Appendix One

DEST - User Manual

DEST - A Definitive Environment for Set Theory

User Manual

John Buckle

DEST is an interactive interpreted language designed to allow users to learn, experiment and investigate the ideas of set theory. The language is based on definitive principles whereby variables in DEST store expressions (“definitions”) not values. Hence complex functional relationships between variables can be stored and maintained, and the user can experiment with different values without having to recalculate intermediate expressions.

DEST is a subset of the language Pecan used for investigating finite lattices. This document contains the minimal user specification for DEST, local versions of the language may contain extra features. The version described here can be used on any standard terminal running on a UNIX† or similar operating system.

† UNIX is a trademark of Bell Laboratories.

1. Introduction

DEST is an interactive interpreted language designed to allow users to learn, experiment and investigate the ideas of set theory. The language is based on definitive principles whereby variables in DEST store expressions (“definitions”) not values. So a session with DEST consists of a dialogue between the user and the computer where the user can enter a sequence of definitions followed by an examination of the resulting expressions. Hence via the dialogue a network of definitions is produced that is maintained by the computer in a dynamic environment. Therefore complex functional relationships between variables can be stored and maintained, and the user can experiment with different values without having to recalculate intermediate expressions.

As well as having data types for handling sets and elements, DEST can also handle ordered pairs and tuples, maps, relations and ordered sets. These higher order or *structured* data types are expressed as combinations of ordered and unordered sets, and special coercion operators exist to transform data between different data types. DEST is equipped with the standard operations common to set theory, such as union, intersection, product, map image and inverse image etc., and how they behave depends on the types of the arguments. For example the union of two relation variables in DEST is treated differently to the union of the equivalent set variables.

DEST also allows a partial implementation of infinite sets, maps and relations. Since these sets are infinite and it is not possible to list every element in them the only operations possible on infinite sets are actions based on set membership. This is because DEST will not perform any action which it believes will cause it to enter into an infinite loop. Even with this constraint it is possible to use maps and relations based on infinite domains.

To aid in formulating definitions of maps, relations and infinite sets DEST uses a system of character variables with pattern matching operators and predicates. Here it is possible to obtain a subset of a set by locating all the elements containing a certain pattern, or to generate new sets containing elements with special patterns, or in the case of an infinite set define a rule that determines if an element is a member of the set.

One of the most important rules in DEST is not to define a variable in terms of itself. A common error for beginners is to type something like the following, expecting it to add a new element to a set:

```
s := s + { _a }
```

While in most programming languages this simply “increases” the value of `s`, in DEST it creates a self-referencing loop. This is because in DEST expressions are stored in variables, not values. Hence the expression `s + { _a }` is stored, when this is evaluated the interpreter will plunge into an ever decreasing pit trying to find the value of `s`. (Obviously it’s not that bad, as soon as DEST sees the same variable twice when it tries to evaluate anything it will complain, however it is important that the point that DEST stores expressions not values is clearly understood.)

2. Getting Started With Sets

The basic data types used in DEST are the types set and literal. Literals consist of the boolean constants true and false, the integer numbers and all the symbolic names of the user's most primitive objects (ie. objects that are not sets themselves). Variable names in DEST can be any string of characters consisting of letters, numbers and underscore beginning with a letter. So that DEST can tell the difference between a variable and a symbolic name, all symbolic names must begin with an underscore. Simple sets consisting of just literals and other variables can be defined by placing the elements of the set between braces { · · · }. For example the following definitions define i to be the intersection of the sets s and t, and define u to be the union.

```
s := { _a, _b, _c, {_a, _b} }
t := { _b, _c, s }

i := s & t
u := s + t

print i, u
{ _b, _c }
{ _a, _b, _c, { _a, _b }, { _a, _b, _c, { _a, _b } } }

s := { _a }

print i, u
{}
{ _a, _b, _c, { _a } }
```

As can be seen from the example the elements of a set's definition do not have to be literals, for example the definition of t uses the variable s. Also there is no requirement for the elements of a set to be all of the same type. The intersection of two sets is denoted by the symbol & (''elements in this *and* that set''), union is denoted by + and the operation of set difference by -. The empty set is represented by two braces with nothing between them {}. The procedure print simply evaluates the arguments given to it and prints them. If only one expression is to be evaluated then the word print can be omitted.

3. Ordered Pairs and Tuples

Ordered pairs and ordered tuples are defined in DEST in the same way as sets except that curved brackets are used instead of braces. Hence to defined a variable p to hold an ordered pair whose first element is _a and second is _b or define alpha to hold the first ten letters of the alphabet in that order then enter

```
p := ( _a, _b )
alpha := ( _a,_b,_c,_d,_e,_f,_g,_h,_i,_j )
```

Since p was defined as an ordered tuple consisting of two elements DEST automatically declared p to be of type pair, similarly alpha was declared to be of type tuple. Also since curved brackets are used in controlling the evaluation of expressions it is not possible to define a 1-tuple directly. The type pair is simply a special case of tuple that has some extra functions associated with it and is used in conjunction with the types

map and relation. In fact ordered pairs and two element tuples are interchangeable, DEST will convert data between these two types when the context requires such a change.

In the above examples the elements of the ordered tuples consisted of literals, however as in sets they can hold any type of expression, and it is not required that all the elements have the same type.

```
a := { _f, _g }
b := ( a, _f )
t := ( a, b, _x, _y )
```

In this example b is the ordered pair ({_f,_g},_f), (not the 3-tuple (_f,_g,_f)), similarly t is a 4-tuple consisting of a set, a pair and two literals.

An ordered pair or tuple can be unordered, producing a set consisting of the elements in no particular order, by *casting* the tuple into a set,

```
s := (set) t
```

will define s to be the unordered set produced from t, (while t is defined by the above expression, s will be {_x,_y,{_f,_g},({_f,_g},_f)}). The union, intersection and difference of tuples differ from that of plain sets. The union of two tuples is the concatenation of the tuples, producing a tuple whose length is the sum of its arguments. The intersection of two tuples is the longest prefix common to both tuples. The difference is the suffix of the first tuple remaining after finding the longest common prefix. While these definitions produce a non-commutative union, they introduce a rich algebra for tuples that parallels the set identity $A=(A \cap B) \cup (A \setminus B)$. The normal definitions of union etc. can be obtained by first removing the ordering on the tuples by casting the arguments into sets.

The cross product of two sets is denoted by the operator *. Given two sets A and B the product A*B is the set of all the ordered pairs (x,y) where x is an element of A and y is an element of B. There is a special type in DEST for sets consisting of just ordered pairs called pairs. Cross product is an example of an operator that returns an expression of type pairs, and is mainly used in defining maps and relations.

4. Control in DEST

DEST incorporates several control constructs to aid in the definition of subsets, evaluation of expressions and the enumeration of results. The use of pattern-matching, pattern generators and labels is discussed in a later section. This section gives details of the control constructs associated directly with sets and elements.

The actual definition of a variable can be printed using the print_def command. This will display what the interpreter currently thinks the variable is defined to be.

An element can be tested for membership in a set by using the in operator which returns either the literal true or false, for example the test $x \in X$ is entered as x in X.

Subsets of a set can be defined by using the specification operator |. The operator takes as its arguments an iterating control variable, a range set and a boolean condition. The operator works by iterating through all the elements of the range set and returns the set containing those elements that comply with the condition. For example in the definition

below if S is a set of integers then T would be the subset of integers divisible by 3,

```
T := { e in S | e mod 3 = 0 }
```

Similarly the elements of a set can be iterated through by using the iteration statement `for`. The iteration operator takes as arguments an iterating control variable, a range set and a sequence of statements. For example if S is a set of sets then the variables S1,S2, . . . , would be defined to be the individual elements of S, .

```
for e in S do S$$ := e ; print e ; end
```

The use of $$ in the above example is a means of generating a sequence of unique variable names inside a `for` loop. It initially starts with a value of one and is increment at every iteration. It should be noted that a `for` loop is a shorthand for producing several definitions or expressions, not a definition in itself. Hence in the above example the variable S1 is defined by an expression which is taken to be independent of S (ie. S1 will keep its old definition even if S is redefined).

The iterating control variable used in defining subsets and in `for` loops is local to the statement and supersedes any other variable of that name. At the end of the statement the control variable is removed from the symbol table.

Sometimes it is desirable for a variable to use the *current* value of an expression in a definition rather than maintaining the dynamic link normally given by DEST. To use the current value the expression should be given as an argument to the function `eval()` which will evaluate the expression and substitute the value. `Eval()` will always return a constant expression except when the expression involves an iterating control variable. In this case the control variable is not substituted, however all other variables will be replaced by their current value.

Integer expressions can be calculated in DEST using the normal arithmetic symbols of `+, -, *, div, mod`. In this case the arguments and results of the expressions are considered to be literals. Integer ranges can be defined in sets by using a similar notation to that of Pascal and Miranda by specifying the bounds separated by two dots, `1..10`. Integer ranges in tuples are ordered with the lowest integer coming first. Arithmetic progressions can also be defined by specifying the first two elements in the sequence followed by the upper bound, `1;4..20`. If the upper bound is less than the lower bound then an empty range is produced.

Expressions can be evaluated conditional by using the `if-then-else` expression. If the condition is true then the expression following `then` is evaluated, otherwise the expression following `else` is evaluated. In DEST the `else` is compulsory unlike most procedural languages. The conditional expression can be any expression returning either the literal `true` or `false` constructed from sub-expressions using `and`, `or` and `not`. Binary comparisons between sets (and between integers) are done by the operators `<=, =>, <, >, =, <>`. These should be read as set inclusion, strict set inclusion, set equality and set inequality. These are the same symbols that are used for the standard ordering on the integers, however this causes no problems since it easy to determine by context what type of comparison is desired.

5. Maps, Relations and Ordered Sets

Variables of type map and relation are treated as sets of ordered pairs with special operations to reflect their use. Order variables are considered as a pair of objects representing the underlying set and an order relation on that set. Since maps, relations and ordered sets tend to be quite large objects (each requiring a large array of ordered pairs) it is not normally possible to define them explicitly. To enable the user to define these objects concisely DEST uses a system of pattern-matching predicates so that sets of ordered pairs can be defined, from which maps, relations and ordered sets can be obtained. This section introduces the operations possible on maps, relations and ordered sets. Methods of defining these objects are introduced in sections seven and eight.

The image of an element x in a map m is given by m(x), the image of a set X in m is m{X}. The preimage of a element in the range of m is given by m^(y) and the preimage of a set is m^{Y}. The type of a preimage or the image of a set is always a set, the type of the image of an element depends, of course, on the image. The composition of two maps f and g is denoted by g@f ("the value of g at f"). The union and intersection of maps is treated as the union or intersection of the corresponding sets of ordered pairs.

If r is a relation then relations in r is examined by the operator ~r~. That is, if $(x,y) \in r$ then x~r~y is true. The set of elements related to an element a (ie. $\{x \mid (a,x) \in r\}$) is denoted by r(a). The set of elements that relate to a is denoted by r^(a). The intersection of two relations is the relation produced by the intersection of the corresponding sets of ordered pairs. The union of two relations is the transitive closure of the union of the corresponding sets of pairs. If the transitive closure is not required then one of the relations should be cast first into pairs.

Order relations on an ordered set p are denoted by the operators <p<, <p=, =p=, <p>, =p>, >p>. The union and intersection of two ordered sets is the union or intersection of the base sets and of the relation. The relation given to an ordered set can represent any kind of ordering, for example it could be a pre-order, partial order or total order. The comparison a =p> b is equivalent to the condition that (a,b) is a member of the order relation.

6. Type Hierarchy

The data types of DEST are organised into a hierarchy so that they reflect the natural structure of the objects they are representing. For example any variable of type map can also be considered as a variable of type pairs. Hence the more specialised types are derived as special types from the more general, as illustrated in figure 1.

set
order
pairs
relation
map
tuple
pair

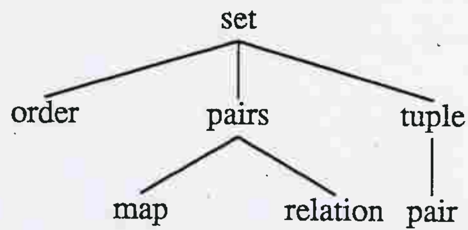

figure 1

The edges of the graph represent built-in coercion operators between two types that either forget part of or restructure the data held in a variable. Types higher in the tree represent more general types, types lower in the tree represent more specialised types. The data type `literal` can be considered as the most primitive data type in DEST since literals can not be transformed into any other type and are hence not part of the hierarchy given in figure 1.

While the variables of DEST are typed they do not need to be declared before use, their first defining assignment is used to specify their type. This is normally done implicitly by determining the type of the expression, however the user can specify the resulting type of an expression by casting the expression to the appropriate type.

While from a purely mathematical point of view it is only necessary to have a data type for sets, it is semantically useful to have a universal data type `element` to handle the members of a set. The `element` data type is a *named union* of the standard types consisting of a field for each type plus a tag entry to record the type currently being held.

