The Green-Marl Language Specification

Version 0.3.0

Last modification: Oct 11, 2011

Revision History

Version	Modifications
0.1	Initial concepts
0.2	Collection types and their initialization, Distinction of function and procedure, Reduction Assignment
0.3	A version that matches ASPLOS Paper

Authors

Version	Authors
0.1	Sungpack Hong (Stanford Univ.)
0.2	Sungpack Hong (Stanford Univ.)
0.3	Sungpack Hong (Stanford Univ.)

Reviewers

Version	Reviewers
0.1	
0.2	Nathan Bronson (Stanford Univ.)
0.3	

Table of Contents

1 Introduction

1.1 The 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:

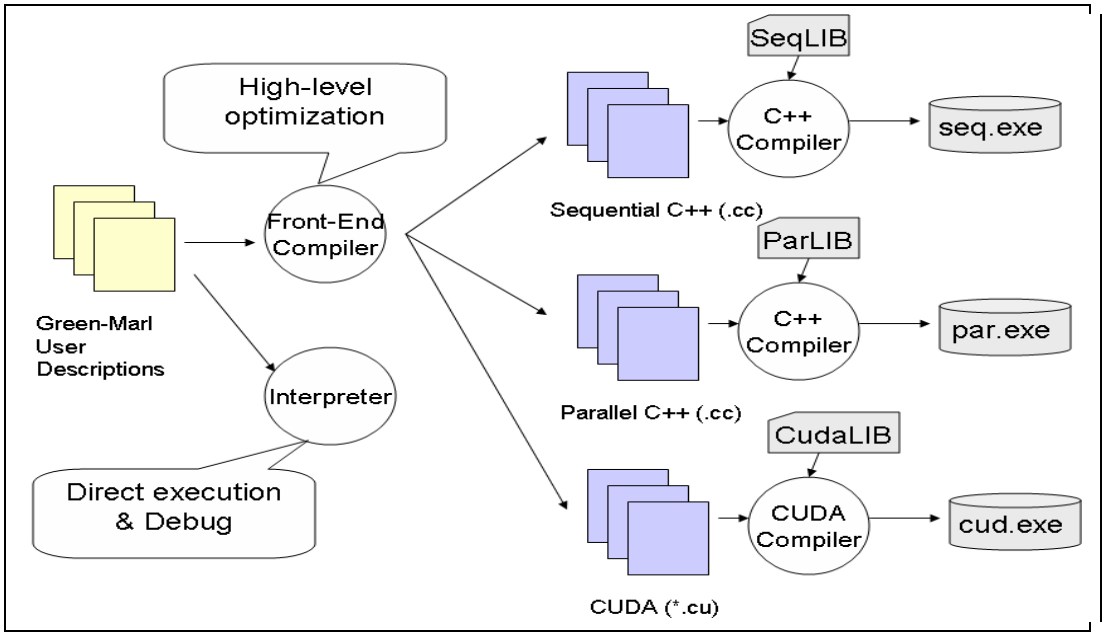


Figure 1 Suggested Usage of Green-Marl DSL

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

[1] *Green-Marl* is a transliteration of two Korean words: 그린 말(*depicted language)*

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 (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 semantic 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 [1].

```
Procedure compute_bc(G: Graph,     // G is an (Directed) Graph
   bc: Node_Property<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
   Node_Property<Float>(G) delta;       // Declare new node properties,
   Node_Property<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 its
                 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 @ r; // delta is accumulated into BC over r-loop.
    }
  }
}
```

Code 1 Betweenness Centrality computation in Green-Marl

Note that the original paper [1] 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 semantic 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. [2] The following figure illustrates this idea. Please refer Section 7 for details about how Green-Marl program can interact with other part of the application.

[2] A Green-Marl compiler is recommended to make its output fairly human-readable.

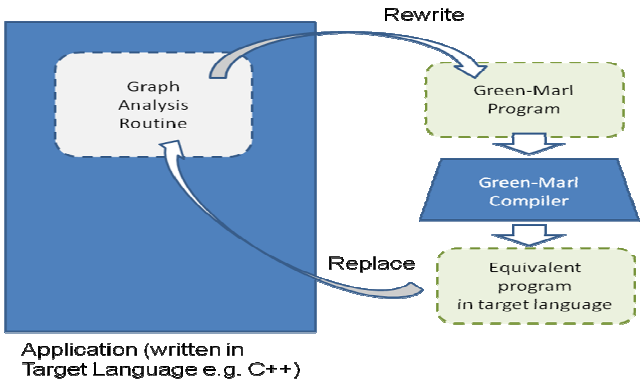


Figure 2 Interaction of Green-Marl program with user application

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 semantic 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.[3]

```
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;
}  }
```

```
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;
}  }
```

Code 2 Example compiler optimization in Green-Marl

- **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 Code 1. See Section 6.3 for more about graph iteration methods

- **Node / Edge Property:** In Green-Marl, the nodes and edges of a graph can be

[3] Suppose 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.

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 4.5 for more about the properties.

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 the 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. 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, +INF or –INF. (See discussion about Integer Type in Section XX for the valid integer value range). Also note that +INF and –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 mark. C escape character rules are applied as same. Unicode string is not (yet) supported in Green-Marl. <todo>

- **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.

- Bold means literals or reserved words
- {}+ means 1 or more repetition, {}* means 0 or more repetition
- [] means optional
- [a | b | c] means a choice among a, b, or c.

```
unit          => {toplevel}+
toplevel      => proc_def
proc_def      => proc_header sent_block

proc_header   => [ Procedure | Proc | Local ] name ( [arg_list] [; outarg_list] ) [:
return_type]
arg_list      => arg_name : type_prop {, name : arg_type_prop }+
outarg_list => arg_list

type_prop     => type | property
type          => prim_type  | graph_type | nodeedge_type |
                 graph_collection_type
return_type   => prim_type | nodedge_type | graph_collection_type | property
prim_type     => Int | Long | Float | Double | String | Boolean
graph_type    => Graph | DGraph | UGraph
                 <Todo: subgraph types>
nodeedge_type => Node ( graph_name )
                 Edge ( graph_name )
graph_collection_type =>
              [Node_Set | N_S] < prim_type > ( graph_name )
              [Edge_Set | E_S] < prim_type > ( graph_name )
              [Node_Order | N_O] < prim_type > ( graph_name )
              [Edge_Order | E_O] < prim_type > ( graph_name )
              [Node_Seq | N_Q] < prim_type > ( graph_name )
              [Edge_Seq | E_Q] < prim_type > ( graph_name )

property   =>
             [Node_Property | Node_Prop | N_P] < prim_type > ( graph_name )
             [Edge_Property | Edge_Prop | E_P] < prim_type > ( graph_name )
```

```
graph_name => name
arg_name   => name
var_name   => name
node_name  => name
node_edge_name => name
iter_name  => name
property_name => name
collection_name => name
name       => identifier
name_comma_list  => name {, name}*


sentence       => sent_block
                | decl_sentence
                | normal_assignment
                | defer_or_reduce_assignment
                | reduction_assignment
                | for_iteration
                | foreach_iteration
                | bfs_iteration
                | dfs_iteration
                | if_then_else
                | while_sent
                | do_while_sent
                | built_in_call_sent
                | proc_call  // red: to be implemented in next versions
                | print_sent
                | error_sent

sent_block   => { {sentence}* }

decl_sentence   => type_prop name_comma_list ;


normal_assignment  => lhs = rhs ;
defer_or_reduce_assignment  => lhs <= rhs [@ iter_name];
             |  lhs reduce_assign rhs [@ iter_name];
             |  lhs_opt_list reduce_assign2 rhs_opt_list [@ iter_name];
reduce_assign => += | *= | &&= | ||=
reduce_assgin2 => max= | min=
lhs_opt_list => lhs
              | lhs < {, lhs}+ >
```

```
rhs_opt_list => rhs
              | rhs < {, lhs}+ >


lhs           => var_name | var_name . property_name
rhs           => expr | bool_exp


literal => integer_literal | floating_literal | string_literal
expr           => literal
                | var_name | var_name.property_name
                | built_in_call
                |  ( expr )
                |  | expr |
                |  ( prim_type ) expr
                |  - expr
                |  reduc_epxr
                |  expr biop expr
                |  bool_exp ? exp : exp
biop          =>  * | / | % | + | -
bool_exp      => boolean_literal
                | var_name | var_name.property_name
                | built_in_call
                | (bool_exp)
                |  ! bool_exp
                |  bool_exp && bool_exp
                |  bool_exp ||  bool_exp
                |  expr comp_op expr
                |  bool_exp bool_comp bool_exp
comp_op     => == | > | < | >= | <= | !=
bool_comp   => && | || | == | !=

reduc_expr    => reduc_type iterator_bound [filter] { expr }
reduc_type    => Sum | Product | Max | Min | Any | All

iterator_bound   => ( iter_name : graph_name . range_word1 )
                 |  ( iter_name : node_edge_name . range_word2 )
                 |  ( iter_name : collection_name . Items )
filter        => ( bool_expr )
range_word1    => Nodes | Edges
range_word2    => Nbrs | InNbrs | OutNbrs | UpNbrs | DownNbrs
              NbrEdges | InEdges | OutEdges | UpEdges | DownEdges
```

```
for_iteration => for iterator_bound [filter] sentence
foreach_iteration => foreach iterator_bound [filter] sentence

bfs_iteration => InBFS iterator_bound2 [filter] [navigator] sentence
                 [ reverse_iteration ]
dfs_iteration => InDFS iterator_bound2 [filter] [navigator] sentence
                 [ reverse_iteration ]
reverse_iteration => InReverse [filter] sentence
navigator => [ bool_expr ]
iterator_bound2  => ( iter_name : graph_name [^] . Nodes [From | ;] node_name )

if_then_else => If (bool_exp) sentence
              | If (bool_exp) sentence Else sentence
while_sent => While (expr) sentence
do_while_sent => Do sentence While (expr) ;

built_in_call_sent => built_in_call ;
built_in_call => built_in_name ( [rhs_comma_list] )
               |  graph_name . built_in_name ( [rhs_comma_list] )
               |  node_dege_name . built_in_name ( [rhs_comma_list] )
               |  collection_name . built_in_name ( [rhs_comma_list] )

rhs_comma_list => rhs {, rhs}*
proc_call   => call_name ( [rhs_comma_list] [; lhs_comma_list] )
lhs_comma_list => (lhs|#) {, (lhs|#)}*

print_sent => Print expr;
error_sent => Error expr;
```

