PLOG

Final Report

Wesley Bruning
(wab2125)

Table of Contents

1. Introduction

This report describes the PLOG language and the work that was carried out to develop it.

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 a language created for academic purposes. It 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 includes details of the capabilities and limitations of the PLOG language.

The report is organized into the following prescribed sections: an informal introduction to PLOG (given as a tutorial), its language definition and reference material, a description of the plan executed to complete the PLOG project, discussion of the language’s evolution, details of the translator architecture, information about PLOG testing, and some concluding remarks about the project itself. Additionally, the full PLOG compiler source code is appended at the end of this document.

2. Tutorial

PLOG is largely an imperative language, similar in syntax to C and Python, for instance, while other (more “declarative”) parts of PLOG are similar to the DOT language—used for describing graphs, their elements (nodes and edges), and graph elements’ properties.

In this section, we’ll introduce ourselves to PLOG by developing an example program highlighting its defining features.

2.1 Creating a Graph

For our example program, we’ll create a graph structure which encodes some information about a neighborhood of characters. Graphs are declared and defined at the global level only, but they and their graph elements (nodes and edges) can be modified elsewhere throughout the program.

We’ll call our graph “Neighborhood” and populate it with three nodes: Me, You, and Them:

```
graph Neighborhood
{
	Me;
	You, Them where dummyProperty = 5;
```

Graph elements are created, modified, and deleted within a graph’s block (between two brackets). We’ll be closing this graph’s block soon, after we’re done our initial additions to it.

The syntax of graph declarations/definitions (and all other components of PLOG) are detailed in the Language Reference section (3) of this document, but it’s enough for now to know that we’ve just declared that there exists a node named “Me” in the Neighborhood graph, and also two other nodes —“You” and “Them”—which both possess a property called “dummyProperty”, which is set to 5 (for both nodes). That is: when you declare multiple nodes in the same statement (as we did for You and Them), any properties specified apply to all nodes in the statement.

We’ll give all three nodes an additional identifying property:

```
	Me where nodeNum = 1;
	You where nodeNum = 2;
	Them where nodeNum = 3;
```

With each statement above, we declare that there exists a node with a particular name, and that it should have a particular property (or properties). If a node with the given name already exists, it’s updated; if no such node already exists, it’s created and given the specified properties.

All graph element (i.e. node or edge) properties in PLOG must have integer values (if the properties exist).

Now we’ll add three edges to the graph, representing some unit of distance between our characters:

```
	Me distance-> You where value = 2;
	Me distance-> Them where value = 6;
```

```
      You distance-> Them where value = 3;
}
```

Every edge has a single direction. The right arrow ("`->`") denotes the one-way direction of an edge.

Every edge must also have a "label". In this case, all three of the edges have the label "distance". At most one edge with a particular label can exist between two particular nodes—in a particular direction. For example, there can be at most one "distance" edge going from Me to You, but there could also be a "distance" edge going from You to Me.

Similar to before, each statement declares the existence of an edge (between two nodes). If an edge with the given label in the given direction doesn't exist between the two nodes, it's created; otherwise, it's updated. If a node doesn't exist with a given name, it's also created. So if we declared an edge and neither of its endpoint nodes existed, not only would the edge be created, but so would its two nodes.

Only one edge can be declared per statement. Properties specified in edge declarations apply to the edge only (not the nodes).

With our Neighborhood populated, we'll move onto the next section: defining a name that will allow us to more concisely search and update graph elements.

2.2 Defining a Named Node

As mentioned in the Introduction, PLOG allows you to perform pattern-matching. Only a basic form of pattern-matching is implemented in the specification of PLOG, but it can still be very useful. We'll see an example of a simple form of pattern-matching next, but we'll be exposed to more interesting examples later in the report.

For our Neighborhood, it might be helpful to distinguish who is a "true neighbor". Let's suppose that 2 units of our distance metric denote such a neighbor (roughly, someone who is "not too close"—like in your home—and "not too far"). The pattern used to identify such a person could be:

```
x where distFromSource = 2
```

This looks very similar to our node declarations above, and for good reason: as you'll discover later, `x` in the above case will match any node in the graph, and we use the same property-setting syntax as before to "filter" nodes—according to the specified properties and their values. The result of applying this pattern (e.g. to a graph structure) returns all of the nodes which have a property "distFromSource" with a value of 2.

We can declare and define a name for this pattern (only) at the global level of a PLOG program:

```
node neighbor = x in x where distFromSource = 2;
```

The "`(x) in`" notation is used to reflect the fact that we're specifying which node in the pattern is to be considered the "neighbor" node. In this case, there's only one node in the pattern, so the choice is trivial. Also note that the use of the identifier `x` in this example is arbitrary; another identifier could have been chosen.

This “named node” will be used later in our program. Next, let’s implement a shortest-distance-calculating algorithm in PLOG.

2.3 Accessing and Updating the Graph

Now we will begin to use functions to access and update the graph we’ve created. Functions in PLOG are very similar to functions in many other languages: they can accept arguments as input, execute statements (some with side effects), and output a value.

We’ll define two functions now: `getClosestNodeToSource` and `computeMinDistsFromSource`.

The first function accepts a list of nodes, examines the “distFromSource” property of each node in the list, and returns the node that has the lowest value for that property:

```
func getClosestNodeToSource( node list nodeList ) return node
{
	node closest;
	int minDist, ndist;

	closest = NIL(node);
	minDist = INF;

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

	return closest;
}
```

There are a few things we’ll note right now:

1) Variables used in a block (with the exception of those declared within a `for` statement) must be declared at the start of that block—before any other statements.
2) `NIL(node)` refers to a node that is “null”.
3) INF refers to (positive) infinity.

You also see that we can access properties of graph elements (e.g. nodes) outside of graph definition blocks (e.g `n.distFromSource`). Next, we’ll see that we can use functions to also update the graph and its elements.

Our second function, `computeMinDistsFromSource`, will accept as input a graph and a “source” node, then use a variant of Dijkstra’s shortest-path-finding algorithm to update the nodes in the graph to possess a “distFromSource” property, containing the shortest distance from each node to the source node:

```
func computeMinDistsFromSource( graph G, node source )
{
	node list unvisited;
	node closest;
	int neighborDist, newDist;

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

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

		for node neighbor in closest distance-> neighbor in G
		{
			neighborDist = G:( closest distance-> neighbor ).value;
			newDist = closest.distFromSource + neighborDist;
			if new_dist < neighbor.distFromSource {
				neighbor.distFromSource = newDist;
				neighbor.nodeToSource = closest.nodeNum; }
		}
	}
}
```

In this function, we see a number of features of PLOG in action:

1) Our Neighborhood graph is being (or rather: will be) referenced with a different name.
2) Node properties are being accessed and updated.
3) Some PLOG standard library functions are being called (`append`, `length`, and `remove`).
4) A `for` loop uses a pattern to filter results for iteration.
5) And a reference to an edge object is directly accessed from a graph.

Let's talk about the second `for` loop in this function:

```
for node neighbor in closest distance-> neighbor in G
```

The `closest` identifier refers to a particular (local variable) node object—as determined by PLOG's static scoping rules (discussed in the Language Reference). The identifiers `distance` and `neighbor`, however, are not local variables. Because of this, both `distance` and `neighbor` are "pattern objects": they will help to return any subgraph in the given graph (`G`) which matches the full pattern. In this case, the pattern essentially matches all of the subgraphs in `G` containing two nodes connected by an edge labeled "distance", where `closest` is the "tail end" of the edge. The `for` loop then iterates over all of the "head end" nodes, as denoted by the "`(neighbor) in`" notation.

We then see that the loop accesses each such "distance"-labeled edge's "value" property; the expression

`G:( closest distance-> neighbor )` returns an `edge` object, which is only used to access its "value" property's value.

Note that several graph elements are having properties accessed and set within the function. If a graph element property is being set (e.g. as in `source.distFromSource = 0`), it's treated like a declaration in the graph definition block: if the property doesn't already exist, it's created; otherwise, it's updated. The update applies immediately to the graph element.

2.4 Tying it All Together

With the bulk of our program completed, we'll use one last function to kick off the whole process.

Every PLOG program must have (exactly) one "main" function. This is essentially where code execution begins when a program is run. The `main` function we'll define will call `computeMinDistsFromSource`, print the shortest distances from the "Me" node to all nodes, and also print out who meets our previous definition of a "true neighbor":

```
func main()
{
	computeMinDistsFromSource( Neighborhood, Neighborhood:(Me) );
	for node n in Neighborhood {
		print( "%d is %d away.\n", n.nodeNum, n.distFromSource );

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

	for node:neighbor n in Neighborhood {
		print( "%d is a \"true neighbor\".\n", n.nodeNum );
	}
}
```

Here, we see the fourth standard library function—`print`—being used with a variable number of arguments. And our "neighbor" named node is finally put to use: in effect, the `for` loop iterates over all of the nodes which match the pattern defined earlier by the `neighbor` named node.

And that's it! The net result is that our program prints out:

```
1 is 0 away.
2 is 2 away.
3 is 5 away.
2 is a "true neighbor".
```

Which is what we expect—with 1 being the `nodeNum` for Me, 2 being the `nodeNum` for You, and 3 being the `nodeNum` for Them: the shortest distance from Me to Them was determined by our program to be 5, rather than the distance of 6 that was initially explicitly encoded.

2.5. Running Your Program

To create the PLOG compiler and run a PLOG program, the basic requirements are OCaml (to build the PLOG compiler) and Python. The reference implementation of PLOG was developed in OCaml and compiles to Python. Both Python 2 and 3 are supported as runtime environments.

In particular, you can compile the program we constructed in this tutorial by copying all of the `"Courier New"`-style font text not italicized above (or by copying the full program at the end of this report). The source code for this and other examples is also included with the PLOG compiler.

Given the PLOG compiler source code, it can be built with the included Makefile (after navigating to the source code directory):

```
$ make
```

This will generate the `plog` compiler, which can be used to compile a PLOG program into Python:

```
$ ./plog -c < [PLOG_program_file].plg > [output_file].py
```

The generated program file can then be executed with Python:

```
$ python [output_filepath].py
```

Or executed directly after compilation with:

```
$ ./plog -c < [PLOG_program_filepath] | python
```

If a PLOG program is syntactically invalid, a parsing error will occur after executing `plog`. Similarly, if a PLOG program is semantically invalid, a (more descriptive) error will occur after executing `plog`. If a runtime error occurs, it occurs when Python executes the generated code.

3. Language Reference

This section details the PLOG language from “low-level” concepts to “higher-level” concepts. 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).

3.1 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.

3.1.1 Comments

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

3.1.2 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.

3.1.3 Identifiers

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

3.1.4 Keywords

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

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

3.1.5 Literals

3.1.5.1 Boolean Literals

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

3.1.5.2 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.

3.1.5.3 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`
Form feed	`\f`
Tab	`\t`
Backspace	`\b`
Double-quote	`\"`
Backslash	`\\`

Characters not appearing in the above escape sequences are not permitted to appear immediately after a backslash ('\'); it is an error to have such a character immediately after a backslash.

3.1.6 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 3.3.3 and 3.3.4.

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

3.1.6.1 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.

3.1.6.2 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.

3.1.6.3 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

3.1.7 Separators

Additionally, the following characters also have specific meanings (referenced in Sections 3.3 and 3.4):

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

The "->" string is reserved for denoting edges, as described in Sections 3.2 and 3.4.

3.2 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.

3.2.1 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 3.2.6) and graph element declarative statements (Section 3.4.1). In order to determine the type of an identifier, it must be explicitly noted upon declaration.

3.2.2 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:

3.2.2.1 Boolean

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

3.2.2.2 Integer

`int`: Represents an integer value. The maximum value and minimum value are currently determined by the "max_int" and "min_int" values—respectively—of OCaml (on the system compiling `plog`).

3.2.2.3 String

`string`: Represents a string value: a sequence of zero or more ASCII characters (as described in section 3.1.5.3).

3.2.3 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:

3.2.3.1 Graph

`graph`: Represents a graph structure, which can possess multiple nodes and edges between those nodes (discussed in Section 3.4.1).

3.2.3.2 Node

`node`: Represents a node object. Nodes can possess properties (see Section 3.2.5 below).

3.2.3.3 Edge

`edge`: Represents an edge object, which must exist between two (not necessarily distinct) objects. Edges can possess properties (see Section 3.2.5 below).

3.2.4 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 3.6).

3.2.5 Properties

Node and edge objects may possess multiple properties, each with an integer value (only). Property names must be given as a string, identifying the property name.
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 an existing graph element, NIL(int) is returned. If attempting to get the value of a property for a graph element that doesn’t exist, 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 3.3.4 for the syntax of the “Property Access” operator.

