A Prological Definition of HASL a Purely Functional Language with Unification Based Conditional Binding Expressions

Harvey Abramson

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada

ABSTRACT

We present a definition in Prolog of a new purely functional (applicative) language HASL (*H.* *A*bramson's *S*tatic *L*anguage). HASL is a descendant of Turner's SASL and differs from the latter in several significant points: it includes Abramson's unification based conditional binding constructs; it restricts each clause in a definition of a HASL function to have the same arity, thereby complicating somewhat the compilation of clauses to combinators, but simplifying considerably the HASL reduction machine; and it includes the single element domain {fail} as a component of the domain of HASL data structures. It is intended to use HASL to express the functional dependencies in a translator writing system based on denotational semantics, and to study the feasibility of using HASL as a functional sublanguage of Prolog or some other logic programming language. Regarding this latter application we suggest that since a reduction mechanism exists for HASL, it may be easier to combine it with a logic programming language than it was for Robinson and Siebert to combine LISP and LOGIC into LOGLISP: in that case a fairly complex mechanism had to be invented to reduce uninterpreted LOGIC terms to LISP values.

The definition is divided into four parts. The first part defines the lexical structure of the language by means of a simple Definite Clause Grammar which relates character strings to "token" strings. The second part defines the syntactic structure of the language by means of a more complex Definite Clause Grammar and relates token strings to a parse tree. The third part is semantic in nature and translates the parse tree definitions and expressions to a variable-free string of combinators and global names. The fourth part of the definition consists of a set of Prolog predicates which specifies how strings of combinators and global names are reduced to "values", ie., integers, truth values, characters, lists, functions, fail, and has an operational flavour: one can think of this fourth part as the definition of a normal order reduction machine.

April 24, 1983

A Prological Definition of HASL a Purely Functional Language with Unification Based Conditional Binding Expressions

Harvey Abramson

Department of Computer Science
University of British Columbia
Vancouver, B.C. Canada

1. Introduction

In this paper we shall use Definite Clause Grammars (DCGs) and Prolog to present a definition of HASL, a purely functional language incorporating the unification based conditional expressions introduced in [Abramson,82a].

Metamorphosis grammars were introduced in [Colmerauer,78] and were shown to be effective in the writing of a compiler for a simple programming language. Definite Clause Grammars, a special case of metamorphosis grammars were introduced in [Pereira&Warren,80] and shown to be effective in "compiling", ie, translating a subset of natural language into first order logic. Metamorphosis grammars (M-grammars) have been used to describe several languages, namely ASPLE, Prolog, and a substantial subset of Algol-68 [Moss,81], [Moss,79]; see also [Moss,82] for the use of Prolog and grammars as tools in language definition. Although neither M-grammars nor DCGs were mentioned in [Warren,77], that paper is of interest in the use of Prolog as a compiler writing tool. The use of DCGs and Prolog for the implementation of SASL [Turner,76,79,81], a purely applicative language, was reported in [Abramson,82b].

The language HASL which we shall define below arose out of the Prolog implementation of SASL. One reason for defining the new language was to incorporate unification based conditional binding expressions; another was to simplify and clean up the combinator reduction machine introduced by Turner to evaluate SASL expressions; a third was to provide a possible functional sublanguage for Prolog; and a fourth was to provide a functional notation for a denotational semantics based translator writing system akin to Mosses' Semantics Implementation System [Mosses,79] but to be tied to DCGs. Although we do not present a purely logical definition of HASL, we feel that the departures from Horn clause logic in the definition presented below (the use of the cut for control; negation as failure; and extension of HASL's database of globally defined functions) are not significant enough to mar the formality of the definition or its comprehensibility. The definition can be used as a specification of HASL, as an interpretive implementation of HASL, and as a guide to a more efficient implementation of HASL in some system programming language.

In section 2 we shall informally and briefly describe HASL. Section 3 contains a description of the general definition strategy: HASL expressions are compiled to variable-free strings of combinators, global names, and uses of the two primitive operations of function application (—>) and pair construction(:, like LISP's CONS). Following this are sections devoted to: the DCG for lexical analysis; the DCG and associated predicates which perform syntactic analysis and parse tree formation; the translation to combinators; and the HASL reduction machine. A final section will suggest some further work which we intend to pursue.

2. HASL - Informally and Briefly

HASL is descended from Turner's SASL (see [Turner,76,79,81]) and obviously owes much to it. We have chosen to designate this language HASL not to suggest that what we present is totally original, but that there are enough departures from SASL to warrant a new designation.

A HASL program is an expression such as

```
[1,2,3] + + [4,5,6]
```

with value

```
[1,2,3,4,5,6]
```

or an expression with a list of equational definitions qualifying the expression:

```
f x
where
x= hd y
hd (a:x) = a,
y = 3:y,
f 0 = 1,
f x = x * f(x-1)
```

with value 6.

We note in this list of definitions that

[1] A function such as *f* may be defined by a list of clauses. The order of the clauses is important: in applying *f* to an argument the first clause will be "tried", then the second, etc.

[2] In the definition of *hd* the argument must be a constructed pair, specified by *(a:x)* where : is the HASL pair constructor. Structure specifications may involve arbitrary list structures of identifiers and constants.

[3] HASL makes use of lazy evaluation ([Henderson & Morris,76]) so that infinite lists such as *y* may be defined, and elements of such lists may be accessed, as in *hd y* without running into difficulties.

A list may be written as

```
[1,2,3]
```

which is syntactic sugaring for

```
1: 2: 3: []
```

where [] denotes the empty list and the notation 'string' is a sugaring for the list of character denotations:

```
%s: %t: %r: %i: %n: %g: []
```

There are functions such as *number, logical, char* and *function* which may be used to check the types of HASL data objects:

```
number 12 = true
logical 5 = false
char %%   = true
function hd = true
```

are all HASL expressions which have the value true.

Functions may be added to HASL's global environment as follows:

```
def
string [] = true,
string (a:x) = char a & string x,
string x = false,
cons a b = a:b
;
```

Each clause defining a HASL function *f* must have the same number of arguments or arity. Thus above, each clause in the definition of *string* has arity one. In the second clause for *string* however, a single structured argument is designated. Although HASL functions may be written as if they had several arguments, such as *cons* above, HASL functions are all considered to have in fact single arguments. The single argument is a HASL data object which may be a character, a truth value, an integer, fail, a list of HASL objects, or a function of HASL objects to HASL objects. The value of such a function may be any HASL object - including a function. Thus the value of

```
cons %a
```

is the HASL function which puts *%a* in front of lists.

The HASL object *fail* is the result of, for example, applying *hd* to a number:

```
hd 5 = fail
```

The object *fail* is not a SASL object and is one of our departures from that language.

Another departure is in the introduction of the restricted unification based conditional binding constructs {- and -} of [Abramson,1982].

```
formals {- exp1 => exp2 ; exp3
```

The meaning of this is that if *exp1* can be unified to the list of *formals,* then the value of this expression is the value of *exp2* qualified by the bindings induced by the match; otherwise, it is the value of *exp3*. This may be expressed somewhat inefficiently using the HASL conditional expression *(a -> b ; c)*:

```
(fail = f exp1
 where f formals = exp2) ->
exp3 ;
(f exp1
 where f formals = exp2)
```

Thus the unification expression may be regarded as the definition and application of an anonymous function.

The unification expression is in fact the basis of the compilation of HASL clausal definitions into a single function. If *member* is defined by the following clauses:

```
def
member a [] = false,
member a (a:x) = true,
member a (b:x) = member a x
;
```

then the HASL specification and interpreter treats this as:

```
member x1 x2 =
a [] {- x1 x2 => false;
a (a:x) {- x1 x2 => true;
a (b:x) {- x1 x2 => member a x;
fail
```

3. The Top Level of the HASL Specification.

A HASL expression denotes a value. We may express this by the notation

```
hasl(Expression,Value).
```

This relation requires some refinement, however. The expression is written as some sequence of characters, including spaces, carriage returns, etc., and the characters must be grouped into a sequence of meaningful HASL "tokens". These tokens must then be grouped into meaningful syntactic units determined by the syntax of HASL expressions. These two relations, the lexical and syntactic, are expressed by means of two DCGs: one DCG defines the relation between a sequence of characters and a sequence of HASL tokens; a second DCG defines the relation between a sequence of HASL tokens and a representation of the syntactic structure of a HASL expression as a tree.