3 Language Entities

3.1 Procedures

3.1.1 Entry Procedures and Local 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`**.

An entry 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). 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 wholly available.

On the other hand, a local procedure is what other Green-Marl procedures can call upon. Local procedure can be called inside a parallel region (Section 5.1) as well. A compiler must do inter-procedural analysis including such local procedure calls, when it validates parallel consistency semantics (Section 5.2.2).

Currently, Green-Marl only allows calling of local procedures that are defined in the same file. However, the user application, which is written in a different file (in a different language), can make calls to any Green-Marl entry functions -- these functions can be defined in separate Green-Marl source files.

A Green-Marl procedure can make a call to another entry procedure, as long as virtually sequential context is guaranteed at the call-site; however, a compiler may require true sequential context in this case as well.

In current Green-Marl specification, recursion is prohibited. [todo: Or is it current compiler implementation that prohibits it?]

```
Local compute_dist(x1, y1, x2, y2:Int): Int    // Local function
{
   Return (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2);
}

Procedure get_max_nbr_dist (                    // Entry function
    G: Graph, S: Node_Set(G), X,Y:Node_Propery<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
                                   // compute distance and reduce by maximum
         max_dist max= compute_dist(s.X, s.Y, n.S, n.Y);
   }
   Return max_dist;
}
```

Code 3 Local Procedure and Entry Procedure

3.1.2 Arguments and Return Values

A Green-Marl procedure has two different 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 Code 4 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 such 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.

```
Local get_max_min(a,b : Int; min: Int) : Int    // min is an output argument
{
    If (a < b) {min = a; Return b;}
    Else       {min = b; Return a;}
}
```

Code 4 Output arguments

	Input	Output/Return
Primitive	By Value	By Value
Node/Edge	By Value	By Value
Property	By Reference	Not available
Graph	By Reference	Not available
Collection	By Reference	Not available <todo: why?>

Table 1 Argument passing convention in Green-Marl procedures.

Table 1 summarizes the argument passing convention in Green-Marl. The argument passing convention is determined by the type of an argument. (See Section 4 for type system in Green-Marl). If the type of an argument is primitive type or node-edge type, the argument is always passed by value; no matter whether the argument is an input or an output. On the other hand, graph, property, and collection type cannot be an output argument (or return value) but always be passed by reference.

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.

However, when a Green-Marl procedure makes a call to another Green-Marl procedure, the Green-Marl compiler should examine each call-site and check if there are any aliases in the calling arguments. If an alias is detected, the compile must give an error. If the compiler believes that an alias could happen at the runtime but cannot be decided at the compile time,

the compiler should give a warning to the user. It is undefined the behavior of a Green-Marl procedure execution when there are aliases in the input arguments.

The behavior of a procedure-call which contains aliases in the output arguments (i.e. same lhs values) is also undefined. The compiler should check this condition and give an error if violated as well.

```
// X and Y are considered to be distinct
Local foo(G: Graph, X,Y: Node_Prop<Int>(G); a,b: INT)
{
  // …
}
Local bar(a,b: INT)
{
  // …
}
Procedure bar(G: Graph)
{
   Node_Prop<Int>(G) X;
   Int y;
   foo (G, X, X; y, y); //  Error - X, X is an alias in input arguments
                        //  Error - y, y is an alias in output arguments
   bar (y, y); // okay - no alias is possible between primitive-types
}
```

Code 5 Green-Marl assumes no alias in procedure arguments

3.2 Variables

3.2.1 Scoping rule

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.

```
   // Arguments names are only meaningful in the procedure.
Procedure foo(G: Graph, x,y: Int, A:Node_Prop<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.
   }
}
```

Code 6 Scope of Green-Marl variables

3.2.2 Initial values

When a variable of a primitive type (Section 4.2) or Node/Edge type (Section 4.4) 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 4.5) is declared, the property value at each node (edge) is undefined.

However, when a collection-type (Section 4.6) variable is declared, an empty collection is automatically created.

Graph-type variable (Section 4.3) cannot be declared inside a Green-Marl procedure: it should be only passed through an input argument to the entry procedure.

```
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.

   Node_Prop<Int>(G) A; // property type variable A is declared
   Z = n.A;             // value of A for each node in graph G is undefined

   Node_Set(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.
}
```

Code 7 Initial values

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 Code 8).

Iterators are read-only and thus cannot be written.

```
Proc foo(G: Graph, A: Node_Prop<Int>(G)) {
   Node(G) m;
   // n is an iterator for the foreach statement
   // the scope of n is the filter (n.A >0) and the body {n.A = n.A + 1; …}
   Foreach(n: G.Nodes)(n.A > 0) {
        n.A = n.A + 1;
        m = n; // This is okay
        n = m; // This is an error. An iterator is read-only.
   }
}
```

Code 8 Iterator Scope Example

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 types means to make a copy of the contents of RHS collection into LHS one. The original contents in LHS collection are lost. (Section 4.6)

```
Proc foo(G: Graph, A: Node_Prop<Int>(G), b: Int, S: Node_Set(G)) {
   Node_Prop<Int>(G) B = A; // Error. Property itself is read-only.
   Int c = b; // okay
   Node_Set(G) P = S; // make a copy of S.
}
```

Code 9 Assignment Example

3.3 Sentences and Expressions

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

The only meaningful entity where this distinction makes actual difference is procedure call:

- If a procedure has a (potential) side-effect, it cannot be called inside an expression.
- If a procedure has output arguments, it can be called inside an expression only if all the output arguments are ignored by using # syntax (Section 6.4).
- As a special case, a call to a potentially side-effecting procedure can be placed at the RHS of an assignment only if the call is the sole element in the RHS expression.

```
Local side_effect(G: Graph, A: Node_Set(G)): Int  // has side effect
{
   A.Clear();     // making a side effect
   Return 3;
}
Local out_arg(A: Int; B: Int): Int    // no side effect
{
   B = 3;         // output argument
   Return A + B;
}
Proc example(G: Graph, A: Node_Set(G)
{
   Int x,y,z;
   side_effect(G, A); // okay;
   z = side_effect(G, A); // okay - there is no other expression
   z = side_effect(G, A) + 3; // Error

   out_arg(x; y);  // okay
   z = out_arg(x; y); // okay
   z = out_arg(x; y) + 1;  // error
   z = out_arg(x; #) + 1;  // okay - output has been "ignored"
}
```

Code 10 Procedure Call in Expressions Example

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.

```
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;
}
```

Code 11 Example of possible optimization of a procedure call

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` / `Double`.

Green-Marl Int is 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 0xFFFFFFFF and 0x7FFFFFFF.