3.2.6 Graph Patterns

Graph patterns facilitate searching for and iterating over graph elements meeting specific characteristics. The syntax of graph patterns is given in Section 3.4.2, an example of which is:

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

Which denotes that the pattern will match: a node that “likes” a node which “dislikes” a node.

3.2.7 Named Nodes

Graph patterns which return a node can be given a name. Such a named pattern is referred to as a “named node”, the syntax of which is also given in Section 3.4.2. An example of a named node is:

```
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.

3.2.8 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 3.4.3.

3.3 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.

3.3.1 Reference Grammar Syntax

In the following sections, 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 3.5. Operator precedence and associativity is defined in Section 3.1.6.

3.3.2 Atomic Expressions

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

ID refers to an identifier, INF refers to the keyword `INF` which represents (positive) 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 taken to be of type integer), respectively.

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

3.3.3 “Unary” Expressions

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

```
expr:
  MINUS expr %prec NEG
| NOT expr
| LPAREN expr RPAREN
| NIL LPAREN typ RPAREN
```

The first refers to integer negation, the second refers to logical negation, while the third 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.

3.3.4 “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 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.
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 3.1.6.2 and 3.2.5.

3.3.5 Function Calls

Functions (mentioned in Section 3.8) can be called as given by the expression below:

```
expr: ID LPAREN exprs_opt RPAREN
```

3.3.6 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: LIST typ LBRACK exprs_opt RBRACK
```

3.4 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.

3.4.1 Graphs

3.4.1.1 Graph Initialization

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

```
graph_dec: GRAPH ID LBRACE gelem_def_list RBRACE
```

3.4.1.2 “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
```

3.4.1.3 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.

3.4.1.4 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
| ID ASSIGN MINUS INTLIT
| gprop_list COMMA ID ASSIGN INTLIT
| gprop_list COMMA ID ASSIGN MINUS 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.

3.4.1.5 Deleting Graph Elements

Nodes and edges can be deleted with the following:

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

3.4.2 Named Node Definitions

Named node definitions (as described in Section 3.2.7) are given by:

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

pattern:
  nen_patt
| nen_patt WHERE gprop_list

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.

3.4.3 Functions

Functions are denoted according to:

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

formals_opt
| formal_list

formal_list
  typ ID
| formal_list COMMA typ ID

vdecls_opt:
| vdecls_opt vdecls

vdecls: typ id_list SEMI

id_list:
  ID
| id_list COMMA ID

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

Calling a function with incorrect arguments (i.e. wrong number of arguments or order of types of arguments) results in an error at compile time.

3.4.3.1 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

block: LBRACE vdecls_opt stmt_list RBRACE
```

As indicated by the structure of block, variable declarations—if variables are declared in a block at all—must appear at the beginning of the block, before any other statements.
Declarations (i.e. a type and identifier) are supported by `vdecls` (defined above).

The syntax for assignments is:

```
stmt: expr ASSIGN expr SEMI
```

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.

Common control flow statements are given by:

```
stmt:
  RETURN expr SEMI
| IF expr block %prec NOELSE
| IF expr block ELSE block
```

And looping can be performed with:

```
stmt: WHILE expr block
```

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

```
stmt:
  FOR typ ID IN expr block
| FOR typ ID IN pattern IN expr block
```

Both statements iterate over the structure given by the expression `expr`: in order for lists; in random order for graphs. The expression may only evaluate to lists or graphs.
The `for` loop iterates over the structure, returning values matching the given type `typ` (and—optionally—pattern, discussed below), where they can be used in the `for` loop's block.
The second statement makes explicit use of pattern-matching: pattern-matching essentially applies a filter on the list or graph structure, after which the `for` loop iterates over any matching types within that filtered substructure.
Pattern-matching can also be used "implicitly"—if the given `typ` is a named node. In this case, any explicit pattern is first applied to the expression's structure, and the `for` loop iterates over any nodes matching the given named node's pattern in the (sub)structure.
One restriction holds when the `typ` is given as an `edge`: in this case, if a pattern is present in the `for` loop statement, there must be exactly 3 identifiers (corresponding to a pattern like: node1 edge`->` node2), and the given `ID` must match with the middle (edge) identifier in the pattern. The `for` loop then iterates over all of the edges that connect the two nodes.

3.4.3.2 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 3.4.1.2):

```
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.

3.4.3.3 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 graph_elem_id RPAREN
graph_elem_id:
  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 `graph_elem_id` 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. If the specified graph element is not contained within the graph, `NIL(t)` is returned, where `t` is the type of the graph access (a node or edge).

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; }
```

3.5 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 ID LBRACE gelem_def_list RBRACE

func_dec:
  FUNC ID LPAREN formals_opt RPAREN RETURN typ LBRACE vdecls_opt stmt_list RBRACE
| FUNC ID LPAREN formals_opt RPAREN LBRACE vdecls_opt 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
|   ID ASSIGN MINUS INTLIT
|   gprop_list COMMA ID ASSIGN INTLIT
|   gprop_list COMMA ID ASSIGN MINUS INTLIT

vdecls_opt:
|   vdecls_opt vdecls

vdecls:
   typ id_list SEMI

id_list:
   ID
|  id_list COMMA ID

formals_opt:
|   formal_list

formal_list:
    typ ID
|   formal_list COMMA typ ID

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

```
stmt_list:
|  stmt_list stmt

stmt:
   expr SEMI
|  expr ASSIGN expr SEMI
|  RETURN expr SEMI
|  IF expr block %prec NOELSE
|  IF expr block ELSE block
|  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 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
|  NOT expr
|  ID COLON LPAREN graph_elem_id RPAREN
|  ID LPAREN exprs_opt RPAREN
|  LIST typ LBRACK exprs_opt RBRACK
|  LPAREN expr RPAREN
|  NIL LPAREN typ RPAREN
|  INF
|  literal
|  ID

graph_elem_id:
   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.

3.6 Standard Library

Included below are descriptions of the standard library functions, along with their input parameters and return values (where applicable).

It is an error to redefine a standard library function—to declare a function with the same name as a standard library function.

3.6.1 print()

```
print( string, args )
```

The “print” function prints a given string literal. It is unique in PLOG in that it can accept a variable number of arguments—and of mixed types. The first parameter must be a string. The string 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, the variable number arguments (indicated by the placeholder `args`) must contain exactly enough corresponding values—expressions evaluating to the expected types. The value of the first expression (beyond the initial string argument) 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.

3.6.2 append()

```
append( type, type list )
```

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.

3.6.3 remove()

```
remove( type, type list )
```

The “remove” function accepts a variable and a list, and removes the first instance of the input variable from the input list, if it exists. Attempting to remove an element from a list in which it does not exist results in no change made to the list—and no errors generated.

3.6.4 length()

