Paladim Compiler

Alexander Worm Olsen - bdj816

Chi Dan Pham - vqr853

Troels Thompsen - qvw203

Group Project

December 20 2013

Departmen of Computer Science
University of Copenhagen

Contents

Introduction

The goal of the group assignment is to implement a compiler for the Paladim language using a bottom-up grammar. The langauge we use for this implementation is Stanard ML. The majority of the compiler has already been implemented, and the changes we made are described in the individual tasks below. The first two tasks of the assigment was completed as a milestone assignment. We have made slight changes since the milestone, which improves the work done in these two tasks. A full description of these tasks and all the changes we made, are also included here.

Task 1

We were asked to implement the grammar production rules for the paladim language, which is described in the group project description document.

In order to implement the grammar, we have edited the empty file *Parser.grm*, which was already included in the source code handout. We also edited the *Driver.sml* and the *Lexer.lex* source code file, to use our new parser structure instead of the LL1Parser which was included in the handout.

For inspirational purposes we used the file *example.pdf* which we found on absalon, and the groupproject description document. In these documents we found examples for creating type precedence, how to construct terminals and non-terminals, and how to use them correctly in the grammar.

The following sections describe the changes made to each file respectively.

Parser.grm

Terminals

The first definition in the parser, is the definition of terminals, which in mosmlyac is called tokens. Here we use the abstract syntax type definitions to define the types of our tokens. The tokens themselves are defined in the *Lexer.lex* file. We simply found all tokens in the lexer, and defined tokens for them.[1]

Precedence Rules

Next we define our precedence rules, to prevent shift / reduce conflicts. The rules at the bottom take the highest precedence. We use left associative precedence for

[1] See Appendix A. Parser.grm Listing 1.

arithmetic operations, ranking *TTimes* and *TSlash* higher than *TPlus* and *TMinus*.

We define comparison operations *TLess* and *TEq* as non-associative, since it is not specified whether comparison is associative in Paladim in the project description.

The logical operation *TNot* is defined as non-associative because it is unary. We define the other logical operations *TAnd* and *TOr* as left-associative (they can be either left- or right-associative, but should not be non-associative). Here *TNot* takes precedence over *TAnd* which takes precedence over *TOr*.

The last two precedence rules are defined to resolve the shift / reduce conflict, which occurs when trying to write productions for If-Then-Else and If-Then. This conflict occurs in the following scenario.

```
if a > b then
    if c = d then
        return a
    else
        return b
```

In this scenario the parser does not know whether to match the production for If-Then-Else, or the production for If-Then. If it shifts, the else belongs to the inner if-statement, but if it reduces, the else belongs to the outer if-statement. In this scenario, we actually always wants to shift, because the else should belong to the closest previous if-then-statement. In order to accomplish this, we define TElse to take precedence over "LowPrec". We later assign "LowPrec" to the If-Then production grammar, and thus resolve the shift / reduce conflict.[2]

Non-Terminals

In this section we define our start symbol, and the non-terminals for our productions. The types used for the non-terminals are defined in *AbSyn.sml*. The actual non-terminals are defined in figure 3 in the group project definition document. The start symbol "Program" was defined by us, in order to handle end of file.[3]

Grammar

In this section we define the grammar[4] used by the parser. For easier analysis we have divided the grammar into sections. In general we built the grammar by looking

[2]See Appendix A. Parser.grm Listing 1.
[3]See Appendix A. Parser.grm, Listing 2
[4]See Appendix A. Parser.grm Listing 2-5

at the production definitions in figure 3 in the group project definition. The type we return in the productions, are defined in the *AbSyn.sml*.

Program structure

The most important change in this section, is the Block production. To begin with we had a DBlock production, as described in figure 3 in the group project definition. This caused a shift / reduce conflict, which we resolved by combining the DBlock with the Block production.[5]

Statements, Values and Expressions

The most important change in this section, is the *TIf Exp TThen Block %prec LowPrec* production. As described in the non-terminal definitions, we need to assign lower precedence to the If-Then production. This is done by adding %prec LowPrec at the end of the production.[6]

The Exp LVal production in this section might be problematic. This production needs to return an AbSyn.LValue(LVal, Pos). However this is not possible, because we call another production which only returns an LVal without a position. To solve this issue, we changed the non-terminal definition from <AbSyn.LVAL> to <AbSyn.LVAL*AbSyn.Pos>[7]. This allows us to give our LValue the correct position, without modifying the syntax definition in AbSyn.sml.