Similarly, Green-Marl Long is 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 +INT and –INT which are compatible to any numeric type. However, it gives an undefined result, doing any numeric operation other than comparison to +INF/-INF (Code 12).

```
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: Node_Prop<Int>(G)
Int y = +INF
Foreach (n: G.Nodes) {
  y min= n.A;  // compute minimum among n.A
}
```

Code 12 Example use of +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 Code 13 for example.

RHS (one operand)	LHS (the other operand)	Coercion	Compiler Action
Int	Long	Int → Long	Implicit Conversion
Int	Float	Int → Float	Implicit Conversion with warning
Long	Float	Long → Float	Implicit Conversion with warning
Long	Double	Long → Double	Implicit Conversion with warning
Float	Double	Float→ Double	Implicit Conversion

Table 2 Coercion Rules in Green-Marl

```
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.
```

Code 13 Coercion and Explicit Type Conversion Example

4.2.3 Boolean Type

Boolean type variable can have only one of two values: True and False.

4.2.4 String Type

String type will be added in a later version of the language. Note that strings are immutable values like integer.

4.3 Graph Types

There are two types of graphs in Green-Marl: `DGraph`, which stands for directed graph, and `UGraph`, or undirected graph. `Graph` is a synonym to `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.

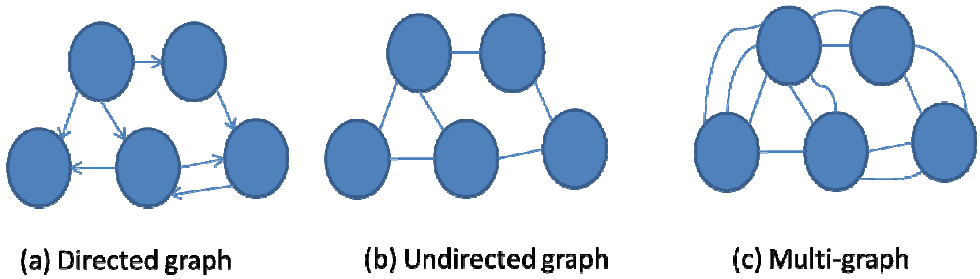


Figure 3 Illustration of graph types in Green-Marl.

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.

```
Procedure foo(G1, G2: Graph) { // Graph can be an input argument
    Graph G3;  // Error- Graph cannot be defined
    G2 = G1;   // Error- Graph cannot be modified (assigned)
}
```

Code 14 Example Use of Graph Type

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 [4]) 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.

4.3.1 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.

Source Type	Range Word	Meaning
`UGraph, DGraph`	`Nodes`	Every node of the graph
	`Edges`	Every edge of the graph

Table 3 Range words of Graph Types

The below table summarizes built-in functions (Section 6.4) for graph types. A compiler implementation may add other built-in functions.

Source Type	Signature	Meaning
`UGraph, DGraph`	`NumNodes():Int`	Number of nodes in the graph
	`NumEdges():Int`	Number of edges in the graph

Table 4 Built-in Functions of Graph Types

4.4 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 type.[4]

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.

```
// 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
}
```

Code 15 Example Use of Graph Type

4.4.1 Range Words and Built-in Functions

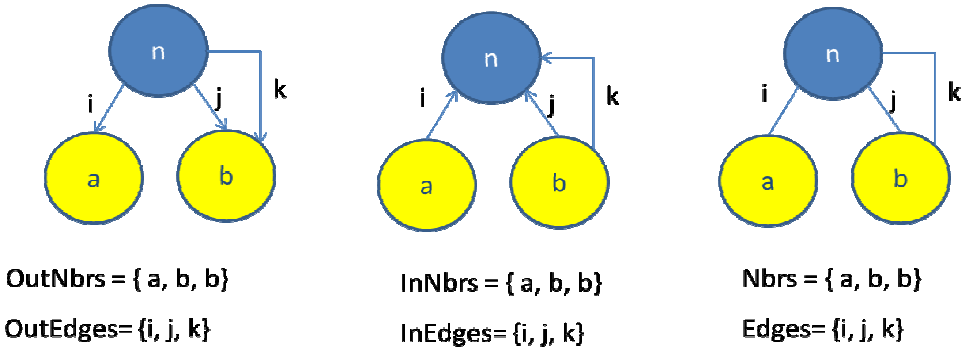
Range word is a syntax that delineates the range of For-iteration and For-each iteration (Section 6.3.1). Table 5 summarizes range words available to node type and edge type; the meaning of each range word is illustrated in Figure 4. There are few things to be noticed

- 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/Out Nbrs 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 XXX). 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 4.

[4] A compiler can still represent nodes and edges as long or integer in implementation.

Source Type	Range Word	Misc
`Node(DGraph)`	`Nbrs, OutNbrs`	Nbrs is a synonym to outNbrs
	`Edges, OutEdges`	Edges is a synonym to outEdges
	`InNbrs`	Require reverse edges
	`InEdges`	
`Node(UGraph)`	`Nbrs`	
	`Edges`	
`Node(DGraph/UGraph)`	`UpNbrs`	Only available during BFS traverse
	`UpEdges`	
	`DownNbrs`	
	`DownEdges`	

Table 5 Range words of Node Type



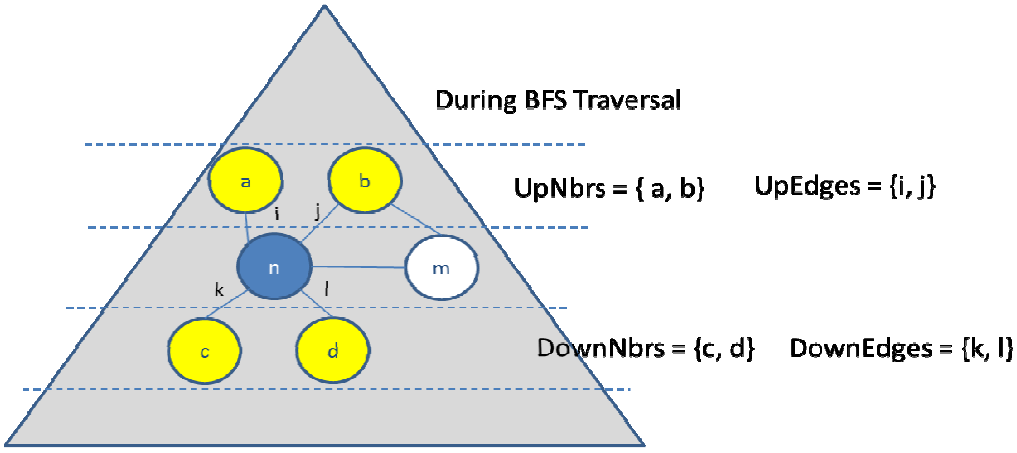


Figure 4 The Illustration of Semantics of Range Words of Node Type

The below table summarizes built-in functions (Section XX) of node types. A compiler implementation may add other built-in functions.

Source Type	Signature	Meaning
Node(DGraph)	NumNbrs(): Int NumOutNbrs(): Int Degree(): Int OutDegree(): Int	Size of the out-neighbor multi-set.
	NumInNbrs(): Int InDegree(): Int	Size of the in-neighbor multi-set.
Node(UGraph)	NumNbrs(): Int Degree(): Int	Size of the out-neighbor multi-set.

Table 6 Built-in Functions of Node Types

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.

```
// Graph G1, G2
// A: Node_Prop<Int>(G)
Node(G1) n;
Node(G2) m;
n = NIL; // okay
m = NIL; // okay
n = m ; // Error
Int x = n.A; // undefined
```

Code 16 NIL value for node/edge type

4.5 Property Types

Although Green-Marl does not support user-defined type, the user can till associate nodes and edges with any number of primitive-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.

- Property declaration syntax is as follows:

  `Node_Prop<target_type>(graph_name)  property_name`

- Properties can be declared just like normal variable and has static scope (Section 3.2.1); see Code 17.
- Currently, only primitive types can be a target type of node property.
- 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 Code 18.

```
Local foo(G: Graph, A: Node_Prop<Int>(G)) {
  Node_Prop<Int>(G) B; // this property is alive only inside the procedure
  Foreach (n: G.Nodes) {
      Node_Prop<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}
  }
}
```

Code 17 Static Scoping Rule of Property Declaration

```
Local foo(G1,G2: Graph, A: Node_Prop<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
}
```

Code 18 Type Rule of Property Declaration

Followings are additional notes for property types.

- Being syntax sugars, `Node_Prop` and `N_P` is a synonym to `Node_Property`; `Edge_Prop` and `E_P` to `Edge_Property`.
- 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)

<todo: Node_Prop<Node>(G) >

<todo: Neighborhood Marker>

4.6 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 7. Noticeably, the difference between each collection type comes from ordered-ness and uniqueness. For example, elements in a set are un-ordered and unique, while elements in a sequence are ordered and can be repeated.

