

SIMULATION OF NUMERICAL PETRI NETS USING
DATA-DRIVEN COMPUTER ARCHITECTURES

Report 2

Contract No. 63228

Dr G.K. Egan

Senior Lecturer, Digital Electronics and
Computing Systems

1. Suggested Second Report Tasks

The following tasks were suggested for the second phase of the study:

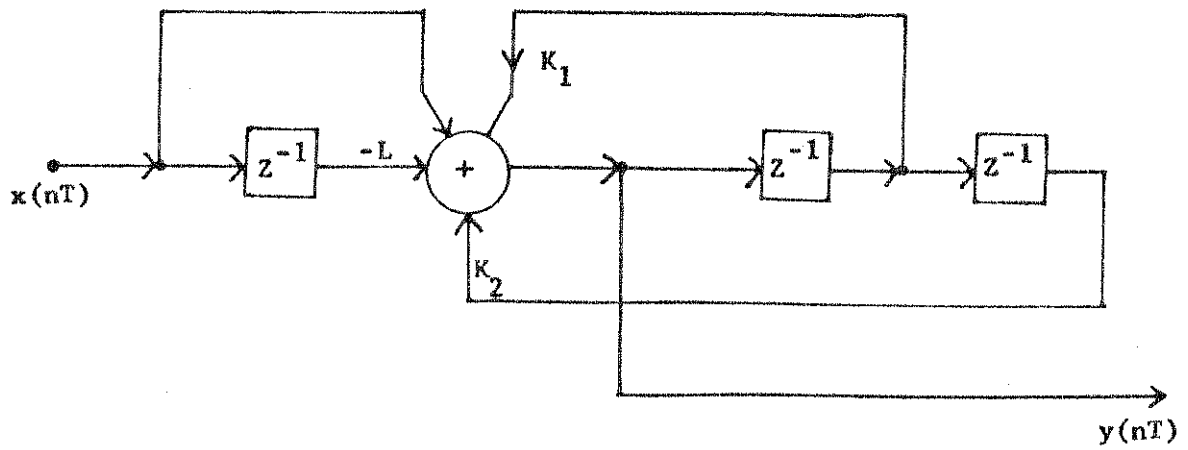
- 1) Propose a suitable graphical language for the description of NPNs in terms of connected sub-nets.
- 2) Establish a suitable format for the transmission of NPN descriptions from the Clayton Telecom VAX to the R.M.I.T. computing facilities.
- 3) Determine exploitable parallelism in a typical NPN. The T6 protocol nets were to be used for this determination.
- 4) Propose a data-driven architecture that might be appropriate to exploit the parallelism of NPN descriptions.

2. Languages

There are many languages for data-flow architectures and usually several for each architecture. Some seek to gain favour for data-flow architectures by emulating textual languages used on conventional Von-Neumann architectures [Whitelock]. Textual languages because they are intrinsically one dimensional tend to obscure any parallelism in the algorithm being described and worse still may by their sequential nature prevent the full expression of that parallelism. Textual language examples for data-flow architectures are usually illustrated by an informal graphical language.

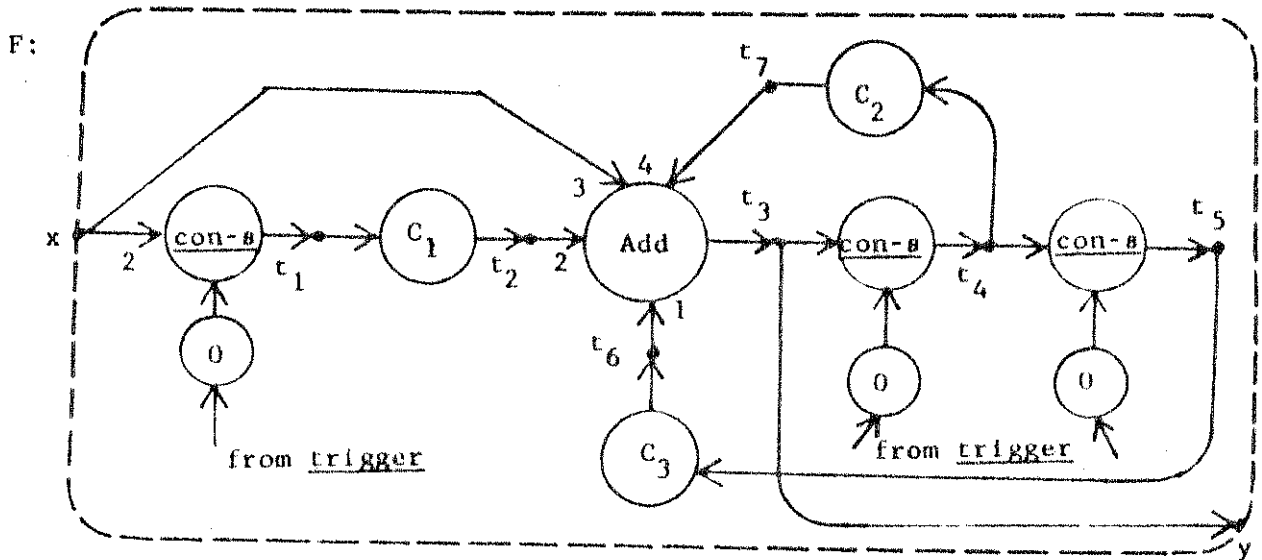
Graphical languages for data-flow architectures also come in many flavours and again there may be several used for any given data-flow architecture [Weng] [Dennis] [Rumbaugh] [Misunas]. Although graphical languages allow a fuller expression of algorithm parallelism few data-flow graphical languages have well defined notations for describing data-flow graphs as nested and connected sub-graphs.

Some examples of mixed textual and graphical languages are given below:



$$y(nT) = k_1 y((n-1)T) + k_2 y((n-2)T) + x(nT) - Lx((n-1)T)$$

Second Order Digital Filter [Weng]



```

c1 : module (x:int;y:int)
      (-L) * x -> y
    mend
c2: module (x:int;y:int)
      K1 * x -> y
    mend
c3: module (x:int;y:int)
      K2 * x -> y
    mend
Add:module (x1:int,x2:int,x3:int,x4:int;y:int)

```

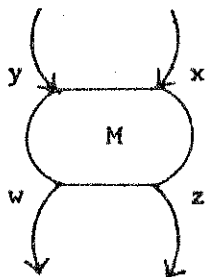
```

x1 + x2 + x3 + x4 -> y
mend
F: perform (x:st int; y:st int)
    con->s(0,x) -> t1; c'1(t1) -> t2;
    con->s(0,t3) -> t4; c'2(t4) -> t7;
    con->s(0,t4) -> t5; c'3(t5) -> t6;
    Add'(t6,t2,x,t7) -> t3;
    t3 -> y
pend

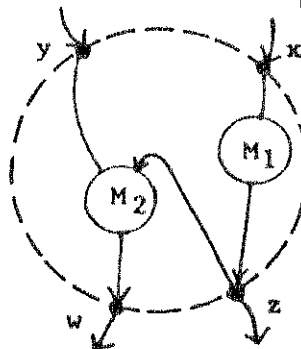
```

TDFL Textual Description of Filter

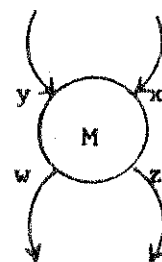
module_application actor
for M



the module M is substituted for
the actor



notation



```

M1: rmodule (x:st int; z:st int)
    if empty(x) then [] -> z
    else get(0) -> head, tail;
        M1(tail) -> more;
    con->s(2*head,more) -> z
    end

```

mend

```

M2: rmodule (y:st int,z:st int;w:st int)
    if empty (y) or empty(z)
    else get(y) -> y1,y2;
        get(z) -> z1,z2;
        M2(y2,z2) -> w1;
    con->s(y1+z1,w1) -> w
    end

