```
#
# To install a new system:
#       1) modify the DEST path
#       2) type "make install"
# After installing the system, add DEST path in your search path.
# To invoke fpl, just type "fpl".
#

#
# DEST specifies where to install the object file of fpl.
#
DEST=/u/mhmcheng/bin
SOURCE=main.pro compile.pro machine.pro type.pro read.pro util.pro\
       stmt.pro op.pro init.pro internal.pro
OBJ=$(DEST)/fpl.obp
STARTUP=$(DEST)/fpl


$(OBJ)  : $(SOURCE)
        cat $(SOURCE) banner > fpl.pro
        alspro fpl -g fpl
        mv fpl.obp $(DEST)
        chmod 755 $(DEST)/fpl.obp
        rm fpl.pro


$(STARTUP):
#
#       We assume "alspro" is accessible from anywhere, otherwise
#       use the following instead.
#
#       echo "$(DEST)/alspro $(OBJ) -q -g fpl" > $(STARTUP)
#
        echo "alspro $(OBJ) -q -g fpl" > $(STARTUP)
        chmod 755 $(STARTUP)


internal.pro :  predefined


install :       $(STARTUP) $(OBJ)
        touch install


print:  Makefile READ.ME banner manual predefined $(SOURCE)
#       lpr -Phw -p -i5 -J NEWFPL $?
        pr -b $? | lpr -Pll -Zu2 -h
        touch print

touch:
        touch Makefile READ.ME banner manual predefined $(SOURCE)
```


The FPL system consists of the following files:

Makefile	- a make file to install the system
manual	- a brief user manual
predefined	- the source file of all predefined functions
main.pro	- the top-level interpreter
type.pro	- the type inference subsystem
compile.pro	- the SECD machine code compiler
machine.pro	- the SECD machine emulator
read.pro	- the tokenizer
stmt.pro	- the parser
op.pro	- the operator parser
init.pro	- the primitive function database
util.pro	- the utility library
internal.pro	- all predefined functions and operators
banner	- the startup banner

For debugging purposes, we prefer to invoke ALS Prolog on the file

```
lisp.pro        - a startup file
```

which would consult all the necessary files to make up the FPL system.
To invoke fpl, you type

```
alspro lisp -g fpl
```

To add a new predefined function or operator,
1) add its definition in the file ''predefined'',
2) remove the files ''internal.pro'' and ''internal.obp'',
3) startup FPL by typing ''alspro lisp -g fpl'',
4) load the file ''predefined'',
5) save the code of all predefined functions in ''internal.pro'' by typing ''save:"internal.pro"'' (see note 3 below),
6) exit the system and restart.

Note:
---

(1)
Because of a bug in ALS Prolog, we temporarily disable the feature of user-definable constructor type. In general, a user can define his own constructor type. For example, to define a new data type of list of numbers, one can enter the following domain declaration:

```
type numlist = NIL  | CONS( num, numlist ).
```

Two constructors, NIL and CONS, are created for the data type ''numlist''. The expression ''CONS(1,CONS(2,NIL))'' is of numlist type.

The only way to bypass the bug (inside assert or retract) is to add the constructor type definition by hand directly into the constructor_type/2 relation in the file ''init.pro''.

(2)

Due to the lack of ''occur-check'' in ALS Prolog, the type checking algorithm may succeed in some cases where it should have failed. Whenever the type expression contains an infinite term, which is not a legal type expression, ALS Prolog will crash with the message ''Parser area exhausted'' as soon as the type expression is being asserted.

(3)
ALS Prolog cannot consult very long clauses (produced by write/1). If a clause is longer than 512 bytes, ALS Prolog would produce a segmentation error. When consulting the file ''internal.pro'', some of the very long clauses, generated by the SECD compiler, would choke ALS. The only way to get around it is to chop the long clauses by hand.


```
%
% Boot up the whole system
%
:- % system('clear') ,
   write( 'TYPE FPL (version 2)' ), nl,
   write( 'Type ''help.'''' for help.' ), nl.
```

F P L

Mantis H.M. Cheng

July 25, 1988

Introduction

FPL is a typed functional programming language supporting lazy evaluation, higher-order functions and universal polymorphism. The notation of FPL is modelled after the language ISWIM of Landin; it is a ''sugar'', typed lambda calculus.

There are two kinds of input to FPL, definitions and applications.
A definition must be preceded by the keyword ''let''. A recursive definition must have a ''rec'' immediately following the ''let''.

All functions in FPL are curried, i.e., all are unary functions.
A definition is a (symbol,expression) pair of the form:

```
let <symbol> = <expression>.
```

or

```
let rec <symbol> = <expression>.
```

This means that <symbol> has as value the value of <expression>. This (<symbol>,<expression>) pair, when successfully compiled, will be added to the current global environment. The global environment stores, in addition to user definitions, all predefined definitions.

All other inputs which are not definitions are to be taken as function applications, i.e., expressions to be evaluated.

Every input expression has a type. The user does not need to define the type of the expression. Rather, the system would infer from the expression a most general type for it, which may be polymorphic. A function is polymorphic if it is applicable to any type. For example, the identity function ''\x.x'' is applicable to an argument of any type; hence it is polymorphic.

Syntax

The following is a description of the language FPL in BNF notation. ''[<item>]'' means that <item> is optional. Terminal symbols are between single quotes ('), non-terminal symbols between angle brackets (< >), and comments between quotes (`` and ''). All other symbols belong to the meta-language, BNF.

```
<statement> ::= ( <expression> | <macro> | <op-declaration> ) '.'

<expression> ::= <simple-exp> |
                 <lambda-exp> |
                 <conditional-exp> |
                 <let-exp> |
                 '('<expression>')' |
                 <op-expression> |
                 <constructor-exp> |
                 <macro-exp>
```

(Note: All comment lines begin with a ''%''.)

```
<simple-exp> ::= <constant> | <list> | <symbol>

<constant> ::= <number> | <boolean> | '"'<string>'"' | '()'
<number> ::= [<sign>] (<integer> | <float>)
<float> ::= <integer>'.'<integer>[[<sign>]('E'|'e')<integer>]
<boolean> ::= 'true' | 'false'

<list> ::= '[]' | '[' <expression> '|' <expression> ']'

<symbol> ::= ''any alphanumeric identifier including '_',
                ''' and '?'''
<string> ::= ''any string of characters between double quotes''
```

Simple expressions are composed of constants, lists and function applications. For example, numbers, e.g.,-1.2,0,1.0e-3,..., are constants; quoted (") strings, e.g., "this", "is", ..., are constants; ''true'' and ''false'' are constants; ''()'' is a unique constant denoting void. ''[]'' denotes the empty list; [H|T] denotes a non-empty list with head H and tail T. For example, [1,2], [1|[2]] and [1|[2|[]]] are equivalent lists of two integers.

```
<lambda-exp> ::= '\'[<bound-vars>]'.'<expression>
<bound-vars> ::= (<symbol> | <constructor-pattern>) [','<bound-vars>]
```

(Note: In a lambda expression, the body expression must immediately follow the ''.'', i.e., there should not be any space.) The lambda

expression \x,y,z.<expression> is equivalent to \x.\y.\z.<expression>.

```
<conditional-exp> ::= <clause> <clauses>
<clause> ::= <condition>'->'<expression> |
             <expression>'when'<condition> |
             <expression>'otherwise' |
             'else' <expression>
<clauses> ::= ';'( <clause> <clauses> | <clause> | <expression> )
<condition> ::= <expression>
```

The general form of a conditional expression is ''C1->E1 ; ... ; [Cn->] En'' where n>1; the condition Cn is optional. An equivalent form of a conditional expression is ''E1 when C1; ... ; En when Cn; [En+1 otherwise]''.

```
<let-exp> ::= 'let' ['rec'] <binding> [<bindings> <body-exp>]
<binding> ::= <symbol>'='<expression> |
              <constructor-pattern>'='<symbol>
<bindings> ::= ','<binding> [<bindings>]
<body-exp> ::= 'in' <expression>
```

There are two forms of let-expression:

```
let [rec] x1=v1, ..., xn=vn in E        (1)
```

and

```
let [rec] N = E.                        (2)
```

(2) defines a new definition (N,E) to be added to the global environment. (1) is equivalent to the lambda expression (\x1,...,xn.E):v1:...:vn. A binding of the form f=\x.E can be rewritten as f:x=E. In general, f=\x1,...,xn.E is equivalent to f:x1:...:xn=E. (Note: There is a limitation of FPL on nested ''let rec'' of the form as in (1). The bindings must be functions (i.e., not simple expressions). Hence, you cannot define ''let rec x=1, y=x+2 in x+y'' where the ''x'' in ''y=x+2'' is ''x=1'' within the let-expression.)

```
<constructor-pattern> ::= ''any constructor with one or more
                           symbols as bound variables''
```

A constructor symbol is one which is used to construct a new data type. For example, the binary constructor 'BRANCH' is used to construct an expression of type ''extree'', which is an external tree. The expression ''BRANCH(x,y)'' is a valid constructor pattern, whereas ''BRANCH(1,x)'' is not. In other words, a constructor pattern is an expression of constructor symbols and bound variables. So, ((a,b),(c,d)) and [a,b,c] are valid

constructor patterns. (Note that ''[|]'' is a binary constructor.)

```
<op-expression> ::= <prefix-op> <expression>
                    <expression> <infix-op> <expression> |
                    <expression> <suffix-op>

<op-declaration> ::= <op-type> <op-assoc> <op-pred> <symbol>
<op-type> ::= 'infix' | 'prefix' | 'postfix'
<op-assoc> ::= 'left' | 'right' | 'non' | 'assoc'
<op-pred> ::= ''a positive integer between 0 and 1200''
```

The following declaration

```
infix left 500 "++".
```

defines a new infix operator ''++'' which is left-associative and has a precedence of 500. (Note: Prefix or postfix operators can only be 'non' (for non-associative) or 'assoc' (for associative).)

```
<macro> ::= 'macro' <named-exp> '=' <expression>
<named-exp> ::= <symbol>['('<parameters>')']
<parameters> ::= <symbol>','[<parameters>]

<macro-exp> ::= <symbol>['('<expressions>')']
<expressions> ::= <expression> [','<expressions>]
```

A macro definition is used to define a pattern to be replaced textually by another pattern. This replacement takes place before any evaluation. For example, the macro definition

```
macro from(x,y) = to:x:y.
```

says that any expression of the form ''from(x,y)'', where x and y are any subexpressions, is replaced by the expression ''to:x:y''. The expression ''from(a,b)'' is replaced by ''to:a:b''.

```
<constructor-exp> ::= <constructor>['('<expressions>')']
```

The set of valid constructors will be discussed in the section on types.

Features

If f is a function and x is an argument, then f:x or f@x is a function application, where ''':'' and ''@'' are infix function application operators which associate to the left and right respectively. For example, f:x:y means (f:x):y and f@x@y means f:(x:y).

Other than ''':'', and ''@'', there is a set of predefined operators denoting either built-in or predefined functions.

```
<relational-op> ::= '==' | '<>' | '<=' | '>=' | '<' | '>'
<arith-op>      ::= '+' | '-' | '*' | '/' | '^' | 'mod' | 'rem'
<bool-op>       ::= 'and' | 'or' | 'not'
<list-op>       ::= '..'
```

Operators	Type	Associativity	Precedence
=	infix	non	1000
;	infix	right	990
->	infix	non	950
when	infix	non	950
otherwise	postfix	non	950
else	prefix	non	950
or	infix	left	900
and	infix	left	850
not	prefix	assoc	800
<>	infix	non	700
>=	infix	non	700
<=	infix	non	700
==	infix	non	700
>	infix	non	700
<	infix	non	700
..	infix	non	600
+	infix	left	500
-	infix	left	500
*	infix	left	400
/	infix	left	400
mod	infix	non	400
rem	infix	non	400
^	infix	right	200
:	infix	left	100
@	infix	right	100
vi	prefix	non	100
vic	prefix	non	100
load	prefix	non	100
-	prefix	non	50
!	postfix	assoc	50

All of the above are predefined operators. The lower the precedence, the stronger the binding strength. For example, the infix ''*'' has a lower precedence than the infix ''+''. The expression ''2+3*4'' means ''2+(3*4)''. Using operator declaration, one can define the factorial function ''!'' as follows.


```
postfix assoc 50 "!".
let rec n! = n==0 -> 1; n*(n-1)!.
```

The expression ''3!!'' means ''(3!)!'' which evaluates to 720. Since all functions are curried, the expressions ''1+2'' and ''+:1:2'' are equivalent. For example, we can define the increment function ''++'' as follows.

```
prefix assoc 50 "++".
let ++ = +:1.
```

The expression ''++ ++3'' evaluates to 5.

A nullary function ''f'' can be defined as a unary function with ''()'' as its argument, e.g.,

```
let f:() = 1.
```

or

```
let f = \.1.
```

To apply this function, we must use the unique constant ''()''. Thus, the expression f:() evaluates to 1. Note that this definition is different from the following:

```
let f = 1.
```

The former is a constant function; the latter is just a constant.

Types

Every expression has a type as well as a value. A well-typed expression is a valid expression. There are three classes of type: basic type, structured type and function type. Numbers (denoted by ''num''), constant identifiers (denoted by ''sym'') and booleans (denoted by ''bool'') are of basic type.

```
type ()   = [ () ].
type num  = ''set of all numbers''.
type sym  = ''set of all symbols''.
```

```
type bool = {true,false}.
```

The simple type is the union of the three basic types with three distinct constructors. For example, NUM(x) is of simple type if x is of num type.

```
type simple = NUM(num) | SYM(sym) | BOOL(bool).
```

Lists are of structured type. Every list must be homogeneous, i.e., its element must be of the same type. For example, [1,2,3], ['a','b','c'] and [true,true,false] are well-typed lists; [1,'a',true] is not. The list type is defined by the following domain equation:

```
type list(x) = []  |  [ x | list(x) ].

type stream(x) = <<>>  |  << x | stream(x) >>.
```

where x is a type variable. It says that ''[]'' is a list of any type x; ''[h|t]'' is a list of type x if h is of type x and t is a list of type x. Hence, a list must be homogeneous. The symbols ''[]'' and ''[|]'' are constructors for the data type ''list''. The data type ''stream'' is used to construct infinite sequences which are to be evaluated ''lazily''. The symbols ''<<>>'' and ''<< | >>'' are constructors for the data type ''stream''. Other than lists and streams, FPL also has the following predefined structured types.

```
type tree(x) = EMPTY | NODE(tree(x),x,tree(x)).

type extree(x) = LEAF(x) | BRANCH(extree(x),extree(x)).

type (x1,...,xn) = TUPLE(x1,...,xn).    (for n>=2)

type op = OP(sym,sym,num).
```

A tree must also be homogeneous. There are two constructors for the data type ''tree'', EMPTY and NODE. But, a pair ''TUPLE(x,y)'', a triple ''TUPLE(x,y,z)'', a quadruple ''TUPLE(w,x,y,z)'', ..., may be of mixed type. For convenience, the constructor ''TUPLE'' may be omitted. The type ''op'' is defined for the set of operators, OP(Type,Assoc,Pred), which means that the operator is of type ''Type'', associativity ''Assoc'' and precedence ''Pred''.

If f is a function which takes arguments of type X and produces results of type Y, then f is of type ''X -> Y''. For example, the function application operator '':'' is a function of type ''(X -> Y) -> (X -> Y)''. If h is a function of type ''A -> B'' then ::h is a function ''A -> B'', and if x is of type A then ::h:x (or h:x) is of type B. All built-in arithmetic functions are of type ''num -> (num -> num)''. Hence, +:1, the increment

function, is of type ''num -> num''. All relational functions, except ''=='', are of type ''simple -> (simple -> bool)''; they all take arguments of simple type and return results of bool type.

As an example of using constructor patterns, the function ''mid'' which selects the middle field of a triple can be defined as:

```
let mid:t = let (u,v,w)=t in v.
```

and it has type ''(X,Y,Z) -> Y''.

The value of a lambda expression, \x.E, is a function of type ''X -> Y'' if x is of type X and E is of type Y. Therefore, if v is of type X, then the expression (\x.E):v is of type Y. For example, the factorial function defined by

```
let rec n! = n==0 -> 1; n*(n-1)!.
```

is equivalent to

```
let rec ! = \n.n==0 -> 1; n*!:(n-1).
```

and it has type ''num -> num''.

A Sample Session

After the fpl system is invoked, the following banner is printed:

```
TYPE FPL (version 2)
Type ''help.'' for help.
[>
```

''[>'' is the system's top level prompt; it indicates that FPL is now ready to accept inputs. The prompt ''<]'' means that FPL expects more inputs for the current expression being entered until a period (.) is encountered.

If you type

```
[> help.
```

then you should see a short help menu:

```
builtins.
```

```
del:"<name>".
exit.
fns.
load:"<filename>".
macros.
op:"<symbol>".
ops.
vic:"<filename>".

val it = help : sym
```

The symbol ``it'' always denotes the most recent expression entered. It can be used to construct more complicated expressions from previously entered expressions.

To find out the list of all built-in functions, just type

```
[> builtins.

val it = [*,+,-,/,<,==,>,^,acos,asin,atan,cos,del,echo,error,exp,fst,hd,
head,ln,load,log,mod,op,print,random,rem,save,sin,snd,sqrt,tail,tan,tl,
trunc,vi] : list(sym)
```

The symbol ``builtins'' is just a predefined constant---a list of symbols. The expression ``length:builtins'' computes the total number of built-in functions.

