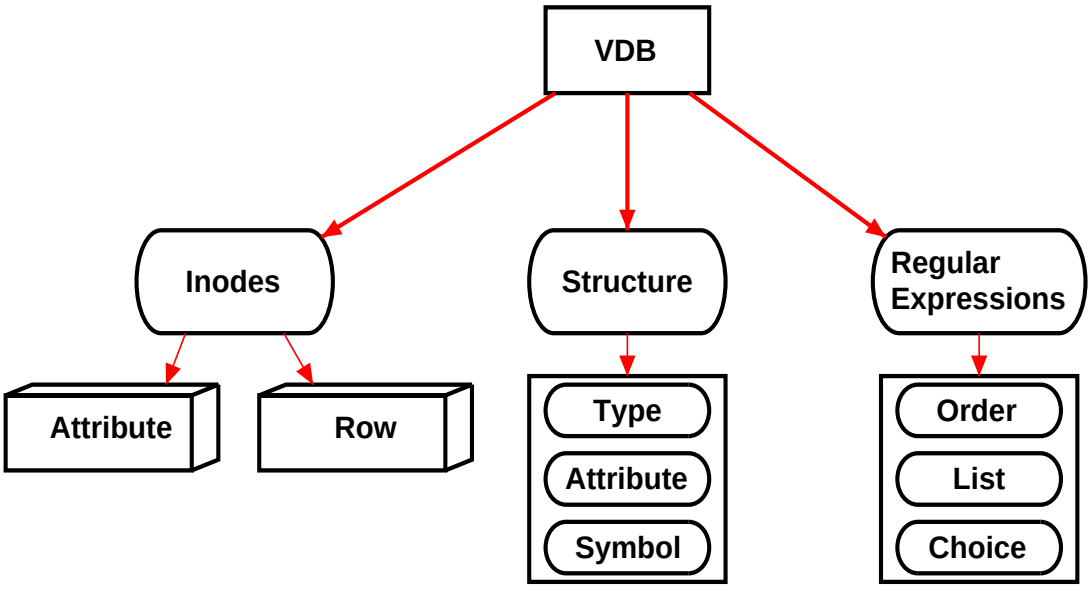
VDB and LaCo^2
Virtual Data Base System
Language Compiler

Stefan Bosse

A Virtual Data Base System for CAD Applications: Principles, Programming, and API



Virtual Data Base (VDB): Overview

A graph based database system for CAD applications.

1. Introduction

The Virtual Data Base (VDB) system is used to manage large structured data sets in CAD systems. It provides different representation levels of structures: filesystem and path level, inode level, and record type level. The VDB inode level maps the elements of a structure graph (the content) to a inode based filesystem. The inode filesystem represents a generic graph, too. Inodes can be searched and retrieved either by their unique inode number or using paths (and regular expressions). Inodes can represent any structure type, thus encapsulating the structure elements.

An inode consists of an identifier, a unique structure type, a attribute table and a row table containing the inode content (data, childs of this structure element). Structures create hierarchical orders of inodes and the mapping of inodes to the graph.

There is a definition structure graph, defining content elements and the order they may appear in the content graph. An element of a structure graph contains child elements and attributes related with this element. The childs can be organized with repeating lists, (ordered) sub-structures (product types), and disjunct lists (sum types).

The virtual database can be saved to and loaded from a file. It is a complete and generic interchange format which can be used by different programs at different runtime states. It can represent the complete state of a CAD program.

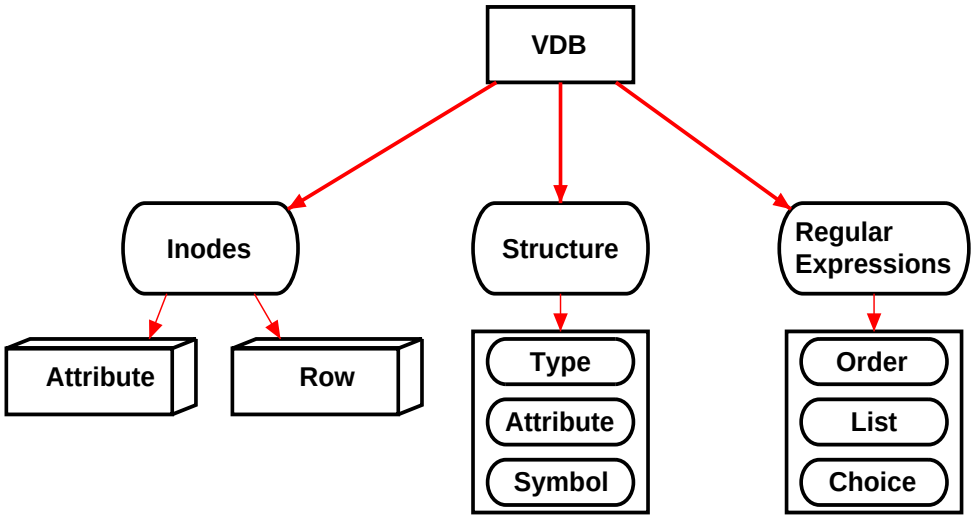
The structure graph or part of the graph can be represented using structural regular expressions. Regular expression ares used 1. to define the structure and the structure graph, and 2. to search and query the database.

The structure graph can be constructed from different sub-structures. Each structure can be imported from or exported to a plain text file using specified lexers, parsers, and printers. Alternatively, each structure can be imported or exported using universal XML representations only requirung the structure type definition.

There are different specification languages used to define the structure graph and for generating different application software modules like parsers and lexers (formatted input) or printers (formatted output) handling content. There is a structure definition language S, a parser and lexer generator language P, and a translation

language T. They are all part of the language compiler system LaCo.

Figure 1. Virtual Database objects and organization



2. VDB terminology

VBD terminology used in this document:

Structure

The structure definition graph consisting of nodes defining structure types, attributes, and symbols. The graph defines relations and the order of elements in this graph.

Type

The structure type: an element of a structure definition graph, mappable to an inode.

Attribute

Attributes can be assigned to types. Attributes are represented by row tuples (A,V).

Symbol

A symbolic element of a structure definition or attribute. Symbols are leafs of the structure graph.

Inode ID

Basic element of the database consisting of an identifier, a structure type indentifier, a attribute list, and a row (array) table. An inode is always accessed by the unique inode number. An inode can represent any structure type.

Row

Tuple element of a table with two columns (format A=V). Each column is of type Value.

Value

A value is a basic type from the set:
INT, INT64, FLOAT, CHAR, STRING, BOOL, CHAR, NODE, DIRECTORY.

Record REC

Record types represent elements of the structure definition graph, mapping inode graphs to product and sum types of the application programming language (ML).

Handle

Handle (identifier) of a record type used by the VDB interpreter.

3. Regular Expressions

Regular expressions are used to define the structure graph and to search the database for elements.

Ordered Structure Set

The ordered (sub-)structure set expression generates a list of elements. They must appear in the specified order. The set must be complete, except it contains optional elements.

Unordered Structure Set

The unordered (sub-)structure set expressions generates a list of elements. The order of the elements is arbitrary. The set must be complete, except it contains optional elements.

List

An element can appear at least one (zero) or several times in the expression.

Choice

One element of a set of elements appears in the expression.

Optional

An element can appear zero or one time in the expression.

Table 1. Structural Regular Expression Syntax

Regular Expression	Description
`(a,b,c,...)`	Ordered Structure Set
`(a&b&c...)`	Unordered Structure Set
`(x)*`	Repeating list with zero, one, or more than one element.
`(x)+`	Repeating list with one or more than one element.

Regular Expression	Description
`(a|b|c|...)`	Selects one of the set the of elements (choice).
`x?`	Element appears zero or one time (option).
`label:x`	Element or regular expression with label name.

Virtual Data Base (VDB): API

Application Programming Interface: Functions & types for database access

1. Basic Types

The Virtual Database is defined on programming level by a set of record and sum types. The ML basic types of the API are shown in definition 1. They are used for inode types, inode names, row elements (both attribute lists and content tables), and for structure definitions.

