overview
As discussed in Section 1 (Figure 2), a Green-Marl program is expected to be a part of a large user application. That is, a Green-Marl program will be compiled (i.e. translated) into equivalent codes in target language whilst each Green-Marl entry procedure becomes a callable routine (e.g. C++ function or java method) in the target language. Those entry routines are expected to be invoked by the user application.

Therefore there must be a 1-1 type correspondence between Green-Marl type and types in the target language for the arguments of the entry function. The compiler should specify such correspondence for each target language that it supports. If a certain Green-Marl type is implemented as a library class (e.g. graph), the compiler should provide the library to the application as well. The following code provides an example mapping for C++ target.

Listing 72: Green Marl Procedure Signature
```
procedure Foo(G: graph, n: node(G), nodeProp<int>(G) A; t: float): bool {
  ...
}
```

Listing 73: Generated C++ Function Signature
```
bool Foo(gm_graph & G, int64_t n, int [] G_A, float & t)
{
  ...
}
```

When invoking Green-Marl entry procedure from the application, the user should ensure below invariant; the execution behavior is undefined otherwise.

- The Green-Marl entry procedure must be invoked under (virtually) sequential context (Section 3.1.1).

- There should be no aliases between graphs, collections and properties in the argument lists (Section 3.1.3).

7.2 Embedding Foreign Syntax
The syntax set of Green-Marl is specially designed for graph related data processing only. However, there can be cases when the user may want to use certain or library calls of the target language even during graph data processing. For such cases, Green-Marl allows embedding of foreign syntax in Green-Marl program. This section explains such mechanism in Green-Marl.
7.2.1 Foreign Types

Green-Marl allows including foreign types in Green-Marl program. A foreign-type is any name followed by a $ symbol, as shown in 74. Foreign type variables can be handed as arguments or be declared inside a procedure (line 1, line 3). Assignment between foreign typed objects is allowed (line 4); a Green-Marl compiler simply regards all the foreign types being compatible with each other. The compiler, however, may have an option to set all the foreign types not compatible with each other; in that case line 4 of 74 is an error.

Listing 74: Foreign Types in Green-Marl

```plaintext
procedure Foo (G: graph, T: $UDT1) {
    $UDT2 X; // The user can declare of foreign-type variable
    X = T; // compiler just believes every foreign type is equivalent
}
```

A Green-Marl compiler simply keeps the original name string (after $ mark) and use the string in place of type name in the generated code; the type rule of the target language will be enforced by the target language compiler.

7.2.2 Foreign Expressions

Green-Marl allows embedding of foreign syntax in place of any expression. Note that, in Green-Marl, expressions are always free of side-effects. Therefore the user should ensure foreign expression has no side-effect, either; otherwise the behavior of those side-effect is undefined. Foreign expression is denoted by any string inside [], in place of any Green-Marl expression. As an example, the string inside [] in line 4 of Code 76 is a foreign expression.

A Green-Marl compiler keeps such string and put it as-is in the generated code except Green-Marl variables used inside the foreign expression; such Green-Marl variables are denoted by names followed after $ symbol inside the foreign expression. In 75 $n.B and $T are the Green-Marl variables used in foreign expression. During target code generation, those Green-Marl variables are translated the same ways as normal Green-Marl variable accesses. Also since Green-Marl knows which variables are being read, it can prevent their anti-dependent statements from being reordered prior to the foreign expression.

Listing 75: Foreign Expressions in Green-Marl

```plaintext
procedure Foo (G: graph; A,B: N_P<float>(G), T: $UDT1) {
    foreach (n: G.nodes) {
        n.A = n.A + [log($n.B) + $T->getValue()] * 0.5;
        n.B = n.B + 1; // compiler should not re-order line 5 and line 4
    }
}
```

For example, if properties are mapped into arrays and node into integer, n.B would be translated into B[n].

71
A Green-Marl compiler regards the result type of the foreign expression being compatible with any type. The type rule in the generated target code will be eventually enforced by the target language compiler.

### 7.2.3 Foreign Statements

Foreign sentences in Green-Marl are similar to foreign expressions except that foreign sentences can have side effects. In Green-Marl foreign sentences are any string inside [], in place of Green-Marl sentences.

As an instance, line 10 - 13 inside [] are foreign sentences. The compiler keeps the string as-is in the generated code, except Green-Marl variables inside the string; Green-Marl variables are denoted by $ in the string. Foreign sentences can be optionally followed by the list of variables modified inside the foreign sentence. Line 14 and 22 in [] are examples of it.