Parameters and Procedures

At the end we have our simple definitions for parameters and declarations.[8]

Driver.sml

In the driver file we have uncommented the two lines which was meant for the LL1 parser and instead inserted the appropriate ones for our parser (line 57 and 69). Furthermore we have uncommented the Parser.ParseError, because this error message will be caught by the Parsing.ParseError, and instead only created compile errors.[9]

[5]See Appendix A. Parser.grm Listing 3.
[6]See Appendix A. Parser.grm Listing 3.
[7]See Appendix A. Parser.grm Listing 2.
[8]See Appendix A. Parser.grm Listing 5.
[9]See appendix B. Driver.sml Listing 6.

Lexer.lex

In the *Lexer.lex* we have changed all instances of LL1parser to Parser, thereby including our newly created Parser.grm instead of the handout. Please note that this code is left out.

Task 2

In this task we were to implement the functionality of multiplication, division, or and not in Paladim. To do this we had to go through several source code files and implement the functionality of each of them. The files we've changed include; *Compiler.sml*, *TpInterpret.sml*, *Type.sml* and *TpAbSyn.sml*. Furthermore we were to make sure our precedence for boolean operators and arithmetic operations were correct, please note that this was done already in task 1[10]

TpAbSyn.sml

In order to implement the functionality of the four operations we started in *TpAbSyn.sml*, where we first uncommented the four of them in the datatype for *exp*, afterwards we made pretty printing for the four of the functions with inspiration from the ones already implemented, then a function that was able to find the position of each of the operations and at last we implemented the functionality to find the type of the operations.[11]

TpInterpret.sml

In this file we implemented the functionality of the four functions for the Interpreter, this was done with inspiration from the ones already implemented, this was done in *evalExp*. In order to implement the to logical operations we had needed the a helper function for each of them. These changes made the functionality of our four functions work in the interpreter.[12]

Type.sml

In order to implement the functions for our compiler aswell we had to include them in the *type.sml*. These were aswell inspired by the ones already there. Please note that the types stated weren't added until task 3, and that the one for multiplication is left out because it's very similar to the one for division.[13]

[10] See Appendix A. Parser.grm Listing 1.
[11] See Appendix C. TpAbSyn.sml Listing 7.
[12] See Appendix D. TpInterpret.sml Listing 8.
[13] See Appendix E. Type.sml Listing 9

Compiler.sml

The last step for the full implementation of multiplication, division, or and not were to implement them in *compiler.sml*. The functionality of the four operations were straight forward and inspired by the ones already implemented. For the functionality of not in Mips code we used the mips instruction $XORI$. This is due to the fact that there doesn't exist a NOT instruction in Mips. XOR returns 1 (true) if one of the arguments is different from 1. In order to use this in the not scenario we pass the value to be negated, together with 1 as the second argument.$XORI$ then returns true if the input argument is 0 and false if the argument is 1, as we would expect of not.

Task 3

In this task we were to implement typechecking for Paladims library functions read() and new()

Type.sml

The major change in the task was modifying the typeCheckExp function, to correctly typecheck the functions *read* and *new*.

In order to accomplish type checking for *read*, the typechecker needs to determine the expected type of the argument calling context. In the majority of the cases we changed from UnknownType to KnownType (<expected type>). E.g. in Plus, the expected type of the operands are now KnownType (BType Int) since Plus is only defined for integers (ie. the operator is typed).
Two operations differ from this; Less and Equal, these two still holds UnkownType for argument1 but here the second argument is the same as what the 1. argument turns out to be. This is due to the fact that they are polymorphic. Even though they are polymorphic, the types of both arguments needs to be equal. We cannot infer exactly what types, but only that they must be equal (as suggested in the assignment description, we have only implemented this so it works one-way).

In order to accomplish type checking for *new*, the typechecker raises an error if not all the arguments are integers. Otherwise it creates a typed FunApp with the arguments (dimensions of the new array) converted to TpAbSyn.Type as well. [14]

[14] See Appendix E. Type.sml Listing 10, 11 and 12.

Task 4

In this task we were to implement type checking and code generation for array indexing.

Type.sml

We added type checking by modifying typeCheckExp for the AbSyn.LValue(AbSyn.Index) case. The function raises an error if the array name doesn't exist in the symbol table, if the number of indices is not equal to the number of ranks in the array (which is nonzero since the array is in the symbol table and therefore have been type checked already), or if not all indices are of type Int.
It then creates new typed versions of all the indices and constructs a typed array index which it returns.
Bounds checking is not possible at this point since it would require a recursive evaluation function for all expressions. This is instead done runtime. [15]

Compiler.sml

We added machine code generation by modifying compileLVal for the Index case. The generated Mips code supports the following functionality:
Each index is added to the stack so they can be accessed in a loop afterwards (we could also have put them in new registers and let the register allocator do the work, but this way, we know exactly where our indices are.).
The Mips code then enters a loop that runs for each index. It gets the next dimension and index, simultaneously checking for bounds and calculating the flat index. This is done without using the store strides since

$$i_{flat} = i_n + d_n(i_{n-1} + d_{n-1}(...i_2 + d_2 i_1))$$

for an array with n dimensions.
After the flat index is calculated, the strides are skipped and the flat index is added to the array (dereferencing the array data pointer).