Similarly, you can find out the lists of macros, system predefined and user defined functions, and operators.

```
[> macros.

val it = [it] : list(sym)

[> fns.

val it = [!,..,:,<=,<>,>=,@,Nth,Nths,abs,and,append,ceiling,cons,deg,
firstNs,floor,frac_part,ints,iota,length,list_to_stream,map,maps,max,
member,min,mkstream,neg,not,null?,or,pi,quotient,rad,reverse,round,
signum,stream_to_list,vic] : list(sym)

[> ops.

val it = [!,',*,+,-,->,..,/,:,;,<,<=,<>,=,==,>,>=,@,^,and,else,mod,not,
or,otherwise,rem,vi,vic,when] : list(sym)
```

Given a predefined operator ``and'', you can find out its declaration by

```
[> ops:"and".

val it = [OP(infix,left,850)] : list(op)
```

It says that ``and'' is a left-associative infix operator with precedence 850.

Let us create the Fibonacci function in a file ``myfile'' and then load it into the system. To do so, you type

```
[> vic:"myfile".
```

The Unix ``vi'' editor will be invoked with the argument ``myfile''. After you exit the editor with the following definition in the file,

```
let rec fib:n =
        1                           when n <= 2;
        fib:(n-1) + fib:(n-2)       otherwise.
```

the system immediately loads and compiles the file, and then prints

```
Loading file: "myfile"

val fib = fn : num -> num


"myfile" is loaded.

val it = true : bool
```

which says that ``fib'' is a function of type ``num -> num'' and the loading of ``myfile'' is successful.

To compute the fifth Fibonacci number, enter the expression

```
[> fib:5.

val it = 5 : num
```

The function ``del'' deletes a function definition permanently from the system. For example, if you enter

```
[> del:"fib".

val it = true : bool
```

the function ``fib'' is removed permanently. Hence, you'll get a type error message if you enter

```
[> fib:1.

Type error at fib

Unrecognisable input fib : 1
```

To load the file ``myfile'' again, you can use the ``load'' function.

```
[> load:"myfile".
Loading file: "myfile"

val fib = fn : num -> num


"myfile" is loaded.
```

```
    val it = true : bool

    [> fib:5.

    val it = 5 : num
```

Since ''vic'', ''vi'', ''load'' and ''del'' are just functions (not commands), you can use them as arugments to other functions. For example, to load a list of files, you can enter

```
    [> map:load:["file1","file2",...].

    val it = [true,true,...] : list(bool)
```

At any time if you want to find the type of a function, just enter its name. For example, to find the type of the factorial function ''!'', enter

```
    [> !.

    val it = fn : num -> num

    [> 3!.

    val it = 6 : num

    [> it!.

    val it = 720 : num
```

To exit the sytem, you can type

```
    [> exit.
```

''quit'', ''halt'' or ''bye''.

Predefined functions

The following is a list of all predefined functions with brief descriptions of their usage. The notation ''E —> V'' means that the expression ''E'' evaluates to the value ''V''.

```
%
% The following operator declarations define the basic syntax of FPL.
%
% (Note: Always define the longest operator first.)
%
postfix non   950  otherwise.
infix   non   950  when.
prefix  non   950  else.
prefix  non   100  load.
infix   left  850  and.
```



```
prefix  assoc 800  not.
infix   non   400  mod.
infix   non   400  rem.
prefix  non   100  vic.
prefix  non   100  vi.
infix   left  900  or.
infix   non   700  ">=".
infix   non   700  "<=".
infix   non   700  "==".
infix   non   950  "->".
infix   non   700  "<>".
infix   non   1000 "=".
infix   right 990  ";".
infix   non   700  "<".
infix   non   700  ">".
infix   non   600  "..".
infix   left  500  "+".
infix   left  500  "-".
infix   left  400  "*".
infix   left  400  "/".
infix   right 200  "^".
infix   right 100  "@".
prefix  non   50   "-".
postfix assoc 50   "!".


%
% f:x —> f:x,  the function application function
%
let ::f:x = f:x.


%
% f@x —> f:x,  the function application function
%
let @:f:x = f:x.


%
% true or false —> true, the disjunction function
%
let x or y = x -> true; y.


%
% true and true —> true, the conjunction function
%
let x and y = x -> y; false.


%
% not true —> false,   the negation function
%
```

```
let not x = x -> false; true.


%
% 1 <> 2 --> true,      the inequality function
%
let x <> y = not (x==y).


%
% 2 >= 1 --> true,      the greater than and equal to function
%
let x >= y = not (x<y).


%
% 2 <= 1 --> false,     the less than and equal to function
%
let x <= y = not (x>y).


%
% neg:9 --> -9,         the negative function
%
let neg:x = -1*x.


%
% signum:-5 --> -1,     the sign function
%
let signum:x =  -1      when x<0;
                0       when x==0;
                1       otherwise.


%
% cons:2:[] --> [2],    the list construction function
%
let cons:x:y = [x|y].


%
% abs:-9 --> 9,         the absolute-value function
%
let abs:x = x           when x>=0;
            -1*x        otherwise.
```



```
%
% round:9.4 --> 9,      round to the nearest integer
%
let round:x =   x               when integer?:x;
                trunc:(x+0.5)   when x>=0;
                trunc:(x-0.5)   otherwise.


%
% quotient:17:3 --> 5,  the integer division function
%
let quotient:x:y =
        integer?:x and integer?:y -> trunc:(x/y);
                error:"non integer argument!".


%
% frac_part:4.3 --> 0.3
%
let frac_part:x = integer?:x -> 0; x-trunc:x.


%
% floor:-9.2 --> -10,   the smallest integer not greater its argument
%
let floor:x =
       let tx = trunc:x
       in x     when frac_part:x==0;
          tx    when x>=0;
          tx-1  otherwise.


%
% ceiling:-9.2 --> -9,  the smallest integer greater its argument
%
let ceiling:x =
       let tx = trunc:x
       in x     when frac_part:x==0;
          tx+1  when x>=0;
          tx    otherwise.


%
% max:"a":"b" --> b,    the maximum function
%
let max:x:y = x>=y -> x ; y.


%
% min:"a":"b" --> a,    the minimum function
```

```
%
let min:x:y = x>y -> y ; x.


%
% null?:[] --> true,    the empty list test function
%
let null? = (==:[]).


%
% List concatenation function
%
% append:[1,2,3]:[4,5,6] --> [1,2,3,4,5,6]
%
let rec append:x:y =
        y                                  when x==[];
        (let [h|t]=x in [h|append:t:y]) otherwise.


%
% List reversal function
%
% reverse:[1,2,3,4,5] --> [5,4,3,2,1]
%
let reverse:x =
        let rec

            rev:l:a =   a                                  when l==[];
                        (let [h|t]=l in rev:t:[h|a])       otherwise

        in rev:x:[].


%
% The length function of lists
%
% length:[1,2,3,4,5] --> 5
%
let length:l =
        let rec

            len:x:a =   a                    when x==[];
                        len:(tl:x):(a+1)     otherwise

        in len:l:0.


%
% The membership test function
%
```



```
% member:3:[1,2,4,5] --> false
%
let rec member:x:l =
                false           when l==[];
                true            when x==hd:l;
                member:x:(tl:l) otherwise.


%
% map:f:[x1,...,xn] returns [f:x1,f:x2,...,f:xn].
%
% map:(+:1):[1,2,3,4] --> [2,3,4,5]
%
let rec map:f:l =
        []                                  when l==[];
        [ f@hd@l | map:f:(tl:l) ]           otherwise.


%
% Nths:n:s returns the nth element of the stream s.
%
% Nths:4:<<1,2,3,4,5,6>> --> 4
%
let rec Nths:n:stream =
                error:"non integer argument"      when integer?:n;
                head:stream                       when n==1;
                Nths:(n-1):(tail:stream)          when n>1;
                error:"negative range!"           otherwise.


%
% ints:n returns an infinite stream of integers starting from n.
%
% ints:1 --> <<1,2,3,4,5,...>>
%
let rec ints:n = <<n | ints:(n+1) >>.


%
% list_to_stream:l returns a stream with elements as in the list l.
%
% list_to_stream:[1,2,3] --> <<1,2,3>>
%
let rec list_to_stream:list =
        <<>>                                              when list==[];
        (let [h|t]=list in << h | list_to_stream:t >>)     otherwise.


%
% stream_to_list:s returns a list of elements as in the stream s; s must
%       be a finite stream.
```

```
%
% stream_to_list:<<1,2,3>> --> [1,2,3]
%
let rec stream_to_list:stream =
        []                                          when stream==<<>>;
        [head:stream | stream_to_list:(tail:stream)]    otherwise.



%
% This is the map function for the data type stream.
%
% maps:(+:1):<<1,2,3>> --> <<2,3,4>>
%
let rec maps:f:stream =
        <<>>                                        when stream==<<>>;
        << f:(head:stream) | maps:f:(tail:stream) >>    otherwise.



%
% Nth:n:l returns the nth element of the list l.
%
% Nth:4:[1,2,3,4,5,6] --> 4
%
let Nth:n:list =
        error:"non positive input!"     when n<= 0;
        error:"non integer argument!"   when float?:n;
        (let rec Nth1:m:list =
                        hd:list                 when m==1;
                        Nth1:(m-1):(tl:list)    otherwise
         in Nth1:n:list)                        otherwise.



%
% firstNs:n:s returns a list of the first n elements of the stream s.
%
% firstNs:4:(ints:1) --> [1,2,3,4]
%
let firstNs:n:stream =
        error:"negative argument!"              when n<0;
        error:"stream out of bound!"            when stream==<<>>;
        (let rec firstNs1:m:s =
                []                                          when m==0;
                [head:s | firstNs1:(m-1):(tail:s)]      otherwise
         in firstNs1:n:stream)                  otherwise.



%
% m..n returns a list [m,m+1,...,n].
%
% 1..5 --> [1,2,3,4,5]
%
let n..m =
```



```
        error:"non integer argument"    when float?:n or float?:m;
        error:"inputs out of bounds"    when m<n;
        (let rec f:n:m = [n]                when n==m;
                         [n|f:(n+1):m]      otherwise
         in f:n:m)                          otherwise.


%
% iota:n returns a list [1,2,...,n].
%
% iota:5 --> [1,2,3,4,5]
%
let iota = ..:1.



%
% mkstream:n returns a finite stream <<1,2,...,n>>.
%
% mkstream:5 --> <<1,2,3,4,5>>
%
let mkstream:n = list_to_stream@iota@n.



%
% The factorial function.
%
% 6! --> 720
%
let n! = error:"negative input!"        when n<0;
         error:"non integer argument"   when float?:n;
         (let rec

             fac:a:m =  a                   when m==0;
                        fac:(a*m):(m-1)     otherwise

         in fac:1:n)                        otherwise.



%
% pi --> 3.1415927
%
let pi = atan:1 * 4.



%
% rad:x returns the radian value of x degree.
%
% rad:180 --> 3.1415927
%
let rad:x = pi*x/180.
```

```
%
% deg:r returns the degree value of r radian.
%
% deg:1 --> 57.29578
%
let deg:x = 180*x/pi.


%
% vic:"file" --> ''edit and load file''
%
let vic:file = load:(vi:file).
```

Built-in Functions

random:()--> n, where 0 < n < 1.

hd:[h|t] --> h.

tl:[h|t] --> t.

head:<<h|t>> --> h.

tail:<<h|t>> --> t.

fst:(f,s) --> f.

snd:(f,s) --> s.

print:<expression> --> true, and prints the value of <expression>.

echo:<expression> --> <value>, and prints the <value> of <expression>.

trunc:9.2 --> 9.

sin:x --> the sine of x.

asin:x --> the arc sine of x.

cos:x --> the cosine of x.

acos:x --> the arc cosine of x.

tan:x --> the tangent of x.

atan:x --> the arc tangent of x.

ln:x --> the natural logarithm of x to the base e.

exp:x --> e to the power x.

log:x --> the logarithm of x to the base 10.

sqrt:x --> the square root of x.

save:"<file>" --> true, and saves the entire database into <file>.

error:"message ..." --> "message ...", and aborts the execution.

integer?:x --> true if x is an integer;
integer?:x --> false otherwise.

float?:x --> true if x is a floating point number;
float?:x --> false otherwise.

+:x:y --> x+y.

-:x:y --> x-y.

*:x:y --> x*y.

/:x:y --> x/y.

mod:x:y --> x mod y.

rem:x:y --> x rem y.

^:x:y --> x to the power y.

==:x:y --> true if x is equal to y, where x and y can be of any type;
==:x:y --> false, otherwsie.

<:x:y --> true if x<y;
<:x:y --> false, otherwise.

