PLOG

Language Reference Manual

Wesley Bruning
(wab2125)

Table of Contents

1. Introduction

This manual describes the PLOG language, a language created for academic purposes. Its structure (and some initial content) is similar to the C language reference manual (Kernighan & Ritchie, “The C Programming Language”, 2nd edition, Prentice Hall, pgs. 191 – 239). First and foremost: it is a reference manual; it can be read from beginning to end, but note that the topics discussed progress from “low-level” concepts to “higher-level” level concepts and structures.

PLOG, a Programming Language for Opérations Graphiques, is designed as an approach to modeling with property graphs. Graphs are familiar structures used throughout mathematics and computer science—structures containing elements commonly known as nodes and edges—and property graphs, more generally, allow nodes and edges to possess “properties” and their associated values. PLOG facilitates the creation and use of property graphs by allowing users of the language to interact with graphs in declarative and imperative fashions, as well as to perform graph queries and updates more concisely with pattern-matching.

PLOG is intended for academic applications that wish to model and interact with data as property graphs, e.g. for (establishing flow rates in) models of computer networks, (establishing relationships and inferring patterns of) social networks, et cetera. This document describes capabilities and limitations of the PLOG language itself.

2. Lexical Conventions

A program consists of a sequence of ASCII characters which undergoes a series of transformations at “compile time”. The first such transformation produces a sequence of tokens, described below.

2.1 Comments

The characters `/*` introduce a comment, which terminates with characters `*/`. Comments do not nest, nor do they occur within string literals.

2.2 Tokens

There are five classes of tokens: identifiers, keywords, literals, operators, and other separators. Spaces, tabs, newlines, formfeeds, and comments—collectively, “whitespace”—are ignored except as they separate tokens. Some whitespace is required to separate otherwise adjacent identifiers, keywords, and literals.

2.3 Identifiers

An identifier is a case-sensitive sequence—one character or longer—of letters, digits, and underscores. The first character must be a letter.

2.4 Keywords

The following identifiers (described in the following sections) are reserved for use as keywords and may not be used otherwise:

and	if	or
append	in	print
del	INF	remove
edge	int	return
else	length	string
false	list	true
for	NIL	where
func	node	while
graph		

2.5 Literals

2.5.1 Boolean Literals

`true` and `false`, corresponding to Boolean (logical) true and false, respectively.

2.5.2 Integer Literals

A sequence of one or more (decimal) digits.
A unary ``integer negation” operator exists, which, when applied to an expression evaluating to an integer (e.g. an integer literal), returns the negation of that integer.

2.5.3 String Literals

A sequence of zero or more ASCII characters between two non-“escaped” double-quotes. The following characters are included as part of a string literal according to their escape sequence:

Character	Escape sequence
New line	\n
Carriage return	\r
Tab	\t
Double-quote	\”
Backslash	\\

2.6 Operators

The tokens representing operators are listed here, along with details of their meaning and use. Additional information on operators may be found in Sections 4.3 and 4.4.

All operators are left-associative, with the exception of Assignment, which is right-associative.

2.6.1 Unary operators

Operator	Name	Description
!	Logical negation ("not")	Applies to boolean expressions. False becomes true; true becomes false.
-	Integer negation	Applies to expressions evaluating to integers. Negates the integer.

2.6.2 Binary operators

Operator	Name	Description
=	Assignment	Assigns the value of an expression to a name.
or	Logical or	Returns the logical 'or' of two boolean expressions.
and	Logical and	Returns the logical 'and' of two boolean expressions.
==	Equality	Returns true iff two expressions are equivalent.
!=	Not equal	Returns false iff two expressions are equivalent.
<	Less than	Applies to expressions evaluating to integers. Returns true iff the left operand is less than the right operand.
<=	Less than or equal	Applies to expressions evaluating to integers. Returns true iff the left operand is less than or equal to the right operand.
>	Greater than	Applies to expressions evaluating to integers. Returns true iff the left operand is greater than the right operand.
>=	Greater than or equal	Applies to expressions evaluating to integers. Returns true iff the left operand is greater than or equal to the right operand.
+	Addition	Applies to expressions evaluating to integers. Returns the sum of the expressions.
-	Subtraction	Applies to expressions evaluating to integers. Subtracts the right operand from the left operand.
*	Multiplication	Applies to expressions evaluating to integers. Returns the product of the expressions.
/	(Integer) Division	Applies to expressions evaluating to integers. Divides the left operand by the right operand. Truncates the result.
.	Property access ("dot")	Left operand must be an expression that evaluates to an object of type node or edge, and right operand must be an identifier. Accesses the property of the node or edge with the property

		name given by the identifier. If no such property exists and its value is being “read”, the behavior is undefined. If no such property exists and its value is being “set”, the property value is set to the given expression (as part of an Assignment)—which must evaluate to an integer.

