The GAL Reference Manual

Athar Abdul-Quader
ama2115@columbia.edu

Shepard Saltzman
sms2195@columbia.edu

Albert Winters
ajw2124@columbia.edu

Oren B. Yeshua
oby1@columbia.edu

1 Lexical Conventions

A GAL program consists of one or more translation units that are stored in files. Each file is written using the ASCII character set and must terminate with '.gal'.

1.1 Comments

GAL recognizes single-line comments that begin with '#' and terminate at the end of the line. Multi-line comments can be achieved by commenting each line individually.

1.2 Whitespace

Comments and the ASCII space are ignored by the GAL compiler. However, characters normally ignored as whitespace play a significant role in the GAL language. The horizontal tab is not considered whitespace because it is used to indicate scope, and the line terminator is used to indicate the end of a statement.

In GAL, scope is indicated by the tab ('\ t') character. Every line in a GAL program must begin with zero or more tabs. Before a GAL program is compiled, it undergoes a scope preprocessing step. As the lines of the program are read by the preprocessor from first to last, a running count is kept of the *indentation level*. The indentation level begins at 0. If the number of tabs at the start of the next line is greater than the current indentation level, a new block is created and the indentation level is incremented by one. If the number of tabs at the start of the next line is less than the current indentation level, the current block is closed and the indentation level is decrimented by one. For a program to successfully compile, the indentation level must be zero when EOF is reached.

1.3 Tokens

There are five classes of tokens: identifiers, keywords, constants (immediates), operators, and separators.

1.3.1 Identifiers

Identifiers begin with a letter followed by any sequence of letters, digits and the underscore character. Two letters are considered the same if their ASCII characters are equal. Two identifiers are considered the same if every character in both identifiers match.

1.3.2 Keywords

The following identifiers are reserved as keywords and cannot be used as anything else:

```
set    vertex  edge   graph
queue  string  bool   num
in     foreach while  and
if      else   path   or
```

1.3.3 Constants

Constants (immediates) provide the GAL programmer with a convenient way to initialize each of the built in types.

Numeric constants A numeric constant consists of an integer part followed by an optional fractional part and optional exponent. The integer part is a string of one or more digits ($0 \ldots 9$). The integer part may be followed by the fractional part, or the exponent part or both. The fractional part is a period ('.') which is followed by one or more digits. The exponent part is an 'e' followed by an optional sign ('$\pm$') followed by one or more digits.

String constants A string constant is a sequence of characters surrounded by double quotes. The double-quote marks are not considered part of the string and are omitted when processing the string. Double-quote marks may be added to a string with \" and similarly, the backslash can be escaped with itself like so \\.

1.3.4 Operators

Arithmetic/Set Operators There are five arithmetic operators: +, -, *, /, and %. The '+' operator denotes addition while the '-' operator denotes subraction. The '*' operator denotes multiplication, and the '/' operator denotes division. The '%' operator denotes the remainder of the division of the first operand by the second operand. The operands must have arithmetic type. The result of using an arithemtic operator on two `num` types is the usual arithmetic operation.

The arithmetic operators perform double duty as set operators as well. The '+' operator indicates the set union operation while the '*' indicates intersection. The '-' operator indicates set difference while the '/' operator denotes symmetric difference. Similarly, '+=', '*=', '-=','/=' are defined in the usual way for sets as well. Furthermore, the `in` operator can be used to determine set membership.

Boolean Operators There boolean operators are as follows: and, or, >, <, <=, >= =, ! . The 'and' operator returns `true` when both the left and right operands evaluate to `true`, `false` otherwise. The 'or' operator returns `true` when either or both the left or right operand evaluate to `true`, `false` otherwise. The '=' operator returns `true` if left operand equals the right operand; otherwise, it returns `false`. The '!' operator is unary and returns `true` if its operand evaluates to `false`, `false` otherwise.

As in most programming languages, the `and` and `or` operators use a shortcut evaluation model. If the first operand of an `or` is true, the second is not evaluated. Similarly, if the first operand of an `and` is false, the second is not evaluated.