A disadvantage to this approach is that the code is generated anew each time an array is indexed even though the procedure is exactly the same. This could be fixed by placing the code once in the assembly file and the jumping to it every time an array index needs to be calculated.
This would require something similar to a Mips function call, because you would

[15] See Appendix E. Type.sml listing 12.

have to pass parameters to it via the stack and handle local variables etc. [16]

Task 5

In this task we were to change the way Paladims procedures passes their arguments from call-by-value to call-by-value-result.

TpInterpreter.sml

In order to make Paladim procedures use call-by-value-result we change the *TpInterpreter.sml*. We use the references from our innerVtable and swap them with the references for the according arguments in the outerVtable.[17] To do this we used the already implemented arguments *fargs* and *aexps*.

Compiler.sml

In order to implement call-by-value-result in the compiler, we created a function similar to putArgs called popArgs, which takes the same arguments as putArgs, plus the Mips codes returned by putArgs.
PopArgs then checks whether or not the expression is an LValue, and the Mips code is an ORI. In this case popArgs takes the caller register which is the second argument of ORI. PopArgs then stores the value in the temporary procedure argument register, in the caller register.
In any other case where the Mips command is not an ORI operation, or the expression is not an LValue expression, we simply continue the recursion. We do this in order to account for procedure arguments which are not variables, but constant values or constant expressions.

PopArgs is called right after putArgs in *compileStmt*, and simply appended to the existing mvcode[18].

We also added a small change to *compileF*. The variable $new_argcode$ is used to store the Mips code created by popArgs, if we are compiling a procedure which is not main procedure. This code is then appended to the existing body of the procedure[19]. This is required in order for the register allocator to correctly swap the register values.

[16] See Appendix F. Compiler.sml listing 14,15,16
[17] See appendix D. TpInterpreter.sml Listing 9
[18] See appendix F. Compiler.sml Listing 17.
[19] See appendix F. Compiler.sml Listing 17.

Testing

We have made one or two specific tests for each task, these can be found in Appendix G. Testing. In the DATA folder we've included all tests that were able to compile with our Paladim compiler, i.e. removed a few of the handed out tests which used functions such as readChar which isn't supported.

Listing 18 in the Appendix includes two tests for task 1, the first one *dangling* is made for the dangling-else case, which we've handled by precedence in our grammar. The second one is testing general functionality of our parser.

Listing 19 tests the different precedence rules of arithmetic and logical operations.

Listing 20 tests the functionality of our task 3 typechecking, in order to test the functionality of read() and type inference. In order to see that our implementation for new() is functional please see listing 21.

In order to test array indexing implemented in task 4 we've made a test shown in listing 21. Here we test for the different types of arrays, sizes and indexing in all of them. The case with negative indexing isn't shown due to the fact that we can't handle exceptions.

The last two tests, listing 22 and 23, tests our implementation of call-by-value-result with procedures in Paladim. The last of the two shows the example where constants and variables are used as procedure arguments.

Appendix A. Parser.grm

Listing 1: The tokens and precedence rules for our grammar in Parser.grm.

```
%{

%}

/* Token type definitions (will often be used in the Lexer)
 * Tokens use position attribute for demonstration
 * (see below for Lexer)
 * As mentioned, the SML code above ends up after
 * this data declaration,
 * so we cannot use any types defined above
 * at this point of the file.
 * tokens needs to be of sml types
 * through the rules the should be converted to an absyn syntaxtree */

%token <AbSyn.Pos> TProgram TFunction TProcedure TVar TBegin TEnd
                   TIf TThen TElse TWhile TDo TReturn TArray TOf
                   TInt TBool TChar TAnd TOr TNot TAssign TPlus TMinus
                   TTimes TSlash TEq TLess TLParen TRParen TLBracket
                   TRBracket TLCurly TRCurly TComma TSemi TColon TEOF
%token <bool*AbSyn.Pos> TBLit
%token <int*AbSyn.Pos> TNLit
%token <char*AbSyn.Pos> TCLit
%token <string*AbSyn.Pos> TSLit TId

%nonassoc LowPrec
%nonassoc TElse
%right TOr
%right TAnd
%nonassoc TNot
%nonassoc TEq TLess
%left TPlus TMinus
%left TTimes TSlash

/* start symbol */
%start Program
```

