A Compiler for the **Paladim** Language

Klaus Møllnitz
nhg665

Konstantin Slavin-Borovskij
wfb603

Zeerak Waseem
csp265

December 20, 2013

1 Table of contents

2 Introduction

In this report we will document the changes we have made to the Paladim language.

We were given the assignment to implement certain tasks. For this report, we were asked to implement a parser, integer multiplication, division, and boolean operators, type inference and type checking for the `new` and the `read` functions, type checking and array indexing, and call-by-value-result semantics for procedures in the Paladim compiler.

We have written our testing section for each sub section in the analysis, so as to add a greater coherence between the implemented tasks and the test of these. All edited code can be found in the provided code.

3 Background

We will briefly introduce the nature of the changes for each of the tasks given. The tasks given were:

- Implement a parser for the Paldim language
- Implement integer division and multiplication and boolean operators in the Paladim compiler
- Type inference and type checking for the `new` function
- Type checking and code generation for Array Indexing
- Call-By-Value-Result semantics for procedures

The parser was implemented by creating a new grammar file, which can be seen in the `SRC` folder.

For the integer multiplication, division and boolean operations; we initially edited the `TpAbsyn.sml` file to reflect the operations we wished to implement (`times, divide, or, not`), we then edited Compiler.sml, where we implemented the translation to `MIPS`, we then edited `TpInterpret.sml` where we implemented the interpretation, finally we edited `Type.sml` where we ensure that the types correspond to the desired values.

For the `new & read` functions, we edited `Type.sml` to implement simple type inference and type checking. This is done in `typeCheckExp`.

To implement type checking and array indexing for arrays, we edited `Type.sml` to reflect required changes for array handling. Further, we edited the `compileLVal` function in `Compiler.sml` to support translation to machine code.

To implement Call-By-Value-Result semantics we have edited `TpInterpret.sml`, implementing `updateOuterVtable`. Furthermore we have also extended `Compiler.sml` to support code generation for Call-By-Value-Result interpretation, by implementing a new function named `rePutArgs`.

Diff files have been provided in `DIFF` folder of all files edited.

4 Analysis

4.1 Parser

4.1.1 Identifying tokens

In our abstract parser definition most of our tokens use the type $< int{*}int >$. This type is the position of the token, which is constructed by line and column. If nothing is stated in the tables, this is the type used.

We have identified the list of terminals and given each a keyword, for later use in the abstract parser definition.

Terminal	Keyword
program	PROGRAM
function	FUNCTION
procedure	PROCEDURE
var	VAR
begin	BEGIN
end	END
if	IF
then	THEN
else	ELSE
while	WHILE
do	DO
return	RETURN

Table 1: Keywords

All the keywords of the Paladim language is listed in table 1, these only use the positional data, meaning they all have token type $< int * int >$, since we have chosen to denote the position by linenumber and column in the line.

We also need keywords for the specific types, since we should be able to write the following as noted in (1). The keyword is written in table 2.

$$\text{VariableName} \; \mathbf{:} \; \text{Type.} \tag{1}$$

Terminal	Keyword
int	INT
char	CHAR
bool	BOOL
array	ARRAY
of	OF

Table 2: Type keywords

Next terminals we need to consider is symbols. These are very simple and shown in table 3.

Terminal	Keyword
;	SEMICOLON
:	COLON
,	COMMA
:=	ASSIGN
EOF	EOF

Table 3: Symbols

All arithmetic operators is found in table 4.

Parantheses is covered in table 5. They have been given their own table as to represent the structure of the lexer.

Finally we have all of the literals, which are supposed to contain both data and a position. As seen in table 6, they all make use of some sort of variable to store their content. We also cover the Identifier here.

Terminal	Keyword
$+$	PLUS
-	MINUS
*	TIMES
$/$	DIVIDE
$=$	EQUAL
$<$	LESS
AND	AND
OR	OR
NOT	NOT

Table 4: Arithmetics

Terminal	Keyword
(	LPAR
)	RPAR
{	LCBR
}	RCBR
[	LBRA
]	RBRA

Table 5: Parentheses

Terminal	Keyword	Type
ID	ID	<string*(int*int)>
NUMLIT	NUMLIT	<int*(int*int)>
LOGICLIT	LOGLIT	<bool*(int*int)>
CHARLIT	CHALIT	<char*(int*int)>
STRINGLIT	STRLIT	<string*(int*int)>

Table 6: Literals

4.1.2 Precedence

Next task is to denote the precedence of the operators as in table 7. The operators are listed by importance, top to bottom. Each operator is given a precedence score ranging from $1-8$. Finally each operator will have its associativity denoted. According to the assignment, all of the binary operators should be left-associative.

4.1.3 Productions

Last task for making the compiler is to write down each of the productions. We ran into a few problems with the handed out language, with both the operators between expression $Exp \rightarrow Exp$ OP Exp but also $Exp \rightarrow LVal$. These issues got solved, by substituting the productions into the production for Exp.

4.1.4 Linking parser

The last step to create a new parser, is to link the compiler to use the new implementation of the SLR(1) parser, instead of the handed out LL(1) parser. This was done in the file *"Driver.sml"*.

Name	Precedence	Associativity
TIMES	1	Left
DIVIDE	1	Left
PLUS	2	Left
MINUS	2	Left
EQUAL	3	Left
LESS	3	Left
NOT	4	Left
AND	5	Left
OR	6	Left
ELSE	7	Non Associative
OptElse	8	Non Associative

Table 7: Operator Precedence

4.1.5 Testing Parser

We have tested our parser implementation with a new test file we have created for the purpose, called *"parserTest.pal"*. This file is testing pretty much all of the different productions used in the language. We got all of them to work like they should. To test the parser while the type checker was not implemented, we simply noted if the Paladim program would fail with `parser error` or `type checking error`, if the last was the case, it meant the parser did the job correctly since its a typechecker error, hence the compiler proceeded after parse checking.

4.2 Integer multiplication/division and boolean operators

The multiplication, division and boolean operators is all fully supported by the parser, next task is to implement the features in the type checker.

To implement the features in the Paladim compiler, we initially edited the typed abstract syntax tree, `TpAbsyn.sml`, by adding `Times & Div` types to the `typeOfExp` function. This function should simply just return the type of first operand, since the type check makes sure both operands are of same type. Another important thing to add in this file is the pretty printing of the expressions, since the Interpreter is using the pretty printing functions to show the code to the user. It is done in a similar matter to the pretty printing of `Plus` and `Minus`.