Definition 1. Basic VDB types (ML): Type of an inode or element, and a inode value (row or element).

```
type id = int*int
type node = int
(*
** Row with two columns:
**
**  +---------------+
**  | desc  | value |   desc: {Type, Attr, Ident}
**  +---------------+
**  +---------------+
**  |       | value |   value: {Int, String, Sym, Node, ....}
**  +---------------+
*)
type row = (type_value*type_value)
(*
** Type signature
*)
type type_kind =
  | INT
  | INT64
  | FLOAT
  | STRING
  | CHAR
  | BOOL
  | SYM of id
  | TYPE of id
  | ATTR of id
  | IDENT of string
  (*
  **  $XYZ
  *)
  | SPECIAL of string
```

```
  | ROWS
  | DIR
  | NODE
  | EMPTY
(*
** Content: Inode type, name, row columns
*)
type type_value =
  | Int of int
  | Int64 of int64
  | Float of float
  | String of string
  | Char of char
  | Bool of bool
  | Sym of id
  | Attr of id
  | Type of id
  | Node of node
  | Ident of string
  | Special of string
  | Dir
  | Rows of row list
  (*
  ** textual regular expression (required for query)
  **
  **  "nx[0-9]+"
  **
  *)
  | Regex of string
  | Empty
```

The inode type is shown in definition 2. An inode is specified by a unique identifier number (type `id`), a type (type `type_value`), and optionally a name (type `type_value`, e.g. `IDENT` <string>). A default value can be specified optionally, too. Not used entries are marked with value `Empty`.
Child elements of this inode are strored in the `rows` table. All child elements get a link to their parents in the `parent` list entry. A child can have more than one parent.

Definition 2. Inode type

```
(*
** Node constructor
*)
type inode = {
  (*
  ** Nodes belong to structure "DIR", s=0
  *)
  id: int;
  (*
  ** Tpye of this inode
  *)
  mutable typ: type_value;
```

```
  mutable name: type_value;
  (*
  ** Default value, if any
  *)
  mutable def: type_value;
  (*
  ** Attribute and content tables
  *)
  mutable attr: row list;
  mutable rows: row array;

  (*
  ** Optional name  (Ident _) <-> row number mapping
  *)
  mutable hash: (string,int) Hashtbl.t option;

  (*
  ** Links to this inode
  *)
  mutable parent: int list;

  (*
  ** Live field required for garbage collection
  *)
  mutable live: int;
}
```

2. Structure Definition Graph API

The structure definition graph, specifying the content of the database or a part of the database content, is defined using the types explained in definition 3. The structure definition graph can consist of sub-graphs, each defining different structures (languages).

Definition 3. Structure Definition Graph API: A type descriptor specifies one element of the structure graph (type, attribute, symbol), and the structure descriptor containes the graph and all of the element definitions.

```
(*
** Type descriptor
*)
type type_desc = {
  mutable name: string;

  (*
  ** Unique TYPE/ATTR/SYM identifier
  *)
  mutable typ: type_kind;
  (*
  ** IF ATTR|TYPE: attribute structure
  *)
  mutable attr: regex;
  (*
  ** IF TYPE: type structure (elements)
```

```
  *)
  mutable rows: regex;
  mutable flags: flag list;
}
and flag =
  | ATTR_generic
  (*
  ** Temporary types consist of only one nameless
  ** row of regular expression type ROW and
  ** have no attributes. They can be
  ** replaced in parent structures with
  ** their row.
  ** Example:
  **  pattern is "signal,event*"
  **  event is "INT#INT"
  **  and is equivalent to "signal,(INT#INT)*"
  **
  *)
  | TEMP
  (*
  ** Compacted types:
  **
  **  type1 [] := type2
  **  type1 [] := type2A
  **
  **  Type2 is compacted and merged with type1.
  **  Type2 can be a choice list, too.
  **
  **  Enables compacted rows: (Type1,Node (Type2))
  **
  *)
  | COMPACT

type struct_desc = {
  mutable name: string;
  mutable hash: (string,type_kind) Hashtbl.t;
  mutable types: type_desc array;
  mutable attrs: type_desc array;
  mutable syms: type_desc array;
  mutable types_top: int;
  mutable attrs_top: int;
  mutable syms_top: int;
}
```

3. Database

The database structure is shown in definition 4. All structure definitions are held in the `structs` entry (types, attributes, symbols). The next available inode in the dynamic growing inode array `inodes` is referenced by the `next` entry. The root inode is assigned to path `"/"`.

Definition 4. Global database structure

```
type db = {
```

```
  mutable structs: struct_desc array;
  mutable inodes: inode array;
  mutable root: node;
  mutable free: node list;
  mutable next: node;
}
```

4. Database access

New indoes can be created conforming to the loaded structure definitions. Existing inodes can be retrieved either by the unqiue inode number or using query functions.
Definition 5 shows the basic functions required to allocate, destroy, and modify inodes.

Definition 5. Inode allocation, destruction, and modification functions

```
val new_inode :
  unit ->
  Vdb_types.node
val make_node :
  name:Vdb_types.type_value ->
  type:Vdb_types.type_value ->
  attribute:Vdb_types.row list ->
  rows:Vdb_types.row array ->
  Vdb_types.node
val make_leaf :
  name:Vdb_types.type_value ->
  type:Vdb_types.type_value ->
  def:Vdb_types.type_value ->
  attribute:Vdb_types.row list ->
  Vdb_types.node
val remove_node :
  Vdb_types.node ->
  unit
val remove_rows :
  Vdb_types.node ->
  rownums:int list ->
  unit
val add_rows :
  Vdb_types.node ->
  Vdb_types.row array ->
  unit
```

new_inode
: Return a new empty inode. If there is a no inode avaiable, the dynamic inode staorage database is expanded.

make_node, make_leaf
: Make a new inode (using new_inodw) and assign name, type, attributes, and the child rows to this inode. Childs nodes are linked to the new parent inode. The `make_leaf` function creates a leaf inode without child elements but with optional default value `def`.

remove_node
: Remove an inode and unlink this inode from parents. That’s all. The rest is done by the garbage collector.

remove_rows
: Remove a numbered list of rows from an inode.

add_rows
: Add rows to inode and update inode (rebuild hash table).

Virtual Data Base (VDB): Query

Application Programming Interface: Functions for database query and lookup of elements

1. Basic Types

The Virtual Database is defined on programming level by a set of record and sum types. The ML basic types of the API are shown in definition 1. They are used for inode types, inode names, row elements (both attribute lists and content tables), and for structure definitions.

Definition 1. Basic VDB types (ML): Type of an inode or element, and a inode value (row or element).

```
type id = int*int
type node = int
(*
** Row with two columns:
**
**  +---------------+
**  | desc  | value |   desc: {Type, Attr, Ident}
**  +---------------+
**  +---------------+
**  |       | value |   value: {Int, String, Sym, Node, ....}
**  +---------------+
*)
type row = (type_value*type_value)
(*
** Type signature
*)
type type_kind =
  | INT
  | INT64
  | FLOAT
  | STRING
  | CHAR
  | BOOL
  | SYM of id
  | TYPE of id
  | ATTR of id
  | IDENT of string
  (*
  **  $XYZ
  *)
  | SPECIAL of string
```

```
  | ROWS
  | DIR
  | NODE
  | EMPTY
(*
** Content: Inode type, name, row columns
*)
type type_value =
  | Int of int
  | Int64 of int64
  | Float of float
  | String of string
  | Char of char
  | Bool of bool
  | Sym of id
  | Attr of id
  | Type of id
  | Node of node
  | Ident of string
  | Special of string
  | Dir
  | Rows of row list
  (*
  ** textual regular expression (required for query)
  **
  **  "nx[0-9]+"
  **
  *)
  | Regex of string
  | Empty
```

The inode type is shown in definition 2. An inode is specified by a unique identifier number (type `id`), a type (type `type_value`), and optionally a name (type `type_value`, e.g. `IDENT` <string>). A default value can be specified optionally, too. Not used entries are marked with value `Empty`.
Child elements of this inode are strored in the `rows` table. All child elements get a link to their parents in the `parent` list entry. A child can have more than one parent.