>:x:y --> true if x>y;
>:x:y --> false, otherwise.

```
%
%  Module:      main
%
% Written by: Mantis H.M.Cheng (July/04/88)
%

% The infinite execution loop.
%
fpl :- main.
fpl :- fpl.

%
% The main driver
%
main :-
        next_expression( S ),
        process( S ),
        !,
        fail.           % recover all of the stack space by failure


%
% A top level command interpreter. (note: it is deterministic.)
%
process( abort ) :- !, abort.
process( exit ) :- nl, halt.
process( halt ) :- nl, halt.
process( quit ) :- nl, halt.
process( bye ) :- nl, halt.
%
% operator declaration
%
process( op(O,T,A,P) ) :-
        !, nl,
        ((add_op(O,T,A,P),
          print_answer_with_type( O, type(op,[]), term('OP',3,[T,A,P]) ))
        ; print_answer_with_type( O, error, error(op(O,T,A)) )).
/*
 * The following features are disabled because of a bug in ALS-Prolog.
 *
process( type(F/_,P1,_) ) :-
        !, nl,
        print_answer_with_type( type, type(F,P1), F ).
 */
%
% Recursive function definitions
%
process( [let,rec,['=',F,B]] ) :-
        !,
        ((atom( F ), not primitive(F),
          expand_macro( B, B1 ),
          compile_define( F, [let,rec,['=',F,B1],F] )
         );
         nl,print_answer_with_type( F, error, error(def(F)) )
        ).
%
```



```
% Non-recursive function definition
%
process( [let,['=',F,B]] ) :-
        !,
        ((atom( F ), not primitive(F),
          expand_macro( B, B1 ), compile_define( F, B1 )
         );
         nl,print_answer_with_type( F, error, error(def(F)) )
        ).
%
% Macro definitions
%
process( [macro|F] ) :- !,
        compile_macro( [macro|F] ).
%
% function applications
%
process( C ) :-
        apply_function( C ), !.
%
% invalid command
%
process( C ) :-
        put( "^G ),
        write( 'Unrecognisable input ' ),
        print_expression( C ), nl, nl.

%
% auxiliary predicates
%

%
% compile_define( F, D, B ) :-
%       this predicate compiles a new definition B (after transformation)
%       If the compilation is successful, then the new definition is added
%       with name F, original source expression D and its compiled code.
%
compile_define( F, B ) :-
        nl,
                % compute the type of expression
        type_of( B, T ),
        add_type_def( F, T ),
        compile( B, C, [join], [] ),
        add_define( F, C ),
                % compute the value of new definition
        compile( F, C1, [stop], [] ), exec( C1, X ),
        print_answer_with_type( F, T, X ).

%
% apply_function( C ) :-
%       this predicate compiles the application C, executes it and prints
%       the result.
%
apply_function( C ) :-
        expand_macro( C, C1 ),
        type_of( C1, T ),
```

```
        compile( C1, C2, [stop], [] ),
                % compute the type of expression
        exec( C2, X ),
        nl,
                %
                % save the most recent evaluated expression as macro
                %
        add_macro( it, it, C1 ),
        print_answer_with_type( it, T, X ).
```



```
%
% Module:       compile
%
% Written by: Mantis H.M. Cheng (July/04/88)
%

%
% compile( E, C1, C2, N ) :- compile the expression E given the
%       environment N and the tail code segment C2. The code
%       generated is prepended to C2 and returned in C1.
%       (note: it is deterministic.)
%
compile( [], L, L, _ ) :- !.
compile( num(I), [ldc,I|L], L, _ ) :- !.
compile( const(S), [ldc,S|L], L, _ ) :- !.
%
% primitive constructor terms
%
compile( term('NUM',1,[num(I)]), [ldc,I|L], L, _ ) :- !.
compile( term('BOOL',1,[B]), [ldc,B|L], L, _ ) :- primitive_const(B), !.
compile( term('SYM',1,[const(S)]), [ldc,S|L], L, _ ) :- !.
%
% singletons (N.B. check local bindings before primitive constants)
%
compile( X, [ld,I|L], L, N ) :-
        atom(X),
        location( X, N, I ), !.
%
% built-in primitive constants
%
compile( $nil, [ldc,[]|L], L, N ).
compile( $void, [ldc,'()'|L], L, N ).
compile( builtins, [ldc,X|L], L, N ) :- !,
        (setof( F, primitive_func(F), X ); X=[]).
compile( fns, [ldc,X|L], L, N ) :- !,
        (setof( F, D^code(F,D), X ); X=[]).
compile( macros, [ldc,X|L], L, N ) :- !,
        (setof( F, (D,B)^macro(F,D,B), X );X=[]).
compile( ops, [ldc,X|L], L, N ) :-
        !, (setof( O, P^T^op_decl(P,T,O), X ); X=[]).
compile( help, [ldc,help|L], L, N ) :- !,
        nl,
        write( 'builtins.' ), nl,
        write( 'del:"<name>".' ), nl,
        write( 'exit.' ), nl,
        write( 'fns.' ), nl,
        write( 'load:"<filename>".' ), nl,
        write( 'macros.' ), nl,
        write( 'op:"<symbol>".' ), nl,
        write( 'ops.' ), nl,
        write( 'vic:"<filename>".' ), nl.
compile( C, [ldc,C|L], L, N ) :-
        primitive_const( C ), !.
%
% external variable
%
```

```
compile( X, [ldx,X|L], L, N ) :-
        atom(X),
        code(X,_), !.
%
% primitive functions
%
compile( F, [ldf,[F,rtn]|L], L, _ ) :-
        atom(F),
        primitive_func(F), !.
%
% constructor terms
%
compile( term(S,1,[I]), L1, L2, N ) :-
        simple_type_constructor(S), !, compile( I, L1, L2, N ).
compile( A, [ldc,A|L], L, _ ) :- atom(A), constructor_symbol(A), !.
compile( term('.',2,[E,Es]),L1,L2,N) :- !,
        compile(E,L3,[acon|L2],N),
        compile(Es,L1,L3,N).
compile( term('CONS',2,[H,T]), L1, L2, N ) :- !,
        compilis( [H,[delay,T]], L1, [bld,'CONS',2|L2], N ).
compile( term(F,A,Args), L1, L2, N ) :- !,
        compilis( Args, L1, [bld,F,A|L2], N ).
%
% selector for constructor terms
%
compile( [head,T], L1, L2, N ) :- !,
        compile( T, L1, [arg,1|L2], N ).
compile( [tail,T], L1, L2, N ) :- !,
        compile( T, L1, [arg,2,ap0|L2], N ).
compile( [$arg,2,'CONS'/2,T], L1, L2, N ) :- !,
        compile( T, L1, [arg,2,ap0|L2], N ).
compile( [$arg,A,_,T], L1, L2, N ) :- !,
        compile( T, L1, [arg,A|L2], N ).
%
% delayed evaluation
%
compile( [force,E], L1, L2, N ) :- !,
        compile( E, L1, [ap0|L2], N ).
compile( [delay,E], [lde,L1|L2], L2, N ) :- !,
        compile( E, L1, [upd], N ).
%
% conditionals
%
compile( [';',C1,Cs], L1, L2, N ) :- !, % (COND <CL1>...<CLn>)
        compile_cond( [';',C1,Cs], L1, L2, N ).
%
% lambda definitions
%
compile( ['\',E1,E2], [ldf,L2|L1], L1, N ) :- !,
        compile( E2, L2, [rtn], [E1|N] ).       % (LAMBDA (X1 ... Xk) E)
%
% local definitions
%
compile( [let,rec|D], [dum|L1], L2, N ) :- !,% (LETREC (X1 E1) ... (Xk Ek) E)
        def_exprs( D, Es, E ),
        exprs( Es, Args ),
```



```
        vars( Es, Ns ),
        compile( E, L3, [rtn], [Ns|N] ),
        compilis( Args, L1, [ldf,L3,rap|L2], [Ns|N] ).
compile( [let|D], L1, L2, N ) :- !,             % (LET (X1 E1) ... (Xk Ek) E)
        def_exprs( D, Es, E ),
        exprs( Es, Args ),
        vars( Es, Ns ),
        compile( E, L3, [rtn], [Ns|N] ),
        compilis( Args, L1, [ldf,L3,ap|L2], N ).
%
% curried the binary function which is defined by an operator
%
compile( [F,A1,A2], L1, L2, N ) :-
        atom(F), not(primitive2(F)), binary_op(F), !,
        compile([[F,A1],A2],L1,L2,N).
%
% function applications
%
compile( [F|As], L1, L2, N ) :-
        compile( F, L3, [ap|L2], N ),
        compilis( As, L1, L3, N ).


%
%
% auxiliary predicates
%


%
% compile_cond( E, L1, L2, N ) :- given a list of clauses in E, we compile
%       it into a list of nested if-then-else expressions.
%
compile_cond( [';',P,Y], L1, L4, N ) :- !,
        conditional( P, C, E ),
        compile_cond( Y, L3, [join], N ),
        compile( E, L2, [join], N ),
        compile( C, L1, [sel,L2,L3|L4], N ).
compile_cond( P, L1, L3, N ) :-
        conditional( P, C, E ), !,
        compile( E, L2, [join], N ),
        compile( C, L1, [sel,L2,[fault,C,stop]|L3], N ).
compile_cond( P, L1, L2, N ) :-
        last_conditional( P, E ), !,
        compile( E, L1, L2, N ).


conditional( [when,E,C], C, E ) :- C\==true, !.
conditional( ['->',C,E], C, E ) :- C\==true.


last_conditional( [otherwise,E], E ) :- !.
last_conditional( [else,E], E ) :- !.
last_conditional( ['->',true,E], E ) :- !.
last_conditional( E, E ).
```

```
%
% def_exprs( L, D, E ) :-
%       L is the body of a let-expression. D is the list of
%       all definitions and E is the expression in the let-expression.
%
def_exprs( [E], [], E ) :- !.
def_exprs( [D|Ds], [D|Ds1], E ) :-
        def_exprs( Ds, Ds1, E ).


%
% vars( B, V ) :- takes a list of bindings B of the form
%       V1=E1,...,Vn=En and returns a list
%       of all the variable names in V=[V1,...,Vn].
%
vars( [], [] ) :- !.
vars( [['=',term('.',2,[H,T]),F]|Ds], Vs ) :- !,
        vars( [['=',H,[hd,F]],['=',T,[tl,F]]|Ds], Vs ).
vars( [['=',term(S,1,[H]),F]|Ds], Vs ) :-
        simple_type_constructor(S), !,
        vars( [['=',H,F]|Ds], Vs ).
vars( [['=',term(_,_,Args),F]|Ds], Vs1 ) :- !,
        vars( Ds, Vs ),
        arg_vars( Args, ArgVs ),
        append(ArgVs,Vs,Vs1).
vars( [['=',$nil,_]|Ds], Vs ) :-                % Vi = F
        !, vars( Ds, Vs ).
vars( [['=',D,_]|Ds], [D|Vs] ) :-               % Vi = F
        atom(D),
        vars( Ds, Vs ).


arg_vars( [], [] ) :- !.
arg_vars( [Arg|Args], [Arg|ArgVs] ) :-
        atom(Arg), arg_vars(Args,ArgVs).
arg_vars( [term(_,_,Args1)|Args], ArgVs2 ) :-
        arg_vars(Args1,ArgVs1),
        arg_vars(Args,ArgVs),
        append(ArgVs1,ArgVs,ArgVs2).


%
% exprs( B, E ) :- takes a list of bindings B of the form
%       V1=E1,...,Vn=En and returns a list
%       of all the expression values in E=[E1,...,En].
%
exprs( [], [] ) :- !.
exprs( [['=',term('.',2,[H,T]),F]|Ds], Es ) :- !,
        exprs([['=',H,[hd,F]],['=',T,[tl,F]]|Ds], Es ).
exprs( [['=',term(S,1,[H]),F]|Ds], Es ) :-
        simple_type_constructor(S), !,
        exprs([['=',H,term(S,1,[F])]|Ds], Es ).
exprs( [['=',term(F,A,Args),V]|Ds], Es1 ) :- !,
        exprs( Ds, Es ),
        construct_arg_exp(Args,1,F/A,V,E),
        append(E,Es,Es1).
```



```
exprs( [['=',$nil,E]|Ds], Es ) :-
        !, exprs( Ds, Es ).
exprs( [['=',_,E]|Ds], [E|Es] ) :-
        exprs( Ds, Es ).


construct_arg_exp([],_,_,_,[]) :- !.
construct_arg_exp([Arg|Args],N,F/A,V,[[$arg,N,F/A,V]|Es] ) :-
        atom(Arg), !,
        N1 is N+1,
        construct_arg_exp(Args,N1,F/A,V,Es).
construct_arg_exp([term(F1,A1,Args1)|Args],N,F/A,V,Es ) :-
        N1 is N+1,
        construct_arg_exp(Args1,1,F1/A1,[$arg,N,F/A,V],Es1),
        construct_arg_exp(Args,N1,F/A,V,Es2),
        append(Es1,Es2,Es).


%
% compilis( Es, L1, L2, N ) :-
%       this predicate compiles a list of expressions, argument list,
%       with given environment N and initial code segment L2, and
%       returns the code in L1.
%
compilis( [], [ldc,[]|L], L, _ ) :- !.
compilis( [E|Es], L1, L2, N ) :-
        compile( E, L3, [acon|L2], N ),
        compilis( Es, L1, L3, N ).


%
% location( X, N, P ) :-
%       P is a location pair (E,O) of X in the namelist N,
%       where E is the env. number and O is the offset within the env. E.
% e.g.  N has the form = [ [a,b], [x,y] ].
%       location( b, N, [0|1] ).
%       location( x, N, [1|0] ).
%
location( X, [E1|Ns], [0|O] ) :-
        member( X, E1 ), !,
        position( X, E1, O ).
location( X, [_|Ns], [E|O] ) :-
        location( X, Ns, [E1|O] ),
        E is 1+E1.


%
% position( X, L ) :- the position of X in L starting from 0.
%
position( X, [X|_], 0 ) :- !.
position( X, [_|E], N ) :-
        position( X, E, N1 ),
        N is 1+N1.


%
```

```
% Module:       compile
%
% macro expansions
%

%
% compile_macro( M ) :-
%       this predicate compiles the macro definition M. If the compilation
%       is successful, it is added into the macro database. otherwise it
%       prints an error message.
%       (note: it always succeeds and is deterministic.)
%
compile_macro( [macro,[F|P],B] ) :-
        nl,
        expand_macro( B, B1 ),
        generate_free_var( P, P1, Vs ),
        replace_var( B1, B2, Vs ),
        add_macro( F, [F|P1], B2 ),
        !,
        print_answer_with_type( F, macro, macro ).
compile_macro( [macro,F,B] ) :-
        nl,
        atom( F ),
        expand_macro( B, B1 ),
        add_macro( F, F, B1 ),
        !,
        print_answer_with_type( F, macro, macro ).
compile_macro( [macro,F,B] ) :-
        nl,
        write( 'Illegal macro definition in ' ),
        print_expression( B ), nl, nl.


%
% generate_free_var( P, X, V ) :-
%       Given a list of bound variables P, this predicate generates a
%       list of unique logical variables for each bound variables in X
%       and returns a list of bindings in V as pairs of (Name,Variable).
%
generate_free_var( [], [], [] ) :- !.
generate_free_var( [P|Ps], [X|Xs], [[P|X]|Vs] ) :- !,
        generate_free_var( Ps, Xs, Vs ).
generate_free_var( P, X, [[P|X]] ).


%
% replace_var( P, T, V ) :-
%       this predicate replaces every bound variables in P by the
%       corresponding logical variable in V and returns the resulting
%       expression in T after replacement.
%       (note: it is deterministic.)
%
replace_var( [], [], _ ) :- !.
replace_var( B, X, Vs ) :-
        atom( B ),
        select_var( B, Vs, X ), !.
```



```
replace_var( [B|Bs], [X|Xs], Vs ) :- !,
        replace_var( B, X, Vs ),
        replace_var( Bs, Xs, Vs ).
replace_var( term(F,A,P), term(F,A,X), Vs ) :- !,
        replace_var( P, X, Vs ).
%
% For type declaration
%
replace_var( s(T), s(T), _ ) :- !.
replace_var( type(F,P), type(F,P1), Vs ) :- !,
        replace_var( P, P1, Vs ).
replace_var( B, B, _ ).


%
% add_macro( N, H, B ) :-
%       this predicate adds a new macro into the macro database.
%
add_macro( N, H, B ) :-
        delete_macro( H ),
        assert( macro(N,H,B) ).


%
% delete_macro( M ) :-
%       this predicate deletes an old macro from the macro database.
%
delete_macro( F ) :-
        retract( macro(F,_,_) ), !.
delete_macro( _ ).


%
% select_var( X, Vs, V ) :-
%       this predicate returns the binding of the bound variable X,
%       a logical variable, in V given the binding list Vs.
%
select_var( X, [[X|V]|_], V ) :- !.
select_var( X, [_|Vs], V ) :-
        select_var( X, Vs, V ).


%
% expand_macro( P, S ) :-
%       this predicate tries all possibilities of macro expansion on
%       the source expression P and returns the result in S.
%       (note: it is deterministic.)
%
expand_macro( [], [] ) :- !.
expand_macro( cl(F,E), cl(F,E) ) :- !.
expand_macro( term(F,A,X), term(F,A,X1) ) :-
        !, try_expand_macro( X, X1 ).
expand_macro( num(X), num(X) ) :- !.
expand_macro( const(X), const(X) ) :- !.
expand_macro( X, X1 ) :-
        atom( X ), !,
```

```
        try_expand_macro( X, X1 ).
expand_macro( [X|Y], Z ) :-
        expand_macro( Y, Y1 ),
        expand_macro( X, X1 ),
        try_expand_macro( [X1|Y1], Z ).


%
% try_expand_macro( X, Y ) :-
%       this predicate retrieves a macro definition that matches X and
%       returns the replacement in Y. If there is no suitable macro, Y
%       is the same of the source expression.
%
try_expand_macro( X, X1 ) :-
        macro( _, X, X1 ), !.
try_expand_macro( X, X ).
```



```
%
% Module:       machine
%
% Written by: Mantis H.M. Cheng (July/04/88)
%

%
% The SECD machine emulator or the evaluator.
%
% exec( F, R ) :- it executes the function F on the
%       SECD machine and returns the result in R.