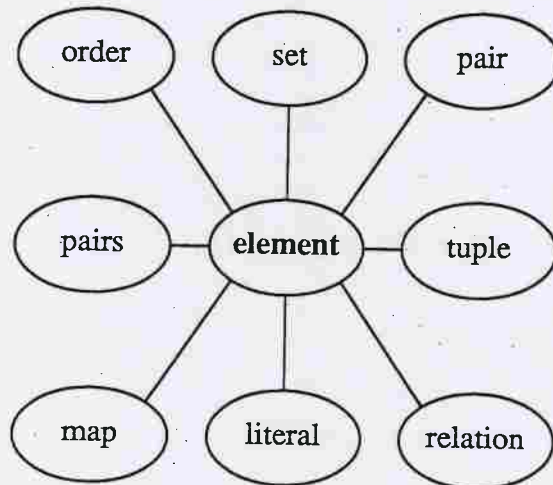

figure 2

Element variables are implicitly used in `for` loops and in subset specification since the iteration control variable is of type `element`. In these cases the control variable adopts the type of the element it is representing. Hence element variables allow generic definitions to be written so that similar structures over different types can be examined. For example without element variables it would be necessary to write individual definitions to examine posets based on literals, sets, relations etc. By writing generic element definitions the definitions can be used in any context, the DEST interpreter will select the appropriate types and operators that the context requires.

The basic data type in DEST is the set, all other data types except `literal` being defined as combinations of ordered and unordered sets. For example the type `order` is an ordered pair consisting of a `set` field and a `relation` field, where the type `relation` is a set of `pair` variables etc. To gain access to this structure all the ''structured'' types in DEST are accompanied with ***member functions*** to retrieve the data. The difference between member fields and member functions is that functions return a value normally based on all the fields of the type, and it is not possible to assign to a function. For example the type `map` has a member function `.domain` which returns the domain of the map.

Pairs and Tuples

Variables of type `pair` have associated with them two member functions `.first` and `.second` which return the first and second element of the ordered pair respectively. Tuple variables have member functions `.head` and `.tail` which when applied to the ordered set $(a_1,a_2,\ldots,a_n)$ return the first element a_1 and the n-1 tuple $(a_2,\ldots,a_n)$. The tail of an ordered singleton is the empty set.

Maps, Relations and Pairs

Variables of type `map` consist of a set of ordered pairs from the product $set_A \times set_B$. There are two member functions, `.domain` returns the set of elements used in set_A and `.range` returns the set of elements used in set_B.

Variables of type `relation` consist of a set of pairs from the product $set_A \times set_A$. Relations have only one member function `.domain` which returns the set of elements used in set_A.

Variables of type `pairs` have the same structure and member functions as the types `map` and `relation`. They are included as a type in DEST to re-enforce the fact that maps and relations are sets of ordered pairs.

Ordered Sets

A variable of type `order` is an ordered pair `(base_set, order_relation)` consisting of a base set and a relation on base_set × base_set. Order variables have two member functions to access their components, `.set` returns the set of elements used and `.order` returns the relation.

Type Conversions

Values can be transferred between variables of different types by *casting* an expression into the destined type:

new_type_variable := (*type name*) *old_type_expression*

Type conversions from a more structured type (ie. a type printed lower in the hierarchy tree above) to a less structured type are performed automatically *(automatic coercion)* according to context. For example the expression `relation_variable.range` is interpreted as `((pairs) relation_variable).range`. Automatic coercion is always defined and normally results in some data being forgotten. Type conversions in the opposite direction *(reverse coercion)* is sometimes undefined and may require extra information to be given by the user.

Automatic Coercion

set of pairs → set :	no data loss, lose use of .domain, .range
map → set of pairs :	no data loss, lose use of map operations
relation → set of pairs :	no data loss, lose use of relation operations
pair → tuple :	no data loss, lose use of .second
order → set :	order loses .order and is just left with .set
tuple → set :	Always defined, returns the set with the ordering removed.

Reverse Coercion

set → set of pairs :	Undefined if any element of the set is not a pair or 2-tuple.
set of pairs → map :	Undefined if there exists $p_1,p_2 \in$ pair_variable such that p_1.first = p_2.first and p_1.second $\neq$ p_2.second.
set of pairs → relation :	Always defined.
set → order :	This requires a pair of objects (base_set , order_relation),
tuple → pair :	Undefined if tuple.tail.tail $\neq \emptyset$
set → tuple :	Undefined if the set is not a singleton.

7. Generators and Pattern-Matching

Since sets are defined over arbitrary domains and not just numbers of some description it is natural for literals and the variables in DEST that deal with them to be based on character representations rather than numerical values. DEST uses a naming system which as well as allowing the user to use meaningful names also has the expressive power to define maps and relations, handle ordered sets, perform numerical calculations and can be used predicates. DEST incorporates a system of *pattern-variables* that exist locally inside definitions and can be used in both predicates and expressions and to create new literals. Pattern-variables begin with a dollar symbol followed by a digit and are bound to a set of literals. When evaluated the pattern-variables will be repeatedly matched against the literals in the set to produce all possible matchings, these matchings can then be used elsewhere in expressions.

Formally a *simple-pattern* is a sequence $T_1V_1T_2 \cdots V_{k-1}T_k$ where T_i's are arbitrary strings of alphanumeric characters plus underscore and V_i's are pattern-variables, a *pattern* is an n-tuple $P=(P_1,P_2,\ldots,P_n)$ of simple-patterns and patterns. A **generator** is an expression:

pattern `in` *tuple_expression*

where *tuple_expression* is an n-tuple $E=(E_1,E_2,\ldots,E_n)$ such that the simple patterns in P correspond to sets in E and patterns in P correspond to tuples of the same arity in E. The generator can be used to assign values to the pattern-variables which are either quantified by a predicate or used directly in an expression. The former method will produce a subset of the tuple-expression while the latter can create new literals. The two methods have the following syntax respectively:

{ *generator* | *quantifier* } (1)

{ *expression* | *generator* } (2)

In cases where both sides of the bar appear to be generators (ie. because they both use `in`) then the left expression is taken to be the generator, this follows the normal interpretation used in mathematics. For example if `L` is a set containing the literals `_L1`, `_L2`,..., `_L10` then the expression

```
T = (relation) { ( _L$1, _L$2 ) in L*L | $1 >= $2 }
```

would define `T` to be a total order on `L`. Here `$1` and `$2` are being used as pattern-matching variables that parse the literal's name according to the template given in the predicate. The set `L` could have been defined by

```
L = { _L$1 | $1 in { 1 .. 10 } }
```

Both of these examples specify their result as a subset of a larger set, in the first example as a subset of `L*L` and in the second as a subset of the set of all literals beginning with `_L`. However semantically the two examples differ in the process of how the subset is generated. In the first example the elements of the subset are generated by the expression on the left hand side of the specification symbol "|" and are quantified by the predicate on the right. In the second example the elements are generated by the predicate and are then transformed by the expression.

It should be noted that pattern-variables, generators and expressions involving pattern-variables can only refer to literals and can not be used to reference variables. To be able to would permit definitions to be defined whose actual definition would change over evaluation as well as its value. Also the use of the "such that" bar with generators should not be confused with its use in section four. While both can be used to specify subsets of a set, in section four the superset could contain any type of elements while with generators it is required that the elements be literals.

Pattern-variables and generators are used through-out DEST as a means of defining maps, relations, orders, tuples etc. because they offer an extremely versatile way of linking literals to expressions and back to literals. They provide a system that as well as being semantically sound is also clear and easy to use. The use of patterns is further enchanced when labels are introduced in section nine.

8. Using Pattern-Matching to Define Maps and Relations

The methods of defining maps and relations are identical since they are both sets of ordered pairs, it is only in usage that they differ. There are three ways in which sets of pairs can be defined. Firstly the ordered pairs that make up the map or relation can be explicitly listed. Secondly the pairs can be calculated from an expression using a generator. Lastly a sequence of guarded-expressions can be given that specify a means of obtaining the corresponding result(s) from elements in the domain or range.

By explicitly listing the pairs that make up the map or relation the map/relation is described exactly and all the information necessary for evaluating inverse images of maps or transitive closures of relations etc. is provided. Obviously though this method can only be used on small domains.

To specify maps and relations on large finite domains generators can be used. Either the resulting set of pairs can be defined as a subset of a cross product quantified by a suitable predicate or by an expression bound to a generator. In the first method given in section seven all the pairs in the cross product will be considered resulting in many evaluations of the predicate, hence to avoid unnecessary computation when calculating the image of maps this method is best left for defining relations. In the second method it is possible to restrict the generator to the domain and hence is more economic when defining functions.

```
rel := { ($1,$2) in {1..12}*{1..12} | $1 mod $2 = 0 }

lat := { (_L$1_$2,_L$3_$4) in L*L | $1>=$3 and $2>=$4 }

fib := { ($1, if $1>1 then fib($1-1)+fib($1-2) else 1)
                | $1 in (0..100)}
```

The first example illustrates how the first method can be used to define a relation to express "is a multiple of". The second example defines a partial order on a set `L*L` by examining the names of the literals. Here it is assumed that the elements of `L` have been assigned names of the form `_L10_3` etc. The last example defines the fibonacci function using recursion. It should be noted that this definition does not contradict the condition of non self-refencing definitions since at no point is any set in `fib` defined in terms of itself and hence no infinite refencing loops occur.

The use of generators can be extended to handle more complicated functions and relations and infinite domains by introducing guards on the expressions. Guards are similar to if-then-else however they can only occur inside a set or tuple definition and it is possible to omit the else clause. If the guard is true then the expression following the guard is evaluated, otherwise the expression following the ? is evaluated.

```
fib := { [$1>1] -> ($1, fib($1-1)+fib($1-2)) ? ($1,1)
                | $1 in (0..100) }

abs := { [$1 <  0] -> ( $1,-$1) ? ($1,$1),
         [$2 >= 0] -> ( $2, $2),
         [$2 >= 0] -> (-$2, $2) | ($1,$2) in Z*Z }
```

The first example is a repeat of the fibonacci function this time using guards rather than the if-then-else. In this case the domain of the generator is finite so the function will be evaluated fully. In the second example an absolute value function is defined. Here there

are four expressions giving both the function and its inverse.

As can be seen from the example of the absolute function the only way to be able to find the inverse of a function when the domain is infinite is to supply the necessary expressions to calculate it. If the inverse function is not provided when the domain is infinite then it is impossible for DEST to determine any inverse images since it can not enumerate the function.

9. Alternative Labelling System

Labels provide a parallel system for referencing elements which can be used to complement the use of the literal's name or to enhance the power of generators. Labels act as a local naming scheme in a set that can act on elements of any type. Also labels are produced automatically for cross products by combining the labels present in the product's arguments. Hence by labelling elements appropriately the interpreter will produce a suitable labelling for the new set. The elements of a set's definition can be referenced by two labelling systems called enumeration labels and set labels.

Enumeration Labels: By requesting an enumeration of a set, all the elements of the set will be associated with a unique integer. For unordered sets the number associated with each element will be time dependent and the element will only have that value while the definition and its associated environment remains constant. For ordered sets the number associated with each element will be in accordance with the element's position in the set, the element at the head having label ":1". :

Set Labels: Set labels are names given by the user to the elements of a set's definition, either when the definition is first given or later by attaching a name to an element using an enumeration label. Unlike enumeration labels, set labels retain their meaning over time. Set labels can be used in patterns and hence provide the same means of specifying structured sets as literals.

For example if S is defined as

```
S := { e1, s1, _a, _b, s2 :: { _a, s1 } }
```

then the enumeration labels might be S:1 = e1, S:2 = s1 etc. The only set label defined so far is S:s2. Note that an underscore is not required for labels since their names are always prefixed by a colon.

Labels provide a means of accessing the value of the elements of a set and not a way to change them, that is it should be noted that labels are not *l-values* and can not have expressions assigned to them. To be able to change an element via a label would usurp the definition of the set and the elements.

The operator cross product will create new labels for the elements of sets defined as a product from the labels of the arguments. Let C:=A*B and let e1 be a variable in the definition of A and e2 a variable in the definition of B where e1 has set label :lab1 and e2 has set label :lab2. The element representing (e1,e2) in C will have set label :lab1_lab2. If either e1 or e2 are labelless then (e1,e2) will be labelless. For example if B is a set of elements with set labels :L0,:L1,:L2,... then the definitions

```
S   := { B * B }
O1  := { (:L$1, :L$2) in B*B | $1 >= $2 }
O2  := { :L$1_L$2 in S | $1 >= $2 }
```

will define O1 and O2 as a set of pairs which defines a total order on B. The definition of O1 uses labels in generators and the definition of O2 uses the creation of labels by the product operator to act as a pattern.

Labels can also be applied to elements that appear in a recursive definition. Since a label is local to a set and appears only in conjunction with a set it is possible for a set to have several instances of the same label at different levels. For example a recursive map could be defined as:

```
rec := { [$1=1] -> ( 1, {t::_a}) ?
                ($1, {t::_a, T::rec($1-1)} ) | $1 in Z }
```

Hence `rec(3)` equals `{t::_a, T::{t::_a, T::{t::_a }}}`, so `rec(3):t` is the literal `_a` and `rec(3):T` is the set `{t::_a, T::{t::_a }}`. Since `rec(3):T` is a set which contains labels the above process can be repeated, for example `rec(3):T:T:t` etc.