2.6.3 Operator precedence

The precedence of the above operators is given below, from lowest to highest precedence and with operators on the same line having the same precedence:

Assignment
Logical or
Logical and
Equality; Not equal
Less than; Less than or equal; Greater than; Greater than or equal
Addition; Subtraction
Multiplication; Division
Logical negation; Integer negation
Property access

2.7 Separators

Additionally, the following characters also have specific meanings (referenced in Sections 4 and 5):

```
( ) [ ] { } ; : , .
```

The “`->`” string is reserved as one way of denoting edges, as described in Sections 3 and 5.
For details, see Section 6.

3. Meaning of Identifiers

Identifiers (names) can refer to objects (variables), functions, properties, matching elements in graph patterns, and “named nodes”. Objects, discussed first, represent values of particular types.

3.1 Type Inference

Type inference is not carried out in PLOG—with the possible exception of the basic inference that is carried out for graph patterns (Section 3.6) and graph element declarative statements (Section 5.1). In order to determine the type of an identifier, it must be explicitly noted upon declaration.

3.2 Basic Types

Note: The keywords displayed throughout this section refer to those keywords which indicate an object of the given type at the time of declaration.
There are three primitive types:

3.2.1 Boolean

`bool`: Represents a Boolean value: true or false.

3.2.2 Integer

`int`: Represents an integer value.

3.2.3 String

`string`: Represents a string value: a sequence of zero or more ASCII characters (as described in section 2.5.3).

3.3 Graphs and Graph Elements

Graph objects and graph element objects (i.e. nodes, edges) are a significant part of the PLOG language. They are all mutable objects. All three types can be considered derived types, for the reasons explained below:

3.3.1 Graph

`graph`: Represents a graph structure, which can possess multiple nodes and edges (between those nodes) in its "scope" (discussed in Section 5.1).

3.3.2 Node

`node`: Represents a node object. Nodes can possess properties (see Section 3.5 below).

3.3.3 Edge

`edge`: Represents an edge object, which must exist between two (not necessarily distinct) objects. Edges can possess properties (see Section 3.5 below).

3.4 Other Derived Types

<type> `list`: Represents a list of objects, all with the given type (where <type> can be any one of the types above). The PLOG standard library has basic functions to modify and access lists (see Section 4.5).

3.5 Properties

Node and edge objects may possess multiple properties, each with an integer value (only). Property names must be given as an identifier.

Node and edge properties can be accessed: "get" and "set" operations can be performed on properties. "Getting" a property retrieves its integer value. "Setting" a property assigns a property an integer value. If attempting to get the value of a property that doesn't exist for a particular object, the behavior is undefined. However, assigning a property that doesn't exist for a particular object results in that property essentially being created, and the given (integer) value being assigned to the (new) property. See Section 5.4 for more on the "Property Access" operator.

3.6 Graph Patterns

Graph patterns facilitate searching for and iterating over graph elements meeting specific characteristics. Roughly, a graph pattern is given by the general structure:

```
node edge-> node edge-> node … where property=intval, …
```

An example of which is:

```
x likes-> y dislikes-> x
```

The same identifier used throughout a graph pattern will always refer to the same graph element. For the syntax of graph patterns, see Section 5.2 or Section 6.

3.7 Named Nodes

Graph patterns which return a node can be given a name. Such a named pattern is referred to as a "named node", e.g. as in:

```
node grandparent = z in x parent-> y parent-> z;
```

The underlying pattern can be invoked when `node:` is prefixed to the named node identifier, in combination with a graph object, e.g. as in:

```
for node:grandparent g in myGraph { /* g accessible locally */ }
```

An identifier with the "named node type" (e.g. `g`) is treated as having the `node` type.

3.8 Functions

Identifiers can also refer to functions. A function can be thought of as a named list of statements that can receive input and produce output. For more information on functions, see Section 5.3.

4. Expressions

This section details all of the expressions present in the current specification of PLOG. Each production (of "terminals" from "non-terminals") is examined and described.