We then edited the type checker, `Type.sml`, extending the function `typeCheckExp`. The two functions is implemented by doing a type check recursively of the expressions on both sides of the multiplication or the division operator. The results is compared with the type `INT`, to make sure both have correct type, if this is not the case we raise an error.

The same has been done for the `OR & NOT` operators, correspondingly changing the typecheck from `INT` to `BOOL`. Furthermore, seeing as `NOT` is a unary operator we have ensured that the check is only done for one expression, rather than two, as with the rest of the operators we were to implement.

4.2.1 Multiplication, division and boolean operators in interpreter

Since multiplication, division and the boolean operators is now implemented in the type checker, it needs to be implemented in the compiler and the interpreter. Here we cover how we achieved this in the interpreter. The interpreter is implemented in the file `TpInterpret.sml`, hence we need to add the missing functions in there.

The interpreter can expect to get type checked expression, therefore we can simply just evaluate each operand of the expression, and calculate the result, just like with `Plus` and `Minus`. This is very trivial for both, the only thing to note here, is in division we have to use the SML function called div, since we are interested in integer division and not floating point division.

Finally we also need to implement the boolean operators in the interpreter. Since the boolean operator `AND` is already implemented, we decided to implement `OR` and `NOT` in the same style, by having a function to evaluate each of the expression respectively `evalOR` and `evalNOT`. These helper functions simply calculate `OR` and `NOT` by using the SML functions `orelse` and `not`. The interpreter is evaluating each of the operands to the function before calling the helper functions, which means the helper function is simply gonna get two or one arguments respectively.

4.2.2 Multiplication, division and boolean operators in compiler

Finally to implement the features in the compiler, we edited the `Compilers.sml` file.

To implement the boolean operators in the compiler, we used the same approach as with the `AND` operator. The function should return a list of Mips instructions to achieve this task, hence the beginning of the list should be the output of the function `compileExp` with the first operand as argument. The list is then followed with some code that check if the first operand is different from zero (meaning the expression is true), if this is the case, it would branch to a label in the end of the expression, since there is no need to check more. If the first operand is indeed zero, there should be no branching, and the second operand should be checked, hence the result is that of `compileExp` of second operand.

The expression `NOT` is implemented in a similar manner, by using the `XORI` instruction from the `MIPS` library on the register containing the value to be negated (by using the function `compileExp`.)

Muliplication and division is a bit more simple, they basically just compile each of the operands with `compileExp`, and then append the `MIPS` instructions `MUL` and `DIV` accordingly.

4.2.3 Testing Multiplication, Division, and Boolean Operators

A number of tests was done, to check that the multiply and divide operators, as well as the boolean operators `OR & NOT`, are behaving as expected. The tests is available in `DATA/parserTest.pal`, which both test a lot of parser-related functionality, but also includes testing of the aforementioned operators.

This results does not need much explanation, as the expected results are trivial. But running the tests, confirm that the operators behave as desired.

4.3 Type inference & Type Checking

4.3.1 Type Inference

This section explains how we chose to implement type inference and type checking, focusing on what was needed to implement these properties into the Paladim compiler.

The implementation for type checking of the `read`-function, was done in `Type.sml`. The type checking is done by checking what type expression is passed to the function, and from that determining an expected type of `read`. If the expression contains a specific operator, it is possible to determine what type, the subexpression are expected to be, and let `read` inherit this property.

For simplicity, the type checking implemented, only works if it is possible to determine at the left-hand side of an expression, as implementing all possible cases was not required. This means, that the left-hand subexpression determines the expected type of the right-hand side subexpression. Furthermore, since there is no unique way of determining the type of an expression like `read() = read()`, there is no implementation to deal with this case, and the type checking will most likely fail.

The implementation demands that for example the `assign` operator must give an expected type to its subexpressions, in our case we first calculate the type of the left-hand side of the operator, then give it as expected type to the right-hand side. In the following pseudocode, the first one would succeed, but the latter would not because of aforementioned problems. The expected types has been implemented to all relevant operators including plus, minus, division etc.

$$INT = read() \tag{2}$$
$$read() = INT \tag{3}$$

4.3.2 Type checking the new-function

The function `new` is also implemented by extending the `typeCheckExp`-function in `Type.sml`. The function generate a list of arguments which is type checked. This list is then used to calculate another list containing the types of all arguments. This new list is then recursively compared against the `INT` type. If all arguments is `INT` the variable called `tpok` would be true, otherwise false. Then the rank of the array is calculated and the return type is created. If the rank is bigger than zero, and the variable `tpok` is true, the function will return the `FunApp` datatype from `TpAbSyn.sml`. The arguments to this datatype is the list of arguments calculated in the beginning, since these arguments is also containing the type of the expressions, which is required in the typed abstract syntax tree.

4.3.3 Testing Type Inference & Type Checking

The testing of type inference and type checking is done in `DATA/parterTest.pal`, line 83-93. The lines are currently commented out, since this specific test file is used to test other features of the compiler. First we test the provided case like in (4). All of our test went as expected.

$$chr(read()) = read() \tag{4}$$

4.4 Array Indexing

4.4.1 Type checking of array indexing

To implement array indexing, we needed to extend the type checker to include this case. We followed the suggested implementation steps. At first we look up the array in the variable table and extracts the type. Next we check that the provided variable is an array, and in that case we extract the rank of the array from the symbol table. We then, in a similar matter, do type checking of the new function, check if all arguments is `INT`, since indexing can only be succeeded with integers. The result of this typecheck is put into the variable `tpok`. Lastly the rank of the provided indexing is calculated, to compare with the rank from the symbol table.

If the variable `tpok` is true, meaning all arguments is integers, and the rank provided is bigger than zero, we just need to check whether the rank provided is the same as the rank in the symbol table, is this not the case, then the array is indexed wrongly, hence we need to raise an error. Otherwise we can proceed creating and returning a new `LValue` datatype of an array (with types).

4.4.2 Array indexing in compiler

To implement the array indexing, we have made 2 helper functions. The function checkInds which generate MIPS code to check if the indexing is inside the boundaries. Another function computeFlatIndex is used to generate MIPS code which calculate the index as a flat array representation, by using the method described in the assignment. The two functions are described in more detail after this section.

The first thing the compiler should do is to look up the array in the symbol table. This will return a register, which keeps the location to the beginning of the array. Then the output from the two helper functions is added. At last the type of the array is used to determine whether the flat index computed should be multiplied by four. This is the case if the array contains integers, since an integer uses four bytes in memory compared to characters and logic variables which all use only one byte. The multiplication is done by using the `MIPS` expression `SLL` (shift-left-logical), which is more efficient than using `MUL`.