10. Infinite Sets

DEST has two predefined infinite sets, the set Z consisting of the integers and N of the natural numbers (including zero). As has been shown earlier infinite sets of literals can be defined by basing a generator on one on these sets:

```
B :=  { _L$1 | $1 in N }
```

Furthermore it is possible to define an order on infinite sets using pattern matching with generators. For example the definition

```
O := (relation) { (_L$1, _L$2) in B*B | $1 >= $2 }
```

will define O to be a total order on the set B. This order can be used to compare literals as in any finite order because by using pattern-matching it is possible to determine if an ordered pair is a member of the relation without enumerating the expression.

DEST will not attempt to evaluate any expression that is considered to involve an infinite set since this may force the interpreter into an infinite loop. The only operator that can be applied to infinite sets is the membership operator `in`, since by pattern-matching it is possible to determine membership without enumerating the set. As can be seen even with this restriction it is possible to define and use maps and relations on infinite domains.

DEST is unable to determine whether an expression that involves infinite sets is infinite, for example the intersection to two infinite sets need not be infinite. This problem is common to most computer interpreters and the normal solution is to classify any set derived from an infinite set as unenumerable. However there are two exceptions to this, when an infinite set is either intersected with a finite set or equivalently specified as a subset of a finite set then the result is considered finite.

11. Syntax of DEST

```
dest :       expr
        |    print expr_list
        |    print_def variable
        |    variable  := expr
        |    label     := variable label
        |    for variable in expr do dest_list end
        |

dest_list :  dest
        |    dest ; dest_list

expr :       { expr_list }
        |    ( tuple_list )
        |    { generator | expr }
        |    ( generator | expr )
        |    { expr_list | generator }
        |    ( expr_list | generator )
        |    eval ( expr )
        |    func_sym ( expr )
        |    variable ( expr )
        |    variable { expr }
        |    variable ^ ( expr )
        |    variable ^ { expr }
        |    variable . member
        |    expr operator expr
        |    expr label
        |    - expr
        |    not expr
        |    if expr then expr else expr
        |    ( type ) expr
        |    ( expr )
        |    variable
        |    literal
        |    label
        |    number
        |    pattern

expr_list :  ele_expr
        |    ele_expr , ele_expr expr_more
        |

tuple_list : ele_expr , ele_expr expr_more
        |    integer_range
        |

expr_more : , ele_expr expr_more
```

```
        |

ele_expr :  variable :: expr
        |   integer_range
        |   [ expr ] -> expr
        |   [ expr ] -> expr ? expr
        |   expr

generator : expr in expr

func_sym :  <predefined function>

operator :  +    -    *    &    @
        |   mod  div  and  in   or
        |   >    <    <=   =>
        |   =    <>   ||
        |   relation_order
        |   ordered_set_order

integer_range :   number .. number
        |   number ; number .. number

literal :   _ <alphanumeric> +
        |   true
        |   false

number :    <numeric> +
        |   - number

variable :  <alphabetic><alphanumeric>*

label :     : <alphanumeric> +

pattern :   <alphanumeric + $> +

alphabetic :      < a..z, A..Z>
alphanumeric :    < a..z, A..Z, 0..9, _>
```

12. Predefined Functions and Operators

<table>
<tr><th>Operator</th><th>Operation</th></tr>
<tr><td>+</td><td>set union, arithmetic addition</td></tr>
<tr><td>&</td><td>set intersection</td></tr>
<tr><td>-</td><td>set difference, arithmetic subtraction</td></tr>
<tr><td>*</td><td>cross product, arithmetic multiplication</td></tr>
<tr><td>@</td><td>composition of maps</td></tr>
<tr><td>mod</td><td>modulo arithmetic</td></tr>
<tr><td>div</td><td>arithmetic division</td></tr>
<tr><td>in</td><td>set membership, generator symbol</td></tr>
<tr><td>and</td><td>logical and</td></tr>
<tr><td>not</td><td>logical not</td></tr>
<tr><td>or</td><td>logical or</td></tr>
<tr><td>f(x)</td><td>map image of an element</td></tr>
<tr><td>f{X}</td><td>map image of a set</td></tr>
<tr><td>f^(x)</td><td>inverse image of an element</td></tr>
<tr><td>f^{X}</td><td>inverse image of a set</td></tr>
<tr><td>x..y</td><td>integer numbers from x to y</td></tr>
<tr><td>x;y..z</td><td>arithmetic progression x,y · · · z</td></tr>
<tr><td><</td><td>arithmetic less than, strict set inclusion</td></tr>
<tr><td>></td><td>arithmetic greater than, strict set inclusion</td></tr>
<tr><td><=</td><td>arithmetic less than or equal to, set inclusion</td></tr>
<tr><td>=></td><td>arithmetic greater than or equal to, set inclusion</td></tr>
<tr><td>=</td><td>equality</td></tr>
<tr><td><></td><td>not equal</td></tr>
<tr><td>||</td><td>not comparable</td></tr>
<tr><td>~r~</td><td>relation test on the relation r</td></tr>
<tr><td><p<</td><td></td></tr>
<tr><td><p=</td><td></td></tr>
<tr><td>>p></td><td></td></tr>
<tr><td>=p></td><td>comparison test on the ordered set p</td></tr>
<tr><td>=p=</td><td></td></tr>
<tr><td><p></td><td></td></tr>
<tr><td>|p|</td><td></td></tr>
<tr><td>::</td><td>definition of a set label</td></tr>
<tr><td>$1</td><td>pattern variable</td></tr>
<tr><td>$$</td><td>iteration count in for loops</td></tr>
</table>

Function	Argument Type	Operation
eval()	--	Evaluate operand and substitute value
type()	--	Return type of variable as a literal
elem()	set	Return an element from the operand if it is a set
union()	set	Return the set union of the operand
inter()	set	Return the set intersection of the operand
is_singleton()	set	True if the operand is a singleton set
is_empty()	set	True if the operand is the empty set
is_set()	--	
is_order()	--	
is_pairs()	--	
is_tuple()	--	
is_map()	--	True if operand is of that type or can
is_relation()	--	automatically coerced to that type
is_pair()	--	
is_literal()	--	
is_element()	--	
is_injective()	map	True if operand is an injective map
is_transitive()	pairs	True if operand is a transitive relation
is_reflexive()	pairs	True if operand is a reflexive relation
is_symmetric()	pairs	True if operand is a symmetric relation
is_antisym()	pairs	True if operand is an anti-symmetric relation
is_equivalence()	pairs	True if operand is an equivalence relation
is_partial()	pairs	True if operand is a partial order
is_preorder()	pairs	True if operand is a pre-order
is_total()	pairs	True if operand is a total order
transitive()	pairs	Return the transitive closure of the set
symmetric()	pairs	Return the symmetric closure of the set
reflexive()	pairs	Return the reflexive closure of the set
equivalence()	pairs	Return the equivalence closure of the set
partial()	pairs	Return the partial order closure of the set
total()	pairs	Return the total order closure of the set

Appendix Two

Pecan - User Manual

Pecan - A Definitive Environment for Lattice Theory

User Manual

John Buckle

Pecan[1] is an interactive interpreted language enabling the user to construct and investigate finite lattices. The language is based on a definitive notation where variables in Pecan store expressions rather than values. This creates a dynamic environment where the user can experiment with several different kinds of lattices while letting the computer maintain the functional relationships between variables.

Pecan contains the language DEST as a subset and it is assumed that the reader is familiar with the language. It is also assumed that the reader is familiar with lattice theory and understands the terms quotient lattice, homomorphism theorem etc. and is aware of the identification between congruences relations and join-irreducibles. This document contains the minimal specification of the language Pecan, local versions of the language may contain extra features.

[1] Pecans are small edible nuts of the walnut family.

1. Introduction

Pecan is a definitive based language designed for interactive analysis of lattices. Pecan contains the language DEST as a subset and incorporates additional data types and operators to handle lattices. While DEST can handle infinite sets to a limited degree, Pecan is restricted to finite lattices so that the identification between lattice congruences and sets of join-irreducibles can be applied[2]. Operations in Pecan are mainly based on the construction of lattices and the investigation of congruences. Congruences in Pecan can either be defined via a relation expression or by giving an appropriate set of join-irreducibles. Special operators exist for defining quotient lattices and natural homomorphisms between lattices and quotients, and the lattice of all congruence relations on a lattice can also be defined.

As well as using the methods available in DEST to define lattices, Pecan introduces a less formal, more intuitive method of constructing lattices using a cut and paste method. In this method the user constructs a Hasse diagram of the lattice from basic blocks which are combined to produce the final order. By this method it is relatively easy to construct complicated lattices.

To be able to work efficiently with lattices Pecan needs to enumerate the lattice and pre-calculate the table of meets and joins. Hence the user is required to *open* a lattice to indicate that this information should be calculated. The process of opening a lattice introduces two levels of definitions in Pecan. On one level the user can define an algebra and on a second level the user can define expressions based on the algebra.

2. New Data Types for Pecan

Pecan has three extra data types not included in DEST for handling lattices, congruences and homomorphisms. The data types `congruence` and `homomorphism` are extensions of the DEST types relation and map and are used in defining quotients. The type `lattice` is an extension of the type order and includes new member functions for accessing the components of the algebra.

Lattices

Abstractly lattices can be defined either as a special type of poset or as a special class of algebras. To reflect these two methods of definition Pecan allows variables of type `lattice` to be defined either by giving a poset or by specifying a set with two operators.

```
L1 := (lattice) P
L2 := (lattice) ( S, m, j )
```

In the example `L1` is defined by giving a partial order and `L2` is defined by giving an algebra. In the second case `S` should be a set and `m` and `j` maps from `S×S` → `S` representing the operators meet and join.

Since lattices can originate from two different sources, variables of type `lattice` have four member functions reflecting the two methods available. The member functions

[2] Buckle, J. *Computational Aspects of Lattice Theory*

`.set` and `.order` of orders are inherited by lattice variables as well as two new functions called `.join` and `.meet`. The operations of join and meet on a lattice `L` can either be expressed using the infix operators `\L/` and `/L\` or by using the member functions, for example the expression "x `\L/` y" is equivalent to "`L.join(x,y)`".

To be able to implement the operators meet and join efficiently and to be able to calculate quotients etc., large amounts of information are required to be calculated for each lattice variable. Hence while lattices can be defined and used in definitions at any time, the elements of a lattice can only be accessed after the lattice has been explicitly *opened* for use by the user. In opening a lattice a full enumeration of the lattice is performed and all join-irreducibles and meet-irreducibles in the lattice are found plus how they are related by the ~-relation. To aid brevity the infix operators of meet and join of the most recently opened lattice may be referred to as `/\` and `\/`.

To stop massive recalculations caused by accidentally redefining a lattice expression, all opened lattices are tagged so that in the event of a redefinition the user is warned and is given the option to cancel the operation. In this way Pecan is arranged as a two layer definitive language where the user defines lattices and then performs calculations in and on them. When the user wishes to investigate another system of lattices these lattices must be re-calculated by opening the definitions.

Once the enumeration of the lattice has been performed the elements of the lattice are given enumeration labels according to the order in which the elements appear in a topological sort of the lattice, the minimal element being given label :0. Enumeration labels are used by operators like quotient and cross product in generating set labels for elements of the newly created lattice.

Several standard lattices are included in Pecan. Chains of arbitrary length can obtained by using the function `chain()` or by using the natural and integer number posets `N` and `Z`. Free distributive lattices can be obtained from the function FDL(n) where `n` is the number of generating variables. All lattices of five or less variables are also defined. These standard lattices can be combined to define arbitrary lattices using a cut and paste technique. More information about this process is given in section four.

The table of meet and join operators on a lattice can be printed using the `display` command. The command will list the element's enumerated and set labels followed by the product table and a list of meet and join irreducibles and how they are related under the ~-relationship. To save space the meet and join tables are printed together with meets occupying the upper triangular portion and joins occupying the lower triangular portion.

Congruences

Congruences can either be defined by coercing an expression of type relation or by specifying the set of join-irreducibles that determines the congruence. If the former method is used then upon evaluation of the congruence all join-irreducibles not related to the element they cover are located and the ~-closure of this set is used to determine the congruence. In the case that the relation expression is a congruence then the resulting congruence is equal to the relation. If however the relation was not a congruence then obviously there may be little connection between the two. (The only thing that can be said is that they agree on a subset of the join-irreducibles.) If the second method is used then the ~-closure of the set of join-irreducibles is calculated and this set is then used to

determine the congruence.

Congruence variables have a new member function `.join` which returns the set of join-irreducibles determining the congruence. Since it is required that the ~-closure of the set of join-irreducibles determining the congruence is calculated, any definition of a congruence has to be bound to a definition of a lattice. This is expressed by casting a congruence or relation expression, stating the lattice to which the congruence is to be bound:

```
C := (congruence on L) R
```

In this case `C` is a congruence on `L` generated by the relation `R`. The congruence class of an element `e` is given by the expression `[e]C`. To change the base lattice of a congruence simply re-cast it onto the new lattice. Since the relation/congruence expression is bound to the lattice `L`, all occurrences of the operators `\/` and `/\` will be taken to apply to `L` rather than the most recently opened lattice.