Collection Name	Ordered-ness	Uniqueness
`Node_Set (N_S)` `Edge_Set (E_S)`	N	Y
`Node_Order (N_O)` `Edge_Order (E_O)`	Y	Y
`Node_Seq (N_Q)` `Edge_Seq (E_Q)`	Y	N
Multiset for Node/Edge	N	N

Table 7 Collections in Green-Marl

Following are notes for collection types.

- Collections are bound to a graph, similar to Property Types. (Section 4.5)
- 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)
- The semantic of an assignment in collection type is to create a copy.
- Comparison operator between collections is not defined. <todo: why? Because it is expensive?>

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

```
// Assume a,b,c are distinct node. (They don't have to be by the syntax)
Local foo(G1,G2: Graph, a,b,c: Node(G1), d: Node(G2)) {
  Node_Set(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 Node_Set
  N_S(G2) S3;  // N_S is a synonym to Node_Set

  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 Seqeunce
  Node_Order(G1) O1;
  Node_Seq(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}
}
```

Code 19 Declaration and Assignment of Collection Types

4.6.1 Built-in Operations Collection Types

The Table 8 and Table 9 summarize built-in operations of Collection Types in Green-Marl.

Operation	Class	Semantic
`Has(Node) : Bool`	Check	Returns True if the set contains the node
`Size() : Int`	Check	Returns the number of elements in the current set.
`Add(Node)`	Append	Add a node to the set
`Add(N_S)`	Append	Add all the nodes in the argument set to the current set.
`Remove(Node)`	Remove	Remove the given node from the set, if the node is in the current set.
`Remove(N_S)`	Remove	Remove all the nodes in the argument set from the current set.
`Clear()`	Remove	Remove all the nodes in the current set

Table 8 Built-in Operations of Node_Set (Edge_Set)

Operation	Class	Semantic
`Has(Node)`	Check	Returns True if the collection contains the node
`Size(): Int`	Check	Returns the number of elements in the set
`Front(): Node`	Check	Returns the front node of the order. <todo: Returns NIL if empty>
`Back(): Node`	Check	Returns the back node of the order. <todo: Returns NIL if empty>
`PushBack (Node)` `Push(Node)`	Append	Add the node at the back of the order. Push is a synonym for PushBack.
`PushBack(N_O)`	Append	Add all the elements in the argument at the back of the order. (Code 20)
`PushFront (Node)`	Append	Add the node at the front of the order.
`PushFront (N_O)`	Append	Add all the elements in the argument at the front of the order. (Code 20)
`PopFront(): Node` `Pop(): Node`	Remove	Remove a node at the beginning of the order. Pop is
`PopBack(): Node`	Remove	Remove a node at the end of the order

Table 9 Built-in Operations of Node_Order (Edge_Order). The same operations are defined for Node_Seq (Edge_Seq).

Notice that in the Table 8 and Table 9 there is a column named 'class', which denotes the class of the operation. For example Has() or Size() operation is classified as *Check* operation, while Push() or Add() operation as *Append*. 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.

```
// Assume a,b,c,d are distinct node. (They don't have to, by the syntax)
Local foo(G: Graph, a,b,c,d: Node(G)) {
  // Order and Seqeunce
  Node_Order(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
  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}
}
```

Code 20 Ordered-ness of elements after operations on an Order

4.6.2 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 21).

```
Local 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}
  }
}
```

Code 21 Sequential iteration over collections

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).

```
  // G: Graph, a,b,c: Node(G)
  // S:Node_Set(G) = {a,b,c}
  // O:Node_Order(G)= {b,a,c}
  // Q:Node_Seq(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.
  }
}
```

Code 22 Parallel iteration over collections and mutation during iteration

<todo: Type conversion between collections>

5 Parallel Execution and Consistency

5.1 Parallel Region

Green-Marl is designed to provide intuitive ways in exploiting data parallelism of graph algorithm. The basic idea is similar to that of OpenMP [2]: 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.

```
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 sentence

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

Code 23 Parallel region example

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.

Syntax	Semantic	Details
`Foreach`	Iterate every element in the given range (e.g., nodes in a graph, elements in a set) in parallel	Section XX
`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 XX

Table 10 Green-Marl syntax for a parallel execution region

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 Code 24 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.

```
1: Foreach (n: G.Nodes) {   // outmost loop (n-loop)
2:    Int x = 0;
3:    Foreach (t: n.Nbrs) {  // (t-loop)
4:      Foreach (r: t. Nbrs) {  // innermost loop (r-loop)
5:        If (r.A > t.A) x += r.A;
6:      }
7:    }
8:    n.B = x*2;
9: }
```

Code 24 Nested parallel execution region

Each parallel region is fork-joined (or split-merged) independently. As an example, consider Code 24 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:

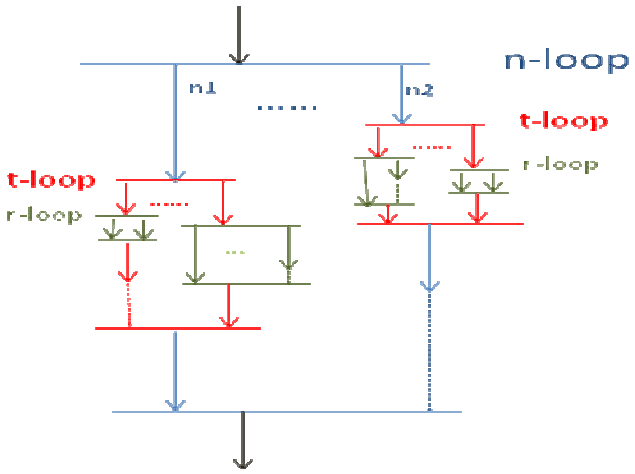


Figure 5 Visualization: independent concurrent executions of a nested parallel region.

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 4.6.2) 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.

```
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;
}
```

Code 25 Sequential consistency example

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

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 executions are merged, every write made by concurrent execution instance of the region becomes visible.

As an example, let's thinks about the case of the following bipartite graph (Figure 6) and apply it to the following code example (Code 26):

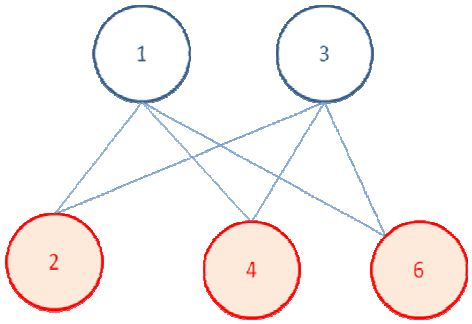


Figure 6 Example: bipartite graph

```
// 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};
}
```

Code 26 Parallel consistency example I

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 Figure 6: x can be either blue or red.

```
// Color is Node_Prop<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.
```

Code 27 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.

```
// Color is Node_Prop<Int>(G)
1: Int x = 0;
2: Foreach (n: G.Nodes) {
3:   x = n.Color;  // concurrent writes to a single location (variable)

    // concurrent reads from the same variable
4:  If (x != n.Color) {
       // The execution can be come into this branch,
       // because a write from anther concurrent execution may have altered
       // value of x between line 3 and line 4.
       ...
5:   }
6: }
```

Code 28 Parallel consistency example III

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 6). It is possible that some node end up with value having LEFT_FIRST, while others having RIGHT_FIRST.

```
// flag, value is nothing but node_prop<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.
    If (left_changed && !right_changed)     n.flag = LEFT_FIRST;
    Else If (!left_changed && right_changed)n.flag = RIGHT_FIRST;
    Else                                    n.flag = SAME;
  }
} // at the end of foreach, value of some node can be LEFT_FRIST while others
RIGHT_FIRST
```

Code 29 Parallel consistency example IV

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 (Code 26 - Code 29) 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 [3] 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.

```
// A is a node_prop<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.
```

Code 30 Bulk-Synchronous consistency example

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 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.

```
1: // A is a Node_Prop<Int>(G)
2: Int x = 0;
3: Int y = 0;
4: Foreach (n: G.Nodes) {
5:   x += n.A;          // Reduction by addition
6:   y = y + n.A;       // Not a reduction. The result is non-deterministic
7: }
```

Code 31 Reduction by Addition (Assignment Form)

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.