```

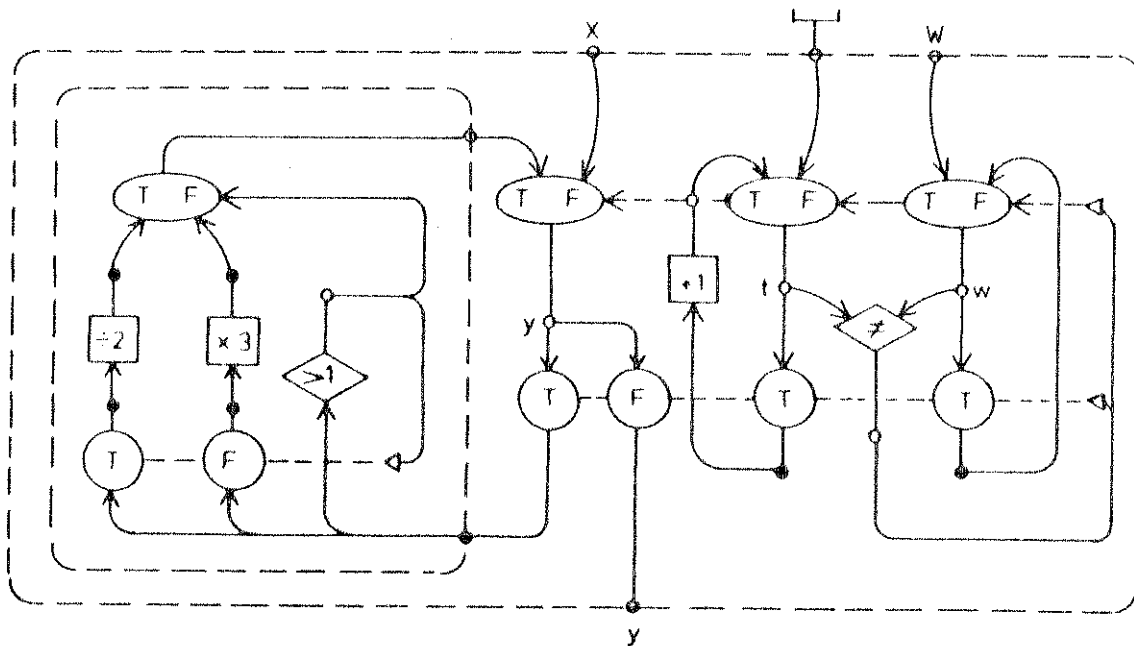
mend

```

M: module (x:st int, y:st int;z:st int,w:st int)
    M1(x;z);
    M2(y,z;w)
mend

```

Recursive TDFL Example and Graphical Representation [Weng]



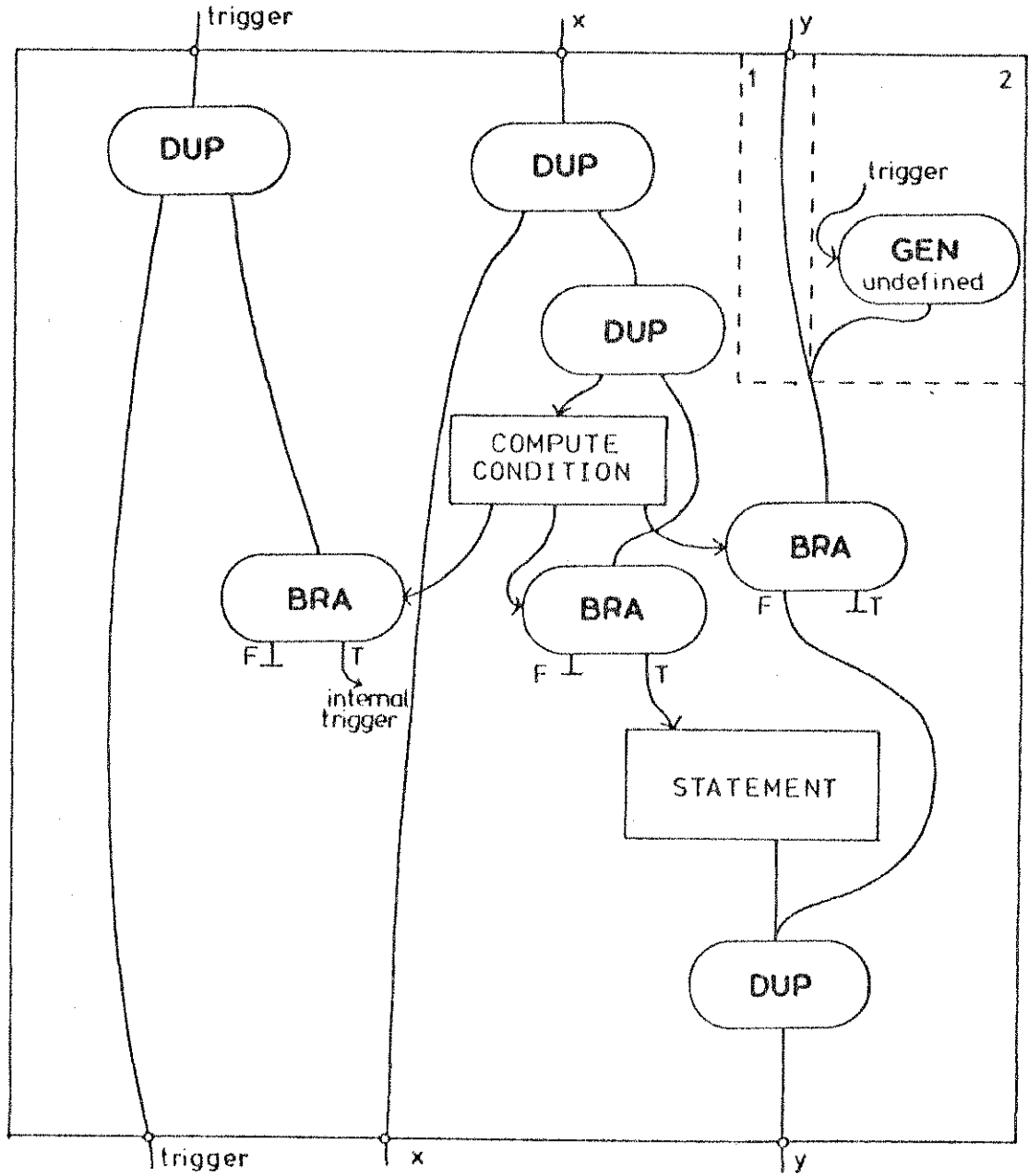
```

INPUT (W,X)
  y:=x; t:=0;
  WHILE t<>w DO
  BEGIN
    IF y>1 THEN
      y:=y div 2
    ELSE
      y:=y *3;
      t:=t+1;
    END
  END
OUTPUT y

```

Data Flow Schema and Program [Dennis]

Both the examples above are from the Massachusetts Institute of Technology Computation Structures Group.

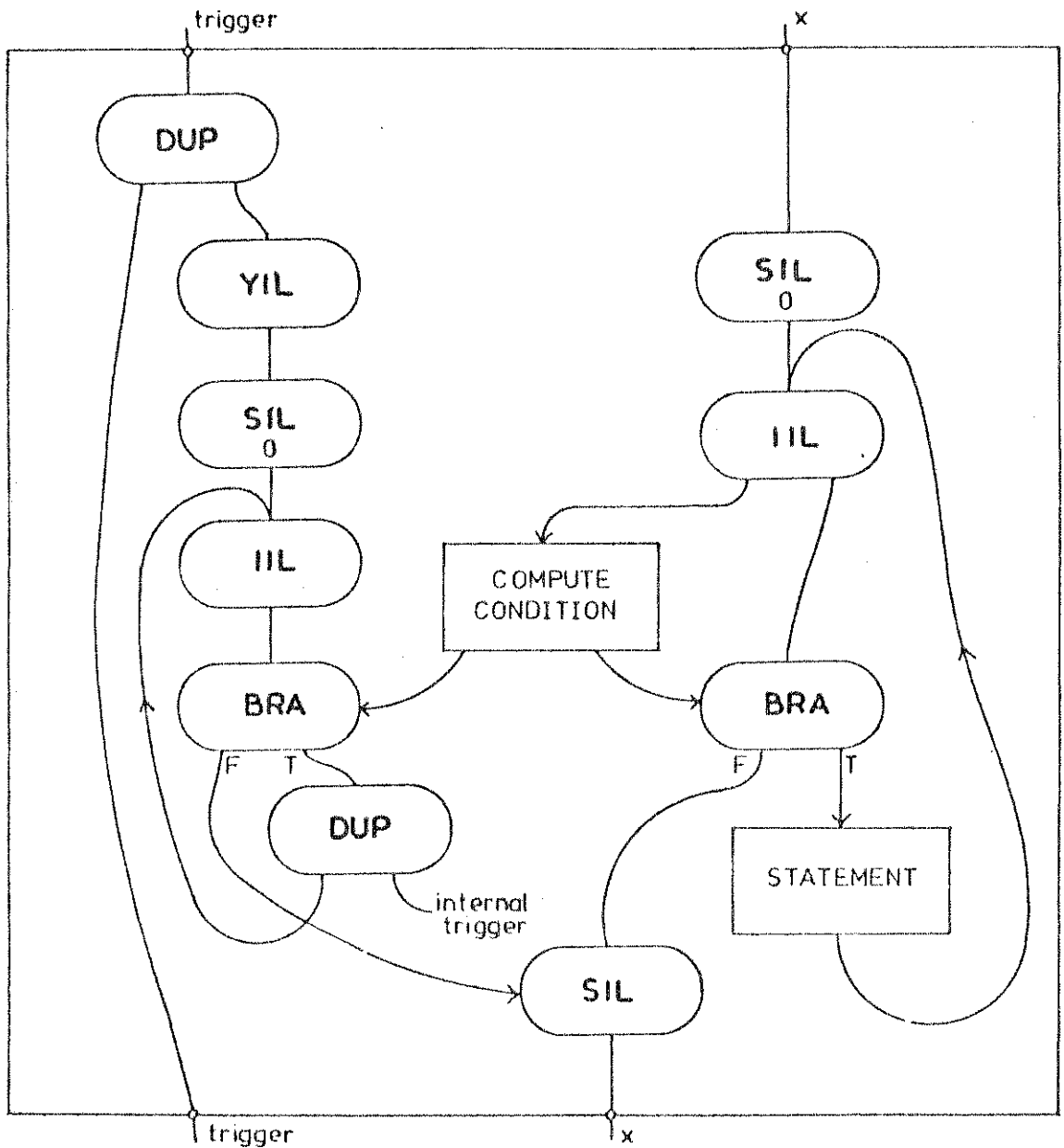


```

IF x>0 THEN
  BEGIN
    y:=y+2
  END;

