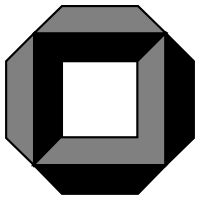
Universität Karlsruhe
Fakultät für Informatik



Optimix Language Report

(for OPTIMIX 7.0)

Uwe Aßmann
Institut für Programmstrukturen und Datenorganisation

Interner Bericht 31/95
July 20, 1995

Abstract

This is the language manual for OPTIMIX, the optimizer generator. It can be used to generate program analyses and transformations. Its input language is based on DATALOG and graph rewriting. Especially two new classes of graph rewrite systems are used: edge addition rewrite systems (EARS) and stratified graph rewrite systems (stratified GRS).
OPTIMIX has been developed in the Esprit project COMPARE (No. 5399). It is currently not free and can be used only in the context of the CoSy compiler framework. For a licence, contact the author or `info@ace.nl`.

Keywords: Program analysis, program transformation, optimizer generator, Datalog, graph rewriting.

This work has been funded by the ESPRIT project COMPARE (No. 5399).

Address: Uwe Aßmann, Universität Karlsruhe, IPD, Vincenz-Prießnitz-Straße 3, D-76128 Karlsruhe
Email: assmann@informatik.uni-karlsruhe.de
The address of the author will be, starting from Aug 1, 1995:
Uwe Aßmann, INRIA Rocquencourt, Domaine de Voluceau BP 105, 78153 Le Chesnay Cedex, France
Email: Uwe.Assmann@inria.fr

Contents

Chapter 1

General topics

This is the language manual for OPTIMIX, the optimizer generator. It can be used to generate program analyses and transformations. Its input language is based on DATALOG and graph rewriting [Aß94b] [Aß95]. Especially two new classes of graph rewrite systems are used: edge addition rewrite systems (EARS) and stratified graph rewrite systems (stratified GRS).
OPTIMIX has been developed in the Esprit project COMPARE (No. 5399). It is currently not free and can be used only in the context of the CoSy compiler framework. For a licence, contact the author or `info@ace.nl`.

1.1 Design procedure for an optimizer

In order to generate optimizer parts with OPTIMIX we propose the following procedure.

1. Write down all preconditions for a transformation, perhaps in text.
2. Define the data model of your application in fSDL [Buh95], i.e. define which parts of the knowledge you want to present should be objects and which should be graphs (relations).
3. Design of the data manipulation, i.e. formulate graph rewrite systems that compute and transform the graphs that were defined in the data model. Build graphs with edge addition rewrite systems (EARS), and transform them via general graph rewrite systems (GRS).
4. Think about the implementation of the graphs. Which algorithms does OPTIMIX generate for a problem and with which graph implementations do these run fastest? Exchange graph implementations (functor calls) accordingly.

1.2 Running OPTIMIX from shell

OPTIMIX can be run as standalone command, or as a filter in a pipe. Thus a previous run of cpp can be used to resolve any conditional #ifdef-commands in a specification.
The options of OPTIMIX are:

Option	Effect
-o *name*	use name for output files
-ff *name*	use name for fSDL flat form file
-silent	be totally silent
-view *name*	use name as view name of the engine
-v1	be a bit verbose (default)
-v2	be fully verbose
-help (-h)	print this message and exit
-v	print version number
-poem	print a poem and exit
	for information/debugging
-nodetypes	print all types of rule test graph nodes (variables)
-RTGpaths	print all paths of path coverings in rule test graphs. Useful for debugging.
-sigs	print all signatures of rules (types of order loop nodes).
-diag *name*	use diagnostic output file *name*
-prio *int*	print test outputs that have priority less than *int*
-write	write internal data structures of ox in ASCII format
-writeRTG	write all rule test graph in VCG format to files
-parser	run only the parser
	for code generation
-nobitsetopt	do not generate bitset optimization
-helpfuns	produce help functions together with other functions
-helpfun *name*	produce help functions in file *name*

Instead of giving the command options on the command line, the user can pass them also via a customization file, `.optimixrc`, which must be located either in the current directory or in the home directory. Each option (maybe also with a value) has to stand on an extra line in the file. Empty lines and lines beginning with `#` are ignored.

Chapter 2

An Optimix specification

2.1 Outline

Note that the grammar parts we give here are not the actual grammar of the parser; they only show the layout of an OPTIMIX specification. The outline of an OPTIMIX-specification is the following:

```
OptimizerSpecification     ::=  GlobalTargetCodeSections
                                [ fSDLImportDecl ] [ InheritanceDeclarations ] GraphRewriteSystems
GlobalTargetCodeSections   ::=  [ 'HFIRST' TargetCode ]
                                [ 'IMPORT' TargetCode ]
                                [ 'EXPORT' TargetCode ]
                                [ 'GLOBAL' TargetCode ]
                                [ 'BEGIN' TargetCode ]
                                [ 'CLOSE' TargetCode ]
GraphRewriteSystems        ::=  [EARS | GRS] *
```

The global target code sections contain code of the target language C (or Smart). The code is copied unchanged to certain parts of the generated files:

HFIRST into .h file; before any code line. Can be used to manipulate inclusions of files

IMPORT into .h file; after the inclusion of stdio.h

EXPORT into .h file; after IMPORT

GLOBAL into .c file; after the prologue

BEGIN into .c file into the begin function `<specfile>_Begin()`

CLOSE into .c file into the close function `<specfile>_Close()`

2.2 Lexical parts

Lexical items of OPTIMIX specifications are the following:

```
String      ::=  ''' any ''' |  '"' any '"'
Digit       ::=  [0-9]
Integer     ::=  Digit +
Ident       ::=  A-z (A-z|Digit)+
Name        ::=  Ident | String
TargetCode  ::=  '{' any '}'
```

Special keywords are:[1]

```
after     any
BEGIN     before
CLOSE     CONSLIST
DAG       DECLARE   DELAYEDREMOVE      DELETE
```

[1] Some of them are not yet used

```
EARS      ears      END       ENDINPUT  EXPORTS
FINER     first     FORALL    FREE      FUNCTION
GLOBAL    GENERIC   GRAPH     GRS        grs
HASH      HEADTAILLIST        HFIRST
IMPORT    IMPORTSDL INDEX     INITIAL
KEY
last
MARK
next      NOT
pred      prev
RANGE     REDUCIBLE REMOVE    RULES
succ
TARGET    THREADED  TREE
VIRTUAL
```

Also the fSDL functors are special keywords which are known to OPTIMIX. These are currently:

```
LIST      DLL       SET       SETF
EGRAPH    SGRAPH    HGRAPH    SEQCLASS
BIPUNI    BITUNI    SETFUNI
```

Delimiter of identifiers (besides white space (space, newline, tab)) are:

```
( ) { } { } . ; : :- >< //  /*  */  (* *) {* *} {| |} {|| ||}
{# #} (| |) -> ~  !~ == < > ! ? :=
<= ==>
```

Comments ending at a newline are started by `//`, other non-nested comments start with `/*` and end with `*/` as in C++. There are also nested comments available as in Modula: `(* stuff *)` It is not allowed to use the string delimiter characters ' and " in comments. The keyword **ENDINPUT** ends the input in a specification file, i.e. all text after it is regarded to be a comment. This is nice for testing; just move text after **ENDINPUT** and OPTIMIX will not see it.
Here we give some syntactical definitions we will need in the following:

```
Type             ::=  Ident
C-Type           ::=  FlatFormType | Ident
Variable         ::=  Ident
GraphName        ::=  Ident
fSDLDomain       ::=  Ident
fSDLOperator     ::=  Ident
fSDLFieldName    ::=  Ident
FlatFormType     ::=  Ident
ActualParameter  ::=  Ident
```

A FlatFormType is a type in C which results from functor flattening. A C-Type is a type which can be understood by the C-Compiler, i.e. a type of the flatform or a normal C type.