```
// A is a node_prop<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;
```

Code 32 Reduction by Addition (Expression Form)

However, the semantic 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.

Expression Form	Assignment Form
target = reduce_operator (iterator: range) *{ body_expression }*	*target = automatic_initializer* **Foreach** (*iterator* : *range*) { *target reduce_assign body_expression* }

Table 11 Reductions in expression form and assignment form

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

Reduce_ Assign	Reduce_ Operator	Automatic_ Initializer	Semantic	Expression Type
+=	**Sum**	0	By addition	Numeric
*=	**Product**	1	By multiplication	Numeric
max=	**Max**	**-INF**	By maximum	Numeric
min=	**Min**	**+INF**	By minimum	Numeric
++	**Count**	0	A syntactic sugar for += 1	Numeric
&&=	**All**	**True**	By logical ‘and’	Boolean
||=	**Any**	**False**	By logical ‘or’	Boolean

Table 12 Reductions in Green-Marl

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.

```
// A,B is a node_prop<Int>
Foreach (n: G.Nodes)
  Foreach (t: n.Nbrs)
    t.A += n.B;   // The entire propery A becomes the target of reduction.
```

Code 33 Reduction to Property

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 Code 34 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.

```
1: // A,B is a node_prop<Int>
2: Int x,y; x= 0; y=0;     // variable to store max
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
```

Code 34 Max and Argmax

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 semantic 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 semantic for this[5].

```
// A,B is a node_prop <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
}
```

Code 35 Example of errors on reduction variables

5.3.2 Deferred Assignments

Green-Marl supports Bulk Synchronous consistency (Section 5.2.3) through deferred assignments. Syntax-wise, deferred assignment is quite similar to reduction in assignment from. (See Code 30 in Section 5.2.3) The only noticeable difference is the use of <= symbol in place of reduction assignment symbols such as +=.

Unlikely to reduction, however, a variable (or a property) can be read in side 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 write is not effective yet but deferred to the end of binding loop. (Section 5.2.3)

The variable, which is being defer-assigned, should not be written otherwise; see Code 36 below for example.

[5] For example, a compiler may define the semantic of reading a reduction-target variable as to give the partial reduction result.

```
// A,B is a node_prop<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;

}
```

Code 36 Bulk-Synchronous consistency example

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 (Code 37). Inside a nested loop, at line 5, reading of y is non-deterministic because of the concurrent writes to y at line 7. However once t-loop is finished, reading of y becomes deterministic at line 10. On the other hand, reading of x at line 11 is still non-deterministic, since there are still concurrent writes inside this outer loop at line 13.

```
1: Int x = 0;
2: Foreach (n: G.Nodes) {
3:   Int y = 0;
     // Nested Loop.
     // Every t writes to the same y.
4:   Foreach (t: n.Nbrs) {
5:     Int z1 = y; // reading of y is non-deterministic (because of line 7)
6:     Int z2 = x; // reading of x is non-deterministic (because of line 8,12)
7:     y = 1;
8:     x = 1;
9:   }
10:   Int z1 = y; // reading of y is deterministic
11:   Int z2 = x; // reading of x is non-deterministic (because of line 13)
12:   y = 2;
13:   x = 2;
14:}
```

Code 37 Parallel consistency example I

The binding loop of deferred assignment (Section 5.3.2) might be ambiguous in case of nested loops. Green-Marl resolves such ambiguity by using @-syntax: at the end of deferred assignment statement follows @ symbol and the iterator name which denotes the binding loop.

The following two code examples (Code 38, Code 39) show how @-syntax can control the visibility of deferred assignment. In Code 38, the binding loop is the inner loop (s2-loop); thus writes to A become visible at the end of s2. In Code 39, on the other hand, the binding loop is the outer loop (s1-loop) – therefore all the writes inside s2-loop is not visible during the end of s1-loop.

```
// G: Graph
// S1,S2: Node_Set(G) S1 = {a,b,c} S2={b,d}
// A,B: N_P<Int>(G). Initially zero for every node.
For (s1: S1.Items) {
  Foreach(s2: S2.Items){ // G: Graph
     s2.A <= s1.A + 1 @ s2; // defer write to A during s2 loop.
  } // Modification of A becomes visible here
}
// The above code is executed as follows
// Content of A at s1-loop=> s1  s2   =>  Content of A after s2-loop
// A[a:0, b:0, c:0, d:0] => {a} {b,d} =>  [a:0, b:1, c:0, d:1]
// A[a:0, b:0, c:0, d:0] => {b} {b,d} =>  [a:0, b:2, c:0, d:2]
// A[a:0, b:0, c:0, d:0] => {c} {b,d} =>  [a:0, b:2, c:0, d:2]
```

Code 38 @-syntax example I

```
// G: Graph
// S1,S2: Node_Set(G) S1 = {a,b,c} S2={b,d}
// A,B: N_P<Int>(G). Initially zero for every node.
For (s1: S1.Items) {
  Foreach(s2: S2.Items){ // G: Graph
     s2.A <= s1.A + 1 @ s1; // defer write to A during s1 loop
  }
} // Modification of A becomes visible here
// The above code is executed as follows
// Content of A at s1-loop=> s1  s2   =>  Content of A after s2-loop
// A[a:0, b:0, c:0, d:0] => {a} {b,d} =>  [a:0, b:0, c:0, d:0]
// A[a:0, b:0, c:0, d:0] => {b} {b,d} =>  [a:0, b:0, c:0, d:0]
// A[a:0, b:0, c:0, d:0] => {c} {b,d} =>  [a:0, b:0, c:0, d:0]
// A[a:0, b:1, c:0, d:1]
```

Code 39 @-syntax example II

Similarly, reductions in nested loops can ambiguous. The same @-syntax can be used to describe the binding loop of the reduction as well. See the following example.

```
// G: Graph
// S1,S2: Node_Set(G)
// A,B: N_P<Int>(G)
Foreach (s: G.Nodes) {
  Foreach(s1: S1.Items){ // Reduction to B
     s1.B += s.A * 2 @s; // reduce to B during loop s.
  }
  Foreach(s2: S2.Items){ // Reduction to B
     s2.B += s.A * 3 @s; // reduce to B during loop s
  }
}
```

Code 40 @-syntax with reduction example

A reduction target cannot be doubly bound in nested loops. (See Code 41)

```
// G: Graph
// S1,S2: Node_Set(G)
// A,B: N_P<Int>(G)
Foreach (s: G.Nodes) {
  s.A += 1 @s;
  Foreach(s1: S1.Items){
     s1.A += s.B * 2 @s1; // Error - A is already bound to s-loop.
  }
}
```

Code 41 Binding Error in reduction with @-syntax

Note that @-syntax is in fact an act of exposition; it tells the compiler and the reader of the program that the variable (property) is being reduced inside a certain loop and that thus should not be accessed otherwise.[6]

As a matter of fact, Green-Marl requires for every reduction assignment or deferred assignment to be followed by @ syntax. However, the user may omit @ syntax, 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 and ask the user to specify the binding loop. See Code 42 for example.

[6] Note that OpenMP [2] does similar declaration with reduce-target list at the beginning of each parallel region.

```
// 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 naïve 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;
   }
   Foreach(t2: s.Nbrs){
       t2.A += s.B;
   }
}
```

Code 42 Example cases of Automatic Detection of Binding Loops

5.4 Operations on Collection Types under Parallel Consistency

Section 4.6 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.

```
// Assume a,b,c,d,e are distinct node.
Node_Order(G) O1,O2,O3;
Node_Set(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.
```

Code 43 Append under parallel consistency

As for '*remove*' class, the semantic is different for unordered collection and ordered collection.

- For unordered collections (i.e. set), the semantic 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 semantic 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 semantic of removal operation under parallel consistency as unordered sequential consistency.

```
// Assume a,b,c,d,e are distinct node.
Node_Seq(G) Q1,Q2;
Node_Set(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}
```

Code 44 Remove under parallel consistency

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

```
Node_Seq(G) Q1,Q2;
Node_Set(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
```

Code 45 Copy under parallel consistency

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 13 summarizes the semantic and compiler action when two different classes of operations are performed on a collection.

	Lookup	Append	Remove	Assign
Lookup	Okay	Warning (non-deterministic)	Warning (non-deterministic)	Warning (non-deterministic)
Append		Okay	Error (undefined)	Error (undefined)
Remove			Set: Okay, Order/ Seq: Error (undefined)	Error (undefined)
Assign				Warning (atomic, non-deterministic)

Table 13 Operations on Collections under Parallel Consistency

Let us explain Table 13 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 semantic 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.

