Compiler Exam 2014 - Task set 2

Nikolaj Dybdahl Rathcke (rfq695)

January 17, 2014

Contents

1 Exam

1.1 Task 1 - Grammar transformation for LL(1)

We have the following grammar $G_{T_{UP}}$ with the 4 terminals: id ( ) ,

$$\begin{aligned} S &\rightarrow id \\ S &\rightarrow (\ T\ ) \\ T &\rightarrow S\ ,\ T \\ T &\rightarrow S \end{aligned}$$

for tuple expressions for LL(1) parsing.

1.1.1 a

Give a short reason why this grammar is not LL(1), and transform the grammar (using a well-known transformation) to obtain a grammar $G'_{T_{UP}}$ suitable for LL(1) parsing.

To make the grammer LL(1), we want to follow Algorithm 2.12.3[1]

1. Eliminate ambiguity
2. Eliminate left-recursion
3. Perform left factorisation where required

Steps 4-6 are covered in the following subtasks. Since the two productions for T begins with the same symbol, this means they have overlapping $FIRST$ sets and is not LL(1). Therefore we need to left-factor T by making a simple production with the same common prefix, S and creating another nonterminal, so we get the following grammar $G'_{T_{UP}}$

$$\begin{aligned} S &\rightarrow id \\ S &\rightarrow (\ T\ ) \\ T &\rightarrow S\ R \\ R &\rightarrow ,\ T \\ R &\rightarrow \varepsilon \end{aligned}$$

[1]Torben Ægidius Mogensen, Introduction to Compiler Design, page 69

1.1.2 b

For $G'_{T_{UP}}$, determine *FIRST* sets for all right-hand sides.

We calculate these through fixed-point iteration using the rules from **Algorithm 2.5**[2].

Right-hand side	Initialisation	Iteration 1	Iteration 2	Iteration 3
id	$\emptyset$	{id}	{id}	{id}
(T)	$\emptyset$	{ (}	{ (}	{ (}
S R	$\emptyset$	$\emptyset$	{id, (}	{id, (}
, T	$\emptyset$	{ , }	{ , }	{ , }
ε	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

And then we reach a fixed point.
So we have the following *FIRST* sets

$$\begin{aligned} FIRST(id) &= \{id\} \\ FIRST(\ (\ T\ )\ ) &= \{\ (\ \} \\ FIRST(S\ R) &= \{id,\ (\ \} \\ FIRST(,\ T) &= \{\ ,\ \} \\ FIRST(\varepsilon) &= \emptyset \end{aligned}$$

1.1.3 c

Add a start production $S' \to S\$$ and determine *FOLLOW* sets for all non-terminals.

This yields the grammar

$$\begin{aligned} S' &\to S\$ \\ S &\to id \\ S &\to (\ T\ ) \\ T &\to S\ R \\ R &\to ,\ T \\ R &\to \varepsilon \end{aligned}$$

[2]Torben Ægidius Mogensen, Introduction to Compiler Design, page 55

To calculate the *FOLLOW* sets, we follow an algorithm[3] and get the following constraints

For the production $S' \to S\$$ we add the following constraint since '\$' is seen as a terminal and is therefore the *FIRST* set of what follows S but '\$' is not nullable.

$$\{\$\} \subseteq \{S\}$$

For the production $S \to id$ there are no nonterminals, so no constraints are added.

For the production $S \to (\ T\ )$ we add the following constraint since ')' is the *FIRST* set of what follows T but is not nullable.

$$\{\ )\ \} \subseteq \{T\}$$

For the production $T \to S\,R$ we make a split for each occurence. Looking at the nonterminal S we get the constraint

$$\{\ ,\ \} \subseteq \{S\}$$

since it is the *FIRST* set of R. Furthermore, since R is nullable, we add the following constraint

$$\{T\} \subseteq \{S\}$$

Now looking at R there is nothing in the *FIRST* set, so no constraint is added. However, that nothing follows means that it is nullable, so we add the constraint

$$\{T\} \subseteq \{R\}$$

For the production $R \to, T$ we get nothing in the *FIRST* set for what follows T meaning it is the same case as above so we add the following constraint

$$\{R\} \subseteq \{T\}$$

For the last production $R \to \varepsilon$ there are no nonterminals, so no constraints are added.