2.3 Global declarations

2.3.1 Import a flat form file

Engines which are generated by OPTIMIX are put into COMPARE compilers. For each OPTIMIX specification there must be an import specification of the flat form file of that compiler (`<compiler>.fdl`). The user can also supply a flatform file via option `-ff` (see man-page). If engines are to be reused in several compilers, -ff is the normal way of telling OPTIMIX what the flatform file is. The specification in the file is:

```
fSDLImportDecl   ::=  'IMPORTSDL' [ String ]
```

This declares that OPTIMIX should read an fSDL flat form file with name `String`. fSDL mode is turned on.

2.3.2 Inheritance declarations

```
InheritanceDeclarations   ::=   'FINER' FinerDecl *
FinerDecl                 ::=   Ident // '<' ';'
```

The flatform does not contain the inheritance information of fSDL anymore because domains are flattened. Thus the user can specify inheritance declarations, such that several flatform types (domains and operators) are finer than others. This is sometimes necessary, when the type inference algorithm of OPTIMIX thinks that two types are distinct and not compatible (e.g. mirSimpleSTMT_mirAssign and mirSTMT_mirAssign). This stems from the fact, that the inheritance relation of mirSimpleSTMT and mirSTMT is lost in the flatform. If the user now specifies `FINER mirSimpleSTMT < mirSTMT;`, the type inference algorithm knows that both types are compatible.
Note that finer types stand to the left.

2.4 Available graph implementations (graph functors)

OPTIMIX provides *graph implementation transparency (functor transparency)*. This means, it is transparent from a predicate specification how a graph (a relation/a predicate) is implemented. This means in effect that it is transparent with which kind of functor a graph is implemented (see section 3.1.6). The given predicate name of the specification is used to analyse the functor call (via the flatform) and the code to traverse graphs is generated accordingly.
OPTIMIX supports *functor-created* as well as *hand-crafted* graphs[2] . The supported functors are:

```
Functor                   ::=   HomogeneousGraphFunctor | BipartiteGraphFunctor | SetFunctor
HomogeneousGraphFunctor   ::=   'EGRAPH' | 'SGRAPH' | 'HGRAPH' | 'SEQCLASS'
BipartiteGraphFunctor     ::=   'BIPUNI' |  'BITUNI' | 'SETFUNI'
SetFunctor                ::=   'SET' | 'LIST' | 'SETF' | 'DLL'
```

If graphs are implemented with these functors, you can test whether certain edges exist, and add or delete edges from them. OPTIMIX also understands simple pointer fields. You are allowed to navigate via them by writing down their field name as predicate.

[2] Within COMPARE they were formerly called explicit and implicit graphs

Chapter 3

Specification of graph rewrite systems

This section describes how graphs can be constructed and manipulated by OPTIMIX. OPTIMIX provides two kinds of graph rewrite systems for this: *edge addition rewrite systems (EARS)* and general terminating *graph rewrite systems (GRS)*.
EARS are equivalent to DATALOG with binary predicates [CGT89b] [CGT89a]. Thus we write their rules down like DATALOG rules (similar to Prolog clauses). However, while in DATALOG rule bodies (rule tests) stand on right hand sides, in graphic graph rewrite rules rule tests form left hand sides. In order to avoid confusion in the following we will denote the left hand side in GRS rules and the right hand side of DATALOG rules with *rule test*, whereas we will denote the right hand side of GRS rules and the left hand sides of DATALOG rules by *rule transformation*.
Note that currently the termination of GRS is not checked.

3.1 Program analysis with EARS

OPTIMIX considers program analysis to be graph construction starting from a start graph (axiom). For this OPTIMIX uses edge addition rewrite systems (EARS). They are equivalent to binary DATALOG [Aß94b] [Aß95]. EARS construct graphs by building a relation between one or two node sets (e.g. by working on the node domain of a homogeneous graph or the two node domains of a bipartite graph). Each successful rule application adds one or more edges to the graph (infers and asserts predicates over the nodes). Because EARS are confluent and terminating, the process stops and yields the desired graph. You can also say that EARS have a unique fixpoint.

```
Ears   ::=  ( 'EARS' | 'GRS') Name '(' Parameters ')' RangeDeclarations [ RuleVariableDeclarations ]
            [ Options ] [ BEGINCode ] Rules [ ENDCode ]
            | ( 'EARS' | 'GRS') Name '(' Parameters ')' Stratum *
```

For each EARS one C routine is generated, having the same name. An EARS can consist of one or several rule groups, called *strata*.
One stratum is grouped by `{` and `}` brackets. It consists of several rules. They are either in the style of DATALOG, or specified as graph rewrite rules.

```
Stratum      ::=  '{' RangeDeclarations [ RuleVariableDeclarations ]
                  [ Options ] [ BEGINCode ] Rules [ ENDCode ] '}'
Rules        ::=  'RULES' (EARSFact | EARSRule | GRSRule) *
EARSRule     ::=  [ Options ] [ BEGINCode ] Predicates ':-' Predicates [ ENDCode ] '.'
Predicates   ::=  Predicate // ','
Options      ::=  '[' Name // ',' ']'
```

For each stratum range declarations for variables (nodes) have to be made (section 3.1.1). Also variable declarations (node declarations, section 3.1.3), options (section 3.1.5), BEGIN- and END-Code may be given (section 3.1.4).

Each rule of a stratum leads to the generation of several rule test loops over the nodes of the mentioned graphs. How the rules are evaluated within a strata, is decided by OPTIMIX according to the evaluation strategy for EARS [Aß94b]. The code for the stratas is generated in their source order. Currently EARS(k), $k > 1$ are allowed. [1]
Also note that if a predicate has a left node type A (which refers to some nodes in a graph G) and another predicate refers also to left node type A, then the user must guarantee that these node domains are the same, i.e. that the graphs consist of the same nodes. We call this *equality on graph universes*. We need this restriction that the order algorithm of [Aß94b] works.
An EARS or a stratum is *recursive*, if it defines a predicate (assigns a graph) which is also used (tested). Then the generated code contains a fixpoint loop to detect the fixpoint. For non-recursive EARS or strata no fixpoint loop is generated.