Listing 2: The types and some of the grammar for the Parser.grm. (continued from listing 1)

```
/* types returned by rules below */
%type <AbSyn.Prog> Program
%type <AbSyn.Prog> Prog
%type <AbSyn.Prog> 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.LVAL*AbSyn.Pos> LVal
%type <AbSyn.Exp option> Ret
%type <AbSyn.Exp> Exp
%type <AbSyn.Dec list> PDecl Params
%type <AbSyn.Dec> Dec
%type <AbSyn.Dec list> Decs
%type <AbSyn.Type> Type
%type <AbSyn.Exp list> CallParams Exps

%%

/* rules - a separate start rule is added automatically */

/* PROGRAM STRUCTURE*/
Program: Prog TEOF                    { $1 }
;

Prog :
    TProgram TId TSemi FunDecs        { $4 }
;

FunDecs :
    FunDecs FunDec                    { $1 @ [$2] }
  | FunDec                            { [$1] }
;
```

Listing 3: More of the grammar for our Parser.grm. (continued from listing 2)

```
FunDec :
    TFunction TId TLParen PDecl TRParen TColon Type Block TSemi
                                    { AbSyn.Func($7, #1 $2, $4, $8, $1) }
  | TProcedure TId TLParen PDecl TRParen Block TSemi
                                    { AbSyn.Proc(#1 $2, $4, $6, $1) }
;

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

SBlock :
    TBegin StmtSeq TSemi TEnd       { $2 }
  | Stmt                            { [$1] }
;

StmtSeq :
    StmtSeq TSemi Stmt              { $1 @ [$3] }
  | Stmt                            { [$1] }
;

/* STATEMENTS */
Stmt :
    TId TLParen CallParams TRParen { AbSyn.ProcCall (#1 $1, $3, #2 $1) }
  | TIf Exp TThen Block TElse Block{ AbSyn.IfThEl ($2, $4, $6, $1) }
  | TIf Exp TThen Block %prec LowPrec
                                    { AbSyn.IfThEl ($2, $4,
                                                    AbSyn.Block([],[]),
                                                    $1)}
  /* prec gives precedence as LowPrec */
  | TWhile Exp TDo Block            { AbSyn.While ($2, $4, $1) }
  | TReturn Ret                     { AbSyn.Return ($2, $1) }
  | LVal TAssign Exp                { AbSyn.Assign ($1, $3, $2) }
;
```

Listing 4: More of the grammar for our Parser.grm. (continued from listing 3)

```
/* L-VALUES AND EXPRESSIONS */
LVal :
    TId                                  { AbSyn.Var (#1 $1) }
  | TId TLBracket Exps TRBracket         { AbSyn.Index (#1 $1, $3) }
;

Ret :
    Exp                                  { SOME $1 }
  |                                      { NONE }
;

Exp :
    TNLit                                { AbSyn.Literal (AbSyn.BVal(
                                           AbSyn.Num(#1 $1)), #2 $1) }
  | TBLit                                { AbSyn.Literal (AbSyn.BVal(
                                           AbSyn.Log(#1 $1)), #2 $1) }
  | TCLit                                { AbSyn.Literal (AbSyn.BVal(
                                           AbSyn.Chr(#1 $1)), #2 $1) }
  | TSLit                                { AbSyn.StrLit (#1 $1, #2 $1) }
  | TLCurly Exps TRCurly                 { AbSyn.ArrLit ($2,$1) }
  | LVal                                 { AbSyn.LValue (#1 $1, #2 $1) }
  | TNot Exp                             { AbSyn.Not ($2, $1) }
  | Exp TPlus Exp                        { AbSyn.Plus ($1, $3, $2) }
  | Exp TMinus Exp                       { AbSyn.Minus ($1, $3, $2) }
  | Exp TTimes Exp                       { AbSyn.Times ($1, $3, $2) }
  | Exp TSlash Exp                       { AbSyn.Div ($1, $3, $2) }
  | Exp TEq Exp                          { AbSyn.Equal ($1, $3, $2) }
  | Exp TLess Exp                        { AbSyn.Less ($1, $3, $2) }
  | Exp TAnd Exp                         { AbSyn.And ($1, $3, $2) }
  | Exp TOr Exp                          { AbSyn.Or ($1, $3, $2) }
  | TLParen Exp TRParen                  { $2 }
  | TId TLParen CallParams TRParen       { AbSyn.FunApp (#1 $1, $3, # 2$1) }
;

/* VARIABLE AND PARAMETER DECLARATIONS, TYPES */
PDecl :
    Params                               { $1 }
  |                                      { [] } ;
```