```

Compiler Template for IF Statement [Whitelock]



```

WHILE x<1000 DO
  BEGIN
    x:=x*x
  END;

```

Compiler Template for While Loop [Whitelock]

The above examples are from the Manchester Data-Flow Machine Group.

Numerical Petri Net Report 2 G.K. Egan 1983

2.1 FLO Languages

There are four languages available to support the FLO system:

- 1) Data-Flow Language 1 (DL1): a block structure textual language [Richardson].
- 2) Newspeak: a Lisp like functional language [Wathanasin].
- 3) Graph: a graphical language [Walkington].
- 4) Intermediate Target Language (ITL): a low level textual language [Egan].

2.1.1 Data-Flow Language 1 (DL1)

DL1 was developed to provide language support for research into object recognition and manipulator control using the FLO system [Richardson]. It was regarded as a temporary expedient forced by the absence of suitable graphical editing devices which would have allowed the direct generation of data-flow graphs. The main aim in the development of DL1 was not to obscure the underlying graphs and sub-graphs that were being described; this aim was only partially achieved.

DL1 which has many similarities to Weng's TDFL is the current working language for the FLO system but it will be abandoned when Graph (Section 2.1.3) is fully commissioned.

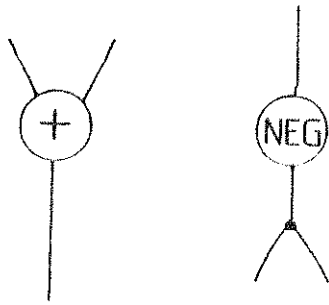
2.1.2 Newspeak

Newspeak was developed some time ago to explore the use of functional languages on data-flow architectures. It remains a valid tool for that research.

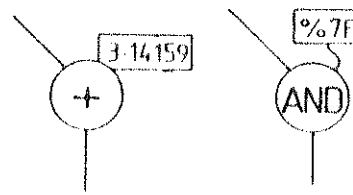
2.1.3 Graph

Graph is the graphical language for FLO [Egan][Richardson]. With the availability of appropriate graphical input devices at R.M.I.T. a full interactive graphical compiler is being commissioned [Walkington] as a third year design project in the Communication and Electronic Engineering Department.

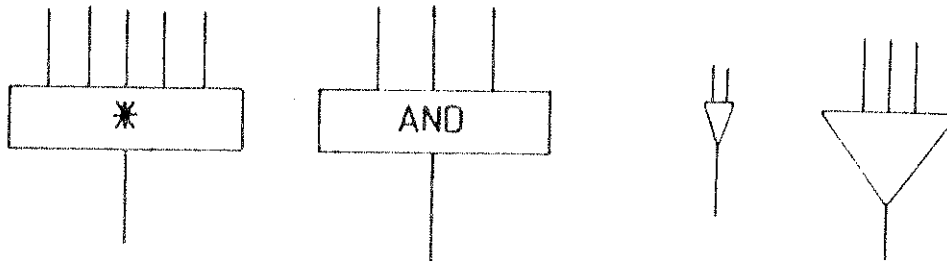
The Graph Compiler allows full interactive top down development of data-flow graphs as connected and nested sub-graphs.



