# SPECIFICATION OF A LANGUAGE-BASED EDITOR FOR HASKELL⁻ USING A GENERATOR SYSTEM

Taufiq Hidayat

Project Report
International Master Program
Computational Logic

Start : 21.11.2003, Submitted : 19.05.2004

Professur Grundlagen der Programmierung
Prof. Dr.-Ing. habil. Heiko Vogler
Fakultät Informatik
Technische Universität Dresden

Supervisor:
Dipl.-Inf. Enrico Bormann

# Contents

1. Introduction ..... 3
2. Dresden Specification Language ..... 3
3. Haskell⁻ Language ..... 5
   3.1. Context-free Syntax ..... 5
   3.2. Context-sensitive Restrictions ..... 7
   3.3. A Subclass First-order Haskell⁻ ..... 9
   3.4. Implementation of Haskell⁻ ..... 9
      3.4.1 Call-by-Value Strategy for First-Order Haskell⁻ ..... 10
      3.4.2. Call-by-Name Strategy for First-Order Haskell⁻ ..... 10
      3.4.3. Call-by-Need Strategy for Haskell⁻ ..... 11
4. The Specification of Language-based Editor for Haskell⁻ ..... 13
   4.1. Abstract Syntax ..... 13
   4.2. Unparsing ..... 16
      4.2.1. A Haskell⁻ Code ..... 17
      4.2.2. Abstract Machine Code for Call-by-value Implementation ..... 18
      4.2.3. Abstract Machine Code for Call-by-name Implementation ..... 19
      4.2.4. Abstract Machine Code for Call-by-need Implementation ..... 19
   4.3. Attribution ..... 20
      4.3.1. Context-Sensitive Restrictions ..... 20
      4.3.2. Abstract Machine Code ..... 28
   4.4. Logic Description ..... 30
   4.5. Structural Transformation ..... 32
   4.6. Parsing ..... 35
   4.7. Auxiliary Functions ..... 38
5. How To Use The Editor ..... 39
6. Conclusion ..... 41
References ..... 42

# 1. Introduction

The objective of this project is to build the specification of a language-based editor for Haskell$^-$ using a generator system. The editor should be able to edit a Haskell$^-$ program that is structurally correct. This correctness is according to context-free syntax for Haskell$^-$. Furthermore, the editor should be able to check whether the program fulfils the context-sensitive restrictions. If the program does not fulfil some restrictions the editor should be able to give appropriate messages in the appropriate places. Textual inputs for certain part of the program is another ability of the editor. With this facility a user of the editor is allowed to give some parts of the program as textual input.

The additional facility of the editor is to translate the edited program into abstract machine code. In [Küh00] there are there abstract machines that depend on the implementation strategy. There are call-by-value, call-by-name, and call-by-need strategy. The editor built in this project should be able to translate the Haskell$^-$ program into abstract machine code for all strategies. Because both call-by-value strategy and call-by-name strategy is not for a general program, only for sub-language first-order Haskell$^-$, the editor should be able to check whether the translated program is first-order. The definition of sub-language first-order Haskell$^-$ will be discussed in chapter 2.

The specification language that will be used in the specification of the editor is DSL (Dresden Specification Language). With this language and its generator, we can build a language-based editor for a particular language by specifying the language using DSL. The main part of DSL is an attribute grammar. With the attribute grammar, we can specify context-free syntax and context-sensitive restrictions of the language of the editor. Furthermore, the grammar is integrated with an extension of MSO-logic over trees.

The structure of this report is follows. Chapter 1 is introduction talking about the objective of this project. Chapter 2 describes the DSL in briefly. Chapter 3 explains the Haskell$^-$ language. The specification of a language-based editor for Haskell$^-$ is explained in Chapter 4. Chapter 5 tells about how to use the editor generated from the specification. Finally, the conclusion is explained in Chapter 6.

# 2. Dresden Specification Language

As language, DSL has context-free syntax and context-sensitive restriction. [Bor03] talks about the syntax and restriction of DSL. Generally, a DSL specification consists of some blocks, i.e.: Data Block, Mag Block, MSO Block, Lexical Block, Parse Block, Input Block, and Haskell Code.

Here is the structure of DSL specification for a particular language L. The structure of the specification is:

1. Abstract syntax

This part is defined in the Data Block. The abstract syntax specifies the structure of the language L by means of an algebraic data type. The algebraic data type is related to a context-free grammar that defines the context-free syntax of the language L.
A tree constructed by the abstract syntax is called an abstract syntax tree. The object edited by the language-based editor is represented by an abstract syntax tree.

2. Attribution
   This part is defined in a Mag Block. The major role of this part is for checking context-sensitive restrictions. With this attribution, we can send information from one node to neighbour nodes of the abstract syntax tree.
   This part consists of rules to send information from a node to another node. There are two types of attribution, i.e.: inherited and synthesized. Roughly speaking, in the tree context the inherited attribution is obtained from *above* whereas the synthesized attribution is obtained from *below*.

3. Logic description
   The role of this part is similar to attribution. We use it for checking context-sensitive restriction. The main concept of logic description is to check whether the abstract syntax tree fulfils some logic formulas. The logic used in DSL is MSO* logic.

4. Unparsing
   This part is also defined in a Mag Block. This part is used to give textual output from the abstract syntax trees. Normally, the textual output of the editor is a program in the language L. But, together with attribution and logic, the editor is able to display error message if the program does not fulfil a context-sensitive restriction.

5. Structural Transformation
   In this part, we can define facility to transform a part of a program in language L by others part. If we look at the abstract syntax tree, this part transforms a subtree by other subtree.
   This part consists of rules of possible transformation of a subtree whose root has certain label. Each rule consists of the label, the name of transformation, and the result of transformation.

6. Parsing
   This part is defined in three blocks: Lexical Block, Parse Block, and Input Block. With parsing, we can add facility our editor by textual input. We can define textual input for many parts of the program. Each part of the program as textual input need at least one parse rule. So, this parsing consists of parse rules as many as the parts of the program as textual input.

7. Auxiliary function.
   In this part we can define some auxiliary functions that are usable in Mag Blocks. The auxiliary functions are written in normal Haskell language. The auxiliary functions are defined in Haskell Code block.

# 3. Haskell⁻ Language

## 3.1. Context-free Syntax

Haskell⁻ is a sub-language of Haskell. In Haskell⁻ a program consists of declarations of some data types and functions.
Each data type has at least one constructor. Haskell⁻ allows a definition of recursive data types.

In declarations of the functions, a program should define the type of the functions and an equation for each function. Each function could have many arguments. A case analysis in Haskell⁻ does not use pattern matching. The case analysis only uses case expression. The case expression allows nested case expressions.

```
<prog>   ::=  "module Main where"                    (Programs)
              <data> { <data> }
              <func> { <func> }
              "main = print " <exp>

<data>   ::=  "data " <typid> "="                    (Data Types)
              <conid> { <typid> }
              {"|" <conid> {<typid>}}
              "deriving Show"

<func>   ::=  <funid> "::" <type>                    (Functions)
              <funid> { <varid> } "=" <rhs>

<type>   ::=  <typid> |                              (Function Types)
              "(" <type> {"->" <type>}
                "->" <typid> ")"

<rhs>    ::=  <exp> |                                (Right-hand-side)
              "(case " <varid> "of"
                 <pat> "->" <rhs>
                 {<pat> "->" <rhs>} ")"

<pat>    ::=  <conid> |                              (Patterns)
              "("<conid> <varid> {<varid>}")"

<exp>    ::=  <varid> |                              (Expression)
              <funid> |
              <conid> |
              "(" <conid> <exp> {<exp>} ")" |
              "(" <exp> <exp> ")" |
              "(let" <varid> "=" <exp>
                    {<varid> "=" <exp>}
               "in" <exp> ")"

<typid> ::= ("A"|...|"Z") ("a"|...|"z")
            {("a"|...|"z")}                          (Type Symbols)
<conid> ::=  ("A"|...|"Z") {("A"|...|"Z")}           (Constructors)
<funid> ::=  ("a"|...|"z") {("a"|...|"z")}           (Func. Symbols)
<varid> ::=  "x1" | "x2" | "x3" | ...                (Variables)
```

**Figure 1**. The Context-free syntax of Haskell⁻

Input and output are not available explicitly in Haskell⁻. A Haskell⁻ a program has a fixed expression as input. After the program evaluates the expression, the result of the evaluation of the expression is the output.

Figure 1 is a set of rules of the context-free syntax of Haskell⁻ in EBNF. The non-terminal symbols are enclosed between < and >. The terminal symbols are enclosed between " and ", and the start symbol is <*prog*>.

A *program* in Haskell⁻ is a string generated with the rules starting from the non-teminal symbol <*prog*>.. A program starts with string "module main where" and ends with string "main = print" followed by an expression. This expression is as an input explained earlier. We denote the expression as the *input expression* of a Haskell⁻ program. Between the two strings, we must define data types and functions, respectively.

The identifiers allowed in Haskell⁻ are to denote *type symbols*, *constructors*, *function symbols)*, and *variables*. They are generated with the rules from the non-terminal symbol <*typid*>, <*conid*>, <*funid*>, and <*varid*>, respectively. In addition, the type symbols, the constructors, and the functions are global identifiers, whereas the variables are local ones. We will discuss the identifiers further in context-sensitive restrictions chapter.

**Example 1**

```
module Main where

data Nat  = S Nat | Z deriving Show
data Bool = TRUE | FALSE deriving Show

add :: (Nat -> Nat -> Nat)
add x1 x2 = (case x1 of
               Z      -> x2
               (S x3) -> (S ((add x3) x2)))

ifthenelse :: (Bool -> Nat -> Nat -> Nat)
ifthenelse x1 x2 x3 = (case x1 of
                         TRUE   -> x2
                         FALSE  -> x3)

main = print (((ifthenelse TRUE) ((add (S Z)) Z)) (S (S Z)))
```

□

Example 1 is a program in Haskell⁻. The program has two data types (Nat, Bool), four constructors (S, Z, TRUE, FALSE), and two functions (add, ifthenelse). The input of the program is a string expression appearing after string *print*. Thus, the *input expression* of the program is *(((ifthenelse TRUE) ((add (S Z)) Z)) (S (S Z)))*.

## 3.2. Context-sensitive Restrictions

We find some context-sensitive restrictions for a program in Haskell⁻ [Küh00], pages: 22 – 24, in formal definition. In this chapter the context-sensitive restrictions are depicted with words.

Before we discuss the context-sensitive restriction, we define a *type* of *expressions*, *right-hand-sides*, *function*, *constructors*, or *variables*. We define recursively what we understand under a *type*:

1. Every defined *data type* is a *type*.
2. Let $t_1, \ldots, t_n$ be *types* and $t_{n+1}$ a defined *data type*, where $n \geq 1$, then $(t_1 \rightarrow ... \rightarrow t_n \rightarrow t_{n+1})$ is a *type*.

The *type* of function *f* that is defined as "f :: t" is t. Whereas the *type* of a constructor *c* that is defined as "data d = ... | c $d_1$ ... $d_n$ | ..." is $(d_1 \rightarrow ... \rightarrow d_n)$. It is noted that $d_1$, ..., and $d_n$ are *data types*. The rank of the constructor *c* is n.

A program p in Haskell⁻ should fulfil some following restrictions:

1. Every *type symbol*, *constructor*, and *function symbol* should have at most one definition as *data type*, in a *data type*, and as *functions*, repectively. By other words, they should be unique.
2. Every *type symbol*, *constructor*, and *function symbol* occurring in a program p should be defined.
3. After the type definition of a function *f*, it is followed by the equation of the function.
4. If function f has *type* $(t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_n \rightarrow t_{n+1})$ and function f has list of parameters $x_1, ..., x_m$, where $0 \leq m \leq n$, then the right-hand side of the function f has *type* $(t_{m+1} \rightarrow ... \rightarrow t_{n+1})$. The right-hand side of the equation of the function can contain variables $x_1, ..., x_m$ where $x_1, ..., x_m$ have *types* $t_1, ..., t_m$, respectively.
5. If function *f* has *type* t where t is a defined *data type*, then the function should have no parameter and the right-hand-side of the equation has *type* t, too. We call the function *f* a *nullary function* and the rank of *f* is 0.
6. The input expression does not contain let expression. The *type* of the expression should be a *data type*.