[3]Torben Ægidius Mogensen, Introduction to Compiler Design, page 59

Now we need to solve these constraints. For each constraint on the form *terminal* $\subseteq$ *nonterminal* we put those into $FOLLOW(nonterminal)$ so we get

$$\begin{aligned} FOLLOW(S) &= \{\$, ',' \} \\ FOLLOW(T) &= \{ \,) \} \\ FOLLOW(R) &= \emptyset \end{aligned}$$

Now for each constraint on the form *nonterminal* $\subseteq$ *nonterminal* we add the content from the *FOLLOW* set of the first nonterminal to the second until a fixed point is reached and we get

$$\begin{aligned} FOLLOW(S) &= \{\$, ',' , \,)\} \\ FOLLOW(T) &= \{ \,) \} \\ FOLLOW(R) &= \{ \,) \} \end{aligned}$$

Whereas a fixed point is reached and we have our *FOLLOW* sets.

1.1.4 d

Determine the look-ahead sets for all productions and point out that $G'_{T_{UP}}$ is LL(1).

Having calculated *FIRST* and *FOLLOW* we can use the following rules[4] to determine look-ahead (LA) sets.

$$LA(X \to \alpha) = \begin{cases} FIRST(\alpha) \cup FOLLOW(X) & \text{if NULLABLE(X)} \\ FIRST(\alpha) & \text{otherwise} \end{cases}$$

So we get the following look ahead sets (disregarding the added start production)

$$LA(S \to id) = \{id\}$$

Since the right hand side is not nullable.

$$LA(S \to (\ T\ )) = \{\ (\ \}$$

[4]From lecture slides

Since the right hand side is not nullable.

$$LA(T \rightarrow S\ R) = \{id,\ (\}$$

Since the right hand side is not nullable.

$$LA(R \rightarrow ,\ T) = \{\ ,\ \}$$

Since the right hand side is not nullable.

$$LA(R \rightarrow \varepsilon) = \{\ )\ \}$$

Since the right hand side is nullable.

Since for each nonterminal X in the grammar, the look ahead sets for each production for this nonterminal are disjoint, it means the grammar is LL(1).

1.2 Task 2 - Extend the equality operation to work on arrays

1.2.1 Major Changes

Changes were done to the "Compiler.sml" and "Type.sml".

Type: Nothing was added to this file, but the following part was removed

```
| typeCheckExp ( vtab, AbSyn.Equal(e1, e2, pos), _ ) =
  let val e1_try = SOME ( typeCheckExp (vtab, e1, UnknownType) ) handle _ => NONE
      val e2_try = SOME ( typeCheckExp (vtab, e2, UnknownType) ) handle _ => NONE
      val (e1_new, e2_new) =
          case (e1_try, e2_try) of
          (SOME a, SOME b) => (a, b)
        | (SOME a, NONE  ) => (a, typeCheckExp (vtab, e2, KnownType(typeOfExp a)))
        | (NONE  , SOME b) => (typeCheckExp (vtab, e1, KnownType(typeOfExp b)), b)
        | (NONE  , NONE  ) => raise Error("in type check equal, neither operand" ^
                                        "type-checks (possibly polymorphism)," ^
                                        " at ", pos)

      val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)

      val _ = typesEqual(tp1, tp2) orelse
              raise Error("in type check equal, argument types do not match, "^
                          pp_type tp1^" isn't "^pp_type tp2^" at ", pos)

#     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 Equal(e1_new, e2_new, pos) end
```

Since this was the restraint in type-checking for arrays for the equality operation. The code above would check for both type and rank and raise and error if these did not match, leaving only shape and elements to be compared.