Further, the expression of the relation between the tree and the value denoted by the original sequence of characters requires refinement. The tree represents the abstract syntax of the HASL expression. A semantic relation holds between this tree and a sequence of combinators, global names, function application operators (-->) and pair construction operators (:). This relation therefore defines a translation from a syntactically sweet string of symbols (HASL) to a mathematically equivalent - but rather unreadable - sequence of symbols suitable for mechanical evaluation or reduction. The reduction relation (=>>) specifies how such a sequence of symbols is related to another sequence of symbols which is the head normal form of the first. A final relation (>>>) between head normal form and HASL values (normal form) completes the specification of the relation *hasl:*

```
hasl(Expression,Value) :-
lexical(Expression,Tokens),
syntactic(Tokens,Tree),
semantic(Tree,Combinators),
Combinators =>> HeadNormal,
HeadNormal >>> Value.
```

The lexical relation may be specified in terms of a relation *lexemes* (see next section):

```
lexical(Expression,Tokens) :- lexemes(Tokens,Expression,[]).
```

and the syntactic relation may be specified in terms of a relation *expression* (see Section 5):

```
syntactic(Tokens,Tree) :- expression(Tree,Tokens,[]).
```

The two relations *lexemes* and *expression* are defined below by definite clause grammars.

4. The Lexical Specification of HASL.

This relation requires little comment. A sequence of characters such as

```
"def fac 0 = 1, fac x = x * fac(x-1);"
```

is grouped into the following string of tokens:

```
[def,id(fac),constant(num(0)),op(3,cEQ),constant(num(1)),comma,id(fac),id(x),
op(3,cEQ),id(x),op(5,cMULT),id(fac),lparen,id(x),op(4,cSUB),constant(num(1)),
rparen,semicolon]
```

Identifiers such as *fac* are represented by *id(fac),* constants such as 0 are represented by *constant(num(0)).* Some reserved words and punctuation are represented by atoms such as *def* and *comma.*

A sequence of definite clause rules such as

```
tIDENT(id(Id)) --> [id(Id)].
```

defines the function symbols which are the terminals for syntactic analysis.

The complete Prolog specification of HASL is at the end of this report following the References.

5. The Syntactic Specification of HASL.

As mentioned above, the *syntactic* relation is between token strings and parse trees which represent the abstract syntax of HASL expressions.

The leaves of a parse tree may be identifiers such as *id(fac),* constants such as *logical(true), num(123), char(C), fail,* or they may be the names of certain known HASL combinators such as *cADD* for addition, or *cMATCH* used in unification, etc. These names follow the convention of a lower case *c* followed by some other letters (usually upper case), digits or underline characters.

There are several kinds of branch nodes. A branch may be labeled by the HASL function application arrow (—>) or by the HASL pair construction colon (:). The arrow associates to the left, the colon to the right. Thus the linear parse tree representation of *a+ 1* is:

```
cADD —> id(a) —> num(1)
```

and that for *hd 'abc'* is:

```
cHD —> %a : %b : %c : []
```

Another kind of branch node is labeled with the functor *where* and has one subtree which is an expression and another which is the subtree for a list of definitions qualifying the expression:

```
where(Exp,Defs)
```

Global definitions are subtrees of a tree where the root is labeled with the functor *global*

```
global(Defs)
```

To each definition there is a branch node labeled with the functor *def* and with three subtrees: the name of the identifier being defined; the arity associated with the name being defined; and, the expression or list of clauses to be associated with the name. For a name with arity 0 such as in:

```
def b = a + 1;
```

the definition node looks like:

```
def(id(b),0,cADD-->id(a)-->num(1))
```

When a function is being defined, the arity is at least one, and the third argument is a list of clauses, each of the form:

```
func(Fseq,Exp)
```

where *Fseq* is a list of arguments of length *arity* for the function being defined, and *Exp* is the expression associated with that clause. Thus, the definition of *member* in Section 2 is represented in a parse tree as:

```
def(id(member),2,
[func([id(a)|flist(id(b):id(x))],id(member)-->id(a)-->id(x)),
func([id(a)|flist(id(a):id(x))],logical(true))|
func([id(a)|const(nil)],logical(false))|])])
```

The functor *flist* is used to label a branch of a tree in which a list structured argument to a function is specified. The context sensitive restriction that each clause defining a function have the same arity is specified by the predicate *mergedef* which merges separate clauses for a function into one node of the above description. (See the next two sections for further discussion of this restriction.)

One other point to note is that a list of definitions of arity 0 such as

```
[x,y,z] = x
```

is represented as a list of definition nodes:

```
[def(id(x),0,cHD-->id(x)),
def(id(y),0,cHD-->(cTL-->id(x))]
def(id(z),0,cTL-->(cTL-->id(x))]
```

This is specified by the predicate *expandef.*

The last remaining kind of branch node is that for a unification based conditional binding expression.

```
(a:x){-y=>x;fail
y-}(a:x)=>x;fail
```

would both be represented as:

```
unify(flist(id(a):id(x)),id(y),id(x),fail)
```

The DCG specifying the syntax of HASL is fairly straightforward. There is some slight intricacy in the specification of the grammar rules for expressions involving the HASL operators: operator precedence techniques are used to build the appropriate subtrees.

The function symbols beginning with a lower case *t* are the terminals for this grammar and specify HASL tokens as defined by the lexical DCG.

6. The Semantic Specification of HASL.

The *semantic* relation is one which holds between parse trees as specified in the previous section, and certain strings of combinators, constants, global names, and the primitive HASL operations of function application (-->) and pair construction (:). These strings may in fact be regarded as modified parse trees in which the *global, where, def, func, flist* and *unify* nodes have been eliminated and replaced by variable-free subtrees. The elimination of these nodes depends on a discovery of the logician Schoenfinkel: that variables, although convenient, are not necessary.

Schoenfinkel's discovery that variables can be dispensed with relies on a sort of cancellation related to extensionality. If in HASL we defined

```
successor x = plus 1 x
plus a b = a + b
```

then we could say that

```
successor = plus 1
```

for both sides, when applied to the same argument, are always equal.

Schoenfinkel related a variable, an expression which may contain that variable, and an expression from which that variable had been abstracted (removed) with the aid of the following combinators:

```
cS x y z = x z (y z)
cK x y = x
cI x = x
```

The specification of the abstraction or removal of a variable is given by the predicate *abstr0:*

```
abstr0(V,X--->Y,cS--->AX--->AY) :- ! ,
      abstr0(V,X,AX) ,
      abstr0(V,Y,AY).
abstr0(V,V,cI) :- !.
abstr0(V,X,cK--->X).
```

V is a variable, *X* is an expression, and the third argument of *abstr0* is the expression with variable removed. So in the following:

```
abstr0(id(x),plus--->num(1)--->id(x),X).
```

we have

```
X = cS--->(cS--->(cK--->plus)--->(cK--->num(1)))--->cI
```

with no variables, and only the constants *plus* and *num(1)*, the combinators and --->.

When the resulting expression is applied to an actual argument, these combinators, speaking anthropomorphically, place the actual arguments in the right places so that the evaluated result is the same as would be given (by extensionality) by evaluating the original expression with variables and by making the appropriate substitutions of actual arguments for variables. The advantage of not using variables, of course, is that an environment is not necessary and that no substitution algorithm is necessary.

It is clear, however, that this abstraction specification albeit elegant, leads to expressions much longer than the original. It is possible, however, to control the size of the resulting expression by introducing combinators which are "optimizing" in the sense that if a variable which is being abstracted is not used in the original expression, then the resulting expression will not have any redundancies. Some of these optimizing combinators are described in [Burge,1975]; a more effective set was introduced by Turner who also extended the notion of abstraction of variables to a context in which there was a primitive operation of pair construction in addition to the primitive operation of function application.

The predicate for abstraction in the specification of HASL's semantics is based on Turner's technique: *abstract* specifies how a list of variables is to be removed; *abstr* specifies how a single variable is removed; and *combine* specifies the optimizations which control the size of the resulting expressions.The first argument to *abstract* is a list uncurrying combinator which splits a structure into its components, and is an aspect of HASL's (restricted) unification. If a formal argument on the left hand side of a clausal definition is being "opened up", the combinator (cU_s) is strict: if the actual argument does not have the appropriate list structure then the value *fail* must result; in other cases, the list uncurrying combinator (cU) need not be strict.