%
exec( [I|B], X ) :-
        cycle( I, s([],[],[_|B],[]), s([X],_,_,_) ).


%
% cycle( I, S, S1 ) :-
%       executes the current instruction I on the machine state S
%       and returns the final state S1.
%       (note: it is deterministic.)
%
cycle( stop, S, S ) :- !.
cycle( _, s([V|_],_,_,_), s([error(X)],_,_,_) ) :-
        nonvar(V), V=error(X), !.
cycle( I, State, State2 ) :-
        apply( I, State, s(S1,E1,[I1|C1],D1) ), !,
        cycle( I1, s(S1,E1,[I1|C1],D1), State2 ).


%
% apply( I, S1, S2 )
%       - apply a single instruction from the SECD machine state
%         S1 to S2 with the current instruction I.
%       where S1, S2 are of form s(S,E,C,D).
%       (note: it is deterministic.)
%
        %
        % instructions for building and accessing constructor terms
        %
apply( bld, s( [Args|S], E, [_,N,A|C], D ),
            s( [term(N,A,Args)|S], E, C, D ) ) :- !.
apply( arg, s( [term(_,_,Args)|S], E, [_,N|C], D ),
            s( [Arg|S], E, C, D ) ) :- I, choose( N, Args, Arg ).
apply( arg, s( [X|S], E, [_,N|C], D ), s( [error(arg(X))|S], E, C, D ) ) :- !.
        %
        % local references
        %
apply( ld, s( S, E, [_,I|C], D ), s( [X|S], E, C, D ) ) :- !,
        locate( I, E, X ).
        %
        % loading a constant
        %
apply( ldc, s( S, E, [_,I|C], D ), s( [I|S], E, C, D ) ) :- !.
        %
        % external references
```

```
        % (Note: save the continuation on the dump, since exteral code
        %        always contains a ''join'' instruction at the end.)
        %
apply( ldx, s( S, E, [_,X|C], D ), s( S, E, B, [C|D] ) ) :-
        code( X, B ), !.
        %
        % conditional expression
        %
apply( sel, s( [X|S], E, [_,Ct,Cf|C], D ), s( S, E, Cx, [C|D] ) ) :- !,
        select( X, Ct, Cf, Cx ).
apply( join, s( S, E, _, [C|D] ), s( S, E, C, D ) ) :- !.
        %
        % loading a function
        %
apply( ldf, s( S, E, [_,Cl|C], D ), s( [cl(Cl,E)|S], E, C, D ) ) :- !.
        %
        % function application
        %
apply( ap, s( [cl(Cl,E1),V|S], E, [_|C], D ),
           s( [], [V|E1], Cl, [S,E,C|D] ) ) :- !.
        %
        % return from function application
        %
apply( rtn, s( [X], _, _, [S,E,C|D] ), s( [X|S], E, C, D ) ) :- !.
        %
        % for recursive definitions
        %
apply( dum, s( S, E, [_|C], D ), s( S, [X|E], C, D ) ) :- !.
apply( rap, s( [cl(Cl,[V|E]),V|S], [V|E], [_|C], D ),
       s( [], [V|E], Cl, [S,E,C|D] ) ) :- !.
        %
        % The following instructions are for ''delayed evaluation''
        %
        % r( Flag, Value, Closure ) = recipe, where Flag = {t,X} to indicate
        %       that the recipe has (or has not) been evaluated. If Flag is
        %       free (X), then we evaulates the closure, otherwise its value
        %       is in Value.
        %
apply( lde, s( S, E, [_,C|Cl], D ), s( [r(X,Y,[C|E])|S], E, Cl, D ) ) :- !.
apply( ap0, s( [r(X,Y,[C|E])|S], E1, [_|Cl], D ),
       s( [], E, C, [[r(X,Y,[C|E])|S],E1,Cl|D] ) ) :- var(X), !.
apply( ap0, s( [r(X,V,_)|S], E, [_|C], D ), s( [V|S], E, C, D ) ) :-
        atom(X), X=t, !.
apply( upd, s( [X], E, [_], [[r(t,X,_)|S],E1,Cl|D] ),
       s( [X|S], E1, Cl, D ) ) :- !.
        %
        % exceptions
        %
apply( fault, s( _, _, [_,I|_], _ ), s( [error(fault(I))], _, _, _ ) ) :- !.
        %
        % arguments construction
        %
apply( acon, s( [A,B|S], E, [_|C], D), s( [[A|B]|S], E, C, D ) ) :- !.
        %
        % system primitives
        %
```



```
apply( F, s( S, E, [_|C], D ), s( [A|S], E, C, D ) ) :-
        primitive0( F ), !, apply0( F, A ).
apply( F1, s( S, [[A]|E], [_|C], D ), s( [R|S], E, C, D ) ) :-
        primitive1( F1 ), !, apply1( F1, A, R ).
        %
        % curried version of binary primitives
        %
apply( [F2,A], s( S, [[B]|E], [_|C], D ), s( [R|S], E, C, D ) ) :-
        primitive2( F2 ), !, apply2( F2, [A,B], R ).
apply( F2, s( S, [[A]|E], [_|C], D ), s( [cl([[F2,A],rtn],E)|S], E, C, D ) ) :-
        primitive2( F2 ), !.
        %
        % normal application of binary primitives
        %
apply( F2, s( S, [[A,B]|E], [_|C], D ), s( [R|S], E, C, D ) ) :-
        primitive2( F2 ), !, apply2( F2, [A,B], R ).
        %
        % error conditions
        %
apply( ldx, s(_,_,[_,C|_],_), s([error(unbound(C))],_,_,_) ) :-
        atom( C ), !.
apply( F, s(_,[A|_],_,_), s([error(exception(F,A))],_,_,_) ) :-
        atom( F ).


%
% Module:       machine
%
% auxiliary predicates or the runtime system
%

%
% nullary built-in functions.
%
apply0( random, X ) :- X is random.


%
% unary built-in functions
%
apply1( error, X, error(X) ) :- !.
apply1( 'integer?', X, true ) :- integer(X), !.
apply1( 'integer?', X, false ) :- !.
apply1( 'float?', X, true ) :- float(X), !.
apply1( 'float?', X, false) :- !.
apply1( hd,    [X|_], X ) :- !.
apply1( hd,    [],    error(arg([])) ) :- !.
apply1( tl,    [_|X], X ) :- !.
apply1( tl,    [],    error(arg([])) ) :- !.
apply1( fst,   term('TUPLE',2,[X,_]), X ) :- !.
apply1( snd,   term('TUPLE',2,[_,X]), X ) :- !.
apply1( print, X, true ) :- !, print_answer( X ), nl.
apply1( echo,  X, X ) :- !, print_answer( X ), nl.
apply1( trunc, X, Y ) :- X >= 0, !, Y is round(X-0.5).
apply1( trunc, X, Y ) :- X < 0, !, Y is round(X+0.5).
apply1( sin,   X, Y ) :- !, Y is sin(X).
```

```
apply1( asin,  X, Y ) :- !, Y is asin(X).
apply1( cos,   X, Y ) :- !, Y is cos(X).
apply1( acos,  X, Y ) :- !, Y is acos(X).
apply1( tan,   X, Y ) :- !, Y is tan(X).
apply1( atan,  X, Y ) :- !, Y is atan(X).
apply1( ln,    X, Y ) :- !, Y is log(X).
apply1( exp,   X, Y ) :- !, Y is exp(X).
apply1( log,   X, Y ) :- !, Y is log10(X).
apply1( sqrt,  X, Y ) :- !, Y is sqrt(X).
%
% system commands
%
apply1( op,  O, L ) :- !,
        (setof( term('OP',3,[T,A,P]), find_op( O, T, A, P ), L ); L=[]).
apply1( del, F, true ) :-
        code(F,_), !, delete_define( F ).
apply1( del, F, true ) :-
        macro(F,_,_), !, delete_macro( F ).
apply1( del, _, false ) :- !.
apply1( load, File, R ) :-
        write( 'Loading file: ' ), put( "" ), write( File ), put( "" ),nl,
        prompt_off,
        seeing( F ),
        ((see( File ), load_file, seen, R=true, nl,
         put( "" ), write( File ), put( "" ), write( ' is loaded.' ), nl)
         ;
         R=false),
        see( F ),
        reset_prompt.
apply1( vi, File, File ) :-
        name( File, List ),
        append( "vi ", List, Cmd ),
        system( Cmd ), !.
apply1( vi, File, error(exception(vi,File)) ) :- !.
apply1( save, File, true ) :-
        save( File ), !.
apply1( save, _, false ).


division_op( '/' ).
division_op( rem ).
division_op( mod ).


%
% binary builtin functions
%
apply2( F, [A,0], error('division by zero') ) :-
        division_op( F ), !.
apply2( rem, [A,_], error('non integer argument in rem') ) :- float(A), !.
apply2( rem, [_,B], error('non integer argument in rem') ) :- float(B), !.
apply2( mod, [A,_], error('non integer argument in mod') ) :- float(A), !.
apply2( mod, [_,B], error('non integer argument in mod') ) :- float(B), !.
apply2( mod, [_,B], error('non positive modulo') ) :- B < 0, !.

apply2( '+', [A,B], R ) :- !, R is A+B.
```



```
apply2( '-', [A,B], R ) :-  !, R is A-B.
apply2( '*', [A,B], R ) :- !, R is A*B.
apply2( '/', [A,B], R ) :- !, R is A/B.
apply2( rem, [A,B], C ) :- !, C is A mod B.
apply2( mod, [A,B], C ) :- !, R is A/B, C is (R-floor(R))*B.
apply2( '^', [A,B], R ) :- !, R is A^B.
apply2( '==', [A,B], C ) :- !, equal( A, B, C ).
apply2( '<', [A,B], C ) :- less_than( A, B, C ), !.
apply2( '>', [A,B], C ) :- !, less_than( B, A, C ).

%
% supporting predicates for the runtime system
%
equal( A, A, true ) :- !.
equal( A, B, false ) :- A \== B.


less_than( false, true,  true ) :- !.
less_than( true,  false, false ) :- !.
less_than( true,  true,  false ) :- !.
less_than( false, false, false ) :- !.
less_than( A, B, true ) :-
        number( A ),
        number( B ),
        A < B, !.
less_than( A, B, true ) :-
        atom( A ),
        atom( B ),
        A @< B, !.
less_than( A, B, false ) :-
        number( A ),
        number( B ),
        A >= B, !.
less_than( A, B, false ) :-
        atom( A ),
        atom( B ),
        A @>= B.


select( true,  Ct,  _, Ct ) :- !.
select( false,  _, Cf, Cf ).


load_file :- load_expressions.
load_file.


load_expressions :-
        next_expression( S ),
        process( S ), !,
        load_expressions.


reset_prompt :-
        seeing(F), F=user, !,
        prompt_on.
```

```
reset_prompt.


save( File ) :-
        telling(Cur),
        tell( File ),
        listing(operatorN),
        listing(operator3),
        listing(operator2),
        listing(operator1),
        listing(op_decl),
        listing(code),
        listing(type),
        retract(macro(it,it,_)),
        listing(macro),
        told,
        tell(Cur).


add_type_def( F, T ) :-
        delete_type_def( F ),
        assert( type( F, T ) ),
        !.
add_type_def( F, _ ) :-
        put( ^^G ), nl,
        write( 'Cannot add type definition ' ), write( F ), nl.


delete_type_def( F ) :-
        retract( type( F, _ ) ), !.
delete_type_def( _ ).


add_define( F, C ) :-
        delete_define( F ),
        assert( code( F, C ) ), !.
add_define( F, C ) :-
        put( ^^G ), nl,
        write( 'Cannot add definition ' ), write( F ), nl.


delete_define( F ) :-
        retract( code( F, _ ) ), !.
delete_define( _ ).

%
% locate( I, E, V ) :- returns the variable binding of I from the
%       environment E in V.
%
locate( [I|N], E, V ) :-
        index( I, E, E1 ),
        index( N, E1, V ).


index( 0, [S|_], S ) :- !.
index( N, [_|Ss], S ) :-
```



```
        N > 0,
        N1 is N-1,
        index( N1, Ss, S ).
```

```
%
% T Y P E   I N F E R E N C E
%
% Module:       type
%
% Written by: Mantis H.M. Cheng (July/04/88)
%

type_of(E,T) :-
        init_type_error,
        type_of(E,T,[]).

%
% type_of(E,T,N)
%       N is the environment of variable types, T is the type of the
%       expression E
%
type_of(num(C),type(simple,[num]),_) :- !.
type_of(const(C),type(simple,[sym]),_) :- !.
type_of(C,T,_) :-
        primitive_const(C), !,
        type_of_primitive_const(C,T).
%
% curried the binary primitive when it is used by itself
%
type_of(F,func([T1],func([T2],R)),_) :-
        atom(F),
        primitive2(F),
        type_of_primitive_func(F,func([T1,T2],R)), !.
type_of(F,T,_) :-
        atom(F),
        primitive_func(F),
        type_of_primitive_func(F,T), !.
type_of(C,T,N) :-       % look the variable type in the env
        atom(C),
        env_type_of(C,N,T), !.
type_of(C,T,_) :-       % look the variable type in the code
        atom(C), code(C,_),
        type(C,T), !.
%
% constructor terms
%
type_of(C,T,_) :-
        atom(C), constructor_type(C,T), !.
type_of(term(F,A,Args),T,N) :-
        constructor_type(cterm(F,A,ATs),T), !,
        type_of_args(Args,ATs,N).
type_of([$arg,A,F/P,S],T,N) :- !,
        type_of(S,CT,N),
        constructor_type(cterm(F,P,ATs),CT), !,
        choose(A,ATs,T).
%
% let expressions
%
type_of([let,rec|D],T,N) :- !,
        def_exprs(D,Es,E),
```



```
        check_struct_type(Es,N),
        exprs(Es,Args),
        vars(Es,Ns),
        make_env_type(Ns,Ns1,NsT),
        type_of_args(Args,NsT,[Ns1|N]),
        type_of(E,T,[Ns1|N]), !.
type_of([let|D],T,N) :- !,
        def_exprs(D,Es,E),
        check_struct_type(Es,N),
        exprs(Es,Args),
        vars(Es,Ns),
        make_env_type(Ns,Ns1,NsT),
        type_of_args(Args,NsT,N),
        type_of(E,T,[Ns1|N]), !.
%
% conditional expressions
%
type_of([';',C1,Cs],T,N) :- !,
        type_of_cond([';',C1,Cs],T,N).
%
% lambda expressions
%
type_of(['\',P,B],T,N) :-
        make_env_type(P,E,PT),
        type_of(B,BT,[E|N]), T=func(PT,BT), !.
%
% curried the binary function which is defined by an operator
%
type_of([F,A1,A2],R,N) :-
        atom(F), binary_op(F),
        type_of(F,func([T1],func([T2],R)),N),
        type_of(A1,T1,N),
        type_of(A2,T2,N), !.
%
% intercept the normal usage of binary primitives for efficiency reason
%
type_of([F,A1,A2],R,N) :-
        primitive2(F),
        type_of_primitive_func(F,func([T1,T2],R)),
        type_of(A1,T1,N),
        type_of(A2,T2,N), !.
%
% function application
%
type_of([F|As],T,N) :-
        type_of(F,FT,N),
        type_of_args(As,P,N),!, FT=func(P,T), !.
type_of(E,T,_) :-
        print_type_error( E ), !, fail.


%
% type_of_cond(Cls,T,N) checks if each clause in Cls is of type T given
%       the envs of type N
%
type_of_cond([';',P,Cs],T,N) :- !,
```

```
        conditional( P, C, E ),
        type_of(C,type(simple,[bool]),N),
        type_of(E,T,N),
        type_of_cond(Cs,T,N).
type_of_cond(P,T,N) :-
        conditional(P,C,E), !,
        type_of(C,type(simple,[bool]),N),
        type_of(E,T,N).
type_of_cond(P,T,N) :-
        last_conditional(P,E),
        type_of(E,T,N).


%
% type_of_args(As,Ts,N) computes the type Ts of each element in the
%       argument list As given the env of types N
%
type_of_args([],[],_) :- !.
type_of_args([P|Ps],[T|Ts],N) :-
        type_of(P,T,N), !,
        type_of_args(Ps,Ts,N).


%
% make_env_type(P,E,T) converts the parameters P into an env of types
%       E and a list of types T
%
make_env_type([],[],[]) :- !.
make_env_type([P|Ps],[P=T|E],[T|Ts]) :-
        make_env_type(Ps,E,Ts).


%
% check_struct_type(E,N) looks for any special binding constructs, if
% so, make sure it is type-compatible with the given type env.
%
check_struct_type([],_) :- !.
check_struct_type([['=',term(F,A,_),V]|Es], N ) :-
        atom(V), !,
        constructor_type(cterm(F,A,_),T),
        env_type_of(V,N,T),
        check_struct_type(Es,N).
check_struct_type([['=',term('.',2,[_,_]),V]|Es], N ) :-
        atom(V), !,
        constructor_type(cterm('.',2,_),T),
        env_type_of(V,N,T),
        check_struct_type(Es,N).
check_struct_type([_|Es], N ) :-
        check_struct_type(Es,N).


%
% env_type_of(A,E,T) finds the type T of the variable A in the envs of
%       type E
%
env_type_of(C,[E|_],T1) :-
```



```
        member(C=T,E), !, make_type(T,T1).
env_type_of(C,[_|Es],T) :-
        env_type_of(C,Es,T).


%
% This predicate is extremely complicated!!!
% In general, we don't want to disturb the env if the bindings of
% functions are known.
%
make_type( T1, T2 ) :- var(T1), var(T2), !, T1=T2.
make_type( T1, T2 ) :-
        nonvar(T1), T1=func(_,_), var(T2),
        !, copy( T1, T2 ).
make_type( T1, T2 ) :-
        nonvar(T1), T1=func(_,_), nonvar(T2), T2=func(_,_),
        !, copy( T2, T3 ), T1=T3.
make_type( T1, T2 ) :-
        nonvar(T1), nonvar(T2), T1=T2.
make_type( T1, T2 ) :-
        var(T1), nonvar(T2), T1=T2.
make_type( T1, T2 ) :-
        nonvar(T1), var(T2), copy(T1,T3), T1=T2.


constructor_symbol(F) :-
        constructor_type(F,_), !.
constructor_symbol(F) :-
        constructor_type(cterm(F,_,_),_), !.


add_new_constructor_type(CT,T) :-
        not constructor_type(CT,T),
        assert(constructor_type(CT,T)).


print_type_error(E) :-
        type_error_on,
        put( ^^G ),
        nl,
        write( 'Type error at ' ), print_expression(E), nl, nl,
        retract( type_error_on ), !.
print_type_error(_).