3.1.1 Range declarations

Code generation for EARS relies on the concept of *EARS order*. The order of an EARS is roughly the same as the number of source nodes in rule tests (rule left hand sides) which have different types. These nodes (thes variables) are called *order loop nodes)*. For each order loop node there has to be a *range declaration*, i.e. a declaration to which *range* or *node set* the order loop node is initialized.

```
RangeDeclarations   ::=   'RANGE' RangeDeclaration *
RangeDeclaration    ::=   Variable '<=' GraphName ['.' 'TARGET']
                          | Variable '<=' SetFunctor '(' fSDLDomain ')'
                          | Variable ':' fSDLDomain '><' Variable '<=' SetFunctor '(' fSDLDomain ')'
                          | Variable ':' fSDLDomain
```

Currently order loop nodes can be initialized to three ranges (node sets):

- node sets of graphs.
  Then the order loop range is initialized to the node domain of a (tested or modified) graph. If the modifier `.TARGET` is specified, the target node domain (domain 2) of a bipartite graph functor is taken, otherwise the source domain (domain 1).

- sets.
  If the range is declared to be an application of a set functor, it is assumed that the user wants to hand over a set or list as parameter to the generated routine. This set is then taken to initialize the order loop nodes.

- single source path problem (SSPP) initialization.
  The range of an order loop node can also be only one single parameter object. Then the rule which contains the order loop node is considered to be an SSPP rule with a single source node and a result solution set which contains all nodes that fulfil the SSPP problem (section 3.3). The result set is thus the second part of the declaration. Source node of the SSPP as well as the result set are inserted automatically as parameters of the generated routine.

- single parameters.
  Then the order loop domain is just a variable, which is included automatically in the parameter list of the generated routine.

3.1.2 Parameters of routines in the generated code

For each EARS one C routine with the same name is generated. For these routines OPTIMIX generates parameter lists which consist of three subsets of parameters: explicitly specified parameters, parameters stemming from range declarations and parameters which are graphs that are tested in rule tests or assigned in rule transformations.

[1] However, if $k > 2$, these EARS have not been tested yet. It may be that if the signatures of the rules do not overlap in list form, incorrect code is generated. However, most EARS are have order smaller than 3.

Explicit parameter specification

```
Parameters   ::=   Parameter // ','
Parameters   ::=   Variable ':' C-Type
```

Explicit parameter specifications serve to hand help variables over to the generated routine. They can serve to pass the engine state, or other variables that may be used in target predicates. Their type must be a C-Type (which can also be a FlatFormType).

Parameters stemming from range declarations

Each range declaration (section 3.1.1) delivers one or two parameter declarations for the generated routine.

Parameters stemming from graph usage

Each graph tested or manipulated by a rule must be passed as parameter of the generated routine. However, the user need not provide declarations for these; OPTIMIX automatically generates a correct parameter list. The graph parameter list is ordered alphabetically.
The user has to take care that these parameter graphs are prepared correctly:

- tested graphs must have nodes (and edges if they are not empty)
- if predicates are stated over the same variable, the universes of the corresponding graphs must be the same. Otherwise unexpected results can occur.
- assigned graphs must have their nodes already, i.e. the nodes must have been added to the graph by calling `addnode`-functions of the graph functors. Then the edges are filled in by the generated routine.
- Note that currently it is very simple to add multiple edges between the same nodes in EGRAPHs. Then the result of the generated routine may be unexpected.

3.1.3 Rule variable declarations

OPTIMIX infers types for variables by looking the predicates up as fields in the flatform. Sometimes it is able only to infer a *domain* of a variable (e.g. mirSTMT) whereas at certain points in the code generation also operators are needed (e.g. for generating access functions). In other situations several types are infered for variables. Then the user can help OPTIMIX by giving additional declarations for variables. They hold for all rules of a GRS.

```
RuleVariableDeclarations   ::=   'DECLARE' VariableDeclaration *
VariableDeclaration        ::=   IdentList ':' DomainOperatorSpec ';'
DomainOperatorSpec         ::=   fSDLDomain | fSDLOperator | fSDLDomain '@' fSDLOperator
```

Such a declaration is much like a variable declaration in Modula; however, as type domains and/or operators have to be given. OPTIMIX then incooperates these declarations into his type inference.

3.1.4 BEGIN and END code for strata and rules

Strata as well as rules can be annotated by a BEGIN and an END target code. This code is printed right after the variable declarations for a stratum (rule), or just before the stratum (rule) end, respectively.

```
BEGINCode   ::=   'BEGIN' TargetPredicate
ENDCode     ::=   'END' TargetPredicate
```

3.1.5 Options for strata and GRS rules

EARS, strata and rules may also be annotated with an option list (*options*). This is a list of strings, enclosed in square brackets `[]`. If such a property is set, the semantical analysis, optimization and code generation phases of OPTIMIX are steered in a rule specific way.
Current available rule options are:

- **JOIN** Use join code generation mode, even if on-the-fly was analysed.
- **LocalTests** Perform the pattern matching on a node always, if an instance of the node is traversed. This option results in more pattern matching tests, but fewer traversals, because the join search space of path problems is diminished.

3.1.6 Different kinds of usable predicates in rules

A rule in an OPTIMIX specification contains a number of predicates, which can be of different forms:

```
Predicate            ::=  PredicateName '(' Pattern ',' Pattern ')'
                          | 'FORALL' Variable ':' Predicate
                          | 'NOT' Predicate
                          | '?' ProcedureCall
                          | TargetPredicate
                          | SmartTargetPredicate
                          | TargetCodeLine
                          | PatternMatchStatement
                          | EqualityTest
PredicateName        ::=  Ident [ '@' DomainOperatorSpec2] [ '.' GraphFieldModifier] [ '.' OrderIndicator ]
DomainOperatorSpec2  ::=  fSDLDomain '@' fSDLOperator
```

Simple predicates

Simple predicates are always binary because they refer to graphs. Simple predicates contain patterns or variables as arguments.
Predicate names must exist as the name of a field in an operator in a certain domain. The predicate

```
.. :-    p(X,Y),..
```

is true if the object `Y` is contained in the set `X.p`. Also (in the generated code) the predicate `p(X,Y)` delivers all objects `Y` which are linked under field (or graph) `p` to object `X`.
In fSDL mode a predicate in a rule test or rule transformation refers to

- a field which has the type of a graph functor call (graph field)
- a field which has the type of a set/list functor call (set field)
- a field which has the type of a simple domain (non-graph) (pointer field).

Type inference

OPTIMIX looks up the field name in the flatform and annotates with the predicate a set of types (operator/domain pairs). This is a set of types because a field can turn up in a lot of operators and again these are in a lot of domains.
These sets of alternative types are then intersected and unified against each other during the ongoing type inference. OPTIMIX always tries to retain *finer* types, i.e. more specific types, which then provides better information for code generation. The rules according two types are compared are the following:

- an operator is finer than a containing domain.
- an operator/domain pair is finer than the domain.