**Example 2**

Following apart of a program is the definition of function gpr:

```
gpr :: (Nat → Bool → Nat)
gpr x1 = <rhs>
```

Then, the *type* of <rhs> is (Bool → Nat)

□

The followings are the context-sensitive restrictions for expression ([Küh00], page:23):

1. If an expression is a variable or a function application with *type* t, then the *type* of the expression is t.

2. If an expression is a constructor with rank 0 and its *type* is t, then the *type* of the expression is t.
3. If $e_1$, ..., $e_n$ are expressions with *types* $t_1$, ..., $t_n$, respectively, and c is a constructor with *type* ($t_1 \rightarrow ... \rightarrow t_n \rightarrow t_{n+1}$), then we can construct an expression (c $e_1$ ... $e_n$) and the *type* of the expression is $t_{n+1}$. Actually, $t_1$, ..., $t_{n+1}$ are *data types*.
4. If $e_o$ and $e_1$ are expressions with *type* ($t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_n \rightarrow t_{n+1}$) and $t_1$, respectively, then we can construct an expression ($e_0$ $e_1$) and the *type* is ($t_2 \rightarrow ... \rightarrow t_n \rightarrow t_{n+1}$). In this case, $1 \leq n$. We call the expression *application expression.* The expression is called partial application only if n>1.
5. If there is a following let-expression

```
let x1 = e1
     ...
    xn = en
in e
```

then the expression must fulfil:

a. the variables $x_1$, ..., $x_n$ should not occur in the expressions $e_1$, ..., $e_n$
b. the variables $x_1$, ..., $x_n$ should be unique in the let-expression and new.
c. the variables $x_1$, ..., $x_n$ can occur in the expression e.
d. the *types* of the variables $x_1$, ..., $x_n$ are the *types* of the expressions $e_1$, ..., $e_n$, respectively.
e. the *type* of the let-expression is the *type* of the expression e.

The followings are the context-sensitive restrictions for right-hand side ([Küh00], page:24):

1. Every expression with set of free-variables X is a right-hand side with set of variables X.
2. Let a right-hand side be a following case-expression:

```
(case xk of
     (c1 x1,1 ... x1,m1) -> r1
          ...
     (cn xn,1 ... xn,mn) -> rn)
```

Then the case-expression must fulfil:

a. $r_1$, ..., $r_n$ are right-hand-sides that each has same *type* and the *type* is the *type* of the case-expression.
b. $x_{1,1}$, ..., $x_{1,m_1}$, ..., $x_{n,1}$, ..., $x_{n,m_n}$ are new variables.
c. for every set of variables $\{x_{j,1}, ..., x_{j,m_j}\}$ where 1<=j<=n, each variable in the each set should be unique in the set.
d. for every pattern ($c_j$ $x_{j,1}$ ... $x_{j,m_j}$) and the *type* of constructor $c_j$ is ($t_{j,1} \rightarrow ... \rightarrow t_{j,m_j} \rightarrow t_j$), the *types* of the variables $x_{j,1}$, ..., $x_{j,m_j}$ are $t_{j,1}$, ..., $t_{j,mj}$, respectively.
e. the *type* of variable $x_k$ should be a *data type* and every constructor $c_1$, ..., $c_n$ should be declared in the declaration of the *data type*.
f. there is no constructor that appears in more than one pattern.

### 3.3. A Subclass First-order Haskell$^-$

First-order Haskell$^-$ is a sub-class of Haskell$^-$. A Haskell$^-$ program is called a first-order Haskell$^-$ program, if the program fulfils the following context-sensitive restrictions:

1. If a *function* has *type* ($t_1 \rightarrow ... \rightarrow t_n \rightarrow t_{n+1}$), then $t_1$, ..., $t_n$ should be *data types*.
2. Number of parameters in equation of the function in point 1 is n and the *type* of right-hand sides of the equation is $t_{n+1}$ where $t_{n+1}$ is a *data type*.
3. No partial application allowed. So, if a function f has *type* ($t_1 \rightarrow ... \rightarrow t_n \rightarrow t_{n+1}$) and expressions $e_1$, ..., $e_n$ have *types* $t_1$, ..., $t_n$, respectively, we can only construct an expression:
   $(...(f\ e_1) ... e_n)$

Example 1 is a Haskell$^-$ program in first-order.

### 3.4. Implementation of Haskell$^-$

There are three strategies of implementation of Haskell$^-$ presented in [Küh00]. The strategies are call-by-value, call-by-name, and call-by-need. The strategies are according to the kinds of deterministic reduction relation presented also in [Küh00]. We do not discuss the reduction relations further. We will concern in the implementations.

In this implementation, Haskell$^-$ will be implemented into abstract machine code. An abstract machine performs the evaluation of expression of Haskell$^-$. An abstract machine has a set of instructions. Because the strategies of implementation are different, they need different abstract machines.

According to this project, this chapter will concentrate on how to translate a Haskell$^-$ program into an appropriate abstract machine code. The abstract machine code consists of a sequence of defined instructions. Actually, the translation is only for the defined functions and the *input expression* of the Haskell$^-$ program. Generally, to translate a Haskell$^-$ program all strategies have two steps.

First step is a translation of a Haskell$^-$ program into a abstract machine code with symbolic addresses. A symbolic address can be a name of a defined function or a list of numbers that each pair of the numbers are separated by dot (‘.’), ie. 0.3.4.5. One instruction can have no, one, or more than one symbolic address.

Second step is linearization. The linearization is transforming the abstract machine code with symbolic address into a new one by assigning successive natural numbers started with 1 to the instructions. The numbers substitute the symbolic addresses as label of the instructions and occurring in the instructions themselves. Some different occurrences of the same symbolic addresses are substituted by the same number. If there is a symbolic address occurring in an instruction but does not occur as label then the symbolic address is substituted by 0.

Note that the implementations of call-by-value and call-by-name strategies are only for first-order Haskell$^-$.

### *3.4.1 Call-by-Value Strategy for First-Order Haskell⁻*

The abstract machine according to call-by-value strategy is presented in [Küh00] chapter 5, pages: 41-50. In short description, here are the instructions in this abstract machine:

1. NODE (δ,*n*) : δ is a constructor and *n* is its rank.
2. LOAD *i* : *i* is an index of a local variable
3. STORE *i* : *i* is an index of a local variable
4. CALL (*f*,*n*,*k*) : *f* is an address of first instruction of a function, *n* is the rank of the function, *k* is maximum index of local variable in the function
5. RET
6. JUMP *l* : *l* is an address
7. MATCHORJUMP(δ,*l*) : δ is a constructor and *l* is an address.

We can see the semantic of the instructions in [Küh00], pages:41-42.
For translation some functions are defined. The names of functions are:

1. *trans* : translates a first-order Hakell⁻ program.
2. *functrans* : translates a function
3. *rhstrans* : translates a right-hand-side
4. *pattrans* : translates a pattern
5. *exptrans* : translates an expression.

Example 3 is a translation of the Haskell⁻ program in Example 1 after linearization. Instructions with addresses from 1 to 9 are translation of the input expression. Instructions with addresses from 11 to 21 and from 22 to 28 are translations of function *add* and function *ifthenelse*, respectively.

**Example 3**

The abstract machine code for implementation of call-by-value from Example 1:

```
 1: NODE (Z,0);                  15: MATCHORJUMP (S, 0);
 2: NODE (S,1);                  16: STORE 3;
 3: NODE (S,1);                  17: LOAD 2;
 4: NODE (Z,0);                  18: LOAD 3;
 5: NODE (Z,0);                  19: CALL (11,2,3);
 6: NODE (S,1);                  20: NODE (S,1);
 7: CALL (11,2,3);               21: RET;
 8: NODE (TRUE,0);               22: LOAD 1;
 9: CALL (22,3,3);               23: MATCHORJUMP (TRUE, 26);
10: RET;                         24: LOAD 2;
11: LOAD 1;                      25: JUMP 28;
12: MATCHORJUMP (Z, 15);         26: MATCHORJUMP (FALSE, 0);
13: LOAD 2;                      27: LOAD 3;
14: JUMP 21;                     28: RET;
```

□

### *3.4.2. Call-by-Name Strategy for First-Order Haskell⁻*

Chapter 6 of [Küh00], pages: 51-66, explains deeply the implementation of call-by-name strategy.

In implementation of call-by-name strategy, the instruction set for the abstract machine is an extension of the instruction set for the abstract machine of implementation of call-by-value strategy. The instruction set contains all instructions in implementation of call-by-value strategy and is added by the new following instructions:

1. SUSPEND *l* : *l* is an address.
2. INITIATE
3. UPDATE

We need some functions for translating a first-order Haskell$^-$ program into abstract machine code. In call-by-name, we redefine translation functions in call-by-value implementation, except *functrans* and *pattrans*, and add new function, *delaytrans*. This function translates an expression.

In the definition of *exptrans* function, all arguments of constructor application, all local expression in let-expression, and the argument of function application are translated by the *delaytrans* function.

Example 4 is the abstract machine code of the first-order Haskell$^-$ program in Example 1.

**Example 4**

The abstract machine code for implementation of call-by-name from Example 1:

```
 1: NODE (Z,0);                    21: STORE 3;
 2: NODE (S,1);                    22: SUSPEND 24;
 3: NODE (S,1);                    23: JUMP 28;
 4: SUSPEND 6;                     24: LOAD 2;
 5: JUMP 11;                       25: LOAD 3;
 6: NODE (Z,0);                    26: CALL (14,2,3);
 7: NODE (Z,0);                    27: UPDATE;
 8: NODE (S,1);                    28: NODE (S,1);
 9: CALL (14,2,3);                 29: RET;
10: UPDATE;                        30: LOAD 1;
11: NODE (TRUE,0);                 31: INITIATE;
12: CALL (30,3,3);                 32: MATCHORJUMP (TRUE,36);
13: RET;                           33: LOAD 2;
14: LOAD 1;                        34: INITIATE;
15: INITIATE;                      35: JUMP 39;
16: MATCHORJUMP (Z,20);            36: MATCHORJUMP (FALSE,0);
17: LOAD 2;                        37: LOAD 3;
18: INITIATE;                      38: INITIATE;
19: JUMP 29;                       39: RET;
20: MATCHORJUMP (S,0);
```

□

### *3.4.3. Call-by-Need Strategy for Haskell$^-$*

Chapter 7 of [Küh00], pages: 67-76, explains the implementation of call-by-need strategy.

We discuss here the implementation of call-by-need for Haskell$^-$, not only for first-order Haskell$^-$. Although chapter 6 of [Küh00] also talks the implementation of call-by-

need for first-order Haskell$^-$, we do not discuss it. They are not different in the abstract machine and its instructions. The difference is only the semantic of *UPDATE* instructions. Thus, the abstract machine codes for implementation call-by-name and call-by-need are similar.

In abstract machine for implementation of call-by-need for Haskell$^-$, we add some following instructions:

1. FNODE ($l$,$n$,$k$) : $l$ is an address of first instruction of a function, $n$ is the rank of the function, $k$ is maximum index of local variables in the function
2. APPLY

For translation function, we only redefine the definition of *exptrans* function in implementation of call-by-name. Other functions are not changed. The definition of *exptrans* function is modified with some followings:

1. to translate an expression of a function application. If the rank of the function is 0, then the translation of the expression is CALL instruction. Otherwise, the translation is FNODE instruction.
2. to translate an *application expression*. We add APPLY instruction to the definition of the function in implementation of call-by-name.

Example 5 is the abstract machine code of the Haskell$^-$ program in Example 1.

**Example 5**

The abstract machine code for implementation of call-by-need from Example 1:

```
 1: NODE (Z,0);                    24: JUMP 36;
 2: NODE (S,1);                    25: MATCHORJUMP (S,0);
 3: NODE (S,1);                    26: STORE 3;
 4: SUSPEND 6;                     27: SUSPEND 29;
 5: JUMP 13;                       28: JUMP 35;
 6: NODE (Z,0);                    29: LOAD 2;
 7: NODE (Z,0);                    30: LOAD 3;
 8: NODE (S,1);                    31: FNODE (19,2,3);
 9: FNODE (19,2,3);                32: APPLY;
10: APPLY;                         33: APPLY;
11: APPLY;                         34: UPDATE;
12: UPDATE;                        35: NODE (S,1);
13: NODE (TRUE,0);                 36: RET;
14: FNODE (37,3,3);                37: LOAD 1;
15: APPLY;                         38: INITIATE;
16: APPLY;                         39: MATCHORJUMP (TRUE,43);
17: APPLY;                         40: LOAD 2;
18: RET;                           41: INITIATE;
19: LOAD 1;                        42: JUMP 46;
20: INITIATE;                      43: MATCHORJUMP (FALSE,0);
21: MATCHORJUMP (Z,25);            44: LOAD 3;
22: LOAD 2;                        45: INITIATE;
23: INITIATE;                      46: RET;
```

□

# 4. The Specification of Language-based Editor for Haskell⁻

To explain the specification of the language-based Editor for Haskell⁻, we divide this chapter according to the structure of a specification in DSL. This structure is explained in chapter 2.

## 4.1. Abstract Syntax

This part talks about the abstract syntax of Haskell⁻. The abstract syntax is defined in a *data block*. The data block consists of the block name and a list of data definitions and is enclosed between two string 'begindata' and 'enddata'. The data definition specifies the rules of an algebraic data type defining the abstract syntax of Haskell⁻.