init_type_error :-
        not( type_error_on ),
        assert( type_error_on ), !.
init_type_error.


%
% copy( Term, NewTerm )
%       NewTerm is new copy of Term with fresh variables
%
copy( Term, NewTerm ) :-
```

```
        copy( Term, NewTerm, [], _ ).


copy( X, Q, V1, V2 ) :- var( X ), !, lookup( var(X,Q), V1, V2 ).
copy( X, X, V, V ) :- atomic( X ), !.
copy( [X|Y], [X1|Y1], V1, V3 ) :-
        !, copy( X, X1, V1, V2 ), copy( Y, Y1, V2, V3 ).
copy( C, C1, V1, V2 ) :-
        C =.. [F|As], copy( As, As1, V1, V2 ), C1 =.. [F|As1].


lookup( var(X,Q), [], [var(X,Q)] ) :- !.
lookup( var(X,Q), [var(Y,Q)|Vars], [var(Y,Q)|Vars] ) :-
        X == Y, !.
lookup( var(X,Q), [var(Y,P)|Vars], [var(Y,P)|Vars1] ) :-
        X \== Y, lookup( var(X,Q), Vars, Vars1 ).
```



```
%
% Module:       read
%
% Written by: Mantis H.M. Cheng (July/04/88)
%

%
% next_expression( S ) :- returns the next input expression.
%
next_expression( S ) :-
        next_char( [], L, '[> ' ),
        get_next_expression( L, S ), !.


get_next_expression( L, S ) :-
         get_next_token( L, L1, T ),
         tokenise( T, L1, Ts ),
         statement( S, Ts, _ ), !.
get_next_expression( _, _ ) :-
         put( ~^G ), nl,
         write( 'Syntax error '), nl, nl, !, fail.


tokenise( $end, L, [$end|L] ) :- !.   % terminator
tokenise( T, L, [T|Ts] ) :-
        next_char( L, L1, '<] ' ), !,
        get_next_token( L1, L2, T1 ),
        tokenise( T1, L2, Ts ).


get_next_token( L1, L2, T ) :-
        next_token( T, L1, L2 ), !.


%
% returns the next non-white-space character
%
next_char( [], S, P ) :- !,             % end of line
        ask_input( S0, P ),
        next_char( S0, S, P ).
next_char( [~%|_], S, P ) :- !,         % comment line
        ask_input( S0, P ),
        next_char( S0, S, P ).
next_char( [C|S0], S1, P ) :-
        is_white_space( C ), !,
        next_char( S0, S1, P ).
next_char( S, S, _ ).

%
% valid tokens
%
next_token( ',' ) --> [~,].
next_token( '[]' ) --> [~[,~]].
next_token( '()' ) --> [~(,~)].
next_token( '[' ) --> [~[].
next_token( ']' ) --> [~]].
```

```
next_token( '|' ) --> [~|].
next_token( '(' ) --> [~(].
next_token( ')' ) --> [~)].
next_token( '\' ) --> [~\].
next_token( '<<>>' ) --> [~<,~<,~>,~>].
next_token( '<<' ) --> [~<,~<].
next_token( '>>' ) --> [~>,~>].
next_token( $end ) --> [~.,C], {is_white_space(C),!}.
next_token( O ) --> operator(O).
next_token( ':' ) --> [~:].
next_token( '@' ) --> [~@].
next_token( '.' ) --> [~.].
next_token(num(T)) --> [C], {is_digit(C)}, number(N), {aton([C|N],T)}.
next_token(T) --> [C], {is_alpha(C)}, identifier(A),
        {name(T1,[C|A]), symbol_val(T1,T)}.
next_token(const(T)) --> [~"], quote_id(A), {name(T,A)}.


quote_id([~"|Cs]) --> [~",~"], {!}, quote_id(Cs).
quote_id([C|Cs]) --> [C], {C\==~"}, quote_id(Cs).
quote_id([]) --> [~"], {!}.


number([~.,C|Cs]) --> [~.], [C], {is_digit(C)}, float(Cs).
number([C|Cs]) --> [C], {is_exponent(C)}, exponent(Cs).
number([C|Cs]) --> [C], {is_digit(C)}, number(Cs).
number([]) --> [].


float([C|Cs]) --> [C], {is_digit(C)}, float(Cs).
float([C|Cs]) --> [C], {is_exponent(C)}, exponent(Cs).
float([]) --> [].


exponent([C|Cs]) --> [C], {is_sign(C)}, digits(Cs).
exponent(Cs) --> digits(Cs).


digits([C|Cs]) --> [C], {is_digit(C)}, digits(Cs).
digits([]) --> [].


identifier([C|Cs]) --> [C], {is_valid(C)}, identifier(Cs).
identifier([]) --> [].

%
% auxiliary predicates
%

%
% tokens delimiters
%
reserve( ~( ).
reserve( ~) ).
reserve( ~] ).
reserve( ~[ ).
```



```
reserve( ~| ).
reserve( ~; ).
reserve( ~, ).
reserve( ~% ).
reserve( ~. ).


symbol_val( S, S ) :- keyword(S), !.
symbol_val( S, S ) :- op_decl(_,_,S), !.
symbol_val( S, id(S) ).


keyword(macro).
keyword(type).
keyword(rec).
keyword(let).
keyword(in).


token( ':' ).
token( ',' ).
token( '[]' ).
token( '()' ).
token( '[' ).
token( ']' ).
token( '|' ).
token( '(' ).
token( ')' ).
token( '\' ).
token( '<<>>' ).
token( '<<' ).
token( '>>' ).

%
% Module:       read
%
% auxiliary predicates
%

%
% valid characters in a symbol
%
is_valid( C ) :- is_alpha( C ), !.
is_valid( C ) :- is_digit( C ), !.
is_valid( ~_ ).
is_valid( ~? ).
is_valid( ~' ).


is_digit( C ) :-
        ~0 =< C,
        C =< ~9.


%
% valid characters starting a symbol
```

```
%
is_alpha( C ) :-
        ~a =< C,
        C =< ~z.
is_alpha( C ) :-
        ~A =< C,
        C =< ~Z.


is_sign( ~+ ).
is_sign( ~- ).


is_exponent( ~e ).
is_exponent( ~E ).


%
% tokens delimiters
%
is_special( C ) :- is_white_space( C ), !.
is_special( C ) :- reserve( C ).


is_white_space( ~  ).
is_white_space( ~^I ).


%
% change the internal sign of a number if necessar.
%
sign_extension( ~+, S, S ) :- !.
sign_extension( ~-, S1, S2 ) :- S2 is -S1.

%
% aton( L, N ) :-
%       converts a list of digits L into a number (integer or float).
%
aton( L, N ) :-
        integral_part( L, L1, 0, I ),
        fractional_part( L1, L2, 0, F ),
        exponent_part( L2, [], E ),
        N is (F+I)*E.


integral_part( [~.|L], [~.|L], N, N ) :- !.
integral_part( [C|L], [C|L], N, N ) :-
        is_exponent( C ), !.
integral_part( [], [], N, N ) :- !.
integral_part( [C|L], L1, A, V ) :-
        V1 is (C-~0)+A*10,
        integral_part( L, L1, V1, V ).


fractional_part( [], [], F, F ) :- !.
fractional_part( [C|L], [C|L], F, F ) :-
```



```
        is_exponent( C ), !.
fractional_part( [~.|L], L1, _, F ) :- !,
        fractional_part( L, L1, 0.0, F ).
fractional_part( [C|L], L1, _, F ) :-
        fractional_part( L, L1, 0.0, F1 ),
        F is ((C-~0)+F1)/10.


exponent_part( [], [], 1 ) :- !.
exponent_part( [C|L], L1, E ) :-
        is_exponent( C ), !,
        exponent_part( L, L1, E ).
exponent_part( [C|L], L1, E ) :-
        is_sign( C ), !,
        exponent_part( L, L1, E1 ),
        exp_sign_extension( C, E1, E ).
exponent_part( L, L1, E ) :-
        integral_part( L, L1, 0, I ),
        E is 10^I.


exp_sign_extension( ~+, E, E ) :- !.
exp_sign_extension( ~-, E, E1 ) :- E1 is 1/E.


%
% I/O handling
%
ask_input( S, N ) :-
        prompt_user( N ),
        get0( C ),
        read_line( C, S ).


read_line( -1, [] ) :- !, fail.
read_line( ~^J, [~ ] ) :- !.
read_line( ~^M, [~ ] ) :- !.
read_line( C, [C|Cs] ) :-
        get0( C0 ),
        read_line( C0, Cs ).



%
% turn the system prompt on.
%
prompt_on :-
        retract( prompt(_) ),
        assert( prompt(on) ).


%
% turn the system prompt off.
%
prompt_off :-
        retract( prompt(_) ),
```

```
        assert( prompt(off) ).

%
% show the system prompt if it is on.
%
prompt_user(N) :-
        prompt(on), !,
        write( N ).
prompt_user(_).


prompt(on).
```



```
%
% Module:       util
%
% Written by: Mantis H.M. Cheng (July/04/88)
%
% Utility predicates
%


%
% print_answer_with_type( N, T, A ) prints the answer A with type T
%       associated with name N.
%
% N.B.  It won't print the contents of a function, only its type.
%
print_answer_with_type( N, _, error(E) ) :- !,
        put( ^^G ),
        write( 'val ' ), write( N ), write( ' = ' ),
        print_error( E ),
        write( ' : error' ), nl, nl.
print_answer_with_type( N, macro, _ ) :-
        !,
        write( 'val ' ), write( N ), write( ' : macro' ),
        nl, nl.
print_answer_with_type( N, func(P,T), _ ) :-
        !,
        write( 'val ' ), write( N ), write( ' = fn : ' ),
        print_type( func(P,T) ),
        nl, nl.
print_answer_with_type( N, T, V ) :-
%       T <> func(_,_),
        write( 'val ' ), write( N ), write( ' = ' ),
        print_answer( V ),
        write( ' : ' ),
        print_type( T ),
        nl, nl.

%
% print_type(T) prints the type T in infix notation
%
print_type(func([],T)) :-
        !,
        write( '() -> ' ),
        var_names( N ),
        print_type(T,N,_).
print_type(func(P,T)) :-
        P \== [], !,
        var_names( N ),
        print_type(P,N,N1),
        write( ' -> ' ),
        print_type(T,N1,_).
print_type(T) :-
        var_names( N ),
        print_type(T,N,_).
```

```
print_type(X,[N|Ns],Ns) :-
        var(X), !,
        write( N ),
        X = N.
print_type(A,Ns,Ns) :-
        atom(A), A=$void, !,
        write( '()' ).
print_type(A,Ns,Ns) :-
        atom(A), !, write( A ).
print_type(type(simple,[X]),N,N) :-
        var(X), !,
        write( 'simple' ).
print_type(type(simple,[X]),N,N) :-
        atom(X), !,
        write(X).
print_type(type(tuple,P),N1,N2) :-
        put("("),
        print_struct_type(P,N1,N2),
        put(")").
print_type(type(T,[]),N1,N2) :-
        !, print_type(T,N1,N2).
print_type(type(T,[A|AT]),N1,N2) :- !,
        write(T),put("("),
        print_struct_type([A|AT],N1,N2),
        put(")").
print_type(func([],T),N1,N2) :-
        !,
        put( "(" ),
        write( '() -> ' ),
        print_type(T,N1,N2),
        put( ")" ).
print_type(func(P,T),N1,N3) :-
        !,
        put( "(" ),
        print_type(P,N1,N2),
        write( ' -> ' ),
        print_type(T,N2,N3),
        put( ")" ).
/*
 * Since all functions are curried, we don't need the following clauses.
 *
print_type([],N,N) :- !.
print_type([P1,P2|Ps],N1,N3) :-
        print_type(P1,N1,N2),
        write( ' x ' ),
        print_type([P2|Ps],N2,N3).
 */
print_type([P],N1,N2) :-
        print_type(P,N1,N2).


print_struct_type([],N,N) :- !.
print_struct_type([X],N1,N2) :-
        print_type(X,N1,N2).
print_struct_type([X1,X2|Xs],N1,N3) :-
        print_type(X1,N1,N2),
```



```
        write( ' x ' ),
        print_struct_type([X2|Xs],N2,N3).


print_error( arg(T) ) :- !,
        write( 'Illegal argument in constructor term ' ),
        print_answer(T).
print_error( unbound(C) ) :- !,
        write( 'Unbound variable ' ), write( C ).
print_error( exception(F,A) ) :- !,
        write( 'Exception in "' ), write( F ), write( '" with argument ' ),
        print_ans_list( A ).
print_error( fault(P) ) :- !,
        write( 'Missing condition after ' ), print_expression( P ).
print_error( def(F) ) :- !,
        write( 'Illegal definition ' ), print_expression( F ).
print_error( op(O,T,A) ) :-
        invalid_op_decl(T,A), !,
        write( 'Invalid operator declaration ' ), write( O ).
print_error( op(O,T,A) ) :-
%       not(invalid_op_decl(T,A)),
        !,
        write( 'Cannot redefine operator ' ), write( O ).
print_error( E ) :- write( E ).


invalid_op_decl(infix,assoc).
invalid_op_decl(prefix,left).
invalid_op_decl(prefix,right).
invalid_op_decl(postfix,left).
invalid_op_decl(postfix,right).


member( X, [X|_] ).
member( X, [_|T] ) :-
        member( X, T ).


append( [], L, L ) :- !.
append( [U|X], Y, [U|Z] ) :- append( X, Y, Z ).


var_names(['X','Y','Z','X1','Y1','Z1','T1','T2','T3','T4','T5','T6',
           'T7','T8']).


choose( 1, [H|_], H ) :- !.
choose( N, [_|T], E ) :-
        N > 1, !, N1 is N-1, choose( N1, T, E ).
choose( _, T, error(arg) ) :- print_answer(T).


print_answer( [] ) :- !,
        write( '[]' ).
print_answer( X ) :-
        atomic( X ), !, write( X ).
```

```
print_answer( cl(_,_) ) :-
        !, write( 'fn' ).
print_answer( r(_,_,_) ) :-
        !, write( '...' ).
print_answer( [X|Y] ) :- !,
        put( "[ ),
        print_ans_list( [X|Y] ),
        put( "] ).
print_answer( term('CONS',2,P) ) :-
        !, write( '<<' ),
        print_ans_list( P ),
        write( '>>' ).
print_answer( term('TUPLE',_,P) ) :-
        !, put( "( ),
        print_ans_list( P ),
        put( ") ).
print_answer( term(F,_,P) ) :-
        write( F ), put( "( ),
        print_ans_list( P ),
        put( ") ).


print_ans_list( [] ) :- !.
print_ans_list( [X] ) :- !,
        print_answer( X ).
print_ans_list( [X1,X2|Y] ) :-
        print_answer( X1 ),
        put( ", ),
        print_ans_list( [X2|Y] ).



%
% print an expression in its original syntax based on the operator
% declarations
%
print_expression( X ) :-
        print_term( 1200, X ), !.


print_list( term('.',2,[H1,term('.',2,[H2,Hs])]) ) :- !,
        print_term( 1200, H1 ),
        put( ", ),
        print_list( term('.',2,[H2,Hs]) ).
print_list( term('.',2,[H,$nil]) ) :- !,
        print_term( 1200, H ).
print_list( term('.',2,[H,Hs]) ) :-
        Hs \== $nil, !,
        print_term( 1200, H ),
        put( "| ),
        print_term( 1200, Hs ).


print_term( _, $nil ) :- !,
        write( '[]' ).
print_term( _, $void ) :- !,
```



```
        write( '()' ).
print_term( _, num(A) ) :- !,
        write( A ).
print_term( _, const(A) ) :- !,
        write( A ).
print_term( _, A ) :-
        atomic( A ), !,
        write( A ).
print_term( _, [let,rec,B] ) :- !,
        write( 'let rec ' ),
        print_term( 1200, B ).
print_term( _, [let,rec,B1|Bs] ) :-
        Bs \== [], !,
        write( 'let rec ' ),
        print_let_exp( [B1|Bs] ).
print_term( _, [let,B] ) :- !,
        write( 'let ' ),
        print_term( 1200, B ).
print_term( _, [let,B1|Bs] ) :-
        Bs \== [], B1 \== rec, !,
        write( 'let ' ),
        print_let_exp( [B1|Bs] ).
print_term( H, [F,A] ) :-
        op_decl( L, fx, F ), !,
        L1 is L-1,
        print_left_paren( H, L1 ),
        write( F ),
        put( ~  ),
        print_term( L1, A ),
        print_right_paren( H, L1 ).
print_term( H, [F,A] ) :-
        op_decl( L, fy , F ), !,
        print_left_paren( H, L ),
        write( F ),
        put( ~  ),
        print_term( L, A ),
        print_right_paren( H, L ).
print_term( H, [F,A] ) :-
        op_decl( L, xf, F ), !,
        L1 is L-1,
        print_left_paren( H, L1 ),
        print_term( L1, A ),
        put( ~  ),
        write( F ),
        print_right_paren( H, L1 ).
print_term( H, [F,A] ) :-
        op_decl( L, yf, F ), !,
        print_left_paren( H, L ),
        print_term( L, A ),
        put( ~  ),
        write( F ),
        print_right_paren( H, L ).
print_term( _, ['\',P,E] ) :- !,
        put( "\ ),
        print_lambda_exp( P, E ).
print_term( H, [F,A1,A2] ) :-
```

```
        op_decl( L, xfx, F ), !,
        L1 is L-1,
        print_left_paren( H, L ),
        print_term( L1, A1 ),
        put( ~  ),
        write( F ),
        put( ~  ),
        print_term( L1, A2 ),
        print_right_paren( H, L ).
print_term( H, [F,A1,A2] ) :-
        op_decl( L, xfy, F ), !,
        L1 is L-1,
        print_left_paren( H, L ),
        print_term( L1, A1 ),
        put( ~  ),
        write( F ),
        put( ~  ),
        print_term( L, A2 ),
        print_right_paren( H, L ).
print_term( H, [F,A1,A2] ) :-
        op_decl( L, yfx, F ), !,
        L1 is L-1,
        print_left_paren( H, L ),
        print_term( L, A1 ),
        put( ~  ),
        write( F ),
        put( ~  ),
        print_term( L1, A2 ),
        print_right_paren( H, L ).
print_term( H, [F,P] ) :- !,
        print_term( H, [':',F,P] ).
print_term( H, [F] ) :- !,
        print_term( H, [':',F,$void] ).
print_term( _, term('.',2,[H,Hs]) ) :- !,
        put( ~[ ),
        print_list( term('.',2,[H,Hs]) ),
        put( ~] ).
print_term( _, term('CONS',2,P) ) :-
        !, write( '<<' ),
        print_cons( P ),
        write( '>>' ).
print_term( _, term('TUPLE',_,P) ) :-
        !, put( ~( ),
        print_tuple( P ),
        put( ~) ).
print_term( H, term(F,_,P) ) :- !,
        write( F ), put( ~( ),