Furthermore, the array header containing the dimension and strides needs to be added to the flat index, which is done by taking the amount of indexes and multiply by eight (dimension and strides takes one word of space which is four bytes). Finally the current location in memory has to be added to get the direct location in memory where our index is located.

The register to the final location is returned together with all the MIPS code to calculate the location.

BONUS: Calculating flat index without strides

As stated in the assignment it is possible to calculate the flat index without use of the stored strides. This is because the strides are simply calculated from dimensions. As example the first stride is all of the dimension except the first one multiplied together, this could easily be calculated in MIPS, instead of loading the stride from the memory, it should calculate the stride and put in a new register. By calculating from last dimension first, we would always have the last "stride" in a specific register. This stride could be used, and recalculated for next dimension. By following this approach the strides could be left out of memory and since we do not have to load each of the strides from memory, it should be more efficient.

4.4.3 Testing Array Indexing

The array indexing is tested in file `arrayInd.pal`, we decided to test arrays of both integers and characters. We test both assigning a value to the array, but also reading an value. In addition we tested the cases where the indexing was out of boundaries, and where the indexing is not integers at all (characters). All tests went as expected.

We have furthermore tested our implementations of various kind with the provided test files.

4.5 Call-by-value-result semantics

The last thing to implement in our compiler is call-by-value-result policy for procedures. This means that all variables a procedure takes as arguments, should be updated once the procedure is ending. Every single time a function or procedure is called, a new symbol table is created, our approach to implement this policy is to match the actual and formal arguments, and use the value in the new symbol table to update the former symbol table. The policy needs to be implemented in both the interpreter and compiler, which is stated in the following two sections.

4.5.1 Call-by-value-result in interpreter

This policy is implemented in interpreter by editing the function `callFun`. The function extracts the return type of the Paladim function and uses this to determine whether the Paladim function is a function or procedure. If it is a procedure, it wouldn't have any return type. If that is the case, it should simply execute our helper function called `updateOuterVtable`, with the former and new symbol tables, and the list of formal and actual arguments.

Our helper function first extracts a list of strings from the list of actual arguments. This list is used the generate another list, which is the indexes of the argument list. The second list is used to iterate over all of the arguments and for each of the arguments, the actual argument is looked up in the new symbol table, this gives us the location to an address which contains the location of the value of the actual argument. The next step is to look up the formal argument in the former symbol table, and update this value with the result from the actual argument. In the end the function returns `NONE`, since a procedure should not return anything.

4.5.2 Call-by-value-result in compiler

To implement the policy in the compiler, we have to edit the `compileF` and `compileStmt` functions in the compiler. First of all in `compileStmt`, we need to edit the generated `MIPS` code such that the arguments which are put into register by the function `putArgs` is moved back again after the procedure terminates. To do this, we have created a similar function to `putArgs`, called `rePutArgs`, this new function basically just does the inverse of `putArgs`. We also had to change the tabulate list to start from minReg instead of zero. This is because the arguments used in the procedures start with the lowest register as minReg. Finally the new functions output should be put after the `MIPS JAL` instruction.

Our helper function `rePutArgs` extracts the first expression and corresponding register from the arguments. Then it extract the name of the expression to look it up in the symbol table. The registers value is then moved to the address from the symbol table. The rest of the list is appended by recursively using the same function. If it doesn't get a LValue as argument, it should return the empty list. It could be the case if we for example give a procedure an argument directly without using a variable.

The last place to edit for enabling the policy for procedures is the `compileF` function. This function is running the register allocator, which in our case has to be modified, since we do not use register two for a return value. Instead the register allocator needs to have all of the used registers in our procedure (the arguments). Furthermore we also needed to add the MIPS code to move the variables back to registers. This second task is achieved by simply using a reverse map of `argcode`, we have called it `argcode_rev`. The first task is achieved by extracting all the registers with map from the movePairs function provided.

4.5.3 Testing Call-by-Value-Result semantics

Our call-by-value-result policy is tested by using the provided test files together with our own test file. The files used for testing are the following.

- `DATA/procReturnSimul.pal`
- `DATA/procSwap.pal`
- `DATA/proctest.pal`

All test went as expected, without issues.

5 Conclusion

The compilers demands has been met, according to the assignment specification. All of the test are being called without unexpected results.

We have successfully implemented parser, type checker, code generator and interpreter of all asked features. All of the tests has been executed with the MARS simulator, to verify the solution is working as stated.

6 References

[1] Torben Ægidius Mogensen. *Introduction to Compiler Design*. Springer, London, 2011.

[2] Cosmin Oancea. *Interpretation*. Department of Computer Science, University of Copenhagen, 2013.

[3] Jost Berthold. *Syntax Analysis*. Department of Computer Science, University of Copenhagen, 2013.

[4] Cosmin Oancea. *Type Checking*. Department of Computer Science, University of Copenhagen, 2013.

[5] Cosmin Oancea. *Machine Code Generation*. Department of Computer Science, University of Copenhagen, 2013.

[6] Cosmin Oancea. *Machine Code Generation for Function*. Department of Computer Science, University of Copenhagen, 2013.

[7] Cosmin Oancea. *Liveness Analysis And Register Allocation*. Department of Computer Science, University of Copenhagen, 2013.

[8] Jost Berthold. *Lexical Analysis*. Department of Computer Science, University of Copenhagen, 2013.

[9] Moscow ML – a light-weight implementation of Standard ML (SML). `http://mosml.org`, 2013. New release 2.10 in 2013 with substantial improvements.

[10] Mars, a MIPS Assembler and Runtime Simulator, version 4.4. `http://courses.missouristate.edu/kenvollmar/mars/`, 2002-2013. Manual: `http://courses.missouristate.edu/KenVollmar/MARS/Help/MarsHelpIntro.html`.

[11] Jost Berthold. MIPS, Register Allocation and MARS simulator. Based on an earlier version in Danish, by Torben Mogensen. Available on Absalon., 2013.

[12] David A. Patterson and John L. Hennessy. *Computer Organization & Design, the Hardware/Software Interface*. Morgan Kaufmann, 1998. Appendix A is freely available at `http://www.cs.wisc.edu/~larus/HP_AppA.pdf`.