```
length( type list )
```

The “length” function returns the (integer) number of elements in a given list.

4. Project Plan

The design and development of PLOG was carried out solely by the author. Because of his other interests and commitments (a full-time position, a part-time position, and other coursework), careful use of time was necessary for the successful completion of the project. The summer session during which this was completed was also relatively condensed, requiring the language to be limited in scope and project milestones to be completed in relatively quick succession.

Three milestones were prescribed. In addition to those items (highlighted in bold below), further milestones were planned by the author, which can be roughly decomposed as:

Initial language design

Initial OCaml exploration

Project proposal (Language "definition" version 0)

Further Python exploration

Language definition version 1

Scanner/tokenizer creation

Parser creation

Integration test pipeline created

Language reference manual

AST generator (via Parser) creation

Language definition version 2

Static semantic analyzer creation

Code generator creation

Final integration tests

Project final report

Initially, I had little experience with several aspects of the project: previously, I had never developed a programming language, I had never designed a compiler, I had never programmed in OCaml (the prescribed compiler language), and I had little experience with Python (the chosen target language). I used this project to gain experience in all of those aspects—though I knew they had to be absorbed quickly.

The creation of PLOG and its compiler occurred predominantly during the ~2 months from the acceptance of its proposal to the deadline for the project final report. The scanner and parser were largely completed (pending some minor adjustments later) before the Language Reference Manual was created, and the grammar was checked to be unambiguous (with ocamlyacc). The full testing pipeline

was put in place by the time of the Language Referenace Manual, as well.

Shortly after that, the parser was made to generate the full AST. The parser was then verified by printing out the generated AST for any given program: traversing the AST and printing out each node resulted in a program that was essentially identical (minus whitespace, etc.) to the input program. A static semantic checker was then used to enforce the remaining “rules” of PLOG—that the input program was indeed a semantically-valid PLOG program.

After the components performing PLOG program analysis were completed, the code generator was developed (discussed more in Section 7). Once the code generator was successfully generating Python code, the testing pipeline was frequently used to validate the generated code.

5. Language Evolution

PLOG was designed largely for academic purposes—to learn about and gain experience with (programming) language design and implementation. It was established early on that it would be a language meant to interact with graphs—modeling abstract entities and the relations between them. Ideas for a more complex language—though in the same vein—were considered. But PLOG was only destined to explore a subset of language features that I thought were interesting, such as (very basic) pattern-matching within graph structures, and how names can aid the interaction with those structures (e.g. iterating or searching over them).

The first “definition” of PLOG—roughly described in the PLOG Proposal—included some features that did not continue on to the definition of PLOG by the time of the Language Reference Manual. The main reason for the removal of those features—such as enforcing “graph invariants” (assertions that would always hold true in a graph)—were that they would have required more time than I had to implement them. Inexperienced with programming language design and implementation, I had little idea of the effort required to implement certain components. It was important to pick a clear subset of features, which would be interesting and challenging enough to implement in the time given.

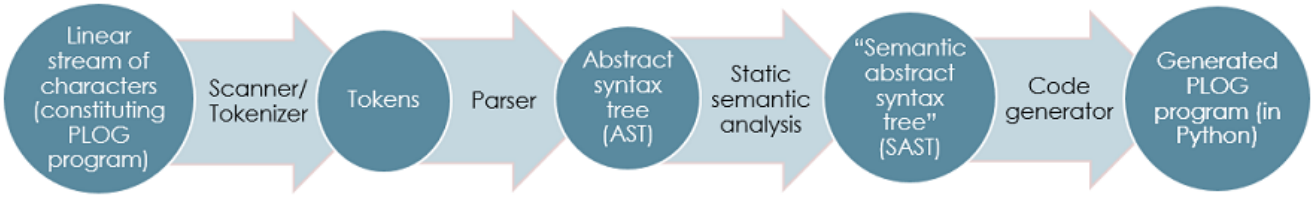
After the Language Reference Manual (LRM), a few more simplifications were made, but the language was largely fixed in place. For example, functions could no longer return multiple values (and of potentially differing types), owing to the fact that a significant-enough amount of code would have had to have been added to support that facet. Restrictions on graph patterns (e.g. how many times an identifier can appear in a pattern) were made for a similar reason.

Most of the time after the LRM was spent implementing the “rules” of the language—the static semantic analyzer—and generating the code that ensured the behavior of the language that was defined.

6. Translator Architecture

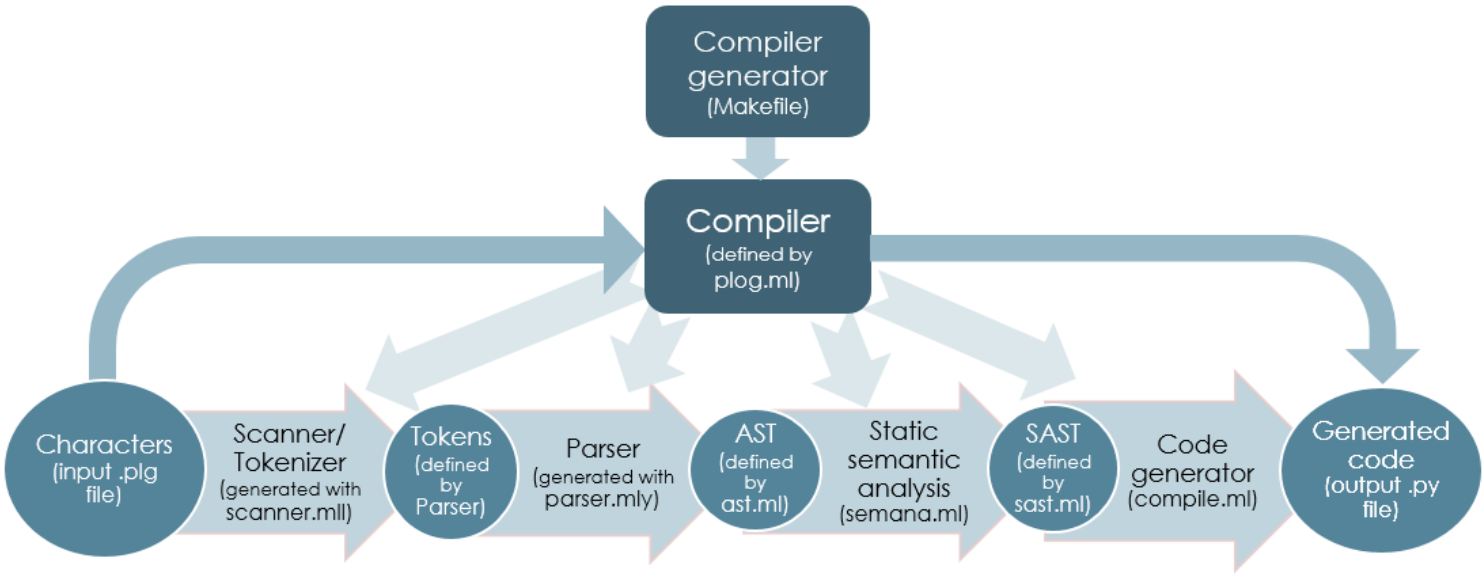
6.1 Walkthrough

The PLOG compiler followed the general architecture discussed in the course:



In particular, a structure deemed the "semantic abstract syntax tree" (SAST)—which is generated as a result of (successful) static semantic analysis—was recommended. Generating code from the SAST was found to be useful, as discussed below.

The reference implementation used the following specific components:



A PLOG program undergoes the following steps from source code to executable code:

1) Source code is input to the compiler (generated by the Makefile).
2) The compiler initiates the scanning/tokenization of the source file: the scanner, generated with scanner.mll and ocamllex, produces a series of tokens (defined as part of the parser).
3) The compiler then inputs the sequence of tokens to the parser, which was generated with parser.mly and ocamlyacc. If parsing is successful, an abstract syntax tree (AST) is constructed.
4) Static semantic analysis is then initiated by the compiler, using the procedures laid out in semana.ml. If the PLOG program is semantically valid, a SAST is constructed.
5) From the SAST, the compiler carries out the translation into Python, as detailed in compile.ml.

6) Finally, a Python program semantically equivalent (for practical purposes) to the input PLOG program is output (specifically: to standard output). The generated Python can then be executed in the Python runtime environment.

6.2 SAST Differences

Creating the scanner was relatively straightforward, as was the parser, once the grammar was specified. Static semantic analysis essentially traverses the AST—starting from the "program" root—and ensures that the input program is valid according to PLOG's semantics: type checking is performed, use of identifiers is checked, and numerous rules about declarations, scoping, graph accesses, and patterns are enforced.

As a result of this procedure, a SAST is generated, which is almost entirely identical to the AST structure, with the exception that it also contains important details about local variables' identifiers—in different scopes—and the types of certain expressions. In fact, the SAST was designed to be only "as helpful as it needed to be": when PLOG was first defined, it was noted that the scoping behavior (and variable references) could be carried out in Python as long as particular segments of code only had certain additional information about local variables and some variables' types. Specifically: `for` loops, "graph access" expressions, and "graph definition" statements all had the potential to reference local variables (as with other expressions and statements). However, these expressions and statements in particular followed unique scoping rules: `for` loops could possess patterns, which could themselves reference local variables or "pattern objects"; graph access expressions could similarly refer to local variables, but if an identifier didn't match with one, it would reference an element within the given graph; and graph definition statements could similarly (to graph access expressions) reference local variables. In all of these cases, the identifiers corresponding to local variables had to be known, so the SAST augmented the AST with string lists at the appropriate places. Generated Python code could then simply be prefixed (essentially) with a string that denoted the identifier was for a local variable.

The one other location where the SAST differs from the AST is again in the `for` loop. Depending on whether or not a `for` loop iterates over a variable corresponding to a graph or a list, the code is different: iterating over a graph (i.e. an identifier referring to a graph object) requires different code to access the graph's nodes or edges, compared to iterating over a list (whether it's a list literal or an identifier referring to a local list variable). So the `for` loop in the SAST also includes a Boolean value, indicating whether or not the loop is iterating over a graph structure.

6.3 Code Generation and the Runtime Environment

Code generation was, in many ways, aided by the power of a runtime environment like Python (at least for developing a compiler in a relatively short amount of time); support for objects, data structures like dictionaries and lists, and automatic garbage collection helped to reduce the amount of code (and time) necessary to implement the code generator. However, with a high-level language such as Python, there are many details of the language that—if you're not aware of them—can result in unexpected behavior.

There were a number of aspects to consider when compiling to Python. One small but obvious example was the fact that whitespace has a more important role to play in Python, and that had to be considered when generating code. (This issue was easily solved by—more or less—generating the program as a list of strings, and applying a function that added tabs to those strings at appropriate places, e.g. blocks of code.) Additionally, the order of operations/expression evaluation had to be explicitly denoted (e.g. in instances where a PLOG programmer had not explicitly done so). In this case, outputting expressions appropriately nested with parentheses was a simple and effective solution.

Knowing the runtime environment also requires knowing the particular version of the runtime environment. Amongst Python programmers, Python is currently commonly known to be "split" largely into two different versions: Python 2 and Python 3. The differences between the two, while relatively small in number, can be significant for certain programs: behavior of division and iteration over structures such as dictionaries—facilities both used in PLOG programs—vary, for instance.

Originally, Python 2 was the only runtime environment planned to be supported, but relatively minor changes were incorporated so that PLOG programs would execute under both Python 2 and Python 3.

6.4 (Lacking) Optimization

Optimization was a low priority in the development of PLOG; the performance of generated code was a relatively minor concern. From the set of simple test cases conducted for this report, the performance of PLOG programs running under Python was entirely satisfactory to the author, however.

Mechanisms were put in place (in hopes) to ensure that the generated Python code was valid for all PLOG programs. Attempts to optimize special cases were not made—even for some obvious improvements. Nor was generated code made to be easy for a human to parse: to simplify the architecture, all generated Python code is located within a single Python file—which includes standard library functions and Python classes which store graph data—and even such "underlying" functions that are unused in a particular PLOG program remain in the generated code.

7. Test Plan and Scripts

The test plan for PLOG was simple, but strived to be comprehensive. A test "pipeline" was established early on in the project—enabling changes to PLOG to be tested with a single command. Professor Stephen Edwards' "testall.sh" script, taken from his MicroC compiler, was used to automate testing procedures, mainly: compiling all PLOG program test cases and comparing compiled program output to expected output.

Test cases were constructed by going through the PLOG Language Reference Manual and creating PLOG programs which exercised the stated behavior. Additionally, more complex programs (such as the one covered in the Tutorial/Section 2) were developed, in effect performing "integration testing".

Test case-related files used by the author to test PLOG are included later in this report, as well as packaged with the PLOG source code archive.

8. Conclusions

This course and its project were both very rewarding to me. I placed a relatively high value on many aspects of the course prior to signing up for it, and after learning about and working through the details of a (albeit simple) programming language's design and development, those valuations have not faltered. In fact, my plans to learn more about language will carry on—perhaps now with even more "gusto".

I learned first-hand many considerations that must be made when designing a programming language. First and foremost: know your languages (the source, target, and implementation language(s)). And on a related note: know the details of your runtime environment(s). At a minimum, you need to be sure that your translation is semantically equivalent (for practical purposes). Optimizations can come later.

Particular features of a language that seem simple may not be so straightforward to implement. But I also learned that "there is no magic": details of how a human-interpretable language is—and can be—translated from a static specification into an executable procedure are more clear to me. And the understanding of how symbols can be translated into action is a valuable thing.

9. Full Code Listing

9.1 AST – ast.ml

```
type unop = Neg | Not

type biop = Add | Sub | Mul | Div | Eq | Neq | Lt | Leq | Gt | Geq | And | Or

type typ =
  TEdge
| TNode
| TNNode of string
| TGraph
| TInt
| TBool
| TStr
| TList of typ
| TVoid

type typdname = typ * string
```

```
type graph_elem =
  NodeId of string
| EdgeId of string * string * string

type expr =
  IntLit of int
| BoolLit of bool
| StrLit of string
| Lst of typ * expr list
| Id of string
| Unop of unop * expr
| Biop of expr * biop * expr
| Call of string * expr list
| GraphAccess of string * graph_elem
| Property of expr * string    (* NOTE: Or consider expr * expr with use of []
chars *)
| Nil of typ
| Inf

type patt = {
  pids : string list;
  preds : (string * int) list; (* NOTE: Consider supporting id.prop = val *)
}

type graph_stmt =
  GraphSet of graph_elem list * (string * int) list  (* (string * int) is for
properties *)
| GraphDel of graph_elem list

type stmt =
  Assign of expr * expr
| Block of typdname list * stmt list
| Expr of expr
| Return of expr
| If of expr * stmt * stmt
| For of typ * string * patt * expr * stmt
| While of expr * stmt
| GraphDef of string * graph_stmt list

type node_decl = {
  nname : string;
  nidpatt : string;
  npatt : patt;
}

type graph_decl = {
  gname : string;
  gbody : graph_stmt list;
}

type func_decl = {
  fname : string;
  freturns : typ;
  formals : typdname list;
  flocals : typdname list;
  fbody : stmt list;
}
```

```
type program = node_decl list * graph_decl list * func_decl list

let string_of_unop = function
  Neg -> "-"
| Not -> "!"

let string_of_biop = function
  Add -> "+"
| Sub -> "-"
| Mul -> "*"
| Div -> "/"
| Eq -> "=="
| Neq -> "!="
| Lt -> "<"
| Leq -> "<="
| Gt -> ">"
| Geq -> ">="
| And -> "and"
| Or -> "or"

let rec string_of_typ = function
  TEdge -> "edge"
| TNode -> "node"
| TNNode(s) -> "node:"^ s
| TGraph -> "graph"
| TInt -> "int"
| TBool -> "bool"
| TStr -> "string"
| TList(t) -> string_of_typ t ^" list"
| TVoid -> "void"

module Typ =
  struct
    type t = typ
    let compare t1 t2 =
      String.compare (string_of_typ t1) (string_of_typ t2)
    let equal t1 t2 = compare t1 t2 = 0
  end

let string_of_gelem = function
  NodeId(s) -> s
| EdgeId(n1, e, n2) -> n1 ^" "^ e ^"-> "^ n2

let rec string_of_expr = function
  IntLit(i) -> string_of_int i
| BoolLit(true) -> "true"
| BoolLit(false) -> "false"
| StrLit(s) -> "\""^ String.escaped s ^"\""
| Lst(t, el) -> string_of_typ t ^" ["^ String.concat ", " (List.map
string_of_expr el) ^"]"
| Id(s) -> s
| Unop(o, e) -> "("^ string_of_unop o ^" "^ string_of_expr e ^")"
| Biop(e1, o, e2) -> "("^ string_of_expr e1 ^" "^ string_of_biop o ^" "^
string_of_expr e2 ^")"
| Call(f, el) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^")"
(*| GraphScope(e, s) -> string_of_expr e ^"::"^ s
| ElemProp(s1, s2) -> s1 ^"."^ s2 *)
| GraphAccess(s, ge) -> s ^":("^ string_of_gelem ge ^")"
```

```
| Property(e, s) -> string_of_expr e ^"."^ s
(* | Property(e1, e2) -> string_of_expr e1 ^"["^ string_of_expr e2 ^"]" *)
| Nil(t) -> "NIL("^ string_of_typ t ^")"
| Inf -> ""

let string_of_propassn (s, i) = s ^" = "^ string_of_int i

let string_of_patt pat =  (* This currently doesn't display right arrows ("->")
for edges. *)
  String.concat " " (List.rev pat.pids) ^" where "^ String.concat ", " (List.map
string_of_propassn pat.preds)
(* NOTE: string_of_patt uses List.rev before printing the pids because we
effectively store them in "reverse order" as a result of parsing. *)

let string_of_typdname (typ, id) = string_of_typ typ ^" "^ id

let string_of_typdname_list tnlist =
  (if List.length tnlist > 0 then (String.concat "; " (List.map
string_of_typdname tnlist)) ^";" else "") ^"\n"

let string_of_gstmt = function  (* NOTE: This currently doesn't check if the
below lists (e.g. `sil`) are empty *)
  GraphSet(gel, sil) -> String.concat ", " (List.map string_of_gelem gel) ^"
where "^ String.concat ", " (List.map string_of_propassn sil)
| GraphDel(gel) -> String.concat ", " (List.map string_of_gelem gel)