        print_tuple( P ),
        put( ~) ).
print_term( _, A ) :-
        write( A ).


print_left_paren( H, L ) :-
        L > H, !,
        put( ~( ).
```



```
print_left_paren( _, _ ).


print_right_paren( H, L ) :-
        L > H, !,
        put( ~) ).
print_right_paren( _, _ ).



print_lambda_exp( [V1], ['\',P,E] ) :-
        !,
        write( V1 ),
        put( ~, ),
        print_lambda_exp( P, E ).
print_lambda_exp( [V], E ) :- !,
        write( V ),
        put( ~. ),
        print_term( 1200, E ).
print_lambda_exp( [], E ) :-
        put( ~. ),
        print_term( 1200, E ).


print_let_exp( [B1,B2,B3|Bs] ) :- !,
        print_term( 1200, B1 ),
        write( ', ' ),
        print_let_exp( [B2,B3|Bs] ).
print_let_exp( [B,E] ) :-
        print_term( 1200, B ),
        write( ' in ' ), print_term( 1200, E ).


print_cons( [E,'<<>>'] ) :- !,
        print_term( 1200, E ).
print_cons( [E1,term('CONS',2,P)] ) :- !,
        print_term( 1200, E1 ),
        put( ~, ),
        print_cons( P ).
print_cons( [E1,E2] ) :-
        print_term( 1200, E1 ),
        put( ~| ),
        print_term( 1200, E2 ).


print_tuple( [] ) :- !.
print_tuple( [E] ) :-
        print_term( 1200, E ).
print_tuple( [E1,E2|Es] ) :-
        print_term( 1200, E1 ),
        put( ~, ),
        print_tuple( [E2|Es] ).
```

```
%
% Module:       stmt
%
% Written by: Mantis H.M. Cheng (July/04/88)
%
% This module defines grammar of FPL in DCG.
%

%
% macro <named-exp> = <expression>
%
statement([macro,H,E]) --> [macro], {!}, function_exp(H), ['='],
        expression(E), [$end].

%
% type <new-type>(<type-vars>) = <constructor-term> | ...
%
% where <new-type> cannot already exist and <type-vars> are type
% variables
%
statement(T) --> [type], {!}, function_exp(F), {make_new_type(F,T)},
        ['='], type_expression(T), type_expressions(T), [$end].

statement(op(O,T,A,P)) --> op_type(T), op_assoc(A), [num(P)],
                {P>=0,P=<1200}, op_name(O), [$end].

%
% expression evaluation
%
statement(E) --> expression(E), [$end].


type_expressions(F) --> ['|'], type_expression(F), type_expressions(F).
type_expressions(_) --> [].


%
% a type <constructor-term> must be of the form
%
%       <constructor>(T1,...,Tn),       n >= 0
%
% where Ti must be a simple type, a type variable  or a recursively
% defined type
%
type_expression(type(F/P,P1,Vs)) --> function_name(F1), {F\==F1}, ['('], {!},
        type_term(F/P,T), type_terms(F/P,Ts), [')'],
        {length([T|Ts],A), replace_var([T|Ts],P2,Vs),
         add_new_constructor_type(cterm(F1,A,P2),type(F,P1))}.
type_expression(type(F/_,P1,Vs)) --> function_name(T),
        {F\==T, add_new_constructor_type(T,type(F,P1))}.


type_terms(F,[P|Ps]) --> [','], {!}, type_term(F,P), type_terms(F,Ps).
type_terms(_,[]) --> [].
```



```
%
% possible terms in a type declaration
%
type_term(_,type(simple,[num])) --> [id(num)].
type_term(_,type(simple,[sym])) --> [id(sym)].
type_term(_,type(simple,[bool])) --> [id(bool)].
type_term(_,type(simple,[_])) --> [id(simple)].
type_term(_/P,T) --> [id(T)], {member(T,P)}.
type_term(F/P,type(F,P)) --> function_name(F), ['('], {!}, parameter_list(P).
type_term(F/[],type(F,[])) --> function_name(F).


%
% make_new_type(FD,T) constructs a new type definition T given the function
%       expression FD.
%
make_new_type([F|P],type(F/P,P1,Vs)) :-
        not constructor_type(_,type(F,_)), !,
        generate_free_var(P,P1,Vs).
make_new_type(F,type(F/[],[],[])) :-
        atom(F),
        not constructor_type(_,type(F,_)), !.


%
% possible operator types
%
op_type(infix) --> [id(infix)].
op_type(prefix) --> [id(prefix)].
op_type(postfix) --> [id(postfix)].


%
% possible operator associativities
%
op_assoc(non) --> [id(non)].
op_assoc(assoc) --> [id(assoc)].
op_assoc(left) --> [id(left)].
op_assoc(right) --> [id(right)].


op_name(O) --> [id(O)].
op_name(O) --> [const(O)].


function_exp([F|P]) --> function_name(F), ['('], {!}, parameter_list(P).
function_exp(F) --> function_name(F).


constructor_exp(T) --> constructor_name(F),
        ['('], {!}, expression(P), arg_list(Ps), [')'],
        {make_constructor_exp([F,P|Ps],T)}.
constructor_exp(F) --> constructor_name(F).


make_constructor_exp([F|P],term(F,N,P)) :-
```

```
        length(P,N), constructor_type(cterm(F,N,_),_), !.
make_constructor_exp([F|P],[F|P]) :- macro(F,_,_).


constructor_name(F) --> [id(F)], {macro(F,_,_)}.
constructor_name(F) --> [id(F)], {constructor_symbol(F)}.


arg_list([E|Es]) --> [','], {!}, expression(E), arg_list(Es).
arg_list([]) --> [].


parameter_list([P|Ps]) --> [id(P)], {!}, bound_vars(Ps), [')'].
parameter_list([]) --> [].


function_name(F) --> [id(F)], {not primitive_const(F), !}.
function_name(F) --> [F], {not keyword(F), not token(F)}.


%=============================================================
% Treating terms as ops, an op is declared as
%     op(N,T,O)
% where N is the precedence between 0 and 1200,
%       T is the type of operator: fx, fy, xf, yf, xfx, xfy, yfx, and
%       O is name of the op.
%=============================================================
%
expression(T) --> term(1200,_,T).

term(N,M,T) --> term1(N,M1,L), term2(N,M,L,M1,T).


%
% non left-recursive terms
%
term1(_,0,num(T)) --> [num(T)].
term1(_,0,$nil) --> ['[]'].
term1(_,0,$void) --> ['()'].
term1(_,0,'<<>>') --> ['<<>>'].
term1(_,0,F) --> [id(F)], {primitive_const(F)}.
term1(_,0,const(F)) --> [const(F)].
%
% a "\x1,...,xn.<expression>" is an expression
%
term1(_,0,E) --> lambda_expression(E).
%
% a "let V1=E1,...,Vn=En [in <expression>]" is an expression
%
term1(_,0,E) --> local_term(E).
%
% a tuple (T1,...,Tn) is an expression
%
term1(_,0,T) --> ['('], expression(E), expressions(Es), [')'],
                 {make_tuple([E|Es],T)}.
%
```



```
% a stream <<T1,...,Tn>> is an expression
%
term1(_,0,term('CONS',2,[E,Es])) --> ['<<'], expression(E),
                 cons_expression_list(Es), ['>>'].
%
% a list [T1,...,Tn] is an expression
%
term1(_,0,term('.',2,[E,Es])) --> ['['], expression(E),
                 expression_list(Es), [']'].
term1(B,N,O) --> op(N,fx,O), {N=<B}.
term1(B,N,O) --> op(N,fy,O), {N=<B}.
term1(B,N,T) --> op(N,fx,O), {N=<B,N1 is N-1}, term(N1,_,T1),
                 { check_sign(O,T1,T) }.
term1(B,N,[O,T]) --> op(N,fy,O), {N=<B}, term(N,_,T).
term1(_,0,E) --> constructor_exp(E).
term1(_,0,T) --> function_name(T).
term1(_,0,':') --> [':'].
term1(_,0,'@') --> ['@'].

%
% left recursive terms
%
term2(B,N1,L,M,T) --> op(N,xfx,O), {N=<B,M<N,B1 is N-1},
                 term(B1,_,R), term2(B,N1,[O,L,R],N,T).
term2(B,N1,L,M,T) --> op(N,xfy,O), {N=<B,M<N}, term(N,_,R),
                 {construct_term([O,L,R],L1)},term2(B,N1,L1,N,T).
term2(B,N1,L,M,T) --> op(N,yfx,O), {N=<B,M=<N,B1 is N-1},
                 term(B1,_,R), {construct_term([O,L,R],L1)},
                 term2(B,N1,L1,N,T).
term2(B,N1,L,M,T) --> op(N,xf,O), {N=<B,M<N}, term2(B,N1,[O,L],N,T).
term2(B,N1,L,M,T) --> op(N,yf,O), {N=<B,M=<N}, term2(B,N1,[O,L],N,T).
term2(_,N,T,N,T) --> [].


%
% two primitive operators which cannot be redefined
%
op(100,yfx,':' ) --> [':'].
op(N,T,O) --> [O], {op_decl(N,T,O)}.


%
% function application and composition
%
construct_term( [':',L,$void], [L] ) :- !.
construct_term( [':',L,R], [L,R] ) :- R \== $void, !.
construct_term( ['@',L,$void], [L] ) :- !.
construct_term( ['@',L,R], [L,R] ) :- R \== $void, !.
construct_term( X, X ).


%
% change the internal sign of a number if necessary.
%
check_sign( '+', num(T), num(T) ) :- !.
check_sign( '-', num(T), num(T1) ) :- !, T1 is -T.
```

```
check_sign( O, T, [O,T] ).


expressions([E|Es]) --> [','], {!}, expression(E), expressions(Es).
expressions([]) --> [].


make_tuple([E1],E1) :- !.
make_tuple(Es,term('TUPLE',N,Es)) :- length(Es,N).


cons_expression_list(E) --> ['|'], {!}, expression(E).
cons_expression_list(term('CONS',2,[E,Es])) --> [','], {!}, expression(E),
                        cons_expression_list(Es).
cons_expression_list('<<>>') --> [].


expression_list(E) --> ['|'], {!}, expression(E).
expression_list(term('.',2,[E,Es])) --> [','], {!}, expression(E),
                        expression_list(Es).
expression_list($nil) --> [].


%
% local block
%
% let X1=E1, ... ,Xn=En in <expression>
% let rec X1=E1, ... ,Xn=En in <expression>
%
local_term([let,rec|B]) --> [let,rec], [!],
                        local_bindings(E,B), body_expression(E).
local_term([let|B]) --> [let], {!}, local_bindings(E,B), body_expression(E).

body_expression([E]) --> [in], {!}, expression(E).
body_expression([]) --> [].

local_bindings(E,[B|Bs]) --> binding_pair(B), bindings(E,Bs).

bindings(E,[B|Bs]) --> [','], {!}, binding_pair(B), bindings(E,Bs).
bindings(E,E) --> [].

binding_pair(['=',N,V]) --> expression(['=',H,E]),
       {make_name_binding(H,E,N,V)}.

make_name_binding(term(F,A,P),E,term(F,A,P),E) :- !.
make_name_binding(N,E,N,E) :-
       atom(N), !.
make_name_binding([F],E,N,V) :- !,
       make_name_binding(F,['\',[],E],N,V).
make_name_binding([F,term(T,A,P)],E,N,V) :-
       !,
       make_name_binding(F,['\',['$'],[let,['=',term(T,A,P),'$'],E]],N,V).
make_name_binding([F,P],E,N,V) :-
       atom(P), !,
       make_name_binding(F,['\',[P],E],N,V).
make_name_binding([F,P1,P2],E,N,V) :-
```



```
       binary_op(F),
       make_name_binding([[F,P1],P2],E,N,V).

%
% lambda expression
%
% \<parameters>.<expression>
%
lambda_expression(['\',[],E]) --> ['\','.'], {!}, expression(E).
lambda_expression(L) --> ['\'],bound_exp(A),bound_vars(P),
       ['.'], expression(E), {make_lambda([A|P],E,L)}.


bound_vars([A|As]) --> [','], bound_exp(A), {!}, bound_vars(As).
bound_vars([]) --> [].


bound_exp(T) --> expression(T), {((T=term(_,_,P),atomlist(P)); atom(T))}.


atomlist([]) :- !.
atomlist([A|As]) :- atom(A), atomlist(As).


make_lambda([],E,E) :- !.
make_lambda([term(F,A,P)|Ps],E,['\',['$'],[let,['=',term(F,A,P),'$'],L1]]) :-
       !, make_lambda(Ps,E,L1).
make_lambda([P|Ps],E,['\',[P],L1]) :- atom(P),make_lambda(Ps,E,L1).
```

```
%
% Module:       op
%
% Written by: Mantis H.M. Cheng (July/04/88)
%

binary_op( O ) :- op_decl(_,xfx,O), !.
binary_op( O ) :- op_decl(_,xfy,O), !.
binary_op( O ) :- op_decl(_,yfx,O).


add_op( O, infix,  non,   P ) :-
        add_op( O, xfx, P ), !.
add_op( O, infix,  left,  P ) :-
        add_op( O, yfx, P ), !.
add_op( O, infix,  right, P ) :-
        add_op( O, xfy, P ), !.
add_op( O, prefix, non,   P ) :-
        add_op( O, fx, P ), !.
add_op( O, prefix, assoc, P ) :-
        add_op( O, fy, P ), !.
add_op( O, postfix, non,   P ) :-
        add_op( O, xf, P ), !.
add_op( O, postfix, assoc,  P ) :-
        add_op( O, yf, P ).


add_op( O, T, P ) :-
        not( op_decl(_,T,O) ),
        name( O, Os ),
        length( Os, N ),
        append( Os, S2, S1 ),
        assert( op_decl(P,T,O) ),
        (is_valid_id(O) -> true; add_new_op( N, O, S1, S2 )).


is_valid_id(O) :-
        name( O, L ),
        valid_id( L ).


valid_id( [] ) :- !.
valid_id( [C|Cs] ) :- is_valid(C), valid_id(Cs).


add_new_op( 3, O, S1, S2 ) :-
        !, assert( operator3(O,S1,S2) ).
add_new_op( 2, O, S1, S2 ) :-
        !, assert( operator2(O,S1,S2) ).
add_new_op( 1, O, S1, S2 ) :-
        !, assert( operator1(O,S1,S2) ).
add_new_op( _, O, S1, S2 ) :-
        assert( (operatorN(O,S1,S2):-!) ).


operator( O, S1, S2 ) :-
```



```
        operatorN( O, S1, S2 ), !.
operator( O, S1, S2 ) :-
        operator3( O, S1, S2 ), !.
operator( O, S1, S2 ) :-
        operator2( O, S1, S2 ), !.
operator( O, S1, S2 ) :-
        operator1( O, S1, S2 ).


find_op(O,infix,non,P) :- op_decl(P,xfx,O).
find_op(O,infix,left,P) :- op_decl(P,yfx,O).
find_op(O,infix,right,P) :- op_decl(P,xfy,O).
find_op(O,postfix,non,P) :- op_decl(P,xf,O).
find_op(O,postfix,assoc,P) :- op_decl(P,yf,O).
find_op(O,prefix,non,P) :- op_decl(P,fx,O).
find_op(O,prefix,assoc,P) :- op_decl(P,fy,O).
```

```
%
% Module:       init
%
% Written by: Mantis H.M. Cheng (July/04/88)
%
% This module contains definitions of all primitive functions and
% constants.
%

primitive( F ) :- primitive_const( F ).
primitive( F ) :- primitive_func( F ).

%
% List of primitive functions
%
primitive_func( F ) :- primitive0( F ).
primitive_func( F ) :- primitive1( F ).
primitive_func( F ) :- primitive2( F ).

%
% constants
%
primitive_const( $void ).
primitive_const( true ).
primitive_const( false ).
primitive_const( fns ).
primitive_const( builtins ).
primitive_const( macros ).
primitive_const( ops ).
primitive_const( help ).

%
% nullary functions
%
primitive0( random ).

%
% unary functions
%
primitive1( hd ).
primitive1( tl ).
primitive1( head ).
primitive1( tail ).
primitive1( fst ).
primitive1( snd ).
primitive1( print ).
primitive1( echo ).
primitive1( trunc ).
primitive1( sin ).
primitive1( asin ).
primitive1( cos ).
primitive1( acos ).
primitive1( tan ).
primitive1( atan ).
primitive1( ln ).
primitive1( exp ).
```



```
primitive1( log ).
primitive1( sqrt ).
primitive1( load ).
primitive1( vi ).
primitive1( op ).
primitive1( del ).
primitive1( save ).
primitive1( error ).
primitive1( 'integer?' ).
primitive1( 'float?' ).

%
% binary functions
%
primitive2( '+' ).
primitive2( '-' ).
primitive2( '*' ).
primitive2( '/' ).
primitive2( mod ).
primitive2( rem ).
primitive2( '^' ).
primitive2( '==' ).
primitive2( '<' ).
primitive2( '>' ).

%
% primitive function types
%
type_of_primitive_const( $void,    type($void,[]) ).
type_of_primitive_const( true,     type(simple,[bool]) ).
type_of_primitive_const( false,    type(simple,[bool]) ).
type_of_primitive_const( help,     type(simple,[sym]) ).
type_of_primitive_const( builtins, type(list,[type(simple,[sym])]) ).
type_of_primitive_const( fns,      type(list,[type(simple,[sym])]) ).
type_of_primitive_const( macros,   type(list,[type(simple,[sym])]) ).
type_of_primitive_const( ops,      type(list,[type(simple,[sym])]) ).

type_of_primitive_func( random, func([],type(simple,[num])) ).

type_of_primitive_func( load, func([type(simple,[sym])],type(simple,[bool])) ).
type_of_primitive_func( vi,   func([type(simple,[sym])],type(simple,[sym])) ).
type_of_primitive_func( del,  func([type(simple,[sym])],type(simple,[bool])) ).
type_of_primitive_func( save, func([type(simple,[sym])],type(simple,[bool])) ).
type_of_primitive_func( print,func([_],type(simple,[bool])) ).
type_of_primitive_func( echo, func([T],T) ).
type_of_primitive_func( op,func([type(simple,[sym])],type(list,[type(op,[])]))).

type_of_primitive_func( F, func([type(simple,[num])],type(simple,[num])) ) :-
      arith_unary_op(F).
type_of_primitive_func( hd, func([type(list,[T])],T) ).
type_of_primitive_func( tl, func([type(list,[T])],type(list,[T])) ).
type_of_primitive_func( head, func([type(stream,[T])],T) ).
type_of_primitive_func( tail, func([type(stream,[T])],type(stream,[T])) ).
type_of_primitive_func( fst, func([type(tuple,[T,_])],T) ).
type_of_primitive_func( snd, func([type(tuple,[_,T])],T) ).
type_of_primitive_func( error, func([_],_) ).
```

```
type_of_primitive_func( 'integer?', func([type(simple,[num])],
        type(simple,[bool])) ).
type_of_primitive_func( 'float?', func([type(simple,[num])],
        type(simple,[bool])) ).

type_of_primitive_func( A, func([type(simple,[num]),type(simple,[num])],
