GCODE: A Revised Standard for a Graph Representation for Functional Programs

Mike Joy

Department of Computer Science,
University of Warwick,
Coventry,
CV4 7AL.

Email: M.S.Joy@dcs.warwick.ac.uk

Tom Axford

School of Computer Science,
The University of Birmingham,
PO Box 363,
Birmingham,
B15 2TT.

Email: T.H.Axford@cs.bham.ac.uk

June 1990

© Copyright 1990 M.S. Joy and T.H. Axford. All Rights Reserved.

ABSTRACT

The data structures used in the authors' functional language graph reduction implementations are described, together with a standard notation for representing the graphs in a textual format. The graphs employed are compatible with the intermediate-level functional language FLIC and with the functional languages in use at Birmingham and Warwick. The textual format is designed to be transmittable easily across networks.

1. Introduction

A common standard for the graphical representation of functional programs was developed at Warwick and Birmingham in 1987 to facilitate cooperation in functional language research and the interchange of functional programs [4] .

At Birmingham, the functional language used for teaching purposes [1] has similarities with the languages SASL [6] and Pifl [3] used at Warwick. It was found possible to represent all of these in a common graphical form and to share programs using a common printable representation of these graphs. In the light of the experience gained, however, it has become apparent that some changes to the graphical representation are desirable.

This paper contains a complete description of our revised graphical representation for functional programs, which is called GCODE.

1.1. Reasons for Revising the Representation

Originally, we used the standard graphical representation [4] in a number of distinct ways. Firstly, our programs for graph reduction, etc. (which were all written in the programming language, "C") all used exactly the same data structures for representing the functional programs that they were processing. The graphical

representation of a functional program could also be printed in an ASCII text form, which was called GCODE. This form could be easily transmitted from one machine to another using email or other data transfer facilities. It was also in a format that was reasonably easily readable by humans and so could be used to write programs directly in their graphical form for test purposes and also for diagnostic output to monitor the progress of graph reduction and other processing operations.

The chosen graphical representation fulfilled all these functions, but some problems arose from trying to get it to do too much. In particular, the operators appropriate to the different functional languages were not identical, so we defined the standard graphical representation to include the union of these operator sets. Each implementation then handled only a subset of the defined operators. Thus programs could be transferred from one system to another only if the operators used in one system were translated into the operators used in the other system. This was done; it is not difficult, although it introduces some inefficiency.

New directions in the continuing research at both Warwick and Birmingham have slightly different needs again, with the unattractive prospect of the proliferation of yet more subsets of the standard representation.

This problem has arisen largely from insisting on the use of common internal data declarations in all our software. In practice, however, this degree of compatibility has been of very limited value to us. We have not often exchanged software at the individual procedure level (although that was done in the very early stages of the work to get things off the ground more quickly). It has usually been much more convenient to share complete programs which accept GCODE input or produce GCODE output, and which achieve compatibility in this way.

So, our revised standard graphical representation is intended purely for the purposes of external data interchange (the 'data' is, of course, a functional program). It is not intended to be used as an internal representation (although it may often be convenient to use a very similar internal graphical representation, and that will have the added advantage of making the translation to or from GCODE particularly easy).

The revised standard is also designed primarily to be efficiently computer readable rather than human readable (although easy human readability has been a secondary aim). Human generated test input and human readable diagnostic output should ideally use a more suitable syntax or a more pictorial representation of the graph, although GCODE could be used for these purposes at a pinch.

1.2. The Revised Standard Graphical Representation

The SASL interpreter developed at Warwick uses FLIC [5] as an intermediate code. FLIC has been defined as a standard intermediate code for functional languages and is used in the implementation of several other functional languages at various places throughout the world. It is the most widely used intermediate code for functional languages and hence is an obvious choice on which to base GCODE. The graphical representation on which GCODE is based may be thought of essentially as an internal representation of FLIC.

The basic data types and structures follow FLIC. It is not possible for users to extend this set of basic types: they must represent their own data types and structures purely in terms of those provided. They can, however, annotate objects so that additional information is not lost (the annotation must not affect the meaning, but it can provide guidance as to efficient implementation and other pragmatic information).

All the FLIC predefined operators are supported by GCODE. Turner combinators are also included. Thus GCODE is able to represent FLIC programs both before and after translation to combinator code.

2. The Graph

The abstract structure of a functional program in GCODE is a connected, directed, and possibly cyclic, graph. It is necessary to understand this underlying structure to understand GCODE. The semantics of this graphical representation is based on the semantics of FLIC (see [5] for a definition of FLIC).