Listing 5: The last of the grammar for our Parser.grm. (continued from listing 4)

```
Params :
    Params TSemi Dec                { $1 @ [$3] }
  | Dec                             { [$1] }
;

Dec :
    TId TColon Type                 { AbSyn.Dec (#1 $1, $3, #2 $1) }
;

Decs :
    Decs Dec TSemi                  { $1 @ [$2] }
  | Dec TSemi                       { [$1] }
;

Type :
    TInt                            { AbSyn.Int ($1) }
  | TChar                           { AbSyn.Char ($1) }
  | TBool                           { AbSyn.Bool ($1) }
  | TArray TOf Type                 { AbSyn.Array ($3,$1) }
;

/* FUNCTION AND PROCEDURE PARAMETERS AND INDEX LISTS */
CallParams :
    Exps                            { $1 }
  |                                 { [] }
;

Exps :
    Exp TComma Exps                 { $1 :: $3 }
  | Exp                             { [$1] }

%%

(* SML trailer *)
```

Appendix B. Driver.sml

Listing 6: The Driver.sml changes for our parser.grm.

```
fun compile arg path =
  let
    val inpath = path
    val outpath= Path.base path ^ ".asm"
    val lexbuf = createLexerStream (BasicIO.open_in inpath)
  in
    let
      (*val pgm = LL1Parser.parse Lexer.Token lexbuf*)
      (* COMMENT LINE ABOVE AND UNCOMMENT  *)
      (* THE LINE BELOW TO USE YOUR PARSER *)
      val pgm = Parser.Program Lexer.Token lexbuf
    in case arg of
      "-ti" => typedInterpret (typeCheck pgm)
    | "-c"  => compileNormal pgm outpath
    | other => print ("'" ^ other ^ "': Unknown mode of operation.\n")
    end
    handle
      Parsing.yyexit ob => errorMsg "Parser-exit\n"
    | Parsing.ParseError ob =>
       (* errorMsgAt "Parsing error" (Lexer.getPos lexbuf) *)
       (* COMMENT LINE ABOVE AND UNCOMMENT  *)
       (* THE LINE BELOW TO USE YOUR PARSER *)
         errorMsgAt "Parsing error" (Lexer.getPos lexbuf)

   (* | Parser.ParseError s =>
         errorMsgAt ("Parse error: " ^ s) (Lexer.getPos lexbuf) *)
```

Appendix C. TpAbSyn.sml

Listing 7: TpAbSyn.sml with the functionality of the four operations implemented for task 2.

```
| 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 (Or    (e1, e2, _))  = "( " ^ pp_exp e1 ^ " | " ^
                                 pp_exp e2 ^ " )"
| pp_exp (Not   (e1,     _))  = "( not " ^ pp_exp e1 ^ " )"
| pp_exp (Times (e1, e2, _))  = "( " ^ pp_exp e1 ^ " * " ^
                                 pp_exp e2 ^ " )"
| pp_exp (Div   (e1, e2, _))  = "( " ^ pp_exp e1 ^ " / " ^
                                 pp_exp e2 ^ " )"

| typeOfExp ( Or    (_,_,_) ) = BType Bool
| typeOfExp ( Not   (_,  _) ) = BType Bool
| typeOfExp ( Times (a,b,_) ) = typeOfExp a
| typeOfExp ( Div   (a,b,_) ) = typeOfExp a

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

Appendix D. TpInterpret.sml

Listing 8: TpInterpret.sml changes for task 2.

```
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 )

  | 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 ( 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 ( 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
```

Listing 9: TpInterpreter changes for task 5.

```
let val new_vtab = bindTypeIds(fargs, aargs, fid, pdcl, pcall)
    val res  = execBlock( body, new_vtab, ftab )
in  ( case (rtp, res) of
        (NONE , _) =>
        let
            fun call_args [] [] = NONE
              | call_args (x::xs) (y::ys) =
                  let
                    val hitler = updateOuterVtable vtab new_vtab (x, y)
                  in
                    call_args xs ys
                  end
              | call_args _ _ = raise Error("Number of functions args" ^
                                            " does not match " ^
                                            "declaration, at ", pdcl)
        in
            call_args aexps fargs
        end

and updateOuterVtable vtabOuter vtabInner (TpAbSyn.LValue (lval1, pos1),
                                           TpAbSyn.Dec ((id2,tp), pos2))
                                                                    =
    let val lenin = (case lval1 of
                       Var(id,tp) => id
                     | Index((id,tp),e) => id)
    in
        case (SymTab.lookup lenin vtabOuter,
              SymTab.lookup id2 vtabInner) of
             (SOME x, SOME y) => x := !y
           | _                => raise Error("Procedure argument " ^
                                             "not in caller", pos1)
    end
 | updateOuterVtable _ _ _ = ()
```

Appendix E. Type.sml

Listing 10: Changes made in type.sml for task 2 and task 3.