Mainly, the algebraic data type is obtained by taking a production of context-free syntax, omitting the terminal symbols, and giving the production a name. However, if we consider the context-free syntax of Haskell⁻ in Figure 1, we find that there are *meta symbols* in some productions. For, these *meta symbols* we create new algebraic data types describing the lists. For example take the production of non-terminal symbol <*prog*>. The algebraic data types for the production are as follows:

```
Program         = PROG DataList FunctionList Exp
DataList        = DECLSINGLE DataDef
                | DECLPAIR DataDef DataList
FunctionList    = FUNCSINGLE Function
                | FUNCPAIR Function FunctionList
```

The two last data types represent the *meta symbols* for *data types* and *functions*, respectively.

From the algebraic data types, we define the data definitions in the Data Block. Each data definition defines an algebraic data type. A data type contains some constructors. Every constructor is followed by list of its successor types.

For some data types, some constructors that are not related with the algebraic data types are defined. The constructors are for some special purposes. We will discuss the constructors and their purposes.

As explained in chapter 1, the editor should be able to translate the edited program into abstract machine codes for all implementations. Thus, the textual outputs of this editor are a Haskell⁻ code and abstract machine codes. For this problem, we need information to refer to which textual output will be displayed. It is solved by adding some new constructors to the data type *Program*. Each constructor refers to which textual output will be displayed from a Haskell⁻ program represented by an abstract syntax tree.

The constructors added to the data type *Program* are *PROG, PROGNOCSR*, *IMPLCALLVALUE*, *IMPLCALLNAME*, and *IMPLCALLNEED*, such that the data definition for the data type *Program* is follows:

```
data Program    = PROG DataList FunctionList Exp
                | PROGNOCSR DataList FunctionList Exp
```

```
                | IMPLCALLVALUE DataList FunctionList Exp
                | IMPLCALLNAME DataList FunctionList Exp
                | IMPLCALLNEED DataList FunctionList Exp
```

The three last constructors, *IMPLCALLVALUE*, *IMPLCALLNAME*, and *IMPLCALLNEED*, refer to the abstract machine codes for call-by-value, call-by-name, and call-by-need implementations, respectively. The constructor *PROG* refers to the Haskell$^-$ code which should also show the error messages for the context-sensitive restrictions. The constructor *PROGNOSCR* is an additional output that refers to the Haskell$^-$ code without the error messages.

Since the data type *Program* consists of five constructors, we can not use the data type as the root type although the data type is related with the start symbol in the context-free syntax of Haskell$^-$. That does not fulfil the restriction of the root type that should consist of one constructor [Bor03]. As the root type, we create a new data type *root*. The data type only consists of one constructor *ROOT*. The successor type of the constructor is Program. The definition of the root type as follows:

```
data Root       = ROOT Program
```

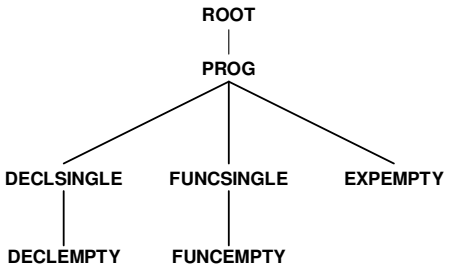


**Figure 2.** The Initial Abstract Syntax Tree.

To represent unspecified part of a Haskell$^-$ program, the some new placeholder constructors are defined. The placeholder constructors have name with suffix *EMPTY* or *NULL*. Some of the constructors are put as first constructors. It makes the constructors as default constructors. The initial abstract syntax tree will be constructed with these constructors. The following are the data definitions having the placeholder constructors as example:

```
data DataDef    = DECLEMPTY
                | DECLARATION Typid ConsList

data Function   = FUNCEMPTY
                | FUNC Funid Type ParamList Rhs

data Exp        = EXPEMPTY
                | VAREXP Varid
                | FUNCEXP Funid
                | CONEXP Conid
                | CONEXPPAR Conid ExpList
                | EXPPAIR Exp Exp
                | LETEXP AssExpList Exp
```

Figure 2 is the initial abstract syntax tree regarding the given data definitions.

Some placeholder constructors are needed in transformation that replaces a subtree by another subtree. The replacing subtree usually contains some placeholder constructors. An example is a transformation from *EXPEMPTY* in the initial abstract syntax tree (Figure 2) into a subtree that represents an expression consisting of a constructor with n rank. In this case, we need one placeholder constructor for both data types, *Conid* and *ExpList*. The definitions of the two data types as follows:

```
data ExpList    = EXPLISTEMPTY
                | EXPSINGLE Exp
                | EXPLISTPAIR Exp ExpList

data Conid      = CONIDNULL
                | CONIDNAME String
```

The result of transformation is showed by Figure 3.

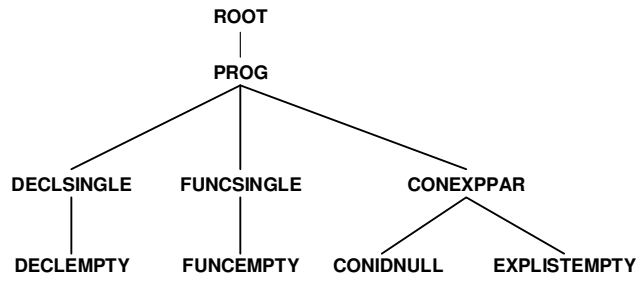


**Figure 3**. An Abstract Syntax Tree.

Figure 4 and Figure 5 are the abstract syntax trees for some parts of Haskell⁻ program in Example 1. Figure 4 is the abstract syntax tree with a triangle representing a subtree whose type is denoted between symbol '{' and '}'. The left picture in Figure 5 is a subtree for declaration of *data types*, whereas the right one is a subtree for the *input expression*.

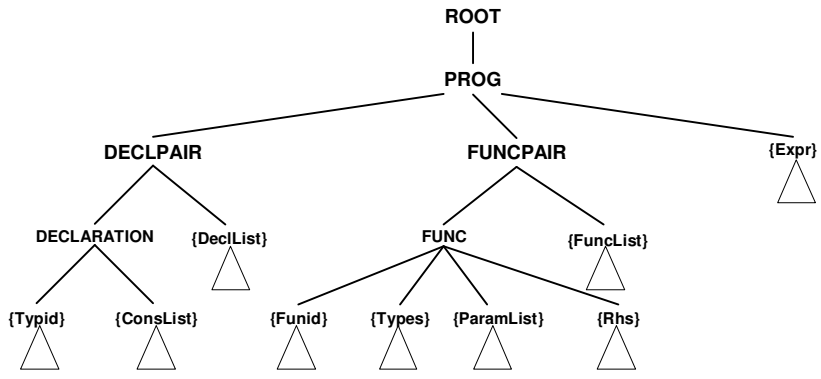


**Figure 4**. The Abstract Syntax Tree of the Haskell⁻ program in Example 1.

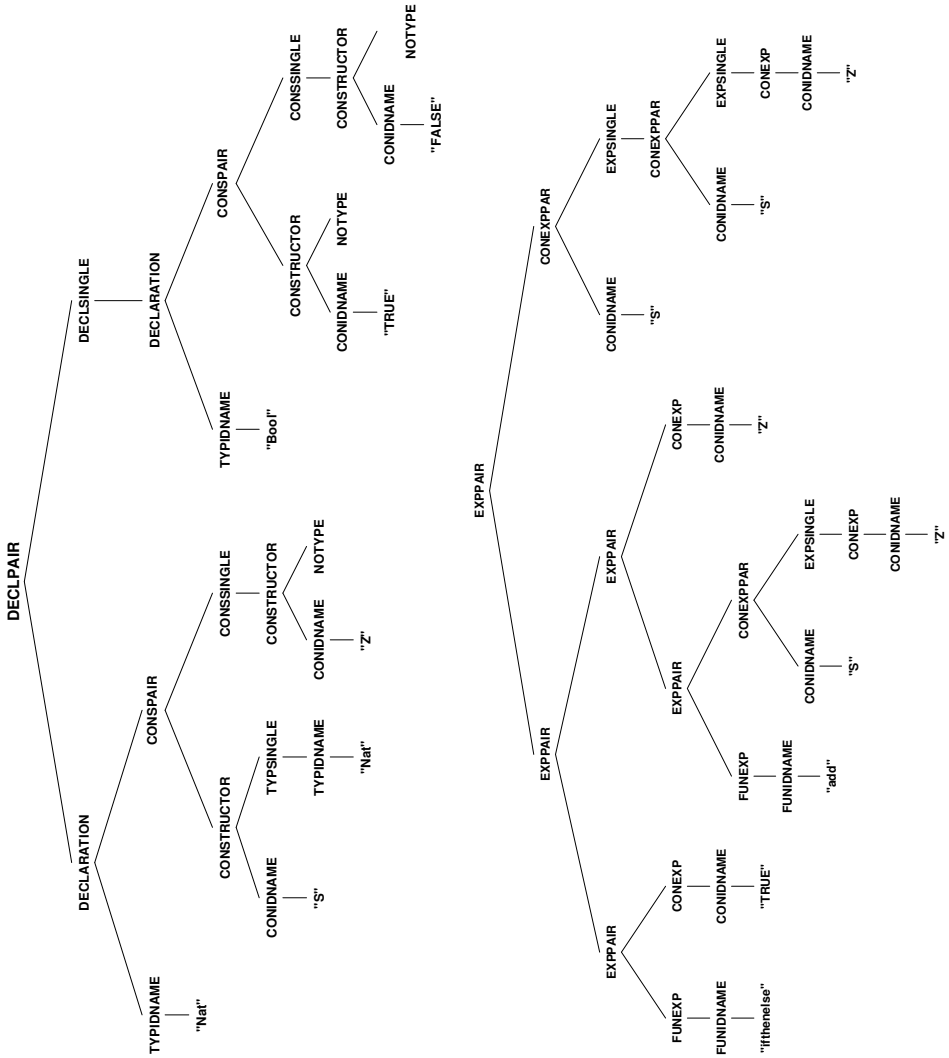


**Figure 5**. The Abstract Syntax Trees of data types (left) and the *input expression* (right) of the Haskell⁻ program in Example 1.

## 4.2. Unparsing

In this part, we want to translate the abstract syntax tree as internal representation into external representation being textually presented to the user. The external representation is in plain text. For the translation we need a recursive function translating an abstract syntax tree into string. We call this function a synthesized attribute.

Since in this project an abstract syntax tree represents a Haskell⁻ program, the unparsing translates the abstract syntax tree into a Haskell⁻ code. The Haskell⁻ code also contains the error messages according to the context-sensitive restrictions. However, the error messages can be suppressed. In addition, the unparsing also

translates the abstract syntax tree into three abstract machine codes for implementation of call-by-value, call-by-name, and call-by-need. We will discuss further each needed textual representations in following subchapters.

### *4.2.1. A Haskell⁻ Code*

Here we will translate an abstract syntax tree into a Haskell⁻ code. We need a synthesized attribute over the abstract syntax tree. The synthesized attribute is a plain text consisting of terminal symbols of the context-free syntax of Haskell⁻ and error messages according to the context-sensitive restrictions of Haskell⁻.

The plain text consists of the error messages if some context-sensitive restrictions are not fulfilled. The error messages will appear whenever the user of this editor edits a Haskell⁻ program. However the checking of the context-sensitive restrictions needs running time. It makes the editing process slower.

We offer two options to solve the problem. First option, the textual representation consists of the error messages whereas in second option the textual representation does not consist of the error messages.

We need two synthesized attributes to translate an abstract syntax tree to Haskell⁻ code. One attribute is for the first option, and another attribute is for the second option. The attributes are over all data types. The definitions of the synthesized attributes are:

```
unparse up :: a -> String
unparse upwe :: a -> String
```

where a denotes the *polymorphic* type.

The synthesized attribute *up* consists of the Haskell⁻ code with the error messages if they exist, and the synthesized attribute *upwe* consists of the Haskell⁻ code without the error messages.

As example showing the difference between both synthesized attributes is the attributions for a subtree whose label is *CONSTRUCTOR*. This subtree represents a definition of a constructor. The attribute *upwe* of the subtree is a string consisting of the name of the constructor and its successor types, whereas the attribute *up* of the substree is a string consisting of the name of the constructor, appropriate error message if a same other constructor is defined, and its successor types. The rule of both attributes as follows, where *t1* is a subtree representing the name of the constructor and *t2* is a subtree representing its successor types:

```
up (CONSTRUCTOR t1 t2)= (up t1) ++ " " ++
                         (check
                            " {mulitple declared constructor} "
                            (en_is_cons_uniq t t1)) ++
                         (up t2)

upwe (CONSTRUCTOR t1 t2)= (upwe t1) ++ " " ++
                           (upwe t2)
```

In the rule of attribute *up*, we found the auxiliary function *check*. The inputs of the function are a string as an error message and a bool as checking of context-sensitive restriction. The function will send the error message if the checking is negative. Otherwise, the function will send an empty string.

There is also an inherited attribute, *en_is_cons_uniq*, in the rule. The attribute shows whether the constructor *t1* is unique.