There are nine different node types, described below.

2.1. Node Types

2.1.1. Integer

Nodes of type *integer* represent known constants of the usual type integer. The value of the constant is stored in the node, which has out-degree 0. No limits are specified for maximum integer size, although a particular machine implementation may well have such limits. This will be addressed in a future revision of this document.

2.1.2. Real

Nodes of type *real* represent known constants of the usual type real. As for integers, the value of the constant is stored in the node, which has out-degree 0. No limits are currently specified for accuracy of real numbers.

2.1.3. Operator

Nodes of type *operator* represent basic operators which are predefined as part of the language. A code for the operator is stored in the node, which has out-degree 0. Provision is made for families of operators (such as `PACK` in FLIC) and up to 2 integer qualifiers can be stored in the node. See the appendix below for a description of the standard operators.

2.1.4. Apply

Nodes of type *apply* represent function application. These nodes have out-degree 2, the left pointer is to the function and the right pointer is to the argument. All functions are curried.

2.1.5. Lambda

This type of node represents lambda abstraction and has an out-degree of 2. The left pointer is to the bound variable and the right pointer is to the body of the lambda abstraction (which may be of any type).

2.1.6. Variable

Nodes of type *variable* represent the bound variables of lambda abstraction, but can also be used to represent free variables (if the programming language which is being represented allows free variables). They have an out-degree of 0.

2.1.7. Sum-Product

These nodes represent tagged Cartesian products and have variable out-degree. The first element is the size of the product (the out-degree), the second is the "tag" of the sum, and the rest are pointers to the consecutive elements of the product (which may be of any type). See [5] for further details of the semantics of such nodes.

2.1.8. Undefined

Nodes of this type represent $\perp$ (bottom). They have out-degree 0.

2.1.9. Recursive Reference

A node of type *recursive reference* is used whenever a pointer introduces a cycle into the graph (e.g. in recursive definitions). Semantically, the *recursive reference* node (with out-degree 1) simply denotes the node to which it points. Its presence is solely a label that the pointer there is different (we call it a *weak* pointer). Graph traversal and memory management algorithms that would not work on cyclic graphs can then be implemented simply by ignoring weak pointers (see next subsection).

2.2. Graph Structure

It has been shown [2] that a simple reference counting scheme for cyclic graphs of functional programs is practicable. The graph structure supports this scheme of reference counting (although it does not require it: mark-scan garbage collection schemes could be used if preferred, and the reference counts ignored).

If the reference counting scheme is to be used, the graph must satisfy certain requirements, the main one being:

(i) If weak pointers (i.e. pointers from *recursive reference* nodes) are ignored, the graph is acyclic and connected.

A further condition is required to ensure that graph reduction operations do not generate graphs which break this rule:

(ii) There must be exactly one point of entry to any cycle, which will be the node pointed to by one or more weak pointers. That is, there must be only one node in the cycle pointed to by weak pointers and that node must also be the only node in the cycle which is pointed to by any nodes outside the cycle.

