A PARALLEL IMPLEMENTATION OF SASL

Jiannis Corovessis

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews

1983

# A PARALLEL IMPLEMENTATION OF SASL

by

Jiannis Corovessis

A thesis submitted for the degree of Doctor of Philosophy

Department of Computational Science

University of St.Andrews

St.Andrews

September 1982

Decleration

I declare that this thesis has been composed by myself and that the work that it describes has been done by myself. The work has not been submitted in any previous application for a higher degree. The research has been performed since my admission as a research student under Ordinance General No 12 on 1st. October 1978 for the degree of Doctor of Philosophy.

/
Jiannis Corovessis

I hereby declare that the conditions of the Ordinance and Regulations for the degree of Doctor of Philosophy (Ph.d.) at the University of St.Andrews have been fulfilled by the candidate, Jiannis Corovessis

A. T. Davie

## ABSTRACT

The applicative or functional language SASL is investigated from the point of view of an implementation. The aim is to determine and experiment with a run-time environment (SASL parallel machine) which incorporates parallelism so that constituent parts of a program (its sub-expressions) can be processed concurrently.

The introduction of parallelism is characterised by two fundamental issues. The type of programs, referred to as parallel and the so called strategy of parallelism, employed by the parallel machine. The former concerns deriving a graph from the program text indicating the order in which things must be done and the notion of "worthwhile" parallelism. In order to obtain a parallel program the original (sequential) program is transformed and/or modified. Certain programs are found to be essentially sequential. Parallelism is expressed as call-by-parallel parameter passing mechanism and by a parallel conditional operator, suggesting speculative parallelism.

The issue of the strategy of parallelism concerns the scheme under which a regime of SASL processors combine their effort in processing a parallel program. The objective being to shorten the length of computation carried out by the sequential machine on the initial program.

The class of parallel programs seems to be non-trivial and it includes both non-numerical and numerical programs. The "speed-up" by appealing to parallelism for such programs is found to be substantial.

# CONTENTS

Chapter Page

# CHAPTER ONE

## Introduction

The essence of program notations referred to as applicative or functional [1] lies in the fact that they possess the familiar properties of the notation of mathematics void of imperative notions. This is the point of difference with conventional program notations referred to as imperative, examples of which are the languages FORTRAN, ALGOL, PASCAL etc, where the basic programming notions are sequencing and store manipulation.

The meaning of a program in an imperative language is the behaviour (history of states) traced by the underlying mechanism executing the program (given its data). Certain states involve performing input or output. Thus computation is expressed in terms of state changes. Each constituent part of the program has to wait for the appropriate state of the machine to be arrived at before it makes its contribution.

The execution of a part may cause a "side-effect" on the execution of another. The presence of side-effects causes the underlying mechanism to be sequential, performing one thing at a time.

On the contrary the meaning of an applicative program is an object in the universe of discourse of the language. The object is referred to as the value of the program. At this level of programming the behaviour of the underlying mechanism evaluating the program is not addressed by the program.

The computation that an applicative program entails is a transformational process [2] of the program (data is part of the program in applicative languages) through a sequence of intermediate representations of its value to a final representation, providing the sequence converges. This is a canonical representation of the value the program denotes. The value is sometimes referred to as its Normal Form [3]. Obtaining this representation achieves termination of the computation since no further transformation is possible.

Computation as a transformation process suggests that there is an invariance relation between the states the evaluation mechanism traces, namely that the meaning of the program is preserved at all times. What actually changes between states (otherwise the machine would be of no use) is the representation of the programs's value. Each transformation results in more detail about the canonical representation being computed. Note we refer to evaluation of an applicative program and to execution of an imperative program for obvious reasons. An imperative program's result is obtained as a side-effect during its execution.

The contribution of a constituent part of a program is also a value which results after it has been transformed (simplified) to the canonical representation of this value. Thus the evaluation of this part has no "effect" on the evaluation of another part. Its purpose is to communicate this value.

We observe the evaluations a program entails are partially ordered with respect to the data dependencies between them. An evaluation is data dependent on another when the latter is a sub-evaluation of the former. This can be determined from the program text, represented as a graph, as will be shown in chapter five.

The standard evaluation mechanisms of applicative languages flatten this partial order to a total order bound by the uni-processor implementation environment. The objective of the present study is the construction of an evaluation mechanism which exploits the "inherent" parallelism of programs, suggested by the partial order and the type of parallelism possessed by various programs.

The investigation is based on the applicative language SASL [4]. The work is organised into chapters as follows.

Chapter two consists of three sections. The first introduces the basic features of SASL, the second describes its computational process and the mathematical properties it possesses and the third introduces the notion of parallel operators and a call-by-parallel parameter passing mechanism. Annotations in the program text are used to specify parallel primitive operators and a system function STRICT encapsulates the parallel parameter passing mechanism. Both parallel primitive operators and the (meta-)

function STRICT can be thought of as "hints" to the evaluation mechanism, specifying possible parallel behaviour.

In chapter three the parallel implementation of SASL developed in S-ALGOL [5] is described. This is based on an earlier implementation of SASL [6]. The evaluator of this SASL system was unsuitable for our purpose so a new parallel evaluation mechanism was constructed and interfaced to the rest of the SASL system. This gave us a complete SASL system to develope and experiment with parallel programs. The level of parallelism concerns the concurrent progress of evaluators each carrying out part of the computation a program entails.

Chapter four describes the model of parallelism where the evaluation mechanism employs different strategies for parallelism. A strategy of spawning, as it is referred to, determines when an assistant evaluator is to be assigned a task. Initially there is a single task and one evaluator. The presence of parallel operators in the program text, translated by the compiler into parallel instructions generate a tree of tasks, which represents the partial order mentioned previously.

A strategy causes either the realisation of the partial order by employing an evaluator on each branch of the tree or its conversion to a total order where a single evaluator traverses the tree simulating the parallel evaluation of tasks. No bound on the number of evaluators is assumed.

The effect of parallelism is measured in terms of the evaluation steps [7] a program entails. This characterises the performance of programs so that comparison between different strategies can be made. The results of the parallel evaluation of programs are presented in graphical form in the appendix.

In chapter five the idea of parallelisation of programs is put forward wherebye a SASL program written initially without consideration of the evaluation mechanism is transformed into a parallel program. Work has been done by Darlington and Burstall [8] on automatic and semi-automatic transformations. None of that is implemented here. Transformations to suitable forms is by hand directed by program graphs (see next paragraph). A parallel program as opposed to a sequential program is one whose evaluation splits into a number of sub-evaluations, each of which may decompose further. Programs structured in this way result from adopting a programming style known as <u>Divide-and-Conquer</u> [9].

In order to identify the sub-evaluations which can be carried out in parallel a program text is represented as a directed graph. The nodes of the graph correspond to operators in the language which construct composite parts (expressions). The arcs of the graph show the data dependencies between evaluations of parts. Arcs out of a node which do not converge onto a common node characterise the corresponding operator as parallel. This implies its operands may be computed in parallel.

A number of graph manipulation, list processing, numerical and symbol manipulation parallel programs have been developed. The parallelism of these programs is investigated by submitting them to the evaluation mechanism under different strategies of spawning.

Chapter six contains the results of testing the example programs developed in chapter five on the parallel evaluator described in chapter three, under different strategies as described in chapter four.

Conclusions are presented in chapter seven.

# CHAPTER TWO

## The language SASL, its computational process and its extension by parallel constructs

In this chapter we present the basic features of the applicative/functional language SASL (more details of which can be found in [4]), the computational process it entails and its extension with constructs which express a certain notion of parallelism. The name SASL stands for "St.Andrews Static Language". "Static" refers to the fact that SASL contains no commands and a data structure once created cannot be altered. "Applicative" or "functional" indicates the programming style of the language where algorithms are specified in terms of functions applied to arguments as the only "control" construct available to the programmer.

A SASL program is an expression which has for its value an object. The outcome of the program is to print a representation of the object, unlike the programs in imperative languages like FORTRAN, ALGOL etc. where a program specifies what behaviour step-by-step the machine is to perform, each construct in the program addresses the "state" of the machine.

In SASL algorithms are specified at an abstraction level over the state of the machine which executes the program. In fact all the states the machine traces in executing (evaluating) a program from the initial one to the final one are equivalent from the point of view that each preserves the meaning (value) of the program. Thus the state of the machine is not addressed in the program. The programmer computes with objects rather than with states. Note that although all machine states preserve the meaning of the program (and data) they are actually different states since each new state must be associated with computing some detail about the program's value not present in the previous state, otherwise the machine would be of no use.

Objects

The data items SASL expressions describe are called objects. Every expression has an object as its value. No significance is attached to expressions other than as a means to talk about objects. Any sub-expression expression can be replaced by any other which has the same value without affecting the value of a larger expression of which is is a part. This is a property of expressions called Referential Transparency [10].

The universe of discource of SASL has six types of object :-

1. Numbers - these are the integers such as -5, 0, 99 etc.

2. Truth-values - there are two such objects true and false.

3. Characters - %a represents the character a, %% represents the character % etc.

4. Lists - a list is an ordered collection of objects called its components. For example

1, 2, 1 and 99,

are lists of length 3 and 1 respectively. Note that repetition of components is allowed. A list may have an infinite number of components. For example the list of all integers is a well defined SASL object. The empty list is represented by the constant ().

5. Functions - a function is a rule which associates to a SASL object (the input of the function) a unique SASL object (its output).

6. Undefined - there is a unique object undefined which is the value of expressions such as %a+1 and of expressions which entail non-terminating computations. Note here we differentiate between a non-terminating computation and a computation whose result is an infinite list. An infinite list is a perfectly well defined object but only a finite number of its components can be printed in finite time.

The language obeys the rule that all six types of object have the same "civil rights" :-

any object can be named

any object can be a component of a list

any object can be the input of a function

any object can be the output of a function the above rule characterises the language as being semantically complete [11].

Expressions

Expressions are either atomic (they have no syntactic structure for example a constant or a name) or they are composite (constructed out of sub-expressions. The usual arithmetic, logical and relational operators construct one sort of composite expressions.

Juxtaposition of two expressions, for example

A B

denotes the application of a function to its argument (the input of the function). It also denotes selection from a list. For example

(1, 99, 4) 2

selects 99, the second component of the list.

List expressions are constructed with the operator :, for example

x : list

constructs a new list by prefixing to list the component x.

Commas are shorthand for list expressions. For example

1, 2, 3 and 1:2:3:()

are equivalent. Concatenation of two lists is denoted by the operator ++. For example

(1,2) ++ (3,)

gives the list 1, 2, 3

Another form of composite expression is the conditional expression constructed with the operator ->. For example

A -> B ; C

denotes the value of B or C respectively depending on whether the value of A is _true_ or _false_, otherwise it denotes the object _undefined_.

An expression may include definitions of names that appear in it using the _where_ construct followed by clauses. Each clause defines a name. For example

```
a + b
where
a = 1
b = 2
```

evaluates to 3.

Nested definitions are allowed, for example

```
a + b
where
a = 1
b = 2 + c
    where
    c = 3
```

evaluates to 6. Multiple definitions are also possible, for example

```
a, b, c = 1, 2, 3
```

is equivalent to

```
a = 1
b = 2
c = 3
```

Definitions in general are of the form LHS = RHS where LHS is a construction of arbitrary complexity built from names and constants using commas, brackets and the operator :. The RHS varies over expressions, for example

```
x : y = 1, 2, 3, 1+3
```

is equivalent to

```
x = 1
y = 2, 3, 1+3
```

the name y denotes the list 2, 3, 4.

In the case of function definition, LHS consists of the name of the function being defined followed by one or more formal parameters. As a formal parameter we can have a name, a constant or a construction of arbitrary complexity enclosed in brackets, as in multiple definitions. Names in formal parameters denote arbitrary input objects and they are local to the clause. For example the clause

sum (x,y) = x+y

defines the function which computes the sum of two integers, passed to it as the components of a 2-list. Another way of defining the same function is

sum x y = x+y

where we use the fact that a function (denoted by sum x) can return as its value another function (that which adds x to its parameter). Note that the (more general) function sum can be <u>partially parameterised</u> [12] to yield the (less general, specialised) functions incr and decr

incr = sum 1

decr = sum (-1)

so that the expressions incr 1 and decr 1 evaluate to 2 and 0 respectively.

Functions can be defined by more than one clause each clause covering a case of the parameters, for example the clauses

```
LENGTH () = 0
LENGTH (a:x) = 1 + LENGTH x
```

define the function which computes the length of its input list. The first clause applies to the case where the input is the empty list. In the second clause the input is a list where the name a denotes the first component of the list and x denotes the list of the rest of the components. Clauses are ordered by the order they are written. Thus in the example below

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

the function factorial expects its actual parameter to be the object zero or an arbitrary object, in that order. Definitions of names as well as definitions of functions can be circular too, for example

```
ones = 1 : ones
```

defines the infinite list 1: 1: 1 : ... Definitions can also be mutually recursive, as in the following program

```
list1
where
list1 = 1 : list2
list2 = 2 : list1
```

the above program denotes the infinite list 1: 2: 1: 2...

The syntax of the language obeys the rule that any expression can be a sub-expression of a composite expression. Wrapping up an expression in brackets does not have any effect on its value, it merely affects the syntax.

Computational process

The use of = in definitions of the form LHS = RHS has two important consequences

(a) It allows an equational proof theory [13] to be built where facts we wish to prove about programs are stated as equations (clauses) in the same language as the programs are written in. The clauses are used as axioms to derive a fact which holds for a program.

(b) It characterises the mechanism of computation the language entails based on the notion of substitution where every instance of the form LHS in an expression is replaced by the RHS providing the scope of names is taken into account in the obvious way. The substitution operation plus those operations such as +, * etc. determine how a computation gets done. We shall discover that this mechanism is flexible enough to allow the introduction of parallelism where the operations along the computation path overlap in time by splitting the computation path into parallel sub-paths. Consider the program for the factorial function, using each clause of the definition of function factorial as a substitution rule and arithmetic rules as simplification rules the computation path the program entails is shown in figure 2.1

factorial 3

3 * factorial 2

3 * 2 * factorial 1

3 * 2 * 1 * factorial 0

3 * 2 * 1 * 1

6

figure 2.1 - a computation path

note each substitution produces a refinement (simplification) of the representation of the object 6. We refer to the above process as being carried out by an "evaluator" for the language.

The evaluator comes up against the problem of which substitution to perform whenever there is a choice, as in the following program

```
g (factorial (-1))
where
g x = 1
```

if the inner substitution is always preferred the path diverges as shown in figure 2.2

```
g (factorial (-1))

g (-1 * factorial (-2))

g (-1 * -2 * factorial (-3))

.
.
.
```

figure 2.2 - inner substitutions, divergent path

if the outer substitution is performed the path converges to 1 in one step, figure 2.3

```
g( factorial(-1) )

1
```

figure 2.3 - outer substitution, convergent path

Another problem with substitutions is the possibility of different paths converging to different results. Both of the above problems are answered in the context of formal systems such as the Lambda Calculus [3] and SRS [14]. The Lambda Calculus is the basis of SASL and other applicative languages [15,16,17]. It is a formal system where concepts such as variable binding and variable abstraction can be studied but it is not a programming language because it lacks a definite universe of discourse. The entities referred to as functions in the Lambda Calculus have general character since they do not express a relation between some definite objects. The introduction into the Lambda Calculus of objects with their associated operations, like those supported by SASL, plus "syntactic sugar" gives a programming language, namely SASL. Mathematical results which hold in the Lambda Calculus by implication are assumed to hold for SASL too, although strictly speaking it must be proved they also hold for objects and operations introduced into the Lambda Calculus. Computation in the Lambda Calculus is carried out in terms of transforming an expression to another by applying certain rules, called reduction rules. These are concerned with renaming names occuring in an expression, simplification of certain expressions and substitution of an expression for the occurrences of a name in an expression. An expression which cannot be transformed any further by application of the reduction rules is said to be in <u>Normal Form</u>. Computation with an expression is a sequence of reduction rules applied to the expression. A finite sequence, producing a Normal Form of the expression, represents a terminating computation.

The central result in Lambda Calculus is the Church-Rosser theorem [18] which states that for expresssions A, B, C if A reduces to B and A reduces to C then there exists an expression D to which both expressions B and C reduce. This is diagramatically represented by completing the diamond where the arrow represents the application of a reduction rule, figure 2.4

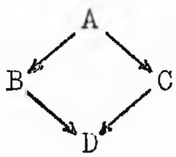



figure 2.4

a corollary of the Church-Rosser theorem guarantees uniqueness of Normal Forms. If two different computation paths which an expression gives rise to terminate, they do so with the same Normal Form. The Church-Rosser theorem secures independence from the order in which evaluations are carried out, except in the cases where the (meta-) algorithm driving the evaluator imposes a particular order so as to ensure that a non-terminaning path is not chosen at the expense of a terminating path.

An algorithm known as Normal Order Reduction which always performs the outermost leftmost reductions first is proved to achieve termination providing there is a Normal Form for the expression [3]. This is reflected in SASL by adopting a parameter passing mechanism referred to as call-by-need [19] where actual parameter expressions are passed unevaluated (no substitutions done on them) to the function. Thus the clause

```
f x = 1
```

defines a proper object (a function) even in the case where x denotes the object undefined.

From the point of view of the proof theory this is necessary in order to use
= as it is used in mathematics. Formally this is stated as the equality being fully substitutive [13].

Consider the following program

```
factorial 4
where
factorial n = fsplit 1 n
fsplit i i = i
fsplit i j = split i mid  *
             fsplit (mid+1) j
             where
             mid = (i+j)/2
```

at each occurence of the operator * we can split the computation path into parallel paths, see figure 2.5

factorial 4

fsplit 1 2 * fsplit 3 4

fsplit 1 1 * fsplit 2 2        fsplit 3 3 * fsplit 4 4

1        2        3        4

2        12

24

figure 2.5 - splitting a computation path

thus the Church-Rosser theorem gives rise to the possibility of several evaluators working simultaneously, each pursuing a sub-path of the computation a program entails. This brings us to the subject of this thesis which is to divise and experiment with such a mechanism.

Parallel operators and Call-by-parallel

In the previous section the possibility of parallelism was noted where a computation path splits (see figure 2.5) when an operator expression of the form A*B is evaluated. In order to identify the expressions that can be evaluated in parallel a program is represented as a directed graph. A node with arcs to other nodes identifies a composite expression constructed by some operator, its arcs point to nodes which identify the operands of the operator. The Clauses (definitions) are used to unfold [8] the graph. The graph shows the structure of a program in terms of the data dependencies between evaluations that it entails. An evaluation is data dependent on another when the former requires the result (value) of the latter. Data dependencies impose an order in which the associated evaluations must be carried out. Representing a program as a graph we see that evaluations are partially ordered with respect to the data dependencies which arise between them hence certain evaluations can be carried out in parallel. Consider the program graph, shown in figure 2.6, of the following program

```
rec 0 = 1
rec n = x + square x
        where
        x = rec (n-1)
square a = a * a
```

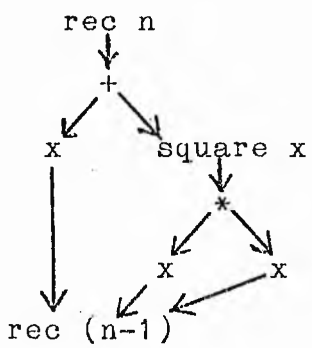



figure 2.6 - program graph of the function "rec"

in the graph, shown in figure 2.6, we see that both operands of the operator + depend on the evaluation of the sub-expression

rec (n-1)

and so they cannot be usefully evaluated in parallel. The same is true of the operands of *. Thus the values of the function rec for n, n-1, ... must be evaluated sequentially. Consider also the graph, shown in figure 2.7, of the function "or" (used in Example 3, chapter five), defined as

```
or m n = m=() -> n
                 m
```

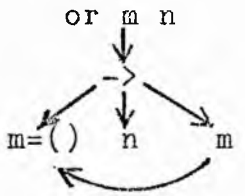



figure 2.7 - program graph of function "or"

the condition, m=(), and the left alternative, n, operands of the conditional operator ->, can be evaluated in parallel whereas the right alternative, m, is data dependent on the condition and hence must be evaluated after the condition has been evaluated. The operands of the relational operator = can be evaluated in parallel but the operand () entails a rather trivial evaluation offering no opportunity for useful parallelism.

Analysis of a program in this way, to discover the data dependencies and the informal analysis of what expressions are "worth" evaluating in parallel characterises the instances of operators which are to be interpreted as being parallel. The conditional operator is said to be strict* in its first operand and non-strict in the second and third. Other non-strict operators are &# and |# (logical and, or respectively) which have their operands evaluated in parallel. In the case of evaluating an expression of the form A&#B termination of one of A or B with the result false causes the evaluation of the other to become irrelevant [20] (even if its value is undefined) Thus non-strict operators involve initiating an evaluation in anticipation that its value might be needed.

Parallelism can also be manifested as parallel evaluation of the arguments of a function. For example expressions of the form

SUM matrix1 matrix2 k

met in a program for matrix multiplication (Example 8, chapter five) where matrix1 and matrix2 are sub-expressions which may be evaluated strictly before the whole expression is evaluated. In order to secure this form of parallelism, a call-by-value parameter passing mechanism, refered to as call-by-parallel must be adopted in this case. For this

* f is said to be strict when f undefined = undefined holds.

purpose a system function STRICT is implemented which effects call-by-parallel. It is described as follows

STRICT f x y = Evaluate strictly x and y and then
Evaluate the expression f x y

now the above expression becomes

STRICT SUM matrix1 matrix2 k

where matrix1 and matrix2 are evaluated in parallel. Note k is not "taken in" by the function STRICT. In fact the function STRICT can be defined in terms of the parallel operator &# as follows

STRICT f x y = x=x &# y=y -> f x y
'dummy"

where the expression x=x evaluates always to <u>true</u> and forces strict evaluation of x.

In general the pattern of the parameters which are taken will vary, for example suppose that in the following expression

F x1 x2 x3 x4

only x1 and x2 need be evaluated (strictly) in parallel. A function GS1 can be defined in terms of STRICT

GS1 F x1 x2 x3 x4 = STRICT aux x1 x3
<u>where</u>
aux a b = F a x2 b x4

Call-by-need is being retained in all other cases of function application where parallelism is not required. "Lazy evaluation" is another name for the call-by-need mechanism, mentioned previously, concerning parameters of functions and list constructors [21,22].

We use STRICT in a number of similar cases where operands of functions or infix operators : and ++ need to be called by value. Consider for example the function

```
FOR a b f = a > b -> ()
                  f a : FOR (a+1) b f
```

whose result is a list. Parallelism can be effected by replacing : by a function cons and forcing simultaneous call-by-value on its parameters

```
FOR a b f = a > b -> ()
                  STRICT cons
                         (f a)
                         (FOR (a+1) b f)
```

the graph of FOR is shown in figure 2.8

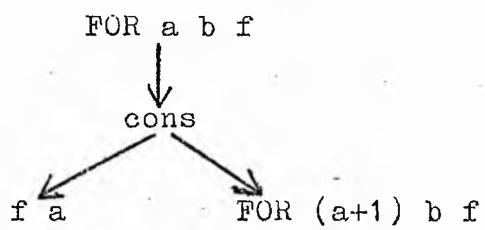



figure 2.8 - program graph of function FOR

however one of the parallel computations accomplishes little.

In order to balance the evaluations of the list's components FOR is modified as follows

```
SPLITFOR a b f = a = b -> f a ,
                 STRICT
                 APPEND
                 (SPLITFOR a mid f)
                 (SPLITFOR (mid+1) b f)
                 where
                 mid = (a+b)/2
```

and its graph is shown in figure 2.9

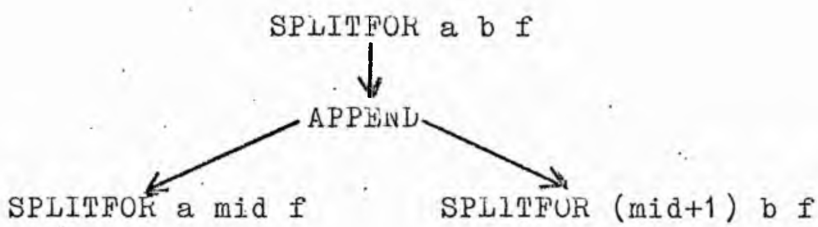



figure 2.9 - program graph of function SPLITFOR

note the need for an APPEND function in order to flatten back the result to a linear list.

The last two cases of parallelism suggest that in order to extract parallelism lazy evaluation has to be forced to do some work. Replacing call-by-need by call-by-value cannot be introduced safely without risking non-termination [23]. The approach to effecting parallelism adopted here is based on the parallel call-by-parallel scheme and on using annotation symbols which mark strict (infix) operators as being parallel, the parallel + operator, for example, is written as +# which the compiler takes note of and produces a parallel PLUS instruction for the evaluation mechanism.

# CHAPTER THREE

## Implementation

In chapter two we saw that the computational process SASL entails is based on the notion of substitution. This process is implemented on an abstract machine. "Abstract" refers to the fact that the machine's behaviour is simulated in software. An implementation of a substitution machine in hardware is reported in [24]. Substitution machines are of two basic types, characterised by the way they support the notion of substitution.

The first type consists of the Reduction machines, where substitutions are performed literally on the machine representation of a program. Each substitution results in modifying part of the representation. Termination is reached when there are no further substitutions to perform, a canonical representation of the object the program denotes has been obtained. The machine representation of a program is either a graph or a string. In graph reduction parts of the graph are shared through pointers. Reducing a shared part is felt simultaneously by all other parts which have pointers to it. In string reduction a substitution may produce multiple copies of a part and each has to be reduced separately. In graph reduction substitutions are performed on the program graph directly using the clauses (definitions) of the program as substitution rules [25] or the program is compiled into a fixed set of constants called combinators. This incorporates a process of removing all the variables which appear in the program based on a technique borrowed from Logic [26,43]. The operation of substitution on combinatory code is much simpler than that on program graphs where attention must be paid to conflicts of names.

The second type consists of the Interpreters or fixed program machines, where substitutions are simulated [27,28,29]. The machine representation of a program remains unmodified throughout the computation but the data mutates. The source text of the program is compiled into a code tree where each node of the tree represents an instruction of the machine. This is interpreted by the machine causing it to modify its state.

The present investigation is based on fixed program machines, known as the SECD machines [28]. The state of such machines consists of a Stack, an Environment, a Control and a Dump component. SECD machines represent the original attempts to implement applicative/functional languages, influenced by the machines of algol-like languages.

We shall describe an implementation of SASL based on the SECD type machines and then we shall modify it so that several machines can combine their effort in carrying out the computation a program entails.

## The SASL machine

The SASL machine is simulated by a program+ written in S-ALGOL. It is based on the original SASL machine [29] which supported a weaker version of SASL without infinite lists and multiple definitional Clauses with a call-by-value parameter passing mechanism. These features are supported in a later implementation of SASL [6]. This latter implementation consists of three parts, a monitor which handles interaction with the user, a compiler which translates a program, the user submits to the system, into a code tree and an evaluator which evaluates the code tree by recursively evaluating its sub-trees. The evaluator does not suit our purposes, for it simulates the SASL machine at a higher level not allowing us to examine its progress step-by-step. Thus we constructed a new evaluator++ and interfaced it to the rest of the SASL system. This has enabled us to obtain a full SASL system and experiment with a number of non-trivial programs.

Since SASL distinguishes between the different data types at run-time rather than at compile-time the machine has a "tagged" architecture. The memory of the machine consists of a number of cells each of which contains two data items, a head and a tail. In this implementation the management of the memory is left to S-algol. This facilitates the implementation effort and makes simulating the interaction of machines less painful. The machine's other components are a Stack (S) and three special

\+ [appendix II]

++ [appendix II, line 1124]

registers. A Control (C) register , an Environment (E) register and a Dump (D) register.

<u>The Control</u>

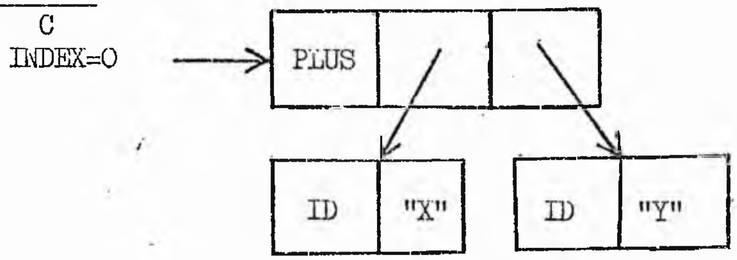



figure 3.1 - code tree for the expression X + Y

The C register points to the node of a code tree currently being evaluated (or interpreted) by the machine, such nodes contain instructions. Their sub-trees denote the operands of the instructions. The number of operandsdepends on the type of instruction. In figure 3.1 the code tree for the expression X+Y is shown.

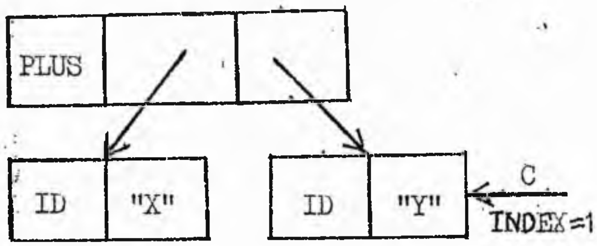



figure 3.2 - pre-order evaluation of X + Y

The C register has also a sub-component INDEX which parameterises the action of the machine for that instruction depending on whether none, one or both of the operands to the instruction are accessible on the Stack. This is necessary since a code tree is traversed (evaluated) in pre-order. The INDEX takes the integer values 0, 1, 2.

