JOINT ROYAL MELBOURNE INSTITUTE OF TECHNOLOGY AND
COMMONWEALTH SCIENTIFIC AND INDUSTRIAL RESEARCH ORGANISATION
PARALLEL SYSTEMS ARCHITECTURE PROJECT

The RMIT Data Flow Computer
DL1 User's Manual
TR 112059R

M. Rawling †
C.P. Richardson ‡

† Department of Communication and Electronic Engineering
Royal Melbourne Institute of Technology
124 Latrobe St
Melbourne 3000

‡ Department of Computer Science,
University of Manchester,
Oxford Road, Manchester, M13 9PL,
England.

Version 1.0 Original Document May 1980 Revised Sept 1987

ABSTRACT:

DL1 is a high level language which produces code for the RMIT dataflow machine. This document is a user's manual for this langauge.

TABLE OF CONTENTS

1. INTRODUCTION

DL1 is an applicative, dataflow language which does most of its processing by the application of (functional) operators to values to produce new values. This value oriented approach is highly compatible with the run time mechanism of the target machine and is typical of languages designed for dataflow multi-processors [7].

The language was designed to aid in the development of a dataflow multi-processor system built [1,5] at Manchester University. The need for such a language was indicated by previous research [2] in which a system to support object recognition programs was constructed. A compiler for DL1 written in Pascal is available on the MU5 computing system at Manchester University [3] and an upgraded version runs under Unix in the Department of Communication and Electronic Engineering RMIT, Melbourne.

Although DL1 is a high level textual language, its format allows for easy visualisation of the object graph described by the source program. The syntax is similar to TDFL as proposed by Weng [4]. Syntax charts are appended and many examples are given in the following sections.

Output from the compiler is in the form of an intermediate language (ITL) which consists of a list of node descriptions and priming tokens. The form of this is specified in a previous document [5].

Recent work at RMIT [8] has led to the introduction of an extended node set and high level language (DL1) features which take advantage of this. In particular, support for streams and eager/lazy evaluation has been successfully incorporated into DL1.

2. STATEMENTS

There are five basic types of statement:

i Functional Definitions.
ii 'SWITCH' Statements.
iii 'JOIN' Statements.
iv 'PRIME' Statements.
v 'PROTECT' Statements.

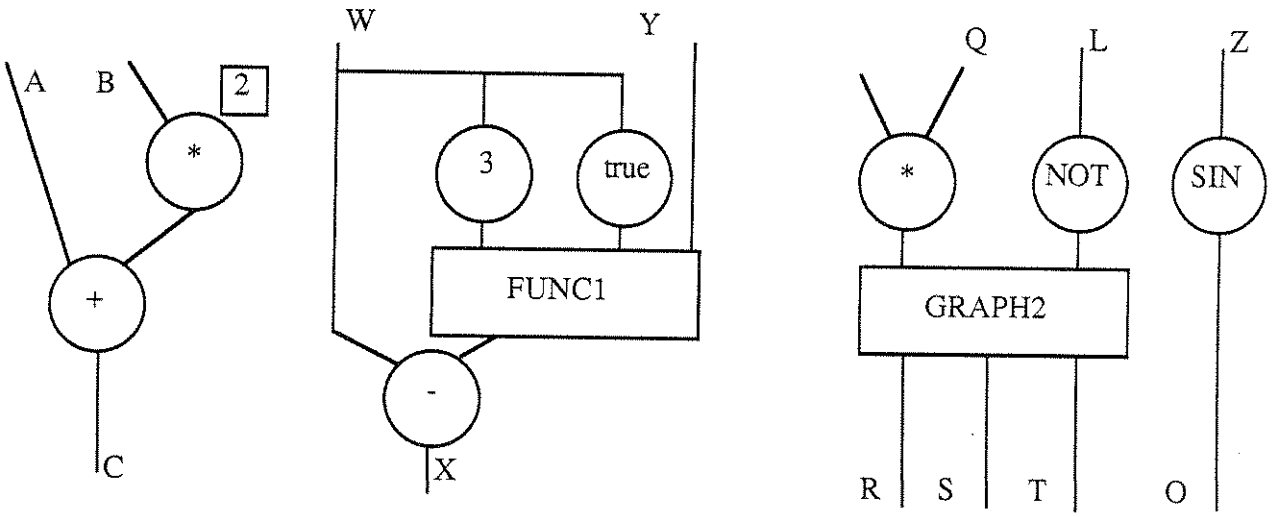
2.1 The Functional Definition

In DL1, the equivalent of an imperative assignment statement is the functional definition. This statement simply defines the relationship between one set of identifiers and another, it does not imply any order of execution (control flow), nor does the 'truth' of the definition depend upon its position in the program segment over which it applies (usually a syntactic element such as 'body') or the time at which it executes. This freedom of interaction due to a lack of side effects is what allows statements to execute in parallel, constrained only by the data dependencies between them.

A DL1 program may be viewed as a high-level representation of a graph consisting of nodes interconnected by arcs, a definition is the specification of one or more arcs in terms of some expression involving other arcs. For example:

```
A + B*2 -> C;                                          (1)
W - FUNC1( Y, 3, TRUE) -> X;                           (2)
GRAPH2( P*Q, NOT L ), SIN(Z) -> R, S, T, O;            (3)
```

Note statement (3) which shows a multiple definition. In general there is no limit on the number of outputs a function may have. The corresponding graphical forms produced by the DL1 compiler are:



Example 1 Example 2 Example 3

Figure 2.1.1

2.2 The 'SWITCH' Statement

A **switch** statement is used when the path of a computation depends upon some boolean condition. It is a nonfunctional form since it does not produce a token on each output arc and without careful use may lead to poorly formed graphs. It provides a type of data driven control by explicitly defining a branching point in the dataflow graph and can be used to force *conditional statement execution* by appropriately directing token traffic.

The following are examples of how the **switch** statement is used to direct a set of tokens depending on a boolean expression and the graph which is compiled for each statement.

switch B **or not** (X = Y) **then** C-A, 3, D ->
D, E, **null**
else
null, F, G;

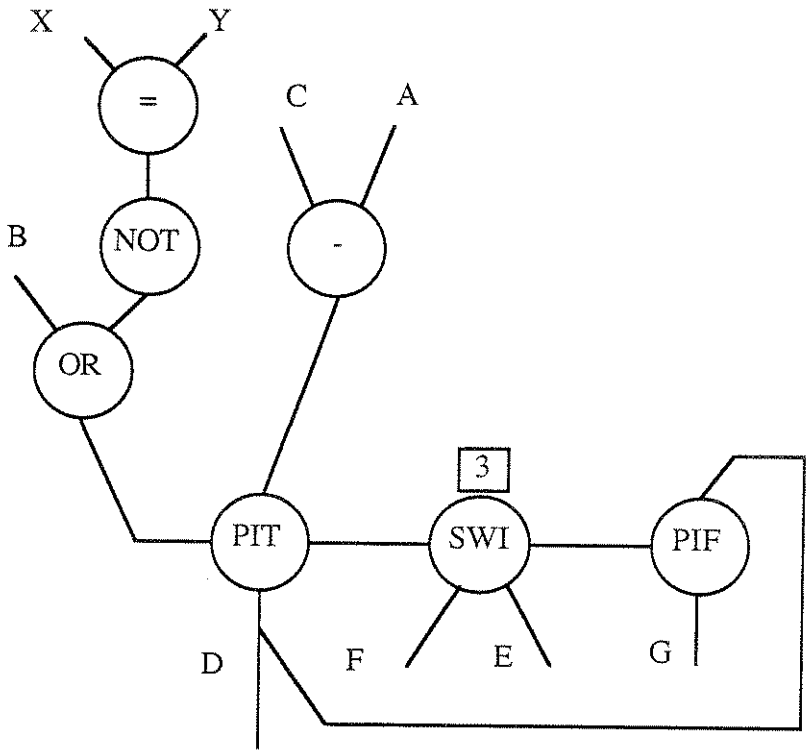


Figure 2.2.1

switch BOOL **then** X, Y, Z -> U, V, W;

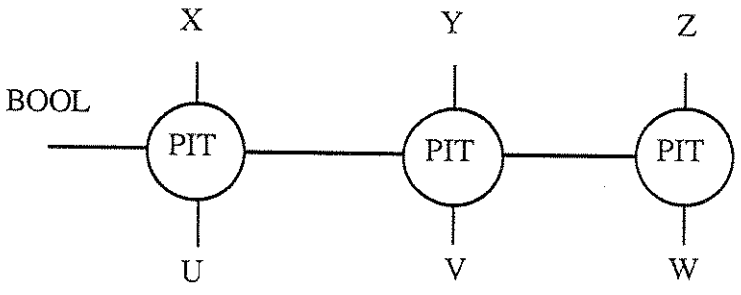


Figure 2.2.2

switch BOOL **then** A, B -> **else** C, D;

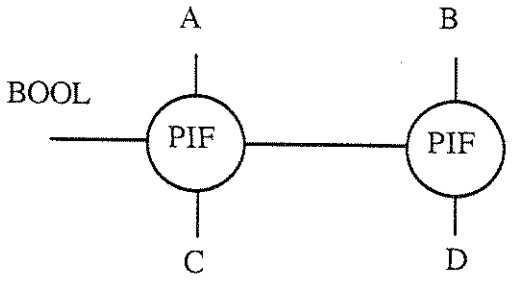


Figure 2.2.3

Either side of the output specifications may be left out as in figures 2.2.2 and 2.2.3. This textual form is far neater than if the full **switch** statement were employed by padding out the output lists with **null** destinations.