- a domain is finer than another if it has been declared so in an FINER inheritance declaration.

At the end of the type inference process there should be unique types for all variables in rules. If not, OPTIMIX will prompt an error. Either this is a real typing error or the user can give more type information to OPTIMIX by providing inheritance declarations (section 2.3.2) or variable declarations (section 3.1.3).
However, this scheme currently has one restriction. If a field is contained in several operators, and is not a shared field, then the user has to specify with the field a domain/operator specification. E.g. in the CCMIR the field `Then` occurs in operator `mirIf` as well as in `mirTryAcquire`. A predicate using it in domain `mirIf` should look like:

```
Then@mirSimpleSTMT@mirIf( Stmt, ThenPart)
```

If the field is a shared field between all operators that use it, the field alone is sufficient as predicate name.

Graph field modifiers

```
GraphFieldModifier  ::=  'succ' | 'pred'
```

From OPTIMIX's point of view a functor-created graph defines two default fields for the parameter domains of the functor application. These two default fields can be used as predicates in clauses. For instance, if the field name of the graph in `mirProcBody` is `BlockGraph` (in a field definition like `Procedure < BlockGraph: EGRAPH(mirBasisBlock,EgraphEdge) >;`) and the node domain/operator is `mirBasicBlock`, then for each `mirBasicBlock` two default fields `BlockGraph` and `BlockGraph.pred` are virtually created. These field names denote all successor resp. predecessors of a `mirBasicBlock` concerning the functor-created graph `BlockGraph`.
`pred` is an example of an fSDL field modifier. It serves to indicate which kind of DMCP calls should be generated in the code. With `p.succ` or `p` the successor relation of graph `p` is denoted, with `p.pred` the predecessor relation is denoted.

Order indicators

```
OrderIndicator  ::=  'first' | 'last' | 'next' | 'prev' | 'before' | 'after' | 'any'
```

If users specifies predicates that refer to fields of LIST-functor type, these neighbor sets are ordered. Then special *order indicators* can be used to find out certain special elements of the list, e.g. the first or last element, or the next or previous element.
The order indicators refer to fields which have an ordered set functor type (like LIST) and generate a loop over a specific element of a list or an access to a specific list element. There are the following types, exemplified by the statements of a block:

- `Stmts.after(Block,S1,S2)` generates a loop over all successors `S2` of `S1` in the `Stmts` of `Block`.
- `Stmts.before(Block,S1,S2)` generates a loop over all predecessors `S2` of `S1` in the `Stmts` of `Block`.
- `Stmts.next(Block,S1,S2)` generates an access to the successor `S2` of `S1` in the `Stmts` of `Block`.
- `Stmts.prev(Block,S1,S2)` generates an access to the predecessor `S2` of `S1` in the `Stmts` of `Block`.
- `Stmts.first(Block,S1)` generates an access to the first element `S1` in the `Stmts` of `Block`.
- `Stmts.last(Block,S1)` generates an access to the last element `S1` in the `Stmts` of `Block`.
- `Stmts.any(Block,S1)` generates an access to an arbitrary element `S1` in the `Stmts` of `Block`. For sets this is chosen by the choose-function of the set functor; for lists the head of the list is taken.

`Block` must be an already known variable.
Note that the arity of several ordered predicates is 3.

All quantified predicates

Normally all predicates are existentially quantified in their variables. However, one predicate in a rule is allowed to be preceeded by an all quantifier, e.g. `FORALL V: p(X, V)`, where the variable `V` must be the right variable of the predicate. Predicates in the head of a rule cannot be allquantified.
However, currently the concept of allquantifiers is rather restricted. Actually it works only in two situations:

- The allquantified variable is the middle variable of a path with two predicates and the rule test is a single path. This is the standard situation for MUST dataflow analyses.
- The allquantified variable is the sink of an RTG. This RTG must also be either a path or a dag (see example engine copyprop).

Negated predicates

If a predicate is preceeded by a `NOT`, it is negated. Negation is allowed in the following contexts:

- In rule transformations. Then in the code an item of the denoted graph is deleted, not added.
- In rule tests if predicates are used that are graph functor instantiations. Negation can only be performed if a universe is known against the completion of a set of nodes is performed. This is the case only for graph functors, where the set of graph nodes represents this universe.

  Negation is performed by a loop over the universe, skipping those nodes which are the neighbor set of the predicate.
- In rule tests for bitset predicates. They also have a universe which consists of all nodes the bits refer to. Negation is performed by a bitset complement.

Checked calls to external predicate functions

```
ProcedureCall   ::=   Ident '(' ActualParameter // ',' [ '==>' ActualParameter // ',' ] ')'
```

If a predicate starts with a `?`, then OPTIMIX assumes that the rest is a call to a C function returning a boolean. Thus it generates this call and checks its result with TRUE. If the called predicate fails, also the rule fails. Otherwise the rule test is continued.
The list of actual parameters to a call must consist of simple variables. There is an IN parameter list (before the `==>`) and an OUT parameter list (after the `==>`). The IN parameters are considered to be pattern variables which are handed over to the called routine. The OUT parameters are also handed over as reference parameters, i.e. their addresses are handed over.

Target code predicates

```
TargetPredicate       ::=   '{*' any '*}'
SmartTargetPredicate  ::=   '{||' any '||}'
```

It is possible to specify C and Smart target code in the place of a predicate. This code is copied unchanged to the generated file. If the code is embraced by Smart-like alternative brackes (`{||` and `||}`) the Smart code is also copied unchanged except that an new alternative is appended at the outer level (`{|` and `|}` are reserved for future use). By this Smart code never fails on outer level and at least enters this last alternative. In this alternative a `continue` statement is encountered, which then continues the next loop iteration of OPTIMIX-generated code. Otherwise, if a Smart pattern match would fail, it would never return to the OPTIMIX-generated code. If the user wants a different behaviour than continue, he must give a last default alternative himself.
Target predicates normally are attached to their preceeding predicates and thus are copied after the code that was generated for that predicate, i.e. normally in a loop which was caused by that predicate. If a target

predicate appears as first predicate of a rule test part, it is copied at the beginning of the rule test, right after the declaration of all rule test variables. Thus the user can define his own variables for use in target predicates.
If a target code predicate appears in the rule transformation of a rule it is printed after the addition/deletion of the preceeding edge in the innermost rule test loop. If it appears as first predicate in a rule transformation, it is printed at every addition/deletion of an edge.