The Environment

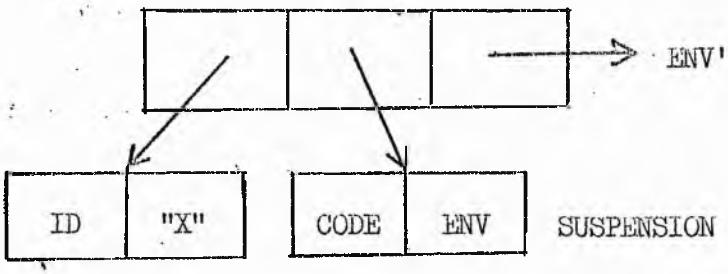



figure 3.3 - the Environment state component

The E register points to a linked list of name-value pairs, figure 3.3. The list is organised as a stack to reflect the nesting of environments. Thus the environment is a structure which keeps track of the names that are currently in scope and their associated values. Nested definitions result in nested environments.

Initially all names in the Environment are associated with suspensions. A suspension is a data structure with two data fields. A CODE field and an ENV field. It represents a "frozen" computation which on demand of its value the machine carries out by initialising its C and E registers from the suspension. On termination the value obtained overwrites the CODE field of the suspension and the ENV is used as a flag to indicate to subsequent accesses that it has been evaluated. Thus, if frozen code is ever evaluated, it is only evaluated once.

<u>The Dump</u>+

<table>
<tr><td>NEXTC</td><td rowspan="2">ENVR</td><td rowspan="2">LASTD</td></tr>
<tr><td>INDEX</td></tr>
</table>

figure 3.4 - the state component Dump

The D register points to linked list of nodes each of which is a data structure with three fields, see figure 3.4. Each node identifies a state of the machine to be restored when the evaluation of the code subtree currently pointed to by the C register is completed. Since code trees are traversed in pre-order the C register pointing to the node of the tree is saved in the NEXTC field and set to the node of the left sub-tree from where it can be restored.

The evaluation of some code sub-trees is carried out by extending the current environment with local definitions. This extension to the current environment has to be undone when control (the C register) returns to the father node of the sub-trees. Thus prior to extending it, the current environment is stored in the ENVR field of the data structure. The third field LASTD is used to organise the

+ [app. II, l. 1031]

list as a stack. Note the NEXTC field has a sub-field INDEX which indicates on restoring the state from the Dump what action remains to be done for the instruction. For example it may require checking the type of the value of the operand on top of the stack. An empty Dump indicates termination of the whole program.

Output

Initially the C register of the machine points to the root node of the code tree which the compiler produces from the program source. The root node identifies a special instruction PRINT with the rest of the code tree as its operand. The execution of PRINT causes the machine to save a "print" state on the Dump and continue with the evaluation of the operand of PRINT. Restoring the print state from the Dump causes the machine to output to the world outside the object referenced by the top of the Stack. [app. II, l. 2601]

A sequence of PRINT/EVALUATE actions can be performed with this mechanism which enables the machine to handle the case where a list is to be printed. The machine evaluates a component of the list, it prints it and then goes on to evaluate the next component, printing an infinite list is handled in the same way except that the end of the list is never reached.

In general a list is computed as follows. Initially none of the components of the list are computed. A data structure (a suspension, see above) with all the information to generate the list is passed around instead. The components of the list are evaluated, so that part of the list is actually generated, when access to the components of the list is required. This occurs when a list's component is an operand to a strict instruction (eg. arithmetic) or the whole list is operand to the PRINT instruction. This is known as lazy evaluation.

The "unfreezing" of computations is print-driven (ie. nothing is evaluated unless it contributes to the calculation of an object to be printed).

Instruction set

The operation of the machine on each instruction comprises the following five basic actions which manipulate the components of the machine [app. II, lines 1040-1094]

I. pushstack (item):

The object denoted by item is pushed onto the Stack. In fact a reference to the run-time object is pushed onto the Stack.

II. popstack:

The top element of the Stack is popped.

III. cont.state (code):

The C register is set to point to the code sub-tree denoted by code . The INDEX component of the C register is set to 0. This indicates that no operands are available on the Stack for the instruction at the node of the sub-tree.

IV. <u>save.state</u> (i):

The contents of the C, E and D registers are saved on the Dump. The INDEX component of the C register is set to the value i . This is the value of INDEX when the state is restored from the Dump.

V. <u>load.state</u>:

The top node of the Dump, pointed to by the D register, is popped and its contents initialise the C, E and D registers. This can be thought of as coming back to a "continuation" [30] left behind. The top of the Stack is the value passed to the continuation. If this value is suspended then it is evaluated and the result overwrites the code field of the suspension. The env field is used as a flag to indicate that the suspension has been evaluated.

The component INDEX of the C register is also initialised from the corresponding sub-field of the Dump. This indicates the number of operands to be expected on the Stack, currently accessible.

INDEX is also used to implement step-by-step actions of the evaluation mechanism such as parameter binding, clause matching (in the TRYS instruction below) and list selection (in the APPLY instruction below).

Each following instruction is given by its mnemonic followed by the names which denote its operands (sub-trees). The effect of each instruction on the current state of the machine is parameterised by the INDEX of the Control component of the state and it is described in terms of the five basic actions I-V.

ID name

INDEX = 0:

lookup name in the current E

push its value on the Stack

push an error value if not found

if the top of the Stack is a suspension

perform the following actions:

save.state (1)

set C to the code field of the suspension

set E to the env field of the suspension

cont.state (C)

Otherwise:

load.state

INDEX = 1:

If the Stack top is a suspension

perform the following actions:

save.state (1)

initialise C and E from the suspension

cont.state (C)

Otherwise:

overwrite name in current E

load.state

CONDITIONAL e, e1, e2

INDEX = 0:

save.state (1)

cont.state (e)

INDEX = 1:

If the top is the object true:

pop the top element of the Stack

cont.state (e1)

If the top is the object false:

pop the top element of the Stack

cont.state (e2)

Otherwise:

Replace the top element of the Stack with an error value.

load.state

BLOCK d, e

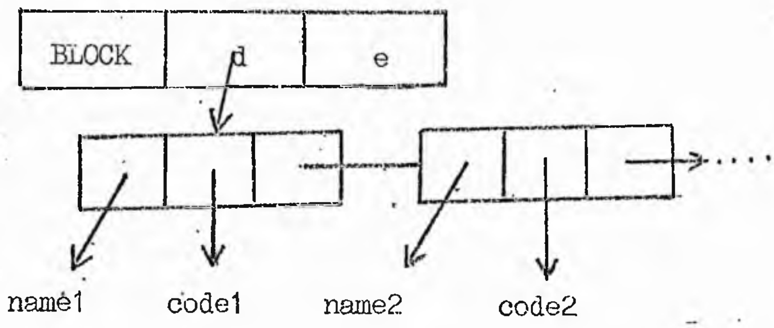



figure 3.5 - a BLOCK code tree

The name d denotes a linked list of names and code sub-trees which represent computations of their values, see figure 3.5.

extend current E with the definitions from d
each name is bound to a suspension
CODE fields initialised with corresponding code
sub-trees and ENV fields initialised with the
extended current E
cont.state (e)

MAP defs

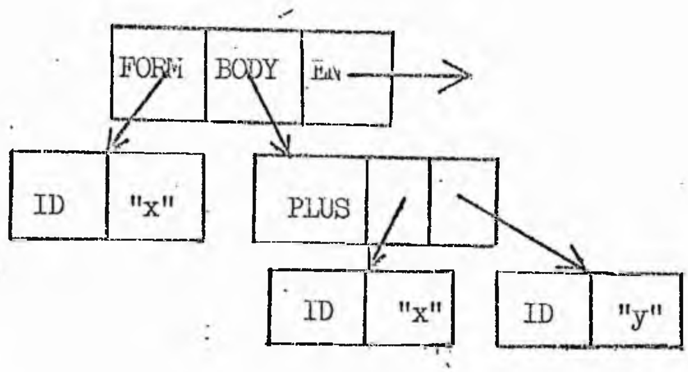



figure 3.6 - a closure representing the function

f x = x+y where E defines the free variable y

construct a closure from the code sub-tree
denoted by defs and current Environment (E)
push it onto the Stack
load.state

A closure is the machine's representation of a function, see figure 3.6. The field FORM denotes the parameter of the function being defined. It can be a constant, a name or a template the actual parameter must match. The field BODY denotes the code sub-tree to be evaluated as a result of applying the function to an argument. The field EN denotes the define-time Environment which is the current Environment E. On applying the function to an argument the sub-tree body is evaluated in environment en possibly extended with the binding of the formal parameter to the actual parameter (argument).

COLON e1, e2

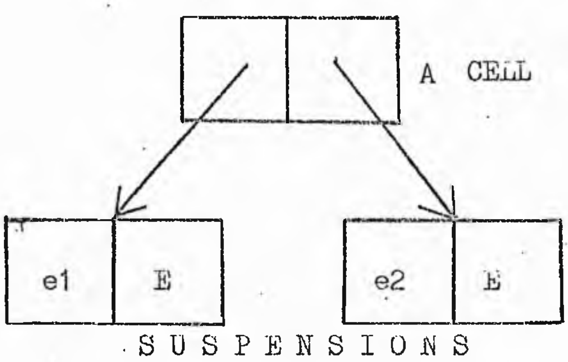



figure 3.7 - list cell creation

claim a new list cell (see figure 3.7)
create a suspension from e1 and current E
and initialise the head data field of the cell
create a suspension from e2 and current E
and initialise with it the tail data
field of the cell
push a reference to the cell onto the Stack
load.state

CHECKLIST e

INDEX = 0:

save.state (1)
cont.state (e)

INDEX = 1:

if top of the Stack is a list perform the following:

load.state

Otherwise:

pop the top element of the Stack

push onto the Stack an error value

load.state

HD e

INDEX = 0:

save.state (1)

cont.state (e)

INDEX = 1:

if the top of the Stack is a list:

pop the top element of the Stack

push the contents of the head onto the Stack

load.state

Otherwise:

pop the top of the Stack and push an error value

load.state

TL e

Perform the same actions as when applying the operator HD to a list except that the tail of the list is pushed onto the Stack instead of the head.

APPLY e1, e2

INDEX = 0:

save.state (1)

cont.state (e)

INDEX = 1:

if the Stack top is a list or a BASIC FUNCTION:

save.state (2)

cont.state (e2)

if the Stack top is a closure:

create a suspension from e2 and E

push it onto the Stack

bind formal parameter to actual parameter

(the top element of the Stack)

Binding may involve a matching process and may fail to match formal parameter to actual or it extends the current E with the formal parameter and its associated suspended value. This is referred to as the call-by-need parameter passing mechanism. If the binding process is successful it returns a new Environment otherwise it generates an error value which becomes the value of the function application.

set E to the result of binding

set C to the body of the function closure

cont.state (C)

<u>INDEX</u> = <u>2</u>:

if the second from the top of the Stack element is a list:

select the ith element of the list, where the top of the Stack denotes i and the second from the top element denotes the list

load.state

The selection process may involve generating the part of the list which is suspended in order to reach the ith element, see figure 3.8 (a),(b),(c) and (d)

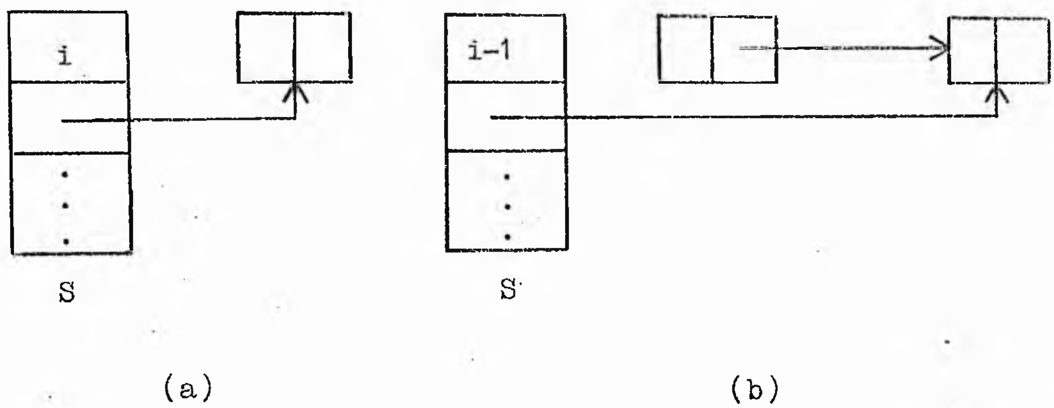
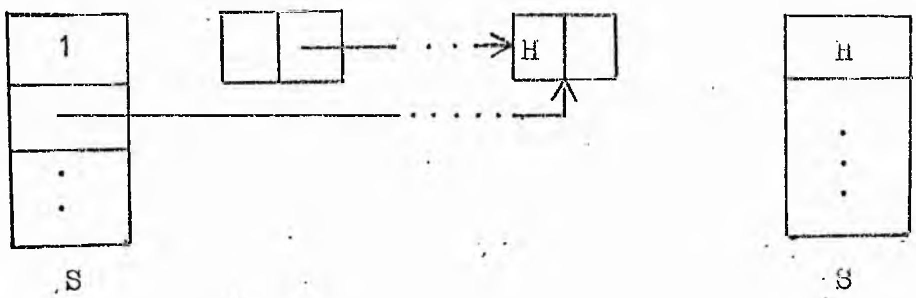



figure 3.8

the ith element is reached when the top of the Stack, used as list selector is the value 1




(c) (d)

figure 3.8

If the second from the top of the Stack element
is a BASIC FUNCTION: [app. II, 1. 986]
replace the top two elements of the Stack with the result of applying the function to the object on top of the Stack.
load.state

Note that BASIC FUNCTION is a predicate which tests the type of the object it is being applied to and it returns a Truth-value <u>true</u> or <u>false</u> on top of the Stack.

The class of instructions referred to by the mnemonics BINOP and UNOP represent the following arithmetic, relational and logical operators, defined over the appropriate type of objects

"+", "*", "-", "/", "rem", "~", "|", "&", "=", "<=", ">=", "<", ">", "~="

These are mapped by the compiler to machine instructions PLUS, TIMES, MINUS etc.

BINOP e1, e2

INDEX = 0:

    save.state (1)
    cont.state (e1)

INDEX = 1:

    save.state (2)
    cont.state (e2)

INDEX = 2:

    replace the top two elements of the Stack
    with the result of applying the instruction to
    these elements, the result is an error value
    when the type of the operands are not of
    type expected by the instruction
    load.state

UNOP e

INDEX = 0:

save.state (1)

cont.state (e)

INDEX = 1:

replace the top element of the Stack
with the result of applying the instruction
to this element

load.state

This completes the instruction set of the machine except for the instruction TRYS, similar to MAP but the closure it constructs represents a function defined by more than one clause. The implementation of multiple definitional Clauses is rather involved, relying on giving different values to the INDEX subcomponent of the Control component, each identifying a "state" of the process which selects the representation of the function whose formal parameter matches the actual parameter the closure is being applied to. The effect of the instruction TRYS is desrcibed in detail in [6]. [app. II, l. 1626]

Introduction of parallelism

In chapter two we proposed the use of annotation marks which induce the compiler to produce parallel instructions. Executing a parallel instruction, such as PAR-PLUS+ for example, has the effect of the current machine switching to

+ [appendix II, line 2086]

a WAIT state. The code sub-trees (operands) and current Environment of the machine initialise the C and E registers of new assistant machines. On termination of its assistants the machine may resume its computation.

The machine described in the previous section needs to be modified so that several machines combine their effort in executing a program. Each machine now has an additional Destination register whose contents identify its father machine. A machine is identified as a slot which receives a result. On termination a machine sends its result to this slot.

Since new machines are initialised with the same Environment, it is possible for them to access a suspension simultaneously or a machine to access a suspension which is currently being "coerced" to the value it denotes by another machine. Simultaneous access to a suspension which has been "coerced" (or "unfrozen") to the value it represents, poses no problem since all accesses are read-only. Otherwise simultaneous access or access while the suspension is being "coerced" to its value does present a problem.* In order to prohibit the same evaluation being carried out by different machines only the first machine must be allowed exclusive access. This saves unnecessary work being done. For this reason a suspension now has an extra "lock" field which is set by the machine which accesses it first and reset when it is overwriten. A machine which finds a suspension locked switches to a

* Let us examine how deadlock may arise between, say, two evaluations A and B. This can only occur when they are data dependent upon each other.

A : Requires the value of a sub-expression
let us name it X which is itself
data dependent on the sub-expression Y

B : Requires the value of the sub-expression Y
which is data dependent on the sub-expression X

In SASL this arises from certain definitions of the form

X = .... Y ....

Y = .... X ....

for example

X = 1+Y

Y = 1-X

which give the equation X = 2-X satisfied by the object <u>undefined</u>

Note however the equation X = 1:X admits a solution, the infinite list 1:1:1..... and <u>undefined</u> is not a solution.

Thus deadlock only arises when a SASL program denotes the object <u>undefined</u>

LOCKED state until it is evaluated.

The operation of the machine is extended by the following two actions: [app. II, lines 2352-2513]

VI. spawn (code, env, slot):

This action is invoked when a machine meets a parallel instruction. It causes a new assistant machine to be initialised by a code sub-tree denoted by code and by an Environment denoted by env. The slot initialises the Destination register of the machine. It denotes a place on the Stack of its father machine.

VII. kill (machine_id):

This action is invoked on two occasions. Firstly, it is invoked by a machine which terminates its operation normally. Secondly, it is invoked by a father machine which no longer requires the result of the computation carried out by its child machine. The identity of the child machine is denoted by machine_id. We can think of the father machine sending a kill signal to its child. The kill signal is propagated by the child to all of its children and so on.

The effect of a parallel instruction is described below. The cases for the instructions PAR-OR+ and PAR-AND are treated as special ones since they are more powerful than the corresponding sequential ones.

PAR-BINOP e1, e2

+ [appendix II, line 2133]

PAR-BINOP

spawn(e1, ENV, slot1)

spawn(e2, ENV, slot2)

switch to WAIT state

The top two places on the Stack are reserved as slots to receive the results from the evaluation of the operands to the parallel instruction. A machine in WAIT state checks its Stack for results from its children, it then applies the instruction to these results. The machine resumes its progress by invoking the load.state action.

PAR-OR e1, e2

spawn (e1, ENV, slot1)

spawn (e2, ENV, slot2)

switch to WAIT state

WAIT state:

if top or second from the top element
is the object true:

kill (child)
pop the two top Stack elements
push onto the Stack the object true
load.state

if one slot is the object false
and the other is an error value:

push onto the Stack an error value
load.state

if both slots contain error values:

push onto the Stack an error value
load.state

if both slots contain the object false

pop the Stack twice
push this object onto the Stack
load.state

Otherwise:

remain in WAIT state

Similarly for the other parallel instructions PAR-AND and PAR-CONDITIONAL etc. [app. II, l. 1180]

Error Handling [app. II, l. 2597]

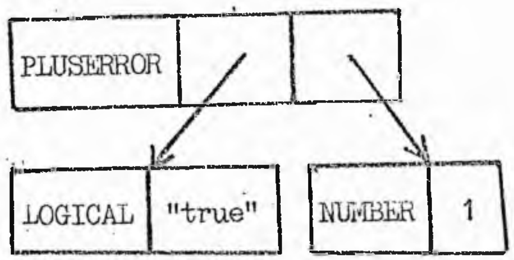



figure 3.9 - the error value "true + 1 "

The sequential (lazy) evaluator whenever it detects an error it terminates its progress and prints it as the value of the program. This is represented by the evaluated (or partially evaluated) code sub-tree. The node contains the instruction and the branches point to its evaluated operand (s), as shown in the example above. Since the control of the parallel evaluator is distributed this error value is treated as any other value. The corresponding task sends it to its father task this to its father and so on until the top task is reached which reports a partially evaluated code sub-tree (built bottom-up).

The partially evaluated code tree represents a trace of the computation carried out. The trace can be suppressed by having each task just propagating the smallest sub-tree (the error value) so that the error value climbs the tree of tasks unmodified.

# CHAPTER FOUR

## A model of parallelism

The parallel evaluator described in chapter three decomposes the evaluation of a program into a tree of tasks. The execution of a parallel instruction causes the current task to switch to a wait state until the operands of the instruction are available. These are to be evaluated as newly created tasks. In the implementation a new task is created by the primitive action spawn.

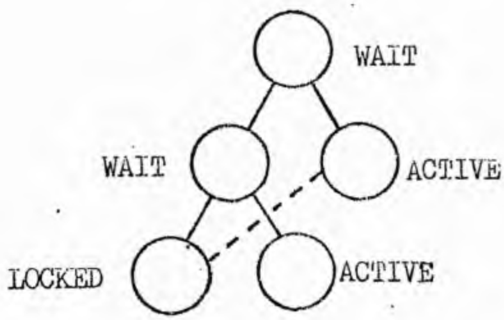



figure 4.1 - a tree of tasks

If a snapshot is taken at the parallel evaluation of a program the overall state is a composite of "smaller" states which form a tree, see figure 4.1. A node identifies a task in a particular state. ACTIVE states indicate the tasks are being processed, WAIT states identify tasks waiting for results of other tasks. A task is in a LOCKED state when it requires the value of a common suspension currently being "unfrozen" by another task. It remains in LOCKED state until the suspension is overwritten with the value it represents. The possibility of interference of the above kind between tasks where a task becomes (dynamically) dependent on the value of another task other than the direct father/son dependency suggests that there is a graph of tasks and not just a tree. The broken line in figure 4.1 indicates that temporary data-dependencies arise between tasks, when the dependencies are resolved the related tasks still continue in existence. Unbroken lines show the flow of values which are obtained with the completion of tasks.

Conflict in the form of simultaneous access to the value of a suspension is a consequence of the efficient implementation of lazy evaluation in the environments model of computation (SECD type implementation). This is the technique by which non-strict functions and infinite lists are supported. In the graph reduction model of computation conflict would also arise between tasks due to "sharing" parts of the graph. Evaluating a shared part of the graph is felt simultaneously by all other references to it. On the contrary, string reduction gets round this problem by duplicating effort on common parts.

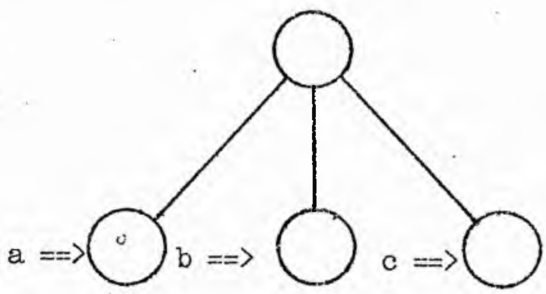



figure 4.2 - ACTIVE tasks are associated with evaluators a, b, c

ACTIVE tasks are associated with the loci of control of evaluators which process them, shown by arrows in figure 4.2. In order to model the behaviour of a multi-processor machine we must take into account the following two observations.

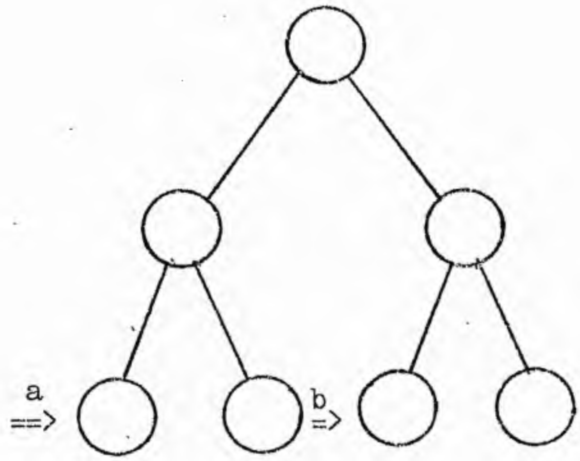



figure 4.3 - Tasks exceed evaluators

First, as active tasks are being processed they generate new tasks. The number of created tasks, for a program of modest size, soon overwhelms the number of evaluators, see figure 4.3.

Second, the assignment of tasks to evaluators may involve considerable communication overheads. The above observations suggest that active tasks should not necessarily receive the attention of evaluators as soon as they are created. Thus in the model an assistant evaluator to the current evaluator is employed only after a certain amount of "time" has elapsed (see below).

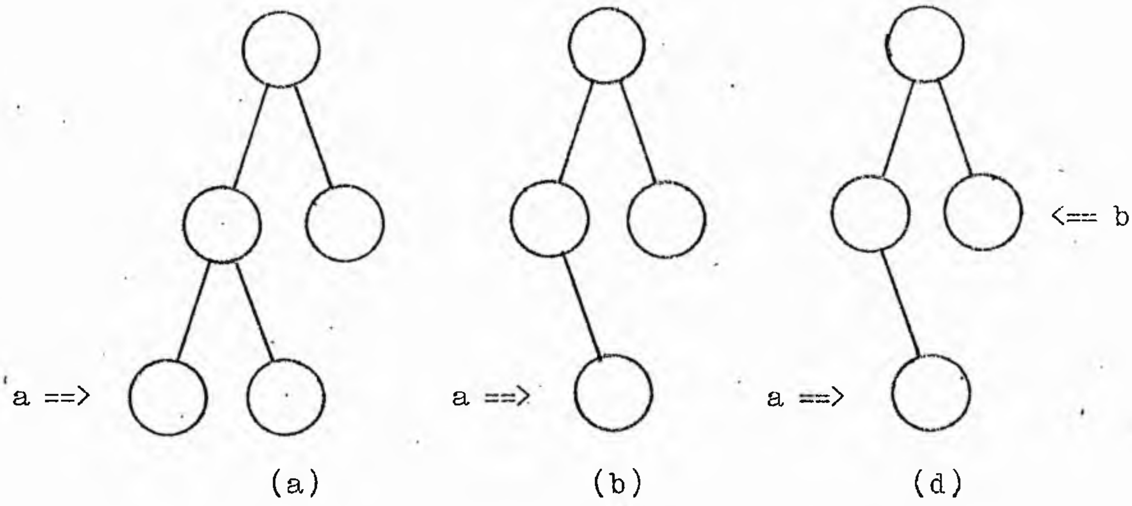



figure 4.4 - each evaluator simulates
the parallel evaluation of tasks
a: main evaluator, b: assistant

In the absence of assistant evaluators the locus of control of the current evaluator traces a bottom first leftmost path over the tree of tasks, see figures 4.4 (a),(b) and (c). Thus each evaluator will attempt to simulate the parallel evaluation of tasks it creates.

When an assistant evaluator to the current one is actually employed it is assigned the last task to be processed by the current evaluator. Further assistants similarly are assigned the next to last tasks. Thus assistant evaluators take load off the current evaluator. These may have the benefit of other assistants in the same fashion and so on.

The effect of a "parallel run" of a program in the model is measured by the amount of effort the initial evaluator exerts. This is the number of steps it goes through to evaluate its input program. A step is equivalent to the execution of one instruction, as described in chapter three. Thus in the model the locus of control of the initial evaluator is associated with a count of the number of instructions it performs. Also a count of the number of lock steps which occurred during its progress as well as the total number of lock steps is noted for each run.

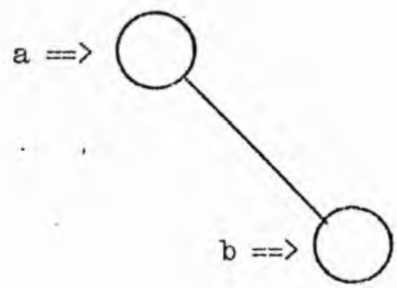



figure 4.5

In the case shown in figure 4.5 where the main evaluator has come back to a task assigned to an assistant evaluator which has not completed it yet, it is assumed from this moment onwards that the effort of the assistant evaluator counts as if it was exerted by the main evaluator.

<u>When to spawn</u>

It has been mentioned that an assistant evaluator is employed after some "time" has elapsed. During this time several (or no) new tasks may have been created pending processing. Time is related to the amount of work the evaluator performs. Thus its locus of control is associated with a clock which registers its effort. The clock is set to a certain threshold which, when it gets exceeded, causes the initiation of an assistant evaluator. This occurs every time the threshold is exceeded providing there are tasks to be processed. Time has been measured in three different ways.

First, as the number of instructions executed.

Second, as the number of COLON instructions (list cell creations).

Third, as the number of APPLY instructions (function applications).

In order to make this quantity ("time") relative to the evaluation of each program experimented with, a threshold is computed as a percentage of the total number of instructions performed under sequential evaluation where a single evaluator is employed.

Note that the delay a threshold imposes is finite so that the correct result of parallel operators such as "|#" (PAR-OR) and "&#" (PAR-AND) is computed. If the evaluation of one of the operands diverges then eventually the threshold which prohibits the spawning of the task for the other operand will be exceeded. This would cause the second operand task to be processed. If its value is true for OR or false for AND then the application of the parallel operator will return this value as its result.

Each program is evaluated in the model under different strategies of spawning where a strategy is determined by the particular threshold imposed. All strategies are bounded by two extremum cases.

The Totally sequential case where only a single evaluator is employed. This evaluator simulates the parallel evaluation of all tasks. So the partial order of tasks represented by the graph in figure 4.4 (a) is flattened to a total order.