A Source code type checker

Only edited code is shown, when code is omitted . . . is inserted instead.

```
---------------------------- edited code - SRC/Type.sml ----------------------------
...
 | typeCheckExp( vtab, AbSyn.LValue( AbSyn.Index(id, inds), pos ), etp ) =
     (* Look up id in symbol table, and extract type *)
     ( case SymTab.lookup id vtab of
         SOME id_tp =>
             let
                 (* Check if id_tp is an array, and extract rank in case *)
                 val (id_rank) = case id_tp of
                     Array (r,_) => (r)
                     | tp      => raise Error("in type checking call to array indexing, the variable is not an array, at", pos)

                 (* Create new list of arguments with types attached *)
                 val new_ids = map ( fn (e) => typeCheckExp(vtab, e, KnownType (BType Int))) inds

                 (* Generate a list of types from the new arguments *)
                 val ids_tps = map ( fn (e) => typeOfExp(e)) new_ids

                 (* Check that all types in the list args_tps is BType Int by
                 comparing and using binary AND operation on the result with
                 all the other items in the list. *)
                 val tpok = foldl (fn (e, b) => b andalso (BType Int) = e) true
                                  (ids_tps)

                 (* Get the rank of the indexes *)
                 val rank = length inds
             in
                 (* If all is of type BType Int *)
                 if tpok then
                     (* If rank is bigger than zero *)
                     if rank > 0 then
                         (* If rank in vtable equals rank given in inds *)
                         if id_rank = rank then
                             LValue(Index((id, id_tp), new_ids), pos)
                         else raise Error ("in type checking call to array indexing, rank is wrong, at", pos)
                     else raise Error ("in type checking call to array indexing, rank cannot be zero, at ", pos)
                 else raise Error ("in type checking call to array indexing, args not int, at ", pos)
             end
         | NONE    => raise Error("in type check variable, var "^id^" not in VTab, at ", pos)
     )
...
 | typeCheckExp( vtab, AbSyn.Plus (e1, e2, pos), _ ) =
     let val e1_new = typeCheckExp( vtab, e1, KnownType (BType Int) )
         val e2_new = typeCheckExp( vtab, e2, KnownType (BType Int) )
         val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)
     in  if  typesEqual(BType Int, tp1) andalso typesEqual(BType Int, tp2)
         then Plus(e1_new, e2_new, pos)
         else raise Error("in type check plus exp, one argument is not of int type "^
                         pp_type tp1^" and "^pp_type tp2^" at ", pos)
     end

 | typeCheckExp( vtab, AbSyn.Minus (e1, e2, pos), _ ) =
     let val e1_new = typeCheckExp(vtab, e1, KnownType (BType Int) )
         val e2_new = typeCheckExp(vtab, e2, KnownType (BType Int) )
         val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)
     in  if  typesEqual(BType Int, tp1) andalso typesEqual(BType Int, tp2)
         then Minus(e1_new, e2_new, pos)
         else raise Error("in type check minus exp, one argument is not of int type "^
                         pp_type tp1^" and "^pp_type tp2^" at ", pos)
     end

 | typeCheckExp ( vtab, AbSyn.Times (e1, e2, pos), _ ) =
     let val e1_new = typeCheckExp(vtab, e1, KnownType (BType Int) )
         val e2_new = typeCheckExp(vtab, e2, KnownType (BType Int) )
         val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)
     in if  typesEqual(BType Int, tp1) andalso typesEqual(BType Int, tp2)
        then Times(e1_new, e2_new, pos)
        else raise Error("in type check times exp, one argument is not of int type "^
                         pp_type tp1^" and "^pp_type tp2^" at ", pos)
     end

 | typeCheckExp ( vtab, AbSyn.Div   (e1, e2, pos), _ ) =
     let val e1_new = typeCheckExp(vtab, e1, KnownType (BType Int) )
```

```
        val e2_new = typeCheckExp(vtab, e2, KnownType (BType Int) )
        val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)
    in if  typesEqual(BType Int, tp1) andalso typesEqual(BType Int, tp2)
       then Div(e1_new, e2_new, pos)
       else raise Error("in type check div exp, one argument is not of int type "^
                        pp_type tp1^" and "^pp_type tp2^" at ", pos)
    end

  | typeCheckExp ( vtab, AbSyn.Equal(e1, e2, pos), _ ) =
     let val e1_new = typeCheckExp(vtab, e1, UnknownType )
         val tp1 = typeOfExp e1_new
         val e2_new = typeCheckExp(vtab, e2, KnownType (tp1) )
         val tp2 = typeOfExp e2_new
         (* check that tp1 is not an array type *)
         val () = case tp1 of
                    Array _ => raise Error("in type check equal, first expression "^pp_exp e1_new^
                                           "is an array (of type) "^pp_type tp1^" at ", pos)
                  | _ => ()
    in  if  typesEqual(tp1, tp2)
        then Equal(e1_new, e2_new, pos)
        else raise Error("in type check equal exp, argument types do not match "^
                         pp_type tp1^" <> "^pp_type tp2^" at ", pos)
    end

  | typeCheckExp ( vtab, AbSyn.Less (e1, e2, pos), _ ) =
    let val e1_new = typeCheckExp(vtab, e1, KnownType (BType Int))
        val e2_new = typeCheckExp(vtab, e2, KnownType (BType Int) )
        val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)
        (* check that tp1 is not an array type *)
        val () = case tp1 of
                    Array _ => raise Error("in type check less, first expression "^pp_exp e1_new^
                                           "is an array (of type) "^pp_type tp1^" at ", pos)
                  | _ => ()
    in  if  typesEqual(tp1, tp2)
        then Less(e1_new, e2_new, pos)
        else raise Error("in type check less exp, argument types do not match "^
                         pp_type tp1^" <> "^pp_type tp2^" at ", pos)
    end

  | typeCheckExp ( vtab, AbSyn.And (e1, e2, pos), _ ) =
    let val e1_new = typeCheckExp(vtab, e1, KnownType (BType Bool) )
        val e2_new = typeCheckExp(vtab, e2, KnownType (BType Bool) )
        val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)
    in  if  typesEqual(BType Bool, tp1) andalso typesEqual(BType Bool, tp2)
        then And(e1_new, e2_new, pos)
        else raise Error("in type check and exp, one argument is not of bool type "^
                         pp_type tp1^" and "^pp_type tp2^" at ", pos)
    end

  | typeCheckExp ( vtab, AbSyn.Or  (e1, e2, pos), _ ) =
    let val e1_new = typeCheckExp(vtab, e1, KnownType (BType Bool) )
        val e2_new = typeCheckExp(vtab, e2, KnownType (BType Bool) )
        val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)
    in
        if typesEqual(BType Bool, tp1) andalso typesEqual(BType Bool, tp2)
        then Or(e1_new, e2_new, pos)
        else raise Error("in type check or exp, one argument is not of bool type "^
                          pp_type tp1^" or "^pp_type tp2^" at ", pos)
    end

  | typeCheckExp ( vtab, AbSyn.Not (e1,    pos), _ ) =
    let val e1_new = typeCheckExp(vtab, e1, KnownType (BType Bool) )
        val tp1 = typeOfExp e1_new
    in  if  typesEqual(BType Bool, tp1)
        then Not(e1_new, pos)
        else raise Error("in type check not exp, one argument is not of bool type "^
                          pp_type tp1^" at ", pos)
    end
...
  (* function call to ‘new’ uses expected type to infer the to-be-read result *)
  | typeCheckExp ( vtab, AbSyn.FunApp ("new", args, pos), etp ) =
    ( case expectedBasicType etp of
        SOME btp =>
          let
            (* Create new list of arguments with types attached *)
            val new_args = map ( fn (e) => typeCheckExp(vtab, e, KnownType (BType Int))) args
          (* Generate a list of types from the new arguments *)
          val args_tps = map ( fn (e) => typeOfExp(e)) new_args

          (* Check that all types in the list args_tps is BType Int by
             comparing and using binary AND operation on the result with
             all the other items in the list. *)
          val tpok = foldl
                     (fn (e, b) => b andalso (BType Int) = e) true
                     (args_tps)
          (* Get the rank of the arguments *)
          val rank = length args

          (* Calculate return type *)
          val rtp = Array ( length args, btp )
        in
          (* If all is of type BType Int *)
          if rank > 0 then
            if tpok
            then
              FunApp(("new", (args_tps, SOME rtp)), new_args, pos)
            else raise Error ("in type checking call to new, args not int, at ", pos)
          else raise Error ("in type checking call to new, rank cannot be zero, at ", pos)
        end
    | NONE     => raise Error("in type check call to new, type inference fails because "^
                             "of unknown expected basic type, at ", pos) )
...
```