Definition 2. Inode type

```
(*
** Node constructor
*)
type inode = {
  (*
  ** Nodes belong to structure "DIR", s=0
  *)
  id: int;
  (*
  ** Tpye of this inode
  *)
  mutable typ: type_value;
```

```
  mutable name: type_value;
  (*
  ** Default value, if any
  *)
  mutable def: type_value;
  (*
  ** Attribute and content tables
  *)
  mutable attr: row list;
  mutable rows: row array;

  (*
  ** Optional name  (Ident _) <-> row number mapping
  *)
  mutable hash: (string,int) Hashtbl.t option;

  (*
  ** Links to this inode
  *)
  mutable parent: int list;

  (*
  ** Live field required for garbage collection
  *)
  mutable live: int;
}
```

2. Structure Definition Graph API

The structure definition graph, specifying the content of the database or a part of the database, is defined using the types explaine din definition 3. The structure definition graph can consist of sub-graphs, each defining different structures (languages).

Definition 3. Structure Definition Graph API: A type descriptor specifies one element of the structure graph (type, attribute, symbol), and the structure descriptor containes the graph and all of the element definitions.

```
(*
** Type descriptor
*)
type type_desc = {
  mutable name: string;

  (*
  ** Unique TYPE/ATTR/SYM identifier
  *)
  mutable typ: type_kind;
  (*
  ** IF ATTR|TYPE: attribute structure
  *)
  mutable attr: regex;
  (*
  ** IF TYPE: type structure (elements)
```

```
  *)
  mutable rows: regex;
  mutable flags: flag list;
}
and flag =
  | ATTR_generic
  (*
  ** Temporary types consist of only one nameless
  ** row of regular expression type ROW and
  ** have no attributes. They can be
  ** replaced in parent structures with
  ** their row.
  ** Example:
  **  pattern is "signal,event*"
  **  event is "INT#INT"
  **  and is equivalent to "signal,(INT#INT)*"
  **
  *)
  | TEMP
  (*
  ** Compacted types:
  **
  **  type1 [] := type2
  **  type1 [] := type2A
  **
  **  Type2 is compacted and merged with type1.
  **  Type2 can be a choice list, too.
  **
  **  Enables compacted rows: (Type1,Node (Type2))
  **
  *)
  | COMPACT

type struct_desc = {
  mutable name: string;
  mutable hash: (string,type_kind) Hashtbl.t;
  mutable types: type_desc array;
  mutable attrs: type_desc array;
  mutable syms: type_desc array;
  mutable types_top: int;
  mutable attrs_top: int;
  mutable syms_top: int;
}
```

3. Database

The database structure is shown in definition 4. All structure definitions are held in the `structs` entry (types, attributes, symbols). The next available inode in the dynamic growing inode array `inodes` is referenced by the `next` entry. The root inode is assigned to path `"/"`.

Definition 4. Global database structure

```
type db = {
```

```
  mutable structs: struct_desc array;
  mutable inodes: inode array;
  mutable root: node;
  mutable free: node list;
  mutable next: node;
}
```


S Language [LaCo/VDB]

Principles, language definition, and Application Programming Interface

1. Introduction

The S language is used to specify the structure definition graph for the virtual database system, and is part of the LaCo compiler. A source file with file suffix `str` consists of different sections, correlation with the elements of a VDB structure definition graph, explained in definition 1. There are definitions of types, attributes of types, and symbols.
The S file is translated by the LaCo compiler into ML code. There are two output formats: 1. inode based VDB structures and register functions applied to these definitions, used to define the structure graph at runtime dynamically, and 2. static equivalent ML record/sum types of the structure definition graph.

Definition 1. S language file structure and sections

```
STRUCTURE <identifier> -> <identifier>;
DESCRIPTION "<text>";
TYPES
BEGIN
  <type-definition>+
END;
ATTRIBUTES
BEGIN
  <attr-definition>+
END;
SYMBOLS
BEGIN
  <sym-definition>+
END;
```

The first statement defines the structure name and maps this name to a unique structure identifier (in general an abbreviation), which can be used in external structures. A description text follows the first statement.
First all types of the structure definition graph are defined in the `TYPES` section, followed by attribute (`ATTRIBUTES`) and symbol definition sections (`SYMBOLS`). Attributes are bound to types, and appear on the left side of a type definition. Symbols appear on the right side of a type definition.
Types and attributes are defined using structural regular expressions.

2. Syntax

The formal definitons of the syntax of the S language and structural regular expressions are shown below in defintions 2 to 3.
Usually symbol, attribute, and type identfiers (on left hand side of definition) are mapped to constructor names (used in ML programming langauge) by changing the first letter to an uppercase letter. Alternatively, a constrcutor name can be specified explicitly using the mapping operator `->`.
The first name identifier is used with references by other sections of the S file and other languages like the P or T languages.

Definition 2. Formal syntax definition of the S language

```
symbol-definition ::= identifier [ '->' identifier ] ';' .
attr-definition ::= identifier ':=' regular-expr [ '->' identifier ] .
type-defintion ::= identifier [ '[' regular-expr ']' ] ':=' regular-expr
                   [ '->' identifier ].
```

Structural regular expressions are use dto define the structure definition graph at design time and performing query requests in the database at runtime. They are used in other LaCo languages, too, though only a subset can be supported.
The `structure` expression defines a list of structure elements, which can be regular expressions, too. There are orderd and unordered structures. In ordered structures, the elements must appear in the order they were specified.
There are two different kinds of lists. The `list0` expression contains zero, one, or more than one elements. The `list1` expression contains at least one element.
The `choices` expression consists of a mutual list of elements (or regular expressions). Only one element of the list can be selected.
The `context` expression contains a hierarchical list of elements, representing a subgraph of the structure defintion graph. This expression is only used for query and lookup requests.

Definition 3. Formal syntax definition of structural regular expressions

```
regular-expr ::= identifier | structure | optional | list |
                 choices | context | label .
identifier ::= [ 'a'-'z' 'A'-'Z' '%' '_' '-' '0'-'9' ]+ .
structure ::= structure-ordered | structure-unordered .
structure-ordered ::= regular-expr // ',' .
structure-unordered ::= regular-expr // '&' .
optional ::= regular-expr '?' .
label ::= identifier ':' regular-expr .
list ::= list0 | list1 .
list0 ::= regular-expr '*' .
list1 ::= regular-expr '+' .
choices ::= regular-expr // '|' .
context ::= ( regular-expr [ '[' attributes ']' ] ) // '<' .
attributes ::= ( identifier | identifier '=' value ) // ',' .
```

3. Example

The following example defines a expression structure graph.

Example 1. S input file expr.str

```
STRUCTURE Expression -> EXP;
DESCRIPTION "Expression Analyzer";
TYPES
BEGIN
  element [selector?] := IDENT;
  expr [operator & guarded?] := (element | expr | value)+;
  value [format, time?] := INT|FLOAT|BOOL|STRING|CHAR;
END;
ATTRIBUTES
BEGIN
  a := INT|element|expr;
  b := INT|element|expr;
  direction := up|down;
  freq := INT,(hz|khz|mhz|ghz)? -> Freq;
  guarded := BOOL;
  index := (INT|element|expr)+;
  method := IDENT;
  operator := add|sub|mul|div|land|lor|lxor|lnot|band|bor|bx
              or|bnot|lsl|lsr|eq|neq|gt|ge|lt|le;
  -- Generic attribute
  position* := INT+;
  range := a,b,direction;
  selector := (range|index|method|struct)+;
  size := INT;
  struct := IDENT;
  time := INT,(ps|ns|us|ms|sec)?;
  format := (%bit|%hex|%decimal|%float|%string|%char|%bool),%vector?;
END;
SYMBOLS
BEGIN
  add;
  band;
  %bit -> FmBit;
  bnot;
  bor;
  bnor;
  bnand;
  bxor;
  %bool -> FmBoolean;
  %char -> FmChar;
  %decimal -> FmDecimal;
  div;
  down;
  eq;
  %float -> FmFloat;
  ge;
  ghz;
  gt;
  %hex -> FmHex;
  ...
```

```
END;
```