The Maximally parallel case where a new evaluator is employed as soon as a task is created. Between these two strategies there is a spectrum of strategies which result in imposing an order on tasks otherwise unordered.

A series of experiments is performed for each program under different strategies. The outcome of each experiment for each program apart from its result provides the following information.

The number of steps performed by the initial evaluator, as a percentage optimisation over the number of steps under totally sequential strategy.

The number of lock steps of the initial evaluator.

These constitute actual delay in the overall evaluation.

The total number of lock steps indicating the amount of interference between evaluators. The performance in each experiment is plotted against the corresponding strategy.

Sample points taken at regular intervals during the evaluation which show the number of tasks being processed at each sample point. The profile of an evaluation is presented as a histogram. The results of experiments appear in chapter six.

Simulation

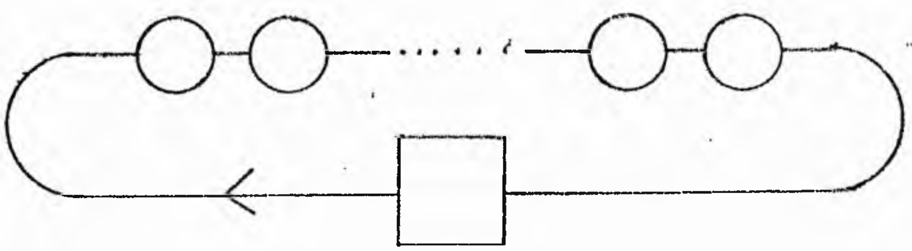


figure 4.6

In the absence of real concurrency the behaviour of the model of parallelism is simulated by a program which executes sequentially. Its locus of control (S-ALGOL's) timeshares over the tasks so that each task is processed for a timeslice equivalent to the execution of one instruction.

The interaction between evaluations of tasks is modelled at instruction execution level. Modelling at sub-instruction level would be required to examine storage management problems for example. Such a simulation is reported in [31]. Figure 4.6 shows how parallelism is achieved. All tasks are arranged in a ring structure with the processor going round the ring giving each task a step equivalent to one instruction. Tasks in wait state just examine their Stack slots to see if their assistant tasks have produced any results. Tasks in locked states examine the field "lock" of the suspension. Pending tasks which have not "fired" yet are ignored.

The action of a task killing its sub-task as it discovers it does not need its result any longer, is assumed to occur instantaneously before any other task changes state. This is a rather ideal situation since the problem of identifying irrelevant tasks and terminating them in order to recover the portion of resources allocated to them is not a trivial problem [20,32,33]. The main problem is that of "chasing" where if the number of newly generated tasks, sub-tasks of a killed task, which receive the attention of processors, grows faster than the rate of killing them then this can result in the machine being taken over by irrelevant tasks. This is analogous to the case where a garbage collection process runs out of space itself while trying to recover unwanted space in a sequential machine.

The simulation works at a level above the problems of resource allocation that a real multi-processor machine would have to deal with. Here the main idea is to discover the amount of parallelism "logically" present which can be exploited. The simulation does not answer the problem of whether such parallelism can be "physically" realised.

From the point of view of a parallel architecture the ring suggests the arrangement shown in figure 4.7.

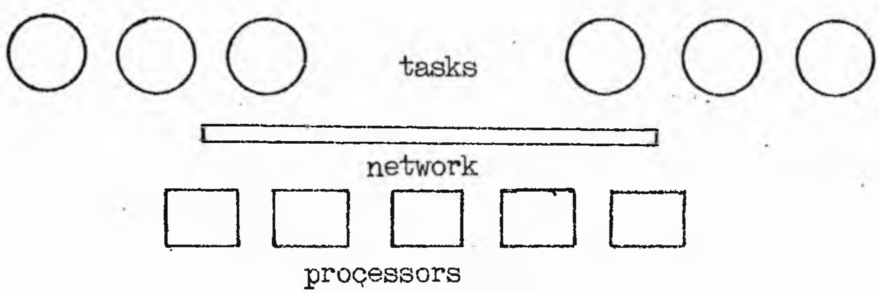



figure 4.7

The machine consists of a pool of processors and a pool of tasks. A task has some portion of the total memory engaged. The fact that the run time structure is highly interleaved suggests that there must be a globally referenced memory divided into blocks. As a task is being processed it generates more tasks which can be processed by the current processor or other processors. The proposals for architectures [31,39,40] take up this problem more fully.

# CHAPTER FIVE

## Parallel Programs

In this chapter we examine a number of SASL programs with the purpose of identifying evaluations that can be carried out in parallel. In some cases the original program must be transformed or even replaced by a more parallel program.

An expression represented as a graph of data dependencies shows the evaluations that can be carried out in parallel. Evaluations are ordered by the data dependencies that arise in their evaluations. A data dependency indicates that computing the value of an expression requires that of another expression.

The complexity of evaluations is important in deciding the grain of parallelism [34]. This is a criterion by which we consider, for example, the operation of multiplying two matrices as appropriate for organising it in parallel, whereas we consider the multiplication of two integers not appropriate because the grain of parallelism in this instance is too fine. Consider the program for computing the exponentiation function

```
expo x 1 = x
expo x n = x * expo x ( n-1 )
```

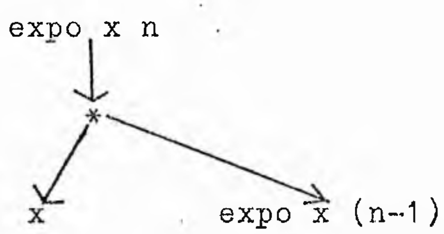



figure 5.1 - program graph of expo

Its graph representation, shown in figure 5.1, indicates sub-expressions x and expo (n-1) may be evaluated in parallel. The complexity of the evaluations though suggests rather unbalanced evaluations. This means there is relatively little amount of work to be done in parallel.

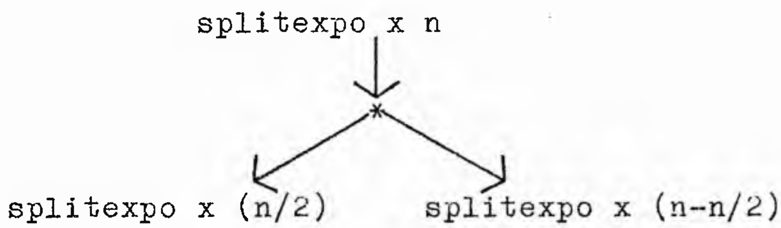



figure 5.2 - program graph of splitexpo

A transformation of the program produces a balanced split exponentiation function, see figure 5.2, defined as

```
splitexpo x 1 = x
splitexpo x n = splitexpo x (n/2) *
                splitexpo x (n-(n/2))
```

Now we can interpret the primitive operator * as being parallel. For that purpose we introduce an annotation symbol # which directs the compiler to generate a parallel instruction for the benefit of the evaluator. Parallel instructions cause an evaluation path to split into parallel paths.

<u>Examples from graph theory</u>

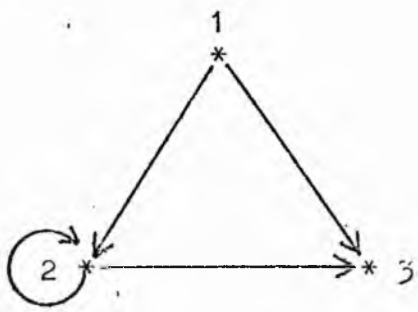



figure 5.3 - a directed graph

Graphs model many real life situations so graph manipulating programs are interesting cases to examine. In particular we will examine graphs of relationships.

A <u>directed</u> graph G consists of a finite number of vertices and arcs labelled by a direction. We choose to name vertices by integers. The directed graph shown in figure 5.3 has vertices 1, 2, 3 and arcs 12, 13, 22, 23. Arrows indicate the direction of each arc. We define a function G to represent the graph.

```
G 0 = 3 || the size of graph in vertices
G 1 = 2,3
G 2 = 2,3
G 3 = ()
```

The function G is passed as a parameter to graph manipulation functions.

<u>Example 1</u>

To compute the reachability relation of a graph G , by following the outgoing arcs from each vertex.

<u>program</u>

```
Rel G =
FOR 1 (G 0) reach
where
reach i = i,'to",extend () i , nl
extend sofar i = MEMBER sofar i -> ()
                 UNION arcs succs
                 where
                 arcs = G i
                 succs = MAPUNITE (extend (i:sofar)) arcs
```

The standard functions FOR, MAPUNITE,UNION and MEMBER are defined by the following Clauses

```
FOR a b f = a > b -> ()
              f a : FOR ( a+1 ) b f
MAPUNITE f () = ()
MAPUNITE f (a:x) = UNION (f a) (MAPUNITE f x)
MEMBER () a = false
MEMBER (a:x) a = true
MEMBER (a:x) b = MEMBER x b
UNION () y = y
UNION (a:x) y = MEMBER y a -> UNION y x
                 a : UNION x y
```

Sets are represented by lists. The output of the function FOR is a list. This represents the reachability relation (can be thought of as a new graph) for the input graph. The components of the list are computed sequentially as the list is being printed.

Parallelism manifests here as parallel evaluation of the list's components. This is effected by defining a function SPLITFOR which computes the list as a balanced tree (represented as a list of lists) and then flattens the tree into a linear list by the APPEND function.

```
SPLITFOR a a f = f a,
SPLITFOR a b f = APPEND (SPLITFOR a mid f)
                        (SPLITFOR (mid+1) b f)
                 where
                 mid = (a+b)/2
APPEND h t = h++t
```

To transform the function SPLITFOR to a parallel function we use the function STRICT which simulates simultaneous call-by-value on the operands of append, this is defined as call-by-parallel. Note there is always a choice to be made concerning the grain of parallelism which selects a certain function to be transformed into a parallel one. This involves apart from the structure of the corresponding flow graph knowledge of the complexity of the function. Whether this is left to the user to decide or for the system to cope with automatically is an open question. For example the function MEMBER which scans a list could also be chosen for parallel transformation.

Example 2

A directed graph G is called cyclic if there exists a vertex which can reach itself. To test whether a given graph is cyclic we use the function extend defined in the previous example.

program

```
cyclic G = cycleat 1
           where
           cycleat i = i > G 0 ->false
                       MEMBER path |# cycleat (i+1)
                       where
                       path = extend () i
```

The evaluations of sub-expressions

MEMBER path i          cycleat (i+1)

operands to the parallel operator |# (parallel-or) can be carried out simultaneously. The relative complexity of the evaluations suggests that they are unbalanced. So we must transform the function cycleat so that the looping it entails is unfolded in a tree structure with the operator |# at the nodes.

```
cycleat i i = MEMBER path i
              where
              path = extend () i
cycleat i j = cycleat i mid |# cycleat (mid+1) j
              where
              mid = (i+j)/2
```

<u>Example 3</u>

Modify the previous program to compute the vertex at which the cycle starts. We can modify the function cycleat to return the name of the vertex instead of <u>true</u> and the empty list instead of <u>false</u>.

<u>program</u>

```
cycleat i i = MEMBER path i -> i ; ()
cycleat i j = or (cycleat i mid)
                 (cycleat (mid+1) j)
or left right = right=() -># left
                right
```

To effect parallelism we define a parallel conditional operator -># which evaluates the predicate expression and the left alternative in parallel. Note the because of the data dependency of the predicate to the right alternative we do not need a full parallel conditional. A second annotation mark would be required to define such an operator. Termination of the predicate with the value <u>false</u> causes the evaluation of the left alternative to be forcibly terminated, if it is still going on, as irrelevant.

<u>Example 4</u>

A vertex of a directed graph is called <u>terminal</u> if a directed cycle cannot be reached from it. If a graph is not cyclic the set of terminal vertices consists of all the vertices of the graph.

Compute the set of terminal vertices of a directed graph.

<u>program</u>

```
terminals G =
FILTER term (COUNT 1 (G 0))
where
term i = ~nont i
nont i = OR (MEMBER path i : MAP nont path)
OR () = false
OR (a : x) = a ¦ OR x
```

we use the function "extend" from example 1. The function COUNT computes the list 1, 2, ....(G 0) which is filtered to leave in only those components (vertices) which satisfy the predicate term. We choose to parallelise the function FILTER. We replace the list 1, 2, ...(G 0) by introducing an extra parameter in FILTER and apply the split transformation to it.

```
FILTER p n n = p n -> n,
               ()
FILTER p n m = APPEND (FILTER p n mid)
                      (FILTER p (mid+1) m)
               where
               mid = (n+m)/2
```

now APPEND is prefixed by STRICT as in example 1. Further parallelism is possible from the function OR which scans a list looking for the object <u>true</u> as soon as it finds this object it returns it as its result, otherwise it returns the object <u>false</u>. The parallel OR function is defined as follows

```
OR () = false
OR (a:x) = a |# OR x
```

note since the sequential function OR does not need to evaluate all the components of the list, only as far as the first <u>true</u>, the parallel OR function involves evaluating components in anticipation that their value might be needed.

<u>Example 5</u>

In a directed graph when a vertex v has an arc to a vertex u then the vertex u is called the <u>successor</u> of v and v is called the <u>predecessor</u> of u. For some vertex i the <u>minimal transition pair</u> with i as the initial vertex is the smallest pair of sets M and N such that

vertex i is a member of M

all successors of M are members of N

all predecessors of N are members of M

The following program computes the sets M and N for a given graph g which satisfy the above conditions.

```
mtpair g
where
MN mset nset = c1 & c2 -> mset,nset
               MN (c1 -> mset ; mset1)
                  (c2 -> nset ; nset1)
               where
               c1 = SUBSET succs mset
               c2 = SUBSET preds nset
               succs = MAPUNITE succ mset
               preds = MAPUNITE pred nset
               nset1 = UNION nset succs
               mset1 = UNION mset preds

succ v = g v
pred v = FILTER (arc v) (COUNT 1 (g 0))
arc i i = false
arc i j = member (succ j) i
COUNT a b = a > b -> ()
            a : COUNT (a+1) b
```

The function COUNT computes the list 1, 2, ....n which is the list of vertices of the input graph g .

An undirected graph, shown in figure 5.4, is represented here with a double arc.

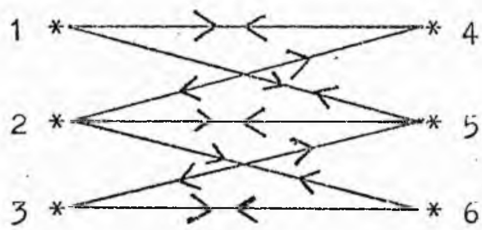



figure 5.4 - an undirected connected graph

such a graph is called connected if every vertex is reachable from any other. An undirected connected graph is called bipartite if its vertices can be partitioned in two sets M and N such that no edge (a double arc) joins two vertices of the same set. To solve the problem whether a given graph is bipartite can be programmed as follows

Let i be an initial vertex, say 1. We can use the function MN, defined above, to assign the vertices to two sets M and N such that vertices joined by an edge are assigned to different sets. As the graph is connected all vertices will be assigned to at least one set. The graph is not bipartite if a vertex has been assigned to both sets.

program

```
bipartite g = empty (INTERSECTION M N)
                  where
                  empty () = true
                  empty  s = false
                  M,N = MN (1,) (succ 1)
```

Note since vertices are joined by double arcs there is no need for the function pred, just use succ. The empty set is represented as the empty list (). The function INTERSECTION computes the denoted set operation. In order to transform mtpair into a parallel program the operator & is replaced by the parallel operator &# so that sub-expressions c1 and c2 are evaluated in parallel. Since &# is strict in only one of its operands (see the PAR-AND instruction in chapter three), termination of one of the evaluations, say c1 for example, giving false causes the termination of the evaluation of c2 as irrelevant.

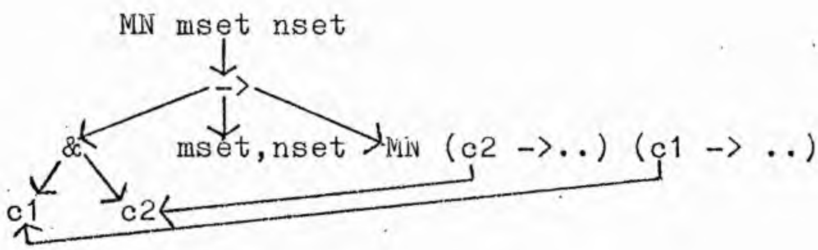



figure 5.5 - program graph of "MN"

Note that it is possible that the value of c2, for example, will be required by the evaluation of the expression

MN ( .... ) ( c2 ....)

the graph of "MN" in figure 5.5 indicates that the latter evaluation is data-dependent on the evaluation, characterised by the operator &# as speculative. This suggests that both the values of c1 and c2 must be found before the operator &# is applied. So the sub-expression

c1 & c2

in the sequential program is replaced by

STRICT and c1 c2

where the function "and" is defined by

```
and x y = x & y
```

Example 6

To test whether a function contains a zero in a given interval within a given accuracy criterion (the local version of SASL does not cope, at present, with real numbers but the program will work on a variety of "scaled" integer functions).

The method of solution is to divide the given interval into two sub-intervals and search for a zero of the function in the sub-interval which indicates the function crosses the x-axis. If neither sub-interval indicates this condition they are searched left to right by being subdivided further.

program

```
Root f x y e = x-y < 2*e ->
                   negsign x y ->'root is ", mid
                   'no root found",
               negsign x mid -> left
               negsign mid y -> right
               ONEOF left right
               where
               negsign a b = f a * f b  < 0
               left = Root f x mid e
               right = Root f mid y
               mid  = (x+y)/2

ONEOF m n = isnroot m -> n
            m
isnroot (mesg:x) = x = ()
```

Parallelism here manifests as splitting the interval and pursuing the test on each sub-interval in parallel. Success on one of the sub-intervals renders the search in the other as irrelevant (if one is looking for just one root).

Again the full parallel conditional operator was not needed. Only the condition and left alternative need be evaluated in parallel. Note that for the particular case where the pattern of searches followed by the sequential program is optimal, this occurs when the sequential program never takes up a right half interval, the introduction of parallelism does not improve the performance. In general though we can safely assume this will not be the case. Note also that as soon as a path hits success this is detected by the immediate application of ONEOF and reports it to the outer application of itself so that the answer reaches the top of the tree causing termination of search paths on its way. This is effected by replacing -> by the parallel operator -># in the body of the function ONEOF (see example 3).

Example 7

The program to compute the moves of discs which solve the towers of Hanoi.

program

```
Hanoi 0 (a,b,c) = ()
Hanoi n (a,b,c) = Hanoi (n-1) (a,c,b),
                  move,
                  Hanoi (n-1) ( b,a,c )
                  where
                  move = 'disc ",a,' to",c
```

To transform the function Hanoi into a parallel function we just replace the two occurences of comma by a function comm2 and use STRICT to force call-by-parallel on the parameters of the function comm2.

```
Hanoi n (a,b,c) = STRICT comm2  l r
                  where
                  comm2 l r = l,move,r
                  l = Hanoi (n-1) (a,c,b)
                  r = Hanoi (n-1) (b,a,c)
```

Note that no transformation of the program to enhance parallelism is required since the evaluation of sub-expressions l and r are of the same complexity.

Example 8

To compute the matrix product of two matrices. In order to present a clearer program let us assume the matrices are square of dimension n, power of 2. A matrix is represented as a list of lists in row order. For example the expression

((1,0),(0,1))

represents the unit square matrix of order 2. We define the product in terms of inner product operations between vectors. A row or a column of a matrix constitutes a vector. The inner product function IP is defined by the following Clauses

```
IP () () = 0
IP (r : x) (c : y) = r * c  +  IP x y
```

a matrix is transposed by the function transpose below

```
transpose M = map hd M : transpose (map tl M)
hd (a : x) = a
tl (a : x) = x
```

The ith row of the product matrix is formed by taking the inner product of the ith row of matrix M with all the columns of matrix N.

program

```
multiply M N = mult M (transpose N)
mult () cols = ()
mult (r : rows) cols = new r : mult rows cols
                       where
                       new row = MAP (IP row) cols
```

We identify parallel evaluations at the level (grain) of function mult where the operands of : can be evaluated in parallel. Similarly at the inner level of MAP used by the function new and finally at the level of function IP.

The infix operator : can be replaced by a function cons and then we can use the function STRICT to force call-by-value on the actual parameters of cons. The complexity of the function new and more obviously of IP with respect to the complexity of the whole program suggests that we only consider parallelism at the level of the function mult. Note that had we decided to consider parallel evaluations, say at the level of function IP, we would need to transform this function in order to balance the tree of evaluations that the parallel + operator gives rise to.

The same criticism applies in introducing parallelism at the level of function mult whose parallel balanced version may be defined as follows

```
mult rows cols = split 1 (LENGTH rows)
                 where
                 split i i = new (rows i),
                 split i j = STRICT APPEND
                                    (split i mid)
                                    (split (mid+1) j)
```

Now the rows of the product matrix are computed in parallel. Closer examination of the algorithm shows that each such evaluation requires access to the column matrix cols. This implies the evaluations cannot proceed independently of each other. In order to obtain an effectively parallel program for matrix multiplication we therefore look for a different algorithm, in fact the function split above provides the idea. The computation of an element is given by the formula

$$c_{ij} = a_{i1} \; b_{1j} + a_{i2} \; b_{2j} + a_{i3} \; b_{3j} + \; \ldots.$$

Let us consider multiplying matrices A and B obtaining matrix C, all of dimension n, a power of 2.

this function in order to balance the tree of evaluations that the parallel + operator gives rise to.

The same criticism applies in introducing parallelism at the level of function mult whose parallel balanced version may be defined as follows

```
mult rows cols = split 1 (LENGTH rows)
                 where
                 split i i = new (rows i),
                 split i j = STRICT APPEND
                                     (split i mid)
                                     (split (mid+1) j)
```

Now the rows of the product matrix are computed in parallel. Closer examination of the algorithm shows that each such evaluation requires access to the column matrix cols. This implies the evaluations cannot proceed independently of each other. In order to obtain an effectively parallel program for matrix multiplication we therefore look for a different algorithm, in fact the function split above provides the idea. The computation of an element is given by the formula

$$c = a \quad b + a \quad b + a \quad b + \ldots.$$

Let us consider multiplying matrices A and B obtaining matrix C, all of dimension n, a power of 2.

we can divide matrices A and B so that they form (2X2) matrices whose elements are (n/2)X(n/2) matrices. An element Cij of the product matrix is computed using the same equation as above, for example

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$

but the operands of addition and multiplication are matrices. The equation indicates the two multiplications can be carried out in parallel. Note each multiplication between the matrices A i j and B i j , of dimension n/2 will give rise to parallel evaluations of matrices of dimension n/4 and so on until multiplication of atomic operands is reached.

The computationthis algorithm implies recursively sub-divides into non-trivial independent evaluations. The new algorithm for matrix multiplication is yet another example of the approach to problem solving known as the Divide-and-Conquer method. In fact we have already encountered many examples of this method when the functions splitexpo, SPLITFOR, cycleat were defined. Programs implementing this type of algorithm are ideally suitable for parallel evaluation, since their evaluation splits evenly into sub-evaluations. These can be carried out in parallel.

In order to change the representation of a (nXn) matrix A so that A i j is not an integer but a (n/2) X (n/2) matrix we define a function make2 to do the conversion.

```
make2 A n = G
            where
            G 1 1 = F
                   where F i j = A i j
            G 1 2 = F
                   where F i j = A i (j+offset)
            G 2 1 = F
                   where F i j = A (i+offset) j
            G 2 2 = F
                   where F i j = A (i+offset) (j+offset)
            offset = n/2
```

Since the results of operations are square matrices printed as lists of lists, we define a function MATRIX which produces a square matrix

```
MATRIX n e =  FOR 1 n r
              where
              r i = FOR 1 n c
                    where
                    c j = e i j
```

The addition of matrices of dimension k, represented as 2X2 matrices with elements matrices of dimension k/2 is defined by function SUM as follows

```
SUM F G 1 = F + G
SUM F G k = MATRIX 2 e
            where
            e i j = SUM (F i j) ( G i j) (k/2)
```

The multiplication of 2X2 matrices is defined in terms of the function MATRIX as follows

```
mult2 A B = MATRIX 2 ((A i 1*B 1 j)+(A i 2*B 2 j))
```

Now we can define matrix multiplication anew in terms of the functions mult2, make2, MATRIX and SUM

program

```
multiply A B 2 = mult2 A B
multiply A B n = mult A2 B2 (n/2)
                 where
                 A2 = make2 A n
                 B2 = make2 B n

mult M N k = MATRIX 2 e
             where
             e i j = SUM (multiply (M i 1) (N 1 j) k)
                         (multiply ( M i 2) (M 2 j) k)
                         k .
```

We identify parallelism at the level of function MATRIX where the elements of the matrix can be evaluated in parallel. This is effected by transforming FOR into a parallel function. Note that since the first parameter of MATRIX is 2 the function FOR produces a 2-list so that there is no need to transform FOR to the function SPLITFOR, we have met this function in example 1.

Parallelism is also identified at the level of function e where the operands of the function SUM can be evaluated in parallel. We choose the level of the function SUM because the grain of the function MATRIX overwhelms the simulator, even for a small (8X8) matrix.

Note the bulk of the work is done at the level of the function SUM and although an element may be evaluated before another is it has to be printed in a particular order. To effect parallelism the function STRICT is used to perform call-by-parallel on the operands of SUM

```
e i j = STRICT SUM (multiply  (M i 1) (N 1 j) k)
                   (multiply (M i 2) (N 2 j) k)
                   k
```

Example 9

To sort a list of integers in ascending order. There are a number of sorting algorithms [35]. We choose the sort by merge algorithm because it employs the Divide-and-Conquer technique. Other sorting methods such as quicksort do this also but are not considered here. Given a list of n numbers, split it into two sub-lists of n/2 and n+2/2 numbers and then merge the sorted sub-lists.

program

```
sort x = split 1 (LENGTH x)
         where
         split n n = x n,
         split n m = STRICT merge
                             (split n mid)
                             (split (mid+1) m)
                     where
                     mid = (n+m)/2

merge () y = y
merge x () = x
merge (a : x) (b : y) = a <= b -> a : merge x (b : y)
                        b : merge (a : x) b
```

Note that all parallel evaluations require access to some element of the list x.

Example 10

To compute a relation from two relations. The relations are two tables of library information. One table gives the relation between books and authors and the other between borrowed books and names of borrowers. The relation to be computed is defined as "the list of authors whose books are lend to other authors".

A table is represented by a list of pairs. Each pair is represented by a 2-list. The list of book-author pairs is denoted by the parameter BAL and the book-borrower list of pairs is denoted by the parameter BBL.

program

```
new_rel BAL BBL = relation BBL
                  where
                  relation () = ()
                  relation(p:x) = rel p -> p 2:relation x
                                     relation x
                  rel(bk,br) = AND(author~=(),author~=br)
                               where
                               author = FIND BAL br

AND () = true
AND (a:x) = a -> AND x ; false
FIND () item = ()
FIND ((item,nm) : x) item = nm
FIND ((bk,item) : x) item = bk
FIND (p : x)   item = FIND x item
```

The function FIND searches the list of pairs for an item, if it finds the item contained in a pair it returns the related object. Both functions relation and FIND may be parallelised. We choose the grain of parallelism offered by the latter function which performs FIND steps recursing on its first parameter. We transform the recursions into a tree whose terminals, left to right, are the unwound recursions.

```
relation x = split 1 (LENGTH x)
             where
             split n n = rel (x n) -> x n 2,
                         ()
             split n m = APPEND (split n mid)
                                (split (mid+1) n)
                         where
                         mid = (n+m)/2
```

By prefixing APPEND with the system function STRICT a parallel program is obtained.

Example 11

A partition of an integer n is a collection of positive integers whose sum is n. The integers in the collections are called the parts of the partition. We do not impose any other restrictions on the partitions. Consider the partitions of the first three integers.

1

2 11

3 21 12 111

We see that the partition of 3 is generated from those of 2 and 1 by extending (prefixing) the partitions of 2 with 1 = (3-2), obtaining 12 111 and of 1 with 2 = (3-1) obtaining 21 and finally of 0 which is empty (nullpart) with 3 = (3-0) to get 3.

<u>program</u>

```
    nullpart = (),
    part 0 = nullpart
    part n = fora 1 n last
             where
             last i = prefix i (part (n-i))
             prefix i () = ()
             prefix i (p : x) = p=nullpart->(i,) : prefix i x
                                 (i : p) : prefix i x

    fora a b f = a > b -> ()
                 f a ++ fora (a+1) b f
```

By paralleling function fora, the partitions may be generated in parallel.

```
    fora a b f = a=b -> f a,
                 STRICT APPEND (fora a mid f)
                               (fora (mid+1) b f)
                 where
                 mid = (a+b)/2
```

Note that same partitions are recomputed, following the technique of [36] we modify the function part to be a memo function which remembers previously generated partitions.

```
partlist = MAP part (from 0)
part n = fora 1 n last
         where
         last i = prefix i (partlist (n-i+1))
```

Example 12

To generate the permutations of set of integers. We take the solution given in the Sasl manual [4].

program

```
perms () = (),
perms x = f x
          where
          f (a : y) = MAP (cons a) (perms y) ++
                           g (y ++ (a,))
          g y = y = x -> ()
                f x
```

Similarly here replacing ++ by the function APPEND and using STRICT to effect simultaneous call-by-value of the parameters of APPEND, the evaluation path splits into parallel sub-paths. One path computes the permutations of a list of numbers where the first element is fixed. The other rotates the list and computes its permutations. Each sub-path follows the same split pattern.