B Source code compiler

Only edited code is shown, when code is omitted ... is inserted instead.

edited code - SRC/Compiler.sml

```
...
    | compileExp( vtable, Times(e1, e2, pos), place ) =
        let val t1 = "times1_" ^ newName()
            val c1 = compileExp(vtable, e1, t1)
            val t2 = "times2_" ^ newName()
            val c2 = compileExp(vtable, e2, t2)
        in c1 @ c2 @ [Mips.MUL (place,t1,t2)]
        end
    | compileExp( vtable, Div(e1, e2, pos), place ) =
        let val t1 = "div1_" ^ newName()
            val c1 = compileExp(vtable, e1, t1)
            val t2 = "div2_" ^ newName()
            val c2 = compileExp(vtable, e2, t2)
        in c1 @ c2 @ [Mips.DIV(place,t1,t2)]
        end
...
    | compileExp( vtable, Or(e1, e2, pos), place ) =
        let val t1 = "or1_" ^ newName()
            val c1 = compileExp(vtable, e1, t1)
            val t2 = "or2_" ^ newName()
            val c2 = compileExp(vtable, e2, t2)
            val lO = "_or_" ^ newName()
        in c1 (* do first part, skip 2nd part if already true *)
          @ [Mips.MOVE (place, t1), Mips.BNE (place, "0", lO) ]
          @ c2 (* when here, t1 was false, so the result is t2 *)
          @ [Mips.MOVE (place, t2), Mips.LABEL lO ]
        end

    | compileExp( vtable, Not(e1, pos), place ) =
        let val t1 = "not1_" ^ newName()
            val c1 = compileExp(vtable, e1, t1)
        in c1 @ [ Mips.XORI (place, t1, "1") ]
        end

   and rePutArgs [] vtable regs = []
     | rePutArgs (LValue (Var(e, _), _)::es) vtable (re::regs) =
      let
         val st = ( case SymTab.lookup e vtable of
           SOME x => x
         | NONE => raise Error("Variable "^e^" not found in vtable", (0,0)))
          val code = rePutArgs es vtable regs
      in
```

```
          [Mips.MOVE (st, re)] @ code
      end
      (* In case of anything else than variable name, we are not interested
         in updating, since we cannot use the value anyway *)
     | rePutArgs _ _ _ = []
...
    | compileLVal( vtab : VTab, Index ((n,t),inds) : LVAL, pos : Pos ) =
        let
            val place = "_cur_"^newName() (* Current memory location *)
            val (line,_)= pos (* Extract line for error messages *)

            fun checkInds(res_reg, place, inds) =
                let
                    val cond_reg = "_cond_"^newName()
                    val cond_r   = "_cond_"^newName()
                    val index    = "_index_"^newName()
                    val e_reg    = "_tmp_"^newName()
                    val arr_rank = length inds
                    val arr = List.tabulate(arr_rank, fn x => x)
                in
                    foldl (fn ((arr,i),code) =>
                        let
                            val code_e     = compileExp(vtab, i, index)
                            val code_check = [
                              Mips.LW(e_reg, place, makeConst (arr*4)),
                              Mips.SLT(cond_reg, index, "0"),
                              Mips.XORI(cond_reg, cond_reg, "1"),
                              Mips.SLT(cond_r, index, e_reg),
                              Mips.AND(res_reg, cond_r, cond_reg),
                              Mips.LI("5", makeConst line), (* Error at line *)
                              Mips.BEQ(res_reg, "0", "_IllegalArrSizeError_")
                            ]
                        in
                          if arr > arr_rank then raise Error("Too many dimensions!", pos) else
                          code_e @ code_check @ code
                        end
                    ) [] (ListPair.zip (arr, inds))
                end
                fun computeFlatIndex(res, place, inds) =
                let
                    val arr_rank = length inds
                    val idx  = (List.tabulate(arr_rank, fn x => x))
                    val e_reg    = "_reg_"^newName()
                    val tmp    = "_tmp_"^newName()
                    val stride = "_stride_"^newName()
                    val code_res_init = [ Mips.ADDI(res, "0", "0") ]
                    val code_strides = foldl (fn ((i, e),code) =>
                        let val code_e = compileExp(vtab, e, e_reg)
                            val code_d = [ Mips.LW  (stride, place,   makeConst ((arr_rank+i)*4)),
                                           Mips.MUL (tmp, stride, e_reg),
                                           Mips.ADD (res, res, tmp) ]
                        in
                            if (i = arr_rank-1) then
                                code_e @ [ Mips.ADD (res, "0", e_reg) ] @ code
                            else
                                code_e @ code_d @ code
                        end
                    ) [] ( ListPair.zip (idx, inds) )
                in
                    code_res_init @ code_strides
                end
        in ( case SymTab.lookup n vtab of
            SOME reg => let
                (* reg has memory location! *)
                val check    = "_check_"^newName()
                val res      = "_res_"^newName()
                val tmp      = "_tmp_"^newName()
                val code = checkInds(check, reg, inds) @
                          computeFlatIndex(res, reg, inds)
            val code = ( case t of
                Array (_,Int) => code @
                    [ Mips.LI(tmp, makeConst (length inds)),
                      Mips.SLL(tmp, tmp, "3"),
                      Mips.SLL(res, res, "2"), (* Int = 4 bytes *)
                      Mips.ADD(res, res, tmp),
                      Mips.ADD(res, res, reg) ]
                | _ => code @
```

```
                    [ Mips.LI(tmp, makeConst (length inds)),
                      Mips.SLL(tmp, tmp, "3"),
                      Mips.ADD(res, res, tmp),
                      Mips.ADD(res, res, reg) ]
                )
            in
                (code, Mem res)
            end
          | NONE => raise Error ("unknown variable "^n^" at, ", pos) )
        end
...
        | ProcCall ((n,_), es, p) =>
          let
              val (mvcode, maxreg) = putArgs es vtable minReg
              val regs = List.tabulate (maxreg - minReg, (fn reg => makeConst (reg + (minReg))))
              val rev_code = rePutArgs es vtable regs
          in
              mvcode
              @ [Mips.JAL (n, regs)]
              @ rev_code
          end
...
  and compileF (isProc, fname, args, block, pos) =
      (* at this point, we do not care about the return type (or no return) *)
      let (* make a vtable from bound formal parameters,
               then evaluate expression in this context, return it *)
          (* arguments passed in registers, "move" into local vars.
             Code generator imposes max. 13 arguments (maxCaller-minReg)
           *)
          val () = if length args <= maxCaller - minReg + 1 then ()
                   else raise Error (fname ^ ": too many arguments (max "
                                     ^ makestring (maxCaller - minReg + 1)
                                     ^")", pos)
          val (movePairs, vtable) = getMovePairs args [] minReg
          val argcode = map (fn (vname, reg) => Mips.MOVE (vname, reg)) movePairs
          val argcode_rev = map (fn (vname, reg) => Mips.MOVE (reg, vname)) movePairs

          val body = compileStmts block vtable (fname ^ "_exit")
          val reg_args = map (fn (_,r) => r) movePairs

          val (body1, _, maxr, spilled) =  (* call register allocator *)
            if isProc then
              RegAlloc.registerAlloc ( argcode @ body @ argcode_rev )
                                     reg_args minReg maxCaller maxReg 0
            else
              RegAlloc.registerAlloc ( argcode @ body )
                                     ["2"] minReg maxCaller maxReg 0
                                     (* 2 contains return val*)
          val (savecode, restorecode, offset) = (* save/restore callee-saves *)
              stackSave (maxCaller+1) maxr [] [] (4*spilled)
        in  [Mips.COMMENT ("Function " ^ fname),
             Mips.LABEL fname,       (* function label *)
             Mips.SW (RA, SP, "-4"), (* save return address *)
             Mips.ADDI (SP,SP,makeConst (~4-offset))] (* move SP "up" *)
          @ savecode                 (* save callee-saves registers *)
          @ body1                    (* code for function body *)
          @ [Mips.LABEL (fname^"_exit")] (* exit label *)
          @ restorecode              (* restore callee-saves registers *)
          @ [Mips.ADDI (SP,SP,makeConst (4+offset))] (* move SP "down" *)
          @ [Mips.LW (RA, SP, "-4"),  (* restore return addr *)
             Mips.JR (RA, [])]       (* return *)
        end
...
```