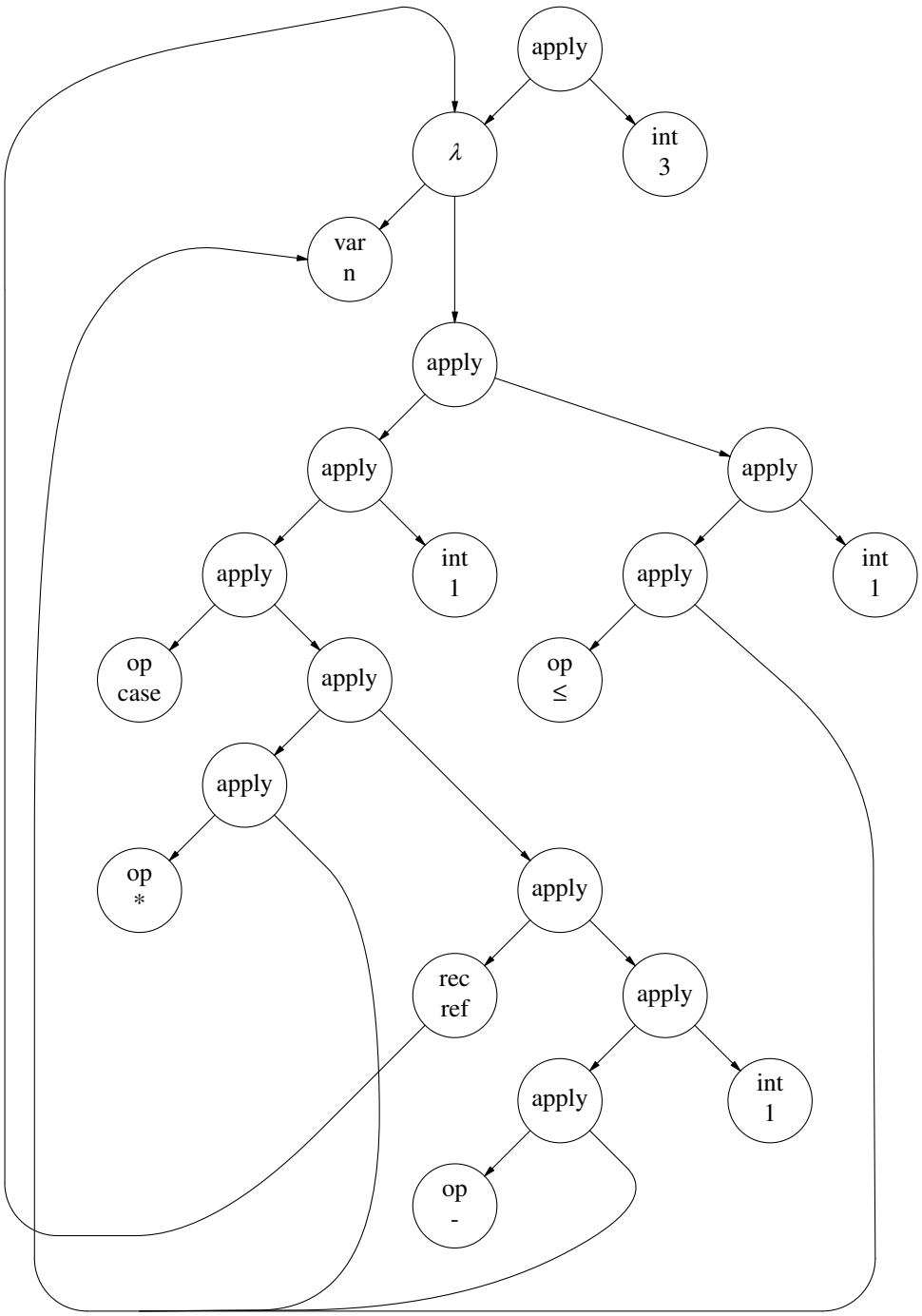
These conditions are not difficult to satisfy and that, provided they are satisfied, reference counting of strong pointers only is all that is needed for safe memory management.

2.3. Example

Consider the graph representing "factorial 3" *before* any graph reduction has taken place. Assuming we have

def fac = λn. **if** n $\leq$ 1 **then** 1 **else** n * fac (n-1)

the graph becomes (or *could* become - it is not unique, due to possible code-sharing):



The operator `case` is a generalised conditional (see [5] for a detailed definition).

3. The "Textual" Format for Describing a Graph

The structure for the textual representation of a graph is a sequence of lines, each representing a separate node in the graph, of the form:

address type usage_count annotation "name" [other fields]

where the fields are separated by blanks or tabs. The field address is an unsigned integer representing the

storage location of the node. The field type is an unsigned integer representing the type of the node (integer, real, application, etc.). The field usage_count is an unsigned integer used for garbage collection purposes (reference counts, etc.), and in our implementations represent the number of *strong* (i.e. not weak) pointers to that node in the graph. The field annotation is an integer currently not assigned, but may be used in the future by particular implementations. The meaning of a program should be unchanged if all the annotations are ignored. The "name" field is a character string naming the node, usually null; for pragmatic reasons, the length of "name" is restricted to 255 characters.

Only printable ASCII characters (codes 32 to 127 decimal inclusive) may be used in GCODE, with the exception of space characters newline, horizontal tab, and formfeed (ASCII 9, 10 and 11). A result of this convention is that blank lines are ignored.

Comments may be included by commencing a line with the character `#`; the comment extends to the next newline character.

If desired, a GCODE line may "overrun" on two or more textual lines; this is desirable for large sum-product nodes.