Let us consider the same algorithm expressed somewhat differently so that parallel paths are of the same complexity, where the loop defined by g has been taken out and externalised.

```
perms x = MAP f x
          where
          f a = MAP (cons a) (perms (diff x a))

diff () a = ()
diff (a : x) a = x
diff (b : x) a = diff x a
```

we replace MAP by the function "split"

```
perms x = split 1 (LENGTH x)
          where
          split n n = f (x n),
          split n m = STRICT APPEND
                                (split n mid)
                                (split (mid+1) m)
                      where
                      mid = (n+m)/2
```

Example 13

The queens problem where the queens are to be placed on an (nXn) board in such a way that none checks any other. We use the solution of [4] where the board is represented by a list. The components of the list represent the columns of the board. Each component is an integer and its value represents the row of the board.

complexity, where the loop defined by g has been taken out and externalised.

```
perms x = MAP f x
          where
          f a = MAP (cons a) (perms (diff x a))

diff () a = ()
diff (a : x) a = x
diff (b : x) a = diff x a
```

we replace MAP by the function "split"

```
perms x = split 1 (LENGTH x)
          where
          split n n = f (x n),
          split n m = STRICT APPEND
                              (split n mid)
                              (split (mid+1) m)
                      where
                      mid = (n+m)/2
```

Example 13

The queens problem where the queens are to be placed on an (nXn) board in such a way that none checks any other. We use the solution of [4] where the board is represented by a list. The components of the list represent the columns of the board. Each component is an integer and its value represents the row of the board.

program

```
soln q b = q > 8 -> alter b
           safe q b -> full q b -> q : b , soln (q+1) b
                                   soln 1 (q:b)
           soln (q+1) b
```

The algorithm starts from an initial position and either extends if the safe condition is satisfied or it modifies it to for a safe condition. It backtracks when a safe condition must be found by altering previously placed queens. Intuitively , we feel that fixing the initial positions and pursuing them in parallel to success or failure without backtracking will give us a parallel program. Allowing backtracking means that parallel paths may converge on the same route.

```
FOR 1 n initial
where
initial = soln (1,)
soln b = safe b -> full b ->b
                   FOR 1 n extend
                   where
                   extend q = soln (q : b)
```

By parallelising the list generator function FOR we easily obtain a parallel program. In fact we only transform the first occurence of FOR otherwise the run time structure overwhelms the simulator.

To program the numerical method of solving Laplaces's equation on a rectangular grid with given boundary values. This problem is programmed on the Data Flow computer [37] using a different approach to parallelism.

Initially the interior points of grid are given estimated (guessed) values and a new point on the grid is computed using the formula

$$U_{nij} = (U_{n\,i-1\,j} + U_{n\,i+1\,j} + U_{n\,i\,j-1} + U_{n\,i\,j-1})/4$$

where n is the iteration step and i and j vary over the rows and columns respectively of the grid. The interior of the grid is iterated until successive values on each point differ by a given amount, which characterises the degree of accuracy of the approximation. The initial grid is given a constant value on all the interior points. The choice of initial value affects the number of iterations required to achieve convergence.

<u>program</u>

```
    output
    where
    R = NO_OF_ROWS
    C = NO_OF_COLS
    BOUND_VALUE = ... || a R-list of C-lists

    output = MAP grid (from 0)
    grid 0 = INIT_GRID
    grid n = FOR 1 R r
             where
             r i = FOR 1 C c
                   where
                   c j = BOUNDARY (i,j) -> BOUND_VALUE i j
                         (output (n-1) i (j-1) +
                          output (n-1) i (j+1) +
                          output (n-1) (i-1) j +
                          output (n-1) (i+1) j
                         )/4

    BOUNDARY (i,j) = OR (i=1,i=R,j=1,j=C)
    OR () = false
    OR (a:x) = a |# OR x
    from n = n : from (n+1)
```

The function "from" it produces an infinite list which plays the role of the loop control variable in imperative programming.

The grid is represented as a list of lists. The output of the program is an infinite list denoted by the identifier "output". Each component of the list is a grid. Note the lazy evaluation mechanism of SASL enables output to be received from such an infinite computation. The pattern the computation follows is "compute a component of the list, print it and do the same for the rest of the list". When convergence is achieved we interrupt the computation. This can be determined by comparing the values printed out. A better solution where convergence is tested from within the program might be preferable but this program is adequate to demonstrate the idea of successive approximations being generated.

The algorithm adopted here can be thought of as a "bottom-up" method of solution. The evaluation of each new point comprises a rather trivial computation path. One way to extract parallelism is to divide the grid into sub-grids and compute each in parallel. The amount of parallelism obtained in this way depends on the size of the grid. But even a relatively small grid may involve a large number of iterations before convergence is achieved. This makes the amount of work on each sub-grid comparatively small.

In order to extract parallelism in the form where the whole evaluation process sub-divides into non-trivial smaller evaluations it seems we must adopt a "top-down" method of solution, where the result of the program is just the grid after a number of iterations. Intermediate grids not being printed. In this way the evaluation of each point on this "final" grid involves a respectable amount of work.

program

```
    output k = FOR 1 R r
               where
               r i = FOR 1 C c
                     where
                    c j = U k i j

    U 0 i j = INIT_GRID

    U n i j = BOUNDARY (i,j) -> BOUND_VALUE i j
              (U (n-1) i (j-1) +
               U (n-1) i (j+1) +
               U (n-1) (i-1) j +
               U (n-1) (i+1) j
              )/4
```

The + operators are marked as parallel +# and the arithmetic expression of the form A + B + C + D is re-arranged to (A + B) + (C + D) in order to have balanced paths.

Since SASL does not support a package for simulating real numbers we were not able to test this program properly but only from the point of view of unfolding the recursions in parallel.

Example 15

To program a parser for Lambda Calculus strings defined by the following syntactic rules expressed in BNF (we use L instead of for typographical reasons).

```
wfe = var | lamb var . wfe | ( wfe ) | wfe wfe
var = a | b | c | .....
```

The syntax specification for a well formed formula of Lambda Calculus is that it is either a variable or a function or a bracketed well formed expression or a concatenation of well formed formulae. First of all immediate recursion is removed from the above syntax specification by introducing extra rules.

```
wfe = e1 | fun
e1  = e2 { e2 }
e2  = var | (wfe)
fun = lamb var . wfe
var = a | b | c | x | y | z
```

where { } indicate zero or more repetitions of the enclosed object. For simplicity the syntactic variable var is assumed to vary over just six names. For each syntactic variable a function is defined which recognises whether that entity occurs at the front of its input string. Such a recogniser function returns two results. A logical indicating success of failure to recognise the item and the remaining of the input string after the item has been taken from the front of the input string. A recogniser corresponding to the left hand side of a syntactic rule uses recognisers corresponding to the right hand sides of rules.

A recogniser for a terminal symbol is a function which tests whether a particular string occurs at the front of its input string, ignoring leading spaces. So the test is string equality. In order to avoid defining a separate recogniser of each terminal, a function which takes a string as its input and returns a recogniser for that string is defined.

```
term pattern string = f pattern string
                      where
                      f p () = false,string
                      f () s = true,s
                      f p (% :s) = f p s
                      f (a:p) (a:s) = f p s
                      f p s = false,string
```

so the symbol ( is recognised by a function bra defined as

```
bra = term '("
```

The alternative (¦) BNF symbols are defined by a function bar which takes a list of alternative recognisers as its first parameter and an input string as its second parameter and tests whether the front of the string can be recognised by any of the recognisers

```
bar () string = false,string
bar (rec1 : x) string = r1 -> true,s1
                         bar x string
                         where
                         r1,s1 = rec1 string
```

thus var is defined as

```
var = bar (a, b, c, x, y, z)
```

and further for a , b , c , x , y , z , lamb

```
var = bar (map term ('a",'b",'c",'x",'y",'z"))
lamb = term 'L"
```

similarly concatenation is defined

```
conc () string = true,string

conc (rec1 : x) = r1 -> r2 -> true,s2
                         false,string
                         where
                         r2,s2 = conc x s1
                   false,string
                   where
                   r1,s1 = rec1 string
```

Using the function conc we can define a recogniser for the category funct as follows

```
funct = conc (lamb , var , dot , wfe)
dot = term '."
```

Using bar and conc defining repetion is developed as follows

```
repet obj = bar (obj.....obj , zero)
zero string = true,string
obj.....obj = conc (obj , repet obj)
```

thus finally in legal SASL

```
repet obj = bar (conc (obj , repet obj) , zero)
```

Now we are in a position to define the complete parser wfe using the recognisers defined already

```
wfe = bar (e1 , func)
e1  = conc (e2 , repet e2)
e2  = bar (var , conc (bra , wfe , ket))
```

To identify what can be done in parallel the functions bar and conc are analysed by unfolding their graghs. The graph of bar is shown in figure 5.6

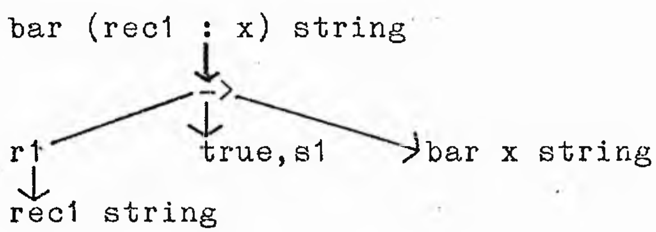



figure 5.6 - program graph of "bar"

it suggests that the condition and right alternative of the conditional operator may be evaluated in parallel. In order to use the parallel operator ->## already defined r1 is replaced by ~r1 and the alternatives of the conditional are swapped.

The graph of conc is shown in figure 5.7

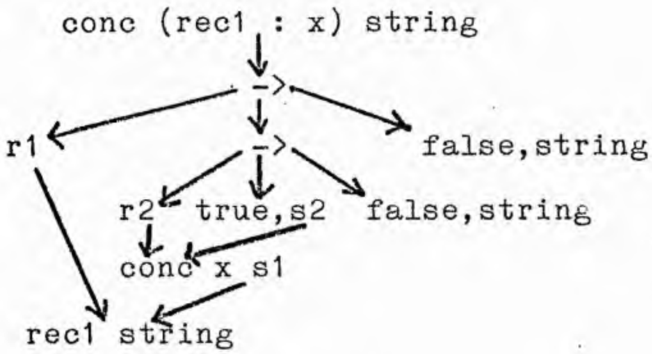



figure 5.7 - program graph of "conc"

it indicates that the sub-expressions

rec1 string          conc x s1

cannot be be evaluated in parallel since the latter is data dependent on the value denoted by s1 which is part of the result of the former. Thus the function conc is characterised as essentially sequential.

Finally, the "split" transformation applied to the function "bar" gives us the following balanced function

```
bar x string = split 1 (LENGTH x)
               where
               split i i = x i string
               split i j = ONEOF
                              (split i mid) (split (mid+1) j
                              where mid = (i+j)/2
```

```
ONEOF l r = ~hd l -># r ; l
```

# CHAPTER SIX

## Results

In this chapter we analyse the results, obtained by running the programs developed in chapter five, on the parallel evaluator (see chapter three and four) and comment on the method by which parallel programs are derived from sequential ones.

Simulation results are presented in the appendix, in the form of tables. Tables numbered n.1 and n.2 correspond to the example program numbered n in chapter five. What do the tables mean ?

As we have discovered in chapter five a (parallel) program task decomposes into a tree of sub-tasks. A special case of this is the program which tests a directed graph for the bipartite property (Example 5) where its evaluation only occasionally decomposes into two sub-tasks and the rest of the time it consists of a single task.

The evaluator has a choice of evaluation schemes at its disposal. This is controlled by an input parameter (see below about "strategy") of the simulation. The most obvious schemes are two, the totally sequential scheme where no parallelism is invoked at all and the other is the maximally parallel (most eager evaluation) scheme where as soon as a sub-task is created it is assigned to an evaluator. The sub-task is processed independently of the main task and its associated evaluator is an assistant to the one processing the main task. In between these extreme evaluation schemes there exist a number of evaluation schemes each dictating when evaluations are "forked out" from ongoing evaluations. Thus certain tasks which would be processed in parallel under the maximally parallel scheme are evaluated in sequential order under an "in-between" scheme. Note that although the evaluator's behaviour seems to vary between eager and lazy evaluation this is not strictly accurate since call-by-parallel has replaced call-by-need (see chapter two) even in the absence of parallelism due to the dictates of a particular scheme. In this case the evaluator simulates the parallel evaluation of sub-tasks. Each evaluation scheme is called a "strategy" of the evaluation mechanism. Below we explain how strategy is quantified.

The simulation we have constructed sets out to discover how to exploit the parallelism "inherent" in the programs of chapter five by testing different parallel evaluation schemes (strategies). The effect of each scheme is measured by the resulting length of computation (number of main evaluator's steps).

The performance under each parallel evaluation scheme is calculated as a percentage improvement over the length of computation under the totally sequential scheme. Thus in table 1.2 for instance a particular strategy (horizontal axis) of 10% (see below) achieves 60% gain in performance (shown in the vertical axis).

Strategies

A particular strategy dictates when each evaluator is to "off-load" (logically) a sub-task to an assistant evaluator. This is when a parallel activity is to be set up. A strategy models the degree of parallelism employed in a real machine consisting of multi-processors for a given program.

So under a strategy a certain amount of work is shared amongst evaluators and a corresponding improvement over the sequential strategy (just a single task, the main one) is expected.

A strategy amounts to an assumption concerning the pattern of resource allocation in a real machine. In this study we have assumed that an evaluator gets the benefit of an assistant after it has performed a certain amount of work. During this time it may have generated some or no new (sub-) tasks. In the former case these are assigned to assistant evaluators as the particular strategy dictates.

The condition of unbounded parallelism (number of assistants or number of "off-loadings") is assumed.

The "amount of work" (or "time") referred to previously is based on three types of measurement

(a) the number of steps

(b) the number of COLON steps (list cell creations)

(c) the number of APPLY steps (function applications) we have found that all three methods of measuring the amount of work give approximately the same results.

Each strategy is represented by a percentage, input to the simulator. For example, a strategy of 10% indicates that each evaluator is allowed to obtain an assistant whenever the work it has done exceeds 10% of the total amount of work the program would have entailed under the totally sequential scheme. This is done in order to meaningfully compare results from programs of different computation lengths (number of steps). If a program is evaluated under a more eager strategy, say 5%, modelling the case where the machine is bigger, we wish to discover the corresponding effect on the performance of the program. Thus 0% represents the maximally parallel scheme and 100% represents the totally sequential scheme.

The simulation has a twofold significance. On one hand we use it to discover the amount of parallelism in programs and on the other it indicates a scheme of machine program organisation suitable for an environment which incorporates parallelism.

The histograms, tables numbered n.1, give us an idea of the run-time profile of each program under the 10% strategy. The vertical axes of tables n.1 show the number of evaluators processing tasks and the horizontal axes show time in terms of computation length. We discovered that the shape of histograms generally remains the same for different strategies so only the 10% case is shown. A histogram indicates the amount of work that can be done in parallel over time. We also compare histograms against our intuition about what programs do.

Parallel programs

In this section we comment on what we have discovered about the method of deriving (by hand) parallel programs from initially sequential ones. First, we have found out that parallelism needs to be expressed (effected) by two language constructs which are introduced into SASL for the purpose of expressing parallel programs. These are the annotation symbol "#" which modifies a primitive operator to a parallel primitive operator. In particular we note that the parallel non-strict primitive operators &# (PAR-AND), |# (PAR-OR) and -># (PAR-CONDITIONAL) express the notion of speculative parallelism where the evaluation of one of their operands is initiated in anticipation that its value might be needed and terminated forcibly when otherwise.

The other parallel construct we found to be needed is the call-by-parallel parameter passing mechanism expressed (forced) by the system function STRICT which operates on a function and its two parameters. It "passes" evaluated the two parameters to the function. Any other more complex case of call-by-parallel was handled by defining an appropriate (user) function in terms of STRICT and auxiliary functions (see chapter two).

Note that our approach does not rely on explicitly creating and synchronising "processes" so that we avoid the problem of the run-time management of parallelism at this level. We have resorted to the use of parallel constructs, taking caution against non-termination, for the purpose of experimentation, of controlling the "grain" of parallelism, avoiding non-useful parallelism and finally since call-by-need cannot be replaced by call-by-parallel (a case of call-by-value) without introducing non-termination (see chapter two).

In order to identify parallelism in a program we proceed from the top (outer) level function definitions to the inner ones. Each time we enter a level the "grain" of parallelism becomes finer. The corresponding program graph identifies the data dependencies. For instance in the example 15 (parsing strings), the graph of the function "conc" indicates a sequential function whereas parallelism was identified at the level of the function "bar" which has a similar structure as "conc" but no prohibiting data dependencies.

Parallelism can be seen from performance graphs to be most enhanced if the program graph is balanced in the sense that the sub-expressions to be evaluated in parallel are of similar complexity. This is enforced when the Divide-and-

In some cases parallelism manifests itself as parallel evaluation of a list's components (when its length is finite). For example expressions of the form

a : b : c .... : ()

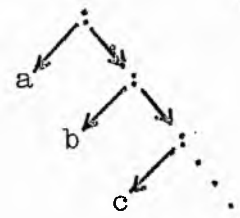



figure 6.1 -program graph of a list evaluation

whose graph, shown in figure 6.1, suggests that the representation of a list by a two field data structure (a cell) gives a rather unbalanced tree of tasks.

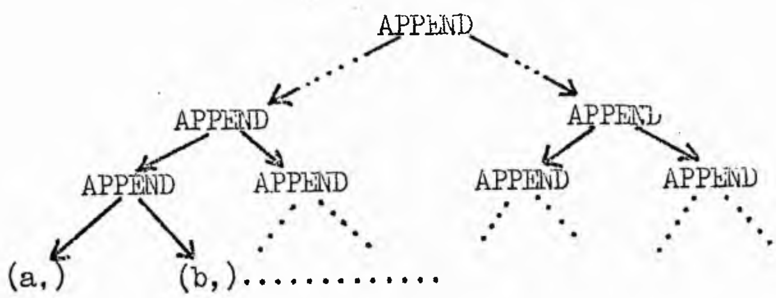



figure 6.2 -transformed program graph

In order to obtain a balanced tree the operator : is replaced by the parallel function APPEND (defined in chapter five), the graph of the transformed expression is shown in figure 6.2. The functionality of the function APPEND requires us to change the components a,b ... into

1-lists (a,), (b,), ..., this is a rather ad-hoc solution. Keller [41] avoids the overhead introduced by the function APPEND by proposing a different data structure to the list cell.

Here a connection with the work of Darlington [45] is apparent. A system of formal derivation of parallel programs from initial sequential ones or from initial specifications of programs is desirable. For example it is interesting to speculate whether the parallel matrix multiplication program (Example 9, chapter five) could be formally derived from an initially sequential one.

The parallel program for the queens (Example 13, chapter four) was obtained by reprogramming where backtracking was eliminated in favour of forward moves. Here we also note a certain inelegance since a path of forward moves which fails to arrive at a solution is represented by the empty list "()" which appears in the output of the program since it is generated. The introduction into SASL of set expressions [35] which evaluate to lists avoids the generation of unwanted components of the output list.

The case of the numerical program for solving a partial differential equation on a rectangular grid (Example 14, chapter five) required reprogramming in order to compute the result of the computation in a "top-down" fashion instead of the "bottom-up" method of the initial sequential

The run-time results

In this section we analyse the results, shown in the appendix, of running the example programs, developed in chapter five, on the simulator we have constructed (see chapter three and four).

The tables 1.2, 2.2, 3.2, 5.2, 9.2, 10.2, 15.2 indicate that the performance of the example programs 1, 2, 3, 5, 9, 10, 15 is related linearly to the strategies of parallelism. Tables 4.2, 7.2, 8.2, 12.2, 13.2, 14.2 show a kind of exponential relationship. This must be due the fact their tree of sub-tasks are well balanced.

The example 6 (computing a zero of a polynomial) was not tested due to lack of real numbers in SASL, though we could have worked with some scaling. Example 11 (generating the partitions of an integer) turns out to be essentially sequential.

The histograms, tables 1.1 - 15.1 give us the profile of the parallel evaluations over time. These agree with our intuitive understanding of what programs do. For example table 5.1 (testing for the bipartite property on an undirected connected graph) where its evaluation can at most decompose into two parallel (sub-) evaluations. The histograms 2.1 (testing a directed graph for a cycle) and 3.1 (computing a vertex where a cycle starts) have a steep end since the completion of some sub-task causes the termination of all other tasks. The histogram of example 15 (parsing strings), table 15.1, indicates that for most of the time there is a single task (sequential evaluation for most of the time) since the sub-tasks terminate rather quickly. This is due to the fact that sub-tasks test the legality of a sub-string of the input string to the parser. Also parallelism is limited since it is only identified with one function, namely "bar", where all the other functions at the same level (grain) as "bar" are sequential. Table 9.1, corresponding to the Sort-by-merge program, indicates that for a large part at the end of the computation there is a single task (the main one) due to the fact that finally two large lists have to be merged to give the result sorted list of numbers. Merging is a sequential "operation". Valleys in the histograms indicate periods of sequential operations. Example 10 (computing a relation in a library) showed a large number of lock steps (see chapter four) due to the fact that all parallel evaluations search two global association lists.

The more or less symmetrical histograms indicate that at the beginning and at the end of computation the number of sub-tasks is low, exponentially increasing (decreasing) in between. This is due to the fact we have a binary tree of sub-tasks and the balancing tends to be good in such case.

Example 11, generating the partitions of an integer, exhibits an interesting point. Under the most eager strategy (0%) it yielded 65% gain over the length of the totally sequential evaluation whereas the application of memo-isation [36] yielded 67%. This strongly suggests that performance gains are obtainable by means other than that of parallelism.

Finally we observe that the maximum number of parallel activities is 18 and although we have experimented with "toy" programs we can speculate that there will be maximum demand upon the resources of a real machine only for a rather limited period.

In table 16 the SPEED UP FACTOR is shown for the maximally parallel strategy (0%). This is related to the PERFORMANCE gain shown in the vertical axes of tables 1.2 - 15.2 calculated as

100 / (100 - PERFORMANCE)

# CHAPTER SEVEN

## Conclusions

The work presented in the previous chapters has focused on two issues. The nature of a parallel implementation of SASL and the amount of parallelism in particular programs exploited by the implementation.

The implementation is based on the SECD implementation of SASL. This has been extended with primitives which handle the interaction of a whole regime of SECD machines, referred to as evaluators. The evaluators combine their effort in processing a single program task. This is possible because a program task decomposes to sub-tasks where each of those may decompose further and so on. A program which simulates a regime of evaluators, an unbounded number of which is assumed, has been constructed.

The evaluation of a program gives rise to a spectrum of behaviours in the simulator each determined by a strategy of spawning. Each strategy of spawning represents a particular degree of parallelism employed during the evaluation of a program. The spectrum varies between a totally sequential computation where just a single evaluator is employed to process a program and a maximally parallel computation where a new evaluator is employed whenever a computation splits into sub-computations. Each behaviour between these extreme cases is characterised by the fact that a new evaluator is employed under a certain constraint. This is imposed by having each evaluator working on some task obtain assistant evaluators after it has performed a certain amount of work and providing it has generated sub-tasks.

The parallelism of a program is investigated by evaluating it under different strategies and noting the corresponding performances. The performance measure is based on the number of evaluator's steps it takes to process a program to completion. Each step is equivalent to the "execution" of an SECD machine instruction. We expect results obtained to hold true for other types of implementations where we have different steps, for example SK machine steps.

We discover by associating a strategy with the degree of parallelism employed in a multi-processor machine, that there is often a linear relationship between the performance of programs investigated and the degree of parallelism. This suggests that in a realistic situation, providing the parallel implementation can be efficiently supported doubling the size of the multi-processor machine approximately halves the run-time.

The parallelism of programs manifests as simultaneous evaluation of the operands of primitive operators and as simultaneous call-by-value on the parameters of functions. The parallel conditional operator gives rise to the notion of an "irrelevant" evaluation where its condition and alternative(s) are evaluated in parallel. The evaluation of the alternative(s) is initiated in anticipation its value might be needed. If it turns out that it is not needed then it must be identified and terminated. Thus in order to extract parallelism from programs the lazy evaluator must be forced to do some work.

There are two problems with replacing call-by-need by call-by-value. On one hand it may introduce non-termination and on the other not all function calls offer the opportunity for useful work to be done in evaluating the actual parameters in parallel. The latter is also true with instances of primitive operators where the evaluations of their operands are ordered by a data dependency or one of them is rather trivial. In both cases parallelism cannot be introduced usefully. The approach we take is to introduce source annotations which mark the primitive operators which are to be interpreted as being parallel. Call-by-value is expressed in terms of primitive operators. We envisage that further work might be able to identify such operators partly automatically but this is, in general not computable. The annotations direct the compiler to produce parallel code (parallel instructions corresponding to parallel operators) which causes the evaluator to generate a tree of tasks. A similar use of functions like "STRICT" directs the simulator to evaluate certain arguments of functions in parallel (call-by-parallel). A task is associated with the operand of a parallel operator. The strategy of spawning, mentioned above, causes assistant evaluators to "take away" tasks, so that when the evaluator comes to process them they are evaluated already.

In order to identify in a program the parallel operators, and simultaneous call-by-values a program is represented as a graph of data dependecies. The definitions of names are used to unfold the graph discovering the data dependecies. In several cases in order to obtain a balanced tree of tasks the original program is transformed to a better parallel program by applying a programming technique known as divide and conquer.

We have gained considerable experience with the parallelism of a variety of programs. The structure of a parallel program seems to be of the following three forms.

1. The Divide and Conquer form where a program's evaluation recursively sub-divides into evaluations of similar complexity, the program for matrix multiplication, developed in chapter four, is an example of a program possessing this form.

2. The speculative form where evaluations are initiated in anticipation their result (value) will be needed. The parser for Lambda Calculus expressions is an example of this form.

3. The evaluation of a program occasionally requires the parallel evaluation of certain sub-expressions before

it continues sequentially, for example the program which tests whether an undirected connected graph is bipartite.

Note that all cases of parallelism concern deterministic programs. The case where parallelism is introduced by non-deterministic constructs has not been dealt with. The introduction of non-determinism enables a certain class of programs to be programmed in (near) applicative style [42]. This notion of parallelism is beyond the scope of the present study.

Parallelism may be extracted from non-numerical as well as numerical problems alike. The example programs investigated cover a wide range of applications.

The final word of course lies with the computer architects. What we have examined here is the "logical" aspects of parallelism, what can be done in parallel and for what programs. There have been a number of proposals for multi-processor machine designs [31,39,40] which set out to support efficiently a notion of parallelism rather similar to the one investigated in the present study. The results of the present research have important implications for such research. It is clear that many algorithms which would not take advantage of such hardware can be transformed into more appropriate forms. It may be that appropriate language constructs would lead to the natural production of parallel programs.

It seems that a risky philosophy of task initiation is almost essential if advantage is to be taken of inherent parallelism and consideration should be given to the efficiency of the killing process for irrelevant computations.