```
// Suppose G has five nodes {a,b,c,d,e} and Color[a,c,e]= Blue
Node_Seq(G) Q1;
Node_Order(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 semantic, 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' on O1.
// In other words, every operation on O1 in q-loop becomes serializable.
```

Code 46 Append under parallel consistency

<todo: @ syntax + collection operation>

5.5 Implicit Parallel Context (Syntactic Sugars)

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 assignment. In Green-Marl, one can assign property values for a group of nodes (edges) in a simple way, using group assignment syntax. See the following code snippet for example.

```
Node_Prop<Int> A,B;
Set_Node(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 = G.A + 1; // For every node in G, set its A value as its B value plus 1

// The above three sentences are equivalent to below sentences
For (n: G.Nodes) n.A = 0;
For (n: S.Items) n.A = 2;
For (n: G.Nodes) n.B = n.A + 2;
```

Code 47 Example of collective assignments

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.

<todo: shall I simply let the collective assignment adopt parallel consistency?>

5.6 Notes for Green-Marl Compilers

5.6.1 Static Compiler Analysis for Data-race Detection.

Throughout Section 5, 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 warnings appropriately. However, Green-Marl is designed in a way that such analyses can be done easily because of high-level syntax, but more precisely because of their semantic information. See the following code as an example.

```
Set_Node(G) S;
Set_Seq(G) Q;
N_P<Int> A;
//...
Foreach (s: S.Items) {
  // There is no read-write data race.
  // Property A is accessed only by the 'Set' iterator, which is unique
  s.A = s.A + 1;
}
Foreach (q: Q.Items) {
  // There IS read-write data race.
  // Property A is accessed by the 'Sequence' iterator,
  // which does not guarantee uniqueness.
  q.A = q.A + 1;
}
```

Code 48 Example of data race detection in Green-Marl

Also note that Green-Marl guarantees that there is no alias between property names or collection names, which eliminates the possibility of data-races through aliases completely.

Table 14 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.

	Read	**Write**	**Reduce**	**Deferred-Write**
Read	Okay	Warning	Error	Okay
Write		Warning	Error	Error
Reduce			Okay, if same op Error, otherwise	Error
Deferred-Write				Warning

Table 14 Default compiler actions for detected data race

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 (Code 49). Although the original user description is a nested parallel region (line 1-5), the compiler can still decide to parallelize the outer loop only.[7] In such case, the compiler is able to replace the reduction (line 3) with normal read and write (line 8) -- parallel reduction typically is implemented using atomic instructions which take 2~3x times slower than normal read and write instructions

```
1: Foreach (s: G.Nodes) {  // outer loop (s-loop)
2:    Foreach (t: s.Nbrs) { // inner loop (t-loop)
3:        s.A += t.B @ t;
4:    }
5: }
// The compiler may choose to parallelize only the outer loop
6: Foreach (s: G.Nodes) {  // outer loop (s-loop) -> parallel
7:    For (t: s.Nbrs) {    // inner loop (t-loop) -> sequential
8:        s.A = s.A + t.B  // reduction can be replaced with normal read & write
9:    }
10:}
```

Code 49 Selection of Parallel Execution

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 Code 50. The first loop (line 2-4) can be safely transformed into a parallel loop (i.e. foreach) since `Node_Set` guarantees uniqueness of elements and property `A` is only accessed through `s`. On the other hand, the second loop (line 5—7) cannot be parallelized naively; there is be a read-write data race between writing to property `A` (via

[7] For 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.

`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.

```
// The compiler can safely change following for into foreach
1: Node_Set S;
2: For (s: S.Nodes) {
3:   s.A = s.A +1 ;
4: }

// Blind parallel execution does not guarantee sequential consistency.
// A compiler may parallelize it with unordered sequential consistency,
// using locks or transactional memory.
5: For (n: G.Nodes) {
6:   n.A = Sum(t:s.Nbrs){t.A};
7: }
```

Code 50 Parallel execution of Sequential Loops

5.6.3 Implementation for collection types

Green-Marl language puts no constraint on the implementation of Collection types (Section 4.6) as long as its behavior under sequential consistency (Section 4.6) 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.

```
// Say Node_Set is mapped to a Queue by compiler specification.
// S1 should be implemented as a Queue.
Procedure foo(G: Graph, S1: Node_Set(G), A: N_P<Int>(G))
{
   Node_Set(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 <,am>  min= s.A <,s>;
      }
      S2.Remove(am);          // S2 is always removed by some min-value.
   }                          // S2 might be implemented as a heap.
}
```

Code 51 Different implementations for Internal Collections

6 Expressions and Sentences: 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 (Section 6.4), (3) conditional parts of If and (Do-)While sentence (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 Type

There are two different kinds of operations defined for numeric types: Arithmetic and Comparison. Arithmetic binary operators are +, -, *, /, and % (only for Int and Long type). Comparison operators are <, >, ==, and !=. The semantic 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.

```
// z becomes 3.5, y becomes 3
Float z = | -3.5 |;
Int y = | -3 |;
```

Code 52 Examples of | | operator.

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 semantic of those operators are same to C.

6.1.3 Conditional Operator

```
Bool_expr ? expr1 : expr2
```

Green-Marl also provides Ternary conditional operator as in C. The semantic 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:

```
reduction_op (iterator_name : source_name . range_word) ( (filter_expr) )
{ 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 Graph (Section 4.3.1), Node, Edge (Secion 4.4.1), or any collection (Section 4.6.2), while each type can be followed by different range words. (See Section 4.3.1, 4.4.1, and 4.6.2 for range words for each type.) Note that the type of an iterator is automatically determined by the type of source and range word.

The semantic 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 6.1.4 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.

```
// Reduction by addition.
// summation: for all nodes of G, such that n.Color == Blue, n.A + n.B
Int z2 = Sum(n:G.Nodes)(n.Color == Blue){n.A +n.B};
```

Code 53 Examples of Count Reduction

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 6.1.4. See the following code example.

```
Node_Set(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
```

Code 54 Examples of Empty Range

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

```
// 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}}
```

Code 55 Nested Reduction

Count is a syntactic sugar for Sum(..){1}.

```
// 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};
```

Code 56 Examples of Count Reduction

6.1.5 Operator precedence

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

Operator	Description	Associativity
( ) | | Reduction operations	Parenthesis Absolute Value Reduction operations	Left to right
- ! (type)	Unary minus Logical negation Type Cast	Right to left
* / %	Multiplication / division / modulus	Left to Right
+ -	Addition/ Subtraction	Left to Right
> >= < <=	Numeric comparison	Left to Right
== !=	Numeric/Logical comparison	Left to Right
&&	Logical And	Left to Right
||	Logical Or	Left to Right
?:	Ternary conditional	Left to Right

Table 15 Operator precedence in Green-Marl

6.2 Assignment States:

6.2.1 LHS and RHS

Green Marl assignment statements have the following form. Its semantic 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 4.5 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 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 (@ variable_name ); // reduction assignment
LHS <= RHS (@ variable_name ); // deferred assignment
```

The optional variable name follows the assignment RHS clarifies the visibility of these statements. See Section 5.3.3 for detailed discussions.

6.3 Control Sentences

6.3.1 For and For-each 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_sentence

`Foreach` **(***iterator_name* **:** *source_name . range_word***)** **(** (filter_expr) **)**
  body_sentence

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 4.3.1), Node, Edge (Secion 4.4.1), or any collection (Section 4.6.2), while each type can be followed by different range words. (See Section 4.3.1, 4.4.1, and 4.6.2 for range words for each type.) Note that the type of an iterator is automatically determined by the type of source and range_word.

The semantic 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 sentence, and the body sentence 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 4.6.2 for the semantics of iteration on collection types.

The following code gives examples of For and Foreach iteration:

```
Node_Set(G) S;    // A set of nodes of graph G
Node_Order(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) {
  ...
}
```

Code 57 Examples of For and Foreach Iteration

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 [5].

DFS traversal has the following syntax:

InDFS (*iterator_name* **:** *source_name* (**^**) . **Nodes [From|;]** *root_name*) ((*filter_expr*))
([*navigator_expr*])
{ *body_sentence* }
(**InPost** ((*filter_expr2*)) { *post_visit_sentence* })

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 semantic is to traverse the graph in depth-first order from the root node and to execute the body sentence at each node. Every (reachable) node is visited once and only once. The body sentence is executed whenever the node is fist visited but prior to visiting its neighbors, i.e. pre-order visitation. When there are multiple non-visited neighborhood nodes from current node, it is not defined which order those neighbors are visited. For example, if the graph in Figure 7 is

applied to Code 58, the result of DFS iteration at line 3 - 5 can be {a,b,c,e,f,d}.

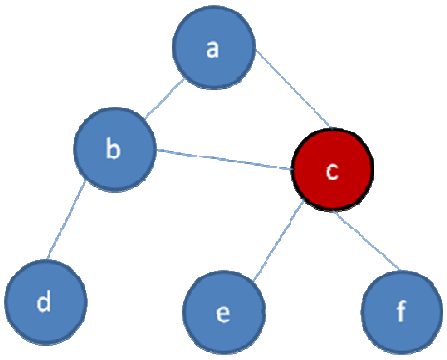


Figure 7 An Example Undirected Graph