The lattice of congruence relations on a lattice can be obtained from the function `congruences()`. The elements of the lattice are sets of join-irreducibles determining the congruences on the base lattice

Homomorphisms

Homomorphisms can be defined in much the same way as maps, however like congruences it is necessary to specify the lattices between which the homomorphism acts:

```
H := (homomorphism L1 to L2) M
```

If the homomorphism is defined by using an expression bound to a generator then all occurrences of the operators `/\` and `\/` in the expression will be taken to be in the range lattice. If the expression has guards then occurrences of the operators in the guards will be taken to be in the domain lattice. In this way it is possible for the same map definition to be used between several different lattices.

3. Quotient and Product Lattices

Lattices, congruences and homomorphisms are all connected by quotient lattices in the Homomorphism Theorem which states that every homomorphic image of a lattice L is isomorphic to a suitable quotient lattice of L. More precisely if $\phi: L \rightarrow L_1$ is a homomorphism from L onto L_1 and Φ is the congruence relation on L defined by

$$x \equiv y \ (\Phi) \text{ if and only if } x\phi = y\phi \qquad (1)$$

then $L/\Phi \cong L_1$ and the map ψ: $[x]\Phi \rightarrow x\phi$ is a isomorphism.

If `L` is a lattice variable and `C` congruence variable bound to `L` then the quotient lattice can be defined as follows:

```
Q := L / C
```

The elements of `Q` will be presented as intervals $[e_1,e_2]$ of `L`. Each element of `Q` will be assigned an enumeration label according to its topological order in `Q` and a set label of the form `:#`n_1_n_2 where n_1,n_2 are the enumeration labels of the minimal and maximal

elements in the congruence class in L.

The natural homomorphism H:L → Q can be defined as follows:

```
H := (homomorphism L to Q) {($1, [$1]C) | $1 in L }
```

This definition however does not specify an inverse transform and is hence very inefficient in calculating the pre-image of an element in Q. Since all of the information necessary to calculate both the image and pre-image of the natural homomorphism is given by the set and enumeration labels of L and Q the natural homomorphism can be defined just by saying:

```
H := (homomorphism L to Q) natural
```

If L1 and L2 are two isomorphic lattices then the function isomorphism() returns an isomorphism from L1 to L2. If L1 and L2 are not isomorphic then an empty map (causing an error if used) is returned. The kernel of a homomorphism (that is the congruence relation defined by (1) above) can be obtained by the function kernel()

```
C1 := kernel( H1 )
```

The congruence C1 is bound to the domain of H1. If S is a subset of a lattice L then the sublattice generated by S is obtained by the function sublattice() and the embedding of S into L is obtained from embedding().

The product of two lattices L and M is given by L*M. Each element (x,y) in the product is given the set label :#n_1_n_2 where n_1,n_2 are the enumeration labels of the elements x and y in L and M. The projection homomorphism from L*M to L and M is defined by saying:

```
P     := L * M
left  := (homomorphism P to L) projection
right := (homomorphism P to M) projection
```

Example

If L, L1 are lattices and H is a homomorphism from L to L1 then the homomorphism theorem can be demonstrated by the following definitions and queries:

```
IH := H(L)                  // Calculate image of H.
L2 := sublattice(IH,L1)
H2 := embedding(L2, L1)
C  := kernel(H)             // Obtain suitable congruence
Q  := L/C                   // and quotient.
H1 := (homomorphism L to Q) natural
I  := isomorphism(Q,L2)

print Q = L1                // True if H is onto.
print Q = L2                // Isomorphic lattices.
print I@H1 = H2@H           // Identical maps.
```

Sets and Orders

Any set or order can be bound to a lattice to express the fact that all lattice operations are to be performed in that lattice. For example a set P can be bound to a lattice M by casting P

```
P := (set on M) { ... }
```

To bind P onto a new lattice then P has just to be recast, not redefined,

```
P := (set on L)
```

By casting sets onto lattices operators like tilde(), meet(), join() etc. know within which lattice to work. If the set is not bound to any lattice then the lattice has to be provided as an argument to the function.

Orders can also be bound to a lattice, in this case only the relation has to be provided since the base set is taken to be the lattice. While general orders do not need to be bound to a lattice, certain functions that return lattice pre-orders do need to be bound to be used. For example the function preorder(X,Y) returns the pre-order determined by the sets of join-irreducibles X and Y.

```
pre := (order on L) preorder(X,Y)
```

Hence the lattice L has two orders defined on it, the partial order =L> and the pre-order =pre>. Orders can be carried through to quotient lattices by using the natural homomorphism. For example the pre-order pre defined above can be defined on a quotient lattice Q of L by saying

```
preQ := (order on Q) natural(pre)
```

It is assumed that the pre-order pre respects the operations of lattice, if it dosn't then the resulting order is undefined.

Computational Equivalence and Replaceability

Special functions exist in Pecan to determine the computational equivalence congruence and replaceability pre-order. If L is a lattice and f an element in L then the functions minimal(L,f) and maximal(L,f) return the set of minimal meet-irreducibles greater than f and the set of maximal join-irreducibles less than f respectively. The set of elements contained in a set P that are less than an element e is given by P(e). In this case the set P must be bound to a lattice. The sets of join-irreducibles that determine the replaceability pre-order are obtained by the functions comp_rep_p(L,f) and comp_rep_q(L,f). Hence the replaceability pre-order and the computational congruence can be defined by

```
Pf  := comp_rep_p(L,f)
Qf  := comp_rep_q(L,f)
pre := (order on L) preorder(Pf,Qf)
equ := (congruence on L) ( Pf + Qf )
```

The pre-order and congruence can be defined directly however by using the functions comp_rep(L,f) and comp_equ(L,f).

4. Constructing Lattices

While lattices can be defined in terms of quotients, images, products etc. of other lattices and partial orders it is necessary at some point to explicitly define a lattice without reference to any other variable. As was indicated in the section two, lattices can be defined by expressing a partial order or an algebra where the corresponding sets of ordered pairs (order relation in the case of a poset and the meet and join operators in the case of an algebra) are given in full. Even in a small lattice the definition of the partial order by this method would be extremely tedious and likely to be error prone. The use of generators and pattern matching allows larger lattices to be defined, however lattices defined this way tend to have an extremely uniform structure (eg. total orders or boolean algebras) and hence restrict their use.

To allow larger and more complex lattices to be defined Pecan includes several *basic* and *pre-defined* lattices which can be used to construct general lattices. The basic lattices consist of all the lattices with five or less elements, the eight element lattice 2^3, the free boolean lattice on two variables FBL(2) and the free distributive and the free modular lattices on three variables, FDL(3) and FML(3). The pre-defined lattices consist of the free distributive lattices on eight or less variables. The difference between basic and pre-defined lattices is that basic lattices are stored explicitly while the pre-defined lattices are represented and manipulated algebraically.

Construction of Lattices from Basic Lattices.

Arbitrary lattices can be constructed by combining several basic lattices to represent an ordered set, which is then coerced into a lattice. The general principal behind this method is that the user draws a Hasse diagram of the lattice then identifies overlapping sublattices of the sort given in the basic set. The Hasse diagram is finally reconstructed by identifying overlapping elements of the basic lattices producing a connected diagram. This process only creates an ordered set, this set has then got to be converted into a partial order and finally a lattice. However it does produce concise and easy method of constructing arbitrary lattices.

To isolate the construction process from the main definitive environment the lattice construction is performed in a separate environment, hence only the resulting ordered set produced by the construction will be visible to the definitive environment. All lattice constructions are stored in a separate file so that the user can redefine and reuse lattices from previous sessions with Pecan. New lattices are constructed by the command `construct` <*latticename*> which opens the file containing the user constructed lattices and allows the user to edit the file. When the lattice has been entered the lattice name can then be used as a basic lattice, including the construction of further lattices. To read in a pre-constructed lattice from a file or write the current definition of a lattice then the commands `read` <*latticename*> and `write` <*latticename*> should be used.

A lattice construction consists of a block of declarations where identifiers are assigned to basic lattices, followed by a block of element identifications where elements in different lattices are identified and finally followed by a block of order relations where individual elements can be ordered. To make the last two stages easier all the elements of a basic lattice are given set labels so that the elements can be identified. These set labels are carried through to the final lattice and provide an efficient method of labelling the elements.

The non-totally ordered basic lattices of five or less elements with their set labels are given in figure 1.

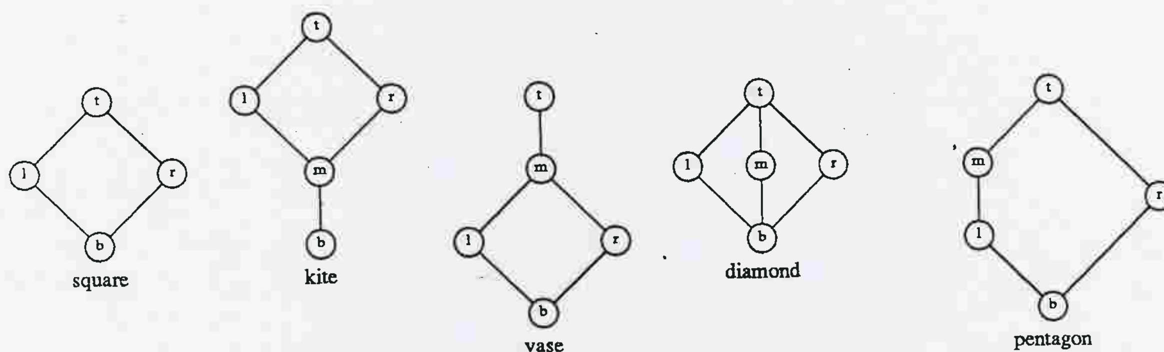

figure 1

The set labels given by default to these lattices can be overridden when the basic lattice is first used in the declaration section by specifying alternative labels as arguments to the basic lattice. The alternative labels should be given in the order bottom, left, middle, right, top. For example the declaration

```
V := vase( "", "x", "z", "y", "" )
```

would leave `V:t` and `V:b` unaltered but change `V:l`, `V:m` and `V:r`.

Chains of elements can be declared with the `chain()` function. The elements are given enumeration labels starting at 0, and the top and bottom elements are given set labels `t` and `b`. Unless labels are given as arguments the middle elements in the chain will be given set labels of the form `C`*i*.

As well as being able to use basic lattices in the declaration block it is also possible to use cross products of basic lattices. In this case the set labels of the elements of the lattices are concatenated together. In this way the basic lattices 2^3 and FBL(2) are equivalent to the declaration `chain(2)*square` and `square*square`.

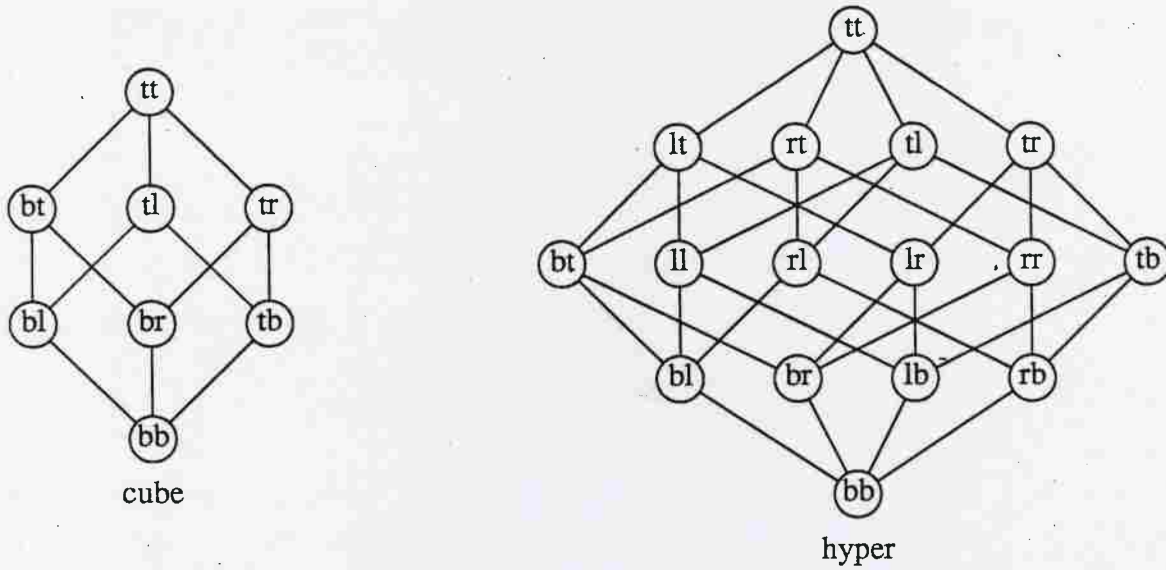

figure 2

As was said earlier set labels from the construction phase are carried forward into the Pecan environment. In the case where a set label of a basic lattice is not altered (eg. `V:t` from above) then the set label given to Pecan will be the concatenation of the variable name and the set label (eg. :`V_t` from above). If the set label of the basic lattice is changed then the label used by Pecan will be just the label given (eg. :x from above). If two elements are identified then the label of the element on the right will be used for both elements.