        type(simple,[num])) ) :- arith_binary_op(A).
type_of_primitive_func( '==', func([N,N],type(simple,[bool])) ).
type_of_primitive_func( R, func([type(simple,[X]),type(simple,[X])],
        type(simple,[bool])) ) :- relational_op(R).


relational_op( '<>' ).
relational_op( '<' ).
relational_op( '>' ).


arith_unary_op( trunc ).
arith_unary_op( log ).
arith_unary_op( ln ).
arith_unary_op( exp ).
arith_unary_op( sqrt ).
arith_unary_op( F ) :- trigo_func( F ).


arith_binary_op( '+' ).
arith_binary_op( '-' ).
arith_binary_op( '*' ).
arith_binary_op( '/' ).
arith_binary_op( rem ).
arith_binary_op( mod ).
arith_binary_op( '^' ).


trigo_func( sin ).
trigo_func( cos ).
trigo_func( tan ).
trigo_func( asin ).
trigo_func( acos ).
trigo_func( atan ).


simple_type_constructor( 'NUM' ).
simple_type_constructor( 'BOOL' ).
simple_type_constructor( 'SYM' ).


%
% primitive constructor types
%

constructor_type( cterm('NUM', 1,[type(simple,[num])]),  type(simple,[_]) ).
constructor_type( cterm('BOOL',1,[type(simple,[bool])]), type(simple,[_]) ).
constructor_type( cterm('SYM', 1,[type(simple,[sym])]),  type(simple,[_]) ).

constructor_type( $nil,  type(list,[_]) ).
```



```
constructor_type( cterm('.',2,[X,type(list,[X])]), type(list,[X]) ).

%
% user defined constructor types
%
constructor_type( '<<>>',  type(stream,[_]) ).
constructor_type( cterm('CONS',2,[X,type(stream,[X])]), type(stream,[X]) ).

constructor_type( 'EMPTY',  type(tree,[_]) ).
constructor_type( cterm('NODE',3,[type(tree,[X]),X,type(tree,[X])]),
        type(tree,[X]) ).

constructor_type( cterm('LEAF',1,[X]), type(extree,[X]) ).
constructor_type( cterm('BRANCH',2,[type(extree,[X]),type(extree,[X])]),
        type(extree,[X]) ).

constructor_type( cterm('TUPLE',N,L), type(tuple,L) ) :- length(L,N).
constructor_type( cterm('OP',3,[type(simple,[sym]),type(simple,[sym]),
                type(simple,[num])]), type(op,[]) ).
```

```
%
% The following operator declarations define the basic syntax of FPL.
%
% (Note: Always define the longest operator first.)
%
postfix non   950  otherwise.
infix   non   950  when.
prefix  non   950  else.
prefix  non   100  load.
infix   left  850  and.
prefix  assoc 800  not.
infix   non   400  mod.
infix   non   400  rem.
prefix  non   100  vic.
prefix  non   100  vi.
infix   left  900  or.
infix   non   700  ">=".
infix   non   700  "<=".
infix   non   700  "==".
infix   non   950  "->".
infix   non   700  "<>".
infix   non   1000 "=".
infix   right 990  ";".
infix   non   700  "<".
infix   non   700  ">".
infix   non   600  "..".
infix   left  500  "+".
infix   left  500  "-".
infix   left  400  "*".
infix   left  400  "/".
infix   right 200  "^".
infix   right 100  "@".
prefix  non   50   "-".
postfix assoc 50   "!".


%
% f:x --> f:x,  the function application function
%
let ::f:x = f:x.


%
% f@x --> f:x,  the function application function
%
let @:f:x = f:x.



%
% true or false --> true, the disjunction function
%
let x or y = x -> true; y.
```



```
%
% true and true --> true, the conjunction function
%
let x and y = x -> y; false.


%
% not true --> false,   the negation function
%
let not x = x -> false; true.


%
% 1 <> 2 --> true,      the inequality function
%
let x <> y = not (x==y).


%
% 2 >= 1 --> true,      the greater than and equal to function
%
let x >= y = not (x<y).


%
% 2 <= 1 --> false,     the less than and equal to function
%
let x <= y = not (x>y).


%
% neg:9 --> -9,         the negative function
%
let neg:x = -1*x.


%
% signum:-5 --> -1,     the sign function
%
let signum:x =  -1      when x<0;
                0       when x==0;
                1       otherwise.


%
% cons:2:[] --> [2],    the list construction function
%
let cons:x:y = [x|y].
```

```
%
% abs:-9 --> 9,          the absolute-value function
%
let abs:x = x            when x>=0;
            -1*x         otherwise.


%
% round:9.4 --> 9,       round to the nearest integer
%
let round:x =   x                when integer?:x;
                trunc:(x+0.5)    when x>=0;
                trunc:(x-0.5)    otherwise.


%
% quotient:17:3 --> 5,  the integer division function
%
let quotient:x:y =
        integer?:x and integer?:y -> trunc:(x/y);
                error:"non integer argument!".


%
% frac_part:4.3 --> 0.3
%
let frac_part:x = integer?:x -> 0; x-trunc:x.


%
% floor:-9.2 --> -10,   the smallest integer not greater its argument
%
let floor:x =
       let tx = trunc:x
       in x     when frac_part:x==0;
          tx    when x>=0;
          tx-1  otherwise.


%
% ceiling:-9.2 --> -9,  the smallest integer greater its argument
%
let ceiling:x =
       let tx = trunc:x
       in x     when frac_part:x==0;
          tx+1  when x>=0;
          tx    otherwise.
```



```
%
% max:"a":"b" --> b,    the maximum function
%
let max:x:y = x>=y -> x ; y.


%
% min:"a":"b" --> a,    the minimum function
%
let min:x:y = x>y -> y ; x.


%
% null?:[] --> true,    the empty list test function
%
let null? = (==:[]).


%
% List concatenation function
%
% append:[1,2,3]:[4,5,6] --> [1,2,3,4,5,6]
%
let rec append:x:y =
        y                                  when x==[];
        (let [h|t]=x in [h|append:t:y]) otherwise.


%
% List reversal function
%
% reverse:[1,2,3,4,5] --> [5,4,3,2,1]
%
let reverse:x =
        let rec

            rev:l:a =   a                                  when l==[];
                        (let [h|t]=l in rev:t:[h|a])       otherwise

        in rev:x:[].


%
% The length function of lists
%
% length:[1,2,3,4,5] --> 5
%
let length:l =
        let rec
```

```
            len:x:a =   a                   when x==[];
                        len:(tl:x):(a+1)    otherwise

        in len:1:0.


%
% The membership test function
%
% member:3:[1,2,4,5] --> false
%
let rec member:x:l =
                false           when l==[];
                true            when x==hd:l;
                member:x:(tl:l) otherwise.


%
% map:f:[x1,...,xn] returns [f:x1,f:x2,...,f:xn].
%
% map:(+:1):[1,2,3,4] --> [2,3,4,5]
%
let rec map:f:l =
        []                              when l==[];
        [ f@hd@l | map:f:(tl:l) ]       otherwise.


%
% Nths:n:s returns the nth element of the stream s.
%
% Nths:4:<<1,2,3,4,5,6>> --> 4
%
let Nths:n:stream =
        error:"non integer argument"            when float?:n;
        error:"non positive range!"             when n<= 0;
        (let rec Nths1:m:s =
                head:s                  when m==1;
                Nths1:(m-1):(tail:s)    otherwise
         in Nths1:n:stream )                    otherwise.


%
% ints:n returns an infinite stream of integers starting from n.
%
% ints:1 --> <<1,2,3,4,5,...>>
%
let rec ints:n = <<n | ints:(n+1) >>.


%
% list_to_stream:l returns a stream with elements as in the list l.
```



```
%
% list_to_stream:[1,2,3] --> <<1,2,3>>
%
let rec list_to_stream:list =
        <<>>                                            when list==[];
        (let [h|t]=list in << h | list_to_stream:t >>)   otherwise.


%
% stream_to_list:s returns a list of elements as in the stream s; s must
%       be a finite stream.
%
% stream_to_list:<<1,2,3>> --> [1,2,3]
%
let rec stream_to_list:stream =
        []                                              when stream==<<>>;
        [head:stream | stream_to_list:(tail:stream)]    otherwise.


%
% This is the map function for the data type stream.
%
% maps:(+:1):<<1,2,3>> --> <<2,3,4>>
%
let rec maps:f:stream =
        <<>>                                            when stream==<<>>;
        << f:(head:stream) | maps:f:(tail:stream) >>    otherwise.


%
% Nth:n:l returns the nth element of the list l.
%
% Nth:4:[1,2,3,4,5,6] --> 4
%
let Nth:n:list =
      error:"non positive input!"       when n<= 0;
      error:"non integer argument!"     when float?:n;
      (let rec Nth1:m:list =
                      hd:list                   when m==1;
                      Nth1:(m-1):(tl:list)      otherwise
       in Nth1:n:list)                  otherwise.


%
% firstNs:n:s returns a list of the first n elements of the stream s.
%
% firstNs:4:(ints:1) --> [1,2,3,4]
%
let firstNs:n:stream =
        error:"negative argument!"              when n<0;
        error:"stream out of bound!"            when stream==<<>>;
        (let rec firstNs1:m:s =
```

```
            []                                  when m==0;
            [head:s | firstNs1:(m-1):(tail:s)]  otherwise
        in firstNs1:n:stream)                   otherwise.


%
% m..n returns a list [m,m+1,...,n].
%
% 1..5 --> [1,2,3,4,5]
%
let n..m =
       error:"non integer argument"    when float?:n or float?:m;
       error:"inputs out of bounds"    when m<n;
       (let rec f:n:m = [n]            when n==m;
                        [n|f:(n+1):m]  otherwise
        in f:n:m)                      otherwise.


%
% iota:n returns a list [1,2,...,n].
%
% iota:5 --> [1,2,3,4,5]
%
let iota = ..:1.


%
% mkstream:n returns a finite stream <<1,2,...,n>>.
%
% mkstream:5 --> <<1,2,3,4,5>>
%
let mkstream:n = list_to_stream@iota@n.


%
% The factorial function.
%
% 6! --> 720
%
let n! = error:"negative input!"        when n<0;
         error:"non integer argument"   when float?:n;
         (let rec

             fac:a:m =  a               when m==0;
                        fac:(a*m):(m-1) otherwise

         in fac:1:n)                    otherwise.


%
% pi --> 3.1415927
%
```



```
let pi = atan:1 * 4.


%
% rad:x returns the radian value of x degree.
%
% rad:180 --> 3.1415927
%
let rad:x = pi*x/180.


%
% deg:r returns the degree value of r radian.
%
% deg:1 --> 57.29578
%
let deg:x = 180*x/pi.


%
% vic:"file" --> "edit and load file"
%
let vic:file = load:(vi:file).
```

```
%
% The following programming examples are taken from the book
% by W.H.Burge, "Recursive Programming Techniques", page 22.


%
% the combinator W f x = f x x
%
let W:f:x = f:x:x.


%
% double x = x + x
%
let double = W:+.

double:2.


%
% square x = x * x
%
let square = W:*.

square:2.


%
% function composition
%
let B:f:g:x = f:(g:x).


%
% twice f = B f f
%
let twice:f = B:f:f.

twice:square:2.


%
% thrice f = B f (B f f)
%
let thrice:f = B:f:(B:f:f).

thrice:square:2.


%
% v:(x,y) = (x^2+y^2,x^2-y^2)
%
```



```
let v:l =
    let (x,y)=l
    in (x^2+y^2,x^2-y^2).


v@v@(3,2).
```

```
%
% Chebyshev polynomials can be defined recursively as follows:
%
%                 T (x) = 1
%                  0
%
%                 T (x) = x
%                  1
%
%                 T (x) = 2 * x * T   (x)  - T   (x)
%                  k               k-1        k-2
%
%   Define a function which computes the nth Chebyshev polynomial
%   and test it with the following inputs:
%
%                T (0.3) = 0.3448
%                 4
%
%                T (0.3) = -0.792
%                 3
%
%                T (0.3) = -0.82
%                 2
%

%
% The recursive definition can be written in the following form:
%
let rec T:k:x = 1                                 when k==0;
                x                                 when k==1;
                2*x*T:(k-1):x - T:(k-2):x         when k>1.


T:0:0.3.


T:1:0.3.


T:2:0.3.


T:3:0.3.


T:4:0.3.



%
% Using the program transformation technique ''combining recursive
% loops'', we can derive the following equivalent iterative defintion:
%
% T1(0) = \x.1
% T1(1) = \x.x
% T1(k) = \x.H(k,2*x,1,x),      k > 1
```



```
%
% where,
%       H(n,u,v,x) = u*x - v,                n=2,
%       H(n,u,v,x) = H(n-1,2*x*u-v,u,x), n>2.
%
let T1:k:x =    1                           when k==0;
                x                           when k==1;
                (let rec

                    H:n:u:v:x =
                        u*x - v                         when n==2;
                        H:(n-1):(2*x*u-v):u:x   when n>2

                in H:k:(2*x):1:x )      when k>1.


T1:0:0.3.


T1:1:0.3.


T1:2:0.3.


T1:3:0.3.


T1:4:0.3.
```

```
%
% create a new type tree of any type x
%
% type tree(x) = EMPTY | NODE(tree(x),x,tree(x)).

%
% binary tree insertion
%
let rec insert:e:t =
        NODE(EMPTY,e,EMPTY)                                  when t==EMPTY;
        (let NODE(left,key,right)=t
         in t                                 when k==e;
            NODE(insert:e:left,key,right) when k>e;
            NODE(left,key,insert:e:right) otherwise)    otherwise.

%
% list to binary tree conversion
%
let build:l =
        let rec
            buildtree:r:t =
                r==[] -> t ; buildtree:(tl:r):(insert:(hd:r):t)
        in buildtree:l:EMPTY.


build:[3,2,1,4,5].


build:["d","c","a","b","e"].


build:[true,false].


%type (x,y) = TUPLE(x,y).


(1,2).


("a","b").


(1,"a").


("a",1).


(true,false).


fst:(1,2).
```



```
snd:(1,2).


%type (x,y,z) = TUPLE(x,y,z).


let mid:u   = let (x,y,z)=u in y.


(1,2,3).


("a","b","c").


(1,"a",true).


mid:(1,"a",true).


%type (u,v,w,x) = TUPLE(u,v,w,x).


let q2:y = let (u,v,w,x)=y in v.


(1,2,3,4).


q2:(1,"a",true,3).


%type extree(x) = LEAF(x) | BRANCH(extree(x),extree(x)).


BRANCH(LEAF(1),LEAF(2)).


let transpose:((a,b),(c,d)) = ((a,c),(b,d)).


let interchange:[(a,b),(c,d)] = ([a,c],[b,d]).
```

```
%
% product:f:a:next:b computes the sum of the expression
%
%       f(a) * f(next(a)) * f(next(next(a)) * ... * f(b)
%
let rec product:f:a:next:b =
        a>b ->  1;
                f:a * product:f:(next:a):next:b.


%
% sum:f:a:next:b computes the sum of the expression
%
%       f(a) + f(next(a)) + f(next(next(a)) + ... + f(b)
%
let rec sum:f:a:next:b =
        a>b ->  0;
                f:a + sum:f:(next:a):next:b.


%
% integral:f:a:b:dx computes the integral of f between intervals a and b
%       with increment dx using trapezodial method.
%
let integral:f:a:b:dx =
        sum:f:(a+dx/2):(\x.x+dx):b * dx.


integral:(\x.x*x):0:1:0.05.



%
% improve:x:f:p computes the value of Xn
%
%       X1, X2,  X3,     ...,    Xn, ...
%
% where X1=x, X2=f(x), X3=f(f(x)), ..., such that  p:Xn:Xn-1 is true.
%
let rec improve:x:f:close_enough? =
        close_enough?:x:(f:x)
                ->      x;
                        improve:(f:x):f:close_enough?.


%
% sq_root:a:eps computes the square root of a using Newton's method
%       with accuracy within eps.
%
%       Xi+1 = (Xi+a/Xi)/2
%
let sq_root:x:eps = improve:1:(\y.(y+x/y)/2):(\x,y.abs:(x-y)<eps).


sq_root:16:1e-3.
```



```
%
% horner:x:n:a computes the polynomial on x of n terms with coefficient
%       a:m using horner's rule, where m=0,1,2,...,n.
%
let horner:x:n:a =
        let rec

            poly_eval:x:m:n:a =
                m>n -> 0;
                else   a:m + x*poly_eval:x:(m+1):n:a

        in poly_eval:x:0:n:a.


%
% sin2:x:n computes the sine of x using n terms of its Taylor expansion
%
let sin2:x:n =
        let

            coeff_sin:x =
                let remx=x rem 4, factx = x!
                in  1/factx     when remx==1;
                    -1/factx    when remx==3;
                    0           otherwise

        in horner:x:n:coeff_sin.


sin2:(rad:30):8.


%
% cos2:x:n computes the cosine of x using n terms of its Taylor expansion
%
let cos2:x:n =
        let
            coeff_cos:x =