The handling of overflow, division-by-zero, and other exceptions in expression evaluation is not defined by the language.

4.1 Reference Grammar Syntax

Lower-case strings (potentially containing underscores) refer to non-terminals and upper-case strings refer to terminals. Non-terminals produce the empty string if the line directly below the non-terminal name begins with a pipe ('|') character. Terminals are defined at the end of Section 6. Operator precedence and associativity is defined in Section 2.6.

4.2 Atomic Expressions

```
expr:
  literal
| ID
| INF
```

ID refers to an identifier, INF refers to the keyword `INF` which represents infinity, and *literal* has the productions:

```
literal: TRUE | FALSE | INTLIT | STRLIT
```

The three expressions above evaluate to: their literal values, the value of their identifier, or “infinity” (a value of type integer), respectively.

As mentioned above, the `INF` keyword indicates a (singular) object of type `int`. Arithmetic operations (as well as integer negation) involving `INF` are undefined, but comparative operations (i.e. inequalities, equalities) are defined: any non-`INF` and non-`NIL` (described below) integers are considered “less than” `INF`. `INF` is equal to `INF`.

4.3 “Unary” Expressions

The following expressions contain a single non-terminal alongside keywords, operators, or separators:

```
expr:
  MINUS expr %prec NEG
| LPAREN expr RPAREN
| NIL LPAREN typ RPAREN
```

The first refers to integer negation, while the second simply returns the value of the expression contained within parentheses.

Use of the `NIL` keyword and a given type returns a (singular) object with a “null” value of the given type. Only equal/not equal operations are defined for null values: an expression evaluating to a null value of one type is equal to another expression iff the second expression evaluates to a null value of the same type.

4.4 “Binary” Expressions

```
expr:
  expr PLUS expr | expr MINUS expr
| expr TIMES expr | expr DIVIDE expr
| expr LT expr | expr LEQ expr | expr GT expr | expr GEQ expr
| expr EQ expr | expr NEQ expr
| expr OR expr | expr AND expr
| expr ASSIGN expr
| expr DOT ID
```

Arithmetic operations of integers only can be expressed, as reflected in the first two lines above.
The third line represents integer inequalities, returning Boolean values expressing the truth of the binary expression.

Equality of expressions can be determined. For expressions evaluating to primitive types (i.e. Booleans, integers, or strings), comparison is by value. For expressions evaluating to (references to) objects (i.e. nodes, edges, graphs), the object references are compared.
OR and AND accept only Boolean expressions and represent logical 'or' and logical 'and', respectively.
Assignment is carried out by evaluating the expressions to the left and right of the ASSIGN token. The type of both evaluated expressions must match.
The “Property Access” operator DOT allows access to node or edge properties: this expression can be used to “get” or “set” properties. The behavior of “getting” and “setting” properties is described in Sections 2.6.2 and 3.5.

4.5 Function Calls (and: The Standard Library)

```
expr:
  ID LPAREN exprs_opt RPAREN
| PRINT LPAREN STRLIT exprs_opt RPAREN
| APPEND LPAREN expr COMMA expr RPAREN
| REMOVE LPAREN expr COMMA expr RPAREN
| LENGTH LPAREN expr RPAREN
```

Functions (mentioned in Section 3.8) can be called as given by the first expression above.
The remaining expressions above reveal the “standard library” for PLOG:
The “print” function prints a given string literal. The string literal may optionally contain the “value substrings” `%b` , `%d` or `%s,` used to refer to values of type Boolean, integer, or string, respectively. If any such substrings are included in the given string literal, the optional list of expressions (given by `exprs_opt`) must contain (exactly enough) corresponding values (of the expected types). The value of the first expression takes the place of the first value substring, the value of the second expression takes the place of the second value substring, and so on.
The “append” function accepts a variable and a list, and appends the input variable to the input list—provided the variable's type matches the list's element type.
The “remove” function accepts a variable and a list, and removes the input variable from the input list, if it exists.
The “length” function returns the (integer) number of elements in a given list.

4.6 Lists of Expressions

These productions allow expressions to be combined essentially as lists:

```
exprs_opt:
| expr_list

expr_list:
  expr
| expr_list COMMA expr
```

Additionally, the expression that allows the definition of lists (of expressions) is given as:

```
expr: LBRACK exprs_opt RBRACK
```

5. Program Structure: Graphs, Functions, and More

A PLOG program, at the "highest level", is a sequence of named node definitions, graph definitions, and function definitions:

```
program:
  decs EOF
decs:
| decs named_node_def
| decs graph_dec
| decs func_dec
```

It's not required to define named nodes or graphs, but every PLOG program must contain exactly one defined "main" function.