The lattice FML(3) is given as a basic lattice, however as an example it can be constructed as follows.

```
FML3 := {
// Define lattices
     B := cube ; M := diamond ; T := cube

     X := square("", "x"," mx", "" );
     Y := square("", "my", "y", "" );
     Z := square("", "mz", "z", "" );

// Identify common elements
     M:t := T:bb; M:b := B:tt
     M:l := X:mx; M:m := Y:my; M:r := Z:mz

// Add remaining order relations
     T:bl > X:t > X:b > B:bt
     T:br > Y:t > Y:b > B:tl
     T:tb > Z:t > Z:b > B:tr
}
```

There are obviously several methods of constructing FML(3), the above was chosen here to demonstrate all three stages of the construction. The final Hasse diagram with labels is given in figure 3. As can be seen several of the elements have unusual labels, these can be changed by the user using the normal procedure (eg. `L:T_bb:=:mt`). The set labels used for the lattice FDL(3) is given in figure 4.

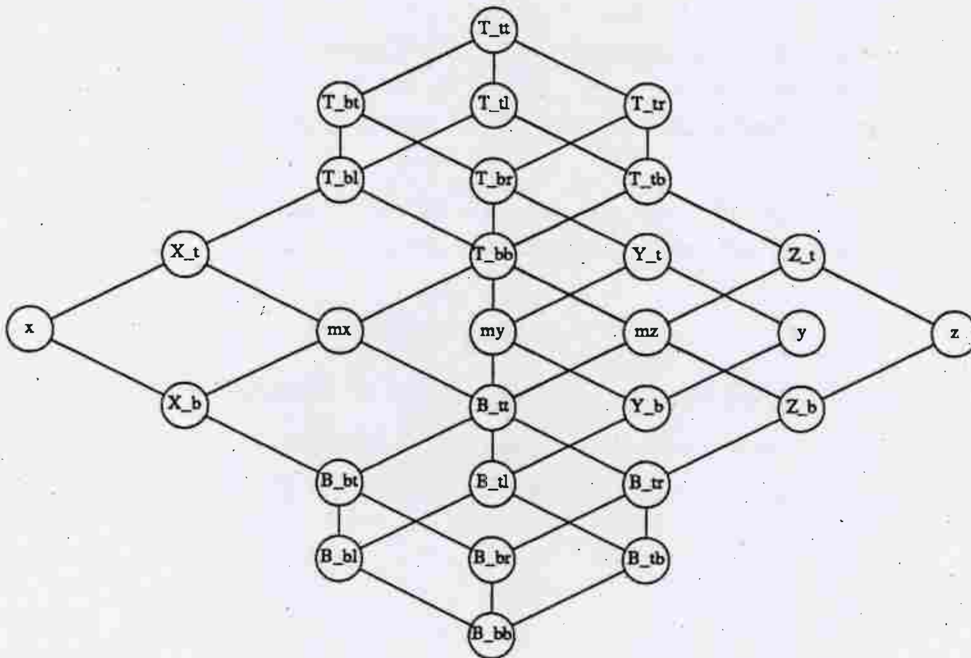

figure 3

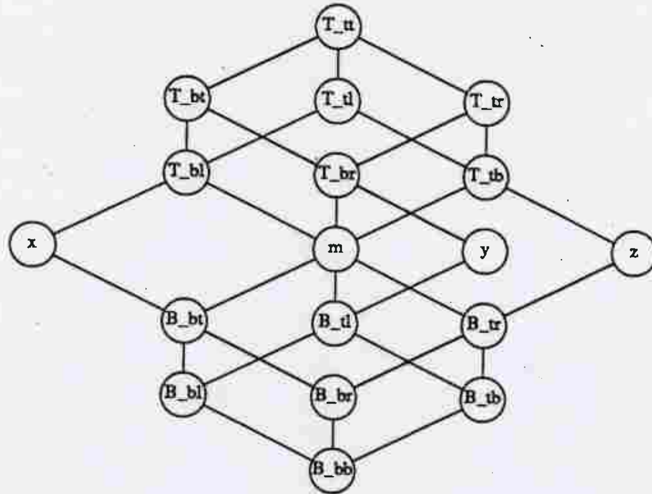

figure 4

Pre-defined Lattices

The pre-defined lattices consist of the free distributive lattices on three to eight variables. These lattices can not be represented in the normal fashion due to their size, however they can still be manipulated in the same way by using algebraic identities. The elements of the lattice are represented in disjunctive normal form by set labels, for example the element $x_1x_2x_4 \vee x_2x_3 \vee x_3x_4$ would be represented by the label :`abd_bc_cd`. Many of the operations associated with normal lattices can be applied to the pre-defined lattices, including quotients, homomorphisms, products etc. However operations which would result in enumerating the entire lattice or explicitly constructing a lattice too large for Pecan will be stopped.

The pre-defined free distributive lattices are obtained from the function `FDL`(). It should be noted that the basic lattice isomorphic to FDL(3) is called `FDL`3, not `FDL`(3).

5. Predefined Functions and Operators

Operator	Operation
`L/C`	Quotient lattice of L over C
`L*M`	Cross product of two lattices
`/\`	meet of two elements
`\/`	join of two elements
`/L\`	meet of two elements in the lattice L
`\L/`	join of two elements in the lattice L
`[e]C`	congruence class of *e* in C
`P(e)`	elements $\leq e$ in P

Function	Argument Type	Operation
kernel(h)	homo	Returns the kernel of h
congruences(L)	lattice	Returns the lattice of congruence relations
comp_rep(f)	element	Returns the replaceability pre-order $\sqsubseteq_f$
comp_equ(f)	element	Returns the computation equivalence congruence $\square_f$
square()	--	
pentagon()	--	
diamond()	--	
vase()	--	
kite()	--	
chain()	--	Returns a basic lattice
cube()	--	
hyper()	--	
FDL3()	--	
FML3()	--	
FDL()	--	Returns a pre-defined lattice
is_lattice()	--	
is_homo()	--	True if operand is of that type
is_congruence()	--	
is_distributive()	lattice	True if operand is an distributive lattice

Function	arg1	arg2	Operation
meet(L,X)	lattice	set	Returns the meet of X
join(L,X)	lattice	set	Returns the join of X
tilde(L,X)	lattice	set	Returns the ˜ of X
dual(L,x)	lattice	element	Returns the dual of x in a pre-defined lattice
sublattice(S,L)	set	lattice	Returns the lattice generated by S in L
embedding(K,L)	lattice	lattice	Returns the embedding K→L
isomorphism(K,L)	lattice	lattice	Returns an isomorphism from K to L
is_join_irr(L,x)	lattice	element	True if operand is a join-irreducible
is_meet_irr(L,x)	lattice	element	True if operand is a meet-irreducible
minimal(L,f)	lattice	element	Returns the minimal meet-irreducible $\geq$ f
maximal(L,f)	lattice	element	Returns the maximal join-irreducible $\leq$ f

is_closed(L,X)	lattice	set	True if operand is closed under the ~-relation
closure(L,X)	lattice	set	Returns the tilde closure of the operand
preorder(X,Y)	set	set	Returns the pre-order determined by X and Y
comp_rep_p(L,f)	lattice	element	Returns the sets determining the replaceability
comp_rep_q(L,f)	lattice	element	pre-order $\sqsubseteq_f$ on L

Appendix Three

Prime
Source Listings

NAME

prime – monotone boolean desk calculator

SYNOPSIS

prime

DESCRIPTION

Prime displays the disjunctive and conjunctive normal forms of monotone boolean expressions entered by the user and constructs planar monotone circuits. Arbitrary monotone expressions can be entered with the lower case letters "a" to "t" representing the free variables and the symbols "+" and "." representing boolean disjunction and conjunction. Conjunction has higher precedence than disjunction and the period symbol can be omitted, brackets can be used to force a particular evaluation order.

There are 14 expression variables denoted by the upper case letters "A" to "N" which can be assigned a value. The statment

F = a(b+c)+de

will set F to the value "a.b+a.c+d.e" and print the DNF and CNF of this expression. The statment

F = a(b+c)+de

will assign F to the same value however will not print the result.

The special variable V stores the conjunction of the default set of free variables, initial set to "abcd". This value is used in the definitions of the threshold function T() and the functions u(), z(), U(), Z() (see below). To change the default set of variables V should be explicitly assigned the new *number* of default variables (hence the default variables always start from the variable "a").

The function **T** (*threshold_number* , *variable_range*) returns the threshold *threshold_number* function on *variable_range* . If *variable_range* is missing then the default variable range is used instead. For example to find the DNF and CNF of the threshold 3 function on variables b,d,e,f,g type

T(3,bdefg)

The functions **u** (*expression*), **z** (*expression*), **U** (*expression*) and **Z** (*expression*), return the unit, zero, μ and λ functions of the expression. Complements are taken with respect to the default set of variables unless the expressions involve new variables which are then included.

The functions **X** (*exp_1* , *exp_2*) and **Y** (*exp_1* , *exp_2*) return the minimum and maximum functions respectively that contain the prime implicants in *exp_1* and prime clauses in *exp_2* . The function **W** (*expression*) returns the dual of the expression.

Planar monotone circuits of functions can be constructed (if they exist) by giving the function as an argument to the **pmc** command:

pmc *expression*

The variables a,b,c... are assumed to be in left-right order and are initially referred as gates 1,2,3... *Prime* lists a sequence of gate operations either explicitly stating the individual gates or using the pyramid shorthand (see below and reference). When the **verbose** option is on a bit-table of gate values against prime implicants and clauses is displayed.

PYRAMIDS

The pmc command constructs the circuit using OR and AND pyramids. These consist of two pyramids slotted tip to tip resulting in a network that replaces the middle gates. For example a "v-pyramid from 1 to 5" is a truncated OR pyramid of 9 gates with an upside down AND pyramid of 6 gates slotted into the vacant apex of the OR pyramid. The three AND gates at the base of the pyramid now replace gates 2,3 and 4. The inputs to the outer AND gates are connected to the outputs of outer OR gates in order, starting with input of the new 2 gate joining the OR of gate 1 and the old gate 2 and ending with the input of the new 4 gate joining the OR of gate 4 and 5.

AUTHOR

John Buckle

SEE ALSO

For details on u(), z(), U() and Z() see:
BEYNON: Replaceability and computational equivalence for monotone boolean functions; (Acta Informatica, 22, 1985).

For details on planar monotone functions see:
BEYNON, BUCKLE: On the planar monotone computation of boolean functions; (Theoretical Computer Science, 53, 1987).

BUGS

This program is not designed to behave well when given wrong data, please do not expect sensible answers.

```
/**************************************************************************/
/*                                                                        */
/*                  P R I M E   D E S K   C A L C U L A T O R             */
/*                                                                        */
/*                  com.c - main arthimatic evaluation module             */
/*                                                                        */
/**************************************************************************/

#include "header.h"

FILE        * fp, * popen() ;                                              popen

NUMBER      zero = 0,
            * variables[2][14] ;

char        * morefilter, * getenv() ;                                     getenv

int         Noofvars = 4 ;

main()                                                                     main
{
register i;

            verbose = TRUE ;

            for ( i = 0; i < 14; i++ )
                    { variables[ DUAL ][i] = variables[ PRIME ][i] = & zero ; }

            if ( (morefilter = getenv( "PAGER" )) == NULL )
                    morefilter = "/usr/ucb/more" ;

            do {
                    printf(">> ") ;
                    fflush( stdout ) ;
            } while ( yyparse() ) ;
}

setvariable( var, poi, printflag )          /* Set a variable to the value       setvariable
int         var, printflag ;                /* of the expression.
POINTER poi ;
{
NUMBER      * primewalk() ;

            if ( var < 'A' || var > 'N' ) return;
            var -= 'A';

            variables[ PRIME ][var]         = primewalk( poi, PRIME ) ;
            variables[ DUAL  ][var]         = primewalk( poi, DUAL ) ;

            if ( ! printflag ) return;

            printvalues( variables[ PRIME ][ var ], variables[ DUAL ][ var ] ) ;
}

NUMBER      *
getvariables( from )                        /* Return the conjunction of         getvariables
int         from ;                          /* the current set of variables
{
NUMBER      * p ;
int         i ;
            if ( from == PRIME ) {
                    p = num_alloc( 2 ) ;
                    p[1] = p[0] = 0 ;
                    for ( i = 0 ; i < Noofvars ; i ++ )
```

```
                        p[0] = ( p[0] << 1 ) | 01 ;
                }
        else    {
                p = num_alloc( Noofvars + 1 ) ;
                p[ Noofvars ] = 0 ;
                for ( i = 0 ; i < Noofvars ; i ++ )
                        p[i] = 01 << i ;
                }
        return( p ) ;
}

treewalk( pointer )                     /* This is called when input is
POINTER pointer;                        /* complete and evaluation begins.
{
NUMBER  * pi, * pq, * primewalk() ;

        pi = primewalk( pointer, PRIME ) ;
        pq = primewalk( pointer, DUAL ) ;

        printvalues( pi, pq ) ;
}
```