Since constants may be HASL arguments, the abstraction predicate must specify what the resulting expression ought to be: in a strict position, removing a constant from an expression *E* means that when the resulting expression is applied to an actual argument, that argument must match exactly the removed constant, and so the parse tree is modified from *E* to *cMATCH* ---> *X* ---> *E* where *X* is the constant being abstracted; otherwise the resulting tree is *cK--->E*.

We may now examine the *semantic* relation in detail. The *semantic* relation specifies a traversal of the parse tree which results in a new tree from which all identifiers except global identifiers have been removed. For a subtree of the form *X : Y* or *X ---> Y*, the resulting tree is specified by:

```
semantic(X:Y,Sx:Sy) :- semantic(X,Sx), semantic(Y,Sy).
semantic(X--->Y,Sx--->Sy) :- semantic(X,Sx), semantic(Y,Sy).
```

Related to a subtree of the form *where(Exp,Defs)* is a subtree *Combinators* specified by

```
semantic(where(Exp,Defs),Combinators) :- abstract_locals(where(Exp,Defs),Combinators).
```

The predcate *abstract_locals* reforms the *where* node into a subtree of the form *AbsE ---> (cY ---> AbsD)*:

```
abstract_locals(where(Exp,Defs),AbsE—>(cY—>AbsD)) :-
comp_defs(Defs,Ids,Abs),
abstract(cU,Ids,Exp,AbsE),
abstract(cU,Ids,Abs,AbsD).
```

cY is HASL's fixed point combinator whose reduction is defined as

```
cY—>X =>> Res :- X —> (cY —> X) =>> Res.
```

This is read as: *cY--->X* reduces to *Res* if *X--->(cY--->X)* reduces to *Res*. In the *abstract_locals* predicate, *Defs* are compiled by *comp_defs* to a list of identifiers defined *(Ids)* and a list of defined expressions from which all local variables have been removed *(Abs)*. The list of variables is abstracted from *Exp* and from *Abs*, specifying the subtrees *AbsE* and *AbsD*, respectively. The abstraction of *Ids* from *Abs* is the method of implementing mutually recursive definitions.

The predicate *comp_defs* builds the list of identifiers and abstractions by compiling each definition in *Def* using the predicate *comp_def*. A definition of arity 0 is left unchanged by the first clause of *comp_def*. As was mentioned in Section 2, the clauses defining a function are compiled as if one large unification expression had been specified. This compilation is specified by the predicate *comp_func*. The variables which are introduced by *comp_func* are of the form *id(1),id(2)*, etc., (these are not HASL variables) and must later be abstracted from *Code0* which is returned by *comp_func* to yield the *Code* tree for a definition:

```
comp_def(def(Name,0,Def),def(Name,0,Def)) :- !.
comp_def(def(Name,Arity,Funcs),def(Name,Arity,Code)) :-
Arity > 0,
comp_func(Funcs,Arity,Code0),
generate_seq(Arity,Ids),
abstract(cU_s,Ids,Code0,Code).
```

The predicate *generate_seq* specifies a relation between *Arity* and the list of introduced identifiers *Ids* which later gets removed!

A function is compiled clause by clause in reverse order. The last clause of any function is compiled by *comp_func* to

```
cCONDF —> Abs —> fail
```

where *Abs* is variable-free. *cCONDF* is a combinator defined as follows:

```
cCONDF —> X —> Y =>> Res :-
X =>> Rx, !,
cond_fail(Rx,Y,Res).
```

and is read: *cCONDF--->X--->Y* reduces to *Res* if *X* reduces to *Rx* and if *Rx* is not *fail* as determined by *cond_fail;* otherwise, *cond_fail* specifies that the value of *Res* is the value of the reduction of *Y*.

Remaining clauses defining a function are compiled by *comp1_func* to:

```
cCONDF —> Abs —> Sofar
```

where *Abs* is the compiled clause and *Sofar* is the code for the clauses already compiled.

A clause is compiled by the predicate *comp_clause:*

```
comp_clause(func(Fseq,Exp),Arity,Abs) :-
note_repeats(Fseq,MarkedFseq),
semantic(Exp,Sexp),
abstract(cU_s,MarkedFseq,Sexp,Aps),
generate_applies(Aps,Arity,Abs).
```

[1] The predicate *note_repeats* relates a list of formals, *Fseq,* to a marked list of formals *MarkedFseq,* where the second, third, etc., occurrences of a formal identifier *id(x)* have been replaced by *match(id(x)).* When *id(x)* is eventually abstracted from the right-hand side of a clause, this insures - by unification - that each occurrence of *id(x)* is matched to the same value. In the definition of *member* for example,

member a (a:x) = true

both occurrences of *a* must be bound to the same value. The *abstract* predicate treats repeated occurrences of an identifier in the way it treats constants.

[2] *Exp* is related by the *semantic* relation to *Sexp.*

[3] The marked formal sequence is abstracted from *Sexp* to yield *Aps.*

[4] The identifiers *id(1), id(2),* etc., are introduced.

The interested reader may follow on his own the specification of the *semantic* relation for subtrees labeled by the functor *unify* and for trees rooted at the functor *global.* It only needs to be said that a global definition such as

```
def suc x = 1 + x;
```

results in the following clause being added to HASL's database:

```
global(suc,cC1—>cCONDF—>(cADD—>num(1))—>fail).
```

Global names in any HASL expression are replaced at reduction time by their value as specified by the second component of *global.*

Some comments are due about the way we have compiled clauses into a function. In SASL, Turner allowed different clauses defining a function to have different arities. For example:

```
f 0 b = c
f 1 = d
f x y z = e
```

Thus, when an application of *f* is encountered in a SASL expression, it is impossible to know in advance, ie, at compile time, how much of the SASL expression to the right of *f* would actually be used by *f.* To cope with this, Turner introduced what he called a combinator "TRY, with rather peculiar reduction rules" [Turner,1981]. We had earlier implemented SASL in Prolog, and the specification of TRY in logic caused an enormous amount of trouble: it seemed to require at reduction time a stack to hold everything to the right of *f* in a SASL expression (ie, either to the end of the SASL expression, or to the first right parenthesis). The TRY combinator itself seemed to come in two arities: one of arity 3 for stacking everything to the right to be passed to each clause to be tried; and one of arity 2 to attempt clauses in order to find the applicable one. No other combinator seemed to require this explicit stack, but at reduction time the stack had to be passed as part of the state of the reduction to each combinator rule in case some clausally defined function were invoked. The presence of the stack in the logical specification seemed too operational and too distasteful, and there seemed no way to write the SASL reduction rules completely without it. This may have been simply a result of our confusion; or more profoundly a case where Wittgenstein's dictum held: *Was sich ueberhaupt sagen laesst, laesst sich klar sagen; und wovon man nicht reden kann, darueber muss man schweigen.* At any rate, HASL was born partly as a result of the hassle of trying to clean up the SASL reduction machine.

The cCONDF combinator was introduced to deal with a kind of conditional expression which arises often in dealing with unification based conditional binding expressions and in applying clausally defined functions: we could simply use the cCOND combinator, but the resulting code would be longer. Either way is simpler and clearer than using the TRY combinator! It should finally be noted that the restriction that all clauses defining a function have the same arity - which makes use of the cCONDF combinator feasible for compiling functions - imposes no loss of generality on what can be expressed in HASL: the sole interesting example in [Turner,1981] which makes use of different arities can be expressed without utilizing clauses of different arities.

7. The Specification of HASL Reduction.

The specification of the HASL reduction relation consists mainly of a set of rules as to how the HASL combinators are reduced. The combinator cS, for example, introduced in the previous section, is reduced as follows:

```
cS ---> Z ---> Y ---> X =>> Res :-
Z ---> X ---> (Y ---> X)  =>> Res.
```

Here, "=>>" is the infix reduction operator. The above specification is read: *cS ---> Z ---> Y ---> X* reduces to *Res* if *Z ---> X ---> (Y ---> X)* reduces to *Res*.

Associated with each combinator is an *arity,* for example:

```
arity(cS,3)
```

which indicates the number of arguments necessary for the reduction to take place. An expression such as

```
cS ---> X ---> Y
```

cannot be further reduced as it is already in head normal form. The reduction rules are listed in order of increasing arity; at the end of each group of rules for a given arity, there is a rule such as:

```
C ---> X ---> Y =>> C ---> X ---> Y :-
arity(C,D) , D >= 3 , !.
```

which would specify that *cS ---> X ---> Y* is already in head normal form.