### *4.2.2. Abstract Machine Code for Call-by-value Implementation*

This translation is from an abstract syntax tree to the abstract machine code for call-by-value implementation. In this case, we should consider the *translation* functions from a Haskell$^-$ program to the abstract machine for call-by-value implementation. The *translation* functions are briefly explained in the subchapter 3.4.1.

Although we have some *translation* functions, we only need one synthesized attribute to translate an abstract syntax tree. The rules of the synthesized attribute can define the *translation* functions since from the subtree's root of an abstract syntax tree we can identify which *translation* function should be applied. The name of the synthesized attribute is *upv*.

Since the translation from a Haskell$^-$ program to abstract machine code is only applied to the definitions of functions and the *input expression*, the synthesized attribute *upv* is over subtrees representing the input expression and the definitions of functions. The definition of the attribute as follows:

```
unparse upv :: a -> String
```

In this translation we need information to give an address to each instruction in abstract machine code. We need attributes showing the first address and the next address of an abstract machine code translated from a subtree of an abstract syntax tree. The attributes are *addrv*, and *nextaddrv*. This attributes will be discussed in chapter 4.3.2.

An example of rules of the synthesized attribute *upv* is for translation of *case-expression*. A *case-expression* consists of a variable that will be matched and a list of pairs of a pattern matching and a right-hand side. In an abstract syntax tree, a *case-expression* is represented by a subtree whose root has a label *CASEEXP* whereas the list is represented by a subtree whose root has a label either *CASESINGLE* or *CASEPAIR*. The rule for the attribution as follows:

```
t@ (CASEEXP t1 t2) in
    upv   t     = (show (addrv t)) ++ " : LOAD " ++ (upv t1)
                  ++ ";\n" ++ (upv t2)

t@ (CASESINGLE t1) in
    upv   t     = (upv t1)

t@ (CASEPAIR t1 t2) in
    upv   t     = (upv t1) ++ ";\n" ++
                  (show (nextaddrv t1)) ++ " : JUMP " ++
                  (show (nextaddrv t2)) ++ ";\n" ++ (upv t2)
```

### *4.2.3. Abstract Machine Code for Call-by-name Implementation*

Chapter 3.4.2 explains a translation from a Haskell⁻ program to an abstract machine code for call-by-name implementation. We have also some *translation* functions. Similar to the unparsing in abstract machine code for call-by-value implementation, we need a synthesized attribute consisting of the translation of an abstract syntax tree to abstract machine code for call-by-name implementation. The name of the synthesized attribute is *upn*.

However, the attribute can not cover the *delaytrans* function that translates an expression. We should define another synthesized attribute to handle the problem. This attribute is only over the subtrees representing expressions. Thus, the attribute is a string consisting of abstract machine code obtained by the *delaytrans* function. The name of the attribute is *upndl*.

The definitions of the two attributes as follows:

```
unparse upndl :: Exp -> String
unparse upn :: a -> String
```

We also need attributes showing the first address and the next address of an abstract machine code translated from a subtree of abstract syntax tree. The attributes are *addrn*, *nextaddrn*, *addrndl*, and *nextaddrnndl*. The two lasts are attributes needed in delay translation function *upndl*. We should use the different attributes because the numbers of instructions in translation of an expression between the attributes *upndl* and *upn* are different.

### *4.2.4. Abstract Machine Code for Call-by-need Implementation*

Chapter 3.4.3 explains a translation from a Haskell⁻ program to an abstract machine code for call-by-need implementation. Since the translation does not have much difference with the translation to an abstract machine code for call-by-name, we do not define new unparsing. So, the synthesized attributes *upn* and *upndl* can consist of the abstract machine code for call-by-name implementation or call-by-need implementation.

According to the translation function in Chapter 3.4.3, the different translations for call-by-name implementation and call-by-need implementation are only in translation of a function application and in translation of an application expression. With this idea, the attributes *upn* and *upndl* have also two different unparsings for subtree representing the function application and the application expression. These unparsings depend on the implementation.

To make the appropriate unparsings for attributes *upn* and *upndl*, we need information to denote which implementation is really applied. We define a MSO* logic formula *?call_by_need* to denote call-by-need implementation and an inherited attribute *callbyname* to denote call-by-name implementation. Actually, the information of the inherited attribute is obtained by MSO* logic formula *?call_by_name*. The formulas are

explained in chapter 4.4. Here are the rules of the attribute *upn* for subtress whose labels are *FUNIDNAME* and *EXPPAIR*, both represent a *function application* and an *application expression*, respectively:

```
t@ (FUNIDNAME t1) in
   upn t  = (if_then_else
               (callbyname t)
               (" : CALL " ++ (show (env_tab t t)))
               (" : " ++ (ins_fc_need (env_tab t t))))

t@ (EXPPAIR t1 t2) in
   upn t  = (upndl t2) ++ ";\n" ++
            (upn t1) ++
            (if_then_else
               (?call_by_need t1)
               (";\n" ++ (show (nextaddrn t1)) ++ " : APPLY")
               "")
```

The auxiliary function *ins_fc_need* determines the instruction for function application, either *CALL* or *FNODE*, if the implementation is call-by-need. The auxiliary function *if_then_else* gives the second argument if the first argument *true*, otherwise gives the third argument.

## 4.3. Attribution

The attribution is defined in a Mag Block. With this attribution we can transfer information over the abstract syntax tree that is defined in the abstract syntax. The transfer of information can be implemented by means of *inherited* and *synthesized* attributes.

A checking of context-sensitive restriction uses the attributes since a part of language can check some context-sensitive restrictions if it can receive some needed information.

In this project, the attributes will be applied in checking of context-sensitive restriction and in process of translation to abstract machine code, especially in translation of function application and linearization of abstract machine code.

### *4.3.1. Context-Sensitive Restrictions*

Many attributes to the abstract syntax tree are related with the context-sensitive restrictions for a Haskell$^-$ program. In Chapter 3.2, we see the context-sensitive restrictions. To check whether some parts of a Haskell$^-$ program fulfil the related context-sensitive restriction, they should refer to other parts of the program. As example, if in the *input expression* of the program contains a constructor then we must refer to the definition of data types to check whether the constructor is defined or not.

First step in attribution, we identify the context-sensitive restrictions that should be fulfilled in a part of a Haskell$^-$ program. The next step is to determine the needed information and in which part the information is available or can be generated. According to the abstract-syntax tree relating with a Haskell$^-$ program, we should observe how the information will be transferred.

We will show the attribution for some context-sensitive restrictions. In attributions, we only discuss the main attributes. We omit some supporting attributes such that we can concern on the main idea of the attributions. Here are the context-sensitive restrictions that use attributions to checking:

1. Every *type symbol* is defined.
   A type symbol is defined in the *data types* part as the name of a *data type*. A type symbol will occur in the definition of constructors. The *type symbol* follows the constructors as successor type. A type symbol also occurs in the type of the function definition.

   In Example 1, the *type symbol Nat* is a successor type of the constructor *S* and occurs in the definition of function *add* and *ifthenelse*. The *type symbol Bool* occurs in the definition of function *ifthenelse*.

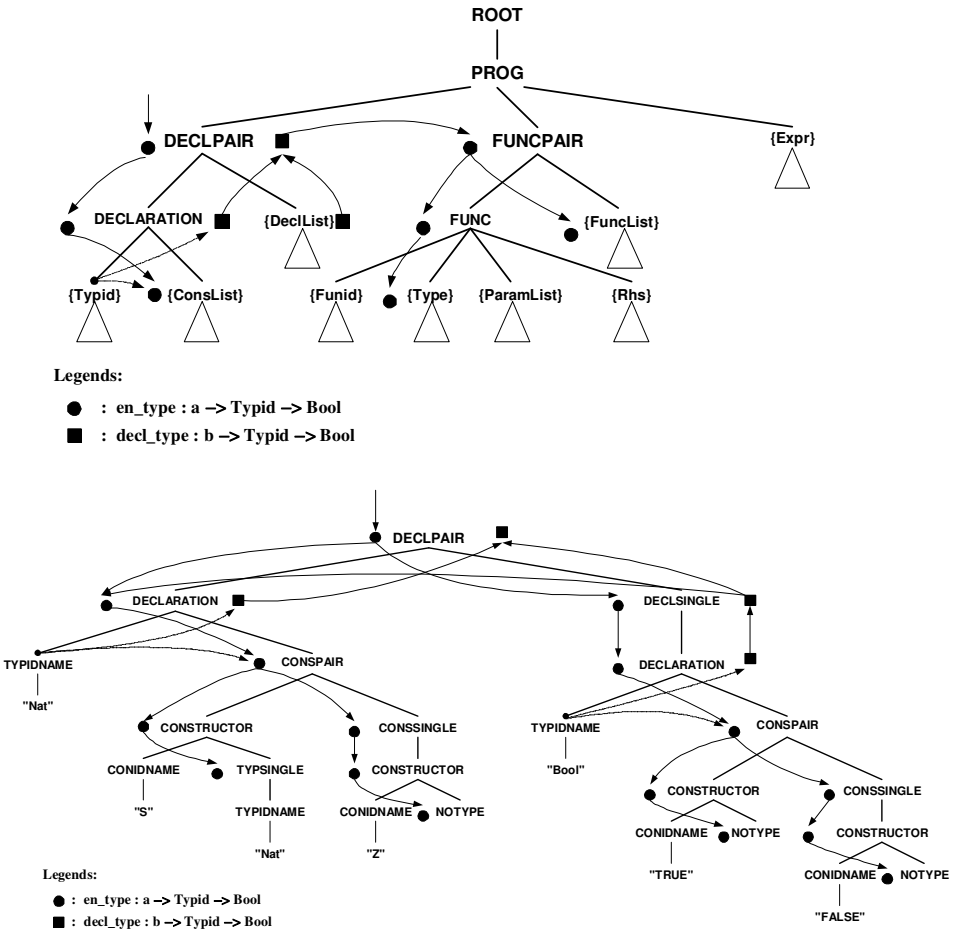


**Figure 6**. Attributed Syntax Trees for transfer of the type symbol information.

The needed information is only bool value stating whether a type symbol is defined. There defines a function *decl_type* and *en_type* that are inherited attribute and synthesized attribute, respectively. Both functions have a context parameter with type *Typid*.

Figure 6 is the attributed syntax tree for the checking of a type symbol. In the node *PROG* the information is sent from the *DataList* to the *FunctionList* that will be inherited to all subtrees in the FunctionList. The bottom picture in the Figure 6 is the attribution in the DataList. This attribution is not only synthesizing the type symbol in which it is defined, but also inheriting. This is needed to check whether a type symbol following a constructor is defined.

Certain symbols in the left and the right of node label denote the attributes of the node. The symbol in the left is an inherited attribute whereas one in the right is a synthesized attribute. The arrows denote how the attributes are obtained. Specially, the broken arrows denote that the attribute is obtained from a certain node or subtree.

2. Every *function symbol* is defined
   A *function symbol* will be occurred in the *input expression* or in the right-hand side of functions in the Haskell$^-$ program. A *function symbol* occurring in those places is called a function application. Besides the information of the existence of a function, the *type* of the function is also needed to check context sensitive restrictions in *type* of an expression.

   For both information the attributes consist of two values, the *type* of a function and its declaration in *Bool*. The value of the attribution is a pair of (*Type*, *Bool*), where *Type* is a type in the abstract syntax. If a function is not defined, then the value of the attribution is (*TYPESEMPTY*, False). Otherwise it is (T,*True*) where T is the *type* of the function. Figure 7 shows an attributed syntax tree to transport information of a particular function.

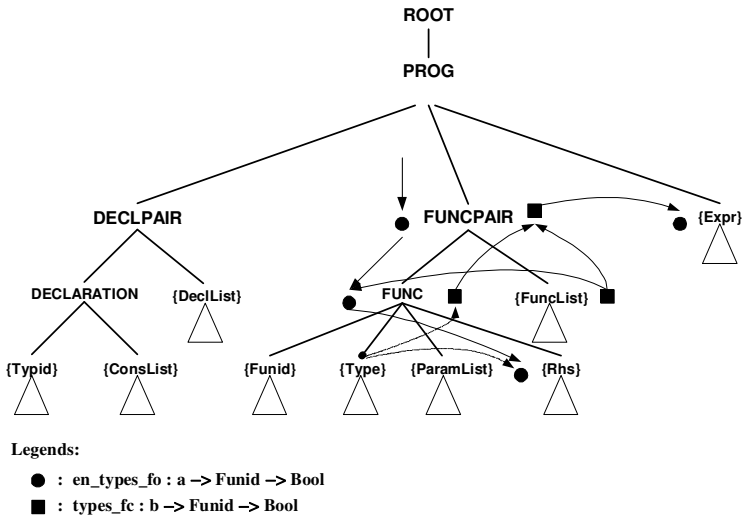


**Figure 7**. An Attributed Syntax Tree for transfer of the function information.

There are two attributes, i.e. *types_fc* and *en_types_fc* are as synthezised and inherited attribute, respetively. Both attributes have one context parameter whose type is *Funid*.