Example 2. Generated ML output of structure definition (inode based) used for the inode VDB system: expr.ml

```
1  open Vdb
2  open Vdb_types
3  enum structs = {
4    EXP=1
5  }
6  enum types = {
7    Element = Type (EXP,1);
8    Expr = Type (EXP,2);
9    Value = Type (EXP,3)
10 }
11 let types_max = 3
12 enum attrs = {
13   A = Attr (EXP,1);
14   B = Attr (EXP,2);
15   Direction = Attr (EXP,3);
16   Format = Attr (EXP,15);
17   Freq = Attr (EXP,4);
18   Guarded = Attr (EXP,5);
19   Index = Attr (EXP,6);
20   Method = Attr (EXP,7);
21   Operator = Attr (EXP,8);
22   Position = Attr (EXP,9);
23   Range = Attr (EXP,10);
24   Selector = Attr (EXP,11);
25   Size = Attr (EXP,12);
26   Struct = Attr (EXP,13);
27   Time = Attr (EXP,14)
28 }
29 let attrs_max = 15
30 enum syms = {
31   FmBit = Sym (EXP,3);
32   FmBoolean = Sym (EXP,9);
33   FmChar = Sym (EXP,10);
34   FmDecimal = Sym (EXP,11);
35   FmFloat = Sym (EXP,15);
36   FmHex = Sym (EXP,19);
37   FmNatural = Sym (EXP,38);
38   FmSigned = Sym (EXP,41);
39   FmString = Sym (EXP,42);
40   FmVector = Sym (EXP,46);
41   Add = Sym (EXP,1);
42   Band = Sym (EXP,2);
43   Bnand = Sym (EXP,7);
44   Bnor = Sym (EXP,6);
45   Bnot = Sym (EXP,4);
46   Bor = Sym (EXP,5);
47   Bxor = Sym (EXP,8);
48   Div = Sym (EXP,12);
49   Down = Sym (EXP,13);
50   ...
```

```
51 }
52 let syms_max = 46
53 let register () =
54   Version.register "EXP Expression Analyzer";
55   register_structure EXP "EXP";
56   register_syms [
57     "%bit",FmBit;
58     "%bool",FmBoolean;
59     "%char",FmChar;
60     "%decimal",FmDecimal;
61     "%float",FmFloat;
62     "%hex",FmHex;
63     "%natural",FmNatural;
64     "%signed",FmSigned;
65     "%string",FmString;
66     "%vector",FmVector;
67     "add",Add;
68     "band",Band;
69     "bnand",Bnand;
70     "bnor",Bnor;
71     "bnot",Bnot;
72     "bor",Bor;
73     ...
74       ];
75   register_attrs [
76     "a",A,"(INT|element|expr)";
77     "b",B,"(INT|element|expr)";
78     "direction",Direction,"(up|down)";
79     "format",Format,"(%bit|%hex|
80              %decimal|%float|%string|%char|%bool),%vector?";
81     "freq",Freq,"INT,(hz|khz|mhz|ghz)?";
82     "guarded",Guarded,"BOOL";
83     "index",Index,"(INT|element|expr)+";
84     "method",Method,"";
85     "operator",Operator,"(add|sub|mul|div|land|lor|
86                lxor|lnot|band|bor|bxor|bnot|
87                lsl|lsr|eq|neq|gt|ge|lt|le)";
88     "position",Position,"(INT)+";
89     "range",Range,"a,b,direction";
90     "selector",Selector,"(range|index|method|struct)+";
91     "size",Size,"INT";
92     "struct",Struct,"";
93     "time",Time,"INT,(ps|ns|us|ms|sec)?";
94   ];
95   register_types [
96     "element",Element,"selector?","";
97     "expr",Expr,"operator&guarded?","(element|expr|value)+";
98     "value",Value,"format,time?","(INT|FLOAT|BOOL|STRING|CHAR)";
99   ];
```

Example 3. Generated ML output of structure definition (record type based) used for the inode VDB system: expr_types.ml

```
1 type identifier = string
2 (* ATTRIBUTES *)
```

```
and a =
       | A_int of int
       | A_Element of element
       | A_Expr of expr
and b =
       | B_int of int
       | B_Element of element
       | B_Expr of expr
and direction =
       | Direction_Up
       | Direction_Down
and format =
  {
    mutable format: format_choice;
    mutable format_FmVector: bool;
  }
  and format_choice =
       | Format_FmBit
       | Format_FmHex
       | Format_FmDecimal
       | Format_FmFloat
       | Format_FmString
       | Format_FmChar
       | Format_FmBoolean
and freq =
  {
    mutable freq_int: int;
    mutable freq: freq_choice option;
  }
  and freq_choice =
       | Freq_Hz
       | Freq_Khz
       | Freq_Mhz
       | Freq_Ghz
and guarded = bool
and index =
  index_choice list
  and index_choice =
       | Index_int of int
       | Index_Element of element
       | Index_Expr of expr
and metho = identifier
and operator =
       | Operator_Add
       | Operator_Sub
       | Operator_Mul
       ...
       | Operator_Lt
       | Operator_Le
and position = int list
and range =
  {
    mutable range_A: a;
    mutable range_B: b;
```


```
57      mutable range_Direction: direction;
58    }
59  and selector =
60    selector_choice list
61    and selector_choice =
62        | Selector_Range of range
63        | Selector_Index of index
64        | Selector_Metho of metho
65        | Selector_Struc of struc
66  and size = int
67  and struc = identifier
68  and time =
69    {
70      mutable time_int: int;
71      mutable time: time_choice option;
72    }
73    and time_choice =
74        | Time_Ps
75        | Time_Ns
76        | Time_Us
77        | Time_Ms
78        | Time_Sec
79  (* TYPES *)
80  and element = element_cont * element_attr
81  and element_attr = selector option
82  and element_cont = identifier
83  and expr = expr_cont * expr_attr
84  and expr_attr =
85    {
86      mutable expr_Operator: operator;
87      mutable expr_Guarded: guarded option;
88    }
89  and expr_cont =
90    expr_choice list
91    and expr_choice =
92        | Expr_Element of element
93        | Expr_Expr of expr
94        | Expr_Value of value
95  and value = value_cont * value_attr
96  and value_attr =
97    {
98      mutable value_Format: format;
99      mutable value_Time: time option;
100   }
101 and value_cont =
102       | Value_int of int
103       | Value_float of float
104       | Value_bool of bool
105       | Value_string of string
106       | Value_char of char
```


P Language [LaCo/VDB]

Principles, language definition, and Application Programming Interface

1. Introduction

The P language is used to specify lexers and parsers, and is part of the LaCo compiler. A source file with file suffix `par` consists of different sections, explained in definition 1. A P language file requires an already processed structure graph defintion S.
The P file is translated by the LaCo compiler into ML code: a lexer file with ending `*_lexer.mll` and a parser file with ending `*_parser.mly`.

Definition 1. P language file structure and sections

```
PARSER <identifier>;
STRUCTURE <identifier>;
DESCRIPTION "<text>";
TOKENS
BEGIN
  <token-definition>+
END;
EVALUATE
BEGIN
  <eval-definition>+
END;
CONVERSION
BEGIN
  <conv-definition>+
END;
RULES
BEGIN
  <rule-definition>+
END;
```

The first statement defines the parser name, the second statement binds the parser to an existing structure. A description text follows the second statement.
First all lexer tokens are defined in section `TOKENS`. Two optional sections follow. The `EVALUATE` section can be used to define evaluation orders of tokens for the parser. The `CONVERSION` section can be used to define value conversion rules, mapping a token-datatype tuple to a (ML) conversion function, converting from string format to the desired target format.
Finally, the `RULES` section defines parser rules, mapping token patterns to structure types, attributes, and symbols.

2. Syntax

The formal definitons of the syntax of the P language are shown in defintions 2 to 3.

Definition 2. Formal syntax definition of the P language

```
token-definition ::= identifier ':=' regular-expr ';' .
parser-rule ::= identifier [ '*' ] ':' BEGIN pattern-matching+ END ';' .
pattern-matching ::= 'WHEN' regular-expr '=>' constructor ';' .
constructor ::= struct-elem | terminal-constructor | type-constructor |
                row-constructor | reference .
terminal-constructor ::=
(INT|INT64|CHAR|FLOAT|BOOL|STRING|NODE|ROWS|TYPE)
                          struct-elem | reference .
row-constructor ::= terminal-constructor '=' terminal-constructor .
type-constructor ::= struct-elem '[' [row-constructor // ','] ']'
                                 '[' [row-constructor // ','] ']' .
reference ::= '$' number .
struct-elem ::= Identifier '.' identifier .
```

Structural regular expressions are used to define lexer tokens and parser rules. Only a subset of regular expressions are supported, depending of the kind of definition.