The general reduction rule is to reduce the leftmost node of the combinator tree (the leftmost redex); if that node has not been reached, none of the combinator reduction rules apply. To handle the case of moving to the leftmost redex, the following (last but one) reduction rule applies:

```
X ---> Y ---> Z =>> Res :-
X ---> Y =>> Rxy,
not same(X--->Y,Rxy),
Rxy ---> Z =>> Res.
```

The reduction rules are recursively applied to try and reduce *X ---> Y* to head normal form; if *X ---> Y* is not in head normal form, then *Rxy* is head normal form for *X ---> Y* and *Rxy ---> Z* is reduced to *Res*.

The last reduction rule *X =>> X* specifies that *X* is already in head normal form.

Some combinators, such as the combinator *cCONDF,* defined in the last section, recursively call on the reduction machine. So does the combinator *cMATCH* which specifies unification:

```
cMATCH ---> X ---> Y ---> Z =>> Rees :-
X =>> Redx, !,
Z =>> Redz, !,
eqnormal(Redx,Redz,Y,fail,Req),
Req =>> Res.
```

X and *Z* are reduced to Redx and Redz, respectively, and if they have the same normal form, *Y* is unified with *Req* and is reduced to *Res;* otherwise, *fail* is unified with *Req* and a trivial reduction reduces the entire match to *fail.* The binding of arguments to HASL formal variables - another part of HASL's restricted unification - is accomplished at the reduction stage by the combinators simply placing rhe actual arguments in their proper places for evaluation!

HASL numbers, truth values, and characters are tagged by the functors *num, logical* and *char*. (Lists are tagged by :.) Various parts of the reduction machine use these functors for type checking. For example, the addition component of the "arithmetic unit" specifies that addition is strict:

```
add(num(X),num(Y),num(Z)) :- Z is X + Y, !.
add(X,Y,fail).
```

HASL type checking functions such as *number* are defined globally and apply type checking combinators such as

```
cNUMBER --> X ==>> Res :- type_check(num(X),Res).
```

The predicate *type_check* is specified by:

```
type_check(Form,logical(true)) :- Form, !.
type_check(Form,logical(false)).
```

The reduction from head normal form to normal form is specified by the relation >>> whcih also has the side effect of printing the value of the original HASL expression in an appropriate format.

8. Applications, Conclusions, Further Work.

[1] One of our interests is in building a logical translator writing system based on Scott-Strachey denotational semantics. The general idea is to use DCGs for lexical and syntactic analysis and to produce an applicative expression which denotes the "value" of a program. The applicative expression must then be reduced to its value. It is our intent to construct the system so that HASL expressions are used as the applicative expressions which denote the values of programs.

Peter Mosses [Mosses,1979] Semantics Implementation System (SIS, implemented in BCPL) provides a "hard-wired" model for this project. It allows one to specify a grammar and the denotational semantics of a language, and produces for any program in that language an applicative expression in a language called DSL which is a slightly sugared version of a lambda calculus language LAMB. As a first step in our project we will probably compile DSL expressions to combinators and use the HASL reduction machine to reduce DSL expressions to the values which they denote.

[2] HASL may be thought of as a functional sublanguage of Prolog. More generally, we can think of a deduction machine (eg, Prolog) which has a reduction machine (eg, HASL) as a component. Another model here is LOGLISP [Robinson&Siebert,1980] in which LOGIC is the deduction machine and LISP is the reduction machine. In the case of LOGLISP, however, it took quite a lot of work to define a suitable reduction mechanism for LISP: the notion of reduction of LISP expressions is fairly complex and is not identical to evaluation of LISP expressions. We suggest that since HASL is defined in terms of a notion of reduction *ab initio,* it is simpler and perhaps cleaner mathematically to consider a deduction-reduction machine with HASL as the reduction component. LOGLISP, however, treats the LOGIC machine and the LISP machine as equal components able to call on each other for computations; it remains to be investigated how HASL might call on the deduction machine.

[3] The HASL reduction machine has some notion of partial evaluation. If one defines

```
def f cond a b = cond -> a ; b:
```

then *f true* is the function which when applied to two arguments selects the first one. In terms of combinators, the reduction of *f true* is:

```
cC—>(cB1—>(cB1—>cC1)—>cCONDF—>
        cCOND—>logical(true))—>fail
```

Another observation is that the abstraction of variables from an expression is a relation between a variable, an expression, and another expression without that variable. The abstraction may be run "backwards" and a variable may be put into a variable-free combinator expression to get something more readable. For example, in:

```
abstr0(id(x),E,cS—>(cS—>(cK—>plus)—>(cK—>num(1)))—>cI).
```

we have

```
E = plus—>num(1)—>id(x).
```

HASL abstraction is more complicated than this, but in principle one may think of decompiling variable-free expressions.

One might think of combining these two observations to get a notion for a debugging method for applicative languages: a partially evaluated expression may have some variables put back into it, and then one might try using the lexical and syntactic DCGs as generators rather than as recognizers to produce a readable HASL expression.

It may not be entirely frivolous to think in fact of generating programs which compute a given value. The reduction relation may be run backward to derive combinator strings which could be translated into HASL expressions. Of course there are infinitely many such expressions and most of them are trivial and/or uninteresting. Could one place constraints on the searching of the space of HASL expressions which compute a given value to produce interesting expressions?

[4] Pragmatically, Prolog is ideal for designing and testing experimental languages. One tends not to carry out language experiments other than on paper - or in one's head - if implementation requires extensive coding in a low level language. But - the HASL interpreter described here, implemented in CProlog to run on a VAX 780 under Berkeley UNIX, is *slow*. A Prolog compiler which optimizes tail recursion and runs under UNIX is an absolute necessity.

9. Acknowledgements.

This work was supported by the National Science and Engineering Research Councilof Canada. I must also thank the UBC Laboratory for Computational Vision for time on its VAX running Berkeley Unix: modern and adequate computing facilities are not currently made available to the computer science department by UBC.

10. References.

[Abramson,1982a]

Abramson, H., *Unification-based Conditional Binding Constructs,* Proceedings First International Logic Programming Conference, Marseille, 1982.

[Abramson,1982b]

Abramson, H., *A Prolog Implementation of SASL,* Logic Programming Newsletter 4, Winter 1982/1983.

[Burge,1975]

Burge, W.H., *Recursive Programming Techniques,* Addison-Wesley, 1975.

[Colmerauer,1978]

Colmerauer, A., *Metamorphosis Grammars,* in Natural Language Communication with Computers, Lecture Notes in Computer Science 63, Springer, 1978.

[Henderson&Morris,1976]

Henderson, P. & Morris, J.H., *A lazy evaluator,* Conference Record of the 3rd ACM Symposium on Principles of Programming Languages, pp. 95-103, 1976.

[Moss,1979]

Moss, C.D.S., *A Formal Description of ASPLE Using Predicate Logic,* DOC 80/18, Imperial College, London.

[Moss,1981]

Moss, C.D.S., *The Formal Description of Programming Languages using Predicate Logic,* Ph.D. Thesis, Imperial College, 1981.

[Moss,1982]

Moss, C.D.S., *How to Define a Language Using Prolog,* Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, Tittsburgh, Pennsylvania, pp. 67-73, 1982.

[Mosses,1979]

Mosses, Peter, *SIS - Semantics Implementation System: Reference Manual and User Guide,* DAIMI MD-30, Computer Science Department, Aarhus University, Denmark, 1979.
[Pereira&Warren,1980]

Pereira, F.C.N. & Warren, D.H.D, *Definite Clause Grammars for Language Analysis,* Artificial Intelligence, vol. 13, pp. 231-278, 1980.

[Robinson&Siebert,1980a]

Robinson, J.A. & Siebert, E.E., *LOGLISP - an alternative to Prolog,* School of Computer and Information Science, Syracuse University, 1980.

[Robinson&Siebert,1980b]

Robinson, J.A. & Siebert, E.E., *Logic Programming in LISP,* School of Computer and Information Science, Syracuse University, 1980.

[Turner,1976]

Turner, D.A., *SASL Language Manual,* Department of Computational Science, University of St. Andrews, 1976, revised 1979.