For example, the node at address 111 which is an apply node called "fred", with left and right descendants at addresses 222 and 333 respectively, usage count of 1, with no annotation, would be represented by:

```
111 5 1 0 "fred" 222 333
```

Standard "C" language conventions apply in the name, thus for example a node called `"aSilly\012\013\n\tName"` would be acceptable. Similarly all other fields use the appropriate "C" lexical conventions. The allowed escapes are:

`\t`	tab
`\n`	newline
`\'`	single-quote
`\"`	double-quote
`\a`	"alarm" or "bell"
`\b`	backspace
`\ooo`	octal number - o in range 0-7
`\Xnnn`	hex number - n in range 0-F

The first line of the file *must* be a header of the form

`#GCODE`...

where "`#GCODE`" is a "magic number" used to identify the file. See the section on Headers below. The following lines of GCODE may come in *any* sequence - since they merely describe a graph, their order is immaterial.

The available types are

Undefined	0
Integer	1
Real	2
Sum-Product	3
Lambda	4
Apply	5
Recursive Reference	6
Operator	7
Variable	8

The following subsections define the fields in detail for the various types of nodes described by GCODE.

3.1. The Type "Integer"

address 1 usage_count annotation "name" value_of_the_integer

where value_of_the_integer is a (signed) integer. No restriction on the size of the integer is currently imposed, though machine dependencies will naturally come into play.

3.2. The Type "Real"

address 2 usage_count annotation "name" value_of_the_real_number

where value_of_the_real_number is a real number written using "C" conventions. Again, no size restrictions are currently imposed.

3.3. The Type "Sum-Product"

address 3 usage_count annotation "name" size tag [first [second ...]]

where a sum-product domain is thought of as a tagged tuple. size is the size of the tuple, first is an integer which is the address of the first element of the tuple, and so on.

3.4. The Type "Lambda"

address 4 usage_count annotation "name" bv body
where bv and body are integers which are the addresses of the bound variable of the lambda node and the body respectively.

3.5. The Type "Apply"

address 5 usage_count annotation "name" left right

where left and right are the addresses of the descendants of the apply node. We can think of left as a function taking one argument (right).

3.6. The Type "Recursive Reference"

address 6 usage_count annotation "name" recref

where recref is the address of the node which is used recursively, that is, which is pointed to by a *weak* pointer.

3.7. The Type "Operator"

address 7 usage_count annotation "name" operator [qualifier1 [qualifier2]]

where operator is an unsigned integer representing a predefined operator, and the last two fields are integers for use in the case where there is a family of operators all of the same name (such as the `SEL` of FLIC).

3.8. The Type "Variable"

address 8 usage_count annotation "name"

is used for bound or free variables. Where the variable is bound within a lambda binding, each occurrence of that variable must be the *same* shared node.

3.9. The Type "Undefined"

address 0 usage_count annotation "name"

is a non-strict $\perp$.

3.10. Example: The Factorial Function

The following GCODE program describes the example in section 2 above.

```
#GCODE,TEXT Factorial using standard recursive algorithm
   0  5   1    0 "MAIN" 1 2
   1  4   1    0 "lambda" 3 4
   2  1   1    0 "integer" 3
   3  8   4    0 "n"
   4  5   1    0 "apply" 5 11
   5  5   1    0 "apply" 7 14
   6  5   1    0 "apply" 8 9
   7  5   1    0 "apply" 10 6
   8  5   1    0 "apply" 12 3
   9  5   1    0 "apply" 15 16
  10  7   1    0 "CASE" 30 2
  11  5   1    0 "apply" 13 21
  12  7   1    0 "INT*" 4
  13  5   1    0 "apply" 20 3
  14  1   1    0 "integer" 1
  15  6   1    0 "recursive reference" 1
  16  5   1    0 "apply" 17 18
  17  5   1    0 "apply" 19 3
  18  1   1    0 "integer" 1
  19  7   1    0 "INT-" 3
  20  7   1    0 "INT<=" 16
  21  1   1    0 "integer" 1
```

4. The "Binary" Format for Describing a Graph

In order to facilitate speedy inter-process communication, a binary compact description of a graph has been produced. This can be thought of as containing, in a compact non-textual format, the same information as the textual format GCODE. When this code is used, it is expected that communication speed will be the goal.

To distinguish this code from the text GCODE, we use the name *BINARY GCODE*.

The general structure of BINARY GCODE is a sequence of logical "lines" (as GCODE), but stored in a processor-dependent binary format. This is portable only between compatible processors, and will not be discussed further here.

However, the first few characters of a BINARY CODE file will be a "header", commencing with the "magic number" `#GCODE` and followed by the keyword `BINARY`. Following this there may be a number of printable ASCII characters followed by a newline character '`\n`'. See the next section on Headers.