C Source code driver

Only edited code is shown, when code is omitted ... is inserted instead. This change gets the compiler to use the new parser.

edited code - SRC/Driver.sml -

```
...
   let
     (*val pgm = LL1Parser.parse Lexer.Token lexbuf*)
     (* COMMENT LINE ABOVE AND UNCOMMENT  *)
     (* THE LINE BELOW TO USE YOUR PARSER *)
     val pgm = Parser.Prog Lexer.Token lexbuf
   in case arg of
...
```

D Source code parser

Only edited code is shown, when code is omitted ... is inserted instead.

edited code - SRC/Parser.grm -

```
/** Tokens **/

/* Literals */
%token <string*(int*int)> ID           /* Type, position(line, char position) */
%token <int*(int*int)> NUMLIT          /* Type, position (line,char position) */
%token <bool*(int*int)> LOGLIT         /* Type, position (line,char position) */
%token <char*(int*int)> CHALIT         /* Type, position (line,char position) */
%token <string*(int*int)> STRLIT       /* Type, position (line,char position) */

/* Keywords */
%token <(int*int)> PROGRAM FUNCTION PROCEDURE VAR BEGIN END IF THEN ELSE
%token <(int*int)> WHILE DO RETURN

/* Type Keywords */
%token <(int*int)> INT CHAR BOOL ARRAY OF

/* Symbols */
%token <(int*int)> SEMICOLON COLON COMMA ASSIGN EOF  /* ASSIGN = ':=' */

/* Arithmetic */
%token <(int*int)> PLUS MINUS TIMES DIVIDE EQUAL LESS AND OR NOT

/* Parantheses */
%token <(int*int)> LPAR RPAR RCBR LCBR LBRA RBRA  /* RCBR = Right Curly Brace, ... */

/* VARIOUS - Dont know if they are neccessary! */
%token <(int*int)> TRUE FALSE

%start Prog

/* Lowest precedence in top */
%nonassoc OptElse
%nonassoc ELSE
%left OR
%left AND
%left NOT
%left EQUAL LESS
%left PLUS MINUS
%left TIMES DIVIDE
//%left       /* Left associative */
//%nonassoc   /* Non associative*/
/** Highest precedence in bottom */

/** Types **/
%type <AbSyn.FunDec list> Prog
%type <AbSyn.FunDec list> FunDecs
%type <AbSyn.FunDec> FunDec
%type <AbSyn.StmtBlock> Block
%type <AbSyn.Stmt list> SBlock
%type <AbSyn.Stmt list> StmtSeq
%type <AbSyn.Stmt> Stmt
%type <AbSyn.Exp option> Ret
%type <AbSyn.Exp> Exp
%type <AbSyn.Dec list> PDecl
%type <AbSyn.Dec list> Params
%type <AbSyn.Dec> Dec
%type <AbSyn.Dec list> Decs
%type <AbSyn.Type> Type
%type <AbSyn.Exp list> CallParams
%type <AbSyn.Exp list> Exps

%%

/* Rules  */
/* Program structure */
Prog : PROGRAM ID SEMICOLON FunDecs EOF      $4
;

FunDecs : FunDecs FunDec                     $1 @ [$2]
        | FunDec                             [$1]
;

FunDec : FUNCTION ID LPAR PDecl RPAR COLON Type Block SEMICOLON  AbSyn.Func ($7, #1 $2, $4, $8, $1)
                                                                /* Type, Identifier, PDecl, Block, func pos */
       | PROCEDURE ID  LPAR PDecl RPAR Block SEMICOLON           AbSyn.Proc (#1 $2, $4, $6, $1)
                                                                /* Identifier, PDecl, Block, proc pos */
;

Block : VAR Decs SBlock                       AbSyn.Block ($2, $3)
      | SBlock                                AbSyn.Block ([], $1)
;

SBlock : BEGIN StmtSeq SEMICOLON END         $2
       | Stmt                                [$1]
;

StmtSeq : StmtSeq SEMICOLON Stmt             $1 @ [$3]
        | Stmt                               [$1]
;

/* Statements */
Stmt : ID LPAR CallParams RPAR               AbSyn.ProcCall (#1 $1, $3, #2 $1)
     | IF Exp THEN Block ELSE Block          AbSyn.IfThEl($2, $4, $6, $1)
     | IF Exp THEN Block %prec OptElse       AbSyn.IfThEl($2, $4, AbSyn.Block([],[]), $1)
     | WHILE Exp DO Block                    AbSyn.While($2, $4, $1)
     | RETURN Ret                            AbSyn.Return($2, $1)
     | ID ASSIGN Exp                         AbSyn.Assign(AbSyn.Var(#1 $1), $3, #2 $1)
     | ID LBRA Exps RBRA ASSIGN Exp          AbSyn.Assign(AbSyn.Index(#1 $1, $3), $6, #2 $1)
;

Ret : Exp                                    SOME $1
    |                                        NONE
;

Exp : NUMLIT                                 AbSyn.Literal (AbSyn.BVal (AbSyn.Num (#1 $1)) , #2 $1)
    | LOGLIT                                 AbSyn.Literal (AbSyn.BVal (AbSyn.Log (#1 $1)) , #2 $1)
    | CHALIT                                 AbSyn.Literal (AbSyn.BVal (AbSyn.Chr (#1 $1)) , #2 $1)
    | STRLIT                                 AbSyn.StrLit $1
    | LCBR Exps RCBR                         AbSyn.ArrLit ($2, $1)
    | ID                                     AbSyn.LValue(AbSyn.Var (#1 $1),#2 $1)
    | ID LBRA Exps RBRA                      AbSyn.LValue(AbSyn.Index (#1 $1, $3), #2 $1)
    | NOT Exp                                AbSyn.Not ($2, $1)
    | LPAR Exp RPAR                          $2
    | ID LPAR CallParams RPAR                AbSyn.FunApp (#1 $1, $3, #2 $1)