Compiler: Only adjustments to the function `compileExp` for `equal` were made. The following checks if the first argument is an array and creates symbolic registers that will be used in the code.

```
| compileExp( vtable, Equal(e1, e2, _), place ) =
(* Check if arguments are arrays *)
    let val tp1 = typeOfExp e1
        val isArray =
              case tp1 of
              Array _ => true
              | _ => false
        val t1 = "eq1_" ^ newName()
        val c1 = compileExp(vtable, e1, t1)
        val t2 = "eq2_" ^ newName()
        val c2 = compileExp(vtable, e2, t2)
        val lEq = "_equal_" ^ newName()
```

Since if this is not the case it will just proceed with the Mips code that compares 2 elements with eachother. However if this is not the case and the expression is an array

```
(* If arguments are arrays, get rank and type and initiate appropiate 'variables' *)
    in if isArray
       then
       let val (rnk,btp) =
               case tp1 of
               Array(d, b) => (d,b)
               | _ => raise Error2("Something wrong with array input")
           val endOf    = "_endOf_" ^ newName()
           val nextElm = "_nextElm_" ^ newName()
           val ctlDgt  = "_ctlDgt_" ^ newName()
           val chk_shp = "_chk_shp_" ^ newName()
           val chk_elm = "_chk_elm_" ^ newName()
           val shpOk   = "_shpOk_" ^ newName()
           val elmOk   = "_elmOk_" ^ newName()
           val t3      = "_t3_" ^ newName()
           val t4      = "_t4_" ^ newName()
           val t5      = "_t5_" ^ newName()
```

The rank and type (which are checked in Type.sml) is taken out. The rank is to determine how many dimensions to check and to get the pointer to the data. The type is to determine how many bytes to jump when checking each element. An error2 (which is an error created that is just a string) is called if it doesn't match. The Mips code below is to determine if the shapes are the same

```
       in
(* MIPS code to check if shape is the same for the arrays *)
```

```
        c1 @ c2 @ (case btp of
                   Int => [ Mips.ADDI (nextElm, "0", makeConst 4)]
                   | _ => [ Mips.ADDI (nextElm, "0", makeConst 1)] ) @
       [ Mips.LI (place, "0"),
         Mips.ADDI (endOf, "0", "0"),
         Mips.ADDI (t5, "0", "1"),
         Mips.ADDI (endOf, endOf, makeConst rnk),
         Mips.SLL (endOf, endOf, makeConst 2),
         Mips.LABEL chk_shp,
         Mips.BEQ (endOf, "0", shpOk),
         Mips.LW (t3, t1, makeConst 0),
         Mips.LW (t4, t2, makeConst 0),
         Mips.ADDI (t1, t1, makeConst 4),
         Mips.ADDI (t2, t2, makeConst 4),
         Mips.ADDI (endOf, endOf, makeConst ~4),
         Mips.MUL (t5, t5, t3),
         Mips.BNE (t3, t4, lEq),
         Mips.J chk_shp,
         Mips.LABEL shpOk,
```

by making the result ”0” to begin with, and then running through each dimension and comparing them. It has checked all dimensions, when ”endOf” is equal to 0, since this is the rank*4. If two of the dimensions should be different it will jump to ”lEq” which is the end of the Mips code as seen in the following, where the elements are compared

```
(* MIPS code to check if data is the same in the arrays *)
         Mips.ADDI (endOf, "0", makeConst rnk),
         Mips.SLL (endOf, endOf, makeConst 2),
         Mips.ADD (t1, t1, endOf),
         Mips.ADD (t2, t2, endOf),
         Mips.MUL (endOf, t5, nextElm),
         Mips.LABEL chk_elm,
         Mips.BEQ (endOf, "0", elmOk)] @
         (case btp of
          Int => [ Mips.LW (t3, t1, makeConst 0), Mips.LW (t4, t2, makeConst 0) ]
          | _ => [ Mips.LB (t3, t1, makeConst 0), Mips.LB (t4, t2, makeConst 0) ] ) @
         [Mips.ADD (t1, t1, nextElm),
         Mips.ADD (t2, t2, nextElm),
         Mips.SUB (endOf, endOf, nextElm),
         Mips.BNE (t3, t4, lEq),
         Mips.J chk_elm,
         Mips.LABEL elmOk,
         Mips.LI (place, "1"),
         Mips.LABEL lEq]
```

```
    end
```

Which works in the same way as checking the dimension, but jumping the appropiate amount of bytes (nextElm) and using either "LW" or "LB" depending on the type. If all dimensions and elements are the same, it will be set to "1" before terminating.