Assignment Operators The six assignment operators are as follows: ← , +=, -=, *=, /=, and %=. The '← ' operator assigns the value of the right operand to the left operand. '+=' and '-=' adds or subtracts the value of the right operand to the value of the left operand. '*=' and '/=' multiplies or divides the value of the right operand by the value of the left operand. '%=' places in the left operand the remainder of the left operand divided by the right operand.

Access Operator The '.' operator is used to access fields in the left operand. If a field that does not currently exist is accesed, it is created on the fly and inherits its type from whatever is assigned to it. The default type is `num`. The '[]' operator is used to access an element by index within a `set`, `path`, or `queue`.

Range Operator The '..' operator specifies a range of values and provides a convenient shorthand for specifying sets. Given [num1]..[num2], where [num1] and [num2] are integers, GAL will interpret it as a comma separated list of all the integers between [num1] and [num2] inclusive.

Precedence The operator precedence rules follow those of the C language. Since this may result in unexpected order of operations when operating on sets, it is reccomended that parentheses be used to explicitly specify precedence in expressions involving sets (although it is by no means required).

1.4 Separators

The following characters are used as separators: ',' '.' '[' ']'.

1.5 Scope

The scope of an identifier begins after the declaration of the identifier and terminates at the end of the block in which the identifier is found. Blocks are determined by indentation, as opposed to C-style '{' and '}' characters.

2 Types

2.1 Nums, Bools, & Strings

There is only one numeric type in GAL. The `num` is internally represented as a double value. The `bool` type can take on either of the two boolean values `true` and `false`. A `string` is sequence of ASCII characters.

2.2 Sets

A `set` is an undordered collection of objects of any type. Sets support all basic group manipulation functions, such as indexing, concatenation and traversal. Sets support the range operator.

2.3 Vertices

A `vertex` is a very simple type most often used as an element of a graph. By default, a vertex contains just one field - `index` of type `num` (additional fields can be added using the '.' operator).

2.4 Edges

An `edge` is also used to represent an element of a graph. By default, an edge is an ordered pair of two indices, representing a directed edge from the vertex of one index to the vertex of the other. An edge contains three fields by default, each of type `num`. They are: `src`, `dest`, `weight`.

2.5 Graphs

Graphs are intended to concisely represent mathematical graphs. A `graph` simply consists of two sets V and E representing the vertices and edges of the graph respectively. These sets can be accessed with the '.' operator.

2.6 Paths

A `path` is an ordered list of `num`s, (presumably, but not necessarily, representing a traversal of the graph following the vertices of corresponding index).

2.7 Queues

A `queue` is implemented as a priority queue. It maintains a list of (element, priority) pairs. Queues support the push, pop, enqueue, and change_priority operations. Push is given an element and a priority (of type `num`) to be added to the queue. Pop returns the element in the queue with the greatest priority. Change_priority changes the priority of an element already in the queue. If push is used without specifying a priority, the queue automatically assigns the element a priority higher than anything currently in the queue. This allows the queue to be used as a (LIFO)

stack. Similarly, enqueue adds an element to the queue with a priority lower than any elment in the queue, allowing the queue to be used in FIFO fashion.

3 Control Structures

3.0.1 foreach

Foreach is a keyword used to generate loops. With "in", it can be used to run a block of code repeatedly, once on each element in a `set`, `path`, or `queue`. The order of iteration for a set is undefined, while that of path is in order from left to right, and that of queue is in order of priority from greatest to least. For example, a basic block of code to find the sum of weights on edges in a graph could be written as:

```
total = 0
foreach (edge in G.E)
   edge.weight += total
```

3.0.2 while

While is a keyword used to generate loops. While is a header with a condition that encloses a block of code. When that block is first reached and after each execution of that block, while's condition is checked. If the condition is true, the block is (re)executed, otherwise it is skipped.

3.0.3 if

If is a keyword used to create conditionals. If statements always include a condition and a block of code, and are optionally followed by an else block. When the if-block is reached, its condition is checked. If that condition is true, the block of code is executed, otherwise it is skipped.