2.3 The 'JOIN' Statement

Consider the following sets of statements and their compiled graph:

(A) **switch** BOOL **then** BOOL -> ENABLET **else** ENABLEF;
on ENABLET **then** EXPR1 -> 01;
on ENABLEF **then** EXPR2 -> 02;
merge(01,02) -> RESULT;

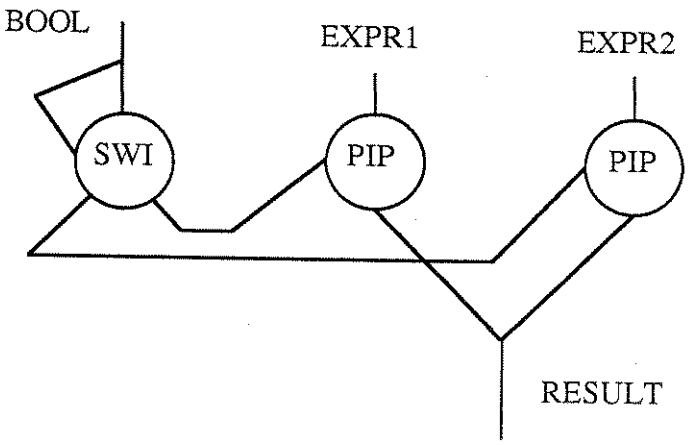


Figure 2.3.1

(B) **switch** A>B **then** A, B -> MAXA, **null else null**, MAXB;
merge (MAXA,MAXB) -> MAX;

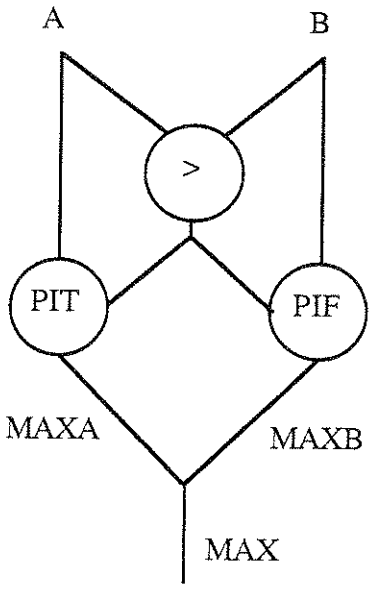


Figure 2.3.2

In both cases there is a selection of one of two expressions for the single output arc, depending on a boolean condition. There is however a slight difference in the two cases. In the first example, the coding method assumes that only one of the separate paths will receive a token for each boolean token and also that the boolean will select the right path. In the second example, the coding method relies on there being a token on each path; one of these is enabled while the other is absorbed.The **oldif** and **join** statements provides a neat high-level form as follows:

(AA) **oldif** BOOL **then either** EXPR1 **else** EXPR2 -> RESULT;
(BB) **oldif** A>B **then else** A **else** B -> MAX;

OR

(AA) **join** BOOL **then** EXPR1 **else** EXPR2 -> RESULT;

Note that **oldif** is provided only to maintain compatability with an older version of DL1 and is only temporary, new programs should not use **oldif**. The **join** form is 'either' by default. The 'else' form of **oldif** is a functional form form that is now provided more flexibly by the conditional expression. The **join** statement on the other hand is a nonfunctional form designed for use with **switch** to give conditional statement execution. Great care must be taken by the user to account for all tokens used and generated by the **switch/join** statement block if a well formed graph is to be produced. Conditional expressions (**if then else**) are now provided by DL1 which guarantee both well formed and determinate graphs.

2.4 The 'PRIME' Statement.

Graphs may be primed by using the **prime** statement. It may be used to specify a list of tokens which are to be placed on a number of arcs.

prime 2,3 -> A;

The order in which the tokens are sent is the same as the order in which they appear in the list. Named constants may be used in the token list. There may be more than one arc to which a particular list is sent, in which case a copy of the priming tokens is sent to each arc, eg:

prime -1, PI -> A,B, C;

A list of **prime** statements may be strung together between a **begin** and an **end** in the following manner:

prime
begin

```
        TRUE -> B1, B2;
        -10, 0, 10 -> 1.2;
        PI, 7.2 -> R1, R2, R3;
end;
```

A shared subgraph may not be primed due to restrictions in the copy number mechanism.

2.5 The 'PROTECT' Statement.

Should a section of graph need to execute in isolation, it would be necessary to isolate it with a set of PIP nodes at the inputs to the critical area and enable a new set of tokens to enter only when all the (critical) results have been returned. The **protect** statement provides this facility in a high-level form. For a critical area with inputs I1, I2, I3 and outputs 01, 02, the protection could be invoked by writing:

protect I1, I2, I3 **with** O1, O2;

The corresponding graph is shown in figure 2.5.1:

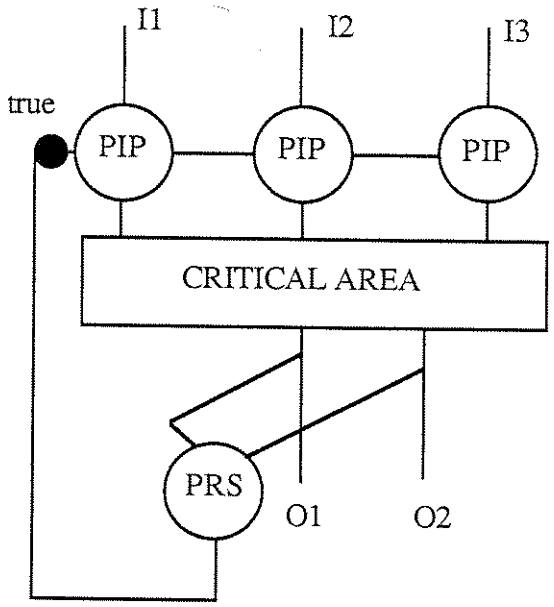


Figure 2.5.1

Note that the previous graph has issued a priming token in order to allow the first set of input tokens to enter the critical area. Such initialisation could also be achieved using the following construct:

```
O1 -> OA; O2 -> OB;
when (OA, OB) -> ENABLE;
on ENABLE then I1, I2, I3 -> IA, IB, IC;
prime TRUE -> ENABLE;
```

When is a pre-defined function which returns a boolean true whenever a token appears on all of its input arcs. **When** uses a node called PRS which emits a boolean TRUE on the presence of a token of any type on both input arcs. A priming token is then needed to start the graph by allowing in the first set of parameters.

The **protect** statement may only be used with arcs that are defined both as input and as output. A new form of **protect** is now available which does not require any priming tokens and can thus be used in a shared subgraph. This is achieved by replacing the PIP nodes with PRT nodes which effectively behave as a primed PIP node. This scheme has the advantage of not only starting in an empty state but also returning to an empty state upon completion. Graphs which have this property

are said to be well formed and successful execution should result in the matching stores of the multiprocessor being empty at the end of each run. This scheme not only saves on memory utilisation but can be used to detect correct execution of a well formed graph.

3. SUBGRAPHS

A facility exists for the definition of subgraphs, the DL1 equivalent of procedures. Two types of subgraph are supported by DL1 at the textual level. Ordinary subgraphs are macros which are expanded when called whereas shared subgraphs are interfaced by means of special nodes and allow recursion. The sharing mechanism is explained in [1]. Subgraphs may be defined inside other subgraphs in a block structured manner. Ordinary subgraphs constitute a new textual level but their level of 'sharing' is the same as the level of 'sharing' of the block in which they are defined.

3.1 Subgraph Declarations

An ordinary subgraph declaration specifies the name of the subgraph and the names of all the input and output arcs and their types as follows:

```
subgraph EXAMPLE (IN1: real; IN2, IN3: integer) ->
        (OUT1: real, integer; OUT2: boolean);
```

This subgraph may be called in the following manner:

```
EXAMPLE( SIN(X), I, 2 ) -> R, B;
```

In DL1, a function is merely a subgraph (ordinary or shared) with one output arc, named or unnamed, so the following subgraph:

```
subgraph MAX(A,B: integer) -> (OUT: integer);
```

or

```
subgraph MAX(A,B: integer) : integer;
```

could be called as a factor in any expression:

```
Y * MAX(X,3) -> Z;
```

3.2 Shared Subgraphs

Recursive subgraphs must of necessity be shared subgraphs. These are declared with the key word **shared** as in the following example:

```
shared subgraph FAC(I:integer) -> (RESULT:integer);
```

DL1 allows the definition of partial functions and so the user may specify the arcs which are to be used as return address triggers (one for each output arc) by appending them to the end of the parameter list in the function call. If they are omitted, as is the usual case, a suitable trigger is generated in the same manner as when a trigger is needed for constants in an expression, i.e., from the first arc named in the statement.

```
FAC( N ) * FAC( R, TRIG ) -> F;
```

Because of the copy number mechanism, a shared subgraph may in general be called a maximum of 8 times at any level of 'sharing'. However at the outermost level, this limit is 250 [2]. DL1 now provides a compiler option which can be used to alter these so called **topoccur** and **maxoccur** values (§ 8.1).