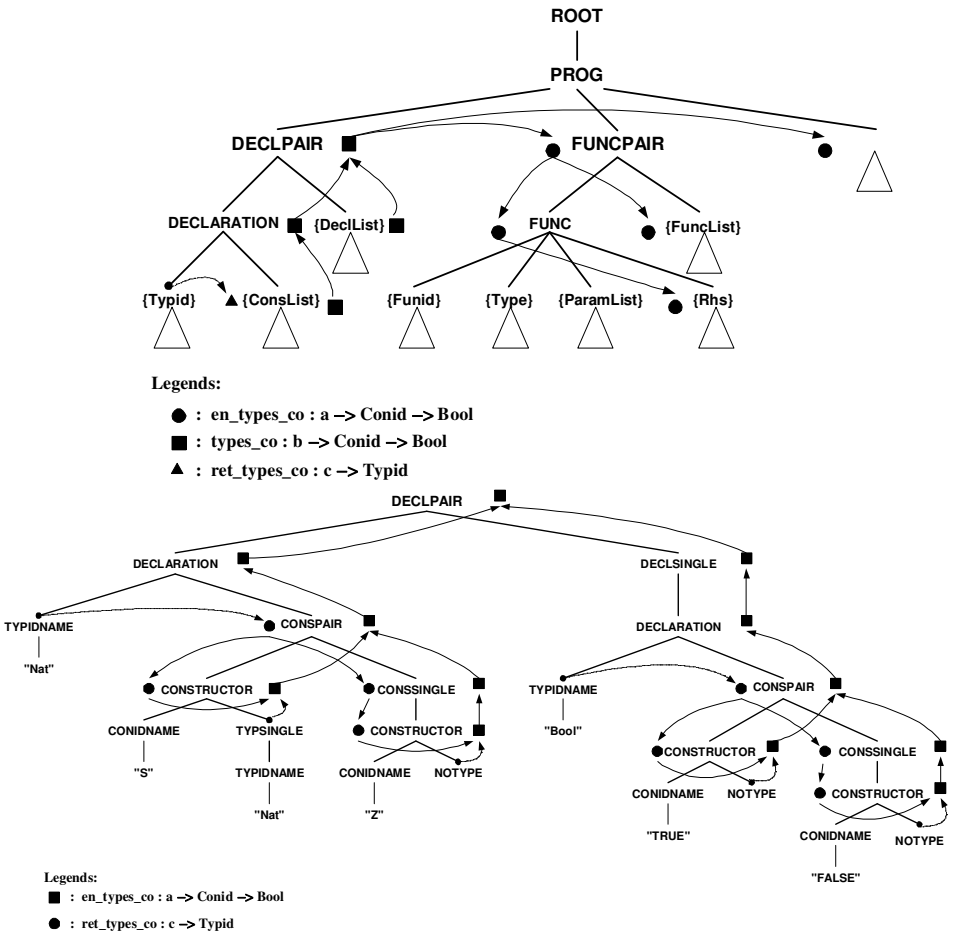


**Figure 8**. Attributed Syntax Trees for transfer of the constructor information.

3. Every *constructor* is defined
   Similar with functions, a constructor can occurs in the *input expression* and the right-hand sides of functions. Thus, we need information both the defined constructor and its *types*. Thus the value of the information is a pair (*Type*,*Bool*).

   Two attributes are defined to transfer the information, ie. *types_co* and *en_types_co* which are synthesized attribute and inherited attribute, respectively. Both attributes have one context argument whose type is *Conid*.

   The information transfer is simpler than the transfer of the information of a function. For a function, it is possible that the function is defined recursively such that the information of the function should be sent to its right-hand side. Furthermore, the information is also sent to other function that is possible to call it.

Meanwhile, the information of a constructor is only transferred from data definition to the function definition and the *input expression*. Figure 8 shows the transfer of constructor information.

4. Every variable is locally defined.
   Variables can be defined in many parts in a Haskell$^-$ program. A variable can be defined in parameters of a function, as argument of a pattern in case expression, or in assignments of a *let-expression*. In which scopes of a variable can occur are explained in the Chapter 2.

   Unlike functions and constructors, variables only occur in right-hand sides of functions. But, we still need the information of both its declaration and its type. Thus, the value of the variable information is also a pair (*Type*,*Bool*).

   The idea of the information transfer of a variable as follows:
   a. In the scope of a function definition, each variables (parameters) information should be sent to the right-hand side. The variable can occur in the right-hand side.
   b. If the right-hand side is a case expression, then the variables information is sent to each pattern in each case statement of the case expression. The patterns need this information to check whether the variables that are possibly defined in the pattern have already defined or not. The variable information is sent to each right-hand side after they are combined with variables that possibly defined in its pattern.
   c. If the right-hand side is an expression, then the variables information is sent to the expression.
   d. If the expression is a let-expression in the form:

      let $x_1$ = $e_1$
          ...
          $x_n$ = $e_n$
      in e

      then the variables information is sent to expressions $e_1$, …, $e_n$, and $e$. Specially for the expression $e$, this information is combined with variables from the assignments in let-expression.
   e. If the expression is not a let-expression, then the variables information is sent to each other expression that possibly occurs in the expression.

   Figure 9 shows how the variables information is transferred in the definition of function *add* in Example 1.

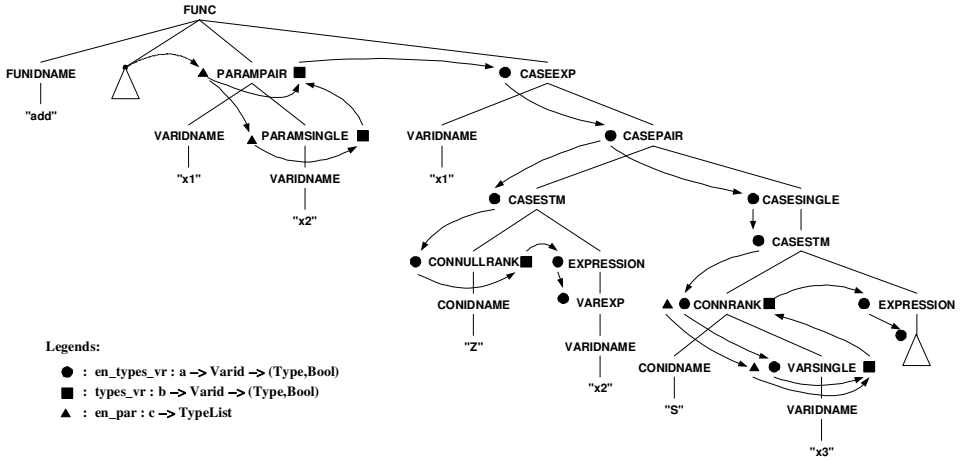


**Figure 9.** An Attributed Syntax Tree for transfer of the variable information.

5. Restriction in *type* of the right-hand side of a function
   In chapter 3.2, there is a restriction for the *type* of a right-hand side of a function. The *type* is determined by the *type* of the function and the number of its parameters. See Example 2.

   If the right-hand side of a function is a case expression then all right-hand sides should have the same *type* with the *type* of the right-hand side of a function. This statement will continue if there are more nested case expressions.

   If the right-hand side is an expression then the *type* of the expression should be same with the *type* of the right-hand side.
   Figure 10 shows how the information of the *type* of the right-hand side is generated and distributed until it reach an expression. Figure 10 is an abstract syntax tree for the definition of function *add* of Example 1.

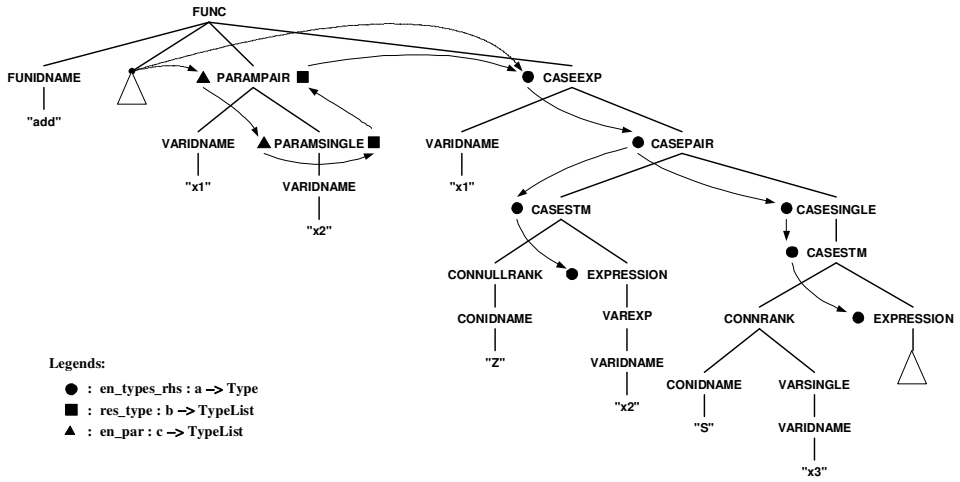


**Figure 10.** An Attributed Syntax Tree for transfer the *type* of the right-hand side.

6. Context-sensitive restrictions for expressions.
   In chapter 3.2, there are some context-sensitive restrictions that should be fulfilled by expressions. For these restrictions we need information the *type* of an expression. The chapter 3.2 also explains how to determine the *type* of an expression.

   The *type* of an expression is also needed in checking context-sensitive restriction of the input expression. As explained in chapter 3, the *type* of the input expression should be a *data type*.

   We define a synthesized attribute *types_exp* to generate the *type* of an expression. Figure 11 shows how the attribute *types_exp* is generated. Figure 11 is an abstract syntax tree for the input expression of Example 1.

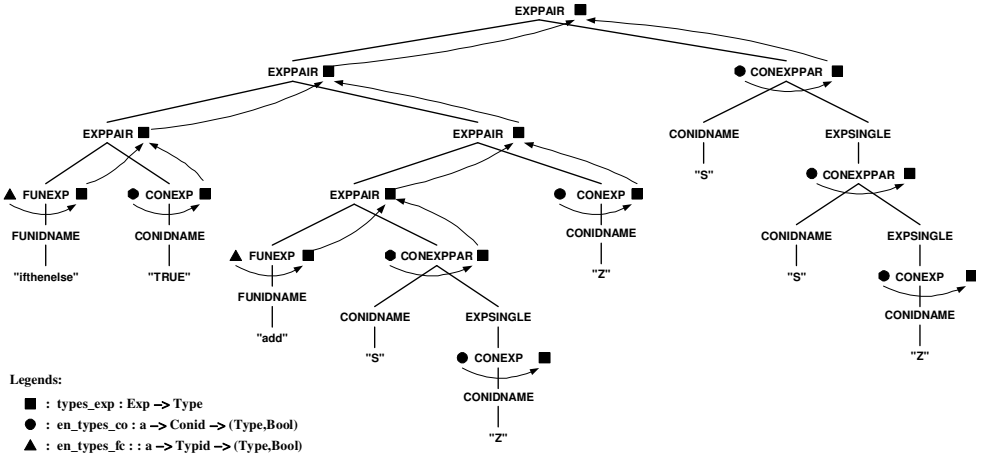


**Figure 11**. An attributed abstract syntax tree for generating the *type* of an expression.

7. Context-sensitive restrictions for case-expressions.
   Some context-sensitive restrictions for case-expression are explained in the context-sensitive restrictions for the variable and the right-hand side. Other context-sensitive restrictions are related with pattern matching. As explained in chapter 3.2, every constructor in patterns of one case-expression should have same data type and no constructor occurs more than one in those patterns.

   We define an inherited attributes to distribute the type that should be fulfilled by the constructors in patterns matching. We also need information that a constructor does not occur in other patterns such that there is no double pattern. The top picture of Figure 12 shows the type of the pattern transfer to every pattern. The bottom picture shows the transfer of the information that a constructor has already occur in a pattern.

8. Every type symbol, constructor, and function symbol are unique.
   The checking of the context-sensitive restriction occurs in the definitions of data types, constructors, and functions. We can use the transfer of type symbol, constructor, and function symbol information. The attributions of the information are explained in point 1, 2, and 3.