Definition 3. Formal syntax definition of regular expressions used for lexer tokens and parser rules

```
regular-expr ::= identifier | variable |
                 structure  | list | choices .
identifier ::= [ 'a'-'z' 'A'-'Z' '%' '_' '-' '0'-'9' ]+ .
Identifier ::= ['A'-'Z'] [ 'a'-'z' 'A'-'Z' '_' '-' '0'-'9' ]* .
variable ::= '$' identifier .
structure ::= structure-ordered | structure-unordered .
structure-ordered ::= regular-expr // ',' .
structure-unordered ::= regular-expr // '&' .
list ::= list0 | list1 .
list0 ::= regular-expr '*' .
list1 ::= regular-expr '+' .
choices ::= regular-expr // '|' .
context ::= ( regular-expr [ '[' attributes ']' ] ) // '<' .
attributes ::= ( identifier | identifier '=' value ) // ',' .
range ::= character '-' character .
character ::= ''' CHAR ''' .
```

3. Example

The following example defines a lexer and parser for the expression structure graph.

Example 1. P input file expr.par

```
PARSER EXP;
STRUCTURE EXP;
```

```
DESCRIPTION "Expression parser";
TOKENS
BEGIN
    LPAREN := '(';
    RPAREN := ')';
    LBRAK := '[';
    RBRAK := ']';
    LCURL := '{';
    RCURL := '}';
    ARGSEP := ',';
    SEL := '.';


    PS := "ps";
    NS := "ns";
    US := "us";
    ...

    LAND := "land";
    LOR := "lor";
    LNAND := "lnand";
    LNOR := "lnor";
    LXOR := "lxor";
    ...

    EOF := $EOF;
    BITVAL  [$LEFT=2,$1=POS,$2=BUF] :=
       "0b",('0'|'1'|'u'|'x'|'U'|'X'|'z'|'Z'|'H'|'L'|'h'|'l');
    BITVEC  [$LEFT=2,$1=POS,$2=BUF] :=
       "0b",('0'|'1'|'u'|'x'|'U'|'X'|'z'|'Z'|'H'|'L'|'h'|'l')+;
    HEXVAL  [$LEFT=2,$1=POS,$2=BUF] :=
       "0x",('0' - '9'|'a' - 'z'|'A' - 'Z')+;
    DECVAL  [$1=POS,$2=INT] := ('0' - '9'),('0' - '9')*;
    DECVALLONG  [$RIGHT=1,$1=POS,$2=INT64] :=
       ('0' - '9'),('0' - '9')*,'L';
    FLOAT   [$1=POS,$2=FLOAT] := ('0' - '9')+,'.' ,('0' - '9')+;
    CHAR    [$LEFT=1,$RIGHT=1,$1=POS,$2=CHAR] := '',$TEXT1,'';
    STRING  [$1=POS,$2=STRING] := '"',$TEXT,'"';
    COMMENT := "--",$TEXT,$NL;
    IDENT   [$1=POS,$2=BUF] := ('a' - 'z'|'A' - 'Z'|'_'),
                               ('a' - 'z'|'A' - 'Z'|'0' - '9'|'_')*;
END;
EVALUATE
BEGIN
  LEFT :=  MUL,DIV;
  LEFT :=  ADD,SUB;
END;
CONVERSION
BEGIN
  WHEN DECVAL,INT => "int_of_string","";
  WHEN DECVAL,INT64 => "Int64.of_string","";
  WHEN HEXVAL,INT => "int_of_string","0x";
  WHEN HEXVAL,INT64 => "Int64.of_string","0x";
END;
```

```
RULES
BEGIN
  -- ATTR: Returns Rrow
  range:
  BEGIN
    WHEN range_expr,DOWNTO,range_expr =>
      EXP.range [EXP.a=$1,EXP.b=$3,EXP.direction=EXP.down];
    WHEN range_expr,TO,range_expr =>
      EXP.range [EXP.a=$1,EXP.b=$3,EXP.direction=EXP.up];
  END;

  -- ATTR: Returns Rterm
  range_expr:
  BEGIN
    WHEN DECVAL =>
      INT $1;
    WHEN DECVALLONG =>
      INT $1;
    WHEN HEXVAL =>
      INT $1;
    WHEN expr =>
      NODE $1;
  END;

  index:
  BEGIN
    WHEN expr => TYPE $1=NODE $1;
  END;


  -- TYPE: Returns Rtype
  identifier:
  BEGIN
    WHEN IDENT => EXP.element [EXP.position=$POS] [$NAME=IDENT $1];
  END;

  element [EXP.position=$POS]:
  BEGIN
    WHEN BITVAL => EXP.value [EXP.format=EXP.%bit]
                             [$NAME=STRING $1,$VAL=STRING $1];
    WHEN HEXVAL => EXP.value [EXP.format=EXP.%bit]
                             [$NAME=STRING $1,$VAL=STRING $1];
    WHEN BITVEC => EXP.value [EXP.format=EXP.%bit,
                              EXP.format=EXP.%vector]
                             [$NAME=STRING $1,$VAL=STRING $1];
    WHEN DECVAL => EXP.value [EXP.format=EXP.%decimal]
                             [$NAME=STRING $1,$VAL=INT $1];
    WHEN STRING => EXP.value [] [$NAME=STRING $1,$VAL=STRING $1];
    WHEN CHAR => EXP.value [] [$NAME=STRING $1,$VAL=CHAR $1];
    WHEN FLOAT => EXP.value [EXP.format=EXP.%float]
                            [$NAME=STRING $1,$VAL=FLOAT $1];
    WHEN IDENT => EXP.element [] [$NAME=IDENT $1];
    WHEN IDENT,selector_list => EXP.element $2 [$NAME=IDENT $1];
  END;
```

```
-- TYPE: Returns Rtype
-- Main entry rule
expr*:
BEGIN
  WHEN element => $1;
  WHEN expr,operation,expr =>
    EXP.expr [EXP.operator=$2]
             [$NAME=$ID,EXP.expr=NODE $1,EXP.expr=NODE $3];
END;

-- SYM: Returns Rterm
operation:
BEGIN
  WHEN ADD => EXP.add ;
  WHEN SUB => EXP.sub ;
  WHEN MUL => EXP.mul ;
  WHEN DIV => EXP.div ;
  WHEN LAND => EXP.land ;
  ...
END;

-- ATTR: Returns Rlist Rrow
--    In the case of separated repeating sequence:
--
--      WHEN arg+ => $1 means arg arg ...
--
--      WHEN (arg,SEP)+ => $1  means arg SEP arg SEP ... arg SEP
--
--      WHEN arg => $1  and
--      WHEN (arg,SEP)+ => $1  means arg SEP arg SEP ... arg
--
selector_list:
BEGIN
  WHEN selector+ => $1;
END;

index_list:
BEGIN
  WHEN index => [$1];
  WHEN (index,ARGSEP)+ => $1;
END;

-- ATTR: Returns Rrow
selector:
BEGIN
  WHEN SEL,identifier => EXP.struct = NODE $2;
  WHEN LBRAK,range,RBRAK => $2;
  WHEN SEL,LBRAK,index_list,RBRAK => EXP.index = ROWS $3;
  WHEN SEL,LBRAK,index,RBRAK => EXP.index = ROWS $3;
END;
```


```
END;
```