REFERENCES

1. J.Backus, Can programming be liberated from the Von Newmann style? a functional programming style and its algebra of programs CACM 21,8 1978, pp 613-641

2. A.P. Ershov, Mixed Computation: Potential applications and problems for study Theoretical Computer Science vol. 18, 1982, pp 41-67

3. P.H. Welch, Lambda Calculus lecture notes 1977 University of Kent Computing Laboratory

4. D.A. Turner, SASL language manual,CS/79/3 Depart. of Computational Science University of St.Andrews, Fife, Scotland

5. R. Morrison, S-ALGOL language manual, CS/79/1 Depart. of Computational Science University of St.Andrews

6. W. Cambell, An abstract machine for a purely functional language, Tech. Report July 1979 University of St.Andrews Computing Laboratory

7. D.A. Turner, A new implementation technique for applicative languages Software Practice and Experience vol. 9, 1979, pp 111-222

8. R. Burstall & J. Darlington, A transformation system for developing recursive programs, JACM vol. 24, 1977, pp 44-67

9. N. Nilsson, Problem solving methods in Artificial Intelligence, McGraw Hill, 1971

10. Wozencraft & Evans, Notes on Programming Linguistics, MIT 1979

11. D.A. Turner, SASL language manual, CS/75/1 Depart. of Computational Science University of St.Andrews

12. L. Lombardi & B. Raphael, The language LISP: its operation and applications, pp 204-219, MIT Press 1964

13. D.A. Turner, Functional programming and proofs of program correctness, Computing Lab. University of Kent

14. M. O'Donnell, Computing in systems described by equations Lecture Notes in Computer Science vol. 58

15. P. Landin, The mechanical evaluation of expressions Computer Journal vol. 6, 1964, pp 308-320

16. Evans, PAL a language for teaching Programming Linguistics Proc. ACM Nat. Conf. 1968

17. J. McCarthy et al., LISP 1.5 programmers manual MIT Press 1965

18. P.H. Welch, Some notes on the Martin-Lof/Tait proof of the Church-Rosser theorem University of Kent Computing Lab. 1975

19. C.P. Wadsworth, Semantics and Pragmatics of the Lambda Calculus Oxford University D.Phil. Thesis 1971

20. D. Grit & R. Page, Deleting Irrelevant tasks in an Expression oriented multi-processor system TOPLAS vol. 3, no. 1, 1981, pp 49-59

21. P.Henderson & J.H. Morris, A lazy evaluator 3rd. ACM symp. Principles of Programming Languages 1976, pp 95-103

22. D.P. Friedmann & D.S. Wise, "CONS should not evaluate its arguments" Proc. 3rd. int. coll. Automata Languages and Programming Edinburgh 1976

23. A. Mycroft, The theory and practice of transforming call-by-need into call-by-value CSR-88-81, Depart. of Computer Science University of Edinburgh

24. J. Clarke et al., SKIM the S,K,I reduction machine LISP Conference 1980 Stanford

25. C. Hoffmann & M. O' Donnell, Programming with Equations TOPLAS vol 4, no.1, 1982, pp 83-112

26. D.A. Turner, Another algorithm for bracket abstraction
Journal of Symbolic Logic July 1979

27. P. Wegner, Programming languages Information
structures and machine organisation
McGraw Hill 1968

28. W.H. Burge, Recursive Programming
Addison Wesley 1975

29. D.A. Turner, An implementation of SASL TR/75/4
Depart. of Computational Science
University of St.Andrews

30. J. Stoy, Denotational semantics
MIT Press 1980

31. R. Keller et al.,
A loosely-coupled applicative
multi-processor system AFIPS 79 pp 613-622

32. H. Baker & C. Hewitt,
The incremental garbage collection
processes SIGPLAN Notices (ACM) vol. 12,
no. 8, 1977

33. E.W. Dijkstra,
Cooperating sequential processes
Programming Languages pp 43-112
Genuys (Ed.) Academic Press 1968

34. W.Burton & R. Sleep,
Executing a virtual tree of processors
School of Computing Studies,
University of East Anglia, Norwich

35. J. Darlington,
A synthesis of several sorting algorithms
Acta Informatica vol. 11, 1978, pp 1-30

36. D.A. Turner, The semantic elegance of applicative
languages Proc. MIT/ACM Conf. on Functional
programming languages and
computer architecture
Portsmouth, New Hampshire, October 1981

37. P. Treleaven,
Principle components of a Data Flow computer
SRM/216 University of Newcastle upon Tyne
Computing Laboratory 1979

38. D. Michie, Memo-Functions
Experimental Programs 1966-67
Depart. of Artificial Intelligence
University of Edinburgh

39. R. Sleep & W. Burton,
Towards a zero assignment parallel processor
Proc. 2nd. int. Conf 1981 on distributed systems

40. J. Darlington & M. Reeve,
ALICE a multi-processor machine for the parallel evaluation of applicative programs
Proc. ACM/MIT Conf. 1981 on Functional programming and computer architecture.
Portsmouth, New Hampshire, October 1981

41. R. Keller, Some theoretical aspects of applicative multi-processing
Lecture Notes in Computer Science vol. 88, pp 58-74

42. P. Henderson,
A purely functional operating system
Functional programming and its applications
J.Darlington, P.Henderson, D.Turner (eds)
C.U.P 1982

43. J.R. Hindlay et al.,
Introduction to Combinatory Logic
C.U.P 1972

44. J. Reynolds, Definitional Interpreters for higher order programming languages
Proc. 25th ACM Ann. Conf. 1972

45. P. Treleaven et al.,
Combining control flow and data flow
Computer Journal vol. 25, 1982, pp 207-216

# APPENDIX I

TABLE 1.1

<u>program</u> To compute the reachability relation in a directed graph

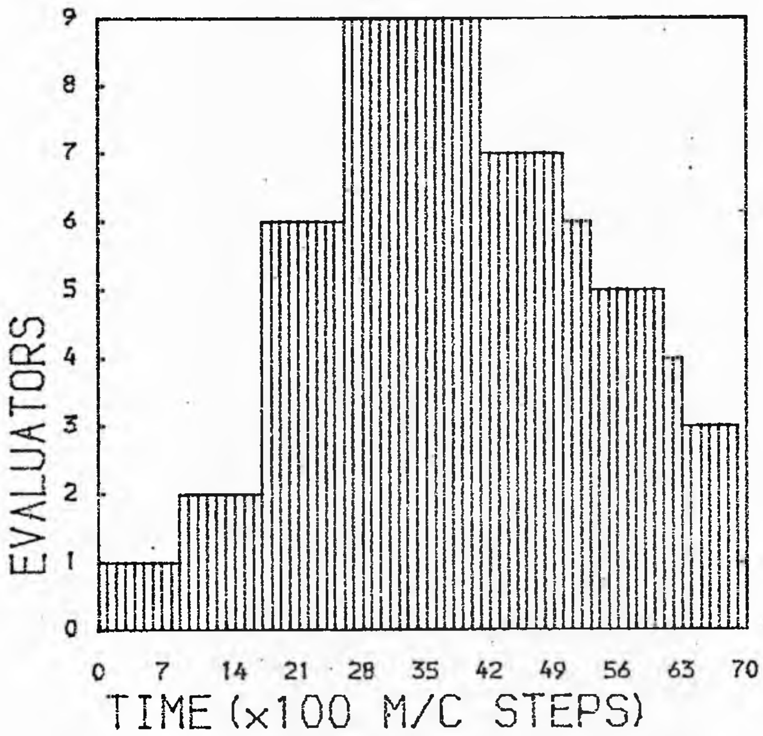



TABLE 1.2

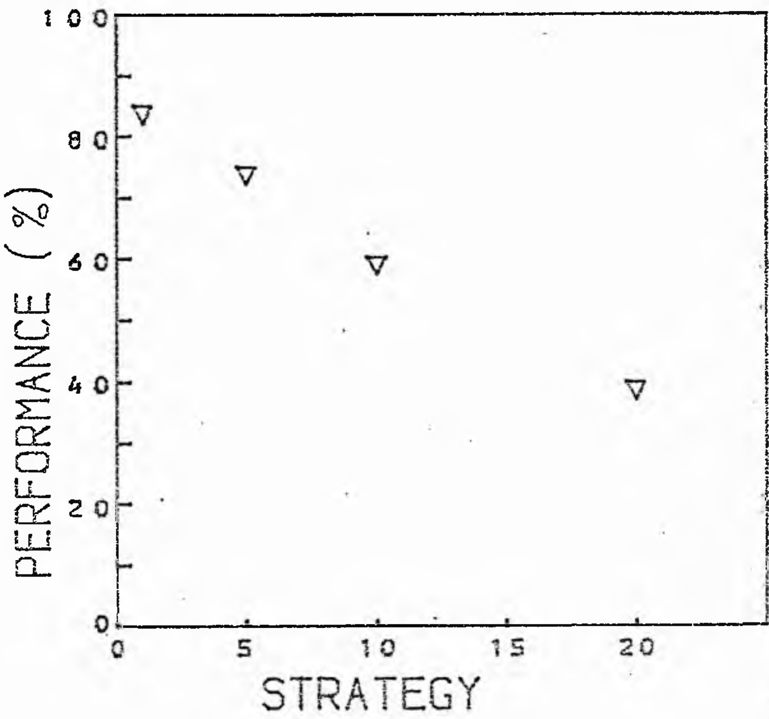


TABLE 2.1

program Testing a directed graph for a cycle

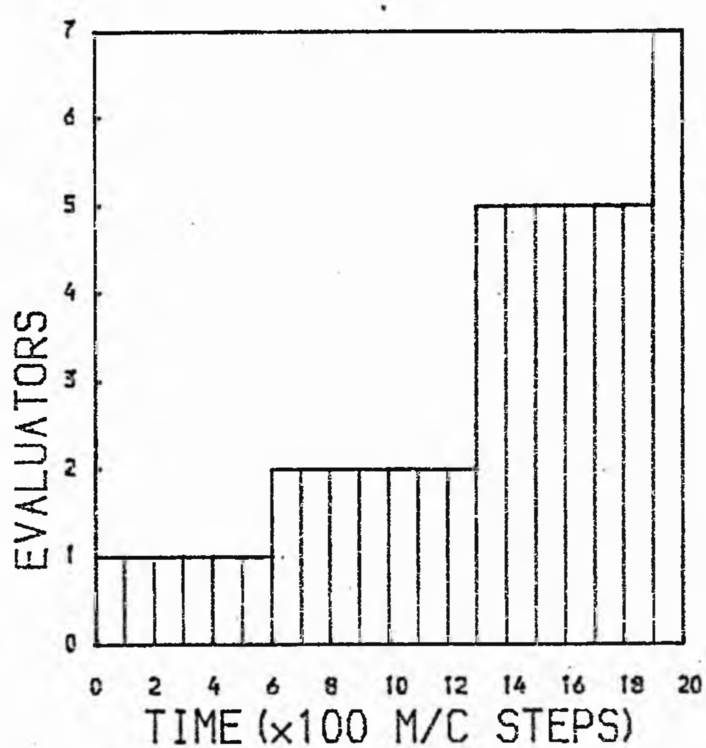



TABLE 2.2

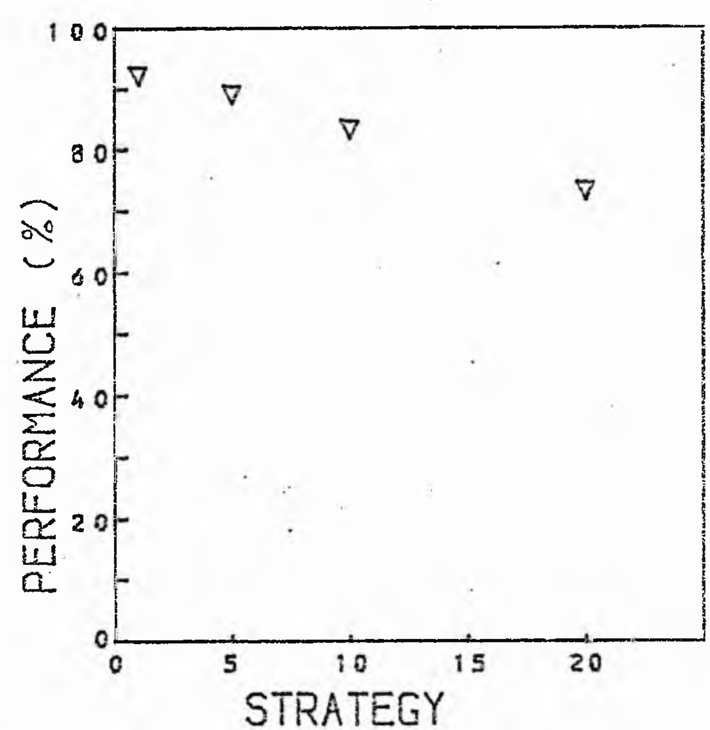
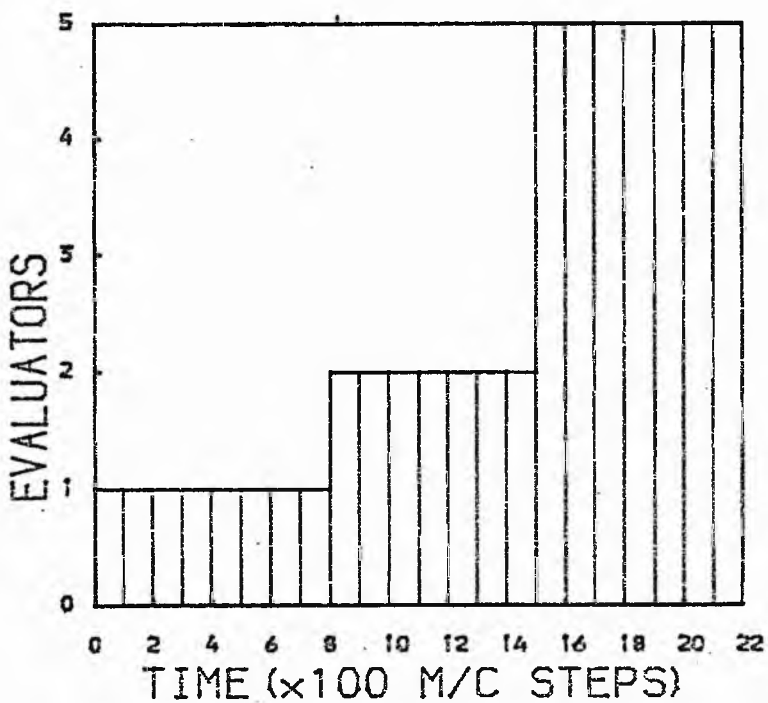


TABLE 3.1

TABLE 3.2

__program__ To compute a cyclic vertex of a directed graph



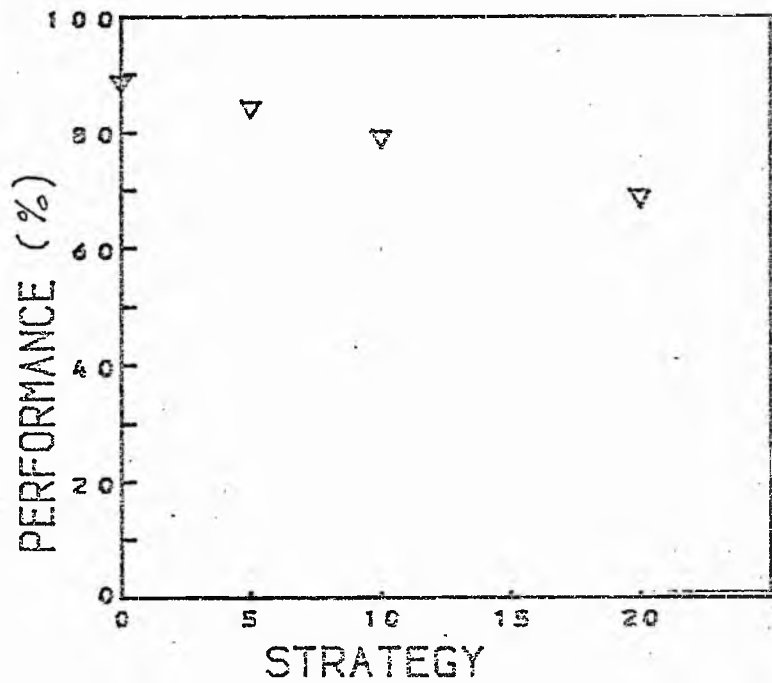
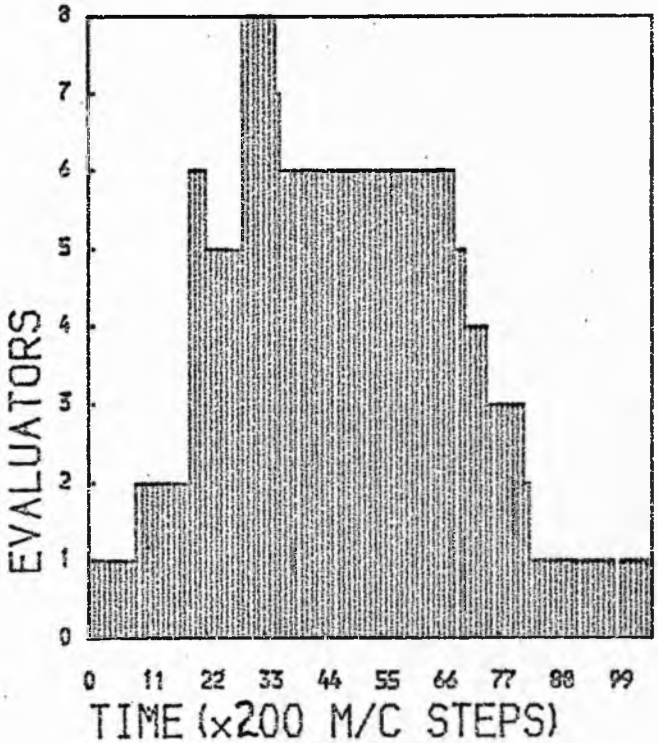



TABLE 4.1

TABLE 4.2

program To compute the terminal vertices of a directed graph



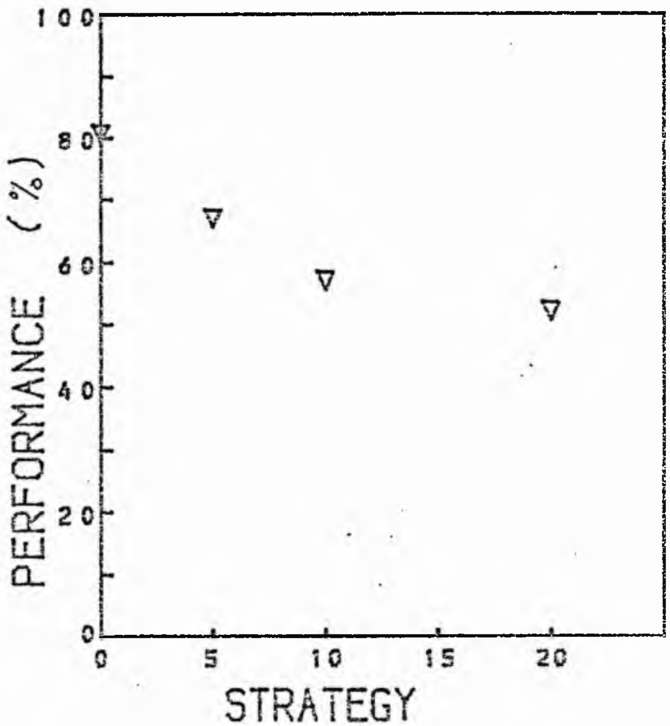
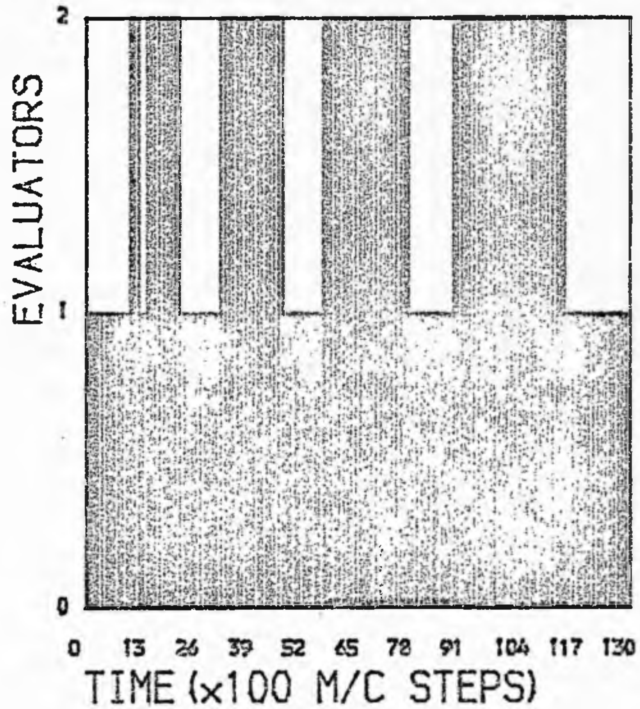



TABLE 5.1

TABLE 5.2

<u>program</u> To test whether an undirected connected graph is bipartite



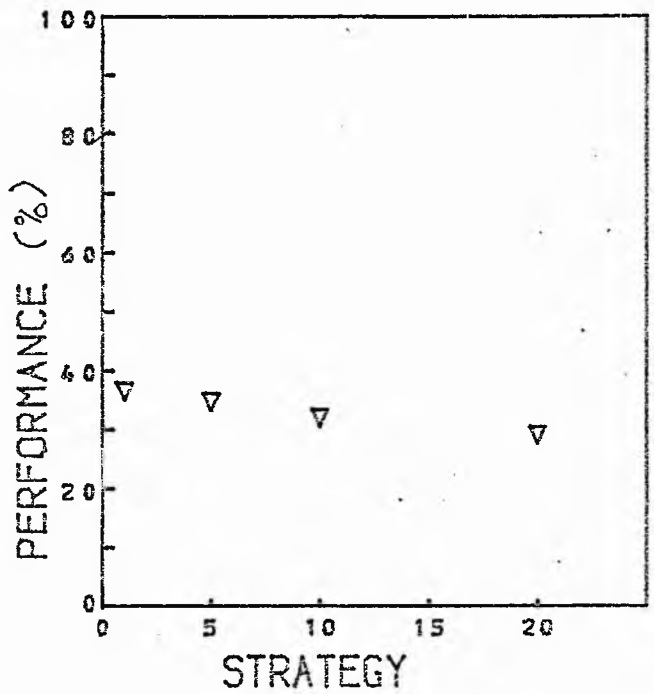
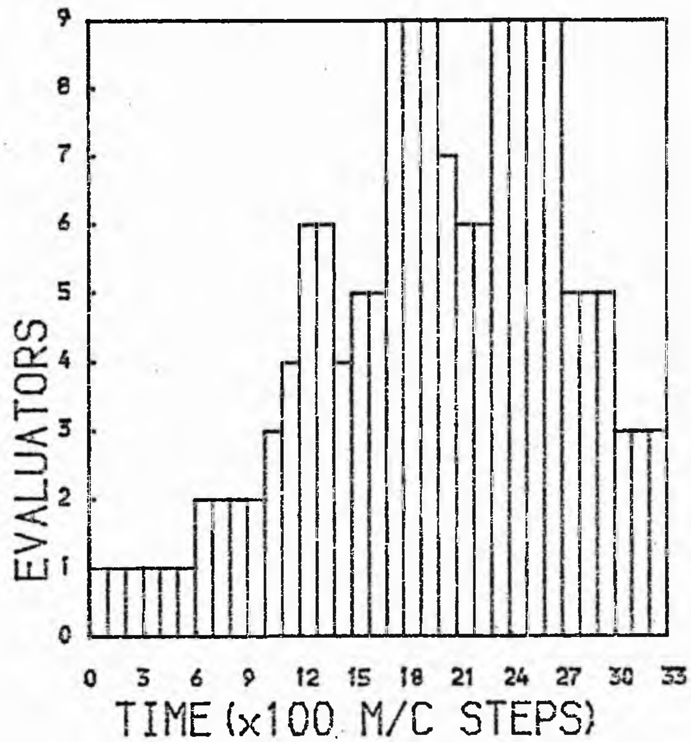
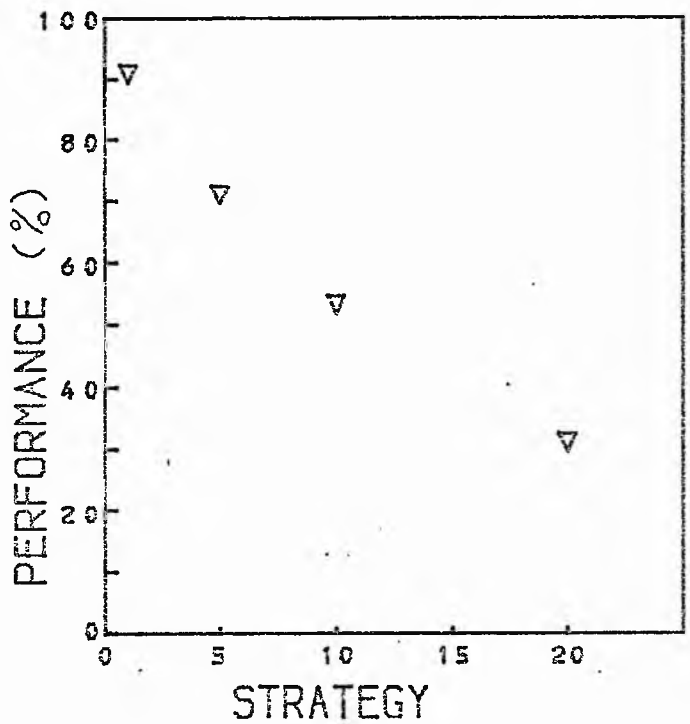



TABLE 7.1

TABLE 7.2

<u>program</u> The Hanoi problem for eight discs



# TABLE 8.1

<u>program</u> 8X8 matrix multiplication

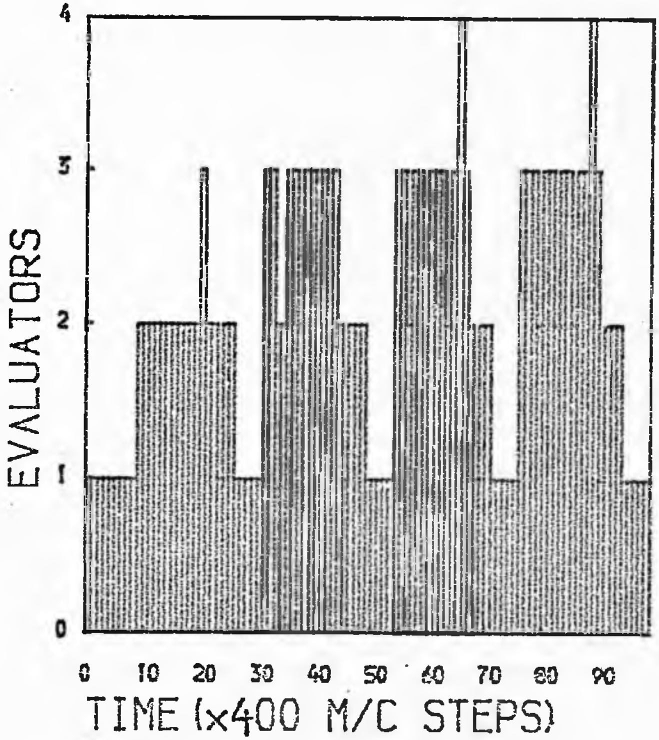


TABLE 8.2

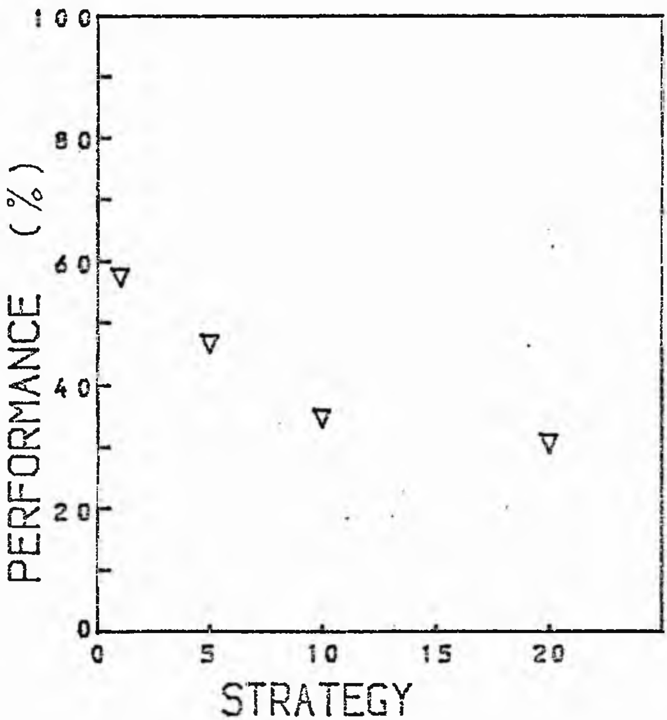


TABLE 9.1

<u>program</u> Merge-sort

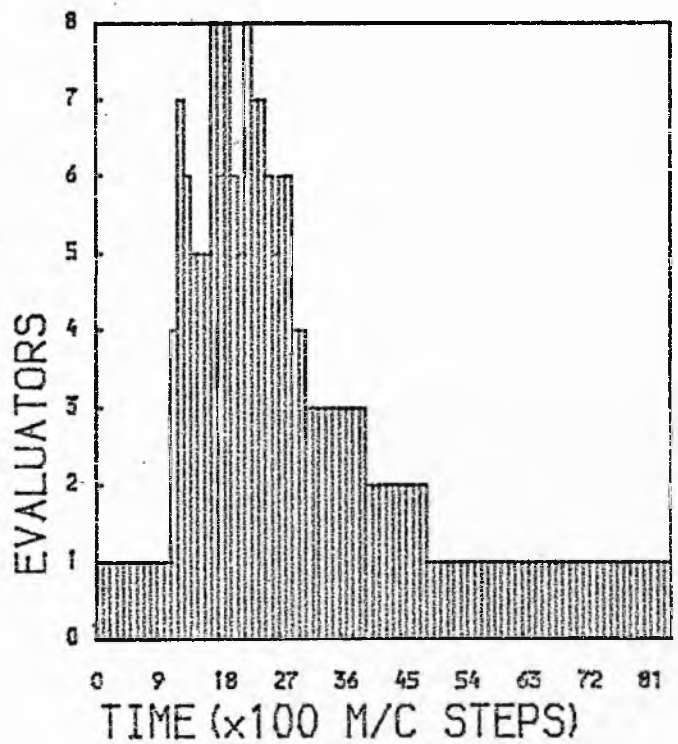



TABLE 9.2

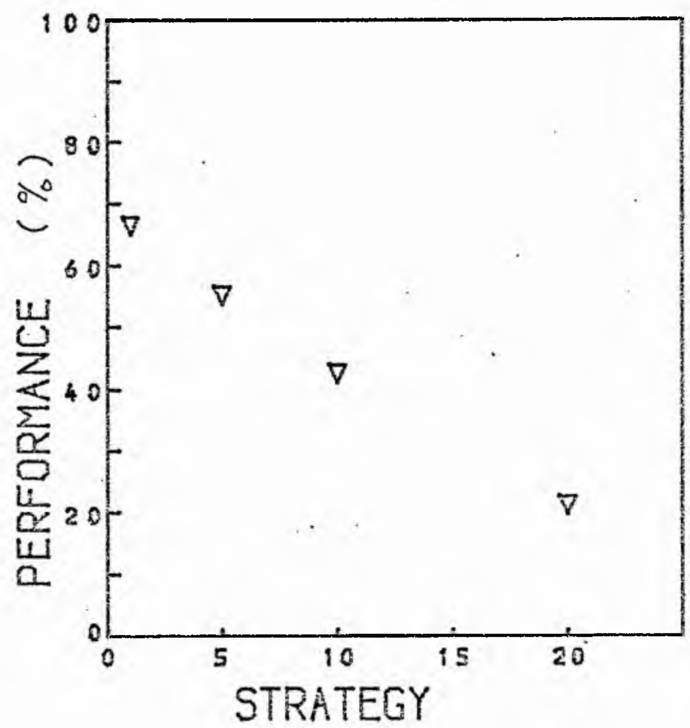
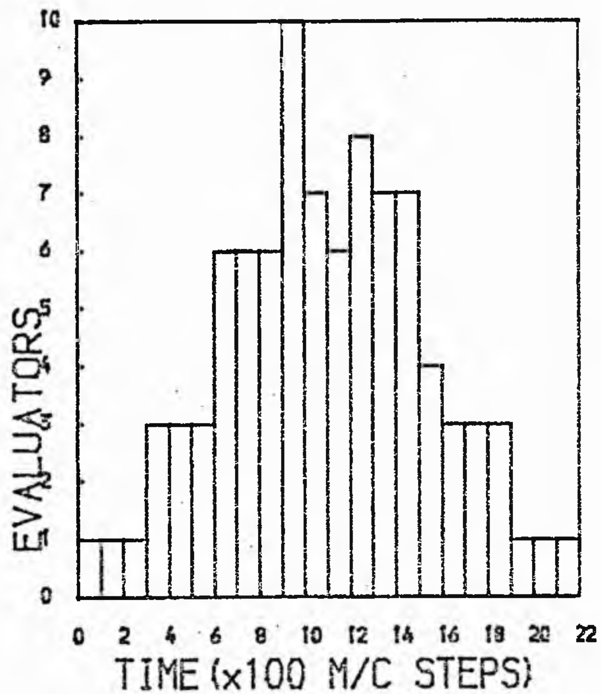