```
(* Must be modified to complete task 3 *)
| 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 minus exp, one argument " ^
                        "is not of int type "^ pp_type tp1^
                        " and "^pp_type tp2^" at ", pos)
   end

(* Must be modified to complete task 3 *)
| 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 and exp, one argument is " ^
                         "not of bool type "^
                           pp_type tp1^" and "^pp_type tp2^" at ", pos)
    end

(* Must be modified to complete task 3 *)
| 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 and exp, one argument is not " ^
                         "of bool type " ^
                           pp_type tp1^" at ", pos)
    end
```

Listing 11: Changes in type.sml for task 3. (continued from listing10)

```
| typeCheckExp ( vtab, AbSyn.Less (e1, e2, pos), _ ) =
    let val e1_new = typeCheckExp(vtab, e1, UnknownType )
        val e2_new = typeCheckExp(vtab, e2, KnownType (typeOfExp e1_new)
        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.Equal(e1, e2, pos), _ ) =
  let val e1_new = typeCheckExp(vtab, e1, UnknownType)
      val e2_new = typeCheckExp(vtab, e2,
                                KnownType (typeOfExp e1_new) )
      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 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
```

Listing 12: changes in type.sml for task 3 and 4.

```
(* 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
            val typedargs = map (fn n =>
                      typeCheckExp(vtab, n,KnownType (BType Int))) args
            val types = map typeOfExp typedargs
            val rtp = Array ( length args, btp)
          in
            if List.all (fn n => typesEqual(BType Int, n)) types
            then
              FunApp(("new", (types, SOME rtp)), typedargs,pos)
            else
              raise Error("declared array dimensions are not " ^
                          "integers, at ", pos)
          end

| typeCheckExp( vtab, AbSyn.LValue( AbSyn.Index(id, inds), pos ), _ ) =
    let
        val newinds = map (fn n=>
                        typeCheckExp(vtab,n,KnownType (BType Int))) inds
        val correctinds = List.all (fn n =>
                        typesEqual(BType Int, typeOfExp n)) newinds
    in
        case SymTab.lookup id vtab of
          SOME (Array(r,t)) =>
            if r = length inds andalso correctinds
            then
              LValue(Index((id,Array(r,t)),newinds),pos)
            else
              raise Error("ill-formed array indexing at ",pos)
         | _ => raise Error("in type check variable, var "
                          ^id^"not in VTab, at",pos)
    end
```

Appendix F. Compiler.sml

Listing 13: The compiler.sml changes for task 2.

```
| compileExp( vtable, Times (e1, e2, _), 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, _), 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, _), place ) =
    let val t1 = "or1_" ^ newName()
        val c1 = compileExp(vtable, e1, t1)
        val t2 = "or2_" ^ newName()
        val c2 = compileExp(vtable, e2, t2)
        val lA = "_or_" ^ newName()
    in
        c1 @ c2 @ [Mips.OR (place,t1,t2)]
    end

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

Listing 14: Compiler.sml for Task 4.

```
| compileLVal( vtab : VTab, Index ((n,t),inds) : LVAL, pos : Pos ) =
    ( case SymTab.lookup n vtab of
        SOME mem =>
          let
            val rank = length inds
            val strides = 4 * (rank - 1)

            (* Variables for generated MIPS code *)
            val arrptr = "_arrptr_" ^ newName()
                (* pointer to a *)
            val ind_reg = "_indx_" ^ newName()
                (* current index i_k *)
            val dim_reg = "_dimx_" ^ newName()
                (* current dimension d_k *)
            val flat = "_flatIndx_" ^ newName()
                (* flat index value *)
            val ctr = "_ctr_" ^ newName()
                (* loop counter (init to rank) *)
            val tmp = "_tmp_" ^ newName()
                (* tmp register to check bounds *)
            val calc_name = "_calc_and_check_" ^ newName()
                (* label name for loop *)
            val str_sz = "_strides_" ^newName()
                (* Number of strides (to skip) *)

            (* Generates code for adding the indices to the stack *)
            fun copy_to_stack ([],code,n,l) =
                          (Mips.ADDI(SP,SP, makeConst(~4*l)))::code
              | copy_to_stack (i::inds,code,n,l) =
                let
                  val temp = "_temp_"^newName()
                in
                  copy_to_stack(inds,
                    code @ (compileExp(vtab, i, temp)) @
                    [Mips.ADD (ind_reg, "0", temp),
                     Mips.SW (ind_reg, SP, makeConst n)],
                  n+4,l)
                end
```