Primitive Nodes

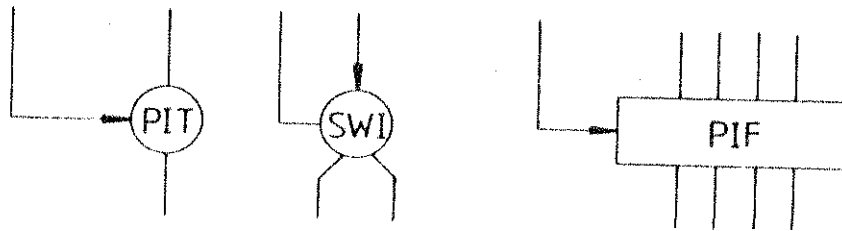


Nodes with Literal Data

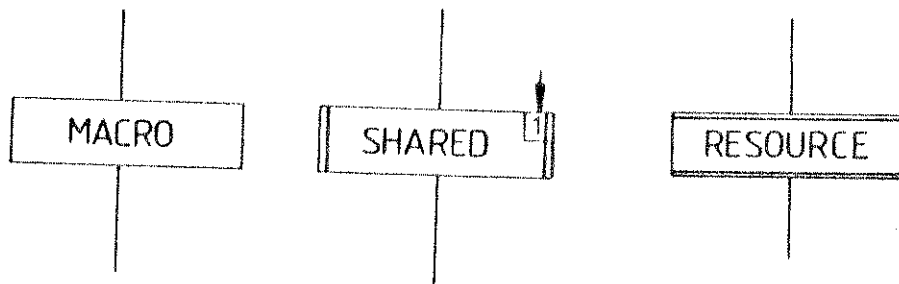


Balanced Trees

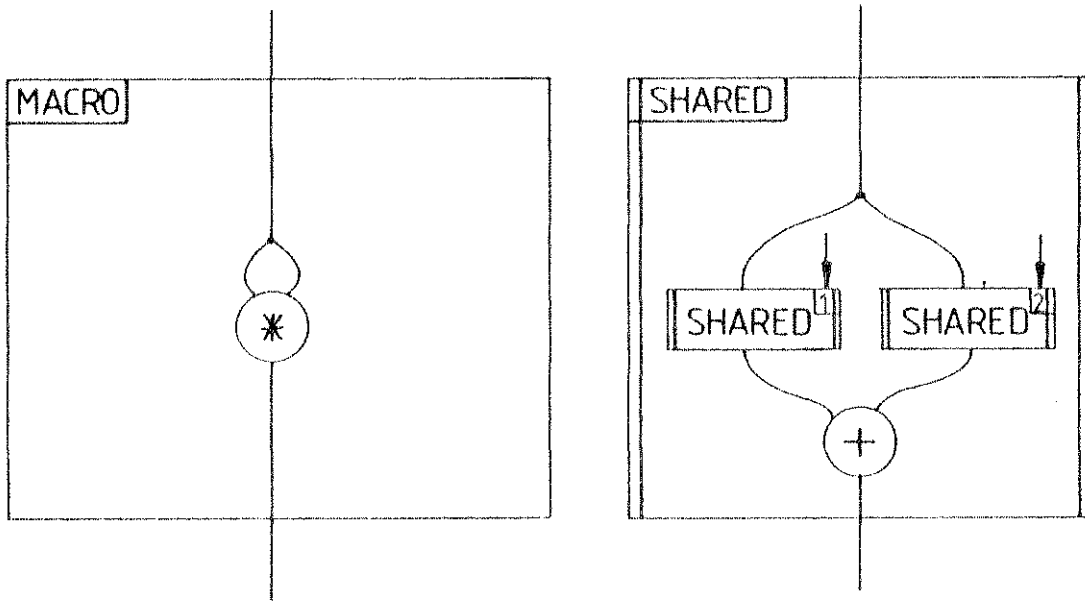
Merge Nodes



Control Functions

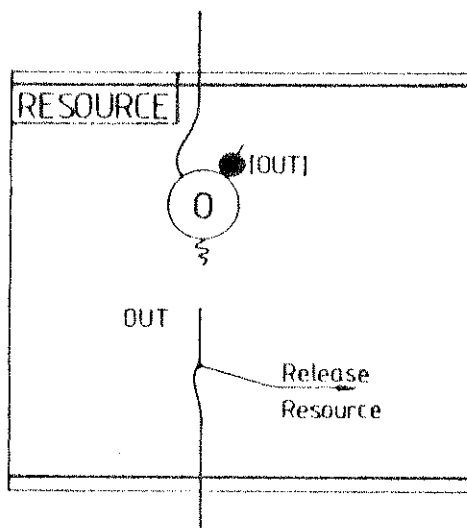


Sub-graph Invocation



Unshared Sub-graph

Shared Sub-graph



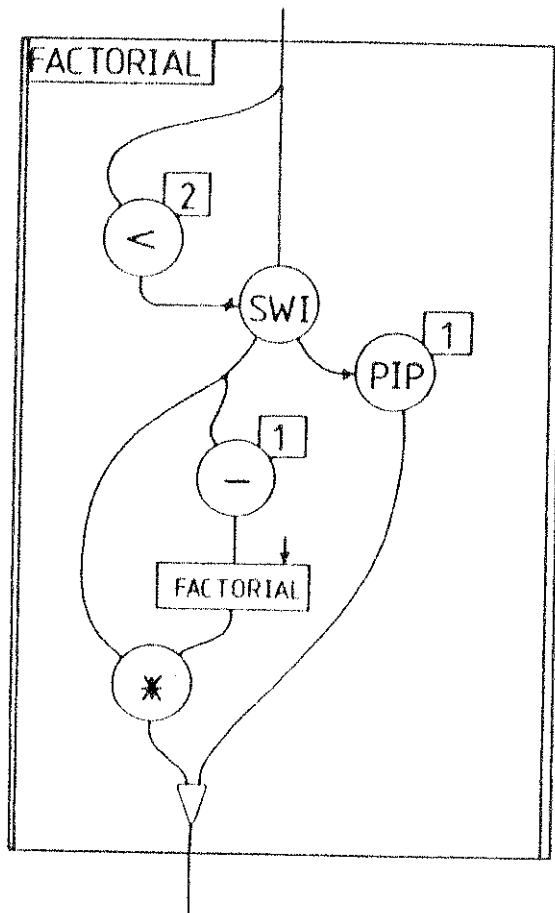
Resource Sub-graph

Sub-graph Definitions

```

shared subgraph factorial(i:integer)->(fac:integer);
begin
  if i<2 then
    i -> i1
  else
    i2;
  on i1 then
    l -> fact1;
  merge(fact1,i2*factorial(i2-1))
end;

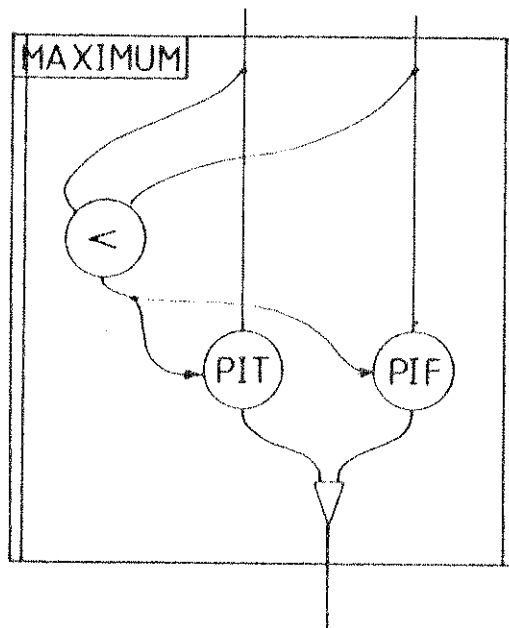
```



```

subgraph maximum(a,b:real)->(max:real);
begin
  if a>b then
    a
  else
    b -> max
end;

```



Examples of Sub-graph Usage in Dll and Graph

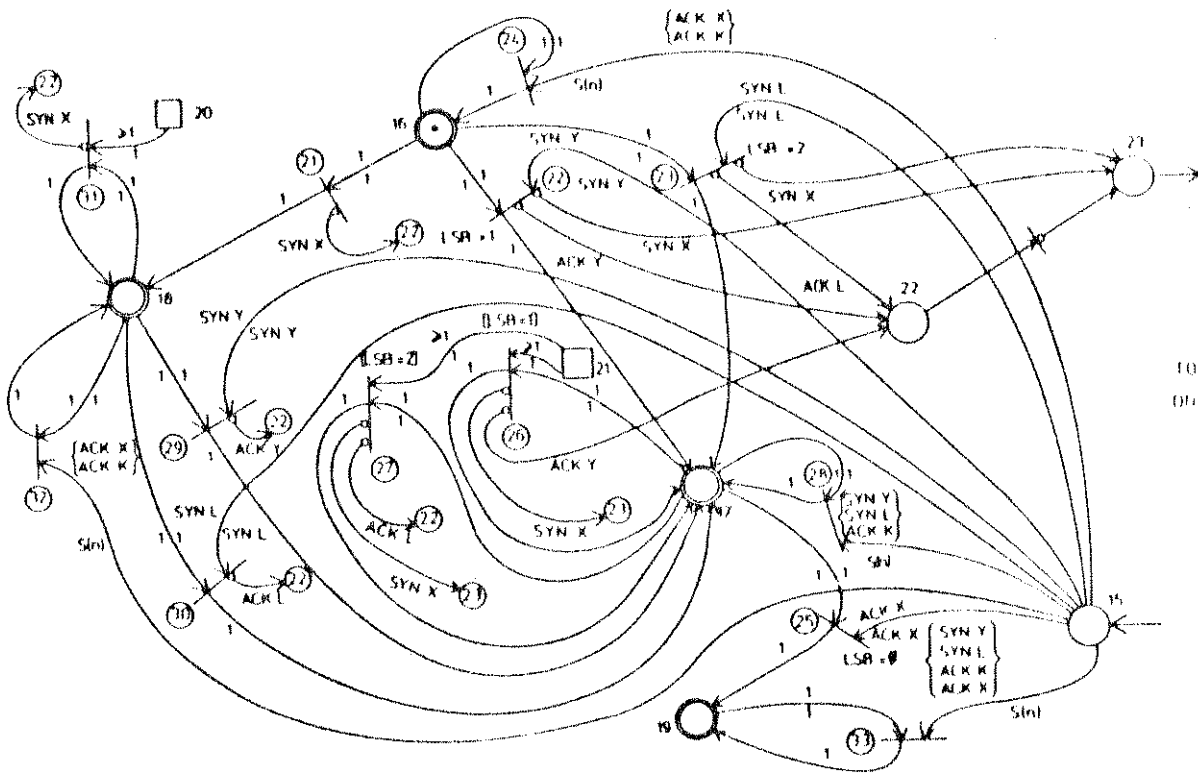
The examples and diagrams above are from Richardson's Ph.D. Thesis. It is allowable to use either the symbolic notation for node functions or the function mnemonic. The complete node function set is defined in the Appendix.

The Chill [Cain] suite of programs was considered as a base for the Graph compiler but while adequate for generating graphs it was not considered sufficiently well developed for editing graphs.

2.1.4 Intermediate Target Language (ITL)

ITL (Appendix) is a low level textual language which is used to describe data-flow graphs at the machine level. All FLO language compilers use ITL as their target language. ITL data-flow graph descriptions can be read directly by the FLO Simulator/ emulator and or may translated directly to the Target Binary Language (TBL) for FLO hardware.

The ITL is akin to assembly language and its direct use for



10
01

The CI nets [Mazurkiewicz] have much to recommend them. Arcs are drawn carefully either horizontally or vertically with transitions between these directions radiused to guide the eye. Scope of reference data is clearly defined in the textual language but not so clearly in the graphical language. Some care has been taken to make the net as planar as possible with the number of arc crossings minimised.