Target code lines

If a target predicate consists of one line of C code there is a special syntactic alternative for it. Target code lines consist of arbitrary C text, terminated by a newline character. The newline also simulates a ',' token to the parser, so that within predicate lists no additional commas are necessary.

```
TargetCodeLine   ::=   '*' any Newline
```

A very nice use for target predicates is the test and set of node attributes or the printing of debug information. E.g. the following rule tests whether a node is marked as deleted and removes it from some graphs:

```
GRS DeleteFromStatementLists()
{
   RANGE Proc <= mirProcGlobal >< list_of_definitions: SET(mirSTMT);
   RULES
      {* /* This target code is printed after the definition */
         /* of rule test variables                          */ *},
      Body(Proc,PBody),
      LinearBlocks(PBody,B),
      * /* and this here is a single line of target code */
      Stmts(B,Ass),
      Ass ~ mirAssign{},
      {* /* target predicate to test, whether a node was really deleted */
         /* is copied to the rule test after the pattern match on mirAssign */
         if (!mirSimpleSTMT_mirAssign_get_deleted(Ass))
            continue;
      *}
   ==>
   DELETE Ass FREE;        // really deallocate Ass
   ==>
   * printf("deleting copy statement %s",mirSTMT_provide_label(Ass));
   NOT Stmts(B,Ass), NOT list_of_definitions(Proc,Ass),
   .
}
```

Pattern match statements

```
PatternMatchStatement   ::=   Variable ('~' | '!~') Pattern
```

Instead of a predicate also pattern match statements on rule test nodes (rule test variables) are allowed. If a variable is linked to a pattern with ~ (see 3.1.7) this pattern match statement succeeds if the variable has the form of the pattern. If a variable is linked to a pattern with !~ the pattern match statement succeeds if the variable has not the form of the pattern.

Equality tests

```
EqualityTest            ::=   PatternVarEqualityTest | RTGNodeEqualityTest
PatternVarEqualityTest  ::=   Variable BoolOp Variable
RTGNodeEqualityTest     ::=   Variable EqualOp Variable
BoolOp                  ::=   EqualOp | '<' | '>' | '<=' | '>='
EqualOp                 ::=   '==' | '!='
```

Apart from pattern matching equality tests on pattern variables or on rule test graph nodes are allowed. In the first case they lead to the generation of equality/inequality functions of the opaque types of the attributes, in the second case `DMCP_equal` is used.

3.1.7 Patterns

In predicates of rule tests[2] or in pattern match statements patterns may appear.

```
Pattern        ::=   OperatorDomainSpec
                   | OperatorDomainSpec
                   | OperatorDomainSpec '{' InnerPattern // ',' '}'
InnerPattern   ::=   fSDLField '=>' InnerPattern
                   | fSDLField '=>' Variable ('~' | '!~')  InnerPattern
                   | fSDLField '=>' Variable
InnerPattern   ::=
```

Variables in patterns are arbitrary identifiers, contary to Prolog, where each variable has to begin with a capital letter. An OPTIMIX pattern match is similar to a Smart pattern match: it compares a structure with a term pattern. There are two kinds of patterns: outer patterns are allowed in pattern match statements, where they match already defined variables. They are also allowed in left or right parameters of simple predicates, however, only at the outer level.
Inner patterns are allowed to occur only in an outer pattern or in another inner pattern. They perform field pattern matching and also variable assignment[3] . Because (due to the fSDL domain calculus) no order is defined on the fields of an fSDL operator, no positional pattern matching is possible, only matching with a field name is allowed. Variable assignment assigns a variable to the field, if the pattern match was successful. If no variable assignment is given, OPTIMIX assigns a temporary variable to the successfully matched subtree.
For instance, the pattern match

```
S ~ mirIf{Then => A ~ mirAssign{}}
```

tests whether a variable S consists of a `mirIf` where the field `Then` is a `mirAssign`. The variable `A` is assigned to the assignment statement.
Note that the variables which are defined in patterns are not allowed to be used for further navigation, only for the use in target predicates, e.g. to test attributes. This is a restriction of the current implementation.

3.2 Non-ground fact specification

```
EARSFact   ::=   [ Options ] [ BEGINCode ] Predicates [ ENDCode ] '.'
```

In OXDML non-ground facts may be specified analogously to Coral [RSS92]. Non-ground facts are facts that contain variables. Non-ground facts in a stratum are always evaluated before other rules of the it are evaluated. Non-ground facts serve to initialize a graph with certain values before other rules manipulate the graph. This can be used especially for data flow analysis: the initialization statements there are non-ground facts. As example consider the specification of available expression dataflow analysis, the first two rules are non-ground facts:

```
// Find available expressions
EARS AvailableExpressions ()
{
   RANGE b <= AVIN; e <= AVIN.TARGET;

   AVIN(b,e).    // non-ground facts: initiallization to FULL set.
   AVOUT(b,e).
   // EARS rules.
```

[2] In rule transformations pattern matching is not allowed.
[3] This is new compared to Smart

```
   AVIN(b,e)  :- FORALL p: Blocks.pred(b,p), AVOUT(p,e).
   AVOUT(b,e) :- COMPOUT(b,e).
   AVOUT(b,e) :- TRANSP(b,e), AVIN(b,e).
}
```

Rules that contain empty rule tests (containing only a single target predicate) are also considered facts. Thus the following two rules are equivalent:

```
AVIN(b,e).
AVIN(b,e) :- {* blabla *}.
```

However, if pattern matching or other predicates occur in the rule test, the rule is not considered a fact.
Also *self-edge facts* may be specified which draw self edges on nodes:

```
EARS ComputeDominators()
{
   RANGE b: mirBasicBlock;
   Dominators(b,b).    // self-edge fact: each block is dominated by itself
   ..
}
```

Non-ground facts also may be negated. Then OPTIMIX generates loops over the graph nodes that delete edges which might have existed earlier.
There are all in all several possibilities, how to initialize a graph:

- make a full graph with a non-ground fact.
- make a graph with self edges with a self-edge fact.
- delete all edges in a graph by a negated fact.
- delete all self edges by a negated self-edge fact.

Before and after facts target predicates can be written. If a target predicate is written before the fact, it is copied directly before the edge addition. If it is written after the fact, it is copied direcly after the edge addition.

3.3 Single source path problems (SSPPs)

There is a special variant of EARS which can solve single source path problems (SSPPs) [Tar81]. An SSPP is a path problem in a graph which is described by a path expression (or a set of predicates, like in EARS) and which is applied to *one single* source node of the graph. It delivers all nodes which are reachable from the source node under the predicates (the path expression). These nodes are called *result set*.
EARS can contain several SSPP rules. The order loop node (the source node of the SSPP) and the result set of such a rule have to be declared with an SSPP range declaration (section 3.1.1). The node is then initialized to the corresponding parameter of the generated routine, and the parameter set of the range declaration is used as the result set.
SSPP rule tests are applied to the start graph starting with one node. They are not printed among those rule tests which result from normal rules (in the order loops). Instead they are extracted and printed after them[4] .
The following example solves a SSPP for a procedure and all its statements. It collects all assignments that are in the blocks' statement lists.

```
EARS PrepareReachingDefinitions()
{
   RANGE Proc <= mirProcGlobal >< list_of_definitions: SET(mirSTMT);
   RULES
   list_of_definitions(Proc,Ass) :-
      Body(Proc,PBody),
      LinearBlocks(PBody,B),
```

[4] This may change.

```
        Stmts(B,Ass),
        Ass ~ mirAssign{}
    .
}
```