A program for calculating N! could be:

```
shared subgraph FAC(I:integer) :integer;
    begin
        switch  I > 0 then I -> N else ONE;
        on ONE then 1 -> R1
```

```
        N * FAC(N-1) -> R2;
        merge(R1,R2) -> FAC;
end;
```

(nb. for 'safe' code production, the **merge** line should be replaced by a **join**:-

```
        join I > 0 then either R2 else R1 -> FAC;
```

this will ensure correct results even when data is streamed through the above graph)

Enhancements to DL1 now allow conditional expressions of arbitrary arity which provide for much more compact prgrams than **switch/join** combinations. An annotation that gives rise to explicit **lazy** evaluation is also provided for both branches of a conditional expression. The definition of FAC would now be as follows:-

```
shared subgraph FAC(I:integer):integer;
    begin
        if I > 0 then `I * FAC(I - 1) else 1 -> FAC;
    end;
```

See section 5.2 for more details on the syntax and an explanation of the "`" symbol used above.

3.3 Forward Referenced Subgraphs

Subgraphs may be forward referenced, but such a subgraph must be a shared subgraph. We may thus write:

```
forward subgraph F(IN1:integer) :integer;

shared subgraph S(IN1:integer) -> (OUT1:integer);
    begin
        F(IN1) -> OUT1;
    end;

subgraph F;
    begin
        S(IN1) -> F;
    end;
```

4. CONSTANTS

Named constants may be declared at the start of each block and may be referred to within the block in which they are declared.

```
constant
        PI = 3.14159;
        KGTOLB = 2.2;
        BOOL1 = %F9A;
```

In the text, we may want to trigger a constant from a particular arc. DL1 provides a pre-defined function for doing so called **issue** which takes two parameters, the first being the arc which is to be used as a trigger and the second is the constant which is required. Using this, we could rewrite the factorial subgraph as follows:

```
shared subgraph FAC(IN1:integer) -> (OUT:integer);
        begin
                I -> S;
                switch S>0 then S -> N else ONE;
                merge(issue(ONE, 1), N*FAC(N-1) ) -> OUT;
        end;
```

5. EXPRESSIONS

Functions are often provided with multiple outputs in dataflow languages to increase their power and flexibility [7], however many languages make only limited use of this feature. The original DL1 provides a good example of this, as the only way a multiple output subgraph could be called was in the multiple definition **assign2**, which consisted of a call to the subgraph followed by the definition of an output list.

The syntax of DL1 (§ 9), now includes several different types of expressions as well as the powerful syntactic object **expression list**. The latest version provides a more unified syntax in that expressions are generalised to multiple outputs and expression lists can be built up incrementally from not just unary expressions but also from multiple output functions, including subgraphs, conditionals and deferred expressions with multiple outputs. The **comma** operator provides the means for building an expression list and allows recursive calls to **expression list** itself, see the diagrams in section 9.

5.1 Deferred Expressions

Frequently when constructing a graph, we want to delay the output(s) of some expression until the arrival of a data token on a particular arc. This action is usually the enabling of other tokens to pass or the issuing of constants. This may be done by using a **deferred expression** as in the following example:

on CONTROL **then** B - C, FUNC (D), 3 -> E, F, G, H, I;

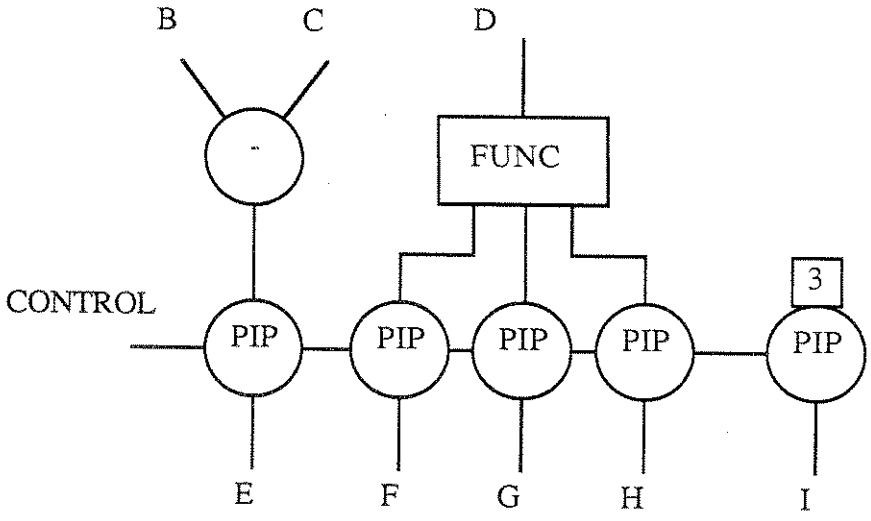


Figure 5.1.1

Fig 5.1.1 shows the compiled graph for the above functional definition involving a deferred expression of arity five. The expression, in this case the arc CONTROL, is used as an enable signal to the root of a duplicate tree. The ends of the duplicate tree are used to enable each subexpression(list) using Pass-If-Present (PIP) nodes. These nodes transmit one token from arc0 on the arrival of a token of any type along arc1. A constant is generated by a PIP node with literal data on input0.

5.2 Conditional Expressions, Lazy and Eager Evaluation

The conditional expression is similar in structure to the deferred expression, however selection

is made between two expressions of the same arity. The syntax for conditional expressions was introduced in the discussion of section 3.2 and we return to that example here:

if I > 0 **then** `I * FAC(I - 1) **else** 1 -> FAC;

The "`' symbol in the conditional expression shown above enforces **lazy** evaluation of the true branch. If **eager** evaluation was used here, then the recursion would continue until copy numbers were exhausted. Eager evaluation is the default mode employed by the DL1 compiler. Lazy evaluation is also recommended to prevent the execution of code which may lead to error conditions such as nonterminating execution or negative square roots, etc. In addition, lazy evaluation may be indicated where eager evaluation would otherwise lead to an excessive amount of wasted computation (our multiprocessor is not an unlimited resource!).

Nested conditionals are allowed by the syntax and any combination of lazy and eager evaluation is allowed. It is perfectly valid to use lazy evaluation for a nested conditional and indeed nested lazy evaluation is both sensible and allowed. It is interesting to note that a similar form of lazy evaluation could also be applied to the deferred expressions of the preceding section but its introduction does not seem necessary at this stage.

An examination of the syntax diagrams will reveal an ambiguity when an expression list begins with the key symbols **if** or **on**. In this case it is not clear whether the parser will find a **conditional (deferred) expression** or a **factor**. This ambiguity arises because the factor **unary conditional (deferred)** uses the same syntax as the more general (multiple output) expression list **conditional (deferred) expression**. The compiler currently resolves the ambiguity in favour of **expression list**. This is the more general case and evaluation as a factor can always be forced by parenthesizing the expression. This slight inconvenience was felt to be more acceptable than introducing a special syntax for the factor form whilst still allowing the use of conditionals and deferals in **terms**. More justification for the syntax used and also complete and detailed code generation templates for DL1 can be found in [8]. Also, the intricacies of nested eager/lazy evaluation combinations and some performance analyses are explained in that reference.

6. SYSTEM FUNCTIONS AND OPERATORS

The system supports a comprehensive set of functions and I/O handling facilities.

6.1 Mathematical Functions

The mathematical functions - (uniry minus), **sin**, **cos**, **tan**, **arcsin**, **arccos**, **arctan**, **log**, **ln**, **exp**, **sqrt**, **sqr**, **abs**, **round** and **trunc** are all provided. Other mathematical functions can be built from these as required.

6.2 Logical Functions

In addition to the normal infix boolean operators, the functions called **setbit**, **clearbit** and **testbit** are supported. These operations set, clear and test the individual bits of a bit-string. They require two parameters, firstly the bit-string for manipulation and secondly the integer position (starting at 0) of the particular bit.

A function for comparing the types of two inputs called **compare** exists and returns a boolean true if the types of both input tokens are identical.

6.3 Input and Output

The functions **read**, **write** and **writeln** are supported and may be called from any position in a graph or subgraph. They are implemented as functions. **Write** returns a copy of the last token written after the operation has been performed. **Read** returns the data read (usually a character). **Read** requires two parameters, the first of which is a device name and the second parameter is the arc on which the request will appear. **Write** also requires the device name as the first parameter but allows more then one arc to specify a list of data which is to be written. Care should be exercised in the use of these functions to prevent writing or reading from two places in a graph at the same time. This could result in a garbled output stream.

Ord and **chr** are provided for character manipulation.

6.4 Stream Functions

The current version of DL1 provides complete support for **streams** which are a 'natural' data structure provided by many dataflow programming languages. Weng [4] introduced the semantics of stream based computation on a static dataflow machine and showed how streams could be used to provide interprocess communications and to increase both available and achieved concurrency. Other researchers have since added stream based computation to their dynamic dataflow architectures. The RMIT target machine is a hybrid architecture which supports both static (queued) and dynamic (coloured) run-time environments. In such an architecture benefits from streams are achieved through enhanced use of static graph code (pipelining) and through significantly more powerful semantics of high level DL1 programs.