[Turner,1979]

Turner, D.A., *A new implementation technique for applicative languages,* Software - Practice and Experience, vol. 9, pp. 31-49.

[Turner,1981]

Turner, D.A., *Aspects of the Implementation of Programming Languages: The Compilation of an Applicative Language to Combinatory Logic,* Ph.D. Thesis, Oxford, 1981.

[Warren,1977]

Warren, David H.D., *Logic programming and compiler writing,* DAI Research Report 44, University of Edinburgh, 1977.

```
/* lexical rules */

reserved(true,constant(logical(true))).
reserved('TRUE',constant(logical(true))).
reserved(false,constant(logical(false))).
reserved('FALSE',constant(logical(false))).
reserved(fail,constant(fail)).
reserved('FAIL',constant(fail)).
reserved(def,def).
reserved('DEF',def).
reserved(where,where).
reserved('WHERE',where).

lexemes(X)      --> space , lexemes(X).
lexemes([X|Y]) --> lexeme(X) , lexemes(Y).
lexemes([])     --> [].


lexeme(Token) -->
        word(W) , ! , { name(X,W) , ( reserved(X,Token) ; id(X) = Token ) }.
lexeme(constant(Con)) --> constant(Con) , !.
lexeme(Punct)           --> punctuation(Punct) , !.
lexeme(op(Pr,Comb))   --> op(Pr,Comb) , !.

space --> " " , !.
space --> [10], !. /* carriage return */

num(num(N)) --> number(Number) , ! , { name(N,Number) }.

number([D|Ds]) --> digit(D) , digits(Ds).

digit(D) --> [D] , { D>47, D<58 }. /* 0...9 */

digits([D|Ds]) --> digit(D) , digits(Ds).
digits([])     --> [].

word([L|Ls]) --> letter(L) , lords(Ls).

letter(L) --> [L] , { (L>96,L<123 ; L>64,L<90) }. /* a-z, A-Z */

lords([L|Ls]) --> ( letter(L) ; digit(L) ) , lords(Ls).
lords([])     --> [].

/* in op(N,O) N designates the binding power of the operator O. */
op(0,cAPPEND) --> "++" , !.
op(0,cCONS)   --> ":" , !.
op(1,cOR)     --> "|" , !.
op(2,cAND)    --> "&" , !.
op(3,cLSE)    --> "<=" , !.
op(3,cGRE)    --> ">=" , !.
op(3,cNEQ)    --> "=" , !.
op(3,cEQ)     --> "=" , !.
op(3,cGR)     --> ">" , !.
op(3,cLS)     --> "<" , !.
op(4,cADD)    --> "+" , !.
op(4,cSUB)    --> "-" , !.
op(5,cMULT)   --> "*" , !.
op(5,cDIV)    --> "/" , !.
op(6,cB)      --> "." , !.

hasl_string(C:Cs) --> stringchar(C) , hasl_string(Cs).
hasl_string(nil)  --> [].

hasl_char(C) --> "%" , stringchar(C) , !.


stringchar(char(A))    --> [C] , { C =\= 39 , name(A,[C]) } , !.
stringchar(char('')) --> "''" , !.

string(S) --> "'" , hasl_string(S) , "'" ,!.

constant(N)   --> num(N) , !.
constant(C)   --> hasl_char(C) , !.
constant(S)   --> string(S) , !.
constant(nil) --> "[]" , !.

punctuation(tilde)          --> "~" , !.
punctuation(comma)          --> "," , !.
punctuation(lparen)         --> "(" , !.
punctuation(rparen)         --> ")" , !.
punctuation(condarrow)      --> "->" , !.
punctuation(rightcrossbow) --> "-}", !.
punctuation(leftcrossbow)  --> "{-", !.
punctuation(lbrack)         --> "[", !.
punctuation(rbrack)         --> "]", !.
punctuation(unifyarrow)     --> "=>", !.
punctuation(semicolon)      --> ";" , !.

/* The following predicates constitute the interface
   between lexical and syntactic analysis. Predicates
   with names starting with 't', eg, tCOLON, are the
   terminals in syntactic analysis.
*/

tCOLON          --> [op(0,cCONS)].
tPLUSPLUS       --> [op(0,cAPPEND)].
tCOMMA          --> [comma].
tLBRACK         --> [lbrack].
tRBRACK         --> [rbrack].
tLPAREN         --> [lparen].
tRPAREN         --> [rparen].
tUNIFYARROW     --> [unifyarrow].
tLEFTCROSSBOW   --> [leftcrossbow].
tRIGHTCROSSBOW --> [rightcrossbow].
tCONDARROW      --> [condarrow].
tEQUAL          --> [op(3,cEQ)].
tSEMICOLON      --> [semicolon].
```

```
tWHERE        --> [where].
tDEF          --> [def].
tNOT          --> [tilde].
tNEGATE       --> [op(4,cSUB)].
tPLUS         --> [op(4,cADD)].
tIDENT(id(Id)) --> [id(Id)].
tCONSTANT(C)  --> [constant(C)].
tOP(Pr,Comb)  --> [op(Pr,Comb)].
```

```
/* syntactic rules */

def(global(Ds)) -->
      tDEF , defs(Ds) , tSEMICOLON.


definition(def(Id,Arity,func(Fseq,Exp))) -->
      tIDENT(Id) , fseq(Fseq) , ! , tEQUAL , expression(Exp) ,
      { seq_length(Fseq,Arity) }.

definition(Def) -->
      formal(Formal) , ! , tEQUAL , expression(Exp) ,
      { expandef(def(Formal,0,Exp),Def) }.

defs(Ds) --> definition(D) , ! , { append_def(D,[],Deflist) } ,
         defs1(Deflist,Ds).

defs1(D,Ds) --> tCOMMA , definition(Def) , ! ,
            { mergedef(D,Def,Dm) } , defs1(Dm,Ds).
defs1(D,D)  --> [].

fseq(Fseq) --> formal(Formal) , ! , fseq1(Formal,Fseq).

fseq1(F1,[F1|F]) --> formal(F2) , ! , fseq1(F2,F).
fseq1(F,F) --> [].

formal(Id) --> tIDENT(Id) , !.
formal(const(C)) --> tCONSTANT(C) , !.
formal(flist(Flist)) --> tLBRACK , flist(Flist) , ! , tRBRACK.
formal(flist(Flist)) --> tLPAREN , fprimary(Flist) , ! , tRPAREN.

flist(F1:F2) --> fprimary(F1) , ! , flist1(F2).
flist(const(nil)) --> [].

flist1(F) --> tCOMMA , flist(F).
flist1(const(nil)) --> [].

fprimary(F) --> formal(F1) , ! , fprimary1(F1,F).

fprimary1(F1,F1:F) --> tCOLON , formal(F2) , ! , fprimary1(F2,F).
fprimary1(F,F) --> [].

expression(E) --> def(E).
expression(E) --> unification(E1) , ! , expression(E1,E).

expression(E1,where(E1,Ds)) --> tWHERE , defs(Ds).
expression(E,E) --> [].

unification(unify(Fseq,E1,E2,E3)) -->
     fseq(Fseq) , tLEFTCROSSBOW , expression(E1) , tUNIFYARROW ,
       expression(E2) , tSEMICOLON , expression(E3).

unification(U) --> condexp(U).
```

```
condexp(E) -> expl(E1,0) , ! , condexpl(E1,E).

condexpl(E1,cCOND --> E1 --> E2 --> E3) ->
      tCONDARROW , expression(E2) , ! , tSEMICOLON , condexp(E3).
condexpl(E1,unify(Fseq,E1,E2,E3)) ->
      tRIGHTCROSSBOW , fseq(Fseq) , tUNIFYARROW ,
        expression(E2) , tSEMICOLON , expression(E3).
condexpl(E,E) -> [].

expl(E,P) -> tPLUS , expl(E1,6) , ! , exp2(E1,E,P).