Listing 15: Compiler.sml task 4. (continued from listing 14)

```
(* If basic type size is k, returns log2(k)
   (easier to calc later) *)
fun element_size w = (case w of
    (Array(a,Int)) => "2"
  | (Array(a,Bool)) => "0"
  | (Array(a,Char)) => "0"
  | _ => raise Error("Impossible!!!, ", pos))
(* Initiates variables *)
val init_code =
    [Mips.ADDI (arrptr, mem, "0"),
     Mips.ADDI (ind_reg, "0", "0"),
     Mips.ADDI (flat, "0", "0"),
     Mips.ADDI (ctr, "0", makeConst rank),
     Mips.ADDI (str_sz, "0", makeConst strides)]

(* Checks if array index is out of bounds *)
val calc_out_of_bounds =
    [Mips.SUB (tmp, dim_reg, ind_reg),
     Mips.SLT (dim_reg, dim_reg, tmp),
     Mips.SLTI (tmp, tmp, "1"),
     Mips.OR (tmp, tmp, dim_reg),
     Mips.BNE (tmp, "0", "_IllegalArrIndexError_")]
```

Listing 16: Compiler.sml task 4. (continued from listing 15)

```
    (* Loop: checks if each index is within bounds
        and calculates flat index *)
    val calc_and_check =
        [Mips.LABEL (calc_name),
         Mips.LW (dim_reg, arrptr, "0"),
             (* loads current dim *)
         Mips.MUL (flat, flat, dim_reg),
         Mips.LW (ind_reg, SP, "0")]
             (* loads current index *)
       @ calc_out_of_bounds @
        [Mips.ADD (flat, flat, ind_reg),
         Mips.ADDI (arrptr, arrptr, "4"),
             (* points to next index *)
         Mips.ADDI (SP, SP, "4"),
             (* points to next dim *)
         Mips.ADDI (ctr, ctr, "-1"),
         Mips.BNE (ctr, "0", calc_name)]

    (* Calculates the address of the array index *)
    val get_address =
        [Mips.ADD(arrptr, arrptr, str_sz),
              (* skip the strides *)
         Mips.LW(arrptr,arrptr,"0"),
         Mips.SLL (flat, flat, element_size t),
         Mips.ADD (arrptr, arrptr, flat)]
in
   (copy_to_stack(inds,[],0,rank) @
   init_code @
   calc_and_check @
   get_address,
   Mem arrptr)
 end
```

Listing 17: Compiler.sml task 5

```
(* Swap temporary registers with caller registers, after
   the procedure has finished *)
and popArgs (TpAbSyn.LValue(lval, pos)::es) vtable
              reg ((Mips.ORI(rd, rs, v))::ts) =
      let
          val code = popArgs es vtable (reg+1) ts
      in
          code @ [Mips.MOVE (rs, makeConst reg)]
      end
  | popArgs (e::es) vtable reg (t::ts) = popArgs (e::es) vtable reg ts
(* if expression has more than 1 mips command, we want to pass the same
 expression along again. *)
  | popArgs _ vtable reg [] = []
  | popArgs [] vtable reg _ = []

| ProcCall ((n,_), es, p) =>
  let
      val (mvcode, maxreg) = putArgs es vtable minReg
      val prod_codes = popArgs es vtable minReg mvcode
      val new_mvcode = mvcode
          @ [Mips.JAL (n, List.tabulate (maxreg, fn reg => makeConst reg))]
          @ prod_codes
  in
      new_mvcode
  end

val new_argcode =
    if isProc andalso (not (fname = "main"))
    then map (fn (vname, reg) => Mips.MOVE (reg, vname)) movePairs
    else []
val body = compileStmts block vtable (fname ^ "_exit")
val (body1, _, maxr, spilled) = (* call register allocator *)
   RegAlloc.registerAlloc ( argcode @ body @ new_argcode )
```