Example 2. Generated output of the ML lexer usable for the inode VDB system: expr_lexer.mll

```
{
  open Expr_parser
  let mutable cur_line = 1
  let mutable cur_pos = 0
  let pos p =
      (cur_line*1000)+p
  let init () = cur_line  <- 1 ; cur_pos <- 0
  (* To buffer string literals *)
  let initial_string_buffer = String.create 256
  let string_buff = ref initial_string_buffer
  let string_index = ref 0
  let reset_string_buffer () =
    string_buff := initial_string_buffer;
    string_index := 0
  let store_string_char c =
    if !string_index >= String.length (!string_buff) then begin
     let new_buff = String.create (String.length (!string_buff) * 2) in
        String.blit (!string_buff) 0 new_buff 0
        (String.length (!string_buff));
        string_buff := new_buff
    end;
    String.unsafe_set (!string_buff) (!string_index) c;
    incr string_index
  let get_stored_string () =
    let s = String.sub (!string_buff) 0 (!string_index) in
    string_buff := initial_string_buffer;
    s
  (* To store the position of the beginning of a string and comment *)
  let string_start_pos = ref 0
  let this_char = ref None
}
rule token = parse
  | '('  { LPAREN }
  | ')'  { RPAREN }
  | '['  { LBRAK }
  | ']'  { RBRAK }
  | '{'  { LCURL }
  | '}'  { RCURL }
  | ','  { ARGSEP }
  | '.'  { SEL }
  | "ps"  { PS }
  | "ns"  { NS }
  ...
  | "bxor"  { BXOR }
  | "bnot"  { BNOT }
  | '+'  { ADD }
  | '-'  { SUB }
  | '*'  { MUL }
  | '/'  { DIV }
  | '='  { EQ }
```

```
51    ...
52    | eof  { EOF }
53    | "0b" ['0' '1' 'u' 'x' 'U' 'X' 'z' 'Z' 'H' 'L' 'h' 'l']
54      {
55          let pos = cur_line,lexbuf.Lexing.lex_curr_pos-cur_pos in
56          let str = Lexing.lexeme lexbuf in
57          let len = String.length str in
58          let str' = String.sub str 2 (len-2) in
59          BITVAL (pos,str')
60      }
61    | "0b" ['0' '1' 'u' 'x' 'U' 'X' 'z' 'Z' 'H' 'L' 'h' 'l']+
62      {
63          let pos = cur_line,lexbuf.Lexing.lex_curr_pos-cur_pos in
64          let str = Lexing.lexeme lexbuf in
65          let len = String.length str in
66          let str' = String.sub str 2 (len-2) in
67          BITVEC (pos,str')
68      }
69    | "0x" ['0'-'9' 'a'-'z' 'A'-'Z']+
70      {
71          let pos = cur_line,lexbuf.Lexing.lex_curr_pos-cur_pos in
72          let str = Lexing.lexeme lexbuf in
73          let len = String.length str in
74          let str' = String.sub str 2 (len-2) in
75          HEXVAL (pos,str')
76      }
77    | ['0'-'9'] ['0'-'9']*
78      {
79          let pos = cur_line,lexbuf.Lexing.lex_curr_pos-cur_pos in
80          let str = Lexing.lexeme lexbuf in
81          let v = int_of_string str in
82          DECVAL (pos,v)
83      }
84    | ['0'-'9'] ['0'-'9']* 'L'
85      {
86          let pos = cur_line,lexbuf.Lexing.lex_curr_pos-cur_pos in
87          let str = Lexing.lexeme lexbuf in
88          let len = String.length str in
89          let str' = String.sub str 0 (len-1) in
90          let v = Int64.of_string str' in
91          DECVALLONG (pos,v)
92      }
93    | ['0'-'9']+ '.' ['0'-'9']+
94      {
95          let pos = cur_line,lexbuf.Lexing.lex_curr_pos-cur_pos in
96          let pos = cur_line,lexbuf.Lexing.lex_curr_pos-cur_pos in
97          let str = Lexing.lexeme lexbuf in
98          let v = float_of_string str in
99          FLOAT (pos,v)
100     }
101   | '"' ['\000'-'\255'] '"'
102     {
103         let pos = cur_line,lexbuf.Lexing.lex_curr_pos-cur_pos in
104         let str = Lexing.lexeme lexbuf in
```

```
105         let len = String.length str in
106         let v = str.[1] in
107         CHAR (pos,v)
108       }
109     | '"'
110       {
111         let pos = cur_line,lexbuf.Lexing.lex_curr_pos-cur_pos in
112         reset_string_buffer();
113         let string_start = Lexing.lexeme_start lexbuf in
114         string_start_pos := string_start;
115         string lexbuf;
116         lexbuf.Lexing.lex_start_pos <-
117           string_start - lexbuf.Lexing.lex_abs_pos;
118         let str = get_stored_string() in
119         STRING (pos,str)
120       }
121     | "--"
122       {
123         reset_string_buffer(); comment lexbuf;
124         COMMENT
125       }
126     | ['a'-'z' 'A'-'Z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_']*
127       {
128         let pos = cur_line,lexbuf.Lexing.lex_curr_pos-cur_pos in
129         let str = Lexing.lexeme lexbuf in
130         IDENT (pos,str)
131       }
132     | '\n' { cur_line<-cur_line+1;cur_pos<-lexbuf.Lexing.lex_curr_pos;
133              token lexbuf}
134     | ' ' { token lexbuf }
135     | '\t' { token lexbuf }
136     | '\r' { token lexbuf }
137 and string = parse
138     | '\n'
139       {
140         cur_line<-cur_line+1;cur_pos<-lexbuf.Lexing.lex_curr_pos;
141         store_string_char(Lexing.lexeme_char lexbuf 0);
142         string lexbuf
143       }
144     | '"'  { () }
145     | eof { raise Exit }
146     | _
147       {
148         store_string_char(Lexing.lexeme_char lexbuf 0);
149         string lexbuf
150       }
151 and comment = parse
152     | '\n'  { () }
153     | eof { raise Exit }
154     | _
155       {
156         store_string_char(Lexing.lexeme_char lexbuf 0);
157         comment lexbuf
```

```
158     }
```