5.1 Graphs

Graphs are objects that can be used in a number of ways.

5.1.1 Graph Initialization

Graphs are declared and initialized only once: at the "global scope" of the program; graphs cannot be declared and initialized anywhere else (e.g. within a function).

```
graph_dec:
  GRAPH graph_def
```

5.1.2 "Graph Scope"

Within the braces of a graph's "scope" (e.g. at graph initialization, or within a function), nodes and edges can be declaratively added, edited, or removed from graphs:

```
graph_def:
  ID LBRACE gelem_def_list RBRACE

gelem_def_list:
| gelem_def_list gelem_def
```

5.1.3 Creating Graph Elements

Nodes can be added to a graph simply with a statement that declares them (with a unique identifier for each node):

```
gelem_def:
  gnode_def_list gelem_props SEMI

gnode_def_list
  ID
| gnode_def_list COMMA ID
```

Using an identifier that refers to an already-existing node simply references that existing node, e.g. as in:

```
  graph g { a; /* new node created */ a; /* no new node created */ }
```

Similarly, edges can be created by declaration:

```
gelem_def:
  gedge_def gelem_props SEMI

gedge_def:
  ID ID RARROW ID
```

Statements referring to edges follow the sequence ID1 ID2 RARROW ID3, where the first identifier is a node identifier, the second identifier is a "label" of an edge, and the third identifier is a node identifier. Upon declaration, an edge from ID1 to ID3 is created, which has label ID2. If either ID1 and/or ID3 are identifiers that did not previously refer to existing nodes in the graph, that/those node(s) are created (and the edge is created between them).
At most one edge of a particular label can exist between any two nodes of a graph.

5.1.4 Updating Graph Elements (Properties)

Edges can be updated: properties can be added and their integer values can be altered, given the following syntax (in addition to the above productions):

```
gelem_props:
| WHERE gprop_list
```

```
gprop_list:
  ID ASSIGN INTLIT
| gprop_list COMMA ID ASSIGN INTLIT
```

For example:

```
         graph g { a eats-> b; a eats-> b where pounds=7; }
```

Using the same productions above, nodes can also be similarly updated.

5.1.5 Deleting Graph Elements

Nodes and edges can be deleted with the following:

```
gelem_def:
  DEL gedge_def SEMI
| DEL gnode_def_list SEMI
```

5.2 Named Node Definitions

Named node definitions (as described in Section 3.7) are given by:

```
named_node_def:
  NODE ID ASSIGN ID IN pattern SEMI

pattern:
  nen_patt
| nen_patt WHERE expr

nen_patt:
  ID
| nen_patt ID RARROW ID
```

The second `ID` in `named_node_def` must be an identifier used in the `pattern` which refers to a node. The named node can then be used to search for and iterate over matching graph elements, globally.
Any number of such named nodes can be defined (only) at the highest (global) level of the program.

5.3 Functions

Functions are denoted according to:

```
func_dec:
  FUNC ID LPAREN formals_opt RPAREN RETURN typ_list LBRACE stmt_list RBRACE
```

```
| FUNC ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE

formals_opt
| formal_list

formal_list
  typ ID
| formal_list COMMA typ ID

typ_list:
  typ
| typ_list COMMA typ

typ: BOOL | INT | STRING | typ LIST | EDGE | NODE | TNNODE | GRAPH
```

5.3.1 Statements

Functions can contain a list of statements, which are executed in sequence. Statements are executed for their effects, and do not have values.

```
stmt_list:
| stmt_list stmt

stmt: expr SEMI | LBRACE stmt_list RBRACE
```

Declarations (i.e. a type and identifier) are supported by:

```
stmt: formal_list SEMI
```

Common control flow statements are given by:

```
stmt:
  RETURN expr_list SEMI
| IF expr LBRACE stmt_list RBRACE %prec NOELSE
| IF expr LBRACE stmt_list RBRACE ELSE LBRACE stmt_list RBRACE
```

And looping can be performed with:

```
stmt: WHILE expr LBRACE stmt_list RBRACE
```

Additionally, `for` statements allow for iterating over lists and graphs:

```
stmt:
  FOR typ ID IN expr LBRACE stmt_list RBRACE
| FOR typ ID IN pattern IN expr LBRACE stmt_list RBRACE
```

The first statement iterates over the given structure: in order for lists; in random order for graphs.
The second statement only applies to graphs (as the `expr`): pattern-matching essentially applies a filter on the graph, and then the `for` loop iterates over any matching types within that filtered (sub)graph.
Note: pattern-matching can also be used in the first statement above, if the given `typ` is a named node (pattern).