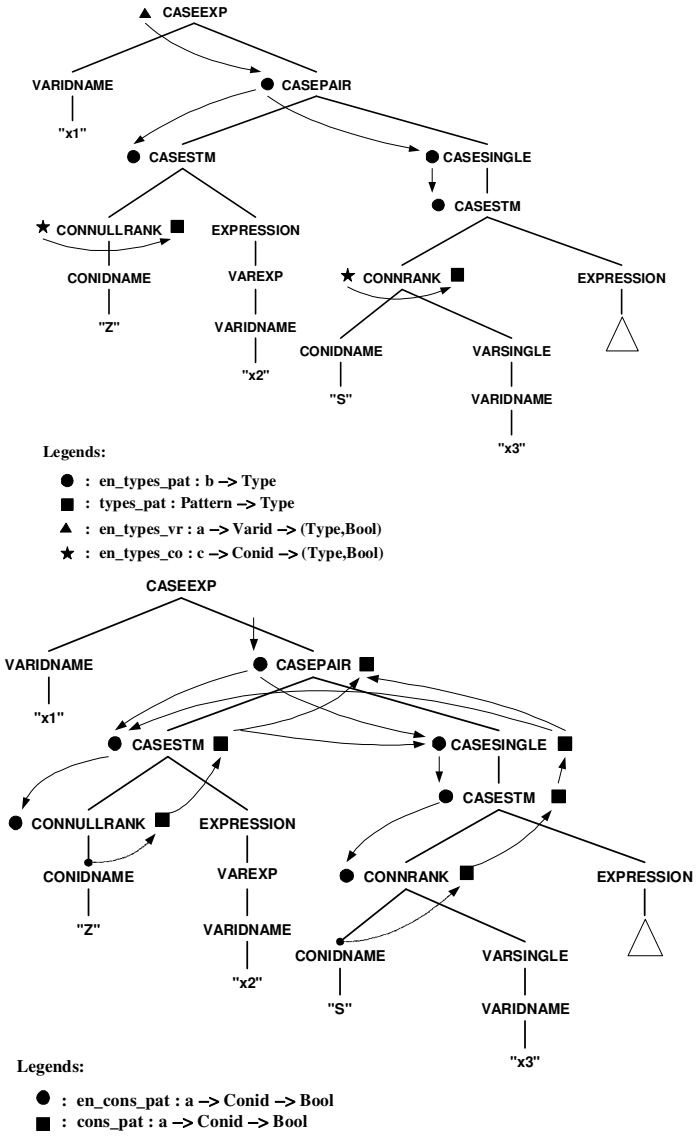


**Figure 12**. Attributed abstract syntax trees for transfer of *type* of pattern (top) and for transfer of the information of a constructor that occurs in pattern (bottom).

### *4.3.2. Abstract Machine Code*

Some attributes are needed in unparsing an abstract syntax tree into abstract machine code for implementation of call-by-value, call-by-name, or call-by-need. An attribution is used to check whether an abstract syntax tree representing a first-order Haskell$^-$ program. As above explanation, the implementation of call-by-value and call-by-name is only for a first-order Haskell$^-$ program. It is necessary to check the restrictions for the sub-class, especially in translation to both strategies.

Another attribute is related with the function information. For all strategies, two instructions of an expression as function application need information of the function. The instructions are *CALL* and *FNODE*. The instructions need the number of parameters, the maximal index of variables, and the first address of the function.

The last attribute is related with linearization of abstract machine code. The linearization gives an address to each instruction of abstract machine code and an appropriate address in some instructions, i.e. *JUMP*, *MATCHORJUMP*, and *SUSPEND*. Although in the translation of a Haskell$^-$ contains two steps, the translation into abstract machine code with symbolic address and the linearization, the unparsing for this purposes will run simultaneously with the linearization.

Here is detailed explanation of the main attributes:

1. Context sensitive restriction for a subclass first-order Haskell$^-$
   The checking of this restriction occurs in all expressions and all definitions of functions. In an expression, it is to check whether the expression contains a *function application* that is not partial application. In the definition of a function, it is to check the *type* definition of the function, the number of parameters, and the right-hand side.

   According to the unparsing for abstract machine code, the checking of the context sensitive restriction occurs before the unparsing. In other word, the checking occurs in the node with data type *Program*.

   First, we need information that an expression or a function is defined in first-order. We define the synthesized attribute *firstorder*. Specially for an expression that is a function application, we need information that the function is not applied partially. We define also the synthesized attribute *fo_inpartial*. The attribute has one context argument with type integer. The argument denotes how many times application expression should be applied to a function such that the function is not applied partially.

   Figure 13 is the attribution in a subtree of the input expression in Example 1.

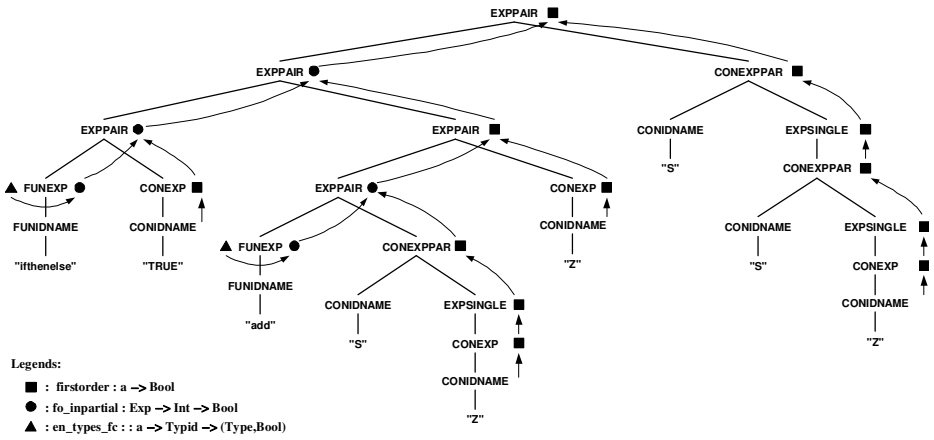


**Figure 13**. An attributed syntax tree.

2. Function information
   This information can be generated in the definition of a function with a synthesized attribute *tab_fc*. The information consists of three values, an address of the first instruction of the abstract machine code for the function, a number of parameters, and a number of maximal indices of variables.

   Here is a rule of the function:

```
t@ (FUNC t1 t2 t3 t4) in
 tab_fc  t  y  = gettab_fc t1 y
                    ((addr t),
                     (narg_fc t3),
                     (max2
                         (maxvar t3)
                         (maxvar t4)))
```

   The function *tab_fc* has one context arguments that refers a function. The function *gettab_fc* is an auxiliary function. (gettab_fc t1 y v) means 'give v if t1=y, otherwise (0,0,0)'. The attribute *addr* is an inherited attribute that gives an address of the first instruction of the context tree. The function *narg_fc* is a synthesized attribute that generates the number of arguments, whereas the function *maxvar* is another synthesize attribute that generates the maximal indices of variables. The function *max2* is an auxiliary function to determine the maximal value of two values.

3. Linearization an Abstract Machine Code.
   This process is to give addresses for instructions in abstract machine code. The main idea of the process is an address given to a subtree of the abstract syntax tree where the subtree will be unparsed. The address is an address for the first instruction translated in the subtree. Depending on the number of instructions translated in the subtree we should identify the next address of the last instruction. This address will be used as first address for another subtree in same level or will be the next address of the parent node.

To make this clear, Figure 14 shows how to distribute the first address (attribute *addrv*) and the next address (attribute *nextaddrv*) for a subtree representing the *input expression* of Haskell⁻ program in Example 1. This figure is only attributes in unparsing to abstract machine code for call-by-value implementation. We will not discuss other implementation since the idea is similar. The only difference is the value of the next address. It is caused by the different number of instructions for the same subtree.

Figure 14 is the attribution in a subtree of the input expression in Example 1. From the root, the first address is sent to the second child since the first child will be unparsed former than the second child. It is according to the translation function for an *application expression* ($e_0$ $e_1$). The translation will translate $e_1$ then translate $e_0$. The first address for first child is obtained from information of next address in second child.

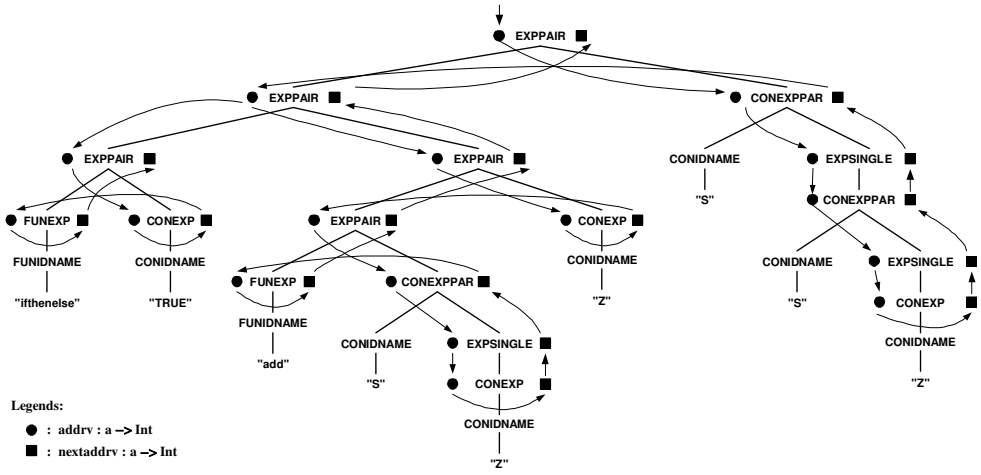


**Figure 14**. An Attributed Syntax Tree.

## 4.4. Logic Description

In this part, we can specify formulas of MSO* logic to test some properties of the abstract syntax tree. In this logic, we can define formulas that consist of attributes. The formulas are defined in MSO Block.

Since the performance of this facility is too slow to complex formulas, this specification for Haskell⁻ only uses some simple formulas for simple problems. Some formulas are to check some context-sensitive restrictions and other formulas are to check the implementation for abstract machine code.

The context-sensitive restrictions that will be checked by an MSO* logic formulas are related with the *input expression* in a Haskell⁻ program. The *input expression* should not contain let-expression.

For the restriction, we need a formula that will check whether an expression contains a let-expression. In abstract syntax tree, a let-expression is known with label *LETEXP*. So, if there is a node with label *LETEXP* under the node representing the checked expression then the formula is true. We define a formula *?exist_let* as follows

```
formula ?exist_let :: Exp -> Bool
?exist_let x
   = Or (Label LETEXP x)
        (ExNode y :: Exp
           (And (Label LETEXP y)
                (Under y x)))
```

Other formulas in this project are related to unparse to an abstract machine code. As we know, there are three implementations to the abstract machine code, i.e. call-by-value, call-by-name, and call-by-need. For implementation of call-by-name and call-by-need, they use the same unparsing because the difference is small. And, for implementation of call-by-value, it uses another unparsing. However in unparsing they use a common attribution. The attribution is about the information of function that consists of address of first instruction, number of parameters, and maximal indices of variables.

The number of parameters and maximal indices of variables are not different between three abstract machine codes. But the addresses of first instruction are different. Thus, in the generating of the information the attribution should check which unparsing will use this information. Let see the attribution function *tab_fc*. The rule of the attribution function should be the following:

```
t@ (FUNC t1 t2 t3 t4) in
 tab_fc  t  y  = gettab_fc t1 y
                      ((if_then_else
                          (?call_by_value t1)
                          (addrv t)
                          (addrn t)),
                       (narg_fc t3),
                       (max2
                           (maxvar t3)
                           (maxvar t4)))
```

The formula *?call_by_value* is to check whether the currect unparsing is call-by-value. If the unparsing is call-by-value then the address of first instruction is *(addrv t)*. Otherwise it is *(addrn t)*. The inherited attribute *addrv* and *addrn* are to linearize abstract machine code of call-by-value and call-by-name, respectively.

The definition of the formula *?call_by_value* is following :

```
formula ?call_by_value :: Funid -> Bool
?call_by_value x
   = ExNode y :: Program (Label IMPLCALLVALUE y)
```

The other formulas are to check whether the unparsing is to abstract machine code for call-by-name or the unparsing is to abstract machine code for call-by-need. The

formulas are needed because the two implementations use a same unparsing and same attributions. The formulas are following:

```
formula ?call_by_name :: Funid -> Bool
?call_by_name x
   = ExNode y :: Program (Label IMPLCALLNAME y)

formula ?call_by_need :: Exp -> Bool
?call_by_need x
   = ExNode y :: Program (Label IMPLCALLNEED y)
```

## 4.5. Structural Transformation

An abstract syntax tree is an edited object in the specification of languages-based editor. There are two methods to edit the abstract syntax tree. One method is editing by textual input. With textual input, some subtrees representing some part of the edited language can be replaced by a parsed text. The textual input is handled in parsing part. Another method is structural transformation.

In the structural transformation method, a subtree can be replaced by another subtree. It is done by computation the abstract syntax tree. We should specify transform functions to compute this transformation. The transform functions are specified in a Mag Block.

A transform function has two arguments. First argument is a pattern matching a subtree that will be transformed. The argument is called *tree argument*. The second argument of the transform function is a context parameter. The context parameter denotes the name of the transformation. The context parameter has a type *TransSelection*.

In this chapter, we will see the specification of structural transformation for language-based editor for Haskell⁻. Because some transformations have same idea, we will not discuss all transformations. For some transformations having same idea, we will choose a transformation as an example.

Some transformations in this specification are for subtrees representing list. We see in the abstract syntax that there are some lists, i.e. list of data types, list of functions, list of constructors, list of successor types of a constructor, list of types in function definition, etc.