    | Exp PLUS Exp                           AbSyn.Plus ($1, $3, $2)
    | Exp MINUS Exp                          AbSyn.Minus ($1, $3, $2)
    | Exp TIMES Exp                          AbSyn.Times($1, $3, $2)
    | Exp DIVIDE Exp                         AbSyn.Div($1, $3, $2)
    | Exp EQUAL Exp                          AbSyn.Equal($1, $3, $2)
    | Exp LESS Exp                           AbSyn.Less($1, $3, $2)
    | Exp AND Exp                            AbSyn.And($1, $3, $2)
    | Exp OR Exp                             AbSyn.Or($1, $3, $2)
;

/* Variable and Parameter, Declarations, Types */
PDecl : Params                               $1
      |                                      []
;

Params : Params SEMICOLON Dec                $1 @ [$3]
       | Dec                                 [$1]
;

Dec : ID COLON Type                          AbSyn.Dec(#1 $1, $3, #2 $1)
;

Decs : Decs Dec SEMICOLON                    $1 @ [$2]
     | Dec SEMICOLON                         [$1]
;

Type : INT                                  AbSyn.Int $1
```

```
     | CHAR                                AbSyn.Char $1
     | BOOL                                AbSyn.Bool $1
     | ARRAY OF Type                       AbSyn.Array ($3,$1)
;

/* Function and Procedure Parameters and Index Lists */
CallParams : Exps                           $1
           |                                []
;

Exps : Exp COMMA Exps                       $1 :: $3
     | Exp                                  [$1]
;

/* Trailer*/
```

E Source code lexer

Only edited code is shown, when code is omitted ... is inserted instead. Only changes here is to use the new parser.

edited code - SRC/Lexer.lex -

```
...
fun keyword (s, pos) =
     case s of
         "program"      => Parser.PROGRAM    pos
       | "function"     => Parser.FUNCTION   pos
       | "procedure"    => Parser.PROCEDURE  pos
       | "var"          => Parser.VAR        pos
       | "begin"        => Parser.BEGIN      pos
       | "end"          => Parser.END        pos
       | "if"           => Parser.IF         pos
       | "then"         => Parser.THEN       pos
       | "else"         => Parser.ELSE       pos
       | "while"        => Parser.WHILE      pos
       | "do"           => Parser.DO         pos
       | "return"       => Parser.RETURN     pos
       | "array"        => Parser.ARRAY      pos
       | "of"           => Parser.OF         pos
       | "int"          => Parser.INT        pos
       | "bool"         => Parser.BOOL       pos
       | "char"         => Parser.CHAR       pos
       | "and"          => Parser.AND        pos
       | "or"           => Parser.OR         pos
       | "not"          => Parser.NOT        pos
       | "true"         => Parser.LOGLIT     (true, pos)
       | "false"        => Parser.LOGLIT     (false, pos)
       (* Else it must be a identifier *)
       | _              => Parser.ID         (s, pos)

   (* "lex" will later be the generated function "Token" *)
   fun repeat lex b
              = let val res = lex b
                in case res of
                         Parser.EOF _ => []
                       | other => other :: repeat lex b
                end
...
rule Token = parse
...
  | [‘0‘-‘9‘]+             case Int.fromString (getLexeme lexbuf) of                                NONE   => lexerError lexbuf "Bad integer"
  | ‘’‘ ([‘ ‘ ‘!‘ ‘#‘-‘&‘ ‘(‘-‘[‘ ‘]‘-‘~‘] | ‘[‘ ‘-‘~‘]) ‘’‘
                           Parser.CHALIT                            ((case String.fromCString (getLexeme lexbuf) of

  | ‘"‘ ([‘ ‘ ‘!‘ ‘#‘-‘&‘ ‘(‘-‘[‘ ‘]‘-‘~‘] | ‘[‘ ‘-‘~‘])* ‘"‘
                           Parser.STRLIT                            ((case String.fromCString (getLexeme lexbuf) of
...
  | ":="                   Parser.ASSIGN    (getPos lexbuf)
  | ‘+‘                    Parser.PLUS      (getPos lexbuf)
  | ‘-‘                    Parser.MINUS     (getPos lexbuf)
  | ‘*‘                    Parser.TIMES     (getPos lexbuf)
  | ‘/‘                    Parser.DIVIDE    (getPos lexbuf)
  | ‘=‘                    Parser.EQUAL     (getPos lexbuf)
  | ‘<‘                    Parser.LESS      (getPos lexbuf)
  | ‘(‘                 Parser.LPAR      (getPos lexbuf)
  | ‘)‘                 Parser.RPAR      (getPos lexbuf)
  | ‘[‘                 Parser.LBRA      (getPos lexbuf)
  | ‘]‘                 Parser.RBRA      (getPos lexbuf)
  | ‘‘                 Parser.LCBR      (getPos lexbuf)   | ‘‘                 Parser.RCBR      (getPos lexbuf)

  | ‘,‘                 Parser.COMMA     (getPos lexbuf)
  | ‘;‘                 Parser.SEMICOLON (getPos lexbuf)
  | ‘:‘                 Parser.COLON     (getPos lexbuf)

  | eof                 Parser.EOF       (getPos lexbuf)
...
```