SSPP rules can also be used nicely to write down walking e.g. over statement lists and perform actions on them. We can easily add to the end of the rule a target predicate that performs a side effect (here adds the assignment statement to a globall class of definitions for objects). This global class is attached to the state handle of the engine, which must then be passed as parameter to the generated routine `PrepareReachingDefinitions`.

```
EARS PrepareReachingDefinitions(state: reachingdefsStateType)
{
   RANGE Proc <= mirProcGlobal >< list_of_definitions: SET(mirSTMT);
   RULES
   list_of_definitions(Proc,Ass) :-
      Body(Proc,PBody),
      LinearBlocks(PBody,B),
      Stmts(B,Ass),
      Ass ~ mirAssign{},
      {* EnterInDefinitionClasses(state,Ass); *}
   .
}
```

3.4 Program transformation with graph rewrite systems

Rules of a GRS may be specified in a similar way to an EARS rule, however, they have an additional transformational part. This transformational part consists of node deletions, node additions, edge deletions and edge additions, also to the newly created nodes.

```
GRSRule   ::=   [ Options ] [ BEGINCode ] RuleTest '==>'
                  [ [ NodesToBeDeleted ] [ NodesToBeAdded ] '==>' ] Predicates [ ENDCode ] '.'
                | [ Options ] [ BEGINCode ] RuleTest '==>' Predicates [ ENDCode ] '.'
```

Strata, strata options, rule options, BEGIN- and END-Code behave in the same way as with EARS rules. Note that the user himself has to guarantee the termination of a GRS. There is no automatic check for that, neither a test for confluence. See also the article [Aß94a].

3.4.1 Node deletion

```
NodesToBeDeleted   ::=   'DELETE' IdentList DeleteProperty *
DeleteProperty     ::=   'MARK' | 'FREE' | 'REMOVE' | 'DELAYEDREMOVE'
```

Nodes from the rule test which have to be deleted are specified after the keyword `DELETE`. The deletion can be done in four modes, which can be combined, e.g. it is possible to specify `MARK REMOVE` with some nodes. The *mark mode* just markes the nodes, which are in a successfully matched redex, by setting the field `deleted`. This is a field which the user has to add to all domains of objects which have to be deleted. Once the nodes are marked like this, they can be recognized as being invalid. Marking is necessary when a node belongs to a lot of graphs, not only those that were tested in the rule. Then subsequent passes over these graphs can remove all incident edges, and in the last pass also the node can be deallocated.
The *remove mode* does not deallocate the nodes but only removes the node from all there containing graphs concerning the rule test. Thus it deletes all incident edges of graphs of the rule test. There still might be other graphs the node is in.
The *delayed-remove mode* is special. It generates a second, artificial EARS, only containing the rule in question. This rule walks the graphs of the rule test a second time, tests on deleted (marked) nodes and then performs removal of incident edges. The walking is done via ITERLIST-LOOPs, not with LIST-LOOPs.

This was due to an early restriction of the LIST functor which could not delete nodes from lists when walking the lists themselvs via LIST-LOOP.[5]
The *free mode* really deallocates the nodes, i.e. calls `DMCP_delete`.

3.4.2 Node addition

```
NodesToBeAdded  ::=  'ADD' VariableDeclarations
```

Nodes which are added by the rule, have to be declared in a similar way as rule local variable declarations. However, it is necessary to specify domains and operators for new nodes, otherwise the correct node allocation function call cannot be generated.

3.4.3 Addition of edges to new nodes

The part following the `ADD` declaration consists again of a sequence of predicates, which specifies edge additions and deletions. Edge additions can refer to new nodes as well as to old nodes; edge deletions can of course only refer to items from the rule test.

[5] It may be that in the current version this is obsolete; so also delayed-remove mode is obsolete. This needs further testing.

Chapter 4

Examples and Miscellaneous

4.1 Examples

Within COMPARE several example engines have been developed as OPTIMIX applications:

- reachdef: compute reaching definitions on CCMIR.
- livecopies: compute live copy statements.
- copyprop: do copy propagation on copy statements.
- exprtab: do value numbering and global structural equivalence on mirEXPR.

Here we will present only some other short examples.
We assume a basic block graph is defined in a procedure as follows (this can be done in a view specification of an engine). We assume a edge type which carries an integer and that the basic block graph has already been constructed (can be done with an EARS(2)).

```
domain EgraphEdge: {  EgraphEdge < value: INT > };
domain mirProcBody <
      Blocks:          EGRAPH(mirBasicBlock,EgraphEdge),
      ReverseBlocks:   EGRAPH(mirBasicBlock,EgraphEdge),
      ReachableBlocks: EGRAPH(mirBasicBlock,EgraphEdge),
      Dominators:      SGRAPH(mirBasicBlock),
      SelfDom:         SGRAPH(mirBasicBlock),
      USED:            BIPUNI(mirBasicBlock,mirLocal),
      Livein:          BITUNI(mirBasicBlock,mirLocal,Livein),
      Liveout:         BITUNI(mirBasicBlock,mirLocal,Liveout)
   >;
```

and the flat form file is `example.fdl`, then we may write the following specifications.

```
IMPORTSDL "example.fdl"

/* Compute the inverse of the basic block */
EARS ComputeReverse()
{
   RANGE b <= Blocks;
   RULES
      ReverseBlocks(b,b1) :- Blocks(b1,b).
}
```

This EARS of order 1 just builds up the reverse basic block graph, all edges in the new graph `ReverseBlocks` are inverted. The range declaration tells that the order loop variable is to be initialized from the node domain of graph `Blocks`. The following shows how the generated routine may be called from C code:

```
Blocks = mirProcBody_get_Blocks(procbody);
mirProcBody_set_ReverseBlocks(EGRAPH_mirBasicBlock_EgraphEdge_create());
ReverseBlocks = mirProcBody_get_ReverseBlocks(procbody);
CopyNodes (Blocks, ReverseBlocks);  // copies the nodes of the EGRAPH
ComputeReverse (Blocks,  ReverseBlocks);
```

The order of the parameter graphs to `ComputeReverse` is alphabetically.
The next example computes the initialization of a dominator analysis. Here all nodes initially dominate all others except that the entry node does not dominate anyone.

```
EARS DominatorInit()
{
   RANGE b: Dominators; b1: Dominators;
   RULES

   Dominators(b,b1) :- Blocks.pred(b,PredecessorBlock).
                       // initially a node dominates each other node.
                       // The dominators of the entry node, however, are left empty.
   SelfDom(b,b).       // this predicate is used for adding each node to a
                       // set Dominators during the processing in ComputeDominators
}
```

Then the final dominator analysis can be called, which is described by the following EARS.

```
EARS ComputeDominators()
{
   RANGE b <= Blocks;
   RULES

   // a node dominates another if all predecessors dominate the other
   Dominators(b,b1) :- FORALL p: Blocks.pred(b,p), Dominators(p,b1).
   Dominators(b,b1) :- SelfDom(b,b1).
}
```