In transformation of a list, we take idea on list operations. There are two operations in list. First operation is an insertion an element into a list. The insertion can put the element before or after a particular element. Another operation is deletion. With the deletion, we can delete a first element or the tail of the list. These operations are applied to this transformation.

For a subtree representing a list, the transformations are *insert before*, *insert after*, *cut first element*, and *cut rest* of list that refer to insertion and deletion in list. As examples are transformations for a list of constructors. Figure 15 shows the transformations. The names of the transformations are *Ins_const_after*, *Ins_const_before*, *Cut_const_first*, and *Cut_const_rest*. The top figure shows transformations for the pattern

*CONSSINGLE*, whereas the bottom one shows transformations for the pattern *CONSPAIR*.

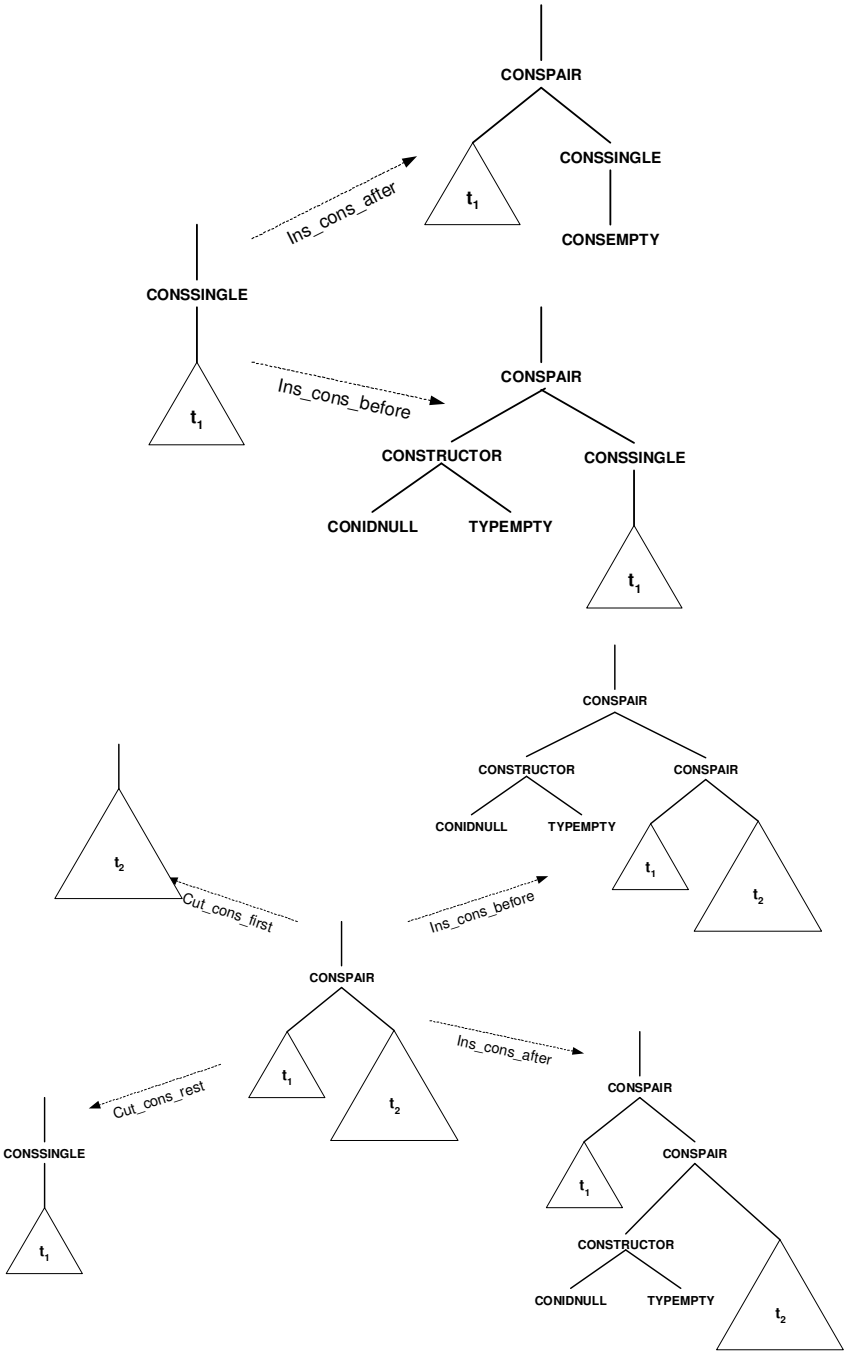


**Figure 15**. Structural transformations for *ConsList* representing list of *constructors*.

Another transformation that is mostly applied is instantiation. This transformation instantiates a placeholder. As examples are transformations a placeholder for an expression. The transformation shall instantiate a placeholder *EXPEMPTY* with one of expression, i.e. a variable expression, a *function application*, a *constructor application*, an *application expression*, or a *let-expression*. Figure 16 shows the transformations.

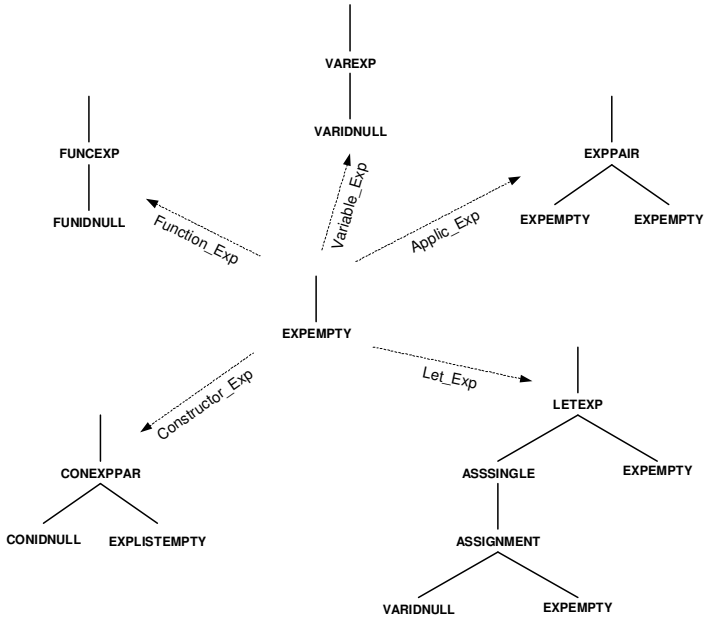


**Figure 16**. Instantiation transformations for a placeholder *EXPEMPTY*.

Some interesting special transformations are specified to an expression. This transformation make a user of this editor is easily in editing of an expression. These are applied transformations to an expression *e*, where *<Exp>*, *<Var>*, and *<Con>* denotes unspecified expression, variable, and constructor, respectively:

| | |
|---|---|
| Cut_exp | : transforms *e* into *<Exp>* |
| As_arg_applic | : transforms *e* into '*(<Exp> e)*' |
| Cut_applic_exp | : transforms *e* = ($e_0$ $e_1$) into $e_1$ |
| As_applic_exp | : transforms *e* into '*(e <Exp>)*' |
| Cut_arg_applic | : transforms *e* = ($e_0$ $e_1$) into $e_0$ |
| As_let_exp | : transforms *e* into *'let <Var> = <Exp> in e'* |
| Cut_assign_let | : transforms *e* = '*let ….. in* $e_0$' into $e_0$. |
| As_arg_const | : transforms *e* into *'(<Con> e)'* |

As examples are transformations to an expression with pattern *EXPPAIR* representing an *application expression*. The transformations which can be applied to the expression are *Cut_exp*, *As_arg_applic*, *Cut_applic_exp*, *As_arg_applic*, *Cut_arg_applic*, and *As_arg_const*. Figure 17 shows the transformations.

Other transformations are changing the label of root of subtree. Examples of the transformations are transformations for a subtree with type *Program*. The possible labels for the root of the subtree are *PROG*, *PROGNOCSR*, *IMPLCALLVALUE,*

*IMPLCALLNAME*, and *IMPLCALLNEED*. The meaning of the labels is explained in chapter 4.1. We define transformations *Source_Code*, *Source_noCSR*, *Impl_by_Value*, *Impl_by_Name*, and *Impl_by_Need* to change the label with *PROG*, *PROGNOCSR*, *IMPLCALLVALUE, IMPLCALLNAME*, and *IMPLCALLNEED*, respectively.

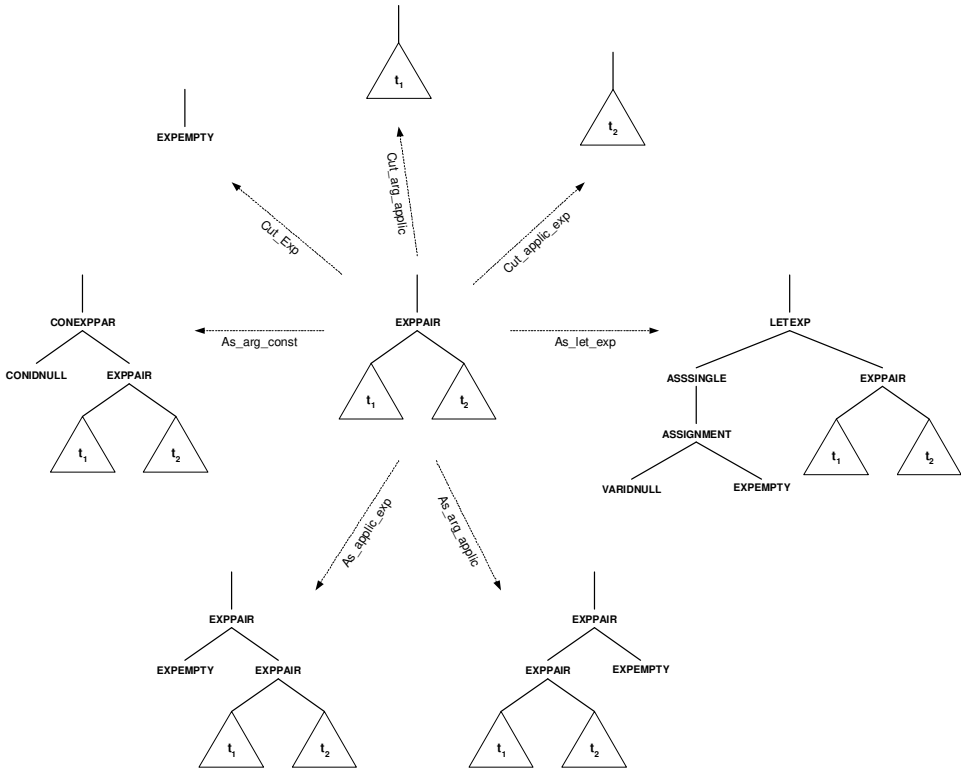


**Figure 17**. Transformations for an expression with pattern *EXPPAIR* representing an appli*cation expression*.

## 4.6. Parsing

In this part, we can specify textual inputs in the languages-based editor. The textual inputs can be some part of the language. To do this, we need the parsing. The parsing will transform a textual input into a part of the abstract syntax tree of the edited object in the language-based editor.

This parsing should be defined in three following blocks:

1. a *lex* block defining a scanner by a list of tokens specified by regular expressions.
2. a *parse* block defining a parser by a context-free grammar using the defined token and constructing the syntax tree of the textual input.
3. an *input* block determining the part of the language that is editable and which parser will be applied.

In this language-based editor for Haskell⁻, some parts of the language can be edited. The parts and their examples are follows:

1. Identifier: *type symbol*, *function symbol*, *constructor*, and *variable*.
   Example: Nat, add, TRUE, x1
2. Single data definition.
   Example: data Nat = S Nat | Z deriving Show
   data List = N Nat List | E deriving Show
3. List of constructors
   Example: S Nat | Z
4. Definition of one constructor
   Example: N Nat List
5. List of types as successor types of a constructor
   Example: Nat List
6. List of expressions following a constructor expression
   A constructor has possibly some arguments. If the constructor is applied in an expression then the constructor is followed by list of expression.
   Example: Z E {may follow the constructor *N* in example of point 2}
7. Expression
   The expression could be a variable, function application, constructor application, partial application (a pair of expressions), a let-expression.
8. List of assigns in let-expression
9. Function
   The textual input can be a complete definition of a function. This definition includes the function's name, its type, arguments, and right-hand side. But, we face a problem. A textual input is only in one line. Thus, it is hard to give a complete definition function only in one line.
10. Type of a function
    Example: ((Nat ->Nat ->Nat) -> Nat -> Nat)
11. Argument types of a function type
    Example from type in point 10: (Nat ->Nat ->Nat) -> Nat
12. List of parameters
    Example : x2 x3 x5
13. Single case statement
    Example : (S x3) -> (S ((add x3) x1))
14. Pattern in a case expression
    Example: (N x3 x4)
15. List of variables in pattern
    A list of variables is similar with a list of parameters.
16. Right-hide side
    This right-hand side can be either a case expression or an expression.