Stream 'variables' are indicated by identifiers beginning with an underscore. Valid stream identifiers include :-

```
_this_is_a_stream
_X1
__List3
```

Predefined funtions which provide stream support are:- **bracket**, **unbracket**, **head**, **tail**, **get**, **empty** and **cons**. **Bracket** returns a copy of the input data followed by a 'Stream-end-token'. **Unbracket** absorbs all 'Stream-end-tokens' passing through it. The new

stream functions **head**, **tail**, **get**, **empty** and **cons** allow for easier and safer stream manipulation. **Get** returns both the head and tail of a stream and is slightly more efficient than a **head/tail** pair. **Cons** allows for non-strict stream creation by concatinating a simple element and a stream element. **Empty** returns true or false when applied to a stream, it should be used to test streams before applying **head**, **get** or **tail** since these functions are not defined for empty streams. In addition, implicit stream support is provided by many other DL1 primitives including merging, protecting, shared subgraph entry and exit, etc..

A graph to sum the elements in a stream could be:-

```
shared subgraph stream_sum( _input:integer):integer;
    begin
        if empty(_input)
        then 0
        else ` head(_input) + stream_sum(tail(_input))
        -> stream_sum;
    end;
```

and to sum the elements of two streams to form a new stream:-

```
shared subgraph _add( _in1, _in2: integer):integer;
    begin
        if empty(_in1) or empty(_in2)
        then ]
        else ` cons(head(_in1) + head(_in2),_add(tail(_in1), tail(_in2)))
        -> _add;
    end;
```

where the symbol ']' is the stream-end symbol and denotes an empty stream when used alone. The '`' symbol denotes **lazy** evaluation of the **else** branch. Note that this subgraph is only 'clean' if the input streams are of identical length.

6.5 Miscellaneous Functions

6.5.1 MERGE

The **merge** function is used to explicitly merge several arcs together. It returns the arc to which all the inputs have been merged. It takes two or more parameters of any type. The merging is non-deterministic in nature and is done in the communications network of the host multiprocessor. No object code is actually compiled for a call to this function.

6.5.2 SETDEST and LABEL

Label returns the literal destination of its one parameter which is an arc. Because of its nature, the function cannot be used to generate named destination constants. This is intentional to stop side effects which may arise from sending tokens into subgraphs other than as formal parameters. **Setdest** sends a copy of the first argument to the address given by the second argument. An example of the use of these functions follows:

```
if BOOL then label(ARCA) else label(BRANCH) -> DESTIN;
setdest(DATA, DESTN) -> null;
```

6.5.3 ISSUE

Before **unary deferred expressions** became available as factors, the DL1 programmer could use the **issue** function as a factor in an expression. Its use is now only required to maintain compatability with old source code. It is expected that **issue** will disappear from DL1 in the future.

Compare:

on A **then** TRUE -> I1;
on B **then** 5 -> I2;
merge(I1, I2) -> O1;

with:

merge(**issue**(A, TRUE), **issue**(B,5)) -> O1;

the new syntax allows this definition to be expressed as:

merge(**on** A **then** TRUE **endon** , **on** B **then** 5 **endon**) -> O1;

6.5.4 STORE

This function is implemented using the storage (STO) node. It takes two inputs. A token arriving on arc0 is stored. If a token has already been stored then it is replaced by the new token.

A token arriving on arc1 results in a copy of the stored token being sent out as a result. If there is no token stored then an error is indicated. The **store** function represents this node. Its two parameters represent arc0 and arc1 of the STO node respectively.

6.5.5 YIELD and SETCOPY

Yield and setcopy plant a **YLC** and **STC** node respectively. These functions can be used to control the context of a particular code segment instantiation as is typically required by a resource sharer subgraph [6].

6.5.6 SSR, SSW and SSRW

Structure read (SSR), structure write (SSW) and structure read before write (SSRW) provide access to a stored, as distinct form transmitted, vector of stored tokens. The tokens may be of mixed types, the type being defined by the token written to any given vector element. More complex structures may be defined using techniques of mapping complex objects in a conventional linearly addressed store. SSR has one parameter being the index. SSW and SSRW have two parameters the first of which is the value to be written and the second the index. The functions return the value written or read.

6.6 Operators

6.6.1 Arithmetic Operators

Diadic arithmetic operators provided by DL1 are +, -, *, /, **div**, **mod**, ^, **. These have their usual infix form. The last two are for raising to the power.

'^' uses logs to achieve the result and may have two expressions as operands.

'**' is for raising to an integer power and must be followed by a positive integer constant.

Consider the following statements and their compiled code (figure 6.6.1.1):-

(A + B) ^ (GAMMA - 1) -> O1 O1 ** 5 -> O2

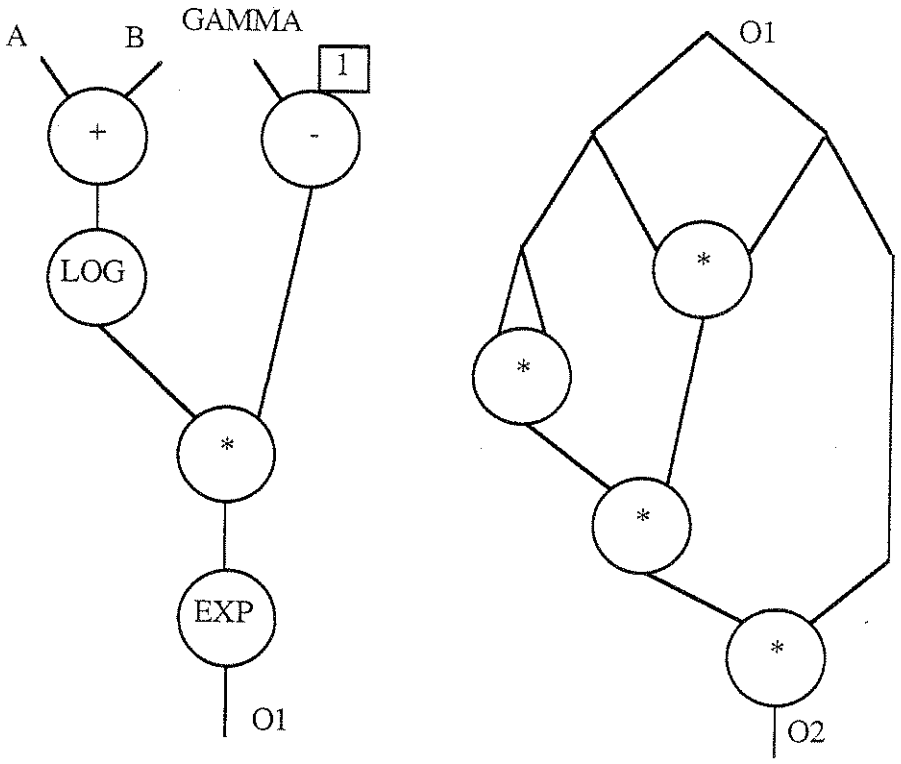


Figure 6.6.1.1

6.6.2 Boolean Operators

The operators **and**, **or**, **xor**, **eqv**, **imp** and **not** are all provided with their usual meanings. **Imp** provides the logical operation of 'implies'.

6.6.3 Relational Operators

'<' '<=' '=' '<>' '>=' and '>'

These are the relational operators, again with their usual meanings.

7. CODING STRATEGY

7.1 Extraction of Parallelism

In order to achieve a high degree of parallelism in the evaluation of a high level expression, the compiler generates balanced trees for the operations of addition, subtraction, OR, multiplication, division and AND :-

A-B+C-D+E-F+G-H -> X;

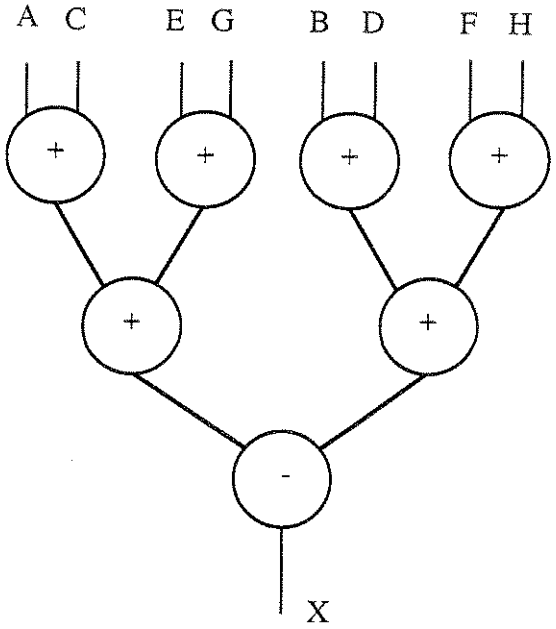


Figure 7.1.1

A/B*C/D*E/F -> X;

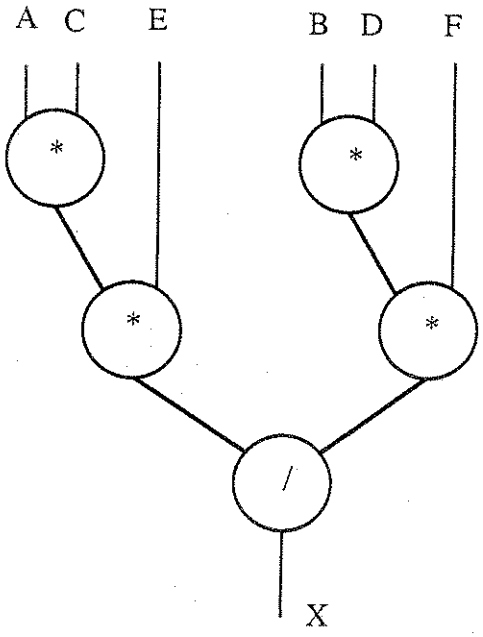


Figure 7.1.2

The compiler thus relieves the programmer of extracting the parallelism by the use of brackets. In the first example the following expression would have been required:

((A+C)+(E+G))-((B+D)+(F+H)) -> X;

.. to achieve this parallelism.