TABLE 10.1

TABLE 10.2

program To compute a relation (association list)

in a data base



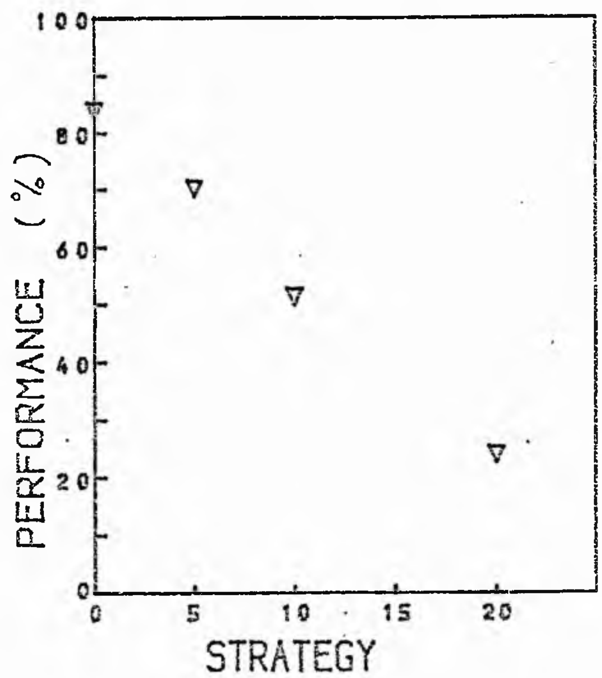
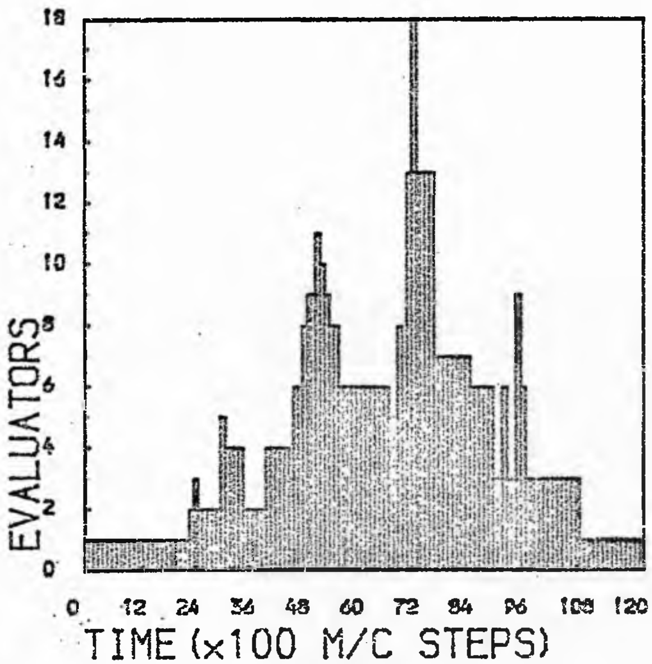



TABLE 12.1

TABLE 12.2

<u>program</u> To compute the permutations of the numbers 1,2,3,4,5



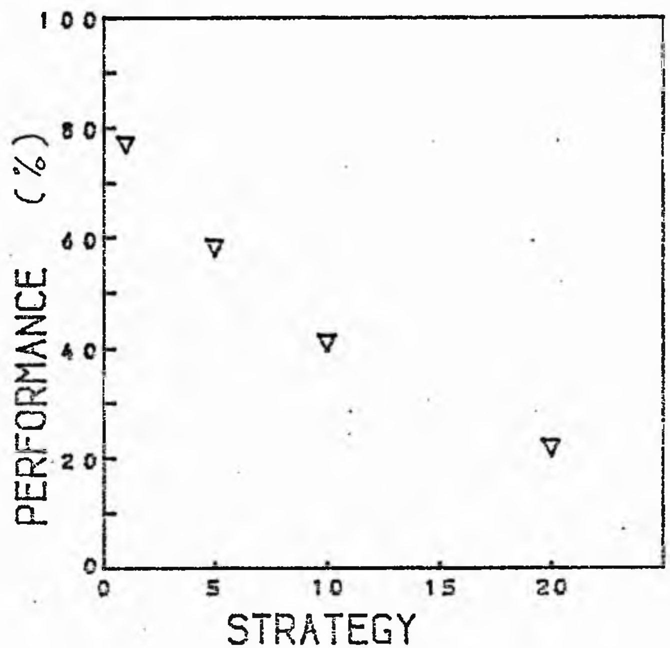
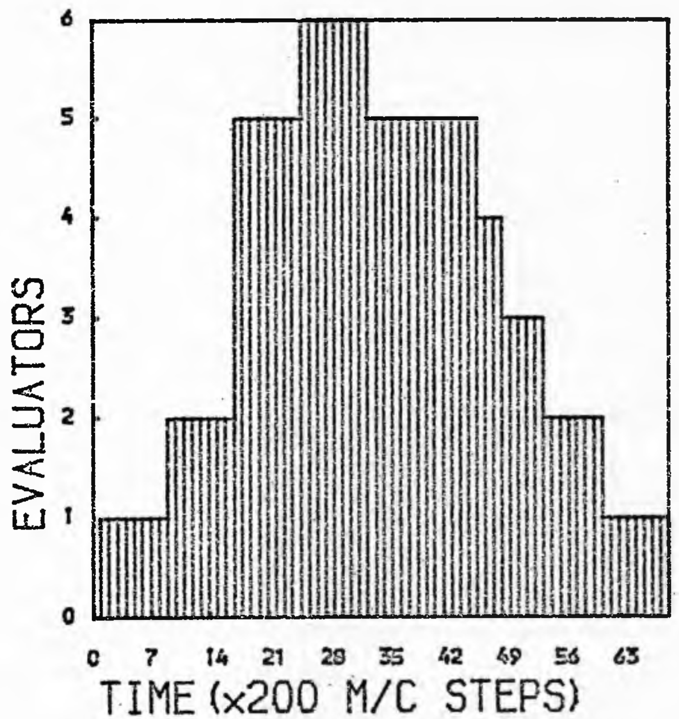



TABLE 13.1

TABLE 13.2

<u>program</u> To solve the queens problem on an 6X6 board



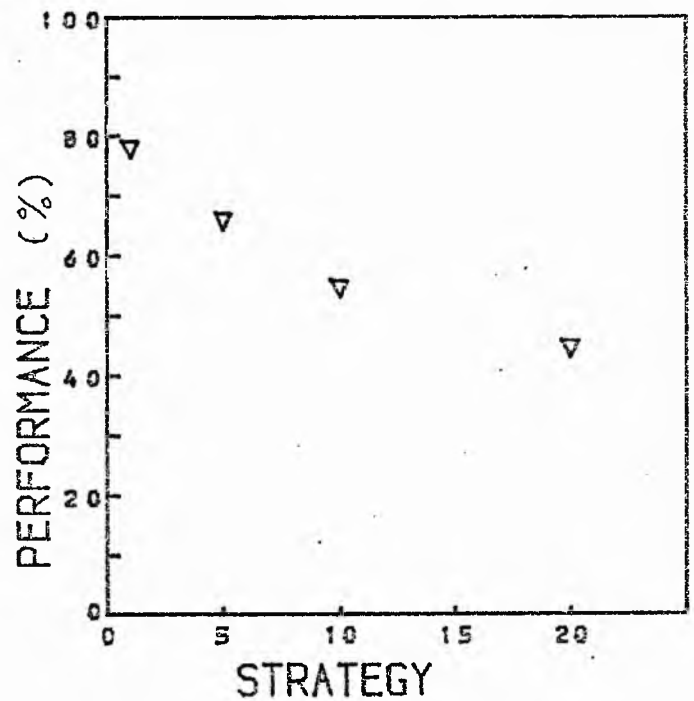



TABLE 14.1

<u>program</u> To numerically solve Laplace's equation by the grid method

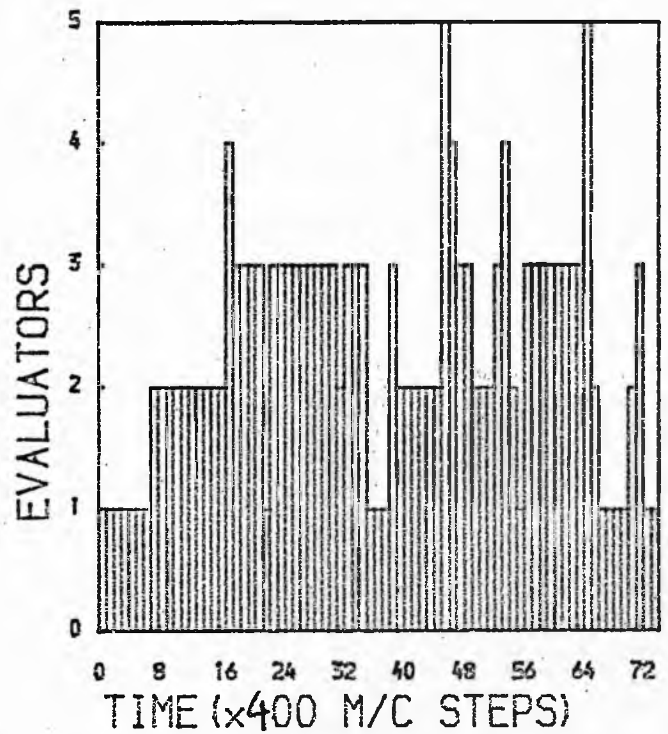



TABLE 14.2

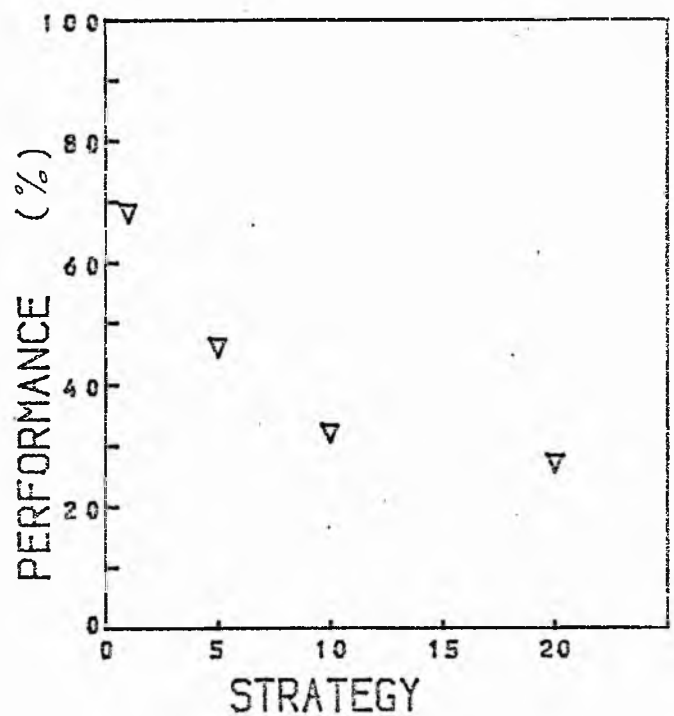
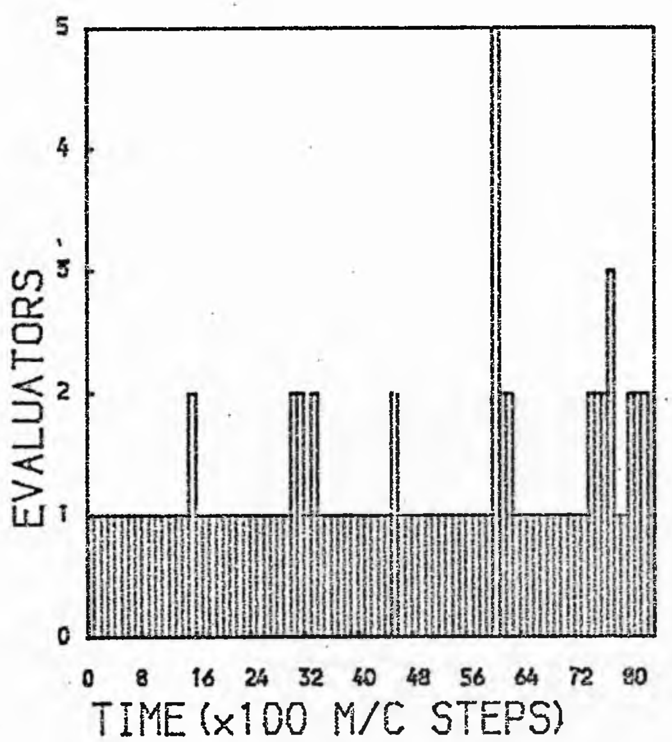
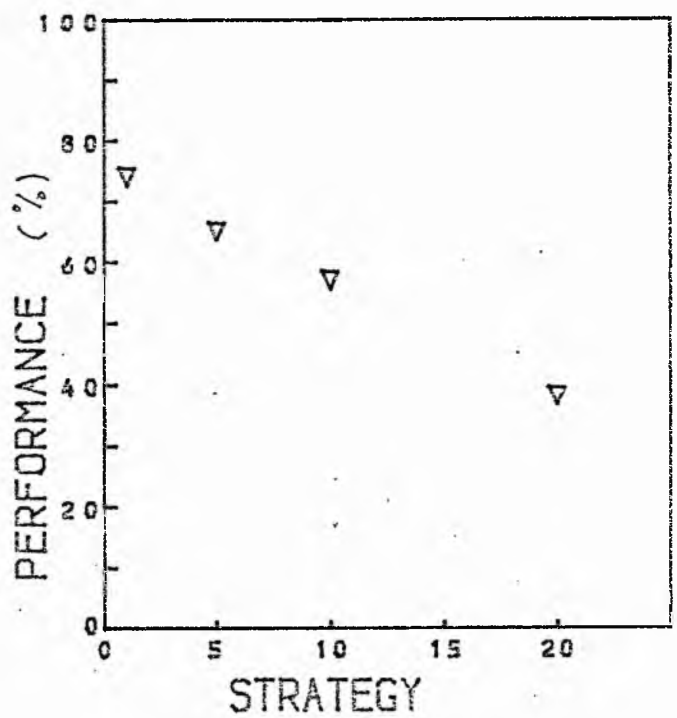


TABLE 15.1

TABLE 15.2

program A parser for Lambda Calculus strings



## TABLE 16

| PROGRAM | SPEED UP FACTOR |
|---|---|
| the reach of a graph | 6 |
| cyclic graph | 14 |
| start of a cycle | 10 |
| terminal vertices | 5 |
| bipartite graph | 3/2 |
| hanoi | 10 |
| matrix product | 5/2 |
| merge-sort | 3 |
| library relation | 6 |
| permutations | 5 |
| 6 queens | 5 |
| laplace grid | 3 |
| parser | 8 |

The speed up factor is related to the PERFORMANCE axis in tables 1.2 - 15.2 by the formula