5.3.2 Graph Scope, Locally

This last statement permits graph elements to be added, updated, or deleted declaratively—as if they were in the “graph scope” (Section 5.1.2):

```
stmt: graph_def
```

Which can be used in the following way, for example:

```
         for node n in myGraph { myGraph { n is-> Tired; } }
```

By default, PLOG uses lexical scoping. For these graph scope statements, it's worth noting that: PLOG first checks the current environment for the value of each given identifier. If the identifier is found in the current environment, its value (e.g. a reference to an object) is used; if the identifier is not found in the current environment, the target graph is checked for the given identifier.
In the above example, if Tired is an identifier local to the current function (which contains the `for` statement) and it references an existing node in myGraph, the result of executing this code would be: all nodes are connected to the (existing) referenced node by edges with label “is”. If Tired is not found in the local environment, the graph scope statement is treated like any other graph scope statement (with respect to the Tired identifier): if it's a node that doesn't already exist, it's declared created.

5.3.3 Graph Access

Graph objects—more specifically: their elements—can be accessed from within functions. There exists one last expression which allows for such “graph accesses”:

```
expr: ID COLON LPAREN nodeoredge RPAREN
nodeoredge:
  ID
| ID ID RARROW ID
```

This expression searches for a single node or edge within a graph (given by the first identifier) and returns it—as an object of `node` or `edge` type. No pattern-matching is performed for graph accesses:

the input `nodeoredge` must be an identifier of type `node`, or it can be a sequence of: identifier for `node`, identifier for edge label (a `string`), RARROW (“`->`”), identifier for `node`.

As with graph scope statements, for graph accesses: PLOG first checks the current environment for the value of each given identifier. If the identifier is found in the current environment, its value (e.g. a reference to an object) is used; if the identifier is not found in the current environment, the target graph is checked for the given identifier.

It's worth explicitly noting that objects which reference graph elements in graphs—e.g. denoted by the identifier `n` in:

```
for node n in myGraph { /* … */ }
```

truly reference those same graph elements: (e.g. local—in a function) modifications to the reference are immediate modifications to the graph element (“as contained in the graph”).
Node and edge references “carry around” their properties (only; not incoming/outgoing edges, for example), so in this way, properties can be added and updated locally, and the changes are immediately reflected in the elements' graph. For example:

```
for node n in myGraph { n.newestProperty=1; n.oldProp=-1; }
```