7.2 Triggering Constants

Not all primitive nodes may have literal data associated with their node descriptions. In such cases the constants must be generated using PIP nodes with the necessary literal on arc0. In parsing an expression a strategy is used for deriving the trigger needed to set off the required constants.

The DL1 compiler uses the first arc name encountered in a statement for the triggering of all the constants in that particular statement. If the expression is 'gated' (eg. a **lazy if** branch), then the gate is used to trigger the constants. A duplicate tree is then planted from this trigger and used to set off the required constants. Constants are planted as literals in two-input nodes where appropriate.

SIN(0.5)*6 - A + ORD('C')/2 -> X;

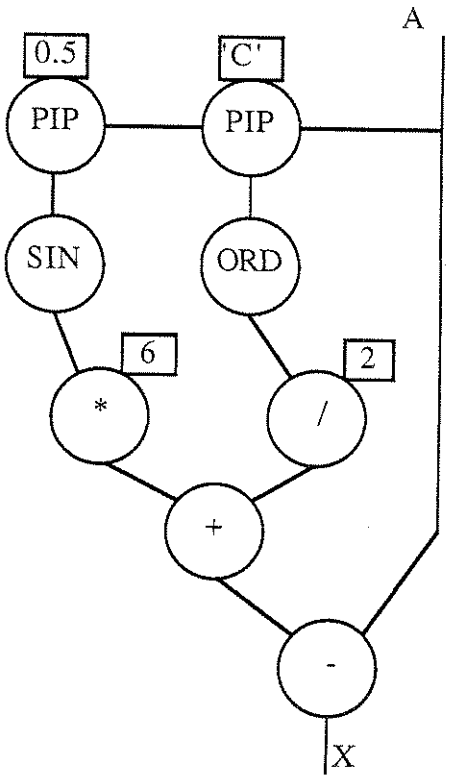


Figure 7.2.1

The above assignment and compiled code illustrates the method of generating constants. Should a statement be encountered which has no suitable trigger, the duplicate tree is primed with a single token to generate the necessary constants and an error is indicated.

SIN(0.5)*6 -> OUT;

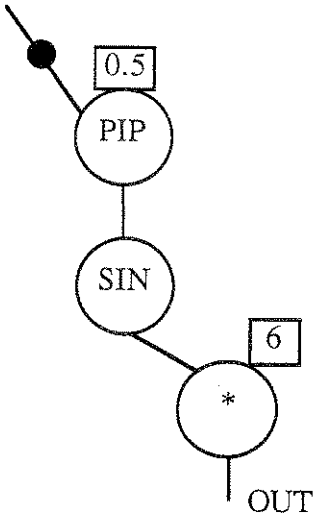


Figure 7.2.2

This is to be discouraged as a means of priming graphs however and the **prime** statement should be used instead.

8. OPTIONS, RESERVED WORDS AND PREDEFINED FUNCTIONS

8.1 Compiler Options

DL1 allows several options to be present in the source file which provide control over code production and compile time statistics display. Options are imbedded between brackets. For example, to provide run time tracing and an informative but brief compile time display, the option string [x+,k+,i+] could be used.

option/default	effect
b -	byte node addressing
c +	compressed ITL listing
d -	determinate code production
e +	error monitoring
f -	finegrain for exponentiation (LNE/MUL vs PWR)
h -	heap cheaking for debugging
i -	display of arc usage
k -	compile time information
l +	compiler generated listing
m +	use new 'O' data format (<maxocc.occ>)
o 8,8	top and max occurrence for shared subgraphs
r +	constant reduction for literals
s -	scanner output for debugging
t -	type checking
w -	extended node set for code production
x -	run time trace between [x+] and [x-]
[0 : 127]	element range for following node assignments

8.2 Reserved Words

The following reserved words have special meanings to the compiler and are not available for any other purpose.

graph delimiters

program, constant, forward, shared, subgraph, begin, end

statement delimiters

switch, join, oldif, protect, prime

expression delimiters

if, then , else, either, endif, on, endon, with

predefined constants

true, false, eos (or ']', end of stream), null (the token bucket)

type identifiers

any, boolean, char, copy, dest, integer, real, stream

operators

and, or, eqv, xor (nqv), not, mod, div, label

i/o streams

input, output, bend, elbow, twist, waist, grip, swivel

8.3 Predefined Functions

The following predefined functions are provided by DL1:

arithmetic

abs, ln, exp, log, pwr, round, trunc, sqr, sqrt,
sin, cos, tan, arccos, arcsin, arctan

bit string

clearbit, setbit, testbit

character

chr, ord

i/o

read, current, write, writeln, las

stream

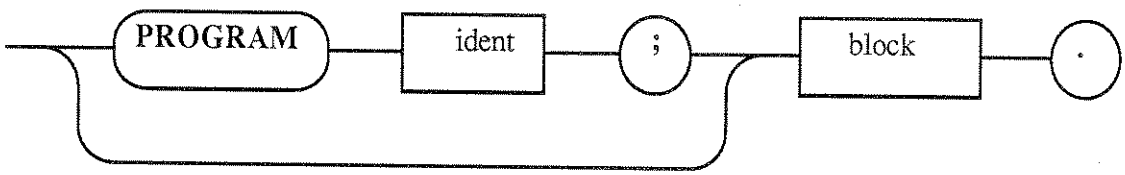
head, tail, get, empty, cons, bracket, unbracket

misc

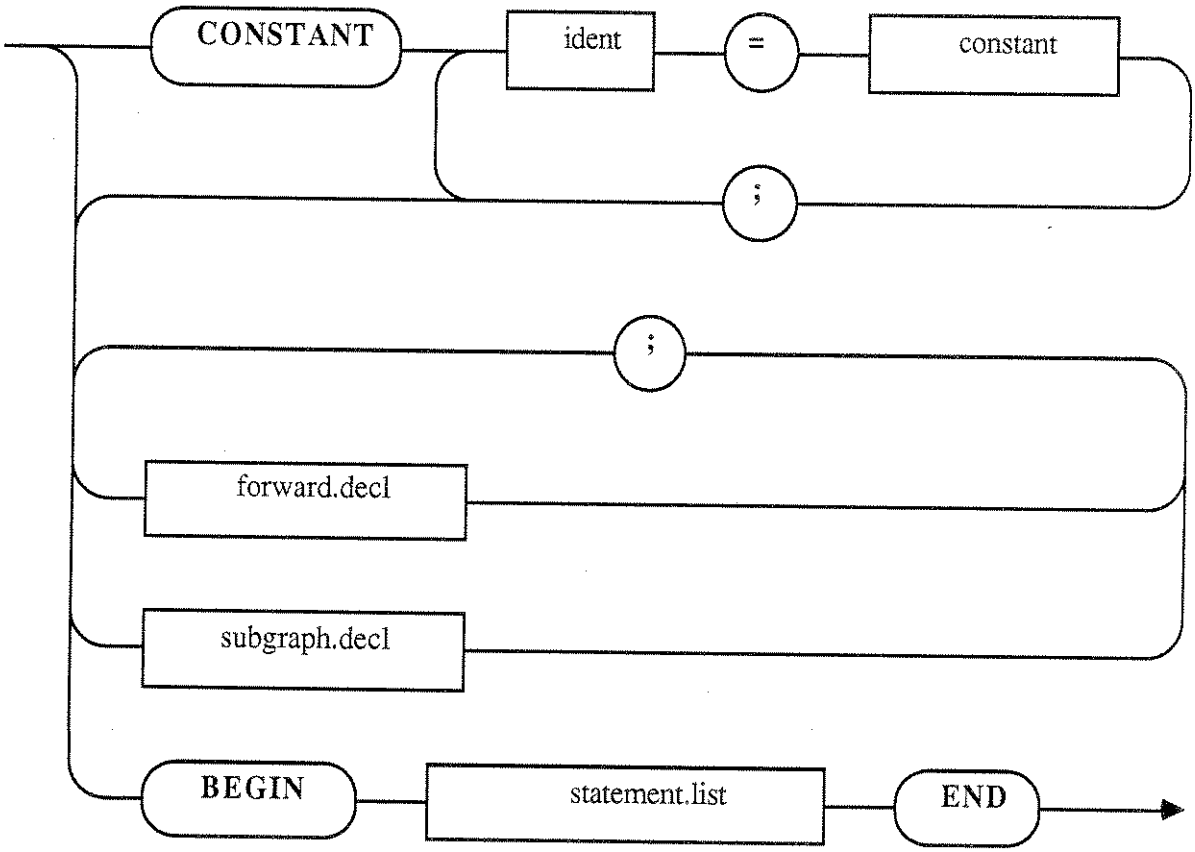
compare, first, issue, merge, pred, succ, store, when,
setdest, setcopy, yield,
ssr,ssw,ssm