1.2.2 Tests

Tests for Task 2 was done with the "ArrayEqual.pal" seen in Annex 1. The tests included creating a multi dimensional array and inserting some data (either char, bool, or ints) and seeing if it provided the correct answer, which all the in the test program does.
It was tested with data for all indexes in the multidimensional array and it was tested with some index left untouched (meaning they were zero), for which it provided the right answer.
It was also tested with a simple array for all the basic types, returning the right answer as well.

There would be raised errors if types got mixed in the arrays or if shapes did not match, as was intended. There would also be raised errors if empty arrays were created.
The program was also inspected using the MARS simulator to see if it acted as thought.
No immediate errors were found.

1.3 Task 3 - A Flat-Reduce Higher-Order Function in Paladim

1.3.1 Major Changes

Two parts had to be modified in ”Type.sml” and ”Compiler.sml” to complete the task

Type: Specifically, only
`typeCheckExp ( vtab, AbSyn.FunApp ("reduce", args, pos), etp )`
was implemented.
Seen below is, some, of the code for typechecking the function.

```
| typeCheckExp ( vtab, AbSyn.FunApp ("reduce", args, pos), etp ) =
(* Check if function name exists and returns the type list and tye option, otherwise errors are given*)
if List.length args = 2
then let val fid = (case (List.nth (args,0)) of
                    AbSyn.LValue(AbSyn.Var(fid),pos) => fid
                   | _ => raise Error ("Wrong argument at ",pos))
         val (fun_arg_tps, fun_ret_tp) =
            case SymTab.lookup fid (!functionTable) of
              NONE => raise Error("In typeCheck function call, function "^fid^" not in function table, at ", pos)
            | SOME (tps, SOME rtp) => if ( (length tps) = (length args) ) then (tps, rtp)
                                     else raise Error("In typeCheckExp function "^fid^" call, the number"^
                                                      " of formal and actual args do not match, at ", pos)
            | SOME ssig => raise Error("In typeCheck function call, function "^fid^
                                      " of signature "^showFunType ssig^" has no return type at ", pos)
```

This checks if the amount of arguments is equal to 2, otherwise it is not valid. After this, it finds a signature for the function by looking it up. This way, a type list is returned which contains the types of the arguments. Errors occur if function does not exist. To guarantee the basic type of the array matches those of the arguments in the function call, we use the following

```
    (* Returns the type of the array *)
             val arr = List.nth(args,1)
             val new_arr = typeCheckExp(vtab, arr, etp)
             val tp1 = typeOfExp new_arr
             val tp2 = case tp1 of
                       Array(_, b) => b
                      | _ => raise Error("Array in argument is not correct, at ", pos)
    (* Checks if arguments in function matches the BType of the array, otherwise errors are given *)
             val args_ok = map ( fn e => typesEqual(e, BType tp2) orelse
                                         raise Error("Arguments not of same type, at ", pos)) fun_arg_tps
```

```
          in
            Red( (fid, (fun_arg_tps, SOME fun_ret_tp)), new_arr, pos)
          end

    else raise Error("Not correct amount of arguments, at ", pos)
```

Which find the basic type of the array and then checks if the types stored in the signature matches this basictype by mapping over the Type list. The typechecked arguments are then passed on. The else statement is in case the argument array isn't of size 2.

Compiler: For this task, only `compileExp( vtable, Red( (id,signat), arr_exp, pos ), place )` was modified. Below is seen a part of the code

```
| compileExp( vtable, Red( (id,signat), arr_exp, pos ), place ) =
(* Initializing appropiate symbol register and retrieves rank and BType of the array
   to determine how far to jump to retrieve data *)
    let val tp1     = typeOfExp arr_exp
        val t1      = "_t1_" ^ newName()
        val c1      = compileExp(vtable, arr_exp, t1)
        val t2      = "_t2_" ^ newName()
        val t3      = "_t3_" ^ newName()
        val t4      = "_t4_" ^ newName()
        val getDim  = "_getDim_" ^ newName()
        val reduce  = "_reduce_" ^ newName()
        val endRed  = "_endRed_" ^ newName()
        val oneElm  = "_oneElm_" ^ newName()
        val nextElm = "_nextElm_" ^ newName()
        val (rnk,btp) = case tp1 of
                         Array(d, b) => (d,b)
                        | _ => raise Error2("Something wrong with array input")
        val pointer = "_pointer_" ^ newName()
    in
```

