Report for ordinary compiler exam at DIKU - Taskset 4

Jonas Brunsgaard (msn368)

January 18, 2013

Contents

Shot commings

- When running the whole test suite with the optimizer enabled, some tests still fail
- The handed out test `testSplitConcat.fo` fails due to bounds checking, which in my opinion are correct

1 Top-Down Parsing for if-then-else

In this assignment we will work our way forward to determine whether or not a grammar is LL(1). This can be determined by deriving the Look-ahead sets for the grammar.

To derive the Look-ahead sets we need to eliminate any left recursion, left-factorize the grammar, determine nullabilility, first sets and follow sets.

The procedure used in this assignment are described in the book section 2.12, With the one exception that we use Look-ahead sets instead of a parse table to determine if the grammar is LL(1). Look ahead sets are described in the slides from the parser lecture.

1.1 Left-factorization of the grammar

Below is the new grammar after left-factorization.

$$
\begin{array}{lcl}
S & \rightarrow & \text{if } BSS_* \\
S & \rightarrow & \text{return NUM;} \\
S_* & \rightarrow & \varepsilon \\
S_* & \rightarrow & \text{else } S \\
B & \rightarrow & (\,\text{NUM}\,)
\end{array}
$$

1.2 Nullablility and First sets

First we determine if terminals and nonterminals are *nullable*.

Right-hand side	Init	First Iter	Sec Iter
if $BSS*$	*false*	*false*	*false*
return NUM;	*false*	*false*	*false*
ε	*false*	*true*	*true*
else S	*false*	*false*	*false*
(NUM)	*false*	*false*	*false*
Nonterminal			
S	*false*	*false*	*false*
S_*	*false*	*true*	*true*
B	*false*	*false*	*false*

Then we use fixed point iteration to determine the $FIRST$-sets:

Right-hand side	Init	First Iter	Sec Iter
if BSS_*	$\emptyset$	{if}	{if}
return NUM;	$\emptyset$	{return}	{return}
ε	$\emptyset$	$\emptyset$	$\emptyset$
else S	$\emptyset$	{else}	{else}
(NUM)	$\emptyset$	{(}	{(}
Nonterminal			
S	$\emptyset$	{if,return}	{if,return}
S_*	$\emptyset$	{else}	{else}
B	$\emptyset$	{(}	{(}

1.3 Calculate Follow sets for all nonterminals

By following the procedure found on page 59 in the book, we derive the table below. To handle the end-of-string condition we add $S' \to S\$$ to the production.

Production	Constraints
$S' \to S\$$	$\{\$\} \subseteq FOLLOW(S)$
$S \to$ if BSS_*	$\{\text{return}, \text{if}\} \subseteq FOLLOW(B)$, $FOLLOW(S) \subseteq FOLLOW(S_*)$, $\{\text{else}\} \subseteq FOLLOW(S)$
$S \to$ return NUM;	
$S_* \to \varepsilon$	
$S_* \to$ else S	$FOLLOW(S_*) \subseteq FOLLOW(S)$
$B \to$ (NUM)	

We first use the constraints $\{\$\} \subseteq FOLLOW(S)$ and constraints of the form $FIRST(\dots) \subseteq FOLLOW(\dots)$ to get the initial sets.

$$\begin{aligned} FOLLOW(S) &\subseteq \{\text{else}, \$\} \\ FOLLOW(S_*) &\subseteq \{\emptyset\} \\ FOLLOW(B) &\subseteq \{\text{if}, \text{return}\} \end{aligned}$$

and then use the constrains on the form $FOLLOW(\dots) \subseteq FOLLOW(\dots)$:

$$\begin{aligned} FOLLOW(S) &\subseteq \{\text{else}, \$\} \\ FOLLOW(S_*) &\subseteq \{\text{else}, \$\} \\ FOLLOW(B) &\subseteq \{\text{if}, \text{return}\} \end{aligned}$$

1.4 Look-aheads sets

From the lecture slides the look ahead set is defined as

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

Below you see the lookahead sets for our productions.

$$\begin{array}{lcl} LA(S \to \text{if } BSS_*) & = & \{\text{if}\} \\ LA(S \to \text{return NUM;}) & = & \{\text{return}\} \\ LA(S_* \to \varepsilon) & = & \{\text{else,\$}\} \\ LA(S_* \to \text{else } S) & = & \{\text{else}\} \\ LA(B \to (\text{NUM})) & = & \{(\} \end{array}$$

From the course slides on parsing we know that if for each nonterminal $X \in N$ in grammar G, all productions of X have disjoint look-ahead sets, then the grammar G is LL(1). This is not the case in out grammar d to the production S_*, an thus our production is not LL(1)

1.5 Resolving the ambiguity in a recursive-decent parser

As mentioned in the assignment the established convention when dealing with the dangling else is to attach the "else" to the nearby "if" statement.

In our case - as in the book - we can achieve this by removing the empty production of S_*. In more general we have to try rewriting the ambiguous grammar to a unambiguous one.

Normally this is done by adding productions to keep track of the statement state(matched/unmatched). An example can be found in the book page 52 or at - http://www.parsifalsoft.com/ifelse.html.

Also ambiguity can be resolved adding more terminals to the grammar, e.g the terminal "fi" - as used in Bash - which indicates the end of the conditionally executed code, or we could add the more common used braces to the grammar.

2 Implementing Band (bitwise and)

2.1 Code changes to implement `band`

In this sub-assignment the bitwise `AND` operator `band` has been implemented. The first changes were made to the structure file - `Fasto.sml` - where I added the AbSyn node as part of the datatype `Exp` and implemented pretty printing functionality for the new node.