$$100 / (100 - \text{PERFORMANCE})$$

# APPENDIX II

```
! New SASL ( non-strict with multiple clauses and functional composition )

! Copyright 1979 by William R Campbell and David A Turner.
! apart from the concurrency stuff
! by J Corovessis

! sasl data stuctures

structure id( cstring the.id ; cpntr next.id )
structure num( int the.num )
structure logic( cstring the.bool )
structure char( cstring the.char )
structure cons( pntr hd , tl )
structure const( cpntr the.const )

structure closure( cpntr fn.def , fn.env )
structure suspended( pntr its.val , its.env ; bool lock )
structure overwrite( cpntr susp )
structure coerse( cpntr the.susp )
structure still.in.try( pntr Cons )
structure anothertry( cpntr succ.clos )
structure binding.err
structure repetition( cpntr the.rpt )
structure strict( cstring strict.op )
structure map( cpntr form , body )
structure trys( cpntr clauses , args.so.far )
structure defn( pntr defn.name , its.defn , next.defn , for.show )
structure er.val( cpntr arg )

structure prefix( cpntr prefix.op , e )
structure infix( pntr infix.op , e1 , e2 )
structure pinfix( cpntr parop , a1 , a2 )
structure opstr( cstring mnemonic )
structure cond( cpntr test , left.fork , right.fork )
```

```
structure parcond( cpntr b ; l ; r )

structure dead

structure lf
structure ri
structure nothere( pntr chld )
structure a.file( cfile the.file ; cstring saved.ch ; cpntr prev.file )


! infix operators

let HD = opstr( "HD" )              ; let LISTERR = opstr( "LISTERR" )
let TL = opstr( "TL" )              ; let COND = opstr( "COND" )
let NOT = opstr( "~" )              ; let UNDEF = opstr( "UNDEF" )
let NEG = opstr( "-" )              ; let CHECKLIST = opstr( "CHECKLIST" )


! prefix operators

let APPLY = opstr( "APPLY" )        ; let PLUSPLUS = opstr( "++" )
let APPLYb= opstr( "APPLYb" )       ! if binding FAILS don't terminate computation
                                    ! just return binding.err
let BLOCK = opstr( "BLOCK" )        ; let MINUS = opstr( "-" )
let COLON = opstr( ":" )            ; let TIMES = opstr( "*" )
let DOT = opstr( "." )              ; let DIV = opstr( "/" )
let EQ = opstr( "=" )               ; let REM = opstr( "REM" )
let NE = opstr( "~=" )              ; let GT = opstr( ">" )
let PAROR = opstr( "!" )            ; let GE = opstr( ">=" )
let PARAND = opstr( "&" )           ; let LT = opstr( "<" )
let PLUS = opstr( "+" )             ; let LE = opstr( "<=" )


! flags
```

```
let errorflag := false
let unrecovered := false
let message.given := false
let messages := false
let echo := false
let willtrace := false
let counting := false
let tracing := false
let Trace := 0
let say.hello := true
let cue := false


structure going
structure notgoing


let activity := going
! m/c registers
! and aux regs

let CODE       := nil
let SUB.CYCL   := ""
let ENV        := nil
let DUMP       := nil
let STACK      := nil

let RES.SLOT   := nil
let PS         := nil
let i := 0 ! total of leaf processes
let j := 0 !       of extension ps
let nodes := 0 !   of node ps
let incycle   := false
```

```
i.w := 1
s.w := 0


! lexicals

let lmargin := 0 ! for offside rule
let posn := 0
let ps := 0

let buffer.size = 120
let text.buffer = vector 0 :: buffer.size of ""
let buffer.ptr := 0

let symb := " "
let its.value := nil ! for NAME's and CONSTANT's
let ch := " "
let names := nil ! only one copy of each idemtifier

let remember := false
let remembered := nil ! for detecting repetitions of ids in definition forms

let NAME = "Name"
let CONSTANT = "Constant"
let EOF = "End of File"
let OFFSIDE = "Offside"
let ENDCH = code( 0 )
let O.QUOTE = "??"
let C.QUOTE = "?"
let zero = decode( "0" )
```

```
let input file := s.i   ! initially standard
let fileq := nil
let o.f  := s.o   ! Output file initially standard
let d.f = create("DATA.DAT","s","a","v",512)
let sampl:=0
let sml:=0
let psptr:=0
let ps.no=vector 1::20 of 0


! evaluation

let noi       := 0   ! controls spawns

let SIZE  := 0; let GLOCK := 0
let CELLS  := 0
let STACKDEPTH := 0


! the process stuff

let ring := nil
let last.in.ring := nil


let LEFT = 14; let RITE = ri
structure process( pntr s ; E ;  c ; d ;
                   string sub.cycle    ;
                   pntr res.slot       ;
                   bool spawnon        ;
                   cpntr line          ;
                   *pntr data.dep      ;
                   pntr father         ;
                   pntr next           )
```

```
                         ! stat. vars
                         bool lhs.set ,
                         int size , cells , stackdepth ;
                         int lock.cycles
                       )


! the sasl environment

let the.env := nil
let oldenv := nil ! used in eval for tracing
let FAIL := defn( nil , nil , nil , nil )
let EVALUATED := defn( nil , nil , nil , nil ) ! to flag evaluation of suspensions


! sasl values

let TRUE = logic( "true" )
let FALSE = logic( "false" )
let NL = char( "'n" )
let SP = char( " " )
let NP = char( "'p" )
let TAB = char( "'t" )
let nill = const( nil )

! predeclaration of standard sasl functions

procedure identifier( string word -> pntr )
begin
     let ids := names
     while ids ~= nil and ids( the.id ) ~= word do ids := ids( next.id )
     if ids = nil then
     begin
```

```
                names := id( word , names )
                names
         end
         else ids
end

procedure predeclare( string name )
the.env := defn( identifier( name ) , strict( name ) , the.env , strict( name ) )

predeclare( "number" )  ;  predeclare( "function" )
predeclare( "logical" ) ;  predeclare( "char" )
predeclare( "list" )    ;  predeclare( "code" )
predeclare( "decode" )  ;  predeclare( "letter" )
predeclare( "digit" )


! predeclare internal function for functional composition
!
! compose f g x = f (g x)

let compose := nil

begin
         let c = identifier( "*compose*" )
         let f = identifier( "f" )
         let g = identifier( "g" )
         let x = identifier( "x" )
         let rhs = infix( APPLY , f , infix( APPLY , g , x ) ) ! f (g x)
         let mapping = map( f , map( g , map( x , rhs ) ) )
         the.env := defn( c , suspended( mapping , nil , false ) , the.env , nil )
         compose := the.env( its.defn )
         compose( its.env ) := the.env ! tie knot for recursion
         compose := c
end
```

```
! predeclare internal function for appending lists (++)
!
! append f g = if f = () then g
!                 else (hd f) : append (tl f) g

let append := nil

begin
     let a = identifier( "*append*" )
     let f = identifier( "f" )
     let g = identifier( "g" )
     let rhs = cond(    infix(    EQ , f , const( nil ) ) ,
                        g ,
                        infix(    COLON ,
                                  prefix(    HD , f ) ,
                                         prefix( CHECKLIST ,
                                               infix( APPLY ,
                                           infix( APPLY , a , prefix( TL , f ) ) ,
                                         g ) ) ) )
     let mapping = map( f , map( g , rhs ) )
     the.env := defn( a ,suspended( mapping , nil , false ) , the.env , nil )
     append := the.env( its.defn )
     append( its.env ) := the.env ! tie knot for recursion
     append := a
end


! forward procedures

forward prompt
forward expr( -> pntr )
forward condexp( -> pntr )
forward listexp( -> pntr )
```

```
-- forward opexp( int -> pntr )
-- forward comp( -> pntr )
-- forward simple( -> pntr )
-- forward defs( -> pntr )
-- forward clause( -> pntr )
-- forward rhs( -> pntr )
-- forward formal( -> pntr )
-- forward namelist( -> pntr )
-- forward starter( -> bool )
-- forward terminator( -> bool )
-- forward have( string -> bool )
-- forward mustbe( string )
-- forward syntax( string )
-- forward show text
-- forward getchar
-- forward layout
-- forward nextsymb
-- forward eval conc()
-- forward system( cpntr , cpntr )
-- forward spawn( cpntr , cpntr , cpntr , cstring )
-- forward late( -> bool )
-- forward kill ( cpntr )
-- forward act reg(cpntr)
-- forward lock wait
-- forward declare( pntr , pntr -> pntr )
-- forward lookup( pntr , pntr -> pntr )
-- forward error( string )
-- forward print( pntr )
-- forward show( pntr )
-- forward show prefix( cpntr )
-- forward showenv( pntr , int )
-- forward function id( pntr , bool )
-- forward err1( pntr , pntr -> pntr )
-- forward err2( pntr , pntr , pntr -> pntr )
```

```
forward isval( cpntr -> bool )
forward pathname( -> string )

forward cons( pntr , pntr -> pntr )
forward infix.block()
forward arith.block()
forward apply.block()
forward try.block()
forward select.block()
forward equal.block()
forward prefix.block()
forward none.block()
forward once.block()
forward or.block()
forward monitor()
forward link( pntr , pntr -> pntr)
forward cont.at( cpntr , cstring )


procedure interact
begin

     procedure display.env
     begin
          write "'nDefinitions:'n'n"
          let env := the.env
          while env ~= nil do
          begin
               let pos := 0
               while env ~= nil  and  pos <= 60 do
               begin
                    let name = env( defn.name , the.id )
                    write name
                    pos := pos + length( name )
```

```
                    while pos rem 20 ~= 0 do
                    {     pos := pos + 1 ; write " " }
                    env := env( next defn )
               end
               write "'n"
          end
          write "'n"
     end


     procedure delete( pntr name )
     begin
          let e := the env
          if the env ~= nil do
               if the env( defn name ) = name then
               begin
                    ! overwrite possible guess node
                    the env( defn name ) := nil
                    the env( its defn ) := nil
                    the env := the env( next defn )
               end
               else
               begin
                    let prev := e
                    e := e( next defn )
                    while e ~= nil and e( defn name ) ~= name do
                    {     prev := e ; e := e( next defn ) }
                    if e ~= nil do
                    begin
                         ! overwrite possible guess node
                         e( defn name ) := nil
                         e( its defn ) := nil
                         prev( next defn ) := e( next defn )
                    end
```

```
                end
            if e = nil do write "'n" , name( the.id ) , " not found'n"
        end


        procedure help
        begin
            let f = open( "sasl.help.file" , "a" , 0 )
            while ~ eof( f ) do write read( f ) ;
            write "'n"
        end


        procedure getfile( string pathname )
        begin
            let f = open( pathname , "a" , 0 )
            if f = nullfile then
                write "cannot open " , pathname , " - get ignored'n"
            else begin
                    fileq := a.file( input.file , ch , fileq )
                    input.file := f
                    ch := " "
                    ps := 0
                end
        end

        procedure show.name( pntr name )
        begin
            let e := the.env
            while e ~= nil and e( defn.name ) ~= name do
                e := e( next.defn )
            if e = nil then write name( the.id ) , " not found'n"
            else begin
                    let d := e( for.show )
```

```
                    if d is map or d is trys or d is strict do write "Function'n"
                    show( d )
               end
     end


     ! interact

     let running := true
     while running do
     begin
          prompt
          errorflag := false
          unrecovered := false
          willtrace := Trace > 0
          counting := messages or willtrace

          message.given := false
          for i = 0 to buffer.size do text.buffer( i ) := ""
          buffer.ptr := 0
          o.f := s.o

          tracing := Trace = 1
          lmargin := 0
          posn := lmargin
          oldenv := the.env
          nextsymb
          if symb = "off" then running := false else
          if symb = "help" then help else
          if symb = "definitions" then display.env else
          if symb = "mess" then messages := true else
          if symb = "nomess" then messages := false else
          if symb = "echo" then echo := true else
          if symb = "noecho" then echo := false else
```

```
          if symb = "get" then getfile( pathname ) else
          if have( "trace" ) then
               if symb = CONSTANT and its.value is num then
                    Trace := its.value( the.num )
               else syntax( "Integer Constant" ) else
          if have( "display" ) then
               if symb = NAME then show name( its.value )
               else syntax( NAME ) else
          if have( "delete" ) then
               if symb = NAME then delete( its.value )
               else syntax( NAME ) else
          if have( "def" ) then
          begin
               let d = defs
               if ~ errorflag do
                    the.env := declare( d , the.env )
          end
          else begin
                    let object = expr
                    if ~ errorflag then
                    begin
                         if symb = "to" do
                         begin
                              o.f := create( pathname ,"s","a","v",512)
                              seek( o.f , 0 , 2 )
                         end
                         system( object , the.env )
                    end else if symb = "to" do nextsymb
                    if o.f ~= s.o do close( o.f )
               end
          if message.given do show text
     end
     write "'ngoodbye from PARALLEL SASL'n"
end
```

```
procedure prompt
begin
      if input.file = s.i do
      if say.hello then
         { write "'nhello from PARALLEL SASL'n" ; say.hello := false }
      else write "'nwhat now?'n"
      cue := true
end


procedure pathname( -> string )
begin
      layout
      let pname := ch
      repeat
            getchar
      while digit( ch ) or letter( ch ) or ch = "." or ch = "/" do
            pname := pname ++ ch
      pname
end


! sasl compiler


procedure tab( -> int )
begin
      let t = lmargin
      lmargin := posn
      t
end
```

```
--   procedure untab( int t )
1-   begin
--        lmargin := t
--        if symb = OFFSIDE  and  posn >= lmargin do nextsymb
-1   end
--
--   procedure expr( -> pntr )
1-   begin
--        let t = tab
--        let v := condexp
--        while have( "where" ) do v := infix( BLOCK , defs , v )
--        untab( t )
--        v
-1   end
--
--
--   procedure condexp( -> pntr )
1-   begin
--        let t = tab
--        let v := listexp
--        if have( "->" ) do
2-        begin
--             let alt = if symb = "#" then { nextsymb ; true }else false
--             let leftarm = condexp
--             if symb = ";" do nextsymb
--             let rightarm = condexp
--             v := if alt then parcond( v , leftarm , rightarm )
--                  else cond( v , leftarm , rightarm )
-2        end
--        untab( t )
--        v
-1   end
--
--
```

```
procedure listexp( -> pntr )
begin

      procedure lexp( -> pntr )
      infix(   COLON , opexp( 0 ) ,
                if have( "," ) then lexp else nill )

      let t = tab
      let v := opexp( 0 )
      if have( "," ) do
            v := infix(   COLON , v ,
                              if terminator then nill else lexp )
      untab( t )
      v
end


procedure opexp( int prio -> pntr )
begin
      let oldv := nil
      let v := if have( "~" ) then prefix(  NOT , opexp( 3 ) ) else
                  if have( "+" ) then opexp( 5 ) else
                  if have( "-" ) then prefix(  NEG , opexp( 5 ) )
                  else comb

      if prio < 6 do
      while have( "." ) do v := infix(  DOT , v , opexp( 6 ) )

      if prio < 5 do
      repeat
      {     oldv := v
            v := if have( "*" ) then
                        if symb="#" then{ nextsymb ; pinfix(TIMES,v,opexp(5)) }
                        else infix( TIMES , v , opexp ( 5 ) ) else
```

```
            if have( "/" ) then infix(   DIV , v , opexp( 5 ) ) else
            if have( "rem" ) then infix(  REM , v , opexp( 5 ) )
            else oldv }
while v ~= oldv

if prio < 4 do
repeat
{     oldv := v
      v := if have( "+" ) then
                 if symb="#" then { nextsymb ; pinfix(PLUS,v,opexp(4)) }
                 else infix( FLUS , v , opexp ( 4 ) ) else
            if have( "-" ) then infix(  MINUS , v , opexp( 4 ) )
            else oldv }
while v ~= oldv

if prio < 3 do
repeat
{     oldv := v
      v := if have( "=" ) then
                 if symb= "=#" then{ nextsymb; pinfix(EQ,v,opexp(3)) }
                 else infix( EQ , v . opexp ( 3 ) ) else
            if have( "~=" ) then infix(  NE , v , opexp( 3 ) ) else
            if have( "<" ) then infix(  LT , v , opexp( 3 ) ) else
            if have( ">" ) then infix(  GT , v , opexp( 3 ) ) else
            if have( "<=" ) then infix(  LE , v , opexp( 3 ) ) else
            if have( ">=" ) then infix(  GE , v , opexp( 3 ) )
            else oldv }
while v ~= oldv

if prio < 2 do
while have( "&" ) do
     v := infix( PARAND , v , opexp(2) )

if prio = 0 do
```

```
      begin
           while have( "|" ) do
             v := infix ( PAROR , v , opexp(1) )
           if have( ":" ) do v := infix( COLON , v ,
                                        prefix(CHECKLIST , opexp( 0 ) ) )
           if have( "++" ) do v := infix( PLUSPLUS , v , opexp( 0 ) )
      end
      v
end


procedure comb( -> pntr )
begin
      let v := simple
      while starter do v := infix( APPLY , v , simple )
      v
end


procedure simple( -> pntr )
begin
      let v := nil
      if symb = NAME then
      {     v := its value ; nextsymb } else
      if symb = CONSTANT then
      {     v := const( its value ) ; nextsymb } else
      if have( "(" ) then
      {     v := expr ; mustbe( ")" ) }
      else syntax( "Expression" )
      v
end


procedure defs( -> pntr )
```

```
begin
     let ds := clause
     while have( ";" ) or starter do
     begin
          let d = clause
          if d( defn.name ) = ds( defn.name ) then
               if d( its.defn ) is map and ds( its.defn ) is map then
                    ! construct a list of alternatives from the two clauses
                    ds( its.defn ) := trys( cons( ds( its.defn ) ,
                                                   cons( d( its.defn ) , nil )),
                                              nil ) else
               if d( its.defn ) is map and ds( its.defn ) is trys then
               begin
                    ! add clause d to list of alternatives for ds
                    let list := ds( its.defn , clauses )
                    while list( tl ) ~= nil do
                         list := list( tl )
                    list( tl ) := cons( d( its.defn ) , nil )
               end
               else error( "Inconsistent definition of " ++ d( defn.name , the.id ) )
          else begin
                    ! distinct names
                    d( next.defn ) := ds
                    ds := d
               end
     end
     ds
end


procedure clause( -> pntr )
begin
     let names = namelist
     if have( "=" ) then defn( names , expr , nil , nil )
```

```
        else begin
                if names isnt id do error( "Function Format" )
                remember := true
                defn( names , rhs , nil , nil )
            end
    end


    procedure rhs( -> pntr )
    begin
        let f = formal
        if symb = "=" or f = FAIL then  ! FAIL prevents infinite recursion
        begin
            mustbe( "=" )
            remember := false
            remembered := nil
            map( f , expr )
        end
        else map( f , rhs )
    end


    procedure formal( -> pntr )
    begin

        procedure member( pntr name , remembered -> bool )
        begin
            let p := remembered
            while p ~= nil and p( hd ) ~= name do p := p( tl )
            p ~= nil
        end

        let f := FAIL
        if symb = NAME then
```

```
        begin
            f := its.value
            nextsymb
            if remember do
                if member( f , remembered ) then f := repetition( f )
                else remembered := cons( f , remembered )
        end else
        if symb = CONSTANT then
        begin
            f := const( its.value )
            nextsymb
        end else
        if have( "-" ) then
        begin
            if its.value is num then its.value( the.num ) := -its.value( the.num )
            else error( "Negation" )
            f := const( its.value )
            mustbe( CONSTANT )
        end else
        if have( "(" ) then
        begin
            f := namelist
            mustbe( ")" )
        end
        else syntax( "Formal" )
        if have( "." ) then cons( f , formal ) else f ! structs
    end


    procedure namelist( -> pntr )
    begin

        procedure nlist( -> pntr )
        cons( formal , if have( "," ) then nlist else nill )
```

```

     let n = formal
     if have( "," ) then
          if terminator then cons( n , nill )
          else cons( n , nlist )
     else n
end


procedure starter( -> bool )
symb = NAME or symb = CONSTANT or symb = "{"

procedure terminator( -> bool )
symb = ")" or symb = "?" or symb = OFFSIDE or symb = "where" or symb = ";" or
     symb = "to" or symb = EOF


procedure have( string target -> bool )
if symb = target then { nextsymb ; true }
else false


procedure mustbe( string target )
if unrecovered then
begin
     while symb ~= target and symb ~= "?" and symb ~= EOF do nextsymb
     if have( target ) do unrecovered := false
end
else if ~ have( target ) do syntax( target )


procedure syntax( string target )
if ~ unrecovered do
begin
```

```
        errorflag := true
        unrecovered := true
        message.given := true
        write "'nSyntax: ", target, " expected where ", symb, " found in:'n'n"
end


procedure show.text
begin
        let p := buffer.ptr
        let lines := 2
        ! find start of last two lines in the circular text buffer
        repeat
        {     p := if p = 0 then buffer.size else p - 1
              lines := if text.buffer( p ) = "'n" then lines - 1 else
                          if text.buffer( p ) = "" or p = buffer.ptr then 0
                          else lines }
        while lines > 0
        ! write out those lines
        repeat
        {     p := ( p + 1 ) rem buffer.size
              write text.buffer( p ) }
        while p ~= buffer.ptr
        message.given := false
end


! lexical analysis routines ( procedure identifier above )

procedure getchar
begin
        ch := if eof( input.file ) then ENDCH
              else read( input.file )
        if echo and input.file ~= s.i do write ch
```

```
        buffer.ptr := ( buffer.ptr + 1 ) rem buffer.size
        text.buffer( buffer.ptr ) := ch
        ps := if ch = "'n" then 0 else
              if ch = "'t" then ( ps div 8 + 1 ) * 8
              else ps + 1
        if message.given and ( ch = "'n" or ch = ENDCH ) do show.text
end


procedure layout
begin
        while ch = " " or ch = "'n" or ch = "'t" do getchar
        posn := ps
end




procedure nextsymb
begin

        procedure read..word( string first -> string )
        begin
              let name := first
              getchar
              while letter( ch ) or digit( ch ) or ch = "_" do
              {     name := name ++ ch ; getchar }
              name
        end


        procedure try( string s )
        begin
              symb := ch
```

```
        getchar
        if symb = s( 1 | 1 ) and ch = s( 2 | 1 ) do
        begin
             symb := s
             getchar
             if symb = "!!" then
             begin
                  while ch ~= "'n" and ch ~= ENDCH do getchar
                  nextsymb
             end else
             if symb = "()" do
             begin
                  symb := CONSTANT
                  its value := nil
             end
        end
   end


   procedure a character( -> pntr )
   begin
        getchar
        let c = char( ch )
        getchar
        c
   end


   procedure a string( int unmatched -> pntr )
   begin
        getchar
        if ch = "'n" then { error( "Unclosed String" ) ; nil } else
        if ch = O.QUOTE then cons( char( ch ) , a.string( unmatched + 1 ) ) else
        if ch = C.QUOTE then
```

```
            if unmatched > 0 then
                cons( char( ch ) , a string( unmatched - 1 ) )
            else { getchar ; nil } ! the closing quote
        else cons( char( ch ) , a string( unmatched ) )
end


procedure numeral( -> pntr )
begin
        let n := decode( ch ) - zero
        repeat
            getchar
        while digit( ch )
        do n := n * 10 - zero + decode( ch )
        num( n )
end


procedure try id( string word )
begin
        let v = case word of
              "true" : TRUE
              "false" : FALSE
              "nl" : NL
              "np" : NP
              "tab" : TAB
              "sp" : SP
              "def" , "where" , "rem" , "help" , "get" , "to" ,
              "new" , "off" , "display" ,
              "mess" , "nomess" , "definitions" ,
              "echo" , "noecho" ,
              "delete" , "trace" : nil
              default : identifier( word )
```

```
        symb := if v is id then NAME else
                if v = nil then word
                else CONSTANT
        its.value := v
end


! nextsymb

layout
if posn < lmargin then symb := OFFSIDE else
case ch of
"%"       : { symb := CONSTANT ; its.value := a.character }
O.QUOTE   : { symb := CONSTANT ; its.value := a.string( O ) }
ENDCH     : if fileq = nil then symb := EOF
            else begin
                      input.file := fileq( the.file )
                      ch := fileq( saved.ch )
                      fileq := fileq( prev.file )
                      if cue do prompt
                      nextsymb
                 end

"-"       : try( "->" )
"+"       : try( "++" )
">"       : try( ">=" )
"<"       : try( "<=" )
"~"       : try( "~=" )
"|"       : try( "||" )
"("       : try( "()" )
default   : if letter( ch ) then try.id( read.word( ch ) ) else
            if digit( ch ) then
            {    symb := CONSTANT ; its.value := numeral }
            else { symb := ch ; getchar }
```

```
        cue := false
end


! sasl evaluation




procedure conS( pntr hd , tl -> pntr )
begin
        CELLS := CELLS + 1
        cons( hd , tl )
end



procedure head( pntr x -> pntr )
if x is cons then x( hd ) else
err1( HD , x )

procedure tail( pntr x -> pntr )
if x is cons then x( tl ) else
err1( TL , x )


 procedure bool.val( bool v -> pntr )
 if v then TRUE else FALSE




 procedure basic( pntr func, arg -> pntr )
 case func( strict.op ) of
```

```
--   "number"   : bool.val( arg is num ) ,
--   "char"     : bool.val( arg is char ) ,
--   "logical"  : bool.val( arg is logic ) ,
--   "function" : bool.val( arg is closure or arg is trys or arg is strict )
--   "list"     : bool.val( arg = nil or arg is cons ) ,
--   "decode"   : if arg is num then
                          char( code( arg( the.num ) ) )
--                  else err2( APPLY , func ,arg ) ,
--   "code"     : if arg is char then
--                        num( decode( arg( the.char ) ) )
--                  else err2( APPLY , func ,arg ) ,
--   "letter"   : if arg isnt char then FALSE
--                  else
1-                  begin
--                        let c = arg( the.char )
--                        bool.val( "a" <= c and c <= "z" or
--                                  "A" <= c and c <= "Z" )
-1                  end ,
--   "digit"    : if arg isnt char then FALSE
--                  else
1-                  begin
--                        let c = arg( the.char )
--                        bool.val( "0" <= c and c <= "9" )
-1                  end ,
--   default    : err2( APPLY , func ,arg )
--
--
--  procedure isval( cpntr saslobj->bool)
--  case true of
--
--  saslobj is cons ,
--  saslobj = nil  ,
--  saslobj is closure ,
--  saslobj is trys ,
```

```
saslobj is num ;
saslobj is char ;
saslobj is ar val ;
saslobj is logic          true

default                   false
     structure Print
     structure main
     structure stack( pntr top , rest ; int dp )
     structure dump( cpntr Code ; cstring Sub.cycl ; cpntr Env , Dump )


     procedure INCYCL
          incycle := true
     procedure OUTCYCL
          incycle := false


     procedure popstack(->pntr)
     if STACK = nil then { write "stack underflow"; nil }  else
     begin
          let result := STACK( top )
          STACK      := STACK( rest )
          if STACK = nil do write "STACK IS NIL'n"
          if result is nothere do result(chld.c):=dead
          result
     end

     procedure pushstack( pntr item ) ! no stack limit assumed
     begin
          STACK := stack( item , STACK , STACK( dp ) + 1 )
          if STACK( dp ) > STACKDEPTH do STACKDEPTH := STACK( dp )
     end
```

```
procedure save.cont( cstring sub.cycl )
   DUMP  := dump( CODE , sub.cycl , ENV , DUMP )


procedure trace.ps
{
 show( CODE )
 write "'n" ,"++",SUB.CYCL,"'n"
}

procedure load.cont
begin
      let next.C = DUMP( Code )
      if next.C is dead or next.C is main then
          if STACK(top) is suspended then
          { CODE  := coerse(popstack);ENV :=nil;SUB.CYCL :="coerse"
            } else
          {
            RES.SLOT(top)      := STACK(top)
            CODE               := next.C
            SUB.CYCL           := DUMP( Sub.cycl )
            incycle            := false
          } else
       {    CODE       := next.C
            SUB.CYCL   := DUMP( Sub.cycl )
            ENV        := DUMP( Env )
            DUMP       := DUMP( Dump)
       }
end

procedure cont.at( cpntr code ; cstring sub.cycl )
begin
```

```
        incycle := false

        CODE     := code
        SUB CYCL := sub cycl
    end


    procedure sizeup(cntr ps)
    {     ps(size):=ps(size)+1
          let father=ps(father)
          if ps(line)=LEFT then
               if father(lhs set) do sizeup(father)
          else ! a RITE or MAIN
          if father~=ps then sizeup(father)
          else ! it is top
          if ps(size)>sampl  do act reg(ps)
    }


    procedure locksup( cntr ps )
    { let father=ps(father)
      if  ps=father then ps(lock cycles):=ps(lock cycles)+1 else
      if  ps(line)=RITE then locksup(father) else
      if  father(lhs set) do locksup(father)
    }

! assumes CODE , ENV , DUMP , STACK ,
!         , SUB CYCL , RES SLOT
! as globals
```

```
procedure eval.conc
begin
      ! register the size of the computation
      ! measure it  in terms of APPLY m/c-instructions


      ! behaviour of a computation

      case true of


      CODE is const:
      begin
            pushstack( CODE( the const ) )
            load.cont
      end


      CODE is id:
      begin
            let val = lookup( CODE , ENV )

            if val is suspended then
          {  cont.at( coerse( val ) , "coerse" ) ; INCYCL   }
            else { pushstack( val ) ; load.cont  }
      end




      CODE is cond:
      case SUB.CYCL  of
      "none":
             begin
```

```
              save cont( "once" )
                cont.at( CODE( test ) , "none" )
          end


     default :
      !"once":
              if STACK( top ) isnt logic then
              {
                  let a = popstack
                  pushstack( arr1( COND , a ) )
                  load cont
              } else

              { let a = popstack
                cont.at( if a = TRUE then CODE( left.fork )
                         else CODE( right.fork ) , "none"
                       )
                 }



    CODE is parcond
    case SUB.CYCL of

    "none":
    { pushstack(nil)
      spawn(CODE(b),ENV.STACK,"right")
      pushstack(suspended(CODE(l),ENV,false))
      cont.at(CODE,"data.wait")
      nodes :=nodes+1
    }

    "data.wait":
```

```
{ let a  = STACK(top) ; let b  = STACK(rest,top)
  case b of

  TRUE:
       if a is nothere then OUTCYCL
       else{ a :=popstack;b :=popstack;pushstack(a);load.cont
             PS(lhs.set):=false }
  FALSE:
       { a:=popstack;a:=popstack;cont.at(CODE(r),"none") }
  default:if b is er.val then
          { a:=popstack;a:=popstack;pushstack(arr1(COND,b))
            PS(lhs.set):=false;load.cont } else monitor     }

default: write "'nnever"



CODE is map or CODE is trys:
begin
     pushstack( closure( CODE , ENV ) )
     load.cont
end


CODE is prefix:
prefix.block


CODE is infix:
infix.block

CODE is pinfix:
arith.block
```

```
CODE is coerse:

if CODE( the.susp , lock ) = true then lock.wait else
if CODE( the.susp , its.env ) = EVALUATED then
begin
     pushstack( CODE( the.susp , its.val ) )
     load.cont
end  else
begin
     CODE( the.susp , lock ) := true
     DUMP :=
       dump( overwrite( CODE( the.susp ) ) , "overwrite" , ENV , DUMP )
     ENV := CODE( the.susp , its.env )
     cont.at( CODE( the.susp , its.val ) , "none" )
end


CODE is overwrite :
if STACK( top ) is suspended then
begin
     let val = popstack
     save.cont( "overwrite" )
     cont.at( coerse( val ) , "coerse" )
   INCYCL
end  else
begin
       ! must be suspended
     CODE( susp , its.val ) := STACK( top )
     CODE( susp , its.env ) := EVALUATED
     CODE( susp , lock    ) := false
     load.cont
end
```

```
      CODE  is Print :

      begin
           let obj = popstack
           print( obj )
      end


       default :
       begin
          pushstack( CODE )
            write "'n oh-no!'n"

            activity := notgoing
       end




  end


  procedure reverse.op
  {

      case CODE( infix.op ) of

      LT : CODE( infix.op ) := GT
      LE : CODE( infix.op ) := GE
      GT : CODE( infix.op ) := LT
```

```
      GE : CODE( infix op ) = LE
      default : {}


       let temp = CODE( e1 )
       CODE( e1 ) := CODE( e2 )
       CODE( e2 ) := temp
  }


  procedure infix block
       case CODE( infix op ) of

       APPLY : apply block


       APPLYb : apply block




       BLOCK :
            begin ! e2 where e1
                 ENV  := declare( CODE( e1 ) , ENV )
                 cont at( CODE( e2 ) , "none" )
            end

       COLON :
            begin
                 let result = cons( suspended( CODE( e1 ) , ENV , false  ) ,
                                    suspended( CODE( e2 ) , ENV , false  )   )
                 pushstack( result )
                 load cont
            end
```

```
DOT
    { let code= infix(  APPLY ,
                                  infix(  APPLY , compose ,
                                                             CODE( e1 )  ) ,
                                  CODE( e2 )  )
      cont.at( code , "none" ) ; INCYCL  }


PLUSPLUS
         { let code = infix(  APPLY ,
                                  infix(  APPLY , append ,
                                                             CODE( e1 )  ) ,
                                  CODE( e2 )  )
           cont.at( code , "none" ) ; INCYCL  }


EQ,NE  equal.block

PAROR,PARAND  or block


default   ! CODE is arith  operation or relation

        case  SUB.CYCL  of
        "none"
               begin
                     save cont( "once" )
                     cont at( CODE( e2 ) , "none" )
               end

       "once"
              if STACK( top ) is suspended then
              begin
```

```
                    let right.opd = popstack
                    save.cont( "once" )
                    cont.at( coerse( right.opd ) , "coerse" )
                  INCYCL
               end  else
               begin
                    let sofar = popstack
                    if  sofar isnt num then
                  { let er = err2( CODE(infix.op),CODE(e1),sofar )
                    pushstack( er ) ; load.cont
                  } else
                    begin
                         pushstack( sofar )
                         save.cont( "twice" )
                         cont.at( CODE( e1 ) , "none" )
                    end

               end

     default : "twice"

              if STACK( top ) is suspended then
              begin
                   let left.opd = popstack
                   save.cont( "twice" )
                   cont.at( coerse( left.opd ) , "coerse" )
                 INCYCL
              end  else

              if STACK( rest , top ) is suspended then
              begin
                   let e1 = popstack
                   let e2 = popstack
                   save.cont( "twice" )
```

```
                    pushstack( e1 )
                    reverse.op
                    cont.at( coerse( e2 ) , "coerse" )
                 INCYCL
              end  else
              begin
                    let val1 = popstack
                    let val2 = popstack
                    if  val1 isnt num or val2 isnt num then

                    !   check again val1  since it may come from a process
                    !   ie we went into "twice" not via "once"
                 { let er = err2( CODE(infix.op) , val1 , val2 )
                    pushstack( er ) ; load.cont
                 }  else
                    begin
                         let a = val1( the.num )
                         let b = val2( the.num )
                         let result = case CODE( infix.op )  of
                                      PLUS  : num( a + b )
                                      MINUS : num( a - b )
                                      TIMES : num( a * b )
                                      DIV   : num( a div b)
                                      REM   : num( a rem b )
                                      GT    : bool.val( a  >  b )
                                      GE    : bool.val( a >=  b )
                                      LT    : bool.val( a  <  b )
                                      LE    : bool.val( a <=  b )
                                      default : nil  !  never occurs
                         pushstack( result )
                         load.cont
                    end ! of arith  relation or operation
              end    ! of default, ie not infix
```

```
procedure link( pntr previous.args , latest.arg -> pntr )
if previous.args = nil then conS( latest.arg , nil )
else conS( previous.args( hd ) ,
          link( previous.args( tl ) , latest.arg ) )

procedure apply.block
    case SUB.CYCL of

    "none" : none.block

    "once" : once.block

  "binding" :
          begin
              let arg     = popstack
              let formal = popstack

              if formal is id then begin
                                   ENV := defn( formal , arg , ENV ,
                                                           nil   )
                                   pushstack( ENV ) ! nested ?
                                   load.cont
                               end  else


              if formal is const then
              begin
                  let next = infix( EQ , formal( the.const ) , arg )
                  pushstack( formal( the.const ) )
                  pushstack( arg    )
```

```
                save.cont( "from.equal" )
                cont.at( next , "twice" )
              INCYCL
           end  else
           if formal is repetition then
           begin
                let itsval = lookup( formal( the.rpt ) , ENV )
                let next = infix( EQ , formal( the.rpt ) , arg )
                pushstack( itsval )
                pushstack( arg    )
                save.cont( "from equal" )
                cont.at( next , "twice" )
              INCYCL
          end  else
          if   arg is suspended then
          begin
               pushstack( formal )
               save.cont( "binding" )
               cont.at( coerse( arg ) , "none" )
             INCYCL
          end  else
          if   arg  is cons then  ! formal is cons too
          begin
               pushstack( formal( tl ) )
               pushstack( arg    ( tl ) )
               pushstack( formal( hd ) )
               pushstack( arg    ( hd ) )

               save.cont( "frombinding" )
             cont.at( CODE , SUB.CYCL ) ; INCYCL
          end  else
          begin
               pushstack( FAIL )
               load.cont
```

```
                    end

                end
   "frombinding"    ! sure there is more binding to do on the STACK
                    ! return FAIL to fromb-outer or to b-done or
                    ! set ENV to env and go directly to b since
                    ! know there is more , no need to use the DUMP
                    ! may think of fromb as sub-part of binding
                begin
                    let env = popstack
                    if  env = FAIL then begin
                                          let throw := popstack
                                              throw := popstack
                                              !  the remaining binding
                                          pushstack( FAIL )
                                              !  to the "binding"-outer
                                          load cont
                                      end  else
                    begin
                         ENV  = env
                         cont at( CODE , "binding" )
                       INCYCL
                    end
               end


  "binding done":
                begin
                    let env = popstack
                    if  env = FAIL and CODE( infix op ) = APPLY then
                                      begin
                                          let rator = popstack
                                          let r = err2( APPLY , rator ,
                                                        CODE( e2 )  )
```

```
                                        pushstack( r )
                                      load.cont
                                   end  else
                    if env = FAIL and CODE( infix.op ) = APPLYb then
                                   begin
                                        let throw = popstack
                                        ! the rator who FAILED
                                        pushstack( binding.err )
                                        load.cont
                                   end  else
                    begin
                         let rator = popstack
                         let newcode = rator( fn.def )( body )
                         cont.at( newcode , "none" )
                         ENV  := env
                    end
               end  ! of applying a single def clause



"post.eval"   ,
"afterbinding",
"try.done"    ,
      "try": try.block



    "twice":
           if STACK( top ) is suspended then
           begin
                let rand  = popstack

                save.cont( "twice")
                cont.at( coerse( rand ) , "coerse" )
```

```
                    INCYCL
                end  else

                begin
                     let rand  = popstack
                     let rator = popstack
                     if rator is cons then
                         if rand isnt num then
                       { pushstack( err2( APPLY , rator , rand )     )
                         load cont
                       }  else
                         begin
                              pushstack( rator )
                              pushstack( rand  )
                              cont at( CODE , "select" )
                            INCYCL
                         end  else
                         if rator is strict then
                         begin
                              let r = basic( rator , rand )
                            pushstack( r )

                              load cont
                         end  else
                         { pushstack( err2( APPLY , rator , CODE( e2 ) )    )
                         load cont
                         }

                end
  "select": select block

 "from equal":
                   begin
                        let result of EG = popstack
```

```
                    if  result of EQ = TRUE then pushstack( ENV )
                    else pushstack( FAIL )
                    load cont
                end

    !  "from eval"
       default:
                begin
                    let unsuspended = popstack ! must be list
                    let index       = popstack
                    if  unsuspended isnt cons then

                    { pushstack(  err2( APPLY , unsuspended , index )    )
                    load cont
                   }  else

                    begin
                        pushstack( unsuspended )
                        pushstack( index       )
                        cont at( CODE , "select" )
                      INCYCL
                    end
              end



  procedure try block
  case SUB CYCL of

  "try":
       begin
           let arglist    = popstack
           let ithclause = popstack
           pushstack( ithclause      )
```

```
            pushstack( arglist( tl ) )

            pushstack( ithclause( form ) )
            pushstack( arglist  ( hd   ) )
            save.cont( "afterbinding" )
                                         ! comeback with the env
            cont.at( CODE , "binding" ) ;
          INCYCL
        end
  "afterbinding":
               begin
                    let env = popstack
                    if  env = FAIL then begin
                                             let throw := popstack
                                                 throw := popstack ! arglist
                                                 !  and ithclause
                                             pushstack( nil )   ! for "try.done"
                                             load.cont
                                         end  else
                    begin
                         let restarg = popstack
                         let claus   = popstack

                         if  restarg = nil then
                             if claus( body ) is map then
                             begin
                                  pushstack( anothertry( claus ) )
                                  !  a flag to try.done so as to return a
                                  !  new trys of this clause , the remaining
                                  !  thus throwing away the unsuccessfully
                                  !  tried onces
                                  load.cont
                             end  else
                             begin
```

```
                                pushstack( conS( claus( body ) , env ) )
                                load cont
                            end else

                        begin
                            let exp := claus( body )
                            let leftovers := restarg

                            while leftovers ~= nil do
                            begin
                                exp := infix( APPLYb , exp ,
                                              leftovers( hd )          )
                                leftovers := leftovers( tl )
                            end

                            let flag.to.done = still.in.try( conS( exp , env))
                            pushstack( flag.to.done )
                            load.cont
                        end
                    end
                end


"try done":
        begin
            let result.of.try = popstack
            let it = result.of.try
            if it is still.in.try then ! ie implementing CURRYing
                { let newcode = it( Cons )( hd )
                  let newenv  = it( Cons )( tl )
                  save cont( "post.eval" )
                  ENV := newenv
                  cont at( newcode , "none" ) } else
              { let clausess      = popstack ! not tried
```

```
let arglist        = popstack   ! for new trys
if it is anothertry then
begin
    let throw  = popstack  ! the closure
    let ok.but = it( succ.clos )
    let get.on = conS( ok.but , clausess )
    let newtrys = trys( get.on , arglist )
    pushstack( newtrys )
    load.cont
end  else
if it = nil then   !  try failed
!  try next clause
   if clausess  ~= nil then
   begin
        pushstack( arglist ) ! for new trys
        pushstack( clausess( tl ) )
        pushstack( clausess( hd ) )
        pushstack( arglist         )

        save.cont( "try done" )
        cont.at( CODE , "try" )
      INCYCL
   end  else  ! all clauses have failed
   begin
        let rator = popstack
        pushstack( err2( APPLY , rator , CODE(e2) ) )
      load.cont
   end  else  ! is not nil it
begin  !  ok
   let throw = popstack  !  the closure
   let newcode = it( hd )
   let newenv  = it( tl )

   cont.at( newcode , "none" )
```

```
                        ENV      = newenv
                    end     }
                end  !  of the applying multiple def clause

    "post.eval"
                begin
                    let res = popstack
                    if  res is binding.err then
                    begin   ! try remaining
                        pushstack( nil ) ! for try.done
                        cont.at( CODE , "try.done" )  ! instead of dumping twice
                      INCYCL
                    end  else ! it is the result of a computation


                    begin
                        let rest     = popstack
                        let arglist = popstack
                        let clos     = popstack


                        if res is closure then
                        pushstack( trys( clos( fn.def , clauses ) , arglist ) )
                        else pushstack( res )

                        load.cont
                    end

                 end

        default:
                begin
                    write "'nshould never occur'n"
                    activity := notgoing
```

```
                end






procedure select block
begin
     let index := popstack
     let list  = popstack
     if index( the.num ) > 1 then
     begin
          let obj = list( tl )
          if  obj is suspended then
          begin

               ! throw list head
               index := num( index( the.num ) - 1 )
               pushstack( index  )
               save.cont( "from.eval" )

               cont.at( coerse( obj ) ,  "coerse" )
             INCYCL
          end  else
          if obj = nil then
          {  pushstack( err2( APPLY , obj , index )  )
           load.cont
          }  else
          begin
               pushstack( list( tl ) )  ! obj = list( tl )
               index( the.num ) := index( the.num ) - 1
               pushstack( index  )
               cont.at( CODE , "select" )
```

```
                INCYCL
            end
      end  else  ! ie <= 1


      if index( the num ) = 1 then
      begin
            let obj = list( hd )
            if obj is suspended then
          {  cont at( coerse( obj ) , "coerse"  )  ; INCYCL  }
            else { pushstack( obj ) ; load cont  }
     end  else  ! ie < 0
     {  pushstack( err2( APPLY , list , index ) )
        load cont
     }
end



procedure equal block
! a recursive instruction of the m/c
case  SUB CYCL  of

"none":
         begin
                save cont( "once" )
                cont at( CODE( ei ) , "none" )
         end




"once":
        begin
```

```
            save cont( "twice" )
            cont at( CODE( e2 ) , "none" )
        end

  "twice"
        case true of


        STACK( top ) is suspended
        begin
            let val1 = popstack
            save.cont( "twice" )
            cont at( coerse( val1 ) , "coerse" )
          INCYCL
        end


        STACK( rest , top ) is suspended:
        begin
            let val1 = popstack
            let val2 = popstack
            pushstack( val1 )  ! here val1 val2 get swopped
            save cont( "twice" )
            cont at( coerse( val2 ) , "coerse" )
          INCYCL
        end

        default:
        begin
            let val1 = popstack
            let val2 = popstack
            if val1 is er.val or val2 is er.val then
            { pushstack(err2(CODE(infix.op),val1,val2)); load cont } else
            if val1 is num and val2 is num  then
```

```
2-         {
--          let result = if CODE( infix op ) = EQ then
--                       val1( the.num) = val2( the.num ) else
--                       val1( the.num) ~=val2( the.num )
--          pushstack( if result = true then TRUE else FALSE )
--          load.cont
-2         }  else
--         if val1 is char and val2 is char then
2-         {
--          let result = if CODE( infix op ) = EQ then
--                       val1( the.char ) = val2( the.char )else
--                       val1( the.char ) ~= val2( the.char )
--          pushstack( if result = true then TRUE else FALSE )
--          load.cont
-2         }  else
--         if val1 is logic and val2 is logic then
2-         begin
--              let result = if CODE( infix op ) = EQ then
--                           val1 =   val2  else
--                           val1 ~=  val2
--              pushstack( if result = true then TRUE else FALSE )
--              load.cont
-2         end  else
--
--         if val1 = nil  and val2 = nil  then
2-         begin
--              pushstack( if CODE( infix op ) = EQ then
--                         TRUE  else FALSE      )
--              load.cont
-2         end  else
--         if val1 is cons and val2 is cons then
2-         begin
--              let newoprd1 = val1( hd )
--              let newoprd2 = val2( hd )
```

```
                    pushstack( val2( t1 ) )
                    pushstack( val1( t1 ) )
                    pushstack( newoprd2 )
                    pushstack( newoprd1 )
                    save.cont( "cond_recurs" )

                    cont.at( CODE , "twice" )   !  recurs for new
                  INCYCL
                                                ! operands from the
                                                ! beginning
               end  else
               if ( val1 is strict or val1 is closure or val1 is trys ) and
                  ( val2 is strict or val2 is closure or val2 is trys ) then
               {  let res = err2( CODE( infix.op ) , val1 , val2 )
                pushstack( res    )
                load.cont
               }  else
               begin
                    pushstack( if CODE( infix.op ) = EQ then FALSE
                                            else TRUE
                                  )
                    load.cont
               end
          end


default:   !  "cond_recurs"
          begin
               let hd.result = popstack
               if (hd.result=TRUE and CODE(infix.op)=EQ)or
               (hd.result=FALSE and CODE(infix.op)=NE  )then
               {  cont.at( CODE , "twice" )   ! ie go on now to
                  INCYCL
                } else                        ! compare the tails
```

```
            ! stop the recursion ( comparing )
            begin
                let throw := popstack
                throw     := popstack ! since the hd result is FALSE
                                      ! or error on comparison
                                      ! the tails must return this
                pushstack( if CODE(infix.op)=EQ then FALSE else TRUE )
                load.cont    ! just as in atoms
            end
        end




procedure prefix.block
    case SUB.CYCL of
    "none" :
           begin
               save.cont( "once" )
               cont.at( CODE( e ) , "none" )
           end

    default :
           if STACK( top ) is suspended then
           begin
                let throw = popstack
                save.cont( "twice" )
                cont.at( coerse( throw ) , "coerse" )
              INCYCL
           end  else
           begin
                let res  := popstack
                case CODE( prefix.op ) of
```

```
                    HD          res := head( res )
                    TL          res :=  tail( res )
                    NOT         if res = TRUE then  res := FALSE else
                                if res = FALSE then res := TRUE  else
                                res := erri( NOT , res )
                    NEG         if res is num then
                                res := num( - res( the.num )  ) else
                                res := erri( NEG , res )
                  CHECKLIST :   if res isnt cons and res ~= nil do
                                res :=  erri( LISTERR , res )
                   default :

                              res    := erri( CODE( prefix.op ) , res )


                   if res is suspended then
                   {  cont at( coerse( res ) , "coerse" )
                    INCYCL
                   }     else { pushstack( res ) ; load.cont }

              end    ! ie not prefix


procedure none.block
begin
      save cont( "once" )
      cont at( CODE( e1 ) , "none" )
end



procedure once.block
if STACK( top ) is suspended then
```

```
1-  begin
--        let rator = popstack
--        save.cont( "once" ) ! come back here
--        cont.at( coerse( rator ) , "coerse" )
--        INCYCL
-1  end  else
1-  begin
--        let rator = popstack
--        if   rator is strict or rator is cons then
2-        begin
--             pushstack( rator )
--             save.cont( "twice" )
--             cont.at( CODE( e2 ) , "none" )
-2        end  else
--        if rator is closure then
--             if rator( fn.def ) is map then
2-             begin  ! a single def clause
--                  pushstack( rator )  ! for error report
--                  pushstack( rator( fn.def )( form ) )
--                  let rand = if CODE( e2 ) isnt suspended
--                             then suspended( CODE( e2 ) , ENV , false )
--                             else CODE( e2 )
--                  pushstack( rand )
--
--                  ENV := defn( APPLY , conS( rator , CODE( e2 ) ) ,
--                               rator( fn.env )     , nil         )
--                  save.cont( "binding done" )
--                                             ! comeback with the env
--                  cont.at( CODE , "binding" )
--                INCYCL
-2             end  else  !  multiple clauses
2-             begin
--                  let arg  = if CODE( e2 ) isnt suspended
--                             then suspended( CODE( e2 ) , ENV , false  )
```

```
                  else CODE( e2 )
           let claus    = rator( fn.def ): clauses )
           let this.try = claus( hd )
           let arglist = link( rator( fn.def )( args.so.far ) ,
                               arg    )

           pushstack( rator )
           pushstack( arglist )   ! in case need of another trys
           pushstack( claus( tl ) )!  throwing away the hd if no
                                       !  match or partial found
           pushstack( this.try      )
           pushstack( arglist       )

           ENV  = defn( APPLY , conS( rator , CODE( e2 ) ) ,
                        rator( fn.env )       , nil            )
           save.cont( "try.done" )
           cont.at( CODE , "try" )                     ! comeback with
         INCYCL
                                                       ! an env
     end  else   ! ie not closure

     if   rator is trys then
     begin
           pushstack( closure( rator , ENV ) )
           cont.at( CODE , "once" )   ! ie "once" once more
         INCYCL
     end  else                        ! not a m/c cycle
     if rator is binding.err then
     begin
           pushstack( rator )
           load.cont
     end  else
     {  pushstack( err2( APPLY , rator , CODE( e2 ) )    )
      load.cont
```

```
-2            )
--
-1   end
--
--
--   procedure got( cpntr val ->bool )
--   if STACK(.top ) = val or STACK( rest , top ) =
--   else false
--
--
--   procedure arith.block
--   case SUB.CYCL of
--
--   "none".
1-   begin
          nodes:=nodes+1
          pushstack(nil)
          spawn(CODE(a1),ENV,STACK,"right")
          pushstack(suspended(CODE(a2),ENV,false))
          cont.at(CODE,"data.wait")
-1   end
--
--   "data.wait"
1-   begin
          let a:=STACK(top);let b:=STACK(rest,top)
          case true of
--
--        a is er.val, b is er.val
2-        begin
               a:=popstack;b:=popstack
               pushstack(err2(CODE(carop),b,a))
               load.cont(PS(lhs.set)):=false
-2        end
--
```