F Source code typed abstract syntax tree

Only edited code is shown, when code is omitted . . . is inserted instead.

edited code - SRC/TpAbSyn.sml -

```
...
    | Times   of Exp * Exp         * Pos      (* e.g., x * 3 *)
    | Div     of Exp * Exp         * Pos      (* e.g., x / 3 *)
...
    | Or      of Exp * Exp         * Pos      (* e.g., (x=5) or y *)
    | Not     of Exp               * Pos      (* e.g., not (x>3) *)
...
    | pp_exp (Times (e1, e2, _))    = "( " ^ pp_exp e1 ^ " * " ^ pp_exp e2 ^ " )"
    | pp_exp (Div   (e1, e2, _))    = "( " ^ pp_exp e1 ^ " * " ^ pp_exp e2 ^ " )"
...
    | pp_exp (Or    (e1, e2, _))    = "( " ^ pp_exp e1 ^ " | " ^ pp_exp e2 ^ " )"
    | pp_exp (Not   (e1, _)    )    = "( not " ^ pp_exp e1 ^ " )"
...
    | typeOfExp ( Times  (a,b,_) ) = typeOfExp a
    | typeOfExp ( Div    (a,b,_) ) = typeOfExp a
...
    | typeOfExp ( Or     (_,_,_) ) = BType Bool
    | typeOfExp ( Not    (_,_)   ) = BType Bool

...
    | posOfExp  ( Times  (_,_,p) ) = p
    | posOfExp  ( Div    (_,_,p) ) = p
...
    | posOfExp  ( Or     (_,_,p) ) = p
    | posOfExp  ( Not    (_,p)   ) = p
...
```

G Source code interpreter

Only edited code is shown, when code is omitted . . . is inserted instead.

edited code - SRC/TpInterpret.sml -

```
...
fun evalOr (BVal (Log b1), BVal (Log b2), pos) = BVal (Log (b1 orelse b2))
  | evalOr (v1, v2, pos) =
        raise Error( "Or: argument types do not match. Arg1: " ^
                     pp_val v1 ^ ", arg2: " ^ pp_val v2, pos )

fun evalNot (BVal (Log b1), pos) = BVal (Log (not b1))
  | evalNot (v1, pos) =
        raise Error( "Not: argument types do not match. Arg1: " ^
                     pp_val v1, pos )
...
        | other    =>
            let val new_vtab = bindTypeIds(fargs, aargs, fid, pdcl, pcall)
                val res  = execBlock( body, new_vtab, ftab )
            in  ( case (rtp, res) of
                    (NONE  , _     ) => updateOuterVtable vtab new_vtab (aexps, fargs)
                  | (SOME t, SOME r) => if   typesEqual(t, typeOfVal r)
                                        then SOME r
                                        else raise Error("in call fun: result does " ^
                                                         "not match the return type! In fun/proc:" ^ fid ^
```

```
                                                          " rettype: "^pp_type t^" result: "^pp_val r, pcall )
                  | otherwise        => raise Error("in call: fun/proc "^fid^" illegal result: ", pcall ) )
            end
      end
...
and updateOuterVtable vtabOuter vtabInner (out_exp, in_arg) =
let
   val strlist = map (fn Dec((s,_),(_,_)) => s) in_arg
   val ind_count  =  (List.tabulate(length strlist, fn x => x))

   val _ = map ( fn (idx) =>
               let val exp = List.nth(out_exp, idx)
                   val s   = List.nth(strlist, idx)
               in
                  ( case SymTab.lookup s vtabInner of
                    NONE => raise Error("Error input not understood", (0,0))
                  | SOME result =>
                           (*raise Error(s^" := "^pp_exp e, (0,0))*)
                           ( case exp of
                              LValue ( Var( id, _), _ ) =>
                              (
                               case SymTab.lookup id vtabOuter of
                                 SOME adr =>
                                 adr := !result
                               | NONE => raise Error("DOESNT WORK", (0,0))
                              )
                           | _ => raise Error("DOESNT WORK", (0,0))
                           )
                  )
               end
         ) ind_count
in
   NONE
end
...
  | evalExp ( Times(e1, e2, pos), vtab, ftab ) =
        let val res1   = evalExp(e1, vtab, ftab)
            val res2   = evalExp(e2, vtab, ftab)
        in  evalBinop(op *, res1, res2, pos)
        end
  | evalExp ( Div(e1, e2, pos), vtab, ftab ) =
        let val res1   = evalExp(e1, vtab, ftab)
            val res2   = evalExp(e2, vtab, ftab)
        in  evalBinop(op div, res1, res2, pos)
        end
...
  | evalExp ( Or(e1, e2, pos), vtab, ftab ) =
        let val r1 = evalExp(e1, vtab, ftab)
            val r2 = evalExp(e2, vtab, ftab)
             in  evalOr(r1, r2, pos)
               end

  | evalExp ( Not(e1, pos), vtab, ftab ) =
        let val r1 = evalExp(e1, vtab, ftab)
             in  evalNot(r1, pos)
               end
...
```