(* NOTE: Update (all) pretty-printing to nicely tab/indent lines, as is done
with with the generated code. "Time permitting." *)
let rec string_of_stmt = function
  Assign(e1, e2) -> string_of_expr e1 ^" = "^ string_of_expr e2 ^";\n"
| Block(tnl, sl) ->
    "{\n"^ string_of_typdname_list tnl ^
    String.concat "" (List.map string_of_stmt sl) ^"}\n"
| Expr(expr) -> string_of_expr expr ^ ";\n";
| Return(expr) -> "return "^ string_of_expr expr ^";"
| If(e, s, Block([], [])) -> "if "^ string_of_expr e ^" {\n"^ string_of_stmt s
^"\n}\n"
| If(e, s1, s2) -> "if "^ string_of_expr e ^" {\n"^ string_of_stmt s1 ^"\n} else
{\n"^ string_of_stmt s2 ^"\n}\n"
| For(t, str, pat, e, stat) ->
    "for "^ string_of_typ t ^" "^ str ^" in "^ string_of_patt pat ^" in "^
string_of_expr e ^" {\n"^ string_of_stmt stat ^"\n}\n"
| While(e, s) -> "while "^ string_of_expr e ^" {\n"^ string_of_stmt s ^"\n}\n"
| GraphDef(s, gsl) -> s ^" {\n"^ String.concat "\n" (List.map string_of_gstmt
gsl) ^"\n}\n"

let string_of_formals formals = String.concat ", " (List.map string_of_typdname
formals)


let string_of_node_decl node_decl =
  "node "^ node_decl.nname ^" = "^ node_decl.nidpatt ^" in "^ string_of_patt
node_decl.npatt ^";"

let string_of_graph_decl graph_decl =
  "graph "^ graph_decl.gname ^" {\n"^
  (if List.length graph_decl.gbody > 0 then "  "^ String.concat "\n  " (List.map
string_of_gstmt graph_decl.gbody)
   else "") ^"\n}\n"
```

```
let string_of_func_decl f_decl =
  "func "^ f_decl.fname ^"("^ string_of_formals f_decl.formals ^") "^
  if f_decl.freturns <> TVoid then "return "^ string_of_typ f_decl.freturns else
"" ^
  " {\n"^ string_of_typdname_list f_decl.flocals ^
  (if List.length f_decl.fbody > 0 then "  "^ String.concat "\n  " (List.map
string_of_stmt f_decl.fbody)
   else "") ^"\n}\n"

let string_of_program (nodes, graphs, funcs) =
  (if List.length nodes > 0 then String.concat "\n\n" (List.map
string_of_node_decl (List.rev nodes)) ^ "\n\n"
  else "") ^
  (if List.length graphs > 0 then String.concat "\n\n" (List.map
string_of_graph_decl (List.rev graphs)) ^ "\n\n"
  else "") ^
  (if List.length funcs > 0 then String.concat "\n\n" (List.map
string_of_func_decl (List.rev funcs)) ^ "\n\n"
  else "") ^ "\n"

type symbol_table = {
    parent : symbol_table option;
    variables : typdname list;
}

type translation_environment = {
    scope: symbol_table;
    return_type : typ;
}
```

9.2 SAST – sast.ml

```
type unop = Neg | Not

type biop = Add | Sub | Mul | Div | Eq | Neq | Lt | Leq | Gt | Geq | And | Or

type typ =
  TEdge
| TNode
| TNNode of string
| TGraph
| TInt
| TBool
| TStr
| TList of typ
| TVoid

type typdname = typ * string

type graph_elem =
  NodeId of string
| EdgeId of string * string * string

type expr =
  IntLit of int
| BoolLit of bool
| StrLit of string
| Lst of typ * expr list
```

```
| Id of string
| Unop of unop * expr
| Biop of expr * biop * expr
| Call of string * expr list
| GraphAccess of string * graph_elem * string list
| Property of expr * string
| Nil of typ
| Inf

type patt = {
  pids : string list;
  preds : (string * int) list;
}

type graph_stmt =
  GraphSet of graph_elem list * (string * int) list
| GraphDel of graph_elem list

type stmt =
  Assign of expr * expr
| Block of typdname list * stmt list
| Expr of expr
| Return of expr
| If of expr * stmt * stmt
| For of typ * string * patt * expr * bool * stmt * string list
| While of expr * stmt
| GraphDef of string * graph_stmt list * string list

type node_decl = {
  nname : string;
  nidpatt : string;
  npatt : patt;
}

type graph_decl = {
  gname : string;
  gbody : graph_stmt list;
}

type func_decl = {
  fname : string;
  freturns : typ;
  formals : typdname list;
  flocals : typdname list;
  fbody : stmt list;
}

type program = node_decl list * graph_decl list * func_decl list

let string_of_biop = function
  Add -> "+"
| Sub -> "-"
| Mul -> "*"
| Div -> "//"
| Eq -> "=="
| Neq -> "!="
| Lt -> "<"
| Leq -> "<="
| Gt -> ">"
```

```
| Geq -> ">="
| And -> "and"
| Or -> "or"
```

9.3 Scanner/Tokenizer – scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*" { comment lexbuf }
| '"' { strlit (Buffer.create 16) lexbuf }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| "==" { EQ }
| "!=" { NEQ }
| '<' { LT }
| "<=" { LEQ }
| '>' { GT }
| ">=" { GEQ }
| '=' { ASSIGN }
| ',' { COMMA }
| '.' { DOT }
| '!' { NOT }
| '(' { LPAREN }
| ')' { RPAREN }
| '{' { LBRACE }
| '}' { RBRACE }
| '[' { LBRACK }
| ']' { RBRACK }
| ';' { SEMI }
| ':' { COLON }
| "->" { RARROW }
| "where" { WHERE }
| "and" { AND }
| "or" { OR }
| "if" { IF }
| "else" { ELSE }
| "for" { FOR }
| "in" { IN }
| "while" { WHILE }
| "func" { FUNC }
| "return" { RETURN }
| "bool" { BOOL }
| "int" { INT }
| "string" { STRING }
| "list" { LIST }
| "edge" { EDGE }
| "node" { NODE }
| "graph" { GRAPH }
| "del" { DEL }
| "true" { TRUE }
| "false" { FALSE }
| "NIL" { NIL }
| "INF" { INF }
| "node:" (['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm) { TNNODE(lxm)
}
```

```
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| ['0'-'9']+ as lit { INTLIT(int_of_string lit) }
| eof { EOF }
| _ as char { raise (Failure("Illegal character "^ Char.escaped char)) }

and strlit buf = parse
  | '"' { STRLIT(Buffer.contents buf) }
  | '\\' '\\' { Buffer.add_char buf '\\'; strlit buf lexbuf }
  | '\\' 'b' { Buffer.add_char buf '\b'; strlit buf lexbuf }
  | '\\' 'f' { Buffer.add_char buf '\012'; strlit buf lexbuf }
  | '\\' 'n' { Buffer.add_char buf '\n'; strlit buf lexbuf }
  | '\\' 'r' { Buffer.add_char buf '\r'; strlit buf lexbuf }
  | '\\' 't' { Buffer.add_char buf '\t'; strlit buf lexbuf }
  | '\\' '"' { Buffer.add_char buf '\"'; strlit buf lexbuf }
  | [^ '"' '\\']+
    { Buffer.add_string buf (Lexing.lexeme lexbuf);
      strlit buf lexbuf
    }
  | _ { raise (Failure("Illegal string character: "^ Lexing.lexeme lexbuf)) }
  | eof { raise (Failure("String is not terminated")) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

9.4 Parser – parser.mly

```
%{
open Ast;;
let get1 (x,_,_) = x;;
let get2 (_,x,_) = x;;
let get3 (_,_,x) = x;;
 %}

%token EOF PLUS MINUS TIMES DIVIDE EQ NEQ LT LEQ GT GEQ COLON
%token ASSIGN COMMA DOT NOT LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK
%token SEMI RARROW AND OR IF ELSE FOR IN WHILE WHERE FUNC RETURN
%token BOOL INT STRING LIST EDGE NODE GRAPH DEL TRUE FALSE NIL INF
%token <int> INTLIT
%token <string> STRLIT ID TNNODE

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT LEQ GT GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc LBRACK RBRACK
%left NEG NOT
%left DOT

%start program
%type <Ast.program> program

%%
```

```
program:
  decs EOF { $1 }

decs:
  { [], [], [] }
| decs named_node_def { ($2 :: get1 $1), get2 $1, get3 $1 }
| decs graph_dec { get1 $1, ($2 :: get2 $1), get3 $1 }
| decs func_dec { get1 $1, get2 $1, ($2 :: get3 $1) }

named_node_def:
  NODE ID ASSIGN ID IN pattern SEMI { { nname = $2; nidpatt = $4; npatt = $6 } }

graph_dec:
  GRAPH ID LBRACE gelem_def_list RBRACE { { gname = $2; gbody = List.rev $4 } }

func_dec:
  FUNC ID LPAREN formals_opt RPAREN RETURN typ LBRACE vdecls_opt stmt_list
RBRACE
    { { fname = $2; freturns = $7; formals = $4; flocals = List.rev $9; fbody =
List.rev $10 } }
| FUNC ID LPAREN formals_opt RPAREN LBRACE vdecls_opt stmt_list RBRACE
    { { fname = $2; freturns = TVoid; formals = $4; flocals = List.rev $7; fbody
= List.rev $8 } }

pattern:
  nen_patt                    { { pids = $1; preds = [] } }
| nen_patt WHERE gprop_list   { { pids = $1; preds = List.rev $3 } }

nen_patt:
  ID                      { [$1] }
| nen_patt ID RARROW ID   { $4::($2::$1) }

graph_def:
  ID LBRACE gelem_def_list RBRACE { GraphDef($1, List.rev $3) }

gelem_def_list:
  { [] }
| gelem_def_list gelem_def { $2 :: $1 }

gelem_def:
  gedge_def gelem_props SEMI { GraphSet([$1], $2) }
| gnode_def_list gelem_props SEMI { GraphSet(List.rev $1, $2) }
| DEL gedge_def SEMI { GraphDel([$2]) }
| DEL gnode_def_list SEMI { GraphDel(List.rev $2) }

gnode_def_list:
  ID { [NodeId($1)] }
| gnode_def_list COMMA ID { NodeId($3) :: $1 }

gedge_def:
  ID ID RARROW ID { EdgeId($1, $2, $4) }

gelem_props:
  { [] }
| WHERE gprop_list { List.rev $2 }

gprop_list:
  ID ASSIGN INTLIT { [($1, $3)] }
| ID ASSIGN MINUS INTLIT { [($1, -$4)] }
```

```
| gprop_list COMMA ID ASSIGN INTLIT { ($3, $5) :: $1 }
| gprop_list COMMA ID ASSIGN MINUS INTLIT { ($3, -$6) :: $1 }

vdecls_opt:
                      { [] }
| vdecls_opt vdecls  { $2 @ $1 }

vdecls:
  typ id_list SEMI   { List.map (fun x -> ($1, x)) $2 }

id_list:
  ID                 { [$1] }
| id_list COMMA ID   { $3 :: $1 }

formals_opt:
  { [] }
| formal_list { List.rev $1 }

formal_list:
  typ ID  { [($1,$2)] }
| formal_list COMMA typ ID { ($3,$4)::$1 }

typ:
  BOOL { TBool }
| INT { TInt }
| STRING { TStr }
| typ LIST { TList($1) }
| EDGE { TEdge }
| NODE { TNode }
| TNNODE { TNNode($1) }
| GRAPH { TGraph }

stmt_list:
  { [] }
| stmt_list stmt { $2::$1 }

stmt:
  expr SEMI      { Expr $1 }
| expr ASSIGN expr SEMI { Assign($1, $3) }
| RETURN expr SEMI { Return $2 }
| IF expr block %prec NOELSE { If($2, $3, Block([], [])) }
| IF expr block ELSE block { If($2, $3, $5) }
| FOR typ ID IN pattern IN expr block { For($2, $3, $5, $7, $8) }
| FOR typ ID IN expr block { For($2, $3, {pids=[];preds=[]}, $5, $6) }
| WHILE expr block { While($2, $3) }
| graph_def        { $1 }

block:
  LBRACE vdecls_opt stmt_list RBRACE { Block(List.rev $2, List.rev $3) }

exprs_opt:
  { [] }
| expr_list { List.rev $1 }

expr_list:
  expr { [$1] }
| expr_list COMMA expr { $3::$1 }

expr:
```

```
  expr PLUS   expr { Biop($1, Add, $3) }
| expr MINUS  expr { Biop($1, Sub, $3) }
| MINUS expr %prec NEG { Unop(Neg, $2) }
| expr TIMES  expr { Biop($1, Mul, $3) }
| expr DIVIDE expr { Biop($1, Div, $3) }
| expr EQ     expr { Biop($1, Eq, $3) }
| expr NEQ    expr { Biop($1, Neq, $3) }
| expr LT     expr { Biop($1, Lt, $3) }
| expr LEQ    expr { Biop($1, Leq, $3) }
| expr GT     expr { Biop($1, Gt, $3) }
| expr GEQ    expr { Biop($1, Geq, $3) }
| expr OR     expr { Biop($1, Or, $3) }
| expr AND    expr { Biop($1, And, $3) }
| expr DOT ID      { Property($1, $3) }
| NOT expr         { Unop(Not, $2) }
| ID COLON LPAREN graph_elem_id RPAREN { GraphAccess($1, $4) }
| ID LPAREN exprs_opt RPAREN { Call($1, $3) }
| LIST typ LBRACK exprs_opt RBRACK { Lst($2, $4) }
| LPAREN expr RPAREN { $2 }
| NIL LPAREN typ RPAREN { Nil($3) }
| INF              { Inf }
| literal          { $1 }
| ID               { Id($1) }

literal:
  TRUE             { BoolLit(true) }
| FALSE            { BoolLit(false) }
| INTLIT           { IntLit($1) }
| STRLIT           { StrLit($1) }

graph_elem_id:
  ID { NodeId($1) }
| ID ID RARROW ID { EdgeId($1, $2, $4) }
```

9.5 Static Semantic Analyzer – semana.ml