```
1: Node(G) a;
2: Node_Order(G) O1, O2, O3; // An order of nodes of graph G
//…

  // DFS order (pre-order) traversal
3: InDFS (s: G.Nodes From a) {
4:   O1.Push(s);
5: }
   // O1 can be {a, b, c, e, f, d} when applied to Figure 7

   // DFS order (post-order) traversal. ';' is a short-hand for From
6:  InDFS (s: G.Nodes ; a) {} // do nothing on pre-visit
7:  InPost  {O2.Push(s);}        // do something on post-visit
   // O1 can be {e, f, c, d, b, a} when applied to Figure 7
8: }
```

Code 58 Example of DFS Iteration

It is also possible to visit the nodes in post DFS order, by using InPost clause, as shown in line 6 - 8 of Code 58.

Two optional Boolean expressions can be attached to DFS traversal: *filter expression* and *navigator expression*. As in For iteration, filter expression is computed at the moment the node is first visited and the body sentence is not executed if the value of filter expression is false. Navigator expression is similarly computed at the visiting moment and body sentence is not executed either if the value is false; however, if the value of navigator expression is

false, its neighborhood nodes[8] are not further considered in traversal. See the following code for example.

```
1: Node(G) a;
2: Node_Order(G) O1, O2; // An order of nodes of graph G
//…

  // DFS order traversal with filters
3: InDFS (s: G.Nodes From a)(s.color == Blue) {
4:   O1.Push(s);
5: }
   // O1 can be {a, b, e, f, d} when applied to Figure 7

  // DFS order traversal with navigators
6:  InDFS (s: G.Nodes From a) [s.color == Blue] {
7:   O1.Push(s);
8: }
   // O1 is { a, b, d } when applied to Figure 7
```

Code 59 Example of filters and navigators in DFS traversal

BFS traversal has a similar syntax to DFS traversal, as shown in the following box

InBFS (*iterator_name* **:** *source_name* (**^**) . **Nodes [From|;]** *root_name*) ((*filter_expr*))
([*navigator_expr*])
{ *body_sentence* **}**
(**InReverse** ((*filter_expr2*)) **{** *reverse_visit_sentence* **}**)

The semantic of BFS traversal is to traverse the graph in BFS order [5]. That is 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 InReverse sentence. See Code 60 for example.

The semantics of filter and navigator in BFS traversal are same as those in DFS traversal.

[8] more accurately, its outgoing edges

```
1: Node(G) a;
2: Node_Order(G) O1, O2; // An order of nodes of graph G

  // DFS order (pre-order) traversal
3: InBFS (s: G.Nodes From a) {
4:   O1.Push(s);
5: }
   // O1 can be {a, b, c, e, d, f} when applied to Figure 7

6:  InDFS (s: G.Nodes ; a) {} // do nothing on pre-visit
7:  InPost  {O2.Push(s);}        // do something on post-visit
   // O1 can be {f, d, e, b, c, a} when applied to Figure 7
8: }
```

Code 60 Example of BFS Iteration

Another big difference between DFS traversal and BFS traversal is, however, 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, whosever hop distance from the root node is same, is visited in parallel. As an example, suppose a BFS traversal is applied to the graph in Figure 7 stating from node a. In this case, node {a} is visited first. Then nodes {b, c} are visited concurrently and then, lastly, nodes {d, e, f} are visited concurrently.

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:

```
If (bool_expr) then_sentence

If (bool_expr) then_sentence
Else else_sentence
```

Repeated (ambiguous) If-Else is parsed as in C. See the following example

```
// 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();
}
```

Code 61 Handing of ambiguous If-Else

While and Do-While statements have following syntax. Their semantic is same to C-language's. While and Do-While statements adopt sequential consistency.

While (*bool_expr*)

body_sentence

Do *body_sentence*

While (*bool_expr*) **;**

6.4 Procedure Calls and Built-in Functions

6.4.1 Procedure calls

A Green-Marl program can make calls to other Green-Marl procedures defined in the same file (Section 3.1.1).

Green-Marl procedure calls take the following form, where actual argument list comes within a parenthesis, after the procedure name.

procedure_name **(***(RHS1, RHS2, …) (***;** *LHS1, LHS2, …)***)**

The actual argument list is divided into two parts – input arguments and output arguments, matching the signature of the procedure being invoked. RHS should be located in place of actual input arguments, while LHS in place of output arguments.

However, an actual output argument can be ignored, by using # symbol in place. The semantic of this is to discard any value returned by corresponding formal argument, at this call-site.

```
// The following If-Else is ambiguous.
Local get_min_max(a,b: Int; min,max: Int)
{
   If (a>b) {min = b; max = a;}
   Else {min = a; max = b;}
}
Procedure foo(x,y,z: Int)
{
   Int t1,t2;
   get_min_max(x,y;  #,t1);  // t1 is the max of x and y
   get_min_max(z,t1; t2,#);  // t2 is the min of z and t1
}
```

Code 62 Ignoring output arguments

As for numeric, Boolean, node/edge types, the value-passing semantic and the type constraint between actual argument and formal arguments are exactly same as normal assignment – both for input arguments and output arguments. In other words, (1) RHS values are copied into LHS location, and (2) RHS and LHS types should be exactly matched except coercions for numeric types. As for collection and property types, which can be only used as input

arguments, the value passing semantic is by reference. See Section 3.1.2 for related discussion.

A procedure call can be a separate sentence or a (sub-) expression. When a procedure call is used as a sentence it has to be followed by semicolon. A procedure call sentence may have side-effects or output arguments.

The procedure calls are used as a sub-expression should be free of side-effect and output arguments. There are two exceptions: (1) it is the sole element of the expression. (2) All output arguments are ignored. (See Section 3.3 for examples)

6.4.2 Built-in Functions and Operations

There are some types which have built-in functions and operations: Graphs (Section 4.3.1), Nodes and Edges (Section 4.4.1) and Collections (Section 4.6.1). 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.

Signature	Semantic
Rand() : Int	Returns a random integer number
Log(Double) : Double	Returns the natural logarithmic value
Pow(Double base, Double exp): Double	Returns the power function
Sqrt(Double): Double	Returns the square root value
Exp(Double): Double	Returns the base-e exponential value

Table 16 Ground-level Built-in Functions in Green-Marl

7 Interacting with Application Codes

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.

```
// Green Marl Program
Procedure Foo(G: Graph, n: Node(G), Node_Prop<Int>(G) A; t: Float): Bool
{
  ...
}
```

```
// C++ function
bool Foo(GM_Graph &G, int64_t n, int[] G__A, float& t)
{
  ...
}
```

Code 63 Signatures of Green-Marl procedure and translated c++ function

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 inCode 64 . 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 Code 64 is an error.

```
// The compiler can safely change following for into foreach
1: Procedure Foo (G: Graph, T: $UDT1)
2: {
3:  $UDT2 X;    // The user can declare of foreign-type variable
4:  X = T;      // compiler just believes every foreign type is equivalent
5: }
```

Code 64 Foreign Types in Green-Marl

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 68 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 `$` symbol inside the foreign expression. In Code 65, `$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.[9] Also since Green-Marl knows which variables are being read, it can prevent their anti-dependent sentences from being reordered prior to the foreign expression.

```
1: Procedure Foo (G: Graph, A,B: N_P<Float>(G),  T: $UDT1)
2: {
3:   Foreach (n: G.Nodes) {
4:     n.A = n.A + [log($n.B) + $T->getValue()] * 0.5;
5:     n.B = n.B + 1; // compiler should not re-order line 5 and line 4
6:   }
7: }
```

Code 65 Foreign Expressions in Green-Marl

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 sentences

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.

In Code 66, as an instance, line 6 – 9 inside [] are foreign sentences. The compiler keeps the string as-is in the generated code, except Green-Marl variables inside the string; Green-Marl

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

variables are denoted by $ in the string. Foreign sentences can be optionally followed by the list of variables modified inside the foreign sentence. Line 10 and 13 in Code 66 are examples of it.