                let remx=x rem 4, factx = x!
                in  1/factx     when remx==0;
                    -1/factx    when remx==0;
                    0           otherwise

        in horner:x:n:coeff_cos.


cos2:(rad:30):8.


%
% e:x:n computes the e of x using n terms of its Taylor expansion
%
%       e(x) = SIGMA(i=1..n) 1/i!
%
```

```
let e:x:n = horner:x:n:(\y.1/y!).

e:1:8.


let pi_factor:x = (2*x*(2+2*x))/((2*x+1)*(2*x+1)).

let pi_prod:a:b = product:(\x.(2*x*(2+2*x))/((2*x+1)*(2*x+1))):a:(+:1):b.

pi_prod:1:20.

%
% pi_sum(a,b) approximates pi given initial value a and used b number of
%       terms
%
let pi_sum:a:b = sum:(\x.1/(x*(x+2))):a:(\x.x+4):b.

8*pi_sum:1:50.
```



```
%
% union:a:x computes the union of two lists a and x.
%
let rec union:a:x =
        a                                       when x==[];
        (let z=union:a:(tl:x), hx=hd:x
         in member:hx:z -> z;
                           [hx|z])      otherwise.


%
% difference:a:x computes the difference of two lists, a-x.
%
let rec difference:x:y =
        []                                              when x==[];
        (let z=difference:(tl:x):y, hx=hd:x
        in member:hx:y -> z;
                           [hx|z])              otherwise.


union:[1,2,3]:[3,4,5].


difference:[1,2,3]:[3,4,5].


let set = union:[].


set:[1,2,2,3,3,4,5].


%
% reduce:g:a:x computes the expression
%
%       g(x1,g(x2,g(...,g(xn,a)...)
%
% giving that x is a list [x1,x2,...,xn].
%
let rec reduce:g:a:x =
        a                                               when x==[];
        (let [h|t]=x in g:h:(reduce:g:a:t))     otherwise.


%
% sum:[1,2,3] -> 1+2+3 -> 6.
%
let sum = reduce:+:0.


sum:(1..4).


%
% prod:[1,2,3] -> 1*2*3 -> 6.
%
```

```
let prod = reduce:*:1.


prod:(2..5).


reduce:cons:[]:(1..3).


%
% listify:[1,2,3] -> [[1],[2],[3]]
%
let listify = map:(\x.[x]).


listify:[1,2,3].


%
% xpl:[1,2,3,4] -> [[1],[1,2],[1,2,3],[1,2,3,4]]
%
let xpl:1 =

    let ll = listify:1
    in let rec

          xpl1:x:y =
                [x]                                          when y==[];
                [x | xpl1:(append:x:(hd:y)):(tl:y)]           otherwise

       in xpl1:(hd:ll):(tl:ll).


xpl:[1,2,3,4].


%
% funlist:1:3 -> [[1],[1,2],[1,2,3]]
%
let rec funlist:m:n =
            []                                               when m>n;
            [ [m] | map:(\x.[m|x]):(funlist:(m+1):n) ]       otherwise.


funlist:1:3.


let funlist1:m:n = xpl:(m..n).


funlist1:1:3.
```



```
%
% 6! -> 720
%
let rec n! = 1          when n==0;
             n*(n-1)!   when n>0.


6!.


%
% compute the fibonacci number using iterative calls
%
let fib2:n =
        n<0 -> error:"input must be positive!";
        let rec
            loop:a:b:cnt = cnt==0 -> a; loop:b:(a+b):(cnt-1)
        in loop:1:1:n.

fib2:10.



%
% compute the fibonacci number using recursive calls
%
let rec fib:n = error:"negative input!"          when n<=0;
                1                                when n<=2;
                fib:(n-1) + fib:(n-1)            when n>2.


fib:5.



%
% gcd:m:n computes the gcd of m and n
%
let rec gcd:m:n = n              when m==0;
                  m              when n==0;
                  gcd:m:(n-m)    when n>=m;
                  gcd:(m-n):n    when n<m.


gcd:49:35.


prefix assoc 50 "++".


%
% the increment function
%
let ++ = +:1.
```

```
++ ++ 3.

prefix assoc 50 "--".

%
% the decrement functioin
%
let --:x = x-1.

-- -- 3.


%
% pow:x:i:p computes the ith power of x using product function p.
%
let rec pow:x:i:p = i==1 -> x; p:x:(pow:x:(i-1):p).


pow:3:2:*.


pow:[1,2,3]:3:append.


%
% divide:m:n -> (quotient,remainder) where m,n >= 0.
%
let rec divide:m:n =
        m < n -> (0,m);
                 let (f,s) = divide:(m-n):n
                 in (f+1,s).

divide:7:2.

let divides:x:y = snd:(divide:x:y)==0.

divides:8:3.



%
% sumprod:[1,2,3] -> (6,6)
%
let sumprod:x =
        let rec
            sp:x:s:p =
                x==[] -> (s,p) ;
                         let [h|t]=x in sp:t:(s+h):(p*h)
        in sp:x:0:1.


sumprod:(1..3).
```



```
let inclist = map:(+:1).


inclist:(1..10).



%
% twolist:1:2 -> [1,2]
%
let twolist:x:y = [x,y].


twolist:1:2.


%
% zip:[1,2,3]:[4,5,6] -> [[1,4],[2,5],[3,6]]
%
let rec zip:x:y =
        x==[] -> [];
        y==[] -> [];
                 let [hx|tx]=x, [hy|ty]=y
                 in [twolist:hx:hy | zip:tx:ty].


zip:[1,2,3]:[4,5,6].


%
% unzip:[[1,4],[2,5],[3,6]] -> [[1,2,3],[4,5,6]]
%
let unzip:x = twolist:(map:(\y.hd:y):x):(map:(\y.hd@tl@y):x).


unzip:[[1,4],[2,5],[3,6]].



%
% The Ackermann function
%
let rec A:x:y = x==0 -> y+1;
                y==0 -> A:(x-1):1;
                        A:(x-1):(A:x:(y-1)).


A:2:2.


A:2:3.


%
% currying
%
```

```
let curry:f = \x,y.f:(x,y).


let add:u = let (x,y)=u in x+y.


curry:add:1:2.


let marks = [77,83,65,86,72,50,93,65,75,88,56,90,91,67,67,58,57,
             86,75,87,84,78,79,81,89,87,85,70,96,72].


let len = length:marks.


let rec sumlist:l =
        0                       when l==[];
        hd:l + sumlist:(tl:l)   otherwise.


let sum = sumlist:marks.


let avg = sum/len.
```



```
let rec stream_accumulator:f:accumulation:stream =
    stream==<<>> -> stream;
        << accumulation |
           stream_accumulator:f:(f:accumulation:(head:stream)):(tail:stream)
        >>.


firstNs:5:(stream_accumulator:*:1:<<2,3,4,5,6,7,8>>).


let rec integers:m:next = << m | integers:(next:m):next >>.


firstNs:5:(integers:1:(+:1)).


firstNs:5:(stream_accumulator:*:1:(maps:(\y.1/y):(integers:1:(+:1)))).


let rec init_segment:pred:stream =
        let h = head:stream
        in pred:h -> <<>>;
                << h | init_segment:pred:(tail:stream) >>.


let rec accumulator:f:neutral:stream =
        stream==<<>> -> neutral;
            f:(head:stream):(accumulator:f:neutral:(tail:stream)).


accumulator:+:0:<<1,2,3,4,5,6,7>>.


let exp_approx:x:eps =
        accumulator:+:0:
            (init_segment:(\x.x<eps):
                (stream_accumulator:*:1:
                    (maps:(\y.x*y):(maps:(\y.1/y):(integers:1:(+:1)))
            )   )   ).


exp_approx:1:1e-5.


let rec add_stream:x:y =
         x==<<>> -> y;
         y==<<>> -> x;
                << head:x+head:y | add_stream:(tail:x):(tail:y) >>.


let rec fibs = <<1, 1 | add_stream:fibs:(tail:fibs) >>.


firstNs:8:fibs.
```

```
let rec reduces:f:neutral:stream =
        stream==<<>> -> neutral;
              f:(head:stream):(reduces:f:neutral:(tail:stream)).


let sum_stream:stream = reduces:+:0:stream.



sum_stream:<<1,2,3,4,5>>.


%
% power:next:m computes an infinite stream of
%
%       m, next(m), next(next(m)), ...
%
let power:next:m =
        let rec stream = << m | maps:next:stream >>
        in stream.


Nths:4:(power:(\x.2*x):1).
```



```
%
% merge:s1:s2 merges the two streams s1 and s2 in increasing order
%
let rec merge:s1:s2 =
        s2      when s1==<<>>;
        s1      when s2==<<>>;
        else
           (let h1=head:s1, h2=head:s2
            in  << h1 | merge:(tail:s1):s2 >>           when h1<h2;
                << h2 | merge:s1:(tail:s2) >>           when h2<h1;
                << h1 | merge:(tail:s1):(tail:s2) >>    otherwise).


let rec scales:factor:stream =
        << factor*(head:stream) | scales:factor:(tail:stream) >>.


%
% hamming --> an infinite sequence of hamming numbers.
%
let rec hamming =
        << 1 |  merge:(scales:2:hamming):
                (merge:(scales:3:hamming):(scales:5:hamming)) >>.


firstNs:5:hamming.
```

```
let rec pairer:f:neutral:stream =
    stream==<<>> -> <<>>;
        let <<first | rest>> = stream
        in rest==<<>> -> <<first | rest>>;
                <<f:first:(head:rest) | pairer:f:neutral:(tail:rest) >>.


let rec balanced_acc:f:neutral:stream =
    stream==<<>> -> neutral;
        let <<first | rest>> = stream
        in rest==<<>> -> first;
                balanced_acc:f:neutral:(pairer:f:neutral:stream).

let merge_sort:stream =
    let rec
         merge:l1:l2 =
              l1==[] -> l2;
              l2==[] -> l1;
                        let [h1|t1]=l1, [h2|t2]=l2
                        in h1<h2 -> [h1 | merge:t1:l2];
                                    [h2 | merge:l1:t2]
    in balanced_acc:merge:[]:(maps:(\x.[x]):stream).


let s = <<5,8,2,4,1>>.

balanced_acc:+:0:s.

merge_sort:s.
```



```
let rec F:f:a:next:b =  0                                   when a>b;
                        f:a + F:f:(next:a):next:b           otherwise.


F:(\x.x*x):1:(+:1):5.


let rec G:f:a:next:b =  1                                   when a>b;
                        f:a * G:f:(next:a):next:b           otherwise.

G:(+:1):1:(+:1):5.


infix left 400 "**".


let rec H:**:f:a:next:b:neutral
        =       neutral                                          when a>b;
                (f:a) ** (H:**:f:(next:a):next:b:neutral) otherwise.


H:+:(\x.x*x):1:(+:1):5:0.
```

```
let sq:x = x*x.


infix right 600 o.

%
% f o g -> the composition of f and g
%
let f o g = \x.f@g@x.


(sq o sq):3.


infix left 400 "**".


%
% f ** n -> f raises to the nth power
%
let rec  f ** n =
        (\x.x)          when n==0;
        f               when n==1;
        f o f**(n-1)    when n>1.


(sq ** 2):3.


(sq ** 4):3.


let twice:f = f o f.


twice:sq:2.


let fourtimes = twice:twice.


fourtimes:sq:2.
```



```
%
% x = pair:(first:x):(second:x)  if x is a pair
%
let pair:u:v = \p.p:u:v.


pair:2:3.


let first:p = p:(\u,v.u).


let second:p = p:(\u,v.v).


first:(pair:2:3).


second:(pair:2:3).
```

```
let rec reduce:f:l:n =
        n                                   when l==[];
        (let [h|t]=l
         in  f:h:(reduce:f:t:n))            otherwise.


reduce:+:(1..5):0.


let fac:n = reduce:*:(1..n):1.


%
% choose:p:[x1,...,xn] --> [ xi | p:xi is true ]
%
let rec choose:p:l =
        []                                  when l==[];
        (let [h|t]=l
         in let rest=choose:p:t
            in  p:h -> [h|rest]; rest)  otherwise.


choose:(\x.(x mod 2)==0):(1..10).


%
% connects:src:dest:graph --> true if there is a path from src to dest in
%                             graph, false otherwise.
%
let rec connects:src:dest:graph =
        false                                       when graph==[];
        true                                        when member:(src,dest):graph;
        (let reachable=choose:(\x.fst:x==src):graph
         in  let alternative= map:snd:reachable,
                 fromhere = \x.connects:x:dest:graph
             in
               choose:(\x.x):(map:fromhere:alternative) <> []) otherwise.

let graph = [(1,2),(2,4),(1,3),(3,5),(4,6),(6,8),(7,8)].


connects:1:8:graph.
```



```
%
% pairs:[1,2,3] --> [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
%
let rec pairs:l =
        []                                                  when l==[];
        append:(map:(\y.(hd:l,y)):l):(pairs:(tl:l)) otherwise.
```

```
let inc = +:1,
    twice:f:x = f@f@x,
    thrice:f:x = f@f@f@x
in ( twice:thrice:inc:0, thrice:twice:inc:0 ).


let
    twice:f:x = f@f@x,
    thrice:f:x = f@f@f@x
in ( twice:thrice, thrice:twice ).


(snd:it):(+:1):0.
```



```
%
% Representing (Church) numerals in pure lambda calculus
%

%
% the constant zero in lambda-term
%
let zero:x:y = y.

%
% succ:x --> the successor of x.
%
let succ:x = \y.\z.y:(x:y:z).

%
% plus:x:y --> x + y.
%
let plus:x:y = x:succ:y.

%
% times:x:y --> x * y
%
% let times:x:y = x:(plus:y):zero.
%
infix right 500 o.

let (f o g):x = f:(g:x).

let times:x:y = x o y.

%
% expt:x:y --> x to the power y.
%
let expt:x:y = y:x.


%
% testing the above lambda terms
%
let one = succ:zero.

let two = succ:one.

let three = succ:two.

%
% The numeral n in pure lambda term applies to a function f
% producing a function n times of f.
%
% Hence, n:(+:1) --> a function of increment by n
%
plus:two:three:(+:1):0.          % = 5.

plus:three:two:(+:1):0.          % = 5.

times:two:three:(+:1):0.         % = 6.
```

```
times:three:two:(+:1):0.        % = 6.

expt:two:three:(+:1):0.         % = 8.

expt:three:two:(+:1):0.         % = 9.

%
% conversion between numbers and lambda numerals
%

%
% H:5 --> a lambda term representing the number 5.
%
let rec H:n =
        zero            when n==0;
        succ:(H:(n-1))  otherwise.


%
% H1:two --> 2.
%
let H1:num = num:(+:1):0.


%
% pred:num --> the predecessor of num, where num is lambda numeral.
%
let pred = H o (\x.x-1) o H1.


%
% isZero:num --> true if num is zero, false otherwise; num must be a
%       lambda numeral.
%
let isZero = (==:0) o H1.


H:3:(+:1):0.


H1:three.


(pred:three):(+:1):0.


isZero:one.
```



```
%
% This file contains test inputs for the predefined functions.
%

::(+:1):1.


@:(+:1):1.


true or false.
true or true.
false or false.
false or true.


true and true.
true and false.
false and false.
false and true.


not true.
not false.


1<>2.
1<>1.
"a" <> "a".
"a" <> "b".
true <> false.
true <> true.


2 >= 1.
2 >= 3.
"a" >= "a".
"a" >= "b".
true >= true.
true >= false.


2 <= 1.
2 <= 3.
"a" <= "a".
"a" <= "b".
true <= true.
true <= false.
```

```
neg:9.
neg:-9.


signum:5.
signum:0.
signum:-5.


cons:2:[].


abs:-9.
abs:9.


round:9.4.
round:9.6.


quotient:17:3.


frac_part:4.3.
frac_part:4.


floor:-9.2.
floor:9.2.


ceiling:-9.2.
ceiling:9.2.


max:"a":"b".
max:1:2.
max:true:false.


min:"a":"b".
min:1:2.
min:true:false.
```



```
null?:[].
null?:[1,2].


1..5.


append:(1..9):(10..20).


reverse:(1..10).


length:(1..10).


member:3:[1,2,4,5].
member:3:[1,2,3,4,5].
member:"c":["a","b","d","e"].
member:true:[false,false].


map:(+:1):(1..10).


mkstream:6.


Nths:4:it.


ints:1.


list_to_stream:(1..5).


stream_to_list@mkstream@5.


maps:(+:1):(mkstream:5).
```

```
Nth:4:(1..6).

firstNs:4:(ints:1).

iota:5.

6!.

pi.

rad:180.

deg:1.
```



```
%
% S-K-I combinators
%
let S:x:y:z = x:z:(y:z).

let K:x:y = x.

%
% let I:x = x.
%
let I = S:K:K.

%
% let B:x:y:z = x:(y:z).
%
let B = S:(K:S):K.

%
% let C:x:y:z = x:z:y.
%
let C = S:(B:B:S):(K:K).

%
% let I1:x:y = y:x.
%
let I1 = C:I.

%
% let B1:x:y:z = y:(x:z).
%
let B1 = C:B.

%
% let K1:x:y = y.
%
let K1 = K:I.

%
% let K11:x:y:z = x:y.
%
```

```
let K11 = B:K.


%
% let W:x:y = x:y:y.
%
let W = S:K:(K:I).


%
% numerals in terms of combinators
%
let Z0 = K:I
let Z1 = (S:B):Z0.
let Z2 = (S:B):Z1.
let Z3 = (S:B):Z2.
```



```
%
% The following example is taken from Z.Manna, "Mathematical Theory of
% Computation", McGraw-Hill, 1974, (pages 356-7).
%
% It shows that f1, f2 and f3 are fixpoints of P.
% f3 is the least (defined) fixpoint of P.
%
%
% P: F(x,y) <= if x=y then y+1
%                     else F(x,F(x-1,y+1))
%

let P = \f,x,y.x==y -> y+1; f:x:(f:(x-1):(y+1)).


let f1 = \x,y.x==y -> y+1; x+1.


let f2 = \x,y.x>=y -> x+1; y-1.


let even?:x = (x mod 2)==0.

let f3 = \x,y.x>=y and even?:(x-y) -> x+1; error:"undefined".


%
% substituting fi (i=1,2,3) into P
%
let F1 = P:f1.


let F2 = P:f2.


let F3 = P:f3.


%
% They should have the same values for the pair (5,1).
%
F3:5:1.

F2:5:1.

F1:5:1.

%
% Some arguments may be undefined in f3, but defined in f1 or f2.
```

```
%
F3:4:1.

F2:4:1.

F1:4:1.
```