#### Listing 76: Foreign Statements in Green-Marl

```c
int y,x,z;
x = 0;
while (x <= 10) {
    // statements inside [] are foreign statements.
    // $x, $y, $z are Green-Marl variable accesses.
    // [y,z] are modified variables.
    // :: can be replaced with a keyword Mutating
    if (x == 0)
        [printf("This is the first execution:%d", $x);
         FILE *f = fopen("data_file", "r"); assert(f!= NULL);
         $z = fscanf(f, "%d", &$y);
         fclose(f);
        ] :: [y,z]
    else {
        y = y + 1;
        x = x + 1;
    }
}
```

### 7.2.4 Foreign Functions

### 7.2.5 Target Header Include

#### Listing 77: Inclusion of Target Header File

```
// compiler will generate appropriate include syntax at the beginning of
// the generated file. String inside <> will be simply copied.
#include<Java.utilRand.*>;
```
procedure Foo() {...}
8 Green-Marl Code Examples

This section shows a few popular graph algorithms written in Green-Marl. Note that Code 1 in Section 1.2 is another good example of graph algorithm written in Green-Marl.

Listing 78: Conductance in Green-Marl

```green-marl
proc conductance (G: graph, member: N_P<int>(G), num: int) : float
{
    int Din, Dout, Cross;
    Din = sum(u:G.nodes)(u.member == num){u.degree()};
    Dout = sum(u:G.nodes)(u.member != num){u.degree()};
    Cross = sum(u:G.nodes)(u.member == num){count(w:u.nbrs)(w.member != num)};
    float m = ((Din < Dout) ? Din : Dout);
    if (m == 0) return (Cross == 0) ? 0.0 : +INF;
    else return Cross / m;
}
```

78 is Green-Marl program to compute Conductance of a Sub-graph (more accurately, of a cut). Note that a sub-graph is represented as a membership property of a node. The program is very close to the definition of the conductance; it counts the number of edges inside the sub-graph, outside the sub-graph, and the crossing edges and computes the value. Note that a GM compiler can first translate three summations in line 4 to 6 into three \texttt{foreach} loops and then merge them together into one loop.

Listing 79: PageRank in Green-Marl

```green-marl
proc pagerank (G: graph, e, d: double, max: int; pg_rank: N_P<double>(G))
{
    double diff;
    double N = (double) G.numNodes();
    G.pg_rank = 1 / N;
    do {
        diff = 0.0;
        foreach(t: G.nodes) {
            double val = (1-d) / N + d * sum(w: t.inNbrs)(w.outDegree()>0) {w.pg_rank / w.outDegree();}
            diff += | val - t.pg_rank |;
            t.pg_rank <= val;
        }
        cnt ++;
    } while ((diff > e) && (cnt < max));
}
```

79 is a Green-Marl program that computes page-rank of a graph. Again Green-Marl program closely resembles the definition of page-rank algorithm. Note that at line 12, pg_rank is being defer-assigned and there is no read-write data race between line 9 and line 12.
Listing 80: Connected Components in Green-Marl

```
proc CC(G: Graph; membership: N_P<int>(G)) : int
{
    int numC = 0;
    G.membership = -1;
    // Sequential Iteration of Nodes
    // Visit nodes that have not included in any component yet.
    for (s: G.items) (s.membership == -1) {
        // Do BFS and mark all reachable nodes
        inBFS(t: G.nodes from s) {
            t.membership = numC;
        }
        numC ++;
    }
    return numC;
}
```

Listing 81: Strongly Connected Components (Kosaraju’s Algorithm) in Green-Marl

```
proc SCC_kosaraju(G: graph; membership: N_P<int>(G)) : int
{
    int numC = 0;
    nodeOrder(G) P;
    G.membership = -1;
    G.filter
    // Obtain post DFS order
    for (s: G.nodes) (! P.has(s)) {
        inDFS(t: G.nodes from s)
        inPost { P.push(t);}
    }
    // In reverse, post DFS order
    for (s: P^- .items) (s.membership == -1) {
        inBFS(t: G^- .nodes from s) [t.membership== -1] {
            t.membership = numC;
        }
        numC ++;
    }
    return numC;
}
```

[80] is a Green-Marl program that obtains connected components[2] of an undirected graph. The idea is to perform BFS traversal from any unmarked node and mark every visited node; all the newly marked nodes belong to the same component. And this BFS is repeated from each unmarked nodes, it until every node is marked.