```
1: Int y,x,z;
2: x=0;
3: While (x <= 10)
4: {
    // sentences inside [] are foreign sentences.
    // $x, $y, $z are Green-Marl variable accesses.
    // [y,z] are modified variables.
    // :: can be replaced with a keyword Mutating
5:   If (x == 0) [
6:       printf("This is the first execution:%d", $x);
7:       FILE *f = fopen("data_file", "r"); assert(f!=NULL);
8:       $z = fscanf(f, "%d", &$y);
9:       fclose(f);
10:  ] :: [y,z]
11:  Else { y = y + 1; x = x + 1; }

    // Compiler knows property A is mutated though iterator n.
12:  Foreach(n:G.Nodes)
13:      [mutate(&$n.A, $y);]::[n.A]
12:}
```

Code 66 Foreign Sentences in Green-Marl

7.2.4 Foreign functions

<todo: declare foreign functions and call them in expression or sentence, more conveniently then foreign expression. Potentially with a little more of type checking>

7.2.5 Target Header Include

<todo: User can add includes in the generated file. Whatever it can be>

```
// compiler will generate appropriate include syntax at the beginning of the
generated file. String inside <> will be simply copied.
Include<Java.util.Rand.*>;

Local Foo() {…}
Local Bar() {…}
```

Code 67 Inclusion of Target Header File

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.

```
1: Proc conductance(G: Graph, member: N_P<Int>(G), num:Int) : Float
2: {
3:    Int Din, Dout, Cross;
4:    Din =  Sum(u:G.Nodes)(u.member == num){u.Degree()};
5:    Dout = Sum(u:G.Nodes)(u.member != num){u.Degree()};
6:    Cross =  Sum(u:G.Nodes)(u.member == num){
7:                Count(w:u.Nbrs)(w.member != num)};

8:    Float m = ((Din<Dout) ? Din : Dout);
9:    If (m==0) Return (Cross == 0) ? 0.0 : +INF;
10:   Else Return Cross / m;
11: }
```

Code 68 Conductance in Green-Marl

Code 68 is Green-Marl program to compute Conductance of a Sub-graph (more accurately, of a cut) [5]. 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 foreach loops and then merge them together into one loop.

```
1: Proc pagerank(G: Graph, e,d: Double, pg_rank: N_P<Double>(G), max:Int)
2: {
3:   Double diff;
4:   Double N = (Double) G.NumNodes();
5:   G.pg_rank = 1 / N;
6:   Do {
7:       diff = 0.0;
8:       Foreach(t: G.Nodes) {
9:          Double val = (1-d) / N + d *
10:             Sum(w: t.InNbrs)(w.OutDegree()>0) {w.pg_rank / w.OutDegree()};

11:          diff += | val - t.pg_rank |;
12:          t.pg_rank <= val @ t;
13:      }
14:      cnt ++;
15:   } While ((diff > e) && (cnt < max));
16:}
```

Code 69 PageRank in Green-Marl

Code 69 is a Green-Marl program that computes page-rank of a graph [6]. 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 12 and line 10.

```
1: Proc CC(G: UGraph, membership: N_P<Int>(G)):Int
2: {
3:   Int numC = 0;
4:   G.membership = -1;

     // Sequential Iteration of Nodes
     // Visit nodes that have not included in any component yet.
5:   For (s: G.Items) (s.membership == -1) {
       // Do BFS and mark all reachable nodes
6:     InBFS(t: G.Nodes From s) {
7:         t.membership = numC;
8:     }
9:     numC++;
10:  }
11:  Return numC;
12:}
```

Code 70 Connected Components

Code 70 is a Green-Marl program that obtains connected components [5] 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.

```
1: Proc SCC_kosaraju(G: Graph, membership: N_P<Int>(G)):Int
2: {
3:    Int numC = 0;
4:    Node_Order(G) P;
5:    G.membership = -1;
6:    G.filter

   // Obtain post DFS order
7:    For(s: G.Nodes) (! P.Has(s)) {
8:       InDFS(t: G.Nodes from s) [! P.Has(s) ]
9:       InPost { P.Push(t);}
10:   }

   // In reverse, post DFS order
11:   For (s: P^.Items) (s.membership == -1) {
12:      InBFS(t: G^.Nodes From s) [t.membership==-1] {
13:           t.membership = numC;
14:      }
15:      numC++;
16:    }
17:   Return numC;
18:}
```

Code 71 Strongly Connected Components (Kosaraju's Algorithm)

Unlikely to the case of connected components in undirected graphs, obtaining strongly connected graphs in undirected graphs requires a little more computation; Code 71 computes strongly connected component, using Kosaraju's Algorithm [5]. 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^. Also this BFS traversal only goes through the nodes that have not been marked yet (line 12).

9 Ideas for Future Versions

- Multi-set

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

[Issue] Not Much

[Syntax Suggestion]

```
Node_Multiset(G) M1;
N_M(G) M2;
Edge_Multiset(G) M3;
E_M(G) M4;
// same operations as set.
```

- 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.

- String primitive

[Idea] String as a primitive; string cannot be mutated. It can only be copied.

[Issue] String is not well handled in CUDA implementation. Maybe CUDA target can simply reject codes that contain string types.

In C++ target, memory leakage should be well handled.

Do we need NIL for String?

[Syntax Suggestion]

```
String s = "Hello World";  // assignment
==, !=, <, <=, >=, >       // comparison operator
::                         // string concatenation (results a new string)
s.contains("Hello"):Bool   // contains substring (returns Bool)
… // a few more built-ins. Compiler implementation can add more
```

- 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; semantic becomes non-deterministic with parallel consistency. (Other loop instances may or may not stop.)

- 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.condiiton)
                Continue @ t;
            Sent_1();
       }
       Sent_2();
   }
   Sent_3();
}
```

[Issue] Under sequential consistency, the semantic 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 semantic under parallel consistency? (i.e. when For->Foreach?)

- Named while loop

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

[Syntax Suggestion]

```
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?

- 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]

```
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 #<=
```

- 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]

```
Node_Set(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.
```

- Node->Edge

[Idea] Direct a node from an edge. And vice versa. "Give me the node at the end of this edge." Or "Give me the edge that connects these two nodes."

[Syntax Suggestion]

```
Node(G) n;
Edge_Set(G) ES;
Foreach (t: s.Nbrs) {
  Edge(G) e = t.Edge(); // an edge from s to t.  t should be an iterator from
nbr family range word.
  ES.Add(e);
}
```

- Neighborhood Marker

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

[Syntax Suggestion]

```
Neighbor_Set(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]

```
Node_Order(G) O;
Node_Set(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
{
  // ...
}
```

- 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]

```
Foreach (n: G.Nodes)
{
    Print "Hello From node " :: n :: "."; // prints are not mixed-up.
}
```

- 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 Execption. Or just a Error Flag, …)

[Syntax Suggestion]

```
Foreach (n: G.Nodes)
{
    If (n.something_is_wrong)
       Error "Something is Wrong";
}
```

- Collection of Collections

[Idea] A homogeneous collection of Set/Order/Sequence/MultiSet.

Always a order (Uniqueness is gaurenteed by 'copying'. Orderedness is easy. Thus essentially becomes a queue.)

Adding an item into the collection is making a copy.

Reference can be obtains via special syntax.

[Issue] Now we are introducing aliases. But hopefully, compiler can figure out where those aliases are.

```
Collection<Node_Set(G)> C;  // Allows homogeneous collection only

Node_Set(G) S;

C.Push(S); // Add a copy of S into C. So there is no alias between any
           // therefore every element in the collection is distinct.

// iterator c is an reference to the element of the collection
Foreach(c: C.Items) {
   // iterator d is again a reference
   Foreach(d: C.Items)(d.Size() >= c.Size()) {
         // here c and d can be an alias.
         // compiler should give an error or warning.
         Foo(c,d);

         // this is a copy. So T cannot be an alias to any other.
         Node_Set(G) T = c;
         Foo(T, d); // therefore this is okay.
   }
}

// Reference can be only obtained by 'structural way'
ChooseMax(c:C.Items){c.Size())
{ // c is a reference to the collection.
  // c is the item of C, which maximizes c.Size()
  do_something_with(c)
}
```

References

1 Brandes, U. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology* (2001), p. 163-177.

2 OpenMP Specification, OpenMP Consortium, www.openmp.org

3 Valiant, L. A brindging model for parallel computation, *Commiunications of the ACM* vol. 33 (1990).

4 Malewicz, G. et al. Pregel: Asystem for large-scale graph processing, Proceeding of 2010 international conference on Management of data (SIGMOD'10).

5 Bollobas, B. Morder Graph Theory, Springer-Verlag, (1998)

6 Brin, S and Page, L. The anatomy of a large-scale hypertextual Web search engine, Computer Networks and ISDN systems 30 (1998), p. 107-117