```
open Ast
module StringMap = Map.Make(String)

let translate (nodes, graphs, functions) =

    let named_nodes = List.map (fun node -> node.nname) nodes in

    (* Raise an exception if the given list has a duplicate *)
    let report_duplicate exceptf lst =
        let rec helper = function
            n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
            | _ :: t -> helper t
            | [] -> ()
        in helper (List.sort compare lst)
    in

    (* Check named nodes. Ensure no duplicate declarations. *)
    report_duplicate (fun n -> "Duplicate named node "^ n)
        (List.map (fun node -> node.nname) nodes);

    (* Node identifers in a pattern must appear exactly once. Edges can appear
multiple times. *)
    let check_idpatt id isedge patt =
```

```
        if List.exists (fun x -> x = id) patt.pids then ()
        else raise (Failure (id ^" must appear in its pattern."));

        let rec count target occurrences = function
            [] -> occurrences
            | x :: xs -> if x = target then count target (occurrences+1) xs
                else count target occurrences xs in
        let nodeids =
            let rec get_nodeids accum = function
                [] -> accum
                | n::[] -> n::accum  (* Order doesn't matter for this check; it
can be reversed. *)
                | n::e::xs -> get_nodeids (n::accum) xs
            in get_nodeids [] patt.pids
        in
        List.iter (fun x -> if count x 0 patt.pids = 1 then ()
            else raise (Failure (x ^" must appear exactly once in a pattern."))
            ) (if not isedge then id::nodeids else nodeids);

        if isedge then (if List.exists (fun x -> x = id) nodeids
            then raise (Failure (id ^" must be in an edge position in a
pattern."))
            else () )
        else (if List.exists (fun x -> x = id) nodeids then ()
            else raise (Failure (id ^" must be in a node position in a
pattern.")))
    in
    List.iter (fun node -> check_idpatt node.nidpatt false node.npatt) nodes;

    report_duplicate (fun g -> "Duplicate graph declaration "^ g)
        (List.map (fun g -> g.gname ) graphs);

    let check_assign lvaluet rvaluet err =
        if lvaluet = rvaluet then lvaluet else raise err
    in

 (* Check functions: *)

    (* Check that a function named "main" is defined *)
    if not (List.mem "main" (List.map (fun fd -> fd.fname) functions))
    then raise (Failure ("Main function not found")) else ();

    (* Check that stdlib functions are not re-defined *)
    if List.mem "print" (List.map (fun fd -> fd.fname) functions)
    then raise (Failure ("Function 'print' may not be defined")) else ();
    if List.mem "append" (List.map (fun fd -> fd.fname) functions)
    then raise (Failure ("Function 'append' may not be defined")) else ();
    if List.mem "remove" (List.map (fun fd -> fd.fname) functions)
    then raise (Failure ("Function 'remove' may not be defined")) else ();
    if List.mem "length" (List.map (fun fd -> fd.fname) functions)
    then raise (Failure ("Function 'length' may not be defined")) else ();

    report_duplicate (fun n -> "Duplicate function "^ n)
        (List.map (fun func -> func.fname) functions);

    (* The definitions for the "standard library" functions are generated at
compile-time.
       The following basic information is only used for the static semantic
analysis. *)
```

```
    let built_in_funcs = [ { fname="print"; formals=[(TStr, "x")];
freturns=TVoid; flocals=[]; fbody=[] };
        { fname="append"; formals=[(TVoid, "x"); (TVoid, "y")]; freturns=TVoid;
flocals=[]; fbody=[] };
        { fname="remove"; formals=[(TVoid, "x"); (TVoid, "y")]; freturns=TVoid;
flocals=[]; fbody=[] };
        { fname="length"; formals=[(TVoid, "y")]; freturns=TInt; flocals=[];
fbody=[] } ]
    in

    let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd
m)
        StringMap.empty (built_in_funcs @ functions)
    in

    let function_decl s = try StringMap.find s function_decls
        with Not_found -> raise (Failure ("Unrecognized function "^ s))
    in

    (* NOTE: There's definitely a better way of doing this in OCaml. *)
    let sast_biop = function
        Add -> Sast.Add | And -> Sast.And | Div -> Sast.Div | Eq -> Sast.Eq
        | Geq -> Sast.Geq | Gt -> Sast.Gt | Leq -> Sast.Leq | Lt -> Sast.Lt
        | Mul -> Sast.Mul | Neq -> Sast.Neq | Or -> Sast.Or | Sub -> Sast.Sub in
    let sast_unop = function
        Neg -> Sast.Neg | Not -> Sast.Not in
    let sast_ge = function
        NodeId(a) -> Sast.NodeId(a) | EdgeId(a,b,c) -> Sast.EdgeId(a,b,c) in
    let rec sast_typ = function
        TEdge -> Sast.TEdge | TNode -> Sast.TNode | TNNode(s) -> Sast.TNNode(s)
        | TGraph -> Sast.TGraph | TInt -> Sast.TInt | TBool -> Sast.TBool
        | TStr -> Sast.TStr | TList(t) -> Sast.TList(sast_typ t) | TVoid ->
Sast.TVoid in
    let sast_tnl tnl = List.map (fun (t,n) -> (sast_typ t, n)) tnl in
    let sast_patt patt = { Sast.pids = patt.pids; Sast.preds = patt.preds } in
    let sast_nds nds = List.map (fun nd ->
        { Sast.nname = nd.nname; Sast.nidpatt = nd.nidpatt; Sast.npatt =
sast_patt nd.npatt }
        ) nds in
    let graph_stmt = function
        GraphSet(gel, sil) -> Sast.GraphSet(List.map sast_ge gel, sil)
        | GraphDel(gel) -> Sast.GraphDel(List.map sast_ge gel)
    in
    let sast_gds gds = List.map (fun gd ->
        { Sast.gname = gd.gname; Sast.gbody = List.map graph_stmt gd.gbody })
gds
    in

    let check_node_name_exists name = if List.exists (fun n-> n = name)
named_nodes
        then () else raise (Failure (name ^" is not a defined named node."))
    in

    let translate_function func =

        let check_named_node = function TNNode(n) -> ignore
(check_node_name_exists n)
            | _ -> () in
```

```
        (* Check if any node types used in the list of formal parameters or
local variables are undefined *)
        List.iter (fun (typ, _) -> check_named_node typ; ()) (func.formals @
func.flocals);

        report_duplicate (fun n-> "In "^ func.fname ^", duplicate formal "^ n)
            (List.map snd func.formals);
        report_duplicate (fun n-> "In "^ func.fname ^", duplicate local "^ n)
            (List.map snd func.flocals);

        let rec find_variable (scope : symbol_table) name =
            try List.find (fun (_, s) -> s = name) scope.variables
            with Not_found -> match scope.parent with
                Some(parent) -> find_variable parent name
                | _ -> raise Not_found
        in

        let rec list_variables_names (scope : symbol_table) =
            List.map snd scope.variables @ (match scope.parent with
                Some(parent) -> list_variables_names parent
                | _ -> [])
        in

        let rec expr env = function
            Biop(e1, o, e2) as ex -> let e1' = expr env e1 and e2' = expr env e2
in
                let t1 = fst e1' and t2 = fst e2' in
                let biop_type = (match o with
                Add when t1 = TInt && t2 = TInt -> TInt
                | And when t1 = TBool && t2 = TBool -> TBool
                | Div when t1 = TInt && t2 = TInt -> TInt
                | Eq when t1 = t2 -> TBool
                | Geq when t1 = TInt && t2 = TInt -> TBool
                | Gt when t1 = TInt && t2 = TInt -> TBool
                | Leq when t1 = TInt && t2 = TInt -> TBool
                | Lt when t1 = TInt && t2 = TInt -> TBool
                | Mul when t1 = TInt && t2 = TInt -> TInt
                | Neq when t1 = t2 -> TBool
                | Or when t1 = TBool && t2 = TBool -> TBool
                | Sub when t1 = TInt && t2 = TInt -> TInt
                | _ -> raise (Failure ("In "^ func.fname ^", illegal binary
operator "^
                    string_of_typ t1 ^" "^ string_of_biop o ^" "^ string_of_typ
t2 ^
                    " in "^ string_of_expr ex))) in
                (biop_type, Sast.Biop(snd e1', sast_biop o, snd e2'))
            | BoolLit(b) -> (TBool, Sast.BoolLit(b))
            | Call(s, el) as call -> (match s with
                (* NOTE: Use of standard library functions would more generally
be validated
                   like the use of any other functions. For now, we e.g. hard-
code variable number of
                   argument support (as used in print()), since PLOG doesn't
provide this support generally. *)
                "append" -> (match el with
                    e1 :: e2 :: [] -> (
                        let lst = expr env e2 in (match fst lst with
                        TList(t) -> let ele = expr env e1 in
                            if t <> fst ele then raise (Failure ("In "^
```

```
func.fname ^", illegal append, "^
                            string_of_typ (fst lst) ^" append "^ string_of_typ
(fst ele) ^" in "^
                            "append( "^ string_of_expr e1 ^", "^ string_of_expr
e2 ^" )"))
                            else (TVoid, Sast.Call(s, [snd ele; snd lst]))
                        | _ -> raise (Failure ("In "^ func.fname ^", expected
list second parameter, in "^
                            "append( "^ string_of_expr e1 ^", "^ string_of_expr
e2 ^" )"))) )
                    | _ -> raise (Failure ("In "^ func.fname ^", append expected
2 parameters, of type: t "^
                        "and TList(t)")) )
                | "length" -> (match el with
                    e :: [] -> let lst = expr env e in (match fst lst with
                        TList(_) -> (TInt, Sast.Call(s, [snd lst]))
                        | _ -> raise (Failure ("In "^ func.fname ^", length
expects 1 TList parameter")) )
                    | _ -> raise (Failure ("In "^ func.fname ^", length expects
1 TList parameter")) )
                | "print" -> (match el with
                    StrLit(str) :: params -> (
                        (* Go through s to create a list of types, based on the
presence of %b, %d, and %s *)
                        let rec tlist subs =
                            let idx = try String.index subs '%' with Not_found
-> -1 in
                            (* NOTE: idx+1 or idx+2 may be at/beyond the end of
subs. *)
                            if idx = -1 then [] else (match subs.[idx+1] with
                                | 'b' -> (try TBool :: tlist (String.sub subs
(idx+2) ((String.length subs) - (idx+2)))
                                    with Invalid_argument(_) -> print_endline
"invalid bool"; TBool :: [])
                                | 'd' -> (try TInt :: tlist (String.sub subs
(idx+2) ((String.length subs) - (idx+2)))
                                    with Invalid_argument(_) -> print_endline
"invalid int"; TInt :: [])
                                | 's' -> (try TStr :: tlist (String.sub subs
(idx+2) ((String.length subs) - (idx+2)))
                                    with Invalid_argument(_) -> print_endline
"invalid str"; TStr :: [])
                                | _ -> (try tlist (String.sub subs (idx+1)
((String.length subs) - (idx+1)))
                                    with Invalid_argument(_) -> print_endline
"not %b, %d, or %s"; [])
                                ) in
                        let lst = tlist str in
                        if List.length params <> List.length lst then
                            raise (Failure ("In "^ func.fname ^", print requires
same number of escaped values"))
                        (* Then compare this list with params, one by one,
making sure that each match *)
                        (* Use fact that List.exists2 returns Invalid_argument
if list lengths don't match? *)
                        else (let pexprs = List.map (expr env) params in
                            if List.exists2 (fun a b -> fst a <> b) pexprs lst
then
                                raise (Failure ("In "^ func.fname ^", print
```