3.0.4 else

Else is a keyword used to create conditionals. An else is a header to a block of code that immediately follows an if-block. If the if-block is executed, the else-block is skipped. Otherwise, the else-block is executed. An else may also be followed directly by an if.

4 Syntax

4.1 Grammar

$$
\begin{aligned}
Program &\rightarrow Var\_decl \mid Func\_decl \\
Var\_decl &\rightarrow Type\ (< id > \mid Assignment) \\
Func\_decl &\rightarrow Func\_def\ '\{'\ Block\ '\}' \\
Func\_def &\rightarrow < id >\ '('\ Parm\_list\ ')'\ < stmt\_term > \\
Parm\_list &\rightarrow < id >\ Id\_list \mid \epsilon \\
Id\_list &\rightarrow ','\ < id >\ Id\_list \mid \epsilon \\
Block &\rightarrow Statement\ Block \mid \epsilon \\
Statement &\rightarrow (\ Expression \mid Func\_call\_stmt \mid Return\_stmt \\
&\quad \mid If\_stmt \mid For\_stmt \mid Foreach\_stmt \\
&\quad \mid While\_stmt \mid Var\_decl \mid '\{'\ Block\ '\}'\ )\ < stmt\_term > \\
If\_stmt &\rightarrow '\text{if}'\ '('\ Expression\ ')'\ '\{'\ Block\ '\}'\ (\ Else\_stmt \mid \epsilon\ ) \\
Else\_stmt &\rightarrow '\text{else}'\ (\ Block \mid If\_stmt\ ) \\
Foreach\_stmt &\rightarrow '\text{foreach}'\ '('\ < id >\ '\text{in}'\ LValue\ ')'\ '\{'\ Block\ '\}' \\
Func\_call\_stmt &\rightarrow < id >\ '('\ (\ Expression\ Exprn\_list \mid \epsilon\ )\ ')' \\
Return\_stmt &\rightarrow \text{return}\ Expression \\
While\_stmt &\rightarrow '\text{while}'\ '('\ Expression\ ')'\ '\{'\ Block\ '\}' \\
Expression &\rightarrow LValue \mid Assignment \mid '!'\ Expression \mid\ Expression\ Operator\ Expression \\
&\quad \mid Immediate \mid '('\ Expression\ ')' \\
Exprn\_list &\rightarrow ','\ Expression\ Exprn\_list \mid \epsilon \\
LValue &\rightarrow LValue\ '.'\ LValue \mid\ < id >\ '['Expression']' \mid\ < id > \\
Assignment &\rightarrow LValue\ AssignOp\ Expression \\
Operator &\rightarrow '+' \mid '-' \mid '*' \mid '/' \mid '\%' \\
&\quad \mid '>' \mid '<' \mid '>=' \mid '<=' \\
&\quad \mid '=' \mid '!=' \\
AssignOp &\rightarrow '<-' \mid '+=' \mid '-=' \mid '*=' \mid '/=' \mid '\%=' \\
Immediate &\rightarrow Set\_const \mid Path\_const \mid Edge\_const \\
&\quad \mid < numeric\ constant > \mid < string\ constant > \mid < boolean\ constant > \\
Type &\rightarrow \text{graph} \mid \text{set} \mid \text{vertex} \mid \text{edge} \mid \text{queue} \\
&\quad \mid \text{path} \mid \text{num} \mid \text{string} \mid \text{bool} \\
Edge\_const &\rightarrow\ '('\ < numeric\ constant >\ ','\ < numeric\ constant >\ ')' \\
Set\_const &\rightarrow '\{'\ (\ < numeric\ constant >\ Num\_list \mid Range\_const \parallel \epsilon\ )'\}' \\
Path\_const &\rightarrow '('\ (\ < numeric\ constant >\ Num\_list \mid \epsilon\ )')' \\
Num\_list &\rightarrow ','\ < numeric\ constant >\ Num\_list \mid \epsilon \\
Range\_const &\rightarrow < numeric\ constant >\ '..'\ < numeric\ constant >
\end{aligned}
$$