Appendix G. Testing

Listing 18: Two tests for task 1 one for dangling else and one for general functionality.

```
// Test for dangling else
program dangling;

procedure main()
var i : int;
begin
   i := 0;
   if false then if true then i := 1 else i := 2;
   if i = 0 then write("passed.\n") else write("failed.\n");
end;

// General test for task 1
program test2;

function test() : bool
var m : int;
    o : int;
    v : int;
    p : int;
begin
    m := 2;
    o := 3;
    v := (4+m)*6100;
    p := 1+1*o*2;
    return v = 36600 and p = 7;
end;

procedure main()
begin
    if test() then
        write("It works")
    else
        write("fail");
end;
```

Listing 19: tests for task 2

```
program precedence;

procedure main()
var
    a : int;
    b : int;
    c : int;
    d : int;
    x : bool;
    y : bool;
    z : bool;
    q : bool;
begin
    a := 10;
    b := 5;
    c := 2;
    x := true;
    y := false;
    z := false;

    q := y = z or not x = y and x = z;
    write(q = true);

    q := x = z or not x = y and x = z;
    write(q = false);

    d := a + b * b - a / c;
    write(d = 30);

end;
```

Listing 20: tests for task 3

```
program task3readtest;

procedure main()
var
    x : int;
    y : bool;
begin
    // The first input should be an integer, hence chr()
    // and the second input a char.
    // for example 1. input: 100, 2. input: d, output = 1 (true)
    y := chr(read()) = read();
    write(y);
    // the input for read should be of type integer
    x := 3 + read();
    write(x);
    // the input for read should be of type integer
    y := 3 = read();
    write(y);
    // the input for read should be a char
    y := 'a' = read();
    write(y);
    // the input for read should be an integer
    y := 3 < read();
    write(y);
    // the input for read should be of type bool
    y := false or read();
    write(y);
    // the input for read should be of type bool
    y := true and read();
    write(y);
    // the input for read should be of type bool
    y := not read();
    write(y);
end;
```

Listing 21: tests for task 4

```
program Inference;

procedure main()
var a : array of int;
    b : array of bool;
    c : array of char;
    d : array of array of array of int;
    e : array of array of array of array of array of array of int;
    v : bool;
begin
   a := {8, 0, 0, 8, 1, 3, 5};
   b := {true, true, true, false, false, true, true, true};
   c := {'w','o','w','s','u','c','h','a','r','r','a','y'};
   d := {
         {
           {1, 2}, {3, 4}
         },
         {
           {5, 6}, {7, 8}
         }
        };
   e := new(4,6,7,4,6,9);
   a[4] := 2;
   b[3] := true;
   c[1] := 'O';
   d[1, 1, 1] := 42;
   d[0, 0, 0] := a[0];
   e[2,2,2,2,2,2] := 5;
   write(e[2,2,2,2,2,2] = 5);
   write(d[0,0,0] = 8);
   write(a[4] = 2);
   e[2,2,2,2,2,2] := d[1,1,1];
   write(e[2,2,2,2,2,2] = 42);
   write(d[1,0,1] = 6);

end;
```

Listing 22: tests for task 5 (please note this spands two pages)

```
program proctest;

procedure f(a : int;
            b : int)
begin
    a := a + 1;
    b := b + 4;
end;

procedure g(a : int;
            b : int)
begin
    a := a + 2;
    b := b + 3;
end;

procedure h(a : int;
            b : int)
begin
    f(a, b);
    a := a + 1;
    b := b + 1;
end;

procedure j(a : array of array of int;
            b : array of array of int)
begin
    a[0, 0] := 2;
    b[0, 0] := 3;
end;

function k(a : int;
           b : int) : int
begin
    a := a + 10;
    b := b + 5;
    return b;
end;
```

```
procedure main()
var
    x : int;
    y : int;
    z : array of array of int;
    q : array of array of int;
    r : int;
begin
   x := 2;
   y := 3;
   f(x, y);
   write(x = 3);
   write(y = 7);
   g(x, y);
   write(x = 5);
   write(y = 10);
   h(x, y);
   write(x = 7);
   write(y = 15);

   z := new(2, 2);
   q := new(2, 2);

   z[0, 0] := 1;
   q[0, 0] := 4;

   j(z, q);

   write(z[0,0] = 2);
   write(q[0,0] = 3);

   r := k(z[0,0], q[0,0]);

   write(r = 8);
   write(z[0,0] = 2);

   f(1, 1);
end;
```

Listing 23: Second test for procedure calls in task 5 including constants

```
program proctest2;

procedure f(d : int; e : int; f : int;)
begin
    d := 3;
    e := 4;
    f := 5;
end;

procedure main()
var
    s : int;
    t : int;
    u : int;
begin
   s := 1;
   t := 1;
   u := 1;

   f(s,3+4,u);

   write(s = 3);
   write(u = 5);

end;
```