5. Headers

Each GCODE input, whether TEXT or BINARY, *must* have a one-line header. The header may be of arbitrary length, terminated by a newline character.

The header line commences

`#GCODE`

and is followed by any number of the following, separated by blanks and/or commas:

`TEXT`	To indicate that the GCODE is TEXT GCODE
`BINARY`	To indicate that the GCODE is BINARY GCODE
`SASL`	To indicate the GCODE was generated automatically from a SASL program
`FP`	To indicate the GCODE was generated automatically from an FP program

`FLIC` To indicate the GCODE was generated automatically from a FLIC program.

It is envisaged that other allowable keywords will be introduced at a later date. A GCODE file which contains neither of the keywords `TEXT` or `BINARY` will be assumed to be TEXT GCODE. Where there is a conflict between elements of the header, such as `TEXT` or `BINARY`, the last one to appear will take precedence.

The rest of the header line, after all keywords have been exhausted, will be arbitrary text typically being a name or brief description of the program.

6. Execution

Logical and character nodes follow the FLIC conventions in which they are represented as sums of products. Thus `FALSE` is regarded as a Sum-Product being a 0-tuple with tag 0, `TRUE` a 0-tuple with tag 1. A character is a 0-tuple with tag the ASCII code for the character.

At the start of a program execution, the node with address 0 is assumed to be the 'root-node'. Lazy evaluation is assumed, unless applicative order is (a) specified by an operator such as `STRICT` in FLIC, (b) implied by an operator such as integer plus, or (c) specified by an annotation.

7. Conclusions

A graphical representation of a functional program in the FLIC intermediate code, or a similar language, has been suggested as a standard internal form for functional programs. Full details of the internal format have not been specified because they are likely to be somewhat machine dependent and, in any event, communication of programs between different sites is not likely to take place via direct memory dumps! Instead, a precisely defined printable format for the graph has been given and this is the level at which communication is expected.

The graphical representation is based on FLIC and supports the FLIC operator set in full. The common Turner combinators are supported as well. The representation also permits (but does not require) all nodes of the graph to be named and annotated with additional information (which must not affect the meaning of the program in the functional sense).

The aim of this standard graphical representation is to encourage the use of compatible representations in work on functional programming languages which is being carried out at many sites, rather than the proliferation of many incompatible representations which differ in arbitrary, but often quite trivial, ways.

8. Changes (June 1990)

The following principal changes to GCODE from the 1987 [4] definition are:

* Type "Unique" has been abolished.
* Types "Sum" and "Product" have been abolished and replaced by a single type "Sum-Product".
* The integers representing the types have been renumbered.
* The "comment" symbol has been changed from `*` to `#`.
* A specification for BINARY GCODE has been introduced.
* The 'root node' is now numbered 0 (not 1).
* Operators now have small integer codes (not 4-digit hex codes).

References

1. T.H. Axford, "Lecture Notes on Functional Programming," Internal Report CSR-86-13, Department of Computer Science, University of Birmingham, Birmingham, UK (1986).
2. T.H. Axford, "Reference Counting of Cyclic Graphs for Functional Programs," *Computer Journal,* 33, 5, pp. 466-470 (1990).
3. D. Berry, *The Pifl Programmer's Manual,* Department of Computer Science, University of Warwick, Coventry, UK (1981).

4. M.S. Joy and T.H. Axford, "A Standard for a Graph Representation for Functional Programs," *ACM SIGPLAN Notices,* 23, 1, pp. 75-82 (1988). University of Birmingham Department of Computer Science Internal Report CSR-87-1 and University of Warwick Department of Computer Science Research Report 95 (1987).

5. S.L. Peyton Jones and M.S. Joy, "FLIC - a Functional Language Intermediate Code," Research Report 148, Department of Computer Science, University of Warwick, Coventry, UK (1989). Revised 1990. Previous version appeared as Internal Note 2048, Department of Computer Science, University College London (1987).

6. D.A. Turner, *SASL Language Manual,* University of Kent, Canterbury, UK (1979). Revised 1983, 1989 and 1990.

9. Appendix: Predefined Operators

We present the operators currently supported, all of which correspond with the semantics of FLIC (except the combinators, which follow their usual definitions), and are all curried operators. The word "var" in the table is simply an abbreviation for "variable".