9. Syntax Diagrams

program

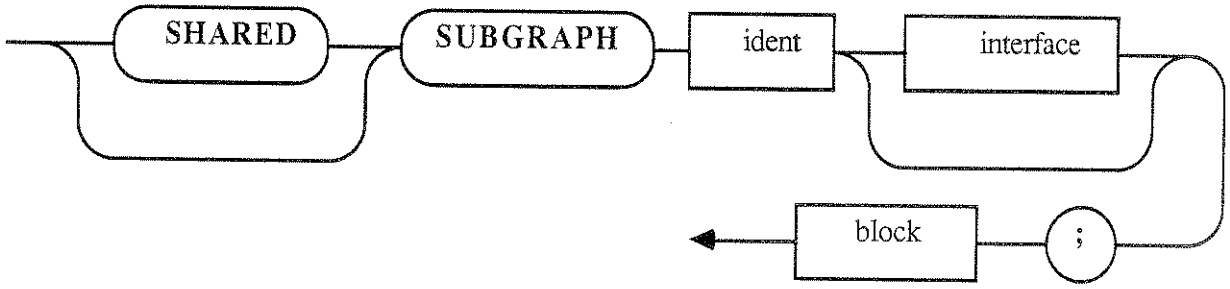


block

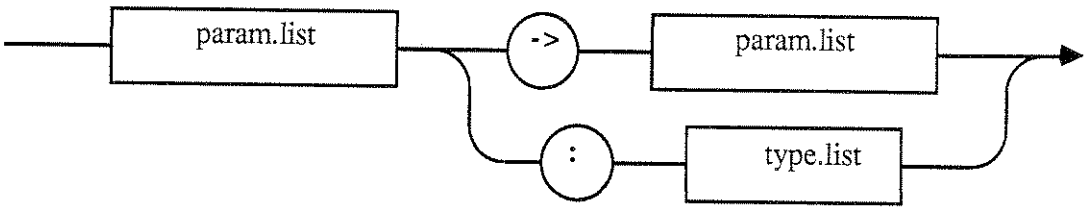


forward.decl

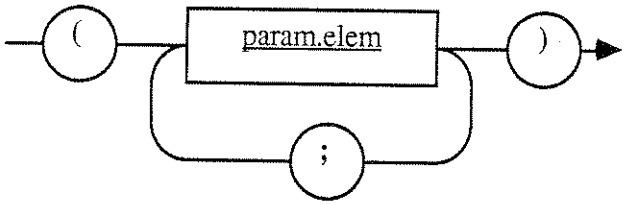
subgraph.decl



interface

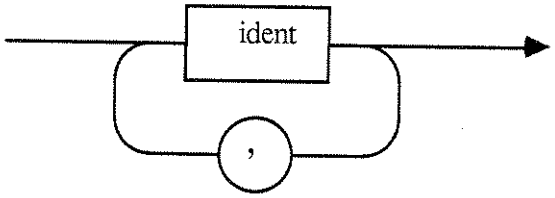


param.list

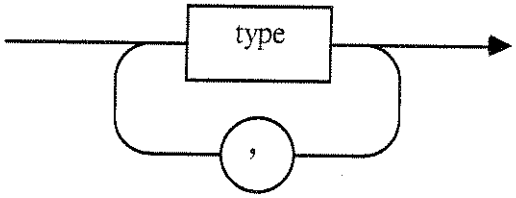


param.elem

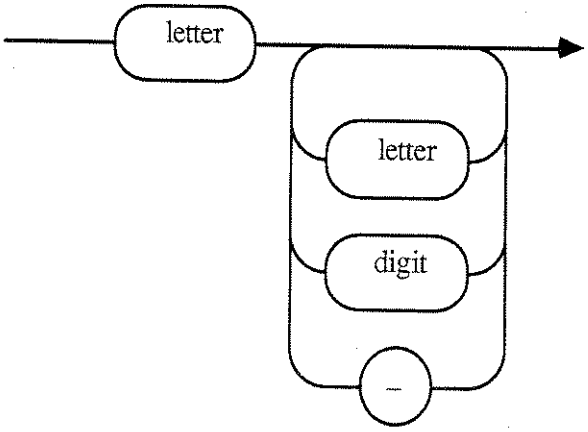
id.list



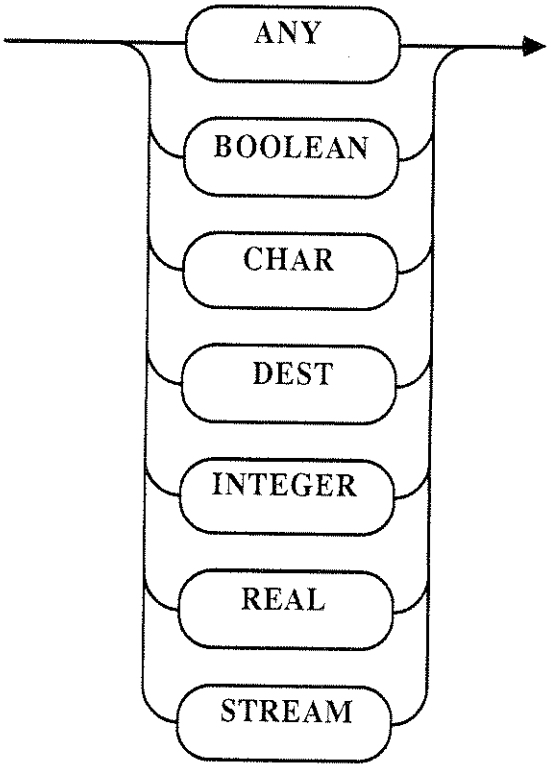
type.list



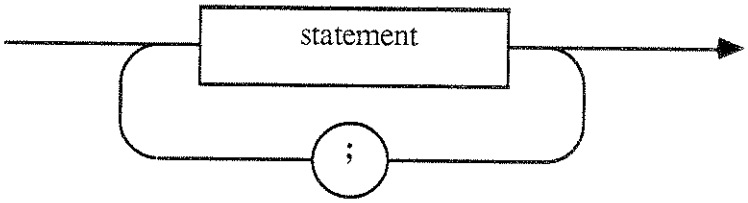
ident



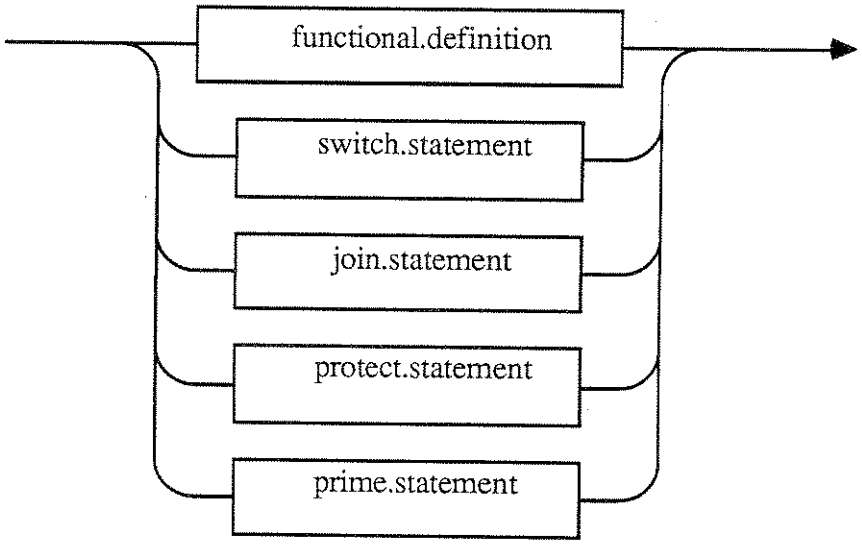
type



statement.list

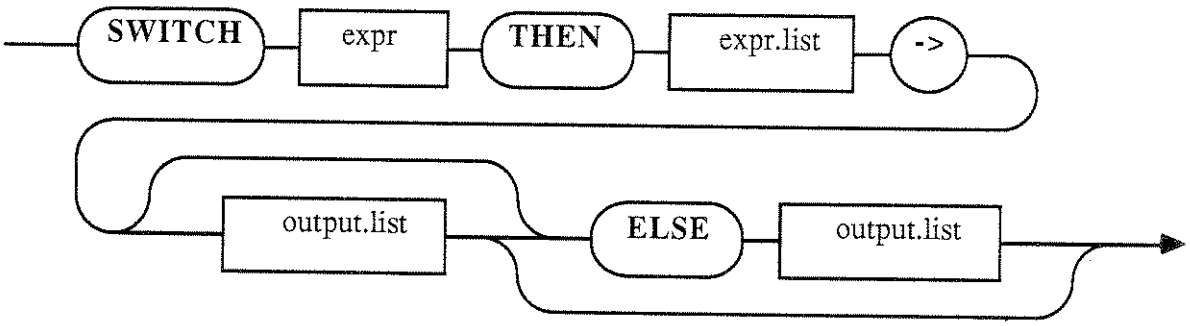


statement

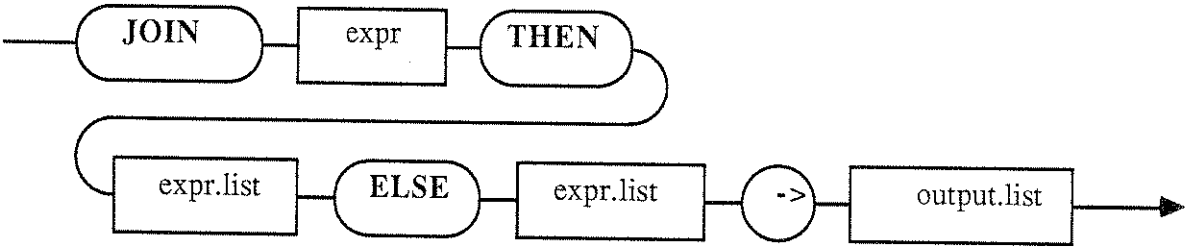


functional.definition

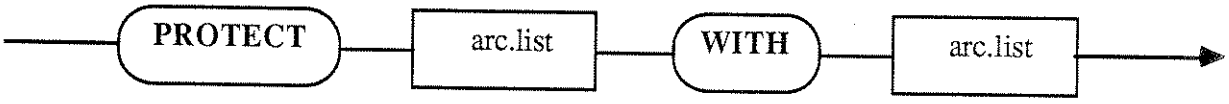
switch.statement



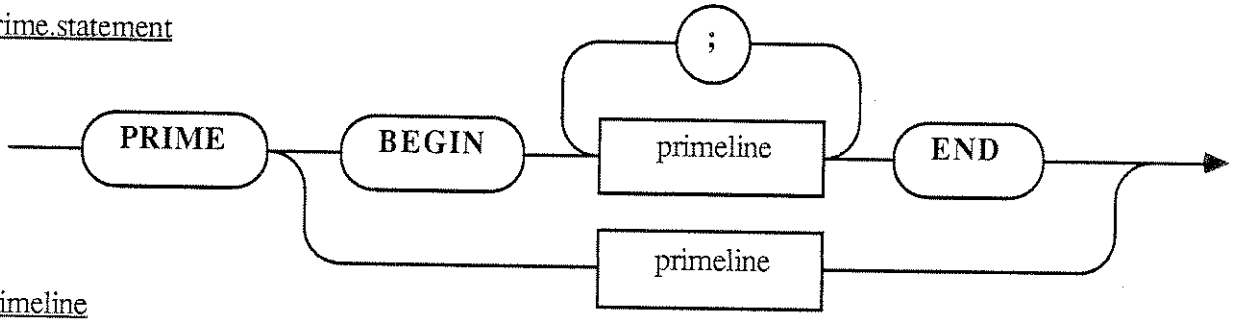
join.statement



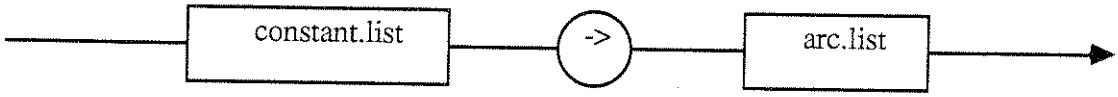
protect.statement



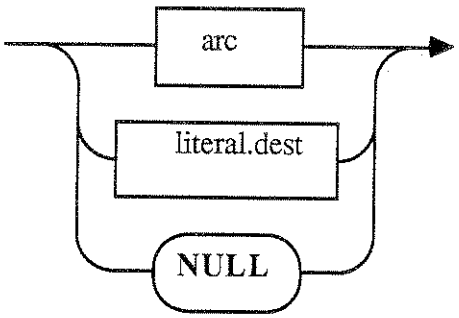
prime.statement



primeline

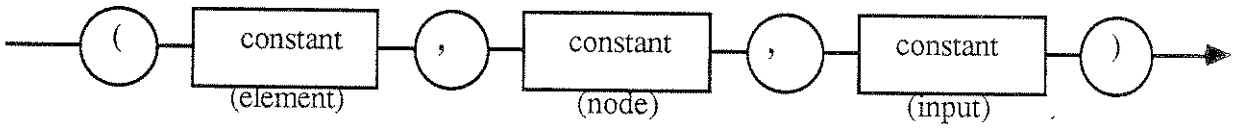