For the textual input, some following tokens are needed:

1. string to represent identifiers
2. string of non-terminal symbol in context-free syntax, i.e. 'data', 'deriving', 'show', 'let', 'in', 'case', 'of'.
3. sequence of special characters, i.e. '(', ')', '=', '::', '->', '|'.

```
beginlex Text_Input
  token Whitespace         = [\ \t\n] | '{' [~}]* '}'
  token OPthsTk            = "("
  token CPthsTk            = ")"
  token LetTk              = "let"
  token InTk               = "in"
  token EqualTk            = "="
  token ArrowTk            = "->"
  token CaseTk             = "case"
  token OfTk               = "of"
  token DataTk             = "data"
  token BarTk              = "|"
  token DerivTk            = "deriving"
  token ShowTk             = "Show"
  token DoubleTk           = "::"
  token VaridTk  String    = "x" [0-9]+
  token TypidTk  String    = [A-Z] [a-z]+
  token ConidTk  String    = [A-Z]+
  token FunidTk  String    = [a-z]+
endlex
```

**Figure 18.** Definition of tokens in Lex Block.

```
  parse T_Exp :: Exp
    | T_Varid              = VAREXP $1
    | OPthsTk
      T_Conid
      T_ExpList
      CPthsTk              = CONEXPPAR $2 $3
    | T_Conid              = CONEXP $1
    | OPthsTk
      T_Exp
      T_Exp
      CPthsTk              = EXPPAIR $2 $3
    | T_Funid              = FUNCEXP $1
    | OPthsTk
      LetTk
      T_AssignList
      InTk
      T_Exp
      CPthsTk              = LETEXP $3 $5

  parse T_ExpList :: ExpList
    | T_Exp
      T_ExpList            = EXPLISTPAIR $1 $2
    | T_Exp                = EXPSINGLE $1
  parse T_AssignList :: AssignList
    | T_Assignment
      T_AssignList         = ASSPAIR $1 $2
    | T_Assignment         = ASSSINGLE $1
  parse T_Assignment :: Assignment
    | T_Varid
      EqualTk
      T_Exp                = ASSIGNMENT $1 $3
```

**Figure 19.** The definition of a parser for expression in a Parse Block.

Figure 18 shows the definition of tokens in the lex Block. The position of token is important. If the position of token is not correct then it is possible that the scanner can not reach the token. Suppose that token *DataTk* placed after token *FunidTK*. The scanner can not reach the token DataTk because the scanner will read every string 'data' as FunidTk.

The second Block, the parse block, consists of definition of parses. Figure 19 shows some parses in the parse block. One of the parses is for an expression whose name is T_Exp. According to the context-free syntax of Haskell$^-$ in Figure 1 the textual input of an expression is as a variable, a function application, a constructor application, a partial application, or a let-expression.

The last block is to determine which part of the abstract syntax tree can be replaced by textual input and which parse will be used. Figure 20 shows the determination of the part as textual input.

```
begininput Text_Input
  input Varid        as T_Varid
  input Typid        as T_Typid
  input Conid        as T_Conid
  input Funid        as T_Funid

  input DataDef      as T_DataDef
  input ConsList     as T_ConsList
  input ConsDecl     as T_ConsDecl
  input TypList      as T_TypList
  input ExpList      as T_ExpList
  input Exp          as T_Exp
  input AssignList   as T_AssignList
  input Assignment   as T_Assignment

  input Function     as T_Function
  input Type         as T_Type
  input TypeList     as T_TypeList
  input ParamList    as T_ParamList

  input CaseStm      as T_CaseStm
  input Pattern      as T_Pattern
  input VarList      as T_VarList
  input Rhs          as T_Rhs
endinput
```

**Figure 20.** Input determination in input Block.

## 4.7. Auxiliary Functions

In this block we can define some auxiliary functions that are usable in mag blocks. The auxiliary functions can be defined as normal *Haskell functions*.

In this specification some auxiliary functions are called in the unparsing, the attribution, and the logic description parts. In this chapter we will discuss two defined auxiliary functions in this specification that are often called.

One of the functions is *check*. The definition of the function:

```
check :: String -> Bool -> String
```

The function *check* will give the first argument if the second argument is true. Otherwise it gives empty string. The function is useful to generate an error message for a context-sensitive restriction. The error message and the context-sensitive restriction are the first argument and the second one, respectively, for the function *check*.

In attribution, some attributes should combines between two values. In this combine the attribute will choose one of both values. For example, the synthesized attribute *en_types_co* in the node with label *CONSPAIR* in Figure 8. The value of the attribute should be one of the two attributes of children. To choose the values the attributes calls the function *comb_pairtypes*. The definition of the function is follows:

```
comb_pairtypes :: (Type,Bool) -> (Type,Bool) -> (Type,Bool)
```

## 5. How To Use The Editor

The result of this project is the language-based editor for Haskell⁻. We will discuss how to use the editor.

The specification of the editor is saved in file whose name is *haskelmin.dsl*. The file consists of the specification in DSL that are explained in Chapter 4.

First step is generating the editor from the specification file. We need a generator system designed to generate the editor from a specification file in DSL and run the generator system. The generator system will ask for the specification file as input. The result of the generating process is an executable file *Start* and its supporting files. We can start the editor from the file *Start*.

Figure 21 shows a window of the editor. We discuss further the parts of the window.

*Menus* consist of the following items:

1. New : to create a new file consisting of an initial abstract syntax tree.
2. Open : to open a file consisting of an abstract syntax tree
3. Save : to save the current abstract syntax tree into a file
4. Exit : to exit from the editor.

In the editing area we can see the edited Haskell⁻ program. The current-edited part of the program is shown by red color. We can edit the current-edited part by an applicable transformation or textual input, if any. The applicable transformation is showed in *Transformation Buttons*. Each *Transformation Button* contains the name of the transformation and a keyboard key. We can choose one transformation by mouse device or the keyboard key. For textual input, we can apply if it is defined as textual input in

*parsing* part of the specification. If it is possible, the input can be given by keyboard keys. To delete the whole current editing part we can use the *insert* key.

According to the algebraic data type defined in the specification of the editor, the type of the current-edited part is displayed in *Current-type Caption*.

We can change the current-edited part by *Change Buttons*. The *Change Buttons* consists of two buttons, i.e. *Rewind* and *Forward*. Regarding to the abstract syntax tree representing the Haskell⁻ program, the *Rewind* will change the current-edited subtree (represents the current-edited part) to the previous subtree in the same level. If there is no previous subtree then the *Rewind* will change to the parent. The *Forward* button works reversely.

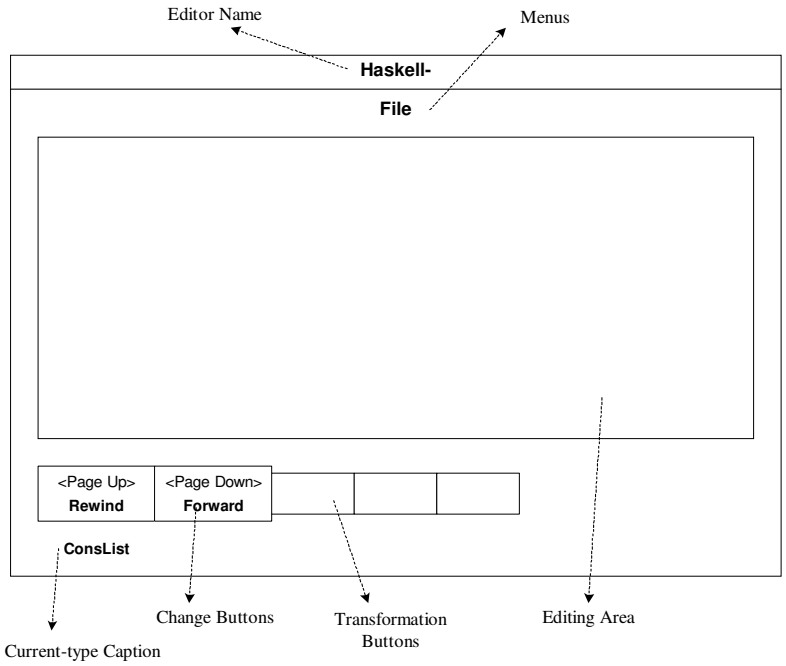


**Figure 21**. A window of running editor.

In the editing area, we can change the appearance with an abstract machine code of the program. To change the appearance of the codes, we should change the current-edited subtree to a subtree with data type *Program*. We can choose one transformation in the *Transformation Button*. The names of the transformations as follows:

1. Source_Code : to show the Haskell⁻ code with error messages
2. Source_noCSR : to show the Haskell⁻ code without error messages
3. Impl_by_Value : to show the abstract machine code for call-by-value implementation
4. Impl_by_Name : to show the abstract machine code for call-by-name implementation

5. Impl_by_Need : to show the abstract machine code for call-by-need implementation

Some defined transformations are to edit an expression. These transformations make a user of this editor is easily in editing of an expression. These transformations are applied to an expression *e*, where <*Exp*>, <*Var*>, and <*Con*> denotes unspecified expression, variable, and constructor, respectively:

| | |
|---|---|
| Cut_exp | : to transforms *e* into <*Exp*> |
| As_arg_applic | : to transform *e* into '*(*<*Exp*> *e)*' |
| Cut_applic_exp | : to transform $e = (e_0\ e_1)$ into $e_1$ |
| As_applic_exp | : to transform *e* into '*(e* <*Exp*>*)*' |
| Cut_arg_applic | : to transform $e = (e_0\ e_1)$ into $e_0$ |
| As_let_exp | : to transform *e* into *'let* <*Var*> = <*Exp*> *in e'* |
| Cut_assign_let | : to transform *e* = '*let ….. in* $e_0$' into $e_0$. |
| As_arg_const | : to transform *e* into *'(*<*Con*> *e)'* |

The user can give textual input for some parts of the Haskell⁻ program. Regarding to the algebraic data types in the specification of this editor, the followings are data types shown by the *Current-type Caption* and their related parts of the Haskell⁻ program, in which the user can give the textual input:

1. *Typid* : a *type symbol*
2. *Funid* : a *function symbol*
3. *Conid* : a *constructor*
4. *Varid* : a *variable*.
5. *DataDef* : a single definition of data type
6. *ConsList* : a list of constructors
7. *ConsDecl* : a definition of one constructor
8. *TypList* : successor types of a constructor
9. *ExpList* : a list of expressions following a constructor expression
10. *Exp* : an expression
11. *AssignList* : a list of assigns in let-expression
12. *Function* : a definition of a function
13. *Type* : a type of a function
14. *TypeList* : argument types of a function type
15. *VarList* : a list of variables or parameters
16. *CaseStm* : a single case analysis
17. *Pattern* : a pattern in a case analysis
18. *Rhs* : a right-hide side

The textual input should fulfil the context-sensitive syntax of Haskell⁻ for appropriate edited part.

## 6. Conclusion

This project builds a specification of language-based editor for Haskell⁻ using generating system. The specification language used in the specification is DSL.

The language-based editor is able to edit a Haskell$^-$ program that is structurally correct according to the context-free syntax. The editor is also able to give appropriate error messages according to all context-sensitive restrictions. However, the editor is able to show the edited Haskell$^-$ program without error messages.

The abstract machine codes of call-by-value, call-by-name, and call-by-need implementation are able to be displayed by this editor. Since the abstract machine codes of call-by-value and call-by-name implementations are only for first-order Haskell$^-$ program, the editor will give a warning if the program is not in first-order.

The editor allows a user to give textual inputs. The textual inputs are for some parts of the Haskell$^-$ program.

A problem faced in this specification is in unparsing. An attribute is possibly called more than one time in one rule of synthesized attributes of unparsing. Every time the attribute is called, the value transfer of the attribute happens. It causes the editor slower. It is better if there is a facility such that the value transfer happens only once.

## References


[Bor00] E. Bormann. *Entwurf und Implementierung eines Systems zur Erzeugungsytaxtgesteuerter Editoren.* Master's Thesis, Dresden University of Technology, 2000.

[Bor03] E. Bormann. *Automatic Generation of Language-Based Editors using DSL.* Manuscript, 2003.

[FHP99] Josep Fasel, Paul Hudak, and John Peterson. *A Gentle Introduction to Haskell 98*. Available on: http://www.haskel.org/tutorial, 1999.

[FV98] Z. Fülöp and Heike Vogler. *Syntax-directed Semantics – Formal Models Based on Tree Transducers*. Monographs in Theoretical Computer Science, An EATCS Series. Springer-Verlag, 1998.

[Küh00] Armin Kühnemann. *Implementation of Functional Programming Language*. Lecture Script, 2000.

[KV94] A. Kühnemann and H. Vogler. *Synthesized and Inherited Functions – a new Computational Model for Syntax-Directed Semantics*. Acta Inform., 31:431-477, 1994.

[Les99] C. Lescher. *Entwurf und Implementierung einer Eingabesprache für ein System zur Erzeugung Sytaxgesteuerter*. Editoren, 1999. Minor Thesis, Dresden University of Technology.