expl(E,P) -> tNEGATE , expl(E1,6) , ! , exp2(cNEGATE --> E1,E,P).
expl(E,P) -> tNOT , expl(E1,3) , ! , exp2(cNOT --> E1,E,P).
expl(E,P) -> comb(E1) , ! , exp2(E1,E,P).

/* since : or cons is a primitive in HASL: */
exp2(E1,E,0) -> tCOLON , expl(E2,0) , ! , exp2(E1 : E2,E,1).

/* since + + or append is the only other zero level operator: */
exp2(E1,E,0) -> tPLUSPLUS , expl(E2,0) , ! , exp2(cAPPEND --> E1 --> E2,E,1).

/* : and + + are right associative; all others are left associative: */
exp2(E1,E,P) -> tOP(Q,Op) , { P < Q } , ! , expl(E2,Q) ,
            exp2(Op --> E1 --> E2,E,P).

exp2(E,E,P) -> [].

comb(C) -> primary(P) ,  ! , combl(P,C).

combl(P1,C) -> primary(P) , ! , combl(P1 --> P,C).
combl(C,C) -> [].

primary(L) -> tLBRACK , explist(L) , ! , tRBRACK.
primary(I) -> tIDENT(I) , !.
primary(C) -> tCONSTANT(C) , !.
primary(E) -> tLPAREN , expression(E) , ! , tRPAREN.


explist(E1 : E2) -> expl(E1,0) , ! , explistl(E2).
explist(nil) -> [].
explistl(E) -> tCOMMA , explist(E).
explistl(nil) -> [].

/* The following predicates are used to check that each clause
   defining a function has the same arity, and to merge all
   definitions made at the same time into a single list of
   definitions.
*/

mergedef(Deflist,Def,Defmerge) :-
      flat(Def,FlatDef) ,
      merge(Deflist,FlatDef,Defmerge).

merge(Deflist,[Def|Defs],Defmerge) :-
    merge(Deflist,Def,Deflist1) ,
    merge(Deflist1,Defs,Defmerge).

merge([def(id(X),0,D)|Deflist],
     def(id(X),0,D1),
     [def(id(X),0,D)|Deflist]) :-
     write(X) ,
     write(' is a constant already defined: ') ,
     write(D) , nl ,
     write('definition ignored: ') ,
     write(D1) , nl.
merge([def(id(X),N,D)|Deflist],
     def(id(X),N,D1),
     [def(id(X),N,[D1|D])|Deflist]) :- !.
merge([def(id(X),N,D)|Deflist],
     def(id(X),M,D1),
     [def(id(X),N,D)|Deflist]) :-
     write('wrong number of arguments in definition of:') ,
     write(def(id(X),M,D1)) ,
     write('should be ') , write(N) , nl.
merge([def(id(Y),M,Dy)|Deflist],
     def(id(X),N,D),
     [def(id(Y),M,Dy)|Deflist]) :-
     defined(X,Deflist,Dx) , ! ,
     write(X) ,
     write(' already defined: ') ,
     write(Dx) , nl ,
     write(def(id(X),N,D)) ,
     write(' ignored.') , nl.
merge(Deflist,
     Def,
     [Def|Deflist]).

defined(Y,[def(id(Y),_,Dy)|_],Dy).
defined(Y,[def(id(X),_,_)|Deflist],Dy) :-
    defined(Y,Deflist,Dy).

seq_length([F|G],N) :- ! ,
        seq_length(G,M) ,
        N is 1 + M.
seq_length(F,1).


append_def(def(A,B,C),Z,[def(A,B,C) |Z]) :- !.
append_def([X|Y],Z,[X|W]) :-
      append_def(Y,Z,W).

flat(def(X,Y,Z),def(X,Y,Z)).
flat([def(X,Y,Z)|Defs],[def(X,Y,Z)|FDefs]) :-
           flat(Defs,FDefs) , !.
flat([DefHd|DefTl],Flat) :-
      flat(DefHd,FlatHd) ,
      flat(DefTl,FlatTl) ,
      append_def(FlatHd,FlatTl,Flat).
```

```
expandef(Defs,Def) :-
    expand(Defs,Def1) ,
    flat(Def1,Def).

expand(def(flist(X:const(nil)),0,Exp),Defx) :-
    expand(def(X,0,Exp),Defx).
expand(def(flist(X:Y),0,Exp),[Defx|Defy]) :-
    expand(def(X,0,cHD ---> Exp),Defx) ,
    expand(def(flist(Y),0,cTL ---> Exp),Defy).
expand(def(flist(X),0,Exp),def(X,0,Exp)).
expand(def(F,0,Exp),def(F,0,Exp)).
```

```
/* semantic rules */

semantic(X:Y,Sx:Sy) :-
      semantic(X,Sx) ,
      semantic(Y,Sy).
semantic(X-->Y,Sx-->Sy) :-
      semantic(X,Sx) ,
      semantic(Y,Sy).
semantic(where(Exp,Defs),Combinators) :-
      abstract_locals(where(Exp,Defs),Combinators).
semantic(unify(Fseq,E1,E2,E3),cCONDF-->Exp-->Se3) :-
      semantic(E1,Se1) ,
      semantic(E2,Se2) ,
      semantic(E3,Se3) ,
      translate_unification(Fseq,Se1,Se2,Exp).
semantic(global(Defs),global) :-
      installdefs(Defs).
semantic(X,X).

abstract(U,nil,Abs,Abs).
abstract(U,flist(Flist),Exp,Abs) :-
      abstract(U,Flist,Exp,Abs).
abstract(U,[X|Y],E,Abs) :-
      abstract(U,Y,E,Abs1) ,
      abstract(U,X,Abs1,Abs).
abstract(U,id(X),E,Abs) :-
      abstr(id(X),E,Abs).
abstract(cU,const(X),E,cK --> E).
abstract(cU_s,const(X),E,cMATCH --> X --> E).
abstract(cU_s,match(X),E,cMATCH --> X --> E).
abstract(U,(X : Y),E,U --> Abs) :-
      abstract(U,Y,E,Abs1) ,
      abstract(U,X,Abs1,Abs).


abstr(V,X --> Y,Abs) :-
      abstr(V,X,AX) ,
      abstr(V,Y,AY) ,
      combine(-->,AX,AY,Abs) , !.
abstr(V,(X : Y),Abs) :-
      abstr(V,X,AX) ,
      abstr(V,Y,AY) ,
      combine(:,AX,AY,Abs) , !.
abstr(id(X),id(X),cI) :- !.
abstr(V,X,cK --> X).

combine(-->,cK --> X,cK --> Y,cK --> (X --> Y)).
combine(-->,cK --> X,cI,X).
combine(-->,cK --> (X1 --> X2),Y,cB1 --> X1 --> X2 --> Y).
combine(-->,cK --> X,Y,cB --> X --> Y).
combine(-->,cB --> X1 --> X2,cK --> Y,cC1 --> X1 --> X2 --> Y).
combine(-->,X,cK --> Y,cC --> X --> Y).
combine(-->,cB --> X1 --> X2,Y,cS1 --> X1 --> X2 --> Y).
combine(-->.X,Y,cS --> X --> Y).
combine(:,cK --> X,cK --> Y,cK --> (X : Y)).
combine(:,cK --> X,Y,cB_p --> X --> Y).
combine(:,X,cK --> Y,cC_p --> X --> Y).
combine(:,X,Y,cS_p --> X --> Y).

generate_seq(1,id(1)) :- !.
generate_seq(N,Y) :-
      N1 is N - 1 ,
      gen_seq(N1,id(N),Y).

gen_seq(1,X,[id(1)|X]) :- !.
gen_seq(N,X,Y) :-
      N1 is N - 1 ,
      gen_seq(N1,[id(N)|X],Y).

generate_applies(X,N,Y) :-
      generate_seq(N,Seq) ,
      gen_applies(X,Seq,Y).

gen_applies(X,[Hd|Tl],Y) :- ! ,
      gen_applies(X --> Hd,Tl,Y).
gen_applies(X,S,X --> S).

restructure(X --> (Y --> Z),W) :- restruct(X,Y --> Z,W).
restructure(X --> Y,X --> Y).

restruct(X,Y --> Z,A --> Z  ) :- restruct(X,Y,A).
restruct(X,Y,X --> Y).

comp_clause(func(Fseq,Exp),Arity,Abs) :-
      note_repeats(Fseq,MarkedFseq) ,
      semantic(Exp,Sexp) ,
      abstract(cU_s,MarkedFseq,Sexp,Aps) ,
      generate_applies(Aps,Arity,Abs).

comp_func([func(Fseq,Exp)|Funcs],Arity,Code) :-
      comp_clause(func(Fseq,Exp),Arity,Abs) ,