output

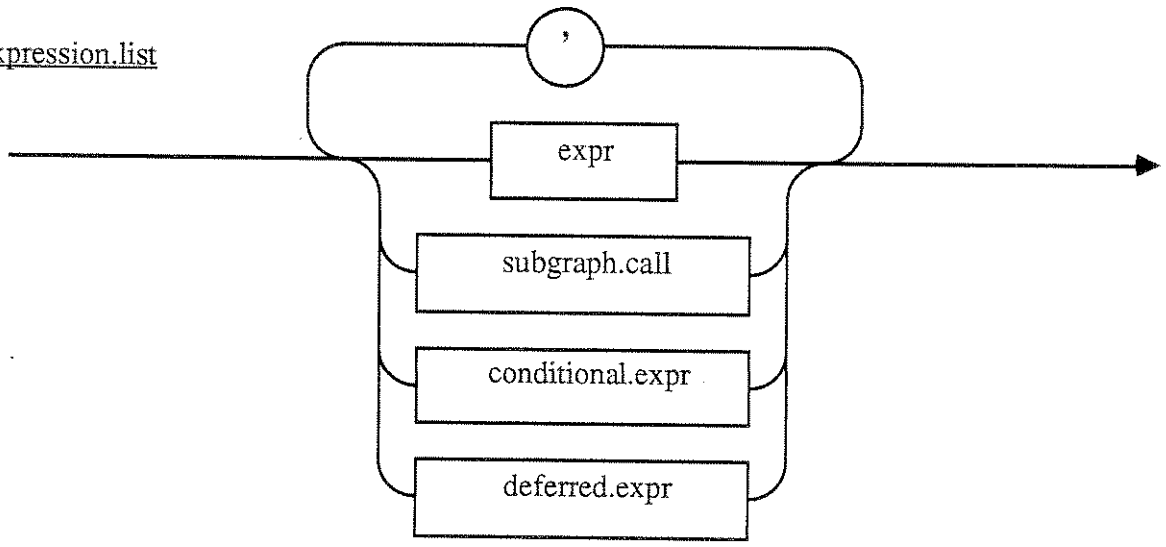


arc

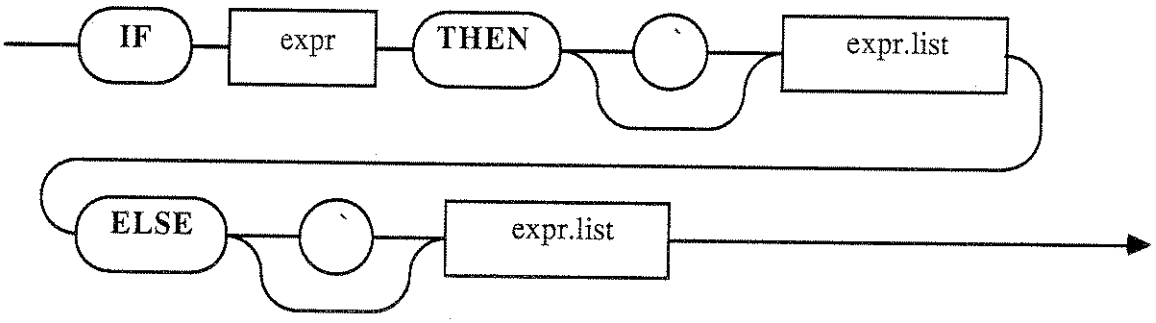
literal.dest



expression.list

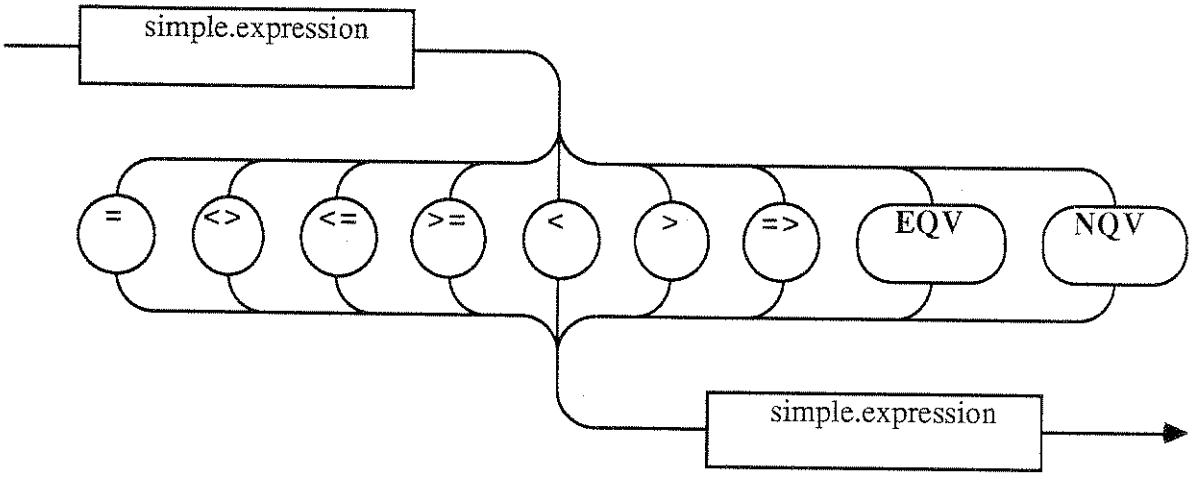


conditional.expr

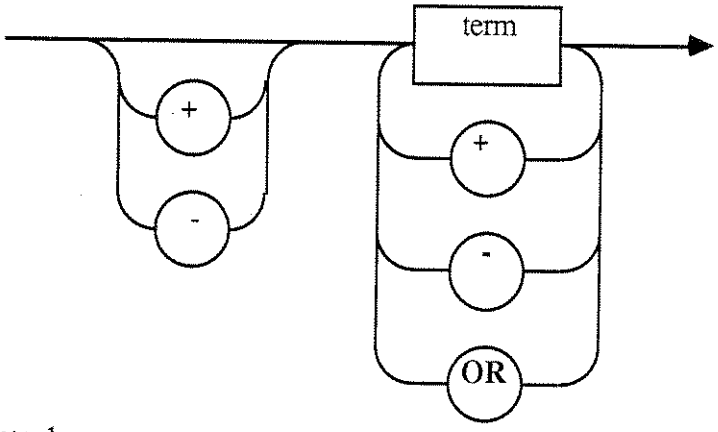


deferred.expr

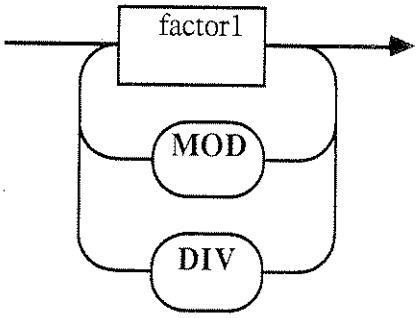
expression



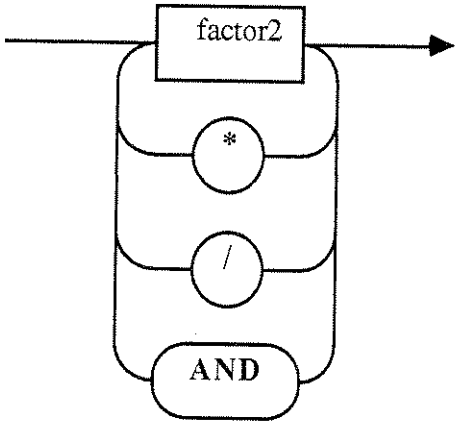
simple.expression



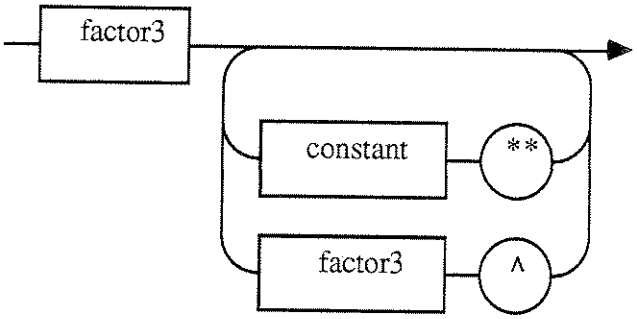
term



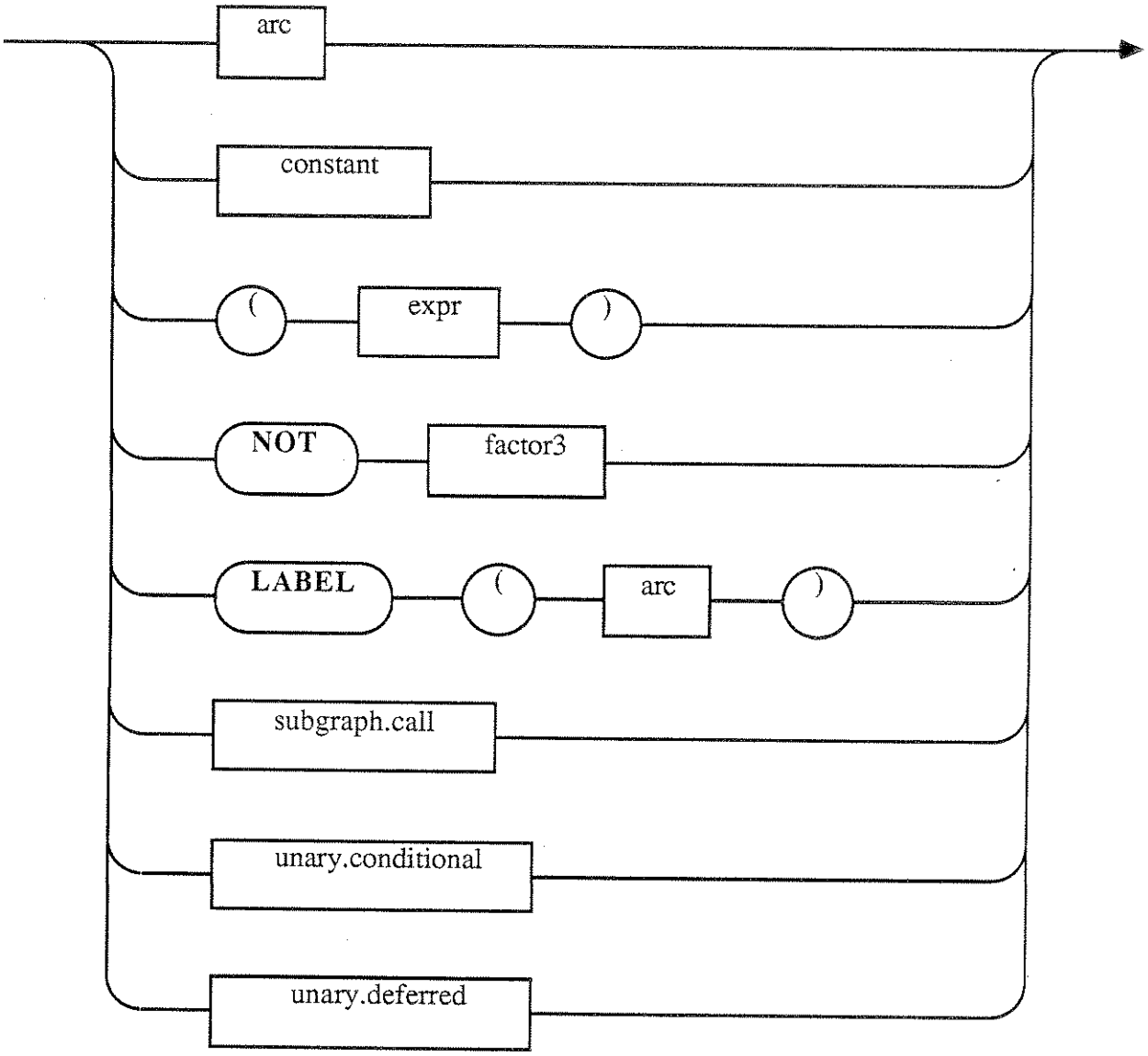
factor1



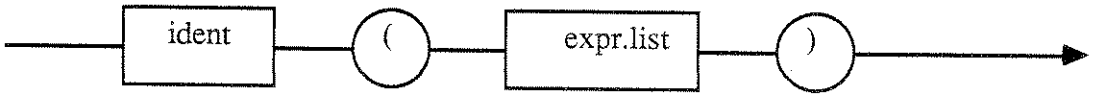
factor2



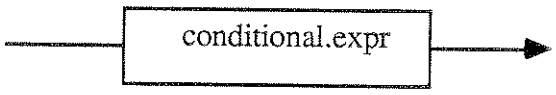
factor3



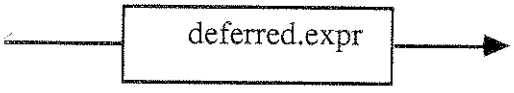
subgraph.call



unary.conditional



unary.deferred



References

[1] G. K. Egan, A Study of Data-flow: Its Application to Decentralised Control, Ph.D. Thesis, Dept. of Computer Science, University of Manchester, 1979.

[2] C. P. Richardson, Object Recognition Using a Data-flow machine: Algorithms for a Laser Range-Finder, M.Sc. dissertation, Dept. of Computer Science, Univerity of Manchester, 1979.

[3] R. N. Ibbet and P. C. Capon, The Development of the MU5 Computer System, CACM Vol 21, No 1, Jan 1978.

[4] K. S. Weng, Stream-oriented Computation in Recursive Data-flow Schemas, Technical memo No 68, Laboratory for Computer Science, Massachussetts Institute of Technology, Oct 1975.

[5] G. K. Egan, FLO: A decentralised Data-flow system, Internal Document, Dept, of Computer Science, University of Manchester, 1980.

[6] C. P. Richardson, Manipulator Control Using a Dataflow Machine, Ph.D. Thesis, Dept. of Computer Science, Univerity of Manchester, 1981.

[7] W. B. Ackerman, Data Flow Languages, Proceedings of the National Computer Conference, pp 1087-95, Vol 48, 1979.

[8] M. W. Rawling, Dataflow: An Implementation and Analysis, M.Eng Thesis, to be submitted, Dept. of Communication and Electronic Engineering, RMIT, 1987.