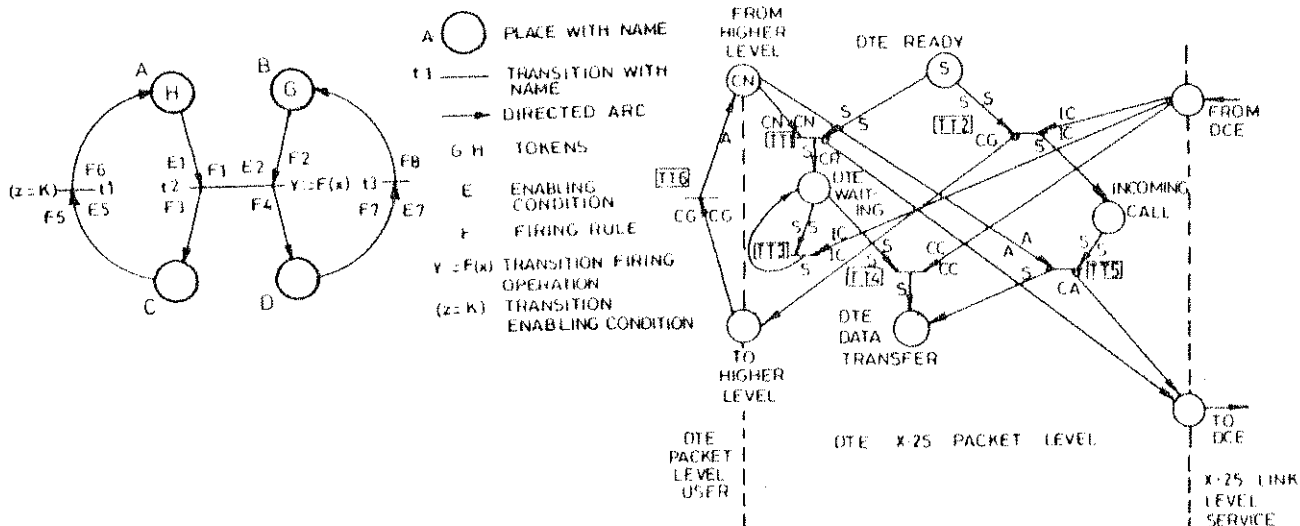
developing graphs, while possible, tends to be laborious and error prone. It is however very suitable for the storage of compiled data-flow graphs and their transmission between computer systems.

N 1	1	:	TF DUP	3 1 0	5 1 0
N 2	1	:	TF DUP	3 1 1	6 1 0
N 3	1	:	FF LT	4 1 0	
N 4	1	:	TF DUP	5 1 1	6 1 1
N 5	1	:	FF PIT	...	
N 6	1	:	FF PIF	...	

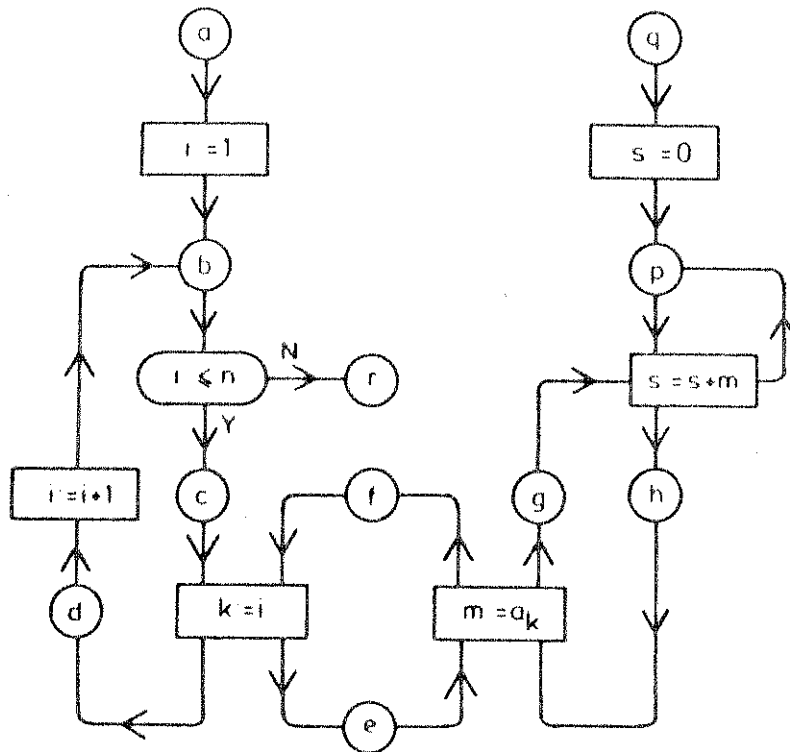
Maximum Sub-graph Expressed in ITL

2.2 NPN Languages

NPN notations also vary [Billington][Symons] although usually only in detail. As with data-flow the nets are often drawn free-hand with each author having his own style; with the absence of familiar constructions or visual patterns it can be difficult for others to quickly grasp the algorithm being described (Section 4.). An informal sub-net notation has been adopted but it does not include scope rules for reference data.



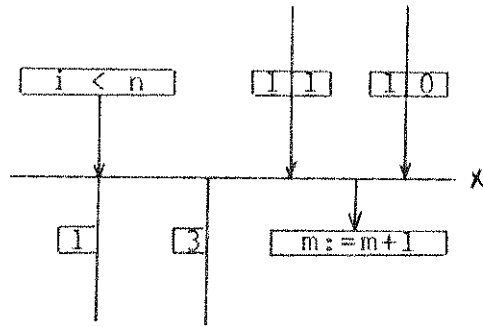
NPN Notation Variation



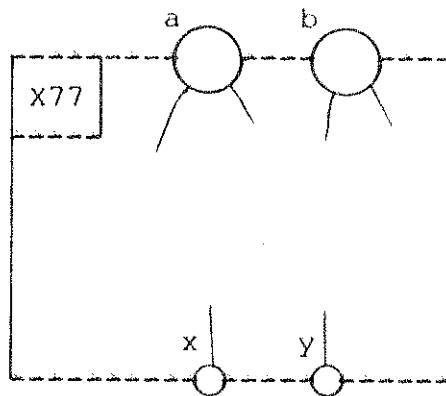
a,b,c,d,r: SCOPE i;
 e,f: SCOPE k;
 g,h: SCOPE m;
 p,q: SCOPE s;
 a: i:=1 and b;
 b: if i<=n then c else r;
 c,f: k:=i and d,e;
 d: i:=i+1 and b;
 e,h: m:=a_k and f,g;
 g,p: s:=s+m and h,p;
 q: s:=0 and p;

CL Program and Net

It is strongly suggested that the CL notation be adopted for NPN nets. If this is not acceptable then the role of reference data should be clarified with the following notation:



The sub-net notation could be of the following form:



The problem of expressing reference data scope is still unresolved.

3. Transfer of NPN Descriptions

The NPN Dialogue Language [Billington] is adequate for transmission of net descriptions. The Dialogue Language should be extended to include sub-net declarations.

```
S x77 ; formal input place names ; formal dummy output places;
...
...      sub-net places, transitions and arcs
...
F
```

The 1200 baud line used by the CHILL project team can be used for transmission.

4. Exploitable Parallelism in NPNs

After several frustrating attempts to encode the T6 protocol

from the NPN diagrams provided the exercise was abandoned. Inspection of this protocol however suggests a low parallelism (<10). Successful transmission of a machine readable version of the protocol will confirm or refute this estimate.

5. Processing Element Architecture

A Masters student [Rawling] supporting the project is currently commissioning a data-flow processing element based on two Motorola 68000 16/32 bit processors. The element architecture is similar to the fast processing element structure described in Report 1 Part 2. This element with the appropriate interpreter code is expected to satisfactorily exploit NPN net parallelism.

A fourth year student has investigated the implementation of the processor elements FIFO structures in VLSI. The preliminary designs are now at AWA for processing. This effort is part of a longer term aim to implement major sections of the architecture and associated communication structures in silicon.

The FLO simulator/emulator has been extended to cater for the fast processing element internal structure and allow time tagging of tokens. The time tags allow comparison of achieved and available parallelism in nets and graphs.

6. Closing Comments

Although progress has been made the project is still greatly constrained by manpower. FLO simulator code for full NPN as stated in Report 1 will take 3 months to generate. Simple petri nets may be interpreted by existing software but the translation from net to graph is still clumsy. Major changes instigated by the author in the digital teaching within the Communication and Electronic Engineering Department in 1981 will bear fruit in 1984 with the availability of design students with appropriate skills to support this and other research.

References and Bibliography

[Billington] Billington J. and Gaylard N., 'NPN Analyser Users' Manual', Telecom Australia, 1981.

Billington J. et al., 'Modelling and Analysis of Communication Protocols Part 1 and 2', Proceedings of IREECON'81 International, Melbourne, 1981.

[Cain] Cain G.J. et al., 'Computer-Aided Chill Code Generation', Report 10, Telecom Contract 53901, March 1983.

[Dennis] Dennis J.B., 'A Preliminary Architecture for a Basic Data-Flow Processor', Computation Structures Group Memo 102, Massachusetts Institute of Technology, August 1974.

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

G.K. Egan, 'FLO: A Decentralised Data-flow System Part 2', Internal document, Dept. of Computer Science, University of Manchester, Jan. 1980.

G.K. Egan, 'A Decentralised Computing System Based on Data-Flow', Proceedings of the IECI'80 Conference, March 1980.

G.K. Egan, 'A Data-Flow Computing System for Decentralised Control and Advanced Automata Applications', Proceedings of IREECON'81 International, AUG. 1981.

G.K. Egan and C.P. Richardson, 'Object Recognition Using a Data-Flow Computing System', Microprocessing and Microprogramming 7, North-Holland, 1981.