      comp1_func(Funcs,Arity,cCONDF --> Abs --> fail,Code) .
comp_func(func(Fseq,Exp),Arity,cCONDF --> Abs --> fail) :-
      comp_clause(func(Fseq,Exp),Arity,Abs).

comp1_func([func(Fseq,Exp)|Funcs],Arity,Sofar,Code) :-
      comp_clause(func(Fseq,Exp),Arity,Abs) ,
      comp1_func(Funcs,Arity,cCONDF --> Abs --> Sofar,Code).
comp1_func(func(Fseq,Exp),Arity,Sofar,cCONDF --> Abs --> Sofar) :-
      comp_clause(func(Fseq,Exp),Arity,Abs).

comp_def(def(Name,0,Def),def(Name,0,Def)) :- !.
comp_def(def(Name,Arity,Funcs),def(Name,Arity,Code)) :-
      Arity > 0 , ! ,
      comp_func(Funcs,Arity,Code0) ,
      generate_seq(Arity,Ids) ,
      abstract(cU_s,Ids,Code0,Code).
```

```
comp_defs([Def|Defs],Ids,Abs) :-
      comp_def(Def,def(id(Id),Arity,Abs1)) ,
      comp_defs1(Defs,id(Id),Abs1,Ids,Abs).

comp_defs1([],Ids,Abs,Ids,Abs).
comp_defs1([Def|Defs],IdsIn,AbsIn,Ids,Abs) :-
      comp_def(Def,def(id(Id),Arity,Abs1)) ,
      comp_defs1(Defs,id(Id):IdsIn,Abs1:AbsIn,Ids,Abs).

abstract_locals(where(Exp,Defs),AbsE —> (cY —> AbsD)) :-
      comp_defs(Defs,Ids,Abs) ,
      abstract(cU,Ids,Exp,AbsE) ,
      abstract(cU,Ids,Abs,AbsD).

translate_unification(Fseq,E1,E2,Exp) :-
      note_repeats(Fseq,MarkedFseq) ,
      abstract(cU_s,MarkedFseq,E2,Abs) ,
      restructure(Abs —> E1,Exp).

installdefs(Defs) :-
       comp_defs(Defs,Ids,Abs) ,
       install(Ids,Abs).

install(id(Id):Ids,Def:Defs) :-
      global(Id,DefId) , ! ,
      write(Id) , write(' already globally defined.') ,
      write(' New definition ignored.') , nl ,
      install(Ids,Defs).
install(id(Id):Ids,Def:Defs) :-
      assertz(global(Id,Def)) ,
      install(Ids,Defs).
install(id(Id),Abs) :-
      global(Id,DefId) , ! ,
      write(Id) , write(' already globally defined.') ,
      write(' New definition ignored.') , nl.
install(id(Id),Def) :-
      assertz(global(Id,Def)).

member(Id,[Id|_]).
member(Id,[_|Ids]) :- member(Id,Ids).

note_repeats(Fseq,Marked) :-
          mark_repeats(Fseq,[],_,Marked).

mark_repeats(id(Id),In,In,match(id(Id))) :-
          member(Id,In) , !.
mark_repeats(id(Id),In,[Id|In],id(Id)).
mark_repeats(flist(Flist),In,Out,flist(Marked)) :-
          mark_repeats(Flist,In,Out,Marked).
mark_repeats(Hd:Tl,In,Out,MarkedHd:MarkedTl) :-
          mark_repeats(Hd,In,In1,MarkedHd) ,
          mark_repeats(Tl,In1,Out,MarkedTl).
mark_repeats([Hd|Tl],In,Out,[MarkedHd|MarkedTl]) :-
          mark_repeats(Hd,In,In1,MarkedHd) ,
        mark_repeats(Tl,In1,Out,MarkedTl).
mark_repeats([],In,In,[]).
mark_repeats(const(C),In,In,const(C)).
```

```
/* reduction rules */

id(X) --> id(Y) =>> Res :-
        global(X,DefX) ,
        global(Y,DefY) ,
        DefX --> DefY =>> Res.

id(X) =>> Def :-
        global(X,Def) , !.

id(X)  =>> _:-
        nl ,
        write('not defined: ') ,
        write(X) , nl , abort.

cI --> X =>> Res :-
        X =>> Res.

cY --> X =>> Res :-
        X --> (cY --> X) =>> Res.

cHD --> (X : Y) =>> Res :- ! ,
        X =>> Res.

cHD --> X =>> Res :-
        X =>> (Hd : Tl) , ! ,
        Hd =>> Res.

cTL --> (X : Y) =>> Res :- ! ,
        Y =>> Res.

cTL --> X =>> Res :-
        X =>> (Hd : Tl) , ! ,
        Tl =>> Res.

cCHAR --> X =>> Res :-
        type_check(char(X),Res).

cFAILURE --> X =>> Res :-
        type_check(failure(X),Res).

cLOGICAL --> X =>> Res :-
        type_check(logical(X),Res).

cFUNCTION --> X =>> Res :-
        type_check(function(X),Res).

cNUMBER --> X =>> Res :-
        type_check(num(X),Res).

cNOT --> X =>> Res :-
        X =>> Rx , ! ,
        choose(Rx,logical(false),logical(true),Res).
```

```
cNEGATE --> X =>> Res :-
          arith(sub,num(0),X,Res).

num(X) --> Y =>> num(X).

logical(X) --> Y =>> logical(X).

char(X) --> Y =>> char(X).

nil --> X =>> nil.

C --> X =>> C --> X :-
          arity(C,D) ,
          D >= 2 , !.

id(X) --> Y =>> Res :-
          global(X,Def) , ! ,
          Def --> Y =>> Res.

id(X) --> Y =>> Res :-
          nl ,
          write('not defined: ') ,
          write(X) , nl , abort.


Y --> id(X) =>> Res :-
          global(X,Def) , ! ,
          Y --> Def =>> Res.

Y --> id(X) =>> Res :-
          nl ,
          write('not defined: ') ,
          write(X) , nl , Y --> fail =>> Res.

(X : Y) --> num(1) =>> Res :- ! ,
          X =>> Res.

(X : Y) --> num(Z) =>> Res :- ! ,
          Z > 1 ,
          Z1 is Z - 1 ,
          Y --> num(Z1) =>> Res.

(X : Y) --> Z =>> Res :-
          Z =>> num(Num) , ! ,
          X : Y --> num(Num) =>> Res.

cK --> X --> Y =>> Res :-
          X =>> Res.

cU --> X --> Y =>> Res :-
          X --> (cHD --> Y) --> (cTL --> Y)  =>> Res.

cU_s --> X --> (Y : Z) =>> Res :- ! ,
          X --> Y --> Z =>> Res.
```

```
cU_s —> X —> Y =>> Res :-
          Y =>> (Hd : Tl) , ! ,
          X —> Hd —> Tl =>> Res.
cU_s —> X —> Y =>> fail.

cAND —> X —> Y =>> Res :-
          X =>> Rx , ! ,
          choose(Rx,Y,logical(false),Res).

cOR —> X —> Y =>> Res :-
          X =>> Rx , ! ,
          choose(Rx,logical(true),Y,Res).

cEQ —> X —> Y =>> logical(Res) :-
          X =>> Rx , ! ,
          Y =>> Ry , ! ,
          eqnormal(Rx,Ry,true,false,Res).

cNEQ —> X —> Y =>> logical(Res) :-
          X =>> Rx , ! ,
          Y =>> Ry , ! ,
          eqnormal(Rx,Ry,false,true,Res).

cAPPEND —> nil —> Z =>> Z :- !.
cAPPEND —> (X : Y) —> Z =>> (X : Res) :- ! ,
          cAPPEND —> Y —> Z =>> Res.
cAPPEND —> X —> Y =>> Res :-
          X =>> Resx , ! ,
          not same(X,Resx) ,
          cAPPEND —> Resx —> Y =>> Res.

cLSE —> X —> Y =>> Res :-
          cNOT —> (cGR —> X —> Y) =>> Res , !.

cGRE —> X —> Y =>> Res :-
          cNOT —> (cLS —> X —> Y) =>> Res , !.

cLS —> X —> Y =>> Res :-
          arith(ls,X,Y,Res) , !.