`Blocks.pred(b,p)` denotes all predecessors of `b` in the graph `Blocks`. For these `p` also the dominator relation to `b1` must hold. Note that `b` and `b1` are existentially quantified variables while `p` is allquantified. The rule with predicate `SelfDom` is necessary because currently additions of single nodes to sets (in clauses) is not possible, everything has to be expressed in terms of edges (predicates).
Note that OPTIMIX provides *functor transparency*, i.e. it is transparent which functors have been used to implement the graphs. This is automatically infered from the flatform. The code for the graph navigations (functor method calls, access function calls) is generated accordingly.
The call sequence in a calling program could be:

```
mirProcBody_set_Dominators(EGRAPH_mirBasicBlock_EgraphEdge_create());
Dominators = mirProcBody_get_Dominators();
CopyNodes (Blocks, Dominators);
DominatorInit(Dominators, SelfDom);
ComputeDominators(Dominators, SelfDom, Blocks);
```

We also can specify `DominatorInit` and `ComputeDominators` together; the non-ground facts are always computed first. Alternatively you can also use two strata.
Last example is a transitive closure over basic blocks. This time we have specified the field modifier `succ` explicitly; it can be left out.

```
EARS ComputeReachableBlocks()
{
   RANGE b <= Blocks;
   RULES

   ReachableBlocks(b,b1) :- Blocks.succ(b,b1).
   ReachableBlocks(b,b1) :- Blocks.succ(b,p), ReachableBlocks(p,b1).
}
```

4.1.1 Live Variables: MAY dataflow analysis

It is also possible to specify MAY data flow analysis. For that we need a bipartite graph functor: BIPUNI. It serves to represent the information which variables live at which basic block, here at which entry and exit of

which block (LIVEIN, LIVEOUT). We also need the information per each basic block, which local variables have been used in a basic block (USED).

```
EARS LiveVariables()
{
   RANGE b <= LIVEOUT;
   RULES

   LIVEOUT(b,o) :- Blocks.succ(b,b1), LIVEIN(b1,o).
   LIVEIN(b,o)  :- USED(b,o).
   LIVEIN(b,o)  :- LIVEOUT(b,o).
}
```

A variable is live at the entry of a block, if it is used in the block, or if it is live at the exit of the block. A variable is live a the exit of the block, if it lives at the entry of a successor block.

4.1.2 BusyVariables: MUST dataflow analysis

If we want to solve a MUST dfa (intersection over all predecessors), we have to use an all quantifier. The following EARS computes busy local variables, e.g. variables that are used in all successor blocks or ar used in the block itself. The change is minimal.

```
EARS BusyVariables()
{
   RANGE b <= BUSYIN;
   RULES
   BUSYOUT(b,o) :- FORALL b1: Blocks.succ(b,b1), BUSYIN(b1,o).
   BUSYIN(b,o)  :- USED(b,o).
   BUSYIN(b,o)  :- BUSYOUT(b,o).
}
```

In similar fashion available expressions or busy expressions can be solved.

4.2 The generated code

Manipulation and debugging of the generated code

We have tried to make the generated code as readable as possible. We hope users are able to read it and also make modifications. One can use optimix to get a skeleton for one's algorithm and then modify and refine it by hand. A lot of typing can be avoided in this respect.
Note that RCS and SCCS ids are already generated, so that files directly can be imported under change control.
OPTIMIX generates some test code which is dependent on the flag `OXDEBUG`. If you set this manually in the code or set the `-D` flag during a make, the running code will produce some test output. Users also can insert target predicates with print-statements and #ifdef-switches in order to print debug information.
However, the actual printing of the test output is dependent on the value of some option/variable of/in the engine. This is

- **-DUSE_SEQPAR_COSY** If this compilation switch is set (within COMPARE CoSy), then a query in the option database of the engine is done for the string `"oxdebug"`. Thus, if the engine has got the option `"oxdebug"`, then test output is printed.

  In order to test the option, optimix generates a call to `engineStateGet`, which delivers the engine state. It is assumed to have a field `options`, which contains the engine option database. Thus the query is

  ```
  if (engineStateGet->options != NULL)
     /* test output */
  ```

  Note that users *must* save the options into the engine state at engine initialization.

- **-UUSE_SEQPAR_COSY** If this compilation switch is not set then the global variable `int oxdebug;` is queried if output is to be printed. Note that this is useful within CoSy only if everything is clustered into one process.

  There is a second test print system which works in the same way. However, it prints less test output and is dependent on the engine option `"oxblip"`, or the global variable `int oxblip;`, respectively.

Unknown types in the generated code

Navigations in the generated code OPTIMIX need that some internal variables are defined. Sometimes their types are not known when the user compiles a generated file. This is often the case e.g. for `SET_mirBasicBlock`, because the functor application `SET(mirBasicBlock)` does not appear in the CCMIR. However, the generated file needs definitions of this type, because sets of mirBasicBlock are constructed during the navigations. The solution is that the user has to instruct fsdc to generate the domain with a `use`-clause. Alternatively in some operator a dummy definition for `SET(mirBasicBlock)` can be introduced so that the fsdc generates this type as a result of this functor call.

4.3 Frequently asked questions

Q: For a certain variable OPTIMIX infers 'type mismatched', i.e. multiple types. What can I do?
A: There are several reasons for this. Reason 1: OPTIMIX infers two domains that are compatible in fSDL, but not in the flat form anymore, because in there the inheritance information of fSDL is lost. Then state a FINER assertion that one domain is finer than the other and it should work.
Reason 2: There are really two different domains/types. Then help OPTIMIX by stating a type for that variable. You can do this either by a DECLARE variable declaration, which holds for all rules of an EARS. Or you can introduce pattern matching statements, whose type informations are then exploited for the type check. Or you qualify a predicate name by a domain/operator specification.
Reason 3: It really was a flaw in your specification. Look into the flatform file, which types occur for your fields.
Q: My specification results in a larger order for my EARS than expected.
A: Maybe OPTIMIX has infered domains for the types of the source nodes of the rules which are different, however are compatible according to the domain calculus. Then insert a FINER statement at the beginning of the specification to tell OPTIMIX that two domains are compatible. OPTIMIX will then choose the coarser domain as type of the source node.
A: Maybe your FINER specification must be more detailed. Currently there is no union over different FINER specifications which contain the same tails. Be sure that you really specify all finer domains of a domain in one line.
Q: I try to compile the generated engine with `-DOXDEBUG`. However, it does not compile, because the `oxdebug` field is unknown.
A: In order to use `OXDEBUG` you have to annotate the engine's state struct with a field `int state->oxdebug`. OPTIMIX-generated code then compares `engineStateGet->oxdebug` with the value of the given command-line option option. If the state does not have such a field, the engine does not compile. Also do not forget to save the value of the command-line option `oxdebug` in the state.

Bibliography

[Aß94a] Uwe Aßmann. Program Optimization with Congruent and Stratifiable Graph Rewrite Systems. Technical report, University Karlsruhe, COMPARE consortium, internal paper, 1994.