Which is taking the type of the array, to determine where in the memory data should be extracted from. Also all the symbolic registers are created (these were added along the way).
The first part of the generating the Mips code was this

```
    (* Mips code to retrieve size of Array by size of dimensions and have the right values for *)
          c1 @
        [ Mips.ADDI (pointer, "0", makeConst rnk),
          Mips.SLL (pointer, pointer, makeConst 2),
          Mips.ADDI (t4, "0", makeConst 1),
          Mips.LABEL getDim,
          Mips.LW (t3, t1, makeConst 0),
          Mips.MUL(t4, t4, t3),
          Mips.ADDI (t1, t1, makeConst 4),
          Mips.ADDI (pointer, pointer, makeConst ~4),
          Mips.BNE (pointer, "0", getDim) ] @
          (case btp of
             Int => [ Mips.ADDI (nextElm, "0", makeConst 4)]
             | _ => [ Mips.ADDI (nextElm, "0", makeConst 1)] ) @
        [ Mips.ADDI (pointer, "0", makeConst rnk),
          Mips.SLL (pointer, pointer, makeConst 2),
          Mips.ADD (t1, t1, pointer),
          Mips.ADD (pointer, t1, nextElm),
    (* If only 1 element in array *)
          Mips.ADDI (t2, t4, makeConst ~1),
          Mips.BEQ (t2, "0", endRed) ] @
    (* return that element *)
```

Which was to establish how many elements were in the array. This is done by looping over the dimensions and extracting this value in the symbolic register "t4" until all dimensions had been run through. After the size of the array is determined it sets the symbolic registers "t1" and "pointer" to look at the first 2 elements in the array. In case the array is of size 1, it will jump to a label "endRed" (that is described later).

```
          (case btp of
            Int => [ Mips.LW (t3, t1, makeConst 4), Mips.LW (t1, t1, makeConst 0) ]
            | _ => [ Mips.LB (t3, t1, makeConst 1), Mips.LB (t1, t1, makeConst 0) ] ) @
    (* Mips code to reduce the array *)
        [ Mips.ADDI (t4, t4, makeConst ~2),
          Mips.LABEL reduce ] @
          mkFunCallCode ((id,signat), (t1::[t3]), vtable, t1) @
        [ Mips.BEQ (t4, "0", endRed) ] @
          (case btp of
            Int => [ Mips.LW (t3, pointer, makeConst 4) ]
            | _ => [ Mips.LB (t3, pointer, makeConst 1) ] ) @
        [ Mips.ADD (pointer, pointer, nextElm),
          Mips.ADDI (t4, t4, makeConst ~1),
          Mips.J reduce,
          Mips.LABEL oneElm] @
          (case btp of
            Int => [ Mips.LW (t1, t1, makeConst 0) ]
            | _ => [ Mips.LB (t1, t1, makeConst 0) ] ) @
        [ Mips.ADDI (t4, t4, makeConst ~1),
          Mips.LABEL endRed,
          Mips.XORI (t2, t4, makeConst 1),
          Mips.BEQ (t2, "0", oneElm),
          Mips.MOVE (place, t1)]
        end
```

Then, another loop, that retrieves the value of "t1" and "pointer" and uses the function "mkFunCallCode" that returns the mips code for the function call on these values. It only retrieves 2 values the first time. This is because the result is stored "t1", which is used in every run through of the loop. So "t1" always holds the newest result. This loop is done "t4"-1 times, which is the amount of elements minus 1.
When the result needs to be returned it is first checked whether "t4" is 1 or not. Since if it is, it means that there was only 1 element in the array. If that is the case, it simply loads the value and places it in "t1". After this check the result in "t1" is returned.