```
requires matching escaped value types"))
                            else (TVoid, Sast.Call(s, Sast.StrLit(str) ::
List.map snd pexprs)) ) )
                    | _ -> raise (Failure ("In "^ func.fname ^", print expects a
StrLit and optional expr list")) )
                | "remove" -> (match el with
                    e1 :: e2 :: [] -> (
                        let lst = expr env e2 in (match fst lst with
                        TList(t) -> let ele = expr env e1 in
                            (if t <> fst ele then raise (Failure ("In "^
func.fname ^
                                ", type mismatch in list remove: "^
string_of_typ (fst lst) ^
                                " remove "^ string_of_typ (fst ele) ^" in:
remove( "^
                                string_of_expr e1 ^", "^ string_of_expr e2
^" )"))
                                else (TVoid, Sast.Call(s, [snd ele; snd lst])) )
                        | _ -> raise (Failure ("In "^ func.fname ^", expected
list second parameter: "^
                            "remove( "^ string_of_expr e1 ^", "^ string_of_expr
e2 ^" )"))) )
                    | _ -> raise (Failure ("In "^ func.fname ^", remove expected
2 parameters, of type: t "^
                        "and TList(t)")) )

                (* Otherwise, call a non-built-in function: *)
                | _ -> (let func_decl = function_decl s in
                    if List.length el <> List.length func_decl.formals then
                        raise (Failure ("In "^ func.fname ^", expecting "^
string_of_int
                        (List.length func_decl.formals) ^" arguments in call to
"^ string_of_expr call))
                    else let pexprs = List.map (expr env) el in
                        List.iter2 (fun (formal, formal_name) exp -> let actual
= fst exp in
                        ignore (check_assign formal actual (Failure ("In "^
func.fname ^
                        ", illegal actual argument "^ string_of_typ actual ^",
expected "^
                        string_of_typ formal ^" for formal parameter '"^
formal_name ^"' in call to "^
                        string_of_expr call)))) func_decl.formals pexprs;
                        (func_decl.freturns, Sast.Call(s, List.map snd pexprs)))
)
            | GraphAccess(s, ge) as ex -> let vdecl = try find_variable
env.scope s
                with Not_found -> raise (Failure ("In "^ func.fname ^",
undeclared identifier "^ s)) in
                if fst vdecl <> TGraph then raise (Failure ("In "^ func.fname
^", "^ s ^" is not a graph in "^
                    string_of_expr ex))
                else ( (match ge with
                    NodeId(_) -> TNode
                    | EdgeId(_,_,_) -> TEdge), Sast.GraphAccess(s, sast_ge ge,
list_variables_names env.scope) )
            | Id(s) -> let vdecl = try find_variable env.scope s
                with Not_found -> raise (Failure ("In "^ func.fname ^",
undeclared identifier "^ s)) in
```

```
                (fst vdecl, Sast.Id(s))
            | Inf -> (TInt, Sast.Inf)
            | IntLit(i) -> (TInt, Sast.IntLit(i))
            | Lst(t, el) -> (TList(t), Sast.Lst(sast_typ t, List.map snd
(List.map (expr env) el))) (* Use composition? *)
            | Nil(t) -> (t, Sast.Nil(sast_typ t))
            | Property(e, s) -> let e' = expr env e in (match fst e' with
                TNode | TEdge | TNNode(_) -> (TInt, Sast.Property(snd e', s))
                | x -> raise (Failure ("In "^ func.fname ^", objects of type "^
string_of_typ x ^
                    " may not have properties.")))
            | StrLit(s) -> (TStr, Sast.StrLit(s))
            | Unop(o, e) as ex -> let e' = expr env e in (match o with
                Neg when fst e' = TInt -> (TInt, Sast.Unop(sast_unop o, snd e'))
                | Not when fst e' = TBool -> (TBool, Sast.Unop(sast_unop o, snd
e'))
                | _ -> raise (Failure ("In "^ func.fname ^", illegal unary
operator "^
                    string_of_typ (fst e') ^" "^ string_of_unop o ^" in "^
string_of_expr ex)))
        in

        let check_bool_expr env e =
            let e' = expr env e in
            if fst e' <> TBool then raise (Failure ("In "^
            func.fname ^", expected a boolean expression in: "^ string_of_expr
e))
            else snd e'
        in

        let rec stmt env = function
            Assign(e1, e2) as st -> let lt = expr env e1 and rt = expr env e2 in
                check_assign (fst lt) (fst rt) (Failure ("In "^ func.fname ^",
illegal assignment "^
                string_of_typ (fst lt) ^" = "^ string_of_typ (fst rt) ^" in "^
string_of_stmt st));
                Sast.Assign(snd lt, snd rt)
            | Block(tnl, sl) ->
                List.iter (fun (typ, _) -> check_named_node typ; ()) tnl;
                let scope' = { parent = Some( env.scope ); variables = tnl } in
                let env' = { env with scope = scope' } in
                Sast.Block(sast_tnl tnl, List.map (fun s -> stmt env' s) sl)
            | Expr e -> Sast.Expr (snd (expr env e))
            | For(t, s, {pids=[];preds=[]}, e, (Block(tnl,sl) as b)) ->
                (* TODO: NOW: We need to add (t,s) to the local vars AFTER this
for loop, somehow, as well. *)
                let e' = expr env e in
                (*let newb = Block( (t,s)::tnl, sl ) in*)
                (* scope' and env' could be defined below, so that they're not
defined if fst e' doesn't match a TGraph or TList *)
                let scope' = { env.scope with variables =
(t,s)::env.scope.variables } in
                let env' = { env with scope = scope' } in
                (match fst e' with
                TGraph -> ignore (match t with
                    TNode | TEdge -> ()
                    | TNNode(nn) -> ignore (check_node_name_exists nn)
                    | _ -> raise (Failure ("For loop must have node, edge, or
named node")));
```

```
                    Sast.For(sast_typ t, s, {Sast.pids=[];Sast.preds=[]}, snd
e', true, stmt env' b, list_variables_names env.scope)
                | TList(lt) ->
                    (* For the purposes of iterating over lists, nodes and named
nodes are both considered nodes. *)
                    let temptyp = function TNNode(_) -> TNode | other -> other
in
                    if temptyp t <> temptyp lt then raise (Failure ("For loop
type doesn't match list type")) else
                    Sast.For(sast_typ t, s, {Sast.pids=[];Sast.preds=[]}, snd
e', false, stmt env' b, list_variables_names env.scope)
                | _ -> raise (Failure ("For loop must iterate over a graph or
list")))
            | For(t, s, p, e, (Block(tnl,sl) as b)) -> if List.exists (fun x ->
x = s) p.pids then
                let e' = expr env e in
                let scope' = { env.scope with variables =
(t,s)::env.scope.variables } in
                let env' = { env with scope = scope' } in
                (match fst e' with
                TGraph -> (match t with
                    TNode -> ()
                    | TEdge -> ignore (if List.length p.pids <> 3 then
                        raise (Failure ("For loops iterating with edges must
have exactly 3 identifiers in the pattern, "^
                            "of the form: node1 edge-> node2")) else ())
                    | TNNode(nn) -> ignore (check_node_name_exists nn)
                    | _ -> raise (Failure ("For loops over graphs must iterate
with nodes, edges, or named nodes")));
                    ignore (check_idpatt s (t = TEdge) p);
                    Sast.For(sast_typ t, s, sast_patt p, snd e', true, stmt env'
b, list_variables_names env.scope)
                | TList(lt) ->
                    let temptyp = function TNNode(_) -> TNode | other -> other
in
                    if temptyp t <> temptyp lt then raise (Failure ("For loop
type doesn't match list type"))
                    else ( (match t with
                        TNode | TNNode(_) -> ()
                        | TEdge -> ignore (if List.length p.pids <> 3 then
                            raise (Failure ("For loops iterating with edges must
have exactly 3 identifiers in the pattern, "^
                                "of the form: node1 edge-> node2")) else ())
                        | _ -> raise (Failure ("For loops with patterns can only
iterate with nodes, edges, or named nodes")) );
                        ignore (check_idpatt s (t = TEdge) p);
                        Sast.For(sast_typ t, s, sast_patt p, snd e', false, stmt
env' b, list_variables_names env.scope) )
                | _ -> raise (Failure ("For loop must iterate over a graph or
list")))
                else raise (Failure ("For loop identifier must exist in the
pattern"))
            | GraphDef(s, gsl) -> Sast.GraphDef(s, List.map graph_stmt gsl,
list_variables_names env.scope)
            | If(e, b1, b2) -> Sast.If(check_bool_expr env e, stmt env b1, stmt
env b2)
            | Return e -> let e' = expr env e in if fst e' = func.freturns then
Sast.Return (snd e')
                else raise (Failure ("In "^ func.fname ^", return type is "^
```

```
string_of_typ (fst e') ^
                "; but expecting "^ string_of_typ func.freturns))
            | While(e, s) -> Sast.While(check_bool_expr env e, stmt env s)
        in

        let newenv = {
            scope = { parent = None;
                variables = (List.map (fun gd -> (TGraph, gd.gname)) graphs) @
func.formals};
            return_type = func.freturns }
        in

        { Sast.fname = func.fname; Sast.freturns = sast_typ func.freturns;
Sast.formals = sast_tnl func.formals;
          Sast.flocals = sast_tnl func.flocals; Sast.fbody =
            [stmt newenv (Block(func.flocals, func.fbody))] }

    in (sast_nds nodes, sast_gds graphs, List.map translate_function functions);
```

9.6 Code Generator – compile.ml

```
open Sast

let prelude = [ "#from plog import *";
    "from __future__ import print_function";  (* NOTE: __future__ is only needed
for calling Python's print(). *)
    "from collections import defaultdict";
    "def PLOG_append( elem, container ):\n"^
    "\tcontainer.append( elem )";
    "def PLOG_remove( elem, container ):\n"^
    "\tif elem in container:\n"^
    "\t\tcontainer.remove( elem )";
    "def PLOG_length( container ):\n"^
    "\treturn len( container )";
    "def PLOG_print( string, *args ):\n"^

"\tprint( string.replace(\"%d\",\"{}\").replace(\"%b\",\"{}\").replace(\"%s\",\"
{}\").format( *args ), end='' )";
    "class PLOG_NODE():\n"^
    "\tdef __init__(self):\n"^
    "\t\tself.in_edges = set()\n"^
    "\t\tself.out_edges = set()\n"^
    "\t\tself.props = defaultdict( lambda: None )";  (* For == NIL checks. *)
    "class PLOG_EDGE():\n"^
    "\tdef __init__(self, fromnode=None, t=None, tonode=None):\n"^ (* `=None`s
were included only because we support declaration of "edges" locally, and they
have to be initialized to something (mainly for access to props) *)
    "\t\tself.from_node = fromnode\n"^
    "\t\tself.to_node = tonode\n"^
    "\t\tself.etype = t\n"^
    "\t\tself.props = defaultdict( lambda: None )";
    "class PLOG_GRAPH():\n"^
    "\tdef __init__(self):\n"^
    "\t\tself.nodes = {}\n"^
    "\t\tself.edges = {}\n" ]

let epilogue = [ "if __name__ == '__main__':\n"^
    "\tPUSER_main()" ]
```

```
let translate (nodes, graphs, funcs) =
    let addTab s = "\t"^ s
    in

    (* ******************* *)

    let rec string_of_expr = function
        IntLit(i) -> string_of_int i
        | BoolLit(b) -> if b then "True" else "False"
        | StrLit(s) -> "\""^ String.escaped s ^"\""
        | Lst(t, el) -> "["^ String.concat ", " (List.map string_of_expr el)
^"]"
        | Id(s) -> "PUSER_"^ s
        | Unop(o, e) -> (match o with
            Neg -> "(-"^ string_of_expr e ^")"
            | Not -> "(not "^ string_of_expr e ^")")
        | Biop(e1, o, e2) -> "("^ string_of_expr e1 ^" "^ string_of_biop o ^" "^
string_of_expr e2 ^")"
        | Call(s, el) -> (if s = "print" || s = "append" || s = "length" || s =
"remove" then
            "PLOG_" else "PUSER_") ^ s ^"( "^ String.concat ", " (List.map
string_of_expr el) ^" )"
        | GraphAccess(s, ge, locals) ->
            "PUSER_"^ (match ge with
            NodeId(id) -> if List.exists (fun x -> x = id) locals then id
                else s ^".nodes.get( '"^ id ^"', None )"
            | EdgeId(fn, et, tn) -> s ^".edges.get( (PUSER_"^
                (if List.exists (fun x -> x = fn) locals then fn else s
^".nodes.get('"^ fn ^"', None)") ^
                ", "^ (if List.exists (fun x -> x = et) locals then "PUSER_"^ et
else "'"^ et ^"'") ^
                ", PUSER_"^ (if List.exists (fun x -> x = tn) locals then tn
else s ^".nodes.get('"^ tn ^"', None)") ^
                "), None )"
            )
        | Property(e, s) -> string_of_expr e ^".props[ '"^ s ^"' ]"
        | Nil(t) -> "None"
        | Inf -> "float('Inf')"
    in

    let string_of_pred (prop, ival) =
        "'"^ prop ^"' in n.props and n.props['"^ prop ^"'] == "^ string_of_int
ival
    in

    let get_node_forloop nidpatt npatt locals e eisgraph namednode =
        let patt_pids_length = List.length npatt.pids in

        let nidpatt_index =
            let rec get_nidpatt_index idx = function
                [] -> raise (Failure "Not found!")
                | nid :: [] -> if nid = nidpatt then idx else raise (Failure
"Not found!")
                | nid :: etype :: xs -> if nid = nidpatt then idx else
get_nidpatt_index (idx+1) xs
            in
            get_nidpatt_index 1 npatt.pids
        in
```