[Aß94b] Aßmann, Uwe. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In J. Cuny, editor, *5th Workshop on Graph Grammars and Their Application To Computer Science*, Nov 1994.

[Aß95] Uwe Aßmann. *Generierung von Programmoptimierungen mit Graphersetzungssystemen.* PhD thesis, Universität Karlsruhe, Kaiserstr. 12, 7500 Karlsruhe, Germany, July 1995.

[Buh95] Claus-Thomas Buhl. fSDL Language Report. Technical report, COMPARE Consortium, 1995.

[CGT89a] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer Verlag, 1989.

[CGT89b] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge And Data Engineering*, 1(1):146–166, March 1989.

[RSS92] R. Ramakrishnan, D Srivastava, and S. Sudarshan. CORAL - Control, Relations and Logic. In *Proceedings of the 18th VLDB Conference*, 1992.

[Tar81] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.

Syntax

```
OptimizerSpecification     ::=   GlobalTargetCodeSections
                                 [ fSDLImportDecl ] [ InheritanceDeclarations ] GraphRewriteSystems
GlobalTargetCodeSections   ::=   [ 'HFIRST' TargetCode ]
                                 [ 'IMPORT' TargetCode ]
                                 [ 'EXPORT' TargetCode ]
                                 [ 'GLOBAL' TargetCode ]
                                 [ 'BEGIN' TargetCode ]
                                 [ 'CLOSE' TargetCode ]
GraphRewriteSystems        ::=   [EARS | GRS] *

String       ::=   ''' any ''' |  '"' any '"'
Digit        ::=   [0-9]
Integer      ::=   Digit +
Ident        ::=   A-z (A-z|Digit)+
Name         ::=   Ident | String
TargetCode   ::=   '{' any '}'

Type              ::=   Ident
C-Type            ::=   FlatFormType | Ident
Variable          ::=   Ident
GraphName         ::=   Ident
fSDLDomain        ::=   Ident
fSDLOperator      ::=   Ident
fSDLFieldName     ::=   Ident
FlatFormType      ::=   Ident
ActualParameter   ::=   Ident

fSDLImportDecl   ::=   'IMPORTSDL' [ String ]

InheritanceDeclarations   ::=   'FINER' FinerDecl *
FinerDecl                 ::=   Ident // '<' ';'

Functor                   ::=   HomogeneousGraphFunctor | BipartiteGraphFunctor | SetFunctor
HomogeneousGraphFunctor   ::=   'EGRAPH' | 'SGRAPH' | 'HGRAPH' | 'SEQCLASS'
BipartiteGraphFunctor     ::=   'BIPUNI' |  'BITUNI' | 'SETFUNI'
SetFunctor                ::=   'SET' | 'LIST' | 'SETF' | 'DLL'

Ears   ::=   ( 'EARS' | 'GRS') Name '(' Parameters ')' RangeDeclarations [ RuleVariableDeclarations ]
             [ Options ] [ BEGINCode ] Rules [ ENDCode ]
             | ( 'EARS' | 'GRS') Name '(' Parameters ')' Stratum *

Stratum      ::=   '{' RangeDeclarations [ RuleVariableDeclarations ]
                   [ Options ] [ BEGINCode ] Rules [ ENDCode ] '}'
Rules        ::=   'RULES' (EARSFact | EARSRule | GRSRule) *
EARSRule     ::=   [ Options ] [ BEGINCode ] Predicates ':-' Predicates [ ENDCode ] '.'
Predicates   ::=   Predicate // ','
Options      ::=   '[' Name // ',' ']'
```

```
RangeDeclarations   ::=   'RANGE' RangeDeclaration *
RangeDeclaration    ::=   Variable '<=' GraphName ['.' 'TARGET']
                        | Variable '<=' SetFunctor '(' fSDLDomain ')'
                        | Variable ':' fSDLDomain '><' Variable '<=' SetFunctor '(' fSDLDomain ')'
                        | Variable ':' fSDLDomain

Parameters   ::=   Parameter // ','
Parameters   ::=   Variable ':' C-Type

RuleVariableDeclarations   ::=   'DECLARE' VariableDeclaration *
VariableDeclaration        ::=   IdentList ':' DomainOperatorSpec ';'
DomainOperatorSpec         ::=   fSDLDomain | fSDLOperator | fSDLDomain '@' fSDLOperator

BEGINCode   ::=   'BEGIN' TargetPredicate
ENDCode     ::=   'END' TargetPredicate

Predicate            ::=   PredicateName '(' Pattern ',' Pattern ')'
                         | 'FORALL' Variable ':' Predicate
                         | 'NOT' Predicate
                         | '?' ProcedureCall
                         | TargetPredicate
                         | SmartTargetPredicate
                         | TargetCodeLine
                         | PatternMatchStatement
                         | EqualityTest
PredicateName         ::=   Ident [ '@' DomainOperatorSpec2] [ '.' GraphFieldModifier] [ '.' OrderIndicator ]
DomainOperatorSpec2   ::=   fSDLDomain '@' fSDLOperator

GraphFieldModifier   ::=   'succ' | 'pred'

OrderIndicator   ::=   'first' | 'last' | 'next' | 'prev' | 'before' | 'after' | 'any'

ProcedureCall   ::=   Ident '(' ActualParameter // ',' [ '==>' ActualParameter // ',' ] ')'

TargetPredicate        ::=   '{*' any '*}'
SmartTargetPredicate   ::=   '{||' any '||}'

TargetCodeLine   ::=   '*' any Newline

PatternMatchStatement   ::=   Variable (' ' | '! ') Pattern

EqualityTest            ::=   PatternVarEqualityTest | RTGNodeEqualityTest
PatternVarEqualityTest   ::=   Variable BoolOp Variable
RTGNodeEqualityTest      ::=   Variable EqualOp Variable
BoolOp                   ::=   EqualOp | '<' | '>' | '<=' | '>='
EqualOp                  ::=   '==' | '!='

Pattern       ::=   OperatorDomainSpec
                  | OperatorDomainSpec
                  | OperatorDomainSpec '{' InnerPattern // ',' '}'
InnerPattern   ::=   fSDLField '=>' InnerPattern
                  | fSDLField '=>' Variable (' ' | '! ')  InnerPattern
                  | fSDLField '=>' Variable
InnerPattern   ::=

EARSFact   ::=   [ Options ] [ BEGINCode ] Predicates [ ENDCode ] '.'

GRSRule   ::=   [ Options ] [ BEGINCode ] RuleTest '==>'
                  [ [ NodesToBeDeleted ] [ NodesToBeAdded ] '==>' ] Predicates [ ENDCode ] '.'
                | [ Options ] [ BEGINCode ] RuleTest '==>' Predicates [ ENDCode ] '.'
```

```
NodesToBeDeleted   ::=   'DELETE' IdentList DeleteProperty *
DeleteProperty     ::=   'MARK' | 'FREE' | 'REMOVE' | 'DELAYEDREMOVE'

NodesToBeAdded   ::=   'ADD' VariableDeclarations
```