val then true

```
        isval(a) and isval(b)
        if a is num and b is num then
        begin
             let r=case CODE(parop) of
                   PLUS num(a(the num)+b(the num))
                   TIMES num(a(the num)*b(the num))
                   default: if a(the num)=b(the num) then TRUE
                           else FALSE
             pushstack(r) load cont PS(lhs set) =false
        end else
        begin
             a :=popstack;b :=popstack
             pushstack(arr2(CODE(parop),b,a))
             load cont;PS(lhs set) =false
        end

        default :
        monitor
   end

   default:write "'nbad"


   procedure or block
   case  SUB.CYCL  of

   "none" :  ! set up a or/and-node

           { nodes := nodes + 1
             pushstack( nil )
             spawn( CODE( e1 ) , ENV , STACK , "right" )
             pushstack(suspended(CODE(e2),ENV,false))
             cont at( CODE , "data wait" )  }
```

```
"data wait" ! the behaviour of or/and-node
            ! if enough info do logic
            if ( CODE( infix.op ) = PAROR and got( TRUE )  )  or
            (  CODE( infix.op ) = PARAND   and got( FALSE)  )  then
            {
               let a  = popstack;a  = popstack
               pushstack( if CODE( infix.op ) = PAROR then TRUE
                                else FALSE
                              )
               load.cont
               PS(lhs.set) =false ! don't expect to be clocked up
            } else
            if isval(STACK(top)) and isval(STACK(rest.top)) then
            { let a =popstack;let b=popstack
              if a isnt logic or b isnt logic then
              pushstack(err2(CODE(infix.op),a,b)) else
              pushstack(if CODE(infix.op)=PARAND then TRUE else FALSE)
              load.cont
              PS(lhs.set) =false
            } else

            ! do monitoring and/or spawning
            monitor
default: write "'nNEVER"


procedure monitor
{ let a=STACK(top);let b=STACK(rest.top)
  if ~(a is nothere and b is nothere) do
      if a is suspended then
          if isval(b) then
          {STACK(top) =b;spawn(a,nil,STACK(rest),"right")}
          else ! b is nothere
```

```
              if PS(spawnon) and late do
              {spawn(a,nil,STACK,"left");let adj =b(chld,size)
               b(chld,size):=adj-nol;b(chld,spawnon):=true}
          else ! a isnt suspended
          if b is suspended then spawn(b,nil,STACK(rest),"right")
          else
          if b isnt nothere do PS(lhs set):=true
              ! ie b is val

      OUTCYCL }

procedure act reg(cpntr p)
{ let act.ps :=0
  let i :=p
  repeat { if i(sub.cycle)~="data.wait" do act.ps:=act.ps+1
           i:=i(next) }
  while i~=p
  ps.no(psptr):=act.ps;psptr:=psptr+1
  sampl:=sampl+sml
  if psptr > 20 do
  { psptr:=1;for i=1 to 20 do
             {output d.f, ps.no(i)," ",ps.no(i):=0}
  }
}

! end of my stuff ***********

! ++:++++++++++++++++++++++++++++++++



procedure system( cpntr input.exp ; input.env )
begin
     procedure processor( cpntr ps )
```

```
if ps(c) is main or ps(c) is dead or ps(sub.cycle)="dead" then
kill(ps) else
begin

    ! load context

    PS          := ps
    STACK       := ps( s )
    ENV         := ps( E )
    CODE        := ps( c )
    DUMP        := ps( d )

    SUB.CYCL := ps( sub.cycle )
    RES.SLOT := ps( res.slot )


    STACKDEPTH    := ps( stackdepth   )
    CELLS         := ps( cells )

    ! a kick



    ! only if it is active
    if SUB.CYCL ~= "data wait" do
    if CODE isnt overwrite and CODE isnt coerse and
    CODE isnt Print and
    ~(CODE is infix and CODE(infix.op)=APPLYb) do
    sizeup(PS)

    SIZE          := ps( size ) ! may have been side-effected
                                ! by sizeup
```

```
! iterate on number of kicks ?

 INCYCL
while incycle do
begin
        eval.conc
 end

  ! save the context

  ps( s ) := STACK
  ps( E ) := ENV
  ps( c ) := CODE
  ps( d ) := DUMP

  ps( sub.cycle ) := SUB.CYCL
  ps( res.slot  ) := RES.SLOT

  ! no need to save size ,taken care by
  ps( stackdepth  ) := STACKDEPTH
  ps( cells       ) := CELLS



end






 output o.f , "'nspawn when no of cycles
 noi := readi
 output o.f , "'n"
```

sizeup

is "

```
let first = process(
                        stack( nil , nil , 0 ) ,
                        input env ,
                        input exp ,
                        dump( Print , "print" , nil ,
                              dump( main , "dummy" , nil , nil )    ) ,
                        "none" ,
                        stack( nil , nil , 0 ) ,  ! a dummy since the first
                                                  ! ps does not need one

                        true ,
                        nil , ! sends its result nor left or right
                        @ 1 of pntr [ nil , nil ] ,
                        nil ,
                        nil ,
                        false , ! lhs not set
                        0 ,   ! size
                        0 ,   ! cells
                        0 ,   ! stack depth
                        0
                     )
first( next ) := first

first( father ) := first




! chain it in
```

```
      ring := first
      last in ring := first

      CLOCK := 0
      i := 0     ! the spawned processes
      j := 0     ! the extension processes
      nodes := 0 ! the number of nodes+ 1printnode
      psptr:=1
      write "'nsampling every ' n"
      smi:=readi
      activity := going
      repeat processor( ring )  !  ring may get expanded
      while  activity is going and ring ~= nil do
             ring := ring( next ).

      write "'n" , i , " leaf processes ",
            "'n" , j , " extension ps ",
            "'n" , nodes , " node ps "

      let i :=1; while ps no(i)>0 do{
      output d f, ps no(i), " "; i :=i+1}

      output d f, -1
end






procedure late( -> bool )
SIZE > noi
```

```
procedure spawn( cpntr code , env , slot , cstring sub.cy )
if ~isval( code ) and code isnt nothere and
code isnt er.val do
case true of
     code is const :
     slot( top ) := code( the.const )


    code is coerse and code( the.susp , its.env ) = EVALUATED ,
    code is suspended and code( its.env ) = EVALUATED          :
    slot( top ) := if code is coerse then code( the.susp , its.val )
                   else code( its.val )

    code is coerse and code( the.susp , lock ) ,
    code is suspended and code( lock )         :
    slot( top ) := if code is coerse then code( the.susp )
                   else code

    sub.cy ~= "right" and ~late
        if code isnt coerse and code isnt suspended then
        slot( top ) := suspended( code , env , false ) else
        slot( top ) := if code is suspended then code
                       else code( the.susp )


    default
    begin
         let env = if code is suspended then code( its.env )
                   else env
```

```
        let code = if code is suspended then code( its.val )
                      else code
         let newps = process(
                                 stack( nil , nil , 0 ) ,
                                 env ,
                                 code ,
                                 dump( dead , sub.cy ,  nil , nil ) ,
                                 if sub.cy = "right" then "none" else
                                 if sub.cy = "coerse"    then "none" else
                                 if sub.cy = "left" then "none" else
                                 sub.cy ,
                                 slot ,
                                 if sub.cy = "right" then false else true ,
                                 if sub.cy="left" then LEFT else RITE ,
                                 @ 1 of pntr[ nil , nil ] ,
                                 PS ,
                                 nil ,
                                 false ,
                                 if sub.cy="right" then PS(size) else 0 ,
                                 0, ! calls
                                 if sub.cy="right" then PS(stackdepth) else
                                 0 ,
                                 0  ! lock
                               )

            if sub.cy = "right" then j := j+1 else i := i+1
            slot( top ) := nothere( newps )



         ! make room in data dep

         PS( data.dep , 2 ) := PS( data.dep , 1 )
         PS( data.dep , 1 ) := newps
```

```
            ! chain it in the ring

            newps( next ) := last.in.ring( next )
            last.in.ring( next ) := newps
            last.in.ring         := newps

      end


procedure find.ps( cpntr ring -> pntr )
begin
     let wanted := ring
     repeat wanted := wanted( next )
     while wanted( next ) ~= ring

     wanted
end

procedure find.father( pntr ps -> pntr )
if ps( next ) = ps then ps  else
begin
     let ptr := ps( next )

     let stop := false
     repeat{
            for i = 1 to 2 do
            if ptr( data.dep , i ) = ps do stop := true
           }
     while stop = false do ptr := ptr( next )

ptr
end
```

```
procedure have.chld( cpntr ps ->int )
{     let child = ps( data dep )
      if child( 1 ) = nil then 0 else
      if child( 2 ) = nil then 1 else
      2
}


procedure unlock( cpntr ps )
{
     if ps( c ) is overwrite do
     ps(c,susp,lock):=false
     let dmp := ps( d )
     while dmp ~= nil do
     {
        if dmp( Code ) is overwrite do
        dmp( Code , susp , lock ) := false

        dmp := dmp( Dump )
     }
}

procedure kill( cpntr ps )

if ps ~= nil  do
if have.chld( ps ) = 0 then
begin
      if ps( c ) is main do ring := nil
      if ring ~= nil do
      {

      ! unchain it

      let previous = find.ps( ps )
```

```
        previous( next ) := ps( next )

        if last.in.ring = ps do last.in.ring := previous
        let p = find.father( ps )
        let i := 1
        while p( data.dep , i ) ~= ps do i := i+1
        if i = 1 do p( data.dep , 1 ) := p( data.dep , 2 )
        p( data.dep , 2) := nil

        if ps(line)=RITE do
        {     p(cells):=p(cells)+ps(cells)
              p(stackdepth):=ps(stackdepth)  }
        }
        if messages and ps(c) is main do
        begin
              output o.f :   "'ncomputation completes'n" ,
                             "'nits size in m/c steps" ,
                             "'nsize = " ,ps( size ) ,
                             "'nmaximum stackdepth = " , ps( stackdepth ) ,
                             "'ncells used = " , ps ( cells ) ,
                             "'nlock cycles = " , ps( lock.cycles ) ,
                             "'n" ,
                             "'ntotal locks = " , GLOCK ,
                             "'n*********************************'n" ,
                             "'n'n"
        end

        unlock( ps )
  end  else
  for i = 1 to have.chld( ps ) do kill(   ps(data.dep,i)  )



  procedure lock.wait
```

```
begin
     incycle := false
     GLOCK := GLOCK + 1
     locksup(PS)
end


! ++|+++++++++++++++++++++

procedure declare( pntr ds , env -> pntr )
begin

     procedure decl( pntr form , expr , guess , oldenv -> pntr )
     if form is id then
      if expr is suspended then defn( form , expr , oldenv , expr )
      else
          defn( form , suspended( expr , guess , false ) , oldenv , expr) else
     if form is const or form is repetition then oldenv
     else
     begin  ! form is cons (a list)
        let com.expr = if expr is suspended then expr else
                       suspended( expr , guess , false )
          let hdcode   = suspended( prefix( HD , coerse( com.expr ) ) ,
                                  nil , false
                                )
        let tlcode   = suspended( prefix( TL , coerse( com.expr ) ) ,
                                  nil , false
                                )

        env := decl( form( hd ) , hdcode , guess , oldenv )
        env := decl( form( tl ) , tlcode , guess , env    )
          env
     end
```

```
      let guess = defn( nil , nil , nil , nil )  ! for recursion
      while ds ~= nil do
      begin
           env := decl( ds( defn.name ) , ds( its.defn ) , guess , env )
           ds := ds( next.defn )
      end
      guess( defn.name ) := env( defn.name )
      guess( its.defn ) := env( its.defn )
      guess( next.defn ) := env( next.defn )
      guess( for.show ) := env( for.show )
      guess
end


procedure lookup( pntr name , env -> pntr )
begin
      while env ~= nil and env( defn.name ) ~= name do
           env := env( next.defn )
      if env = nil do
      begin
           env := the.env
           while env ~= nil and env( defn.name ) ~= name do
                env := env( next.defn )
      end
      if env = nil then
           if name = APPLY then nil
           else errl( UNDEF , name )
      else env( its.defn )
end


procedure error( string message )
```

```
begin
        errorflag := true
        message.given := true
        write "'n*** " , message , " in or near 'n"
end


procedure err1( pntr op , arg -> pntr )
        er.val( prefix( op , arg ) )


procedure err2( pntr op , arg1 , arg2 -> pntr )
        er.val(  infix(  op , arg1 , arg2 )  )


procedure print( pntr v )
! part of the evaluator now , ie a m/c instruction

begin
case true of


v is suspended
                        begin
                                save.cont( "print" )
                                cont.at( coerse( v ) , "coerse" )
                        end

v is char           : output o.f , v( the.char )

v is num            :  output o.f , v( the.num )

v is logic          : output o.f , v( the.bool )

```

```
v is cons or v = nil
if v is cons do
begin
        let Hd = v( hd )
        let Tl = v( tl )
        pushstack( Tl )
        save.cont( "print" ) ! the Tl
        pushstack( Hd )


end


v is closure : function id( v : true )



        default :
                    begin
                            output o.f , "'nIllegal Expression.  "
                            show( v )
                            activity := notgoing  ! a flag to print
                         incycle  := false      ! exit the main process
                      end


! if have output an object

if v is logic or v is num or v is char or v = nil or v is closure do

begin
        flush( o.f )
        load.cont
end
```

```
end



procedure show( pntr v )
if v is const then show( v( the const ) ) else
if v is id then output o.f , v( the.id ) else
if v is num then output o.f , v( the num ) else
if v is char then
     output o.f , case v of
          NL : "nl"
          SP : "sp"
          NP : "np"
          TAB : "tab"
          default : "%" ++ v( the char ) else
if v is logic then output o.f , v( the bool ) else
if v = nil then output o.f , "()" else
if v is cons then
begin
     output o.f , "("
     repeat
     {    show( v( hd ) )
          output o.f , "."
          v = v( tl ) }
     while v is cons
     show( v )
     output o.f , ")"
end else
if v is cond then
begin
     show( v( test ) )
     output o.f , " -> " ; show( v( left.fork ) )
```

```
--       output o.f , " ) " , show( v( right fork ) )
-1   end else
--   if v is closure then function id( v , true ) else
--   if v is suspended then show( v( its val ) ) else
--   if v is strict then output o.f , v( strict op ) else
--   if v is repetition then show( v( the rpt ) ) else
--   if v is map then
1-   begin
--       output o.f , " "
--       let f := v
--       while f is map do
2-       begin
--           show( f( form ) )
--           output o.f , "=>"
--           f := f( body )
-2       end
--       show( f )
--       output o.f , ";"
-1   end else
--   if v is trys then
1-   begin
--       let t := v( clauses )
--       while t ~= nil do
2-       begin
--           show( t( hd ) )
--           output o.f , "'n"
--           t := t( tl )
-2       end
-1   end else
--   if v is er val then show( v( arg ) ) else
--   if v is prefix then show prefix( v ) else
--   if v is infix do
--       case v( infix op ) of
--
```

e

else
e

```
        APPLY      begin
                        output o.f , "("
                        show( v( e1 ) )
                        output o.f , "  "
                        show( v( e2 ) )
                        output o.f , ")"
                   end

        BLOCK    : begin
                        show( v( e2 ) ) ! the expression
                        output o.f , " where ["
                        showenv( v( e1 ) , 4 )
                        output o.f , "] "
                   end

        default    begin
                        output o.f , "("
                        show( v( e1 ) )
                        output o.f , " " , v( infix.op , mnemonic ) , " "
                        show( v( e2 ) )
                        output o.f , ")"
                   end



procedure show.prefix( cpntr v )
case v( prefix.op ) of

CHECKLIST: show( v( e ) )

LISTERR  : begin
                output o.f , "List Wanted:<"
                show( v( e ) )
                output o.f , ">"
```

```
-1                end
--
1-  COND          begin
--                    output o.f , "<"
--                    show( v( e ) )
--                    output o.f , "> =>  "
-1                end
--
1-  UNDEF         begin
--                    output o.f , "Undefined:<"
--                    show( v( e ) )
--                    output o.f , ">"
-1                end
--
1-  default       begin
--                    output o.f , v( prefix op
--                    show( v( e ) )
-1                end
--
--
--
--  procedure showenv( pntr env ; int count )
1-  begin
--      let i := count
--      let ds := env
--      while i > 0 and ds ~= nil do
2-      begin
--          if ds( defn.name ) ~= APPLY do
3-          begin
--              output o.f , "'n"
--              show( ds( defn.name ) )
--              output o.f , " = "
--              show( ds( its defn ) )
--              i := i - 1
```

, mnemonic ) , " "

```
                end
                ds := ds( next.defn )
        end
        output o.f , "'n      'n"
end


procedure function.id( pntr f ; bool brackets )
begin

        procedure equiv( pntr object , definition -> bool )
        ! object is a closure which we are comparing with definition in env
        if definition is suspended and definition( its.val ) is closure then
            object = definition( its.val ) or  ! ie, same closure
                begin
                    let obj = object( fn.def )  ! map or trys
                    let def = definition( its.val , fn.def )  ! map or trys
                    obj = def or
                        obj is trys and def is trys and
                        begin
                            ! is first clause for obj a clause of def ?
                            let obj.first = obj( clauses , hd )
                            let defs := def( clauses )
                            while defs ~= nil and obj.first ~= defs( hd ) do
                                defs := defs( tl )
                            defs ~= nil
                        end
                end
        else false ! definition cannot be a function "looked up" to produce object


        procedure find( pntr obj , env -> pntr )  ! oppisote of lookup
        begin
            let env := env
```

```
          while env ~= nil and ~equiv( obj , env( its.defn ) ) do
               env := env( next.defn )
          if env = nil then nil
          else env( defn.name )
     end


     ! function.id

     let name = find( f , the.env )
     if brackets do output o.f , "<"  ! open top level only
     if name ~= nil then show( name )
     else begin
               let f.env = f( fn.env )
               let name = find( f , f.env )
               if name ~= nil then show( name )
               else begin
                         ! get partial mapping
                         let ap = lookup( APPLY , f.env )
                         if ap = nil then output o.f , "Function id error"
                         else begin
                                   function.id( ap( hd ) , false )
                                   output o.f , " "
                                   show( ap( tl ) )
                              end
                    end
          end
     if brackets do output o.f , ">"  ! close top level only
end



! main program
```

```
2658 --  interact
2659 --  ?

**** Program Compiles ****
```