Code	Arity	Type of Args.	Type of Result	Description
1	1	Int	Int	Unary integer minus (FLIC `INT_`)
2	2	Int,Int	Int	Binary integer plus (FLIC `INT+`)
3	2	Int,Int	Int	Binary integer minus (FLIC `INT-`)
4	2	Int,Int	Int	Integer multiplication (FLIC `INT*`)
5	2	Int,Int	Int	Integer division (truncated) (FLIC `INT/`)
6	2	Int,Int	Int	Integer remainder after division (FLIC `INT%`)
7	1	Real	Real	Unary real minus (FLIC `FLOAT_`)
8	2	Real,Real	Real	Binary real plus (FLIC `FLOAT+`)
9	2	Real,Real	Real	Binary real minus (FLIC `FLOAT-`)
10	2	Real,Real	Real	Real multiply (FLIC `FLOAT*`)
11	2	Real,Real	Real	Real division (FLIC `FLOAT/`)
12	1	Real	Int	Real to integer by truncation (FLIC `FLOAT->INT`)
13	1	Int	Real	Integer to real conversion (FLIC `INT->FLOAT`)
14	2	Int,Int	Sum-Prod	< for integers (FLIC `INT<`)
15	2	Int,Int	Sum-Prod	> for integers (FLIC `INT>`)
16	2	Int,Int	Sum-Prod	<= for integers (FLIC `INT<=`)
17	2	Int,Int	Sum-Prod	>= for integers (FLIC `INT>=`)
18	2	Int,Int	Sum-Prod	= for integers (FLIC `INT=`)
19	2	Int,Int	Sum-Prod	!= for integers (FLIC `INT!=`)
20	2	Real,Real	Sum-Prod	< for reals (FLIC `FLOAT<`)
21	2	Real,Real	Sum-Prod	> for reals (FLIC `FLOAT>`)
22	2	Real,Real	Sum-Prod	<= for reals (FLIC `FLOAT<=`)
23	2	Real,Real	Sum-Prod	>= for reals (FLIC `FLOAT>=`)
24	2	Real,Real	Sum-Prod	= for reals (FLIC `FLOAT=`)
25	2	Real,Real	Sum-Prod	!= for reals (FLIC `FLOAT!=`)
26	var	var	Real	Tags a node to make sum (FLIC `PACK`)
27	var	var	Real	Returns element of sum-tuple (FLIC `SEL`)
28	var	var	Real	Untags a sum node (FLIC `UNPACK`)
29	var	var	Real	Untags a sum node (strict) (FLIC `UNPACK!`)
30	var	var	Real	Extracts value of sum (FLIC `CASE`)
31	1	Sum-Prod	Real	Extracts tag of a sum (FLIC `TAG`)
32	2	var	Sum-Prod	< polymorphic (FLIC `POLY<`)
33	2	var	Sum-Prod	> polymorphic (FLIC `POLY>`)
34	2	var	Sum-Prod	<= polymorphic (FLIC `POLY<=`)
35	2	var	Sum-Prod	>= polymorphic (FLIC `POLY>=`)

36	2	var	Sum-Prod	= polymorphic (FLIC `POLY=`)
37	2	var	Sum-Prod	!= polymorphic (FLIC `POLY!=`)
38	2	var	var	Non-lazy evaluation (FLIC `STRICT`)
39	var	var	var	K n m x0 ... xn-1 -> xm (FLIC `K`)
40	1	Sum-Prod	Sum-Prod	Input a file to a character string (FLIC `INPUT`)
41	1	Real	Real	Real exponential (FLIC `EXP`)
42	1	Real	Real	Real natural logarithm (FLIC `LN`)
43	1	Real	Real	Real sin (FLIC `SIN`)
44	1	Real	Real	Real cos (FLIC `COS`)
45	1	Real	Real	Real inverse tan (FLIC `ARCTAN`)
46	1	Real	Real	Real square root (FLIC `SQRT`)
47	1	Real	Real	Real tan (FLIC `TAN`)
48	1	Real	Real	Real inverse sin (FLIC `ARCSIN`)
49	1	Real	Real	Real inverse cos (FLIC `ARCCOS`)
50	2	Real,Real	Real	^ for reals (FLIC `FLOAT^`)
247	1	var	var	Ix -> x
248	3	var	var	Sfgx -> fx(gx)
249	2	var	var	Kxy -> x
250	3	var	var	Bfgx -> f(gx)
251	3	var	var	Cfgx -> fxg
252	4	var	var	S'kfgx -> k(fx)(gx)
253	4	var	var	B'kfgx -> kf(gx)
254	4	var	var	C'kfgx -> k(fx)g
255	1	var	var	Yx -> x(Yx)