6. Grammar

The full grammar of the current specification of PLOG is given below, where lower-case strings (potentially containing underscores) refer to non-terminals and upper-case strings refer to terminals. Non-terminals produce the empty string if the line directly below the non-terminal name begins with a pipe ('|') character. Terminals are defined further below, according to regular expressions.

```
program: decs EOF

decs:
|   decs named_node_def
|   decs graph_dec
|   decs func_dec

named_node_def:
    NODE ID ASSIGN ID IN pattern SEMI

graph_dec:
    GRAPH graph_def

func_dec:
    FUNC ID LPAREN formals_opt RPAREN RETURN typ_list LBRACE stmt_list RBRACE
|   FUNC ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
```

```
pattern:
    nen_patt
|   nen_patt WHERE expr

graph_def:
    ID LBRACE gelem_def_list RBRACE

gelem_def_list:
|   gelem_def_list gelem_def

gelem_def:
    gedge_def gelem_props SEMI
|   gnode_def_list gelem_props SEMI
|   DEL gedge_def SEMI
|   DEL gnode_def_list SEMI

gnode_def_list:
    ID
|   gnode_def_list COMMA ID

gedge_def:
    ID ID RARROW ID

gelem_props:
|   WHERE gprop_list

gprop_list:
    ID ASSIGN INTLIT
|   gprop_list COMMA ID ASSIGN INTLIT

formals_opt:
|   formal_list

formal_list:
    typ ID
|   formal_list COMMA typ ID

typ_list:
    typ
|   typ_list COMMA typ

typ:
    BOOL
|   INT
|   STRING
|   typ LIST
|   EDGE
|   NODE
|   TNNODE
|   GRAPH

stmt_list:
|   stmt_list stmt

stmt:
    expr SEMI
|   formal_list SEMI
|   LBRACE stmt_list RBRACE
|   RETURN expr_list SEMI
|   IF expr LBRACE stmt_list RBRACE %prec NOELSE
|   IF expr LBRACE stmt_list RBRACE ELSE LBRACE stmt_list RBRACE
|   FOR typ ID IN pattern IN expr LBRACE stmt_list RBRACE
|   FOR typ ID IN expr LBRACE stmt_list RBRACE
|   WHILE expr LBRACE stmt_list RBRACE
|   graph_def

exprs_opt:
|   expr_list

expr_list:
    expr
|   expr_list COMMA expr

expr:
    expr PLUS expr
|   expr MINUS expr
|   MINUS expr %prec NEG
|   expr TIMES expr
|   expr DIVIDE expr
|   expr ASSIGN expr
|   expr EQ expr
|   expr NEQ expr
|   expr LT expr
|   expr LEQ expr
|   expr GT expr
|   expr GEQ expr
|   expr OR expr
|   expr AND expr
|   expr DOT ID
|   PRINT LPAREN STRLIT exprs_opt RPAREN
|   APPEND LPAREN expr COMMA expr RPAREN
|   REMOVE LPAREN expr COMMA expr RPAREN
|   LENGTH LPAREN expr RPAREN
|   ID COLON LPAREN nodeoredge RPAREN
|   ID LPAREN exprs_opt RPAREN
|   LBRACK exprs_opt RBRACK
|   LPAREN expr RPAREN
|   NIL LPAREN typ RPAREN
|   INF
|   literal
|   ID

nodeoredge:
    ID
|   ID ID RARROW ID

literal:
    TRUE
|   FALSE
|   INTLIT
|   STRLIT

PLUS: '+'
MINUS: '-'
TIMES: '*'
DIVIDE: '/'
EQ: “==”
NEQ: “!=”
LT: '<'
LEQ: “<=”
GT: '>'
GEQ: “>=”
ASSIGN: '='
COMMA: ','
DOT: '.'
NOT: '!'
LPAREN: '('
RPAREN: ')'
LBRACE: '{'
RBRACE: '}'
LBRACK: '['
RBRACK: ']'
SEMI: ';'
COLON: ':'
RARROW: “->”
NIL: “NIL”
INF: “INF”
TNNODE: “node:” ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
ID: ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*
INTLIT: ['0'-'9']+
```

EOF refers to the end-of-file symbol. STRLIT is produced by a sequence of characters as described in section 2.5.3. All other terminals are matched by their lower-case strings.

7. Example Program

The following program exhibits most aspects of the PLOG language:

```
graph Neighborhood
{
      Me;
      You, Them where dummyProperty = -5;
      /* Properties applied to all nodes in the above node list */

      Me where nodeNum = 1;
      You where nodeNum = 2;
      Them where nodeNum = 3;
      Me distance-> You where value = 2;
      Me distance-> Them where value = 6;
      You distance-> Them where value = 3;
}

func getClosestNodeToSource( node list nodeList ) return node
{
      node closest = NIL(node);
      int min_dist = INF;

      for node n in nodeList
      {
            int ndist = n.distFromSource;
            if ndist < min_dist {
                  min_dist = ndist;
                  closest = n; }
      }

      return closest;
}

func computeMinDistsFromSource( graph G, node source )
{
      node list unvisited;
      for node n in G {
            n.distFromSource = INF;
            append( n, unvisited ); }
      source.distFromSource = 0;

      while length( unvisited ) > 0
      {
            node closest = getClosestNodeToSource( unvisited, G );
            remove( closest, unvisited );

            for node neighbor in closest distance-> neighbor in G
            {
                  int neighborDist = G:( closest distance-> neighbor ).value;
                  /* A property named “value” */
                  int newDist = closest.distFromSource + neighborDist;
                  if new_dist < neighbor.distFromSource {
                        neighbor.distFromSource = newDist;
                        neighbor.nodeToSource = closest.nodeNum; }
            }
      }
}

func main()
{
      computeMinDistsFromSource( Neighborhood, Neighborhood:(Me) );

      for node n in Neighborhood {
            print( “%d is %d away.\n” closest.nodeNum, closest.distFromSource );

            if n == Neighborhood:(You) {
                  Neighborhood{ Me thanks-> n; }
            }
      }
}
```