Example 3. Generated output of the ML parser usable for the inode VDB system: expr_parser.mly

```
1  %{
2    open Vdb
3    open Vdb_types
4    open Expr
5    let mutable positions = []
6    let add_pos pos =
7      let l,c = pos in
8      let p = l*1000+c in
9      if not (List.mem p positions) then positions <- p :: positions
10   let rec concat sl =
11     match sl with
12     | s :: tl -> s^(concat tl)
13     | [] -> ""
14   let make_Int v = Int v
15   let make_Int64 v = Int64 v
16   let make_Float v = Float v
17   let make_Bool v = Bool v
18   let make_Char v = Char v
19   let make_String v = String v
20   let make_Special v = Special v
21   let make_Node v = Node v
22   let make_Ident v = Ident v
23   let make_Rows rl = Rows rl
24   let make_Row r1 r2 =
25     match r1 with
26     | Special "VAL" -> (Empty,r2)
27     | _ ->
28     begin
29       match r2 with
30       | Special "POS" ->
31       begin
32        match positions with hd :: tl -> positions <- tl; r1,(Int hd);
   | _ -> r1,Empty
33       end;
34       | _ -> (r1,r2)
35     end
36   let rec get_Name rl = match rl with ((Special "NAME"),r2) :: tl ->
   r2 | hd :: tl
37  -> get_Name tl | [] -> Empty
38   let rec get_Names ra =
39     let ra_n = Array.length ra in
40     let mutable names_n = 0 in
41     Array.iter (fun r ->
42       match r with
43      | (Special "NAME"),(Rows rl) -> names_n <- names_n + (List.length
   rl)
44       | (Special "NAME"),r2 -> names_n <- names_n + 1
45       | _ -> ()) ra;
46     let names = Array.create names_n Empty  in
47     let ra' = Array.create (ra_n-names_n) (Empty,Empty)  in
```

```
48      let mutable i = 0 in
49      let mutable j = 0 in
50      Array.iter (fun r ->
51        let rec extract r = match r with
52        | (Special "NAME"),(Rows rl) -> List.iter extract rl
53        | (Special "NAME"),r2 -> names.(i) <- r2; i <- i + 1
54        | _ -> ra'.(j) <- r; j <- j + 1 in extract r) ra;
55      names,ra'
56    let filter_Rows rl = List.filter (fun r ->
57      match r with
58      | (Special "NAME"),r2 -> false
59      | _ -> true) rl
60    let get_type id = let i = db.inodes.(id) in i.typ
61    (*
62    ** Make a new node with name derived from row list ($NAME) of type
   'typ'...
63    ** In the case there is more than one $NAME row, then a directory is
   created
64    ** and the the initial node is created multiple times.
65    *)
66    let make_node typ (attrs:row list) (rows:row array) =
67      match get_Names rows with
68      | [|ident|],rows' ->
69         Vdb.make_node ident typ attrs rows'
70      | names,rows' ->
71         let rows'' = Array.map (fun ident ->
72             typ,(Node (Vdb.make_node ident typ attrs rows'))) names in
73         Vdb.make_node Empty typ [] rows''
74    (*
75    ** Row array management:
76    ** Mapping of parsers recursive list iteration to array manipulations
77    *)
78    type rows = {
79        mutable top: int;
80        mutable max: int;
81        mutable rows: Vdb_types.row array;
82      }
83    let mutable rows = []
84    let add_row r =
85      match rows with
86      | row :: _->
87        row.top <- row.top - 1;
88        row.rows.(row.top) <- r;
89        if row.top = 0 then
90        begin
91            row.top <- shunk_rows;
92            row.max <- row.max + shunk_rows;
93            row.rows <- Array.append (Array.create shunk_rows nilrow)
   row.rows;
94        end;
95      | _ -> progerr "add_row"
96    (*
97    ** Same as add_row, but flattens directories
98    ** on first level
```

```
99    *)
100   let add_row_relax r =
101     let typ,valu =  r in
102     match typ with
103     | Dir ->
104     begin
105       match valu with
106       | Node id ->
107         let i = db.inodes.(id) in
108         let rows = i.rows in
109         let n = Array.length rows in
110         (*
111         ** add_row has reverese order!
112         *)
113         for j = n-1 downto 0 do add_row i.rows.(j) done;
114       | _ -> add_row r
115     end
116     | _ -> add_row r
117   (*
118   ** Same as add_row_relax, but flattens directories
119   ** on second level
120   *)
121   let add_row2_relax r =
122     let typ,valu =  r in
123     match valu with
124     | Node id ->
125     begin
126       let i = db.inodes.(id) in
127       match i.typ with
128       | Dir -> add_row_relax (Dir,Node id)
129       | _ -> add_row r
130     end;
131     | _ -> add_row r
132   let get_rows () =
133     match rows with
134     | row :: _ ->
135       let res = Array.sub row.rows row.top (row.max - row.top) in
136       rows <- List.tl rows;
137       res
138     | _ -> progerr "get_rows = [||]"
139   let get_rows_list () = Array.to_list (get_rows ())
140   let init_rows () =
141     let row = {top=shunk_rows;max=shunk_rows;rows=Array.create
    shunk_rows nilrow}
142  in
143     rows <- row :: rows
144 %}
145 %token  LPAREN
146 %token  RPAREN
147 %token  LBRAK
148 %token  RBRAK
149 %token  LCURL
150 %token  RCURL
151 ...
```

```
%token  TO
%token  DOWNTO
%token  EOF
%token <(int*int)*string> BITVAL
%token <(int*int)*string> BITVEC
%token <(int*int)*string> HEXVAL
%token <(int*int)*int> DECVAL
%token <(int*int)*int64> DECVALLONG
%token <(int*int)*float> FLOAT
%token <(int*int)*char> CHAR
%token <(int*int)*string> STRING
%token  COMMENT
%token <(int*int)*string> IDENT
%left MUL DIV
%left ADD SUB
%start expr
%type <Vdb_types.node> expr
%%
/*
** ROW
*/
range:
   | range_expr DOWNTO range_expr {
     make_Row Expr.Range (make_Rows [
         make_Row (Expr.A) ($1);
         make_Row (Expr.B) ($3);
         make_Row (Expr.Direction) (Expr.Down);
         ]) }
   | range_expr TO range_expr {
     make_Row Expr.Range (make_Rows [
         make_Row (Expr.A) ($1);
         make_Row (Expr.B) ($3);
         make_Row (Expr.Direction) (Expr.Up);
         ]) }
;
/*
** TERMINAL
*/
range_expr:
   | DECVAL { let pos,v = $1 in add_pos pos; make_Int v }
   | DECVALLONG { let pos,v = $1 in add_pos pos; make_Int (Int64.to_int
  v) }
   | HEXVAL { let pos,v = $1 in add_pos pos;
              make_Int (int_of_string (concat ["0x";v])) }
   | expr { make_Node $1 }
;
/*
** ROW
*/
index:
   | expr { make_Row (get_type $1) (make_Node $1) }
;
/*
** TYPE/NODE
```

```
205 */
206 identifier:
207   | IDENT  {
208     make_node Expr.Element
209       [
210         make_Row (Expr.Position) (make_Special "POS");
211       ]
212       [|
213        make_Row (make_Special "NAME") (let pos,v = $1 in add_pos pos;
214                                        make_Ident v);
215       |]
216    }
217 ;
218 /*
219 ** TYPE/NODE
220 */
221 element:
222   | BITVAL  {
223     make_node Expr.Value
224       [
225         make_Row (Expr.Format) (Expr.FmBit);
226         make_Row (Expr.Position) (make_Special "POS");
227       ]
228       [|
229         make_Row (make_Special "NAME")
230                  (let pos,v = $1 in add_pos pos; make_String v);
231         make_Row (make_Special "VAL")
232          (let pos,v = $1 in add_pos pos; make_String v);
233       |]
234    }
235   | HEXVAL  {
236     make_node Expr.Value
237       [
238         make_Row (Expr.Format) (Expr.FmBit);
239         make_Row (Expr.Position) (make_Special "POS");
240       ]
241       [|
242         make_Row (make_Special "NAME")
243                  (let pos,v = $1 in add_pos pos; make_String v);
244         make_Row (make_Special "VAL")
245                  (let pos,v = $1 in add_pos pos; make_String v);
246       |]
247    }
248   | BITVEC  {
249     make_node Expr.Value
250       [
251         make_Row (Expr.Format) (Expr.FmBit);
252         make_Row (Expr.Format) (Expr.FmVector);
253         make_Row (Expr.Position) (make_Special "POS");
254       ]
255       [|
256         make_Row (make_Special "NAME")
257                  (let pos,v = $1 in add_pos pos; make_String v);
258         make_Row (make_Special "VAL")
```

```
259                 (let pos,v = $1 in add_pos pos; make_String v);
260       |]
261    }
262   | DECVAL  {
263     make_node Expr.Value
264       [
265         make_Row (Expr.Format) (Expr.FmDecimal);
266         make_Row (Expr.Position) (make_Special "POS");
267       ]
268       [|
269         make_Row (make_Special "NAME")
270                  (let pos,v = $1 in add_pos pos;
271                   make_String (string_of_int v));
272         make_Row (make_Special "VAL")
273                  (let pos,v = $1 in add_pos pos; make_Int v);
274       |]
275    }
276   | STRING  {
277     make_node Expr.Value
278       [
279       ]
280       [|
281         make_Row (make_Special "NAME")
282                  (let pos,v = $1 in add_pos pos; make_String v);
283         make_Row (make_Special "VAL")
284                  (let pos,v = $1 in add_pos pos; make_String v);
285       |]
286    }
287   | CHAR  {
288     make_node Expr.Value
289       [
290       ]
291       [|
292         make_Row (make_Special "NAME")
293         (let pos,v = $1 in add_pos pos; let s = " " in s.[0] <- v;
294          make_String s);
295         make_Row (make_Special "VAL")
296                  (let pos,v = $1 in add_pos pos; make_Char v);
297       |]
298    }
299   | FLOAT  {
300     make_node Expr.Value
301       [
302         make_Row (Expr.Format) (Expr.FmFloat);
303         make_Row (Expr.Position) (make_Special "POS");
304       ]
305       [|
306        make_Row (make_Special "NAME") (let pos,v = $1 in add_pos pos;
307                                        make_String (string_of_float v));
308         make_Row (make_Special "VAL")
309                  (let pos,v = $1 in add_pos pos; make_Float v);
310       |]
311    }
312   | IDENT  {
```

```
313     make_node Expr.Element
314       [
315       ]
316       [|
317         make_Row (make_Special "NAME")
318                  (let pos,v = $1 in add_pos pos; make_Ident v);
319       |]
320    }
321   | IDENT selector_list  {
322     make_node Expr.Element (get_rows_list ())
323       [|
324         make_Row (make_Special "NAME")
325                  (let pos,v = $1 in add_pos pos; make_Ident v);
326       |]
327    }
328 ;
329 /*
330 ** TYPE/NODE
331 */
332 expr:
333   | element  { $1 }
334   | expr operation expr  {
335     make_node Expr.Expr
336       [
337         make_Row (Expr.Operator) ($2);
338       ]
339       [|
340         make_Row (make_Special "NAME") (make_Special "ID");
341         make_Row (Expr.Expr) (make_Node $1);
342         make_Row (Expr.Expr) (make_Node $3);
343       |]
344    }
345 ;
346 /*
347 ** TERMINAL
348 */
349 operation:
350   | ADD { Expr.Add }
351   | SUB { Expr.Sub }
352   | MUL { Expr.Mul }
353   | DIV { Expr.Div }
354   | LAND { Expr.Land }
355   | LOR { Expr.Lor }
356   | LNOR { Expr.Lnor }
357   ...
358   | GE { Expr.Ge }
359 ;
360 /*
361 ** LIST OF ROW
362 */
363 selector_list:
364   | selector { init_rows (); add_row $1 }
365   | selector selector_list { add_row $1; }
366 ;
```

```
367 /*
368 ** LIST OF ROW
369 */
370 index_list:
371   | index ARGSEP  index_list_entry { add_row $1; }
372 ;
373 index_list_entry:
374   | index  { init_rows (); add_row $1; }
375   | index ARGSEP  index_list_entry { add_row $1; }
376 ;
377 /*
378 ** ROW
379 */
380 selector:
381   | SEL identifier { make_Row (Expr.Struct) (make_Node $2) }
382   | LBRAK range RBRAK { $2 }
383   | SEL LBRAK index_list RBRAK {
384       make_Row (Expr.Index) (make_Rows (get_rows_list ())) }
385   | SEL LBRAK index RBRAK {
386       make_Row (Expr.Index) (make_Rows (get_rows_list ())) }
387 ;
388
```