treewalk

```
        /* This routine returns a pointer to a string of unsigned                */
        /* integers terminated by a zero marker. Variables are stored            */
        /* as bit vectors, variable 'a' being bit 0 and variable 't'              */
        /* being bit 19.                                                          */
NUMBER  *
primewalk( pointer, from )
POINTER pointer ;
int     from ;
{
int     i, k, lenp1, lenp2 ;
NUMBER  * p1 , * p2, * p3, * threshold(), * getvariables(),
        * uzfunc(), * UZfunc(), * XYfunc() ;

        switch ( pointer->OP ) {
        case 'T' :              return( threshold( pointer, from ) ) ;
        case 'V' :              return( getvariables( from ) ) ;
        case 'W' :              return( primewalk( pointer->LHS, ! from ) ) ;
        case 'u' :              return( uzfunc( pointer, from, UFUNC ) ) ;
        case 'z' :              return( uzfunc( pointer, from, ZFUNC ) ) ;
        case 'X' :              return( XYfunc( pointer, from, XFUNC ) ) ;
        case 'Y' :              return( XYfunc( pointer, from, YFUNC ) ) ;
        case 'U' :              return( UZfunc( pointer, from, UFUNC ) ) ;
        case 'Z' :              return( UZfunc( pointer, from, ZFUNC ) ) ;
        }

        if ( islower( pointer->OP ) ){
                /* We are at a leaf, start a new string                  */
                p1      = num_alloc( 2 ) ;
                p1[0]   = 01 << (pointer->OP - 'a') ;
                p1[1]   = 0;
                return( p1 ) ;
                }

        if ( isupper( pointer->OP ) ){
                /* We are at a function, insert value.                   */
                k = pointer->OP - 'A' ;
                if ( k > 14 || k < 0 ) k = 0 ;
                lenp1   = veclen( variables[ from ][k] ) ;
                p1      = num_alloc( lenp1 + 1 ) ;
                for ( i = 0 ; i <= lenp1 ; i++ )
                        p1[i] = variables[ from ][k][i] ;
```

primewalk

```
                return( p1 ) ;
                }

                                        /* Calculate ./+ of the left and right
                                        /* subtrees recursively.
        p1 = primewalk( pointer->LHS, from ) ;
        p2 = primewalk( pointer->RHS, from ) ;
        lenp1 = veclen( p1 ) ;
        lenp2 = veclen( p2 ) ;

        if ( (pointer->OP == '.')  ==  (from == PRIME) ){
                                /* Must concatenate each word form the left
                                /* with each word from the right.
                p3 = num_alloc( lenp1*lenp2+1 ) ;
                p3[ lenp1*lenp2 ] = 0 ;
                for ( i = 0 ; i < lenp1 ; i++ )
                        for ( k = 0 ; k < lenp2 ; k++ )
                                p3[i*lenp2+k] = p1[i] | p2[k] ;
                }
        else {
                                /* Must string all words from both subtrees
                                /* together.
                p3 = num_alloc( lenp1+lenp2+1 ) ;
                p3[ lenp1+lenp2 ] = 0 ;
                for ( i = 0 ; i < lenp1 ; i ++ )
                        p3[i] = p1[i] ;
                for ( k = 0 ; k < lenp2 ; k ++ )
                        p3[k+lenp1] = p2[k] ;
                }

        free( (char *) p1 ) ; free( (char *) p2 ) ;
        removewaste( p3 ) ;
        return( p3 ) ;
}
```

removewaste

```
removewaste( p )                        /* This removes redundent implicants
NUMBER  * p ;                           /* from the function p.
{
int m, k, i ;
        m = veclen( p ) ;
        for ( i = 0 ; i < m-1 ; i++ )
                for ( k = i+1 ; k < m ; k++ ) {
                        if ( (p[k] & p[i]) == p[i] ){
                                p[k--] = p[--m] ;               /* p[i] <= p[k]
                                p[m] = 0 ;                      /* remove p[k]
                                }
                        else if ( (p[k] & p[i]) == p[k] ){
                                p[i] = p[k] ;                   /* p[i] > p[k]
                                p[k] = p[--m] ;                 /* remove p[i]
                                p[m] = 0 ;
                                k = i ;
                                }
                }
}
```

veclen

```
veclen( p )
register NUMBER     *p ;                /* Return the number of clauses
{                                       /* or implicants stored.
register int i = 0 ;
        while ( * p++ ) i ++ ;
        return(i) ;
}
```

printvalues

```
printvalues( p, q )                     /* Open the output filter and
NUMBER   * p, * q ;                     /* then call primeprint().
{
int         pipeflag = TRUE ;

            if ( (fp = popen( morefilter, "w" )) == NULL ) {
                        fp = stdout ;
                        pipeflag = FALSE ;
                        }

            fprintf(fp, "Prime implicants -\t" ) ;
            primeprint( p ) ;

            fprintf( fp, "Dual  implicants -\t" ) ;
            primeprint( q ) ;

            if ( pipeflag) pclose( fp ) ;
}
```

primeprint

```
primeprint( p )                         /* Prints the variables stored in the
NUMBER   *p ;                           /* NUMBER string p.
{
int x, c ;
                                        /* Get the implicants/clauese in a
                                        /* consistent order.
            quicksort( p, 0, veclen( p ) - 1 ) ;

            p -- ;
            while (*++p) {              /* This loops until there are no words
                        c = 'a' ;       /* left.
                        x = * p ;
                        while (x) {     /* Loops until all variables have been
                                        /* printed from this word.
                                    if ( x & 01 ) fprintf(fp,"%c",c) ;
                                    c++ ;
                                    x >>= 1 ;
                                    }
                        fprintf(fp,"\n\t\t\t") ;
                        }
            fprintf(fp,"\n") ;
}
```

quicksort

```
quicksort( p, m, n )                    /* Standard quicksort algorithm.
NUMBER   * p ;
int         m, n ;
{
register i, j ;
unsigned q, k ;
            if ( m < n ){
                        i = m ;
                        j = n+1 ;
                        k = p[m] ;
                        while (i < j){
                                    do i++; while ( p[i] < k && i<=j ) ;
                                    do j--; while ( p[j] > k && i<=j ) ;
                                    if ( i < j ) { q = p[i] ; p[i] = p[j] ; p[j] = q ; }
                                    }
                        q = p[m] ; p[m] = p[j] ; p[j] = q ;
                        quicksort( p, m, j-1 ) ;
                        quicksort( p, j+1, n  ) ;
                        }
}
```

```
/************************************************************************/
/*                                                                      */
/*              P R I M E   D E S K   C A L C U L A T O R               */
/*                                                                      */
/*              func.c - T(), u(), z(), U(), X(), Y(), Z()              */
/*                                                                      */
/************************************************************************/
```

T

```
#include "header.h"

NUMBER  *
threshold( poi, from )              /* Return the prime implicant/clause
POINTER  poi ;                      /* representation of a threshold func.
int      from ;
{
int      i, j ;
NUMBER   * choose(), * ths, k ;
         j = poi->LHS->OP;                      /* Threshold number
         k = poi->RHS->OP;                      /* Identifers

         if (k == 0)
                 for ( i=0 ; i<Noofvars ; i++ ){ /* Default case, use
                         k <<= 1;               /* currently defined
                         k++;                   /* variables.
                         }
         i = bitcount( k ) ;

         if ( j <= 0 || j > i ) {
                 ths      = num_alloc( 1 ) ;    /* Handle zero and one
                 ths[0]   = 0 ;                 /* functions, (return
                 return( ths ) ;                /* the zero function).
                 }
         if ( from == DUAL ) j = i - j + 1 ;    /* Switch to dual case.
         return( choose( i, j, k ) ) ;
}
```

threshold

```
NUMBER *
choose( n, c, a )                   /* Return the list of combinations of c
int      c, n ;                     /* objects out of n objects present in
NUMBER   a ;                        /* the word a.
{
int      k, m ;
NUMBER   *ths, y, *z, u ;

         if ( c == 1 ) {                        /* End of recursion.
                 ths      = num_alloc( n+1 ) ;
                 ths[n]   = 0 ;
                 y        = 01 ;
                 while ( a ) {
                         if (a & 01) ths[--n] = y ;
                         y <<= 1 ;
                         a >>= 1 ;
                         }
                 return( ths ) ;
                 }

         k       = combinations( n, c ) ;       /* Find size of answer.
         ths     = num_alloc( k+1 ) ;
         ths[k]  = 0 ;

         y = 01 ; u = a ;                       /* Get first variable.
         while ( !(u & 01) ) {
                 u >>= 1;
                 y <<= 1;
```

choose

```
            }
        a ^= y;                                     /* Switch off that variable and
        z = choose( n-1, c-1, a );                  /* find all c-1 combinations.

        for (m = 0; z[m]; m++ )                     /* Transfer combinations across
                ths[m] = z[m] | y ;                 /* switching the variable on.
        free( (char *) z ) ;

        if ( n > c ) {                              /* Find all c combinations.
                z = choose( n-1, c, a );
                for ( k = 0; z[k]; k++ )
                        ths[m++] = z[k];
                }
        free( (char *) z ) ;
        return( ths );
}
```

combinations

```
combinations( n, c )                                /* Returns `n' choose `c'.
int     n, c ;
{
int     prod = 1, c2 ;

        if ( c == n-1 ) return( n ) ;               /* Handle big cases here.
        if ( c == n ) return( 1 ) ;

        for ( c2 = c ; c2 ; c2 -- ) prod *= n -- ;
        for ( ; c ; c -- ) prod /= c ;
        return( prod ) ;
}
```

bitcount

```
bitcount( u )                       /* Return the bitcount of the unsigned
NUMBER  u ;                         /* number in u (ie. number of variables
{                                   /* set in the word).
register NUMBER x ;
register j, i = 0 ;
        x = u ;
        for( j = 24 ; j ; j-- ){
                if (x & 01) i++;
                x >>= 1;
                }
        return( i );
}
```

getdual

```
NUMBER  *
getdual( func )                     /* Calculate the dual of func by reconstructing
NUMBER  * func ;                    /* a parse tree in DNF and then returning the
{                                   /* CNF of the tree.
POINTER buildtree() ;
        return( primewalk( buildtree( func, '.' ), DUAL ) ) ;
}
```

buildtree

```
POINTER
buildtree( p, c )                   /* Reconstruct a parse tree from the expression
NUMBER  * p ;                       /* in p, c = '.' or '+' which is the operator
char c ;                            /* which is to go in the inner clauses.
{
POINTER clause() ;
char d ;
        d = (c=='.') ? '+' : '.' ;                  /* Get the other operator.

        if ( p[1] == 0 ) return( clause( 'a', p[0], c ) ) ;
        else    return( makenode( d, clause( 'a', p[0], c),
                        buildtree( p+1, c ) ) ) ;
}
```

clause

```
POINTER
clause( ch, u, c)                       /* Reconstruct a '.' or '+' clause, returning
char ch, c ;                            /* the parse tree.
NUMBER u ;
{
        while (!(u & 01) ){                     /* Get the variable.
                u >>= 1;
                ch ++;
                }
        u >>= 1;                                /* Delete the variable.

        if (u) return( makenode( c,
                makenode( ch, PNULL, PNULL ), clause( ch+1, u, c ) ) );

        return( makenode( ch, PNULL, PNULL ) );
}
```

uzfunc

```
NUMBER *
uzfunc( poi, from, func )               /* Calculate the unit and zero
POINTER poi;                            /* functions of the function in
int from, func;                         /* poi. The u and z functions
{                                       /* are duals of each other.
int i;
unsigned com, *p1;

        if ( func == UFUNC )                    /* Get appropriate type
                p1 = primewalk( poi->LHS, DUAL );       /* of normal form.
        else    p1 = primewalk( poi->LHS, PRIME );

        com = 0 ;
        for (i=0 ; i < Noofvars ; i++){         /* Get range of active
                com <<= 1;                      /* variables.
                com ++;
                }

        for (i = 0 ; p1[i] ; i++ )              /* See if any more have
                while ( p1[i] > com ){          /* introduced.
                        com <<= 1;
                        com ++;
                        Noofvars ++;
                        }

        for (i = 0 ; p1[i]; i++ )               /* Calculate the tilde
                p1[i] ^= com;                   /* of each clause, and
                                                /* interpret the func.
        if ( (func == UFUNC)  ==  (from == PRIME) )     /* in the other form.
                return( p1 ) ;
        else    return( getdual( p1 ) ) ;
}
```

UZfunc

```
NUMBER *
UZfunc( poi, from , func )              /* Calculates the lambda and mu
POINTER    poi ;                        /* functions for the function
int        from, func ;                 /* in poi. Lambda and mu are
{                                       /* dual functions.
int        i, k, lenp1, lenp2 ;
NUMBER     *p1, *p2, *p3 ;

        if ( func == UFUNC ) {          /* Calculate u( poi ) and poi itself
                                        /* in terms of prime implicants.
                p1 = uzfunc( poi, PRIME, UFUNC ) ;
                p2 = primewalk( poi->LHS, PRIME );
                }
```

```
	else {				/* Calculate z( poi ) and poi itself
					/* in terms of prime clauses.
		p1 = uzfunc( poi, DUAL, ZFUNC ) ;
		p2 = primewalk( poi->LHS, DUAL ) ;
		}

	lenp1 = veclen( p1 ) ;
	lenp2 = veclen( p2 ) ;

	p3 = num_alloc( lenp1 + lenp2 + 1 ) ;
	p3[lenp1+lenp2] = 0 ;
					/* Combine the two expressions together
	for (i=0;i<lenp1;i++)		/* to get the final result.
		p3[i] = p1[i];

	for (k=0;k<lenp2;k++)
		p3[i+k] = p2[k];

	removewaste( p3 );

	if ( (func == UFUNC)  ==  (from == PRIME) )
		return( p3 );
	else	return( getdual( p3 ) ) ;
}
```