Listing 81: Strongly Connected Components (Kosaraju’s Algorithm) in Green-Marl

Unlikely to the case of connected components in undirected graphs, obtaining strongly connected graphs in undirected graphs requires a little more computation; [81] computes strongly connected component, using Kosaraju’s Algorithm[2]. This algorithm first obtains post-DFS order for all the nodes of a graph (line 7-10). Then it iterates the nodes in reverse of that post-DFS order (line 11) and performs a BFS traversal on a ‘transposed’
graph of G (line 12). Transposed graph means the graph where the direction of each edge has been reversed, and is denoted by $G^\sim$. Also this BFS traversal only goes through the nodes that have not been marked yet (line 12).
9 Ideas for Future Versions

9.1 Multi-Set

[Idea] Multi-set is a collection that is non-unique and non-ordered. The missing 4th element

[Issue] Not Much

[Syntax Suggestion]

nodeMultiset(G) M1;
N_M(G) M2;
edgeMultiset(G) M3;
E_M(G) M4;
// same operations as set.

9.2 Multiple File Compilation

[Idea] Able to write source codes in multiple files

[Issue] Green-Marl specification demands inter-procedural analysis so that it can figure out data-races. However, if a procedure calls another procedure whose body is not available, such an inter-procedural becomes hard. We can store some information about which of the arguments can be potentially mutated and how.

9.3 Continue, Break

[Idea] Continue and Break Statement

[Issue] Continue => Stop executing current instance; fits well with parallel consistency.

Break => Stop executing all execution instances of current loop; semantics becomes non-deterministic with parallel consistency. (Other loop instances may or may not stop.)

9.4 Named Continue Break

[Idea] Continue and Break Statement for nested loops.

[Syntax Suggestion]

for (t: G.nodes) {
  for (s: t.nbrs) {
    for (r: s.nbrs) {
      if (r.condition)
        continue @ t;
      S1();
    }
    S2();
  }
  S3();
}

[Issue] Under sequential consistency, the semantics is very clear. (Continue at t means to skip all the remaining instances of r and s and begin new iteration with another t.) But what is its semantics under parallel consistency? (i.e. when for->foreach?)
9.5 Named While Loop

[Idea] Giving name indicator to while loop so that it can be used with @ syntax.

[Syntax Suggestion]

```java
while (NAME1: z < 10) {
    y += 3 @ NAME1;
    z = z + 1;
}
```

[Issue] In case of do-while, the declaration of name comes 'after' its use. Does this 'NAME' reside in the same name space as variable names?

9.6 Acknowledged Conflicting Writes

[Idea] "I know that this write will conflict with some other. But it is harmless. So shut up and stop whining, you stupid compiler."

[Syntax Suggestion]

```java
foreach (s: G. nodes) {
    foreach (t: s. nbrs) {
        if (somecondition (G, s, t))
            t. val #= s. val; // User knows it is okay to allow this conflict.
    }
} // There would be #= and #<=
```

9.7 Visibility Control for Collective Types

[Idea] Just like reduction has its bound, operations on collection may do the same as well. "I will grow this set during this loop."

[Syntax Suggestion]

```java
nodeSet(G) S;
foreach (s: G. nodes) {
    foreach (t: s. nbrs) {
        if (somecondition (G, s, t))
            S. add(t) @ s; // add t to s. S is being grown up during s-loop.
    }
} // all the addition to set S can happen here, simultaneously.
```

9.8 Neighborhood Marker

[Idea] Instead of actually go to neighbor and see, mark the information locally.

[Syntax Suggestion]

```java
neighborSet(G) NS;
foreach (t: G. nodes) {
    foreach (s: t. nbrs) {
```
if (s.Color == Blue)
    t->NS.add(s); // Mark Blue Neighbors
}

foreach (t: G.nodes) {
    foreach (s: t->NS) { // Iterate Blue Nodes only. Saves execution time.
        s.val = 0;
    }
}

[Issue] This is only for performance optimization.

**Re-Ordered Iteration**  [Idea] Instead of iterating a collection with the natural order, let the user iterate it any specified order.

[Syntax Suggestion]

```java
nodeOrder(G) O;
nodeSet(G) S;

for (o: O.items)
inOrder(o.A+o.B) {// iterate elements of O as increasing order of o.A + o.B
    // do something in that order
    // if the value of o.A or o.B is being modified inside the loop,
    // the result is undefined. Or an error?
}

for (o: S.items)
deOrder(o.A+o.B) {// iterate elements of O as decreasing order of o.A + o.B
    // ...
}
```

**9.9 Print Statement**

[Idea] Print something. Compiler guarantees each print does not mixed up with other prints. Where-to-print is adjustable by compiler/runtime.

[Syntax Suggestion]

```java
foreach (n: G.nodes) {
    print "Hello From node " :: n :: "."; // prints are not mixed-up.
}
```

**9.10 Error Statement**

[Idea] Stop execution and give control back to the user application. In a parallel region, other instances may continue to execute. The exact mechanism of error handling is compiler specific. (e.g. Java Exception. Or just a Error Flag, ...)

[Syntax Suggestion]
foreach (n: G.nodes) {
    if (n.something_is_wrong)
        error "Something is Wrong";
}
References