T Language [LaCo/VDB]

Principles, language definition, and Application Programming Interface

1. Introduction

The T language is used to specify formatted printers of structured content (for both inode and record type representations), and is part of the LaCo compiler. A source file with file suffix `tra` consists of different sections, explained in definition 1. A T language file requires an already processed structure graph defintion S. The T file is translated by the LaCo compiler into ML code: a printer file with suffix `_printer.ml` .

Definition 1. T language file structure and sections

```
TRANSLATOR <identifier>;
STRUCTURE <identifier>;
DESCRIPTION "<text>";
SYMBOLS
BEGIN
  <sym-translate>+
END;
ATTRIBUTES
BEGIN
  <attr-translate>+
END;
TYPES
BEGIN
  <type-translate>+
END;
```

The first statement defines the translator name, the second statement binds the translator to an existing structure. A description text follows the second statement. First all symbol translations are defined in section `SYMBOLS`.
The `ATTRIBUTES` section defines translation rules for attributes, and the `TYPES` section defines translation for types (structure elements).

2. Syntax

The formal definitons of the syntax of the T language are shown in defintion 2.

Definition 2. Formal syntax definition of the T language

```
sym-translate ::= identifier ':' create-text .
attr-translate ::= attr-conditional | attr-unconditional | attr-list .
attr-unconditional ::= identifier ':' rules .
```

```
attr-conditional ::= identifier '=' (identifier | value) ':' rules .
rules ::= (BEGIN rule+ END ';') | rule .
rule ::= create-text | .. .
create-text ::= 'CREATE' (token // ' ') [ position ] ';' .
token ::= text | get | Value .
get ::= 'GET' '(' (identifier | any) ')' .
any ::= '*' .
text ::= '"' STRING '"' .
position ::= 'AFTER' | 'BEFORE' .
```

3. Example

The following example defines a formatted printer for the expression structure graph.

Example 1. T input file expr.tra

```
TRANSLATOR EXP;
STRUCTURE EXP;
DESCRIPTION "Expression Pretty Printer";
--
--
--
SYMBOLS
BEGIN
  add: CREATE "+";
  band: CREATE "band";
  bnot: CREATE "bnot";
  bor: CREATE "bor";
  bnor: CREATE "bnor";
  bnand: CREATE "bnand";
  bxor: CREATE "bxor";
  ...
  up: CREATE "to";
  us: CREATE "microsec";
END;
ATTRIBUTES
BEGIN
  --
  -- VALUE(<ref>,<format>)
  --  <format>: DEC|HEX|BIN|BOOL|CHAR|STRING|FLOAT|EXP
  --
  a=INT: CREATE VALUE(INT,%d);
  a=element: CREATE GET(element);
  a=expr: CREATE "(" GET(expr) ")";
  b=INT: CREATE VALUE(INT,%d);
  b=element: CREATE GET(element);
  b=expr: CREATE "(" GET(expr) ")";

  direction=up: CREATE "to";
  direction=down: CREATE "downto";

  freq:
  BEGIN
```

```
  CREATE VALUE([1] INT,%d);
  IF DEFINED([2]) THEN
    CREATE " " GET([2]);
END;

--
-- alternative
--
-- freq[1]: CREATE VALUE(INT,%d);
-- freq[2]: CREATE " " GET(*);
--

guarded=true: CREATE "guarded";
--
-- Rule for an element of a list (*)
--
index*:
BEGIN
  IF FIRST THEN
    CREATE "[" BEFORE;
  IF LAST THEN
    CREATE "]" AFTER;
  IF INT THEN
    CREATE VALUE (INT,%d);
  IF element THEN
    CREATE GET(element);
  IF expr THEN
    CREATE "(" GET(expr) ")";
  IF NOT LAST THEN
    CREATE  ",";
END;

operator: CREATE GET(*);
position: REMOVE;
range: CREATE "[" GET(a) " " GET(direction) " " GET(b) "]";

--
-- Rule for an element of a list (*)
--
-- If the structure contains more elements than the list,
-- the row selector must be used.
--
-- Example:
--          (2) S: s:=l+; s:=(a|b|c)+; ...
--              T: s*: ...;
--          (1) S: s:=l+,x; s:=(a|b|c)+,x; ...
--              T: s[1]*: ...;
--
selector*:
BEGIN
  CREATE "." BEFORE;
  CREATE GET(*);
END;
```

```
  size: CREATE VALUE(INT,%d);
  struct: CREATE VALUE(IDENT,%s);
  time[1]: CREATE VALUE(INT,%d);
  time[2]: CREATE " " GET(*);

END;
TYPES
BEGIN
  element: CREATE VALUE(IDENT,%s);
  element (selector): CREATE VALUE(IDENT,%s) GET(selector);
  expr* (operator):
  BEGIN
    IF FIRST AND LAST THEN
      CREATE GET(operator);
    IF FIRST AND NOT LAST AND DEPTH(expr) > 1 THEN
      CREATE "(";
    CREATE GET(*);
    IF NOT LAST THEN
      CREATE "  " GET(operator) " ";
    IF LAST AND NOT FIRST AND DEPTH(expr) > 1 THEN
      CREATE ")";
  END;

  value[INT] (format=FmChar): CREATE "'" VALUE(INT,%c) "'";
  value[INT] (format=FmDec): CREATE VALUE(INT,%d);
  value[INT] (format=FmBit): CREATE "0b" VALUE(INT,%b);
  value[INT] (format=FmHex): CREATE "0x" VALUE(INT,%x);
  value[INT] (format=FmFloat): CREATE VALUE(INT,%r);
  value[INT] (format=FmString): CREATE "'" VALUE(INT,%s) "'";
  value[INT] (format=FmBoolean):
  BEGIN
    IF [1] = Int 0 THEN CREATE  "false"
    ELSE CREATE "true";
  END;
  value[CHAR] (format=FmChar): CREATE "'" VALUE(CHAR,%c) "'";
  value[STRING] (format=FmString): CREATE "'" VALUE(STRING,%s) "'";
  value[FLOAT] (format=FmFloat): CREATE VALUE(FLOAT,%r);
  value[Bool TRUE] (format=FmBool): CREATE "true";
  value[Bool FALSE] (format=FmBool): CREATE "false";
  value[STRING] (format=FmBit): CREATE "0b" VALUE(STRING,%b);
END;
```