XYfunc

```
NUMBER *
XYfunc( poi, from , func )
POINTER	poi ;			/* Calculates the minimum and
int	from, func ;		/* maximum functions that have
{				/* the implicants on the left
NUMBER	* given, * given_poi,	/* and clauses on the right.
	* extra, * extra_poi,
	* result, * result_poi,	/* Minimum function is obtained
	com, com_var, c, u ;	/* using PRIME and DIS, maximum
int	i, sum ;		/* is obtained dually.

	if ( func == XFUNC ) {		/* Minimum function
		given = primewalk( poi->LHS, PRIME ) ;
		extra = primewalk( poi->RHS, DUAL ) ;
		}
	else {				/* Maximum function
		extra = primewalk( poi->LHS, PRIME ) ;
		given = primewalk( poi->RHS, DUAL ) ;
		}
					/* Find the current set of
	com = com_var = 0 ;		/* active variables.
	for ( i = 0 ; i < Noofvars ; i ++ ) {
		com_var <<= 1 ; com_var ++ ;
		}
	for ( given_poi = given ; * given_poi ; given_poi ++ )
		com |= * given_poi ;
	for ( extra_poi = extra ; * extra_poi ; extra_poi ++ )
		com |= * extra_poi ;

	while ( com > com_var ) {	/* Final result is the combined
		com_var <<= 1 ;		/* list of the `given' clauses
		com_var ++ ;		/* plus the complements of the
		Noofvars ++ ;		/* `extra' clauses, each with
		}			/* one variable missing.

	sum = 0 ;			/* Calculate upper bound on result size
	for ( extra_poi = extra ; * extra_poi ; extra_poi ++ )
		sum += bitcount( * extra_poi ) ;
```

```
        result = result_poi = num_alloc( sum + veclen( given ) ) ;

                                /* Go through each result clause and add all
                                /* the new clauses with one variable missing
        for ( extra_poi = extra ; * extra_poi ; extra_poi ++ ) {
                if ( bitcount( * extra_poi ) == 1 ) continue ;
                u = 01 ;
                c = * extra_poi ;
                while ( u <= c ) {
                        if ( u & c ) * result_poi ++ = (c ^ u) ^ com_var ;
                        u <<= 1 ;
                        }
                }
                                /* Add all the given clauses and remove waste
        for ( given_poi = given ; * given_poi ; given_poi ++ )
                * result_poi ++ = * given_poi ;
        * result_poi = 0 ;

        removewaste( result ) ;

        if ( ( func == XFUNC ) == ( from == PRIME ) )
                return( result ) ;
        else    return( getdual( result ) ) ;
}

NUMBER  *
num_alloc( n )
int     n ;
{
        return( (NUMBER *) calloc( (unsigned) n, sizeof( NUMBER ) ) ) ;
}
```

num_alloc

```
/*************************************************************************/
/*                                                                       */
/*        P M C - P L A N A R   M O N O T O N E   C O M P U T A T I O N  */
/*                                                                       */
/*        pmctree.c - planar circuit generation module.                  */
/*                                                                       */
/*************************************************************************/

#include "header.h"

#define   CON     1
#define   DIS     0
#define   PROPER  2
#define   LEFT    -1
#define   RIGHT   1

int       num,                        /* Number of free inputs
          num_ops,                    /* Number of gates used so far
          lenpf, lenqf,               /* Number of prime implicants/clauses
          activegates,                /* Number of gates still alive
          verbose,                    /* Whether to print full tables
          change ;

char  *   dead,                       /* Indicates if a gate is redunant
      **  PFX,                        /* Bit table of inputs and implicants
      ** QFX ;                        /* Bit table of inputs and clauses

pmc( tree )                           /* PMC - construct a circuit of the        pmc
POINTER   tree ;                      /* function specified in the parse tree
{
NUMBER    * pf, * qf ;
int       i ;

          pf = primewalk( tree, PRIME ) ;
          qf = primewalk( tree, DUAL  ) ;

          pmcinit( pf, qf ) ;         /* Initialise PFX, QFX, num etc.

          num_ops     = 0 ;
          change      = TRUE ;
          activegates = num ;

          while (change && activegates) {
                    change = FALSE ;
                    if ( finish() ) break ;
                    pyramid( PFX, QFX, lenpf, lenqf, DIS ) ;
                    pyramid( QFX, PFX, lenqf, lenpf, CON ) ;
                    if ( ! change )
                              for ( i = 0 ; i < num ; i++ ) improve( i ) ;
                    }
          if ( finish() )
                    printf( "Circuit constructed\n" ) ;
          else      printf( "Circuit does not exist\n" ) ;

          if ( ! verbose ) printvec() ;
}

pmcinit( pf, qf )                     /* Initailise the tables PFX, QFX so       pmcinit
NUMBER    * pf, * qf ;                /* PFX[i][j] is true if variable i is
{                                     /* in implicant j.
int       i, j, k ;
NUMBER    u = 0, v = 1 ;
```

...pmcinit

```
        lenpf = veclen( pf );                        /* Get the number of implicants
        lenqf = veclen( qf );                        /* clauses and find the number
                                                     /* of free variables in use.
        for( i = 0 ; pf[i] ; i++ ) u |= pf[i] ;
        for( num = 0 ; u ; num++ ) u >>= 1 ;

        PFX = (char**) calloc( (unsigned) num, sizeof( char* ) ) ;
        QFX = (char**) calloc( (unsigned) num, sizeof( char* ) ) ;
        dead= (char* ) calloc( (unsigned) num, sizeof( char   ) ) ;

        for ( i = 0 ; i < num ; i++ ){
                PFX[ i ] = (char*) calloc( (unsigned) lenpf, sizeof( char ) ) ;
                QFX[ i ] = (char*) calloc( (unsigned) lenqf, sizeof( char ) ) ;
                dead[i]  = FALSE ;
                }

        for( i = 0 ; i < num ; i++ ){
                for ( j = 0 ; j < lenpf ; j++ )
                                PFX[i][j] |= ((pf[j] & v) != 0) ;
                for ( k = 0 ; k < lenqf ; k++ )
                                QFX[i][k] |= ((qf[k] & v) != 0) ;
                v <<= 1 ;
                }

        if (verbose) printvec() ;
}
```

improve

```
improve( middle )                    /* Find two neighbours for middle and then try
int          middle ;                /* to build a circuit around them. Only place a
{                                    /* gate if it is going to be constructive.
int          left, right, alive ;

        if ( dead[ middle ] ) return ;

        alive =    find_neighbour( middle, & right, RIGHT ) &&
                   find_neighbour( middle, & left, LEFT ) ;

        if ( alive ) {
                binaryop( middle, left,  PFX, QFX, lenpf, lenqf, DIS ) ;
                binaryop( middle, left,  QFX, PFX, lenqf, lenpf, CON ) ;
                binaryop( middle, right, PFX, QFX, lenpf, lenqf, DIS ) ;
                binaryop( middle, right, QFX, PFX, lenqf, lenpf, CON ) ;

                tripleop( middle, left, right, PFX, QFX, lenpf, lenqf, DIS ) ;
                tripleop( middle, left, right, QFX, PFX, lenqf, lenpf, CON ) ;
                }
}
```

find_neighbour

```
find_neighbour( middle, neigh, dir )          /* Finds the next variable to
int         middle, * neigh, dir ;            /* middle that is distinct from
{                                             /* it. Returns false if middle
                                              /* is now redundant.
        for ( * neigh = middle + dir ;
                        0 <= * neigh && * neigh < num ; * neigh += dir ) {
                if ( dead[ * neigh ] ) continue ;

                if ( Lesseq( middle, * neigh ) ){
                        dead[ middle ] = TRUE ;
                        printf("Killing variable %d\n", middle+1 ) ;
                        activegates -- ;
                        return( FALSE ) ;
                        }
                else if ( Lesseq( * neigh, middle ) ){
                        dead[ * neigh ] = TRUE ;
```

..find_neighbour

```
                        activegates -- ;
                        printf("Killing variable %d\n", * neigh+1 );
                        }
                else break ;            /* Found a good neighbour.
        }
        return( TRUE ) ;
}
```

binaryop

```
binaryop( middle, other, FX1, FX2, len1, len2, op )
char **     FX1, ** FX2 ;                   /* Combine middle with other so that it
int         middle, other,                  /* gains some new implicants/clauses
            len1, len2, op ;                /* but dosn't lose anything.
{
int         i ;
char        op_char ;

            if ( other < 0 || other >= num ) return ;

            op_char = ( op == DIS ) ? 'v' : '^' ;

            if (        Included( FX1[middle], FX1[other],  len1 ) &&
                    !   Included( FX2[other],  FX2[middle], len2 ) ) {
                        for ( i = 0 ; i < len2 ; i++ )
                                FX2[middle][i] |= FX2[other][i] ;
                        printf("%d = %d %c %d\n", middle+1, middle+1,
                                                op_char, other+1 ) ;
                        if (verbose) printvec() ;
                        change = TRUE;
                        num_ops ++ ;
                        return ;
                        }
            return ;
}
```

tripleop

```
tripleop( middle, left, right, FX1, FX2, len1, len2, op )
char **     FX1, ** FX2 ;
int         middle, left, right,
            len1, len2, op ;
{
int         i ;
            if (left < 0 || right >= num ) return ;

            if (        Con_Union( FX2[left], FX2[right], FX2[middle], len2 ) &&
                    !   Inter_Con( FX1[left], FX1[right], FX1[middle], len1 ) ) {

                        print_triple( middle, left, right, op ) ;

                        for ( i=0; i < len1; i++ )
                                FX1[middle][i] |= (FX1[left][i] & FX1[right][i]) ;
                        if (verbose) printvec() ;
                        change = TRUE ;
                        return ;
                        }

            return ;
}
```

print_triple

```
print_triple( middle, left, right, op )
int         middle, left, right, op ;
{
            printf( "%d = ", middle + 1 ) ;
            num_ops ++ ;

            if (op == CON) {
```

```
                if ( Smalleq( left, middle ) )
                        printf( "%d v ", left + 1 ) ;
                else if ( Smalleq( middle, left ) )
                        printf( "%d v ", middle + 1 ) ;
                else {
                        printf( "(%d ^ %d) v ", left+1, middle+1 ) ;
                        num_ops ++ ;
                        }
                if ( Smalleq( right, middle ) )
                        printf( "%d\n", right + 1 ) ;
                else if ( Smalleq( middle, right ) )
                        printf( "%d\n", middle + 1 ) ;
                else {
                        printf( "(%d ^ %d)\n", middle+1, right+1 ) ;
                        num_ops ++ ;
                        }
                }
        else {
                if ( Smalleq( middle, left ) )
                        printf( "%d ^ ", left + 1 ) ;
                else if ( Smalleq( left, middle ) )
                        printf( "%d ^ ", middle + 1 ) ;
                else {
                        printf( "(%d v %d) ^ ", left+1, middle+1 ) ;
                        num_ops ++ ;
                        }
                if ( Smalleq( middle, right ) )
                        printf( "%d\n", right + 1 ) ;
                else if ( Smalleq( right, middle ) )
                        printf( "%d\n", middle + 1 ) ;
                else {
                        printf( "(%d v %d)\n", middle+1, right+1 ) ;
                        num_ops ++ ;
                        }
                }
}
```

Con_Union

```
Con_Union( first, second, third, length )            /* True if the union of
char    *       first, * second, * third ;           /* 1st and 2nd contains
int             length ;                             /* the 3rd.
{
int             i, contains ;
                contains = TRUE ;
                for (i=0; i<length; i++)
                        switch ( first[i]+second[i]-third[i] ) {
                        case 0 :
                        case 1 :
                        case 2 : break ;
                        default: return( FALSE ) ;
                        }
                return( contains ) ;
}
```

Inter_Con

```
Inter_Con( first, second, third, length )            /* True if the inter-
char    *       first, * second, * third ;           /* section of the 1st
int             length ;                             /* and 2nd is contained
{                                                    /* in the 3rd.
int i, contains ;
                contains = TRUE ;
                for (i=0; i<length; i++)
                        switch ( third[i] - first[i]*second[i] ) {
                        case 0 : break ;
                        case 1 : contains = PROPER ;
                                break ;
```

```
                default: return( FALSE ) ;
                }
        return( contains ) ;
}

Lesseq( first, second )                 /* True if the first is computationally
int first, second ;                     /* less than or equal to the second.    */
{
        if ( Included( PFX[first], PFX[second], lenpf ) &&
             Included( QFX[first], QFX[second], lenqf ) ) return( TRUE ) ;
        return ( FALSE ) ;
}
```

Lesseq

```
Smalleq( first, second )                /* True if the first is less than or
int first, second ;                     /* or equal to the second.
{
        if ( Included( PFX[first], PFX[second], lenpf ) &&
             Included( QFX[second], QFX[first], lenqf ) ) return( TRUE ) ;
        return ( FALSE ) ;
}
```