[Mazurkiewicz] Mazurkiewicz A., 'Invariants of Concurrent Programs', IFIP INFOPOL'76, North Holland, 1977.

[Misunas] Misunas D.P., 'Deadlock Avoidance in a Data-Flow Architecture', Computation Structures Group Memo 116, Massachusetts Institute of Technology, February 1975.

[Rawling] M.W. Rawling and E.A. Zuk, 'Data-Flow Processing Element', Design 3 Manual, Department of Communication and

Electronic Engineering, Royal Melbourne Institute of Technology, 1982.

[Richardson(1)] C.P. Richardson, 'Object Recognition using a Dataflow Machine: Algorithms for a Laser Range-finder', M.Sc. dissertation, Department of Computer Science, University of Manchester, 1979.

[Richardson(2)] C.P. Richardson, 'Manipulator Control Using a Data-Flow Computing System', Ph.D. Thesis, Dept. of Computer Science, University of Manchester, 1981.

[Rumbaugh] Rumbaugh J. 'A Data Flow Multiprocessor', IEEE Transactions on Computers', February 1977.

[Symons] Symons F.J.W., 'The Application of Petri Nets and Numerical Petri Nets', Research Laboratories Report 7520, Telecom Australia, 1982.

[Walkington] Walkington M., 'A Data-Flow Graphical Compiler', Design 2 Report, Department of Communication and Electronic Engineering, R.M.I.T., to be published.

[Wathanasin] S. Wathanasin, 'Proposed Language for the Data-Flow Multiprocessor', Internal document, Dept. of Computer Science, University of Manchester, 1978.

[Weng] K.S. Weng, 'Stream-oriented Computation in Recursive Data-flow Schemas', Technical memo 68, Laboratory for Computer Science, Massachusetts Institute of Technology, Oct. 1975.

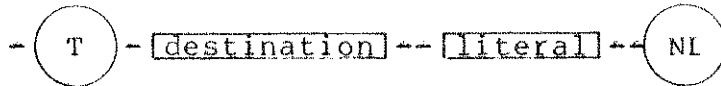
[Whitelock] Whitelock P.J., 'A Conventional Language for Data-Flow Computing', M.Sc. Thesis, Department of Computer Science, University of Manchester, October 1978.

APPENDIX

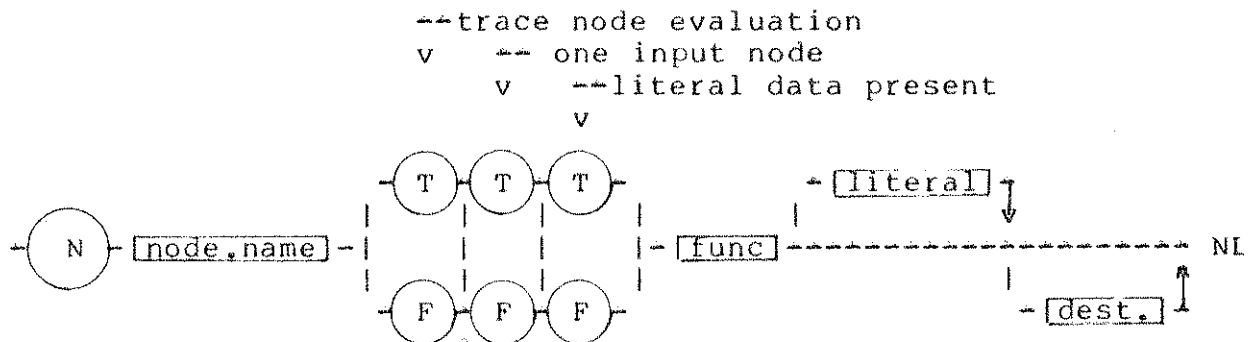
1. Intermediate Target Language (ITL)

Graphical and textual language translators generate Intermediate Target Language (ITL). ITL is accepted directly by the FLO simulator/emulator and is translated to the Target Binary Language (TBL) (Section 2.) for the FLO machine.

1.1 Tokens



1.2 Nodes



1.3 Token and Node Fields

node name

- [element] -- [elem.node] -

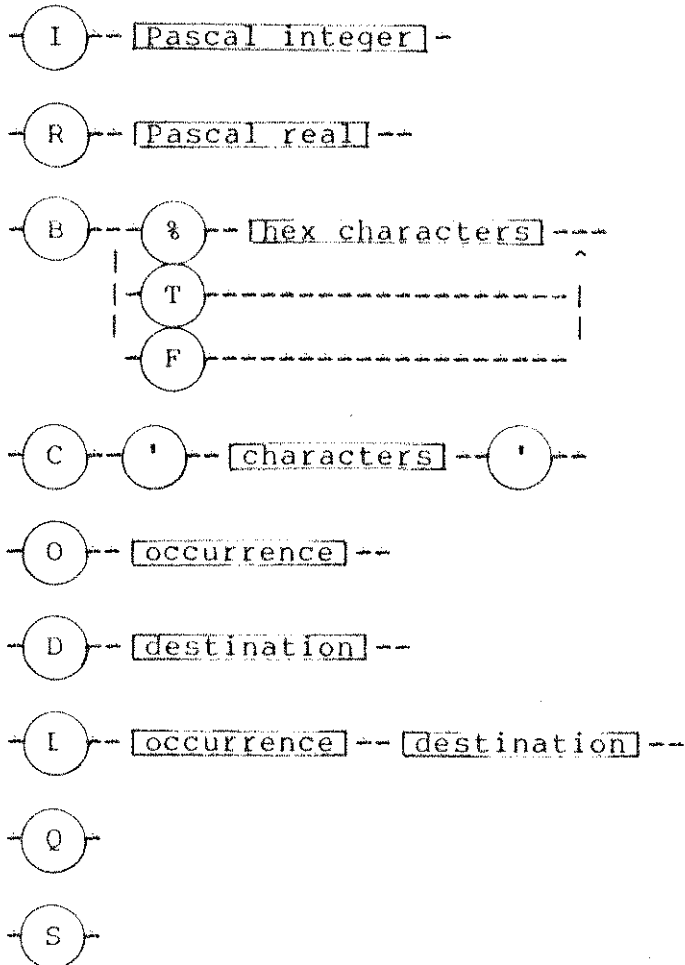
destination

- [node.name] -- [input.point] -

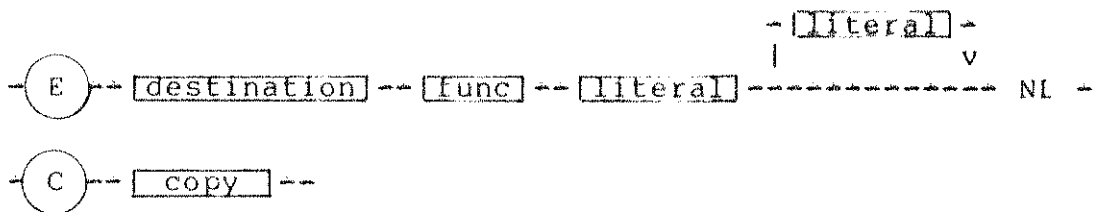
function

ADD SUB MUL DVD DIV MOD EXP NEG ABS SIN COS ATN LNE SQT
 AND IOR IMP EQV NOV TSB STB CLB NOT
 CPT EQ LT GT LE GE NE
 SUC PRE
 [] [] []
 ORD CHR RND TRN
 S
 DUP
 PRS PIT PIF PIP SWI
 STD YLC STC
 A R E
 D

literal



The following are not valid literals but are the representations used by FLO when directing tokens of these types to output files.



input point

{0,1}

element

{1..65535}

element node

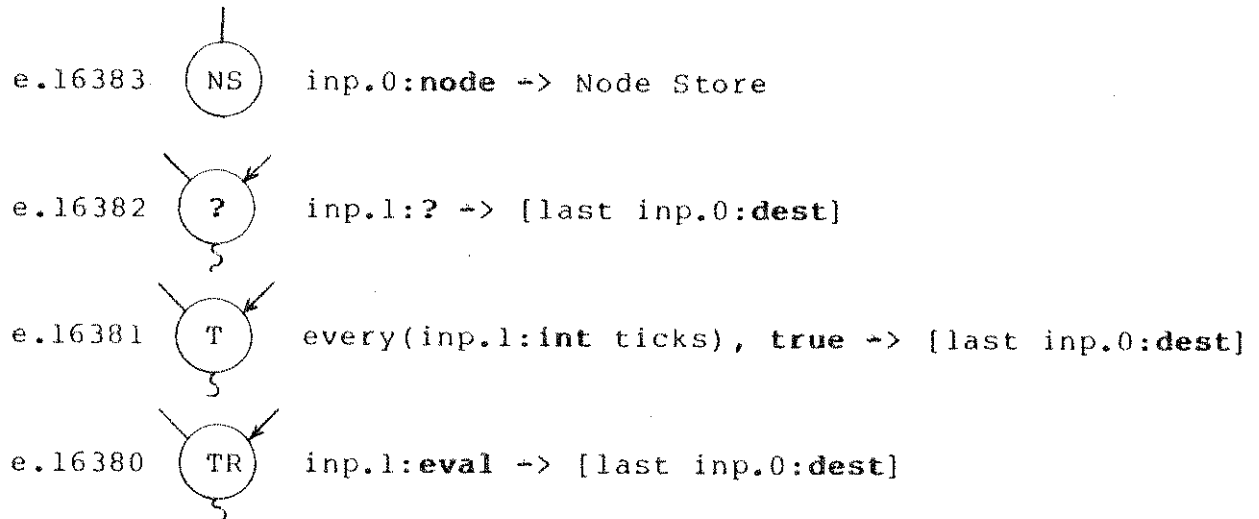
{1..16383}

occurrence

{1..255}

1.4 Reserved Destinations

Element-nodes 16255 to 16383 reserved for FLO system usage. The following node-names are defined in all elements:



The node subrange e.16263-16270 is reserved for input and e.16255-16262 is reserved for output.