```
        let get_node_forloop_body =
            if patt_pids_length = 1 then (
                (* If the "first" node in the pattern is a local var, next line
handles that *)
                (if List.exists (fun x -> x = List.nth npatt.pids 0) locals then
                ("n1_nodes = [ n for n in n1_nodes if n == PUSER_"^ List.nth
npatt.pids 0 ^" ]")
                else "") ::

                (if List.length npatt.preds = 0 then ""
                else "n1_nodes = [ n for n in n1_nodes if "^
                    (String.concat " and " (List.map string_of_pred
npatt.preds))
                    ^" ]")
                ::
                "for n1 in n1_nodes:"
                ::
                [ "\tmatching_nodes.add( n1 )" ]
            ) else (
                let rec funcy i =
                    ("n"^ string_of_int i ^"_nodes = [ e.from_node for e in n"^
string_of_int (i-1) ^
                    ".in_edges if e.etype == '"^ (List.nth npatt.pids (i*2 - 3))
^"' ]")
                    ::
                    (if List.exists (fun x -> x = List.nth npatt.pids (i*2 - 2))
locals then
                    ("n"^ string_of_int i ^"_nodes = [ n for n in n"^
string_of_int i ^"_nodes if n == PUSER_"^ List.nth npatt.pids (i*2 - 2) ^" ]")
                    else "")
                    ::
                    (if List.length npatt.preds = 0 then ""
                    else "n"^ string_of_int i ^"_nodes = [ n for n in n"^
string_of_int i ^"_nodes if "^
                        String.concat " and " (List.map string_of_pred
npatt.preds)
                        ^" ]")
                    ::
                    (if i*2 - 1 = patt_pids_length then
                        (("for n"^ string_of_int i ^" in n"^ string_of_int i
^"_nodes:")
                        ::
                        [ "\tmatching_nodes.add( n"^ string_of_int nidpatt_index
^" )" ])
                    else (("for n"^ string_of_int i ^" in n"^ string_of_int i
^"_nodes:")
                        ::
                        (List.map addTab (funcy (i+1)))) )
                in

                (* If the "first" node in the pattern is a local var, next line
handles that *)
                (if List.exists (fun x -> x = List.nth npatt.pids 0) locals then
                    ("n1_nodes = [ n for n in n1_nodes if n == PUSER_"^ List.nth
npatt.pids 0 ^" ]")
                else "")
                ::
                (if List.length npatt.preds = 0 then ""
                else "n1_nodes = [ n for n in n1_nodes if "^
```

```
                    String.concat " and " (List.map string_of_pred npatt.preds)
                    ^" ]")
                ::
                "for n1 in n1_nodes:"
                ::
                (List.map addTab (funcy 2))
            )
        in
        "matching_nodes = set()\n" ::
        ("n1_nodes = "^ string_of_expr e ^
            (if eisgraph then ".nodes.copy().values()" else "")) ::

        (* String.concat "\n\t" get_node_forloop_body :: *)
        get_node_forloop_body @

        ((match namednode with None -> ""
            (* If we're iterating with a named node, get the subset of the above
results which match the type *)
            | Some(name) -> ("matching_nodes = PLOG_GETTNN_"^ name
^"( matching_nodes )\n"))
        (* Now we set ourselves for the for loop block, where we'll actually
iterate over the matching nodes *)
        :: ["for PUSER_"^ nidpatt ^" in matching_nodes:"])
    in

    let string_of_node_decl node_decl =
        let nidpatt = node_decl.nidpatt in
        let npatt = node_decl.npatt in

        let patt_pids_length = List.length npatt.pids in (* NOTE: We assume it's
odd, based on static sem check *)

        (* This func takes a patt (and nid and idx accumulator) and
        returns the 1-based index of the nid from the `right` of the pattern.
        Assumes List.rev was NOT called on the pids *)
        let nidpatt_index =
            let rec get_nidpatt_index idx = function
                [] -> raise (Failure "Not found!")
                | nid :: [] -> if nid = nidpatt then idx else raise (Failure
"Not found!")
                | nid :: etype :: xs -> if nid = nidpatt then idx else
get_nidpatt_index (idx+1) xs
            in
            get_nidpatt_index 1 npatt.pids
        in

        let get_node_body =
            if patt_pids_length = 1 then (
                (if List.length npatt.preds = 0 then ""
                else "\tn1_nodes = [ n for n in n1_nodes if "^
                    (String.concat " and " (List.map string_of_pred
npatt.preds))
                    ^" ]")
                ::
                "for n1 in n1_nodes:"
                ::
                [ "\tmatching_nodes.add( n1 )" ]
            ) else (
                let rec funcy i =
```

```
                    ("\tn"^ string_of_int i ^"_nodes = [ e.from_node for e in
n"^ string_of_int (i-1) ^
                    ".in_edges if e.etype == '"^ (List.nth npatt.pids (i*2 - 3))
^"' ]")
                    ::
                    (if List.length npatt.preds = 0 then ""
                    else "\tn"^ string_of_int i ^"_nodes = [ n for n in n"^
string_of_int i ^"_nodes if "^
                        String.concat " and " (List.map string_of_pred
npatt.preds)
                        ^" ]")
                    ::
                    (if i*2 - 1 = patt_pids_length then
                        (("\tfor n"^ string_of_int i ^" in n"^ string_of_int i
^"_nodes:")
                        ::
                        [ "\t\tmatching_nodes.add( n"^ string_of_int
nidpatt_index ^" )" ])
                    else (("\tfor n"^ string_of_int i ^" in n"^ string_of_int i
^"_nodes:")
                        ::
                        (List.map addTab (funcy (i+1)))) )
                in

                (if List.length npatt.preds = 0 then ""
                else "\tn1_nodes = [ n for n in n1_nodes if "^
                    String.concat " and " (List.map string_of_pred npatt.preds)
                    ^" ]")
                ::
                "for n1 in n1_nodes:"
                ::
                funcy 2
            )
        in
        "def PLOG_GETTNN_"^ node_decl.nname ^"( n1_nodes ):\n"^
        "\tmatching_nodes = set()\n"^
        String.concat "\n\t" get_node_body ^
        "\n\treturn list( matching_nodes )"
    in

    let string_of_graph_def gname gbody locals =
        let gnodes = "PUSER_"^ gname ^".nodes" in
        let gedges = "PUSER_"^ gname ^".edges" in

        let string_of_nodedec = function
            NodeId(id) -> if List.exists (fun x -> x = id) locals then
                    (("if PUSER_"^ id ^" not in "^ gnodes
^".copy().values():") ::
                    ("\tfint = 1") ::
                    ("\twhile '"^ id ^"'+ str(fint) in "^ gnodes ^":") ::
                    ("\t\tfint += 1") ::
                    ("\tif PUSER_"^ id ^" == None:") ::
                    ("\t\tPUSER_"^ id ^" = PLOG_NODE()") ::
                    ["\t"^ gnodes ^"[ '"^ id ^"'+ str(fint) ] = PUSER_"^ id])
                else (("if '"^ id ^"' not in "^ gnodes ^":") ::
                    ["\t"^ gnodes ^"[ '"^ id ^"' ] = PLOG_NODE()"])
        in

        let string_of_nodeprop sil = function
```

```
            (* NOTE: This function assumes that the node exists in the graph. *)
            NodeId(id) -> List.map (fun (s,i) ->
                (if List.exists (fun x -> x = id) locals then ("PUSER_"^id)
                 else (gnodes ^"[ '"^ id ^"' ]")) ^".props[ '" ^ s ^"' ] = "^
string_of_int i) sil
        in

        let string_of_graph_stmt = function
            GraphSet(gel, sil) -> (match gel with
                [EdgeId(fromn,et,ton)] ->
                    let flocal = List.exists (fun x -> x = fromn) locals in
                    let tlocal = List.exists (fun x -> x = ton) locals in
                    let gfromn = if flocal then "PUSER_"^ fromn else gnodes
^"['"^ fromn ^"']" in
                    let get = if List.exists (fun x -> x = et) locals then
"PUSER_"^ et
                        else "'"^ et ^"'" in
                    let gton = if tlocal then "PUSER_"^ ton else gnodes ^"['"^
ton ^"']" in
                    let etriple = gfromn ^", "^ get ^", "^ gton in
                    (* Check if both nodes exist (before checking if edge
exists). Create if not existing. *)
                    (* NOTE: We have an issue if the nodes are local to the
GraphDef but don't exist in the graph.
                        How we resolve this now is: create a new node in the
graph with the "name" localname+nextfreeint
                            We also effectively "remember" anything about that
node (its props, in/out edges). *)
                    (if flocal then
                        ("if "^ gfromn ^" not in "^ gnodes ^".copy().values():")
::
                        ("\tfint = 1") ::
                        ("\twhile '"^ fromn ^"'+ str(fint) in "^ gnodes ^":") ::
                        ("\t\tfint += 1") ::
                        ("\tif "^ gfromn ^" == None:") ::
                        ("\t\t"^ gfromn ^" = PLOG_NODE()") ::
                        ["\t"^ gnodes ^"[ '"^ fromn ^"'+ str(fint) ] = "^
gfromn]
                    else
                        ("if '"^ fromn ^"' not in "^ gnodes ^":") ::
                        ["\t"^ gfromn ^" = PLOG_NODE()"]
                    ) @
                    (if tlocal then
                        ("if "^ gton ^" not in "^ gnodes
^".copy().values():") ::
                        ("\ttint = 1") ::
                        ("\twhile '"^ ton ^"'+ str(tint) in "^ gnodes ^":") ::
                        ("\t\ttint += 1") ::
                        ("\tif "^ gton ^" == None:") ::
                        ("\t\t"^ gton ^" = PLOG_NODE()") ::
                        ["\t"^ gnodes ^"[ '"^ ton ^"'+ str(tint) ] = "^ gton]
                    else
                        ("if '"^ ton ^"' not in "^ gnodes ^":") ::
                        ["\t"^ gton ^" = PLOG_NODE()"]
                    ) @
                    (* Create the edge based on the two nodes *)
                    ("if ("^ etriple ^") not in "^ gedges ^":") ::
                    ("\t"^ gedges ^"[ ("^ etriple ^") ] = PLOG_EDGE( "^ etriple
^" )") ::
```

```
                    (* Update the nodes: Add this new edge to each node
accordingly *)
                    ("\t"^ gfromn ^".out_edges.add( "^ gedges ^"[ ("^ etriple
^") ] )") ::
                    ("\t"^ gton ^".in_edges.add( "^ gedges ^"[ ("^ etriple ^") ]
)")
                    ::
                    (* Now set any properties for the edge *)
                    List.map (fun (s,i) -> gedges ^"[ ("^ etriple
^") ].props[ '"^ s ^"' ] = "^ string_of_int i) sil
                | ns -> List.concat (List.map string_of_nodedec gel)
                    @
                    List.concat (List.map (string_of_nodeprop sil) ns)
            )
            | GraphDel(gel) -> (match gel with
                [EdgeId(fromn,et,ton)] ->
                    let flocal = List.exists (fun x -> x = fromn) locals in
                    let tlocal = List.exists (fun x -> x = ton) locals in
                    let gfromn = if flocal then "PUSER_"^ fromn else gnodes
^"['"^ fromn ^"']" in
                    let get = if List.exists (fun x -> x = et) locals then
"PUSER_"^ et
                        else "'"^ et ^"'" in
                    let gton = if tlocal then "PUSER_"^ ton else gnodes ^"['"^
ton ^"']" in
                    let etriple = gfromn ^", "^ get ^", "^ gton in

                    (* We need to delete all references of the edge (which might
exist in nodes' in/out_edges lists). *)
                    (* We assume an invariant: that if a node doesn't exist in a
graph, then no edge referencing that node exists in the graph;
                        that is, it's necessary that both nodes (the "from" and
"to" nodes) exist for an edge referencing both nodes to exist. *)
                    ("if "^
                        (if flocal then "PUSER_"^ fromn ^" in "^ gnodes
^".copy().values()"
                         else "'"^ fromn ^"' in "^ gnodes)
                     ^" and "^
                        (if tlocal then "PUSER_"^ ton ^" in "^ gnodes
^".copy().values():"
                         else "'"^ ton ^"' in "^ gnodes ^":")) ::
                    ("\tfor node in "^ gnodes ^".copy().values():") ::
                    ("\t\tnode.in_edges.discard( "^ gedges ^".get(("^ etriple
^"), None) )") ::
                    ("\t\tnode.out_edges.discard( "^ gedges ^".get(("^ etriple
^"), None) )") ::
                    ["\t"^ gedges ^".pop( ("^ etriple ^"), None )"]
                | ns -> let nid_to_del = function
                    NodeId(id) ->
                        if List.exists (fun x -> x = id) locals then
                            ("for nkey, val in "^ gnodes ^".copy().items():") ::
                            ("\tif val == PUSER_"^ id ^":") ::
                            (* Delete all edges (NOTE: only in this graph)
referencing this node. *)
                            ("\t\tfor ekey, edge in "^ gedges
^".copy().items():") ::
                            ("\t\t\tif edge.from_node == val or edge.to_node ==
val:") ::
```