Obviously the branches to check if it's a single element makes it less efficient. However, these branches are only checked for once, so they don't add much to running time. The two 'loops' are necessary to check size of all dimensions and elements, so unless there is a better way to retrieve the size of the arrays, I don't see more improvement besides maybe being able to reduce the amount of 'standard' Mips instruction.

1.3.2 Tests

Tests for Task 3 was done with "Reduce.pal" which can be seen Annex 2.
The test had 3 parts. A basic one which takes arrays of ints and adds them, thus ensuring that the function works on ints.
Then there was another one checking for booleans (which meant that the jump distance should be just one byte). This test simply took the last element in the array that, in the test, was the only "true" in the entire array.
To test the case where there was only one element in the array, a simple array of int, with one element, was created. In this case it does not generate any error, but it just returns the one element in the array since this should be the reduced version of the array.
All tests acted as intended.

1.4 Task 4 - A switch statement for Paladim

Not implemented.

1.5 Annex 1 - ArrayEqual.pal (Task 2)

```
program ArrayEqual;

procedure main()
var
  arr  : array of array of char;
  arr2 : array of array of char;
  arr3 : array of char;
  arr4 : array of char;
  arr5 : array of array of bool;
  arr6 : array of array of bool;
  arr7 : array of array of int;
  arr8 : array of array of int;
begin
  arr2 := new(1,2);
  arr2[0,0] := 'a';
  arr := new(1,2);
  arr[0,0] := 'b';
  if (arr = arr2)
  then write("Passed")
  else write("Failed\n");
// should fail
  arr2 := new(1,2);
  arr2[0,0] := 'd';
  arr2[0,1] := 'b';
  arr := new(1,2);
  arr[0,0] := 'd';
  arr[0,1] := 'b';
  if (arr = arr2)
  then write("Passed\n")
  else write("Failed");
// should pass
  arr3 := {'a','b','c'};
  arr4 := {'a','c','b'};
  if (arr3 = arr4)
  then write("Passed")
  else write("Failed\n");
```

```
// should fail
  arr3 := {'a','b','c'};
  arr4 := {'a','b','c'};
  if (arr3 = arr4)
  then write("Passed\n")
  else write("Failed");
// should pass
  arr5 := new(1,2);
  arr5[0,0] := true;
  arr5[0,1] := false;
  arr6 := new(1,2);
  arr6[0,0] := true;
  arr6[0,1] := true;
  if (arr5 = arr6)
  then write("Passed")
  else write("Failed\n");
// should fail
  arr5 := new(1,2);
  arr5[0,0] := true;
  arr5[0,1] := false;
  arr6 := new(1,2);
  arr6[0,0] := true;
  arr6[0,1] := false;
  if (arr5 = arr6)
  then write("Passed\n")
  else write("Failed");
// should pass
  arr7 := new(2,1);
  arr7[0,0] := 12;
  arr7[1,0] := 131072;
  arr8 := new(2,1);
  arr8[0,0] := 12;
  arr8[1,0] := 121054;
  if (arr7 = arr8)
  then write("Passed")
  else write("Failed\n");
// should fail
  arr7 := new(1,3);
  arr7[0,0] := 5;
  arr7[0,1] := 7;
  arr7[0,2] := 9;
  arr8 := new(1,3);
  arr8[0,0] := 5;
  arr8[0,1] := 7;
  arr8[0,2] := 9;
  if (arr7 = arr8)
  then write("Passed\n")
  else write("Failed");
// should pass
end;
```

1.6 Annex 2 - Reduce.pal (Task 3)

```
program FlatReduceExample;
function plus(x : int; y : int) : int
return (x + y);

function xor(x : bool; y : bool) : bool
return (y);

procedure main()
var x : int; a : array of array of int;
    y : bool; b : array of array of bool;
    z : int; c : array of int;
begin
a := {{2,3,4},{5,6,7}};
x := reduce(plus, a);
write(x);
b := {{false,false,false},{false,false,true}};
y := reduce(xor, b);
write(y);
c := {8};
z := reduce(plus, c);
write(z);
write("\nOutput should be 2718");
end;
```