cGR —> X —> Y =>> Res :-
          arith(gr,X,Y,Res) , !.

cADD —> X —> Y =>> Res :-
          arith(add,X,Y,Res) , !.

cSUB —> X —> Y =>> Res :-
          arith(sub,X,Y,Res) , !.

cMULT —> X —> Y =>> Res :-
          arith(mult,X,Y,Res) , !.

cDIV —> X —> Y =>> Res :-
          arith(div,X,Y,Res) , !.
```

```
cCONDF --> X --> Y =>> Res :-
        X =>> Rx , ! ,
        cond_fail(Rx,Y,Res).

C --> X --> Y =>> C --> X --> Y :-
        arity(C,D) ,
        D >= 3 , !.

cCOND --> X --> Y --> Z =>> Res :-
        X =>> Resx , ! ,
        choose(Resx,Y,Z,Res).

cMATCH --> nil --> Y --> Z =>> Res :-
        match(nil,Z,Y,Res).
cMATCH --> num(X) --> Y --> Z =>> Res :-
        match(num(X),Z,Y,Res).
cMATCH --> char(X) --> Y --> Z =>> Res :-
        match(char(X),Z,Y,Res).
cMATCH --> logical(X) --> Y --> Z =>> Res :-
        match(logical(X),Z,Y,Res).
cMATCH --> X --> Y --> Z =>> Res :-
        X =>> Redx , ! ,
        Z =>> Redz , ! ,
        eqnormal(Redx,Redz,Y,fail,Req) , ! ,
        Req =>> Res.

cS_p --> X --> Y --> Z =>> Res :-
        (X --> Z) : (Y --> Z) =>> Res.

cB_p --> X --> Y --> Z =>> Res :-
        X : (Y --> Z) =>> Res.

cC_p --> X --> Y --> Z =>> Res :-
        X --> Z : Y =>> Res.

cS --> Z --> Y --> X =>> Res :-
        Z --> X --> (Y --> X) =>> Res.

cB --> X --> Y --> Z =>> Res :-
        X --> (Y --> Z) =>> Res.

cC --> X --> Y --> Z =>> Res :-
        X --> Z --> Y =>> Res.

C --> X --> Y --> Z =>> C --> X --> Y --> Z :-
        arity(C,D) ,
        D >= 4 , !.

cS1 --> W --> X --> Y --> Z =>> Res :-
        W --> (X --> Z) --> (Y --> Z) =>> Res.

cB1 --> W --> X --> Y --> Z =>> Res :-
        W --> X --> (Y --> Z) =>> Res.
```

```
cC1 --> W --> X --> Y --> Z =>> Res :-
        W --> (X --> Z) --> Y =>> Res.

X --> Y --> Z =>> Res :-
        X --> Y =>> Rxy ,
        not same(X --> Y,Rxy) , ! ,
        Rxy --> Z =>> Res.

X =>> X.

same(X,X).

choose(logical(true),Y,Z,Res) :-
        Y =>> Res , !.
choose(logical(false),Y,Z,Res) :-
        Z =>> Res , !.
choose(X,Y,Z,fail).

match(X,Y,Z,Res) :-
        Y =>> Ry , ! ,
        eqnormal(X,Ry,Z,fail,R) , ! ,
        R =>> Res , !.

eqnormal(X,Y,T,F,T) :-
        equals(X,Y) , !.
eqnormal(X,Y,T,F,F).

equals(num(X),num(X)).
equals(char(X),char(X)).
equals(logical(X),logical(X)).
equals(nil,nil).
equals((A : B),(X : Y)) :-
        A =>> Reda ,
        X =>> Redx ,
        equals(Reda,Redx) , ! ,
        equals(B,Y).

is_failure((X --> Y)) :-
        is_failure(X).
is_failure(fail).

is_function((X --> Y)) :- is_function(X).
is_function(X) :- arity(X,_).

arity(cI,1).
arity(cY,1).
arity(cV,1).
arity(cHD,1).
arity(cTL,1).
arity(cNOT,1).
arity(cFUNCTION,1).
arity(cCHAR,1).
arity(cLOGICAL,1).
arity(cNUMBER,1).
arity(cFAILURE,1).
arity(cK,2).
arity(cU,2).
arity(cU_s,2).
arity(cEQ,2).
arity(cNEQ,2).
arity(cAND,2).
arity(cOR,2).
arity(cAPPEND,2).
arity(cCONDF,2).
arity(cSUB,2).
arity(cADD,2).
arity(cMULT,2).
arity(cDIV,2).
arity(cGRE,2).
arity(cLSE,2).
arity(cLS,2).
arity(cGR,2).
arity(cS,3).
arity(cC,3).
arity(cB,3).
arity(cS_p,3).
arity(cCOND,3).
arity(cMATCH,3).
arity(cB_p,3).
arity(cC_p,3).
arity(cS,3).
arity(cS1,4).
arity(cB1,4).
arity(cC1,4).

type_check(Form,logical(true)) :- Form , !.
type_check(Form,logical(false)).

char(X) :- X =>> char(_).

logical(X) :- X =>> logical(_).

num(X) :- X =>> num(_).

failure(X) :- X =>> Rx , ! , is_failure(Rx).

list(X) :- id(list) --> X =>> logical(true).

function(X) :- X =>> Rx , ! ,  is_function(Rx).

add(num(X),num(Y),num(Z)) :- Z is X +  Y , !.
add(X,Y,fail).

sub(num(X),num(Y),num(Z)) :- Z is X - Y , !.
sub(X,Y,fail).

mult(num(X),num(Y),num(Z)) :- Z is X * Y , !.
mult(X,Y,fail).
```

```
div(num(X),num(Y),num(Z)) :- Z is X / Y , !.
div(X,Y,fail).

eq(X,Y) :- X =:= Y.

gr(num(X),num(Y),logical(true)) :- X > Y , !.
gr(num(X),num(Y),logical(false)) :- !.
gr(X,Y,fail).

ls(num(X),num(Y),logical(true)) :- X < Y , !.
ls(num(X),num(Y),logical(false)) :- !.
ls(X,Y,fail).

cond_fail(X,Y,X) :- not is_failure(X) , !.
cond_fail(_,Y,Ry) :- Y =>> Ry.

arith(Operation,X,Y,Res) :-
        X =>> Rx ,
        Y =>> Ry ,
        arithop(Operation,Rx,Ry,Res).

arithop(add,X,Y,Z) :-
        add(X,Y,Z).
arithop(sub,X,Y,Z) :-
        sub(X,Y,Z).
arithop(mult,X,Y,Z) :-
        mult(X,Y,Z).
arithop(div,X,Y,Z) :-
        div(X,Y,Z).
arithop(ls,X,Y,Z) :-
        ls(X,Y,Z).
arithop(gr,X,Y,Z) :-
        gr(X,Y,Z).

/* reduce to normal form */

reducelist(nil,nil).
reducelist(Hd : Tl,Nhd : Ntl) :- write(',') ,
                Hd =>> Redhd , ! ,
                Redhd >>> Nhd ,
                Tl =>> Redtl , ! ,
                reducelist(Redtl,Ntl).

reducestring(nil,nil).
reducestring(char(C),char(C)) :- write(C).
reducestring(Hd : Tl,Nhd : Ntl) :-
          Hd =>> Redhd , ! ,
          reducestring(Redhd,Nhd) ,
          Tl =>> Redtl , ! ,
          reducestring(Redtl,Ntl).

num(X) >>> num(X) :- write(X).

char(X) >>> char(X) :- write('%') , write(X).
```

```
logical(X) >>> logical(X) :- write(X).

X >>> fail :- is_failure(X) , write(fail).

nil >>> nil :- write('[]').

Hd:Tl >>> Nstr :-
                id(string) --> (Hd:Tl) =>> logical(true) , ! ,
                write('"') ,
                reducestring(Hd:Tl,Nstr) ,
                write('"').

Hd : Tl >>> Nhd : Ntl :-
                id(list) --> (Hd:Tl) =>> logical(true) , ! ,
                write('[') ,
                Hd =>> Redhd , ! ,
                Redhd >>> Nhd ,
                Tl =>> Redtl ,
                reducelist(Redtl,Ntl) ,
                write(']').

Hd : Tl >>> Nhd : Ntl :-
                Hd =>> Redhd , ! ,
                Redhd >>> Nhd ,
                write(':') ,
                Tl =>> Redtl ,
                Redtl >>> Ntl.

X >>> X :- write(X).
```