```
                            ("\t\t\t\tfor node in "^ gnodes
^".copy().values():") ::
                            ("\t\t\t\t\tnode.in_edges.discard( edge )") ::
                            ("\t\t\t\t\tnode.out_edges.discard( edge )") ::
                            ("\t\t\t\t"^ gedges ^".pop( ekey, None )") ::

                            ("\t\tdel "^ gnodes ^"[ nkey ]") ::
                            ["\t\tbreak"] (* NOTE: We assume there's at most 1
such node in a graph. *)
                        else
                            ("if '"^ id ^"' in "^ gnodes ^":") ::
                            (* Delete all edges (NOTE: only in this graph)
referencing this node. *)
                            ("\tfor ekey, edge in "^ gedges ^".copy().items():")
::
                            ("\t\tif edge.from_node == "^ gnodes ^"[ '"^ id
^"' ] or edge.to_node == "^ gnodes ^"[ '"^ id ^"' ]:") ::

                            ("\t\t\tfor node in "^ gnodes
^".copy().values():") ::
                            ("\t\t\t\tnode.in_edges.discard( edge )") ::
                            ("\t\t\t\tnode.out_edges.discard( edge )") ::
                            ("\t\t\t"^ gedges ^".pop( ekey, None )") ::

                            ["\tdel "^ gnodes ^"[ '"^ id ^"' ]"]
                    in
                    List.concat (List.map nid_to_del ns)
            )
        in

        List.concat (List.map string_of_graph_stmt gbody)
    in

    let string_of_graph_decl gdecl =
        ("PUSER_"^ gdecl.gname ^" = PLOG_GRAPH()")
        :: string_of_graph_def gdecl.gname gdecl.gbody []
    in

    let rec strings_of_local_decs = function
        [] -> []
        (* It's not necessary to initialize a list if it's explicitly
initialized by the user,
           so we can (later) avoid the redundant " = []" in the generated code.
*)
        | (TList(t),s) :: xs -> ("PUSER_"^ s ^" = []") :: strings_of_local_decs
xs
        | (TNode,s) :: xs -> ("PUSER_"^ s ^" = PLOG_NODE()") ::
strings_of_local_decs xs
        | (TEdge,s) :: xs -> ("PUSER_"^ s ^" = PLOG_EDGE()") ::
strings_of_local_decs xs
        | _ :: xs -> strings_of_local_decs xs
    in

    let rec string_of_stmt = function
        Assign(e1, e2) -> [string_of_expr e1 ^" = "^ string_of_expr e2]
        (* NOTE: Check for "empty lines", which make a list (e.g. sl/bsl) appear
as non-empty. *)
        | Block(tnl, sl) -> let bsl = strings_of_local_decs tnl @ List.concat
(List.map string_of_stmt sl) in
```

```
            List.map addTab (if List.length bsl = 0 then ["pass"] else bsl)
        | Expr(e) -> string_of_expr e :: []
        | Return(e) -> ("return "^ string_of_expr e) :: []
        | If(e, s, Block([],[])) -> ("if "^ string_of_expr e ^":") ::
string_of_stmt s
        | If(e, s1, s2) -> ( ("if "^ string_of_expr e ^":") :: string_of_stmt
s1) @
            ("else:" :: string_of_stmt s2)
        | For(t, s, {pids=[];preds=[]}, e, eisgraph, st, locals) -> (match t
with
            TEdge ->
                ("for PUSER_"^ s ^" in "^ string_of_expr e ^
                (if eisgraph then ".edges.copy().values():" else ":")) ::
string_of_stmt st
            | TNode ->
                ("for PUSER_"^ s ^" in "^ string_of_expr e ^
                if eisgraph then ".nodes.copy().values():" else ":") ::
string_of_stmt st
            | TNNode(tnns) ->
                ("matching_nodes = PLOG_GETTNN_"^ tnns ^"( "^ string_of_expr e ^
                    (if eisgraph then ".nodes.copy().values()" else "")
^" )") ::
                ("for PUSER_"^ s ^" in matching_nodes:") :: string_of_stmt st
            | _ -> ("for PUSER_"^ s ^" in "^ string_of_expr e ^":") ::
string_of_stmt st )
        | For(t, s, p, e, eisgraph, st, locals) -> (match t with
            TEdge -> (* NOTE: We assume static semantic analysis checked there
are exactly 3 identifiers in p.pids *)
                "matching_edges = set()" ::
                ("n1_nodes = "^ string_of_expr e ^
                    (if eisgraph then ".nodes.copy().values()" else "")) ::
                (if List.exists (fun x -> x = List.nth p.pids 0) locals then
                    ("n1_nodes = [ n for n in n1_nodes if n == PUSER_"^ List.nth
p.pids 0 ^" ]")
                else "") ::
                "for n1 in n1_nodes:" ::
                "\tn2_nodes = [ e.from_node for e in n1.in_edges ]" ::
                (if List.exists (fun x -> x = List.nth p.pids 2) locals then
                    ("\tn2_nodes = [ n for n in n2_nodes if n == PUSER_"^
List.nth p.pids 2 ^" ]")
                else "") ::
                "\tedges = [ e for n2 in n2_nodes for e in n2.out_edges ]" ::
                (if List.length p.preds = 0 then "" else
                    "\tedges = [ n for n in edges if "^ String.concat " and "
(List.map string_of_pred p.preds) ^" ]") ::
                "\tfor edge in edges:" ::
                "\t\tmatching_edges.add( edge )" ::
                ("for PUSER_"^ s ^" in matching_edges:") :: string_of_stmt st
            | TNode -> get_node_forloop s p locals e eisgraph None @
string_of_stmt st
            | TNNode(tnns) -> get_node_forloop s p locals e eisgraph (Some tnns)
@ string_of_stmt st )
        | While(e, s) -> ("while "^ string_of_expr e ^":") :: string_of_stmt s
        | GraphDef(s, sl, locals) -> string_of_graph_def s sl locals
    in

    let string_of_func fdecl = "def PUSER_"^ fdecl.fname ^"( "^ String.concat ",
" (List.map (fun (_,s) -> "PUSER_"^ s) fdecl.formals) ^" ):\n"^
        let fstmt_list = List.concat (List.map string_of_stmt fdecl.fbody) in
```

```
        (* flocals are already included in the func's fbody (as a Block). Blocks
automatically add tabs. *)
        (*(String.concat "\n\t" (strings_of_local_decs fdecl.flocals))
^"\n\t"^*)
        (* NOTE: "blank lines" from the fstmt_list will be output. *)
        if List.length fstmt_list = 0 then "\tpass" else String.concat "\n"
fstmt_list
    in

    (String.concat "\n" prelude) ^"\n"^
    (String.concat "\n" (List.map string_of_node_decl nodes)) ^"\n"^
    (String.concat "\n" (List.concat (List.map string_of_graph_decl graphs)))
^"\n"^
    (String.concat "\n" (List.map string_of_func funcs)) ^"\n"^
    (String.concat "\n" epilogue) ^"\n"
```

9.7 Compiler – plog.ml

```
type action = Ast | Interpret | Bytecode | Compile

let _ =
  let action = if Array.length Sys.argv > 1
    then List.assoc Sys.argv.(1) [ ("-a", Ast);
.....................................................................
.....................................................................
.....................................................      ("-c", Compile) ]
    else Compile
  in

  let lexbuf = Lexing.from_channel stdin in
  let program = (try Parser.program Scanner.token lexbuf
                 with Parsing.Parse_error -> raise (Failure ("Program did not
parse successfully.")))
  in

  let sprogram = Semana.translate program in

  match action with
    Ast -> let listing = Ast.string_of_program program
           in print_string listing
  | Compile -> print_string (Compile.translate sprogram)
  | _ -> print_string "INVALID OPTION\n"
```

9.8 Makefile

```
TARFILES = Makefile ast.ml sast.ml scanner.mll parser.mly semana.ml compile.ml
plog.ml

OBJS = ast.cmo sast.cmo parser.cmo scanner.cmo semana.cmo compile.cmo plog.cmo

plog : $(OBJS)
.....................................................ocamlc -o plog $(OBJS)

scanner.ml : scanner.mll
.....................................................ocamllex scanner.mll

parser.ml parser.mli : parser.mly
.....................................................ocamlyacc parser.mly
```

```
%.cmo : %.ml
.....................................................................ocamlc -c $<

%.cmi : %.mli
.....................................................................ocamlc -c $<

plog.tar.gz : $(TARFILES)
.........................cd .. && tar zcf lang1/plog.tar.gz $(TARFILES:%=plog/%)

.PHONY : clean
clean :
.........................rm -f plog parser.ml parser.mli scanner.ml *.cmo *.cmi

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
sast.cmo:
sast.cmx:
plog.cmo: scanner.cmo parser.cmi ast.cmo
plog.cmx: scanner.cmx parser.cmx ast.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
semana.cmo: ast.cmo sast.cmo
semana.cmx: ast.cmx sast.cmx
compile.cmo: sast.cmo
compile.cmx: sast.cmx
```

9.9 Tests

The `testall.sh` script and the Tutorial (Section 2) example program are both included here. For all of the test cases used when testing the PLOG compiler, see the PLOG source code archive.

9.9.1 testall.sh

```
#!/bin/sh

LANG1="./plog"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.plg files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}
```

```
SignalError() {
    if [ $error -eq 0 ] ; then
.....................................................................echo "FAILED"
.....................................................................error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
.....................................................SignalError "$1 differs"
.....................................echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
.................................................SignalError "$1 failed on $*"
.....................................................................return 1
    }
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                              s/.plg//'`
    reffile=`echo $1 | sed 's/.plg$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

#   NOTE: We don't have an interpreter (right now), so don't test for it.
#    generatedfiles="$generatedfiles ${basename}.i.out" &&
#    Run "$LANG1" "-i" "<" $1 ">" ${basename}.i.out &&
#    Compare ${basename}.i.out ${reffile}.out ${basename}.i.diff

    generatedfiles="$generatedfiles ${basename}.plg.out" &&
    Run "$LANG1" "-c" "<" $1 "|" "python3" ">" ${basename}.plg.out &&
    Compare ${basename}.plg.out ${reffile}.out ${basename}.plg.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
.....................................................if [ $keep -eq 0 ] ; then
.....................................................    rm -f $generatedfiles
.....................................................................fi
```

```
.....................................................................echo "OK"
.....................................................echo "###### SUCCESS" 1>&2
    else
.....................................................echo "###### FAILED" 1>&2
.....................................................globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
.....................................................k) # Keep intermediate files
.....................................................................    keep=1
.....................................................................    ;;
.....................................................................h) # Help
.....................................................................    Usage
.....................................................................    ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/fail-*.plg tests/test-*.plg"
fi

for file in $files
do
    case $file in
.....................................................................*test-*)
.................................................    Check $file 2>> $globallog
.....................................................................    ;;
.....................................................................*fail-*)
.............................................    CheckFail $file 2>> $globallog
.....................................................................    ;;
.....................................................................*)
.............................................    echo "unknown file type $file"
.....................................................................    globalerror=1
.....................................................................    ;;
    esac
done

exit $globalerror
```

9.9.2 tests/test-TUTORIAL.plg

```
node neighbor = x in x where distFromSource = 2;

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;
    int minDist, ndist;

    closest = NIL(node);
    minDist = INF;

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

    return closest;
}

func computeMinDistsFromSource( graph G, node source )
{
    node list unvisited;
    node closest;
    int neighborDist, newDist;

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

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

        for node neighbor in closest distance-> neighbor in G
        {
            neighborDist = G:( closest distance-> neighbor ).value;
            /* A property named "value" */
            newDist = closest.distFromSource + neighborDist;
            if newDist < 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", n.nodeNum, n.distFromSource );

        if n == Neighborhood:(You) {
```

```
            Neighborhood { Me thanks-> n; } }
    }

    for node:neighbor n in Neighborhood {
        print( "%d is a \"true neighbor\".\n", n.nodeNum );
    }
}
```

9.9.2.1 tests/test-TUTORIAL.out

```
1 is 0 away.
2 is 2 away.
3 is 5 away.
2 is a “true neighbor”.
```