The following node-names are associated with FLO input files:

if present(inp.1), 'input':any -> [last inp0:dest]

1.16255	INPUT
2.16255	FLOINP1
3.16255	FLOINP2
4.16255	FLOINP3
5.16255	FLOINP4
6.16255	FLOINP5
7.16255	FLOINP6
8.16255	FLOINP7

Tokens input to FLO from files or INPUT must conform to the ITL syntax for literals.

e.g. R 1.534

The following node-names are associated with FLO output files:

inp.1:any -> [last inp.0dest], inp.1:any -> 'output'

1.16263	OUTPUT
2.16263	FLOOUT1
3.16263	FLOOUT2
4.16263	FLOOUT3
5.16263	FLOOUT4
6.16263	FLOOUT5
7.16263	FLOOUT6
8.16263	FLOOUT7

The FLO simulator/ emulator models a laser rangefinder and a simple object space containing a cone, cylinder and sphere. The laser may be vectored using output nodes LTHETA:int and LPHI:int. The range to an object or the space background is obtained from the input node RANGE:real.

9.16263	LTHETA
9.16264	LPHI
9.16255	LRange

The simulator also models a six axis manipulator. The output nodes associated with the manipulator all accept real tokens.

10.16263	WAIST
11.16263	SHOULDER
12.16263	ELBOW
13.16263	TWIST
14.16263	BEND
15.16263	SWIVEL
16.16263	GRASP

The input node-names associated with manipulator status are:

10.16255	WAIST
11.16255	SHOULDER
12.16255	ELBOW
13.16255	TWIST
14.16255	BEND
15.16255	SWIVEL
16.16255	GRASP

Null FLO element

Element 0 is the null FLO element.

2. Target Binary Language (TBL)

This section describes the extended token and node formats for the FLO machine. Individual node-functions are illustrated using a simple single-assignment textual language.

2.1 General Token and Node Formats

2.1.1 Tokens

Normal Form

```
[ destination field | token data field ]
```

Shared Sub-Graph Form

```
[ destination field | copy-field | token data field ]
```

Data tokens are of the above general forms; the second form is only used by tokens involved in shared sub-graph usage.

2.1.2 Nodes

Diadic Nodes

```
[ function fields | destination fields ]
```

Monadic Nodes

```
[ function fields | destination fields ]
```

Literal Form

```
[ function fields | token data fields | destination fields ]
```

Node-descriptions are of the above general form. Normally there is only one destination field. Data fields are usually reserved for literal node-function arguments with the input-point of the literal being implied by the arriving token.

2.2 Node and Token Field Definitions

The number below each field is the width of the field in bits.

2.2.1 Function Fields

```
[ trace | one-input | data-present | function-name ]  
      1         1         1         13
```

trace	set if node operands and node-name are to be sent to the TR node.
one-input	set if the node has one input arc.
data-present	set if the data field is present.
function-name	node-function name.

2.2.2 Destination Fields

element	copy-present	input-point	element-node
16	1	1	14

element processing-element containing destination
node-description. Range is 0-65535.
copy-present copy fields follow the destination.
input-point destination node input-point.
element-node byte address of the node-description.

2.2.3 Token Data Fields

General Form

structure	type	length	data
1	4	11	16->

structure when set indicates that the token carries
a structured object.
type the token data type.
length the length of the token data field.
data the token data.

A brief description of the more unusual token types is given below:

Don't-know (?)

The first two bytes of the data field contains the reason for the token type becoming unknown. The last four bytes contain the name of the node at which the type became unknown.

Node

The first two bytes of the data field contains the element-node. The rest of the data field contains the node description.

Link-destination

This token type is generated by return-entry nodes and used by exit nodes. The data field carries the occurrence number of the shared sub-graph in the first two bytes, and the destination to which the exit node should send result-tokens in the subsequent four bytes.

Evaluation

This token type is generated when the trace bit is set in a nodes function fields. The first four bytes of the data field contain the nodes name, the next two bytes the function fields, and the following bytes the operand token data fields.

2.3.4 Copy Field

copy

16

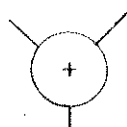
copy the copy field is used to distinguish between tokens involved in different invocations of the same sub-graph.

2.3 Primitive Nodes

2.3.1 Arithmetic

Diadic Nodes

*	0	0	function	destination
1	1	1	13	32

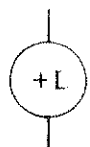


+	ADD	inp.0:int,real + inp.1:int,real -> out.0:int,real
-	SUB	inp.0:int,real - inp.1:int,real -> out.0:int,real
*	MUL	inp.0:int,real * inp.1:int,real -> out.0:int,real
/	DVD	inp.0:int,real / inp.1:int,real -> out.0:int,real
div	DIV	inp.0:int div inp.1:int -> out.0:int
mod	MOD	inp.0:int mod inp.1:int -> out.0:int
^	EXP	inp.0:int,real ^ inp.1:int,real -> out.0:int,real

literal form

*	1	1	function	literal-token-data	destination
1	1	1	13	16 ->	32

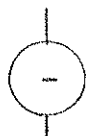
Function names as above. e.g.



inp.0:int,real + literal:int,real -> out.0:int,real

Monadic nodes


*	1	0	function	destination
1	1	1	13	32

	NEG	- inp.0:int,real -> out.0:int,real
abs	ABS	absolute(inp.0:int,real) -> out.0:int,real
sin	SIN	sine(inp.0:int,real) -> out.0:real
cos	COS	cosine(inp.0:int,real) -> out.0:real
atn	ATN	arc.tangent(inp.0:int,real) -> out.0:real
ln	LNE	logarithm(inp.0:int,real) -> out.0:real
sqrt	SQT	square.root(inp.0:int,real) -> out.0:real

2.3.2 Logical and Bit-manipulation

Diadic Nodes

*	0	0	function	destination
1	1	1	13	32

	AND	inp.0:bits and inp.1:bits -> out.0:bits
V	IOR	inp.0:bits or inp.1:bits -> out.0:bits
D	IMP	inp.0:bits implies inp.1:bits -> out.0:bits
≡	EQV	inp.0:bits equivalent inp.1:bits -> out.0:bits
≠	NQV	- inp.0:bits equivalent inp.1:bits -> out.0:bits
tsb	TSB	test.bit[inp.0:int] of inp.1:bits -> out.0:bits
stb	STB	set.bit[inp.0:int] of inp.1:bits -> out.0:bits
clb	CIB	clear.bit[inp.0:int] of inp.1:bits -> out.0:bits

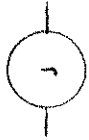
literal form

*	1	1	function	literal-token-data	destination
1	1	1	13	16 ->	32

Function names as above.

Monadic nodes

*	1	0	NOT	destination
1	1	1	13	32



NOT - inp.0:bits -> out.0

2.3.3 Relational

Diadic Nodes

*	0	0	function	destination
1	1	1	13	32



CPT compare.type(inp.0,inp.1) -> out.0:bits

<> NE inp.0:int,real,char <> inp.1 -> out.0:bits

= EQ inp.0:int,real,char = inp.1 -> out.0:bits

< LT inp.0:int,real,char < inp.1 -> out.0:bits

> GT inp.0:int,real,char > inp.1 -> out.0:bits

<= LE inp.0:int,real,char <= inp.1 -> out.0:bits

>= GE inp.0:int,real,char >= inp.1 -> out.0:bits

literal form


*	1	1	function	literal-token-data	destination
1	1	1	13	16 ->	32

Function names as above.