Smalleq

```
finish()                        /* Returns true if there is a variables whose
{                               /* PFX and QFX rows are all ones.
register i, j, ok ;
        ok = FALSE ;
        for ( i = 0 ; i < num && ! ok ; i++ ){
                ok = TRUE ;
                for ( j = 0 ; j < lenpf && ok ; j++ ) ok &= PFX[i][j] ;
                for ( j = 0 ; j < lenqf && ok ; j++ ) ok &= QFX[i][j] ;
                }
        return( ok ) ;
}
```

finish

```
printvec()
{
register i, j ;
        printf( "Vectors are :-\n" ) ;
        for ( i = 0 ; i < num ; i++ ){
                printf( "%3d - ", i+1 ) ;
                for ( j = 0 ; j < lenpf ; j++ ) printf( "%u ", PFX[i][j] ) ;
                printf( "  " ) ;
                for ( j = 0 ; j < lenqf ; j++ ) printf( "%u ", QFX[i][j] ) ;
                printf( "\n" ) ;
                }
        fflush( stdout ) ;
}
```

printvec

```
Included( set1, set2, length )
char *set1, *set2 ;
int  length ;
{
int contains = TRUE, i ;
        for ( i = 0 ; i < length ; i++ ){
                if (set1[i] > set2[i]) return( FALSE ) ;
                if (set1[i] < set2[i]) contains = PROPER ;
                }
        return( contains ) ;
}
```

Included

```
/*****************************************************************************/
/*                                                                           */
/*      P M C - P L A N A R   M O N O T O N E   C O M P U T A T I O N       */
/*                                                                           */
/*      pmc_pyr2.c -        pyramid construction for planar circuit          */
/*                          generation module.                               */
/*                                                                           */
/*      Proceeds by constructing AND/OR pyramids so that components          */
/*      can be bridged.                                                      */
/*                                                                           */
/*****************************************************************************/

#include "header.h"

#define CON        1
#define DIS        0

#define    R       0
#define    S       1
#define    A       2

pyramid( F1, F2, len1, len2, op )          /* Find all components in F1
char **    F1, ** F2 ;                      /* that arn't blocked in F2 and
int        len1, len2, op ;                 /* place a op-pyramid over it.
{
char   * in_gap ;                      /* These variables are used to find the
int    * start,                        /* longest gap between components for
       *      gap,                     /* each variable.
              i, v, filled ;

       in_gap     = malloc( (unsigned) len1 ) ;
       start      = ( int * ) calloc( (unsigned) len1, sizeof( int ) ) ;
       gap        = ( int * ) calloc( (unsigned) num,  sizeof( int ) ) ;

       for ( v = 0 ; v < num  ; v++ ) gap[v] = 0 ;
       for ( i = 0 ; i < len1 ; i++ ) {
              start[i] = 0 ;
              in_gap[i] = R ;
              }

       for ( v = 0 ; v < num ; v++ ) {
              if ( dead[v] ) continue ;
              for ( i = 0 ; i < len1 ; i++ )
                     switch ( in_gap[i] ) {
                     case R:   if ( F1[v][i] ) {
                                      in_gap[i] = S ;
                                      start[i] = v ;
                                      }
                               break ;
                     case S:   if ( F1[v][i] == 0 ) {
                                      in_gap[i] = A ;
                                      }
                               else   start[i] = v ;
                               break ;
                     case A:   if ( F1[v][i] ) {
                                      in_gap[i] = S ;
                                      if ( notblock( F2, len2, start[i], v ))
                                             gap[ start[i] ] = v ;
                                      start[i] = v ;
                                      }
                               break ;
                               }
              }
```

pyramid

```
	for ( filled = 0, v = 0 ; v < num ; v ++ )
		if ( gap[v] > filled ) {
			filled = gap[v] ;
			make_pyramid( v, gap[v], F1, F2, len1, len2, op ) ;
			}
	free( (char *) gap ) ;
	free( (char *) in_gap ) ;
	free( (char *) start ) ;
}
```

notblock

```
notblock( F2, len2, start, end )		/* Tests to see if there is a
char ** F2 ;					/* band of 1's totally within
int	len2, start, end ;			/* the start/end markers.
{
int	state, v, i ;

	for ( i = 0 ; i < len2 ; i ++ ){
		state = R ;
		for ( v = start; v <= end; v ++ ){
			if ( dead[v] ) continue ;
			switch ( state ) {
			case R:	if ( F2[v][i] == 0 ) state = S ;
				break ;
			case S:	if ( F2[v][i] == 1 ) state = A ;
				break ;
			case A:	if ( F2[v][i] == 0 ) return( FALSE ) ;
			}
		}
	}
	return( TRUE ) ;
}
```

make_pyramid

```
make_pyramid( start, end, F1, F2, len1, len2, op )
int	start, end, len1, len2, op ;
char **	F1, ** F2 ;
{
char	op_char ;
int	i, v, alive, middle, * last ;

	change = TRUE ;

	op_char = ( op == CON ) ? '^' : 'v' ;

	for ( alive = 0, v = start + 1 ; v < end ; v ++ )
		if ( ! dead[v] ) {
			alive ++ ;
			middle = v ;
			}
	if ( alive == 1 ) {
		tripleop( middle, start, end, F1, F2, len1, len2, op ) ;
		return ;
		}

	last	= ( int * ) calloc( (unsigned) len1, sizeof( int ) ) ;

	for ( i = 0 ; i < len1 ; i ++ )
		for ( v = end ; v >= start ; v -- )
			if ( F1[v][i] && ! dead[v] ) {
				last[i] = v ;
				break ;
				}

	for ( i = 0 ; i < len1 ; i ++ ) {
		if ( last[i] == 0 ) continue ;
```

```
            for ( v = start ; F1[v][i] == 0 || dead[v] ; v ++ ) ;
            for ( ; v <= last[i] ; v ++ ) F1[v][i] = 1 ;
            }

        printf( "%c - pyramid from %d to %d\n", op_char, start+1, end+1 ) ;
        if (verbose) printvec() ;
}
```

```
/************************************************************************/
/*
/*          P M C - P L A N A R   M O N O T O N E   C O M P U T A T I O N        */
/*                                                                               */
/*          header.h - external variables and functions                          */
/*                                                                               */
/************************************************************************/        */

#include <stdio.h>
#include <ctype.h>
#define     TRUE    1
#define     FALSE   0

#define     PRIME   1
#define     DUAL    0

#define     UFUNC   1
#define     ZFUNC   0

#define     XFUNC   1
#define     YFUNC   0

#define     PNULL   (POINTER) NULL

typedef     struct tnode {
            char OP ;                       /* '.' '+' or lower case letter */
            struct tnode *LHS ;             /* Left sub tree */
            struct tnode *RHS ;             /* Right sub tree */
            } TREE, *POINTER ;

typedef     unsigned NUMBER ;

char        * malloc(), * calloc() ;                                        malloc
NUMBER      * num_alloc() ;                                                 num_alloc

extern      int Noofvars ;
extern      int veclen() ;                                                  veclen

POINTER     makenode() ;                                                    makenode
NUMBER      * primewalk() ;                                                 primewalk

extern
int         num,                    /* Number of free inputs
            num_ops,                /* Number of gates used so far
            lenpf, lenqf,           /* Number of prime implicants/clauses
            activegates,            /* Number of gates still alive
            verbose,                /* Whether to print full tables
            change ;

extern
char *      dead,                   /* Indicates if a gate is redunant
     **     PFX,                    /* Bit table of inputs and implicants
     ** QFX ;                       /* Bit table of inputs and clauses
```

```
/****************************************************************************/
/*                                                                          */
/*                P R I M E   D E S K   C A L C U L A T O R                 */
/*                                                                          */
/*                bnf.y - grammar for the YACC(1) parser.                   */
/*                                                                          */
/****************************************************************************/

%{
#include "header.h"
%}

%union {
        int             INT ;
        unsigned        UNINT ;
        POINTER         NODES ;
        }

%token <INT>    IDENT   DIGITS  END  VAR  PMC  VERBOSE  ON  OFF  errSYM

%type   <UNINT>   idlist    Tidlist
%type   <NODES>   fact      exp       o_term    a_term
%type   <INT>     number

%start first

%%
first       :   prime   '\n'
                    { return( TRUE ) ; }

            |   error '\n'
                    { printf( ">> " ) ;  yyerrok ; }
                first
                    { ; }

            |
                    { return( FALSE ) ; }
            ;

prime   :   exp
                    { treewalk( $1 ) ; }

        |   '#' VAR '=' exp
                    { setvariable( $2, $4, 0 ) ; }

        |   VAR '=' exp
                    { setvariable( $1, $3, 1 ) ; }

        |   'V' '=' number
                    { Noofvars = $3;  if (Noofvars > 20) Noofvars = 20 ; }

        |   PMC  exp
                    { pmc( $2 ) ; }

        |   VERBOSE ON
                    { verbose = TRUE ; }

        |   VERBOSE OFF
                    { verbose = FALSE ; }

        |

        ;
```

YACC

return

printf

return

treewalk

setvariable

setvariable

pmc

```
exp         :   o_term
            ;

o_term      :   a_term
            |   a_term '+' o_term
                    { $$  =  makenode( '+', $1, $3 ) ; }                    makenode
            ;

a_term      :   fact
            |   fact a_term
                    { $$  =  makenode( '.', $1, $2 ) ; }                    makenode
            |   fact '.' a_term
                    { $$  =  makenode( '.', $1, $3 ) ; }                    makenode
            ;

fact        :   IDENT
                    { $$ = makenode( yylval.INT, PNULL, PNULL) ; }          makenode
            |   VAR
                    { $$ = makenode( yylval.INT, PNULL, PNULL) ; }          makenode
            |   'u' '(' exp ')'
                    { $$ = makenode( 'u' , $3, PNULL ) ; }                  makenode
            |   'z' '(' exp ')'
                    { $$ = makenode( 'z' , $3, PNULL ) ; }                  makenode
            |   'T' '(' number  Tidlist ')'
                    { $$ = makethreshold( $3 , $4 ) ; }                     makethreshold
            |   'U' '(' exp ')'
                    { $$ = makenode( 'U' , $3, PNULL ) ; }                  makenode
            |   'V'
                    { $$ = makenode( 'V', PNULL, PNULL ) ; }                makenode
            |   'W' '(' exp ')'
                    { $$ = makenode( 'W' , $3, PNULL ) ; }                  makenode
            |   'X' '(' exp ',' exp ')'
                    { $$ = makenode( 'X' , $3, $5 ) ; }                     makenode
            |   'Y' '(' exp ',' exp ')'
                    { $$ = makenode( 'Y' , $3, $5 ) ; }                     makenode
            |   'Z' '(' exp ')'
                    { $$ = makenode( 'Z' , $3, PNULL ) ; }                  makenode
            |   '('  o_term  ')'
                    { $$ = $2 ; }
            ;

number  :   DIGITS
                    { $$ = yylval.INT ; }
            ;

Tidlist :
                    { $$ = 0 ; }
            |   ',' idlist
                    { $$ = $2 ; }
            ;

idlist  :   IDENT
                    { $$ = 01 << $1 - 'a';}
            |   IDENT idlist
                    { $$ = ( $2 | (01 << $1 - 'a'));}
            ;

%%

POINTER
makenode( op, lhs, rhs )                /* This routine is called during the     makenode
```

...makenode

```
POINTER   lhs, rhs ;            /* parse of the input. Build a node to
int       op ;                  /* hold the operation and pointers to
{                               /* the subtrees.
POINTER pointer ;

          pointer = (POINTER) malloc( sizeof(TREE) ) ;
          pointer->OP = op ;
          pointer->LHS = lhs ;
          pointer->RHS = rhs ;

          return( pointer ) ;
}
```

makethreshold

```
POINTER
makethreshold( num, a )
int       num ;
NUMBER    a ;
{
POINTER makenode() ;
          return( makenode( 'T' , makenode( num, PNULL, PNULL ) ,
                                  makenode( (int) a, PNULL, PNULL ) ) ) ;
}
```

yyerror

```
yyerror( s )
char * s ;
{
          fprintf( stderr, "%s\n", s ) ;
}
```

```
/**********************************************************************/
/*                                                                    */
/*         P M C - P L A N A R   M O N O T O N E   C O M P U T A T I O N  */
/*                                                                    */
/*         text - tokens for Lex(1) lexical analyser generator        */
/*                                                                    */
/**********************************************************************/

%{
#include "header.h"
#include "y.tab.h"
%}

%%

quit            { return( 0 ) ; }
pmc             { return( PMC ) ; }
verbose         { return( VERBOSE ) ; }
on              { return( ON ) ; }
off             { return( OFF ) ; }

[\t ]           { ; }

[a-t]           { yylval.INT = yytext[0] ; return( IDENT ) ; }

[A-N]           { yylval.INT = yytext[0] ; return( VAR ) ;}

[TUVWXYZuz]     { return( yytext[0] ) ; }

[#,=+()$.]      { return( yytext[0] ) ; }

[0-9]+          { yylval.INT = atoi( yytext ) ; return( DIGITS ) ; }

[\n]            { return( '\n' ) ; }

%%
```

Lex

return
return
return
return
return
return
return
return
return
atoi
return