2.3.4 Sequence Position

Monadic nodes

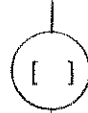
	*	1	0	function	destination
	1	1	1	1	32

 suc	SUC	successor(inp.0:char,int) -> out.0:char,int
pre	PRE	predecessor(inp.0:char,int) -> out.0:char,int

2.3.5 Stream

Monadic nodes

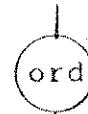
	*	1	0	function	destination
	1	1	1	13	32

 []	BRA	inp.0:any] -> out.0:stream.any
] [UNB	inp.0:stream.any except] -> out.0:any

2.3.6 Explicit Type Conversion

Monadic nodes

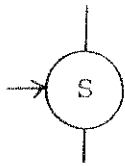
	*	1	0	function	destination
	1	1	1	13	32

 ord	ORD	ordinal(inp.0:int,char) -> out.0:int
chr	CHR	character(inp.0:int,char) -> out.0:char
rnd	RND	round(inp.0:int,real) -> out.0:int
trn	TRN	truncate(inp.0:int,real) -> out.0:int

2.3.7 Storage

Monadic node

*	1	0	STO	destination
1	1	1	13	32



S on inp.1, last inp.0 → out.0

2.3.8 Duplicate

Monadic node

*	1	0	DUP	dest.0	dest.1
1	1	1	13	32	32



DUP inp.0 → out.0, out.1

2.3.9 Control

Diadic Nodes

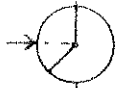
*	0	0	function	destination
1	1	1	13	32



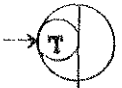
PRS if present(inp.0,inp.1) then true → out.0:bits



PIT if inp.1:bits then inp.0 → out.0

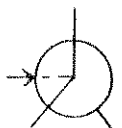


PIF if ¬ inp.1:bits then inp.0 → out.0



PIP if present(inp.1) then inp.0 → out.0

*	0	0	SWI	dest.0(false)	dest.1(true)
1	1	1	13	32	32



SWI if inp.1:bits then inp.0 → out.1 else out.0

Switching and passing of tokens is conditional on the least significant bit of the bit-string token on input-point-1. The internal convention adopted is all bits set for **true** and all bits clear for **false**.

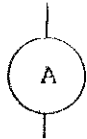
literal form

*	1	1	function	literal-token-data	destination(s)
1	1	1	13	16 ->	32-48

Function names as above.

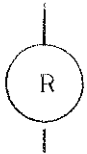
2.3.10 Shared Sub-graphs

*	1	1	A	occurrence	dest.
1	1	1	13	16	32



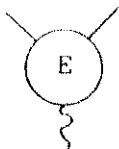
A inp.0 (with new copy) -> out.0

*	1	1	R	link-destination	dest.
1	1	1	13	64	32



R on inp.1, literal (new copy) -> out.0:link.dest

*	0	0	E		
1	1	1	13		



E inp.0 (with new copy) -> [inp.1:link.dest]

Argument tokens for the sub-graph are provided by arg-entry nodes. Return-entry nodes provide the destinations to which exit nodes send sub-graph result tokens.

On entering a sub-graph the token's copy number is computed as:

$$\text{newcopy} = (\text{oldcopy} * \text{maxoccurrence}) + \text{occurrence}$$

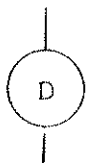
On exiting the copy number is computed as:

$$\text{newcopy} = (\text{oldcopy} - \text{occurrence}) \text{ div } \text{maxoccurrence}$$

Maxoccurrence, in the initial implementation, is set to 250 at graph-level 0 and 8 at subsequent levels. If the computed copy number is not zero then the copy bit is set in the token's destination and the copy number is appended.

2.3.11 Token Structure

*	1	0	D	destination
1	1	1	13	32

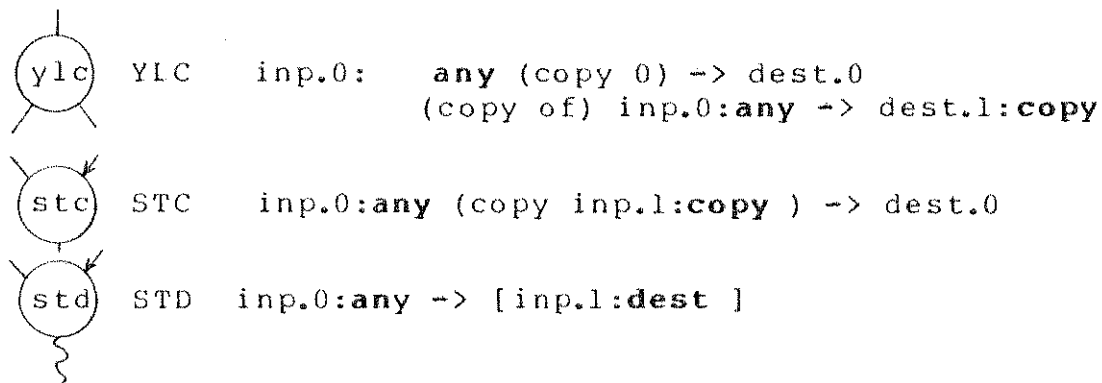


D decode(inp.0) -> out.0:stream.any

The fields of the input token are returned as a stream of tokens. A specific use of this node is to decode ? tokens. The inverse function, which may be used when forming graphs, has not been implemented.

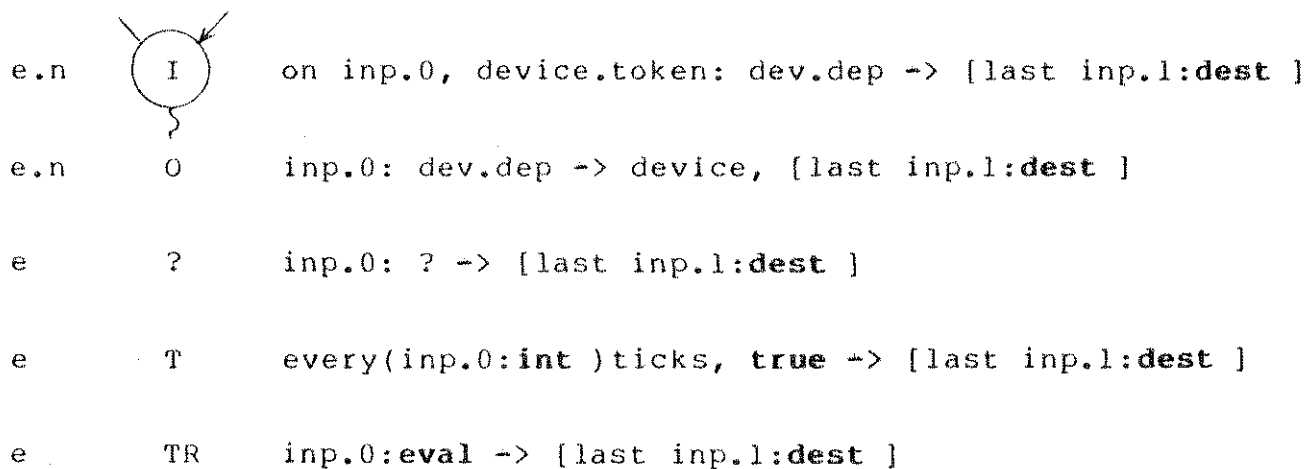
2.4 Restricted Organisational Nodes

As these nodes change the connectivity of the graph dynamically, and thus introduce non-determinacy, they should be used with some care.

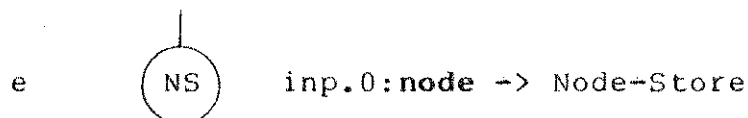


2.5 System Nodes

Diadic Nodes



Monadic Nodes



All system node-names are reserved with their node-descriptions

existing in specific processing-elements; e is the processing-element name and n is the element node-name. Although a particular system-node may be referred to at a number of places in the graph, it represents a single-resource. Multiple referencing therefore, implies non-deterministic merging on the node's input-points. Unless this is intentional, the node should be referenced once within an encapsulating resource manager. As input-token and output-token nodes have physical devices associated with their node-names, there will be restrictions on the type of tokens produced or accepted by these nodes. Type and length information is preserved in all input/output operations.

2.6 Non-deterministic Merge



inp.0, inp.1 -> out.0

This node does not appear in object graphs but should be used to enforce explicit non-deterministic merging of arcs in source graphs. Two or more arcs which are directed at a single input-point should be treated as errors by language translators.