Afterwords `band` was added to the lexer by adding the keyword `band` and thereafter the parser, by adding the token `BAND` and providing it with left association and the correct precedence, as shown below.

```
...
%left PLUS MINUS NEGATE
%left BAND
%left TIMES DIV
...
```

Also in the parser `BAND` was added to the *Exp* production as an independent expression(creating a `Fasto.Band` node), as well as part of any SOAC expressions taking an operator as an argument.

Then `Fasto.Band` was added to the typechecker with the following code:

```
| Fasto.Band (e1, e2, pos) =>
   let
     val (ts, es) = ListPair.unzip (List.map (expType vs) [e1,e2])
     (* ts holds the type, we unify with int *)
     val t  = List.foldl (unifyTypes pos)(Fasto.Int pos) ts
   in
     (Fasto.Int pos, Fasto.Band (List.nth (es,0),
                                 List.nth (es,1), pos))
   end
```

The code works by creating the list `ts` holding the types of the of the two expressions applied to `band`. The types in `ts` is unified with `Int`. The function `unifyTypes` will raise an error exception if the types are not `Int`. If no error is raised a tuple is returned containing the type of `band` and the `band` expression.

Finally I added the case for `Fasto.Band` to the compiler:

```
| Fasto.Band (e1,e2,pos)=>
    let val t1 = "_band1_"^newName()
        val t2 = "_band2_"^newName()
        val code1 = compileExp e1 vtable t1
        val code2 = compileExp e2 vtable t2
    in  code1 @ code2 @
        [Mips.AND (place,t1,t2)]
    end
```

The code above recursively (by calling compileExp) determines the values of the input expressions and place them in registers `t1` and `t2`. `Mips.AND` then perform the bitwise operation on the two registers and place the result in `place`. Note that I have used only `Mips.AND`. One would be able to check, if one of the expressions were a `Fasto.Num` and then use `Mips.ANDI` and thus save a register. I did not do that to keep the code in line with the existing code base (see the `compileExp` case for for `Fasto.Plus, Fasto.Minus` etc..

Finally cases for `Fasto.Band` were implemented in `Optimization.sml` for all existing functions, ass well as the functions for for constant propagation and constant folding. Below the node for constant folding are shown. I used the MosML word-library to achieve this functionality:

```
| Fasto.Band(e1,e2,p) =>
    let
      val (s1, e1new) = ctFoldP( e1, vtab )
      val (s2, e2new) = ctFoldP( e2, vtab )
      fun bitwiseand (n1,n2) =
        let
          val w1 = Word.fromInt(n1)
          val w2 = Word.fromInt(n2)
        in
          Word.toInt(Word.andb(w1,w2))
        end
    in
      if
        (isCtVal e1new) andalso (isCtVal e2new)
      then
        (true,evalBinop(bitwiseand, e1new, e2new, p))
      else
        case (e1new, e2new) of
            ( Num(0,_), _         ) => (true,Fasto.Num(0, p))
          | ( _       , Num(0,_ ) ) => (true,Fasto.Num(0, p))
          | ( _       , _         ) => (s1 orelse s2,
                                        Fasto.Band(e1new,e2new,p)
```

2.2 Correctness of Band

To ensure correctness of `band` the follow tests has been added to the `DATA` folder. The content of the .out files have been calculated by hand.

testBand.fo
: Testing the correctness of `band` when applied to integers.

testBandprecedens.fo
: Testing the correctness of `band` in regard to precedence and associativity.

testBandArrayError.fo
: Testing the correctness of `band` when used with `op` as function argument..

testBandArrayError.fo
: Type testing with an Array as one of the arguments.

testBandBoolBothOperands.fo
: Type testing with booleans as both arguments.

testBandBoolFstOperandErr.fo
: Type testing with boolean as first argument.

testBandBoolSecondOperandErr.fo
Type testing with boolean as second argument.

testBandCharBothOperands.fo
Type testing with chars as both arguments.

testBandCharFstOperandErr.fo
Type testing with char as first argument.

testBandCharSecondOperandErr.fo
Type testing with char as second argument.

2.3 Source code

The full source code can be found in the `SRC` folder and where changes have been made there should be a comment before and after the code block. I put my exam number(msn378) in this comment so it would be easy to find by searching the files.

3 Implementation of the second order array combinators `split` and `concat`

In this sub assignment task was to implement the second-order-array-combinators `split` and `concat` into the Fasto language and the compiler.

3.1 Code changes

I started by adding the AbSyn nodes for `split` and `concat` to the structure in `Fasto.sml` as described in the handed out exam assignment. Also pretty printing functionality were implemented here.

3.1.1 Lexer

As specified in the exam paper, the tokens `split` and `concat` were added to the list of keywords in the lexer with the following code:

```
| "split"        => Parser.SPLIT pos
| "concat"       => Parser.CONCAT pos
```

3.1.2 Parser

In the parser `split` and `concat` were added to the right hand side of the *Exp* production.

```
| SPLIT LPAR Exp COMMA Exp RPAR
                { Fasto.Split ($3, $5, Fasto.UNKNOWN, $1) }
| CONCAT LPAR Exp COMMA Exp RPAR
                { Fasto.Concat ($3, $5, Fasto.UNKNOWN, $1) }
```

Note, at this point we do not know the type of the elements in the input array. Therefore we set the type in the to `Fasto.UNKNOWN`. The type is determined in the type checking phase.

3.1.3 Type checker

For `split` the following case was added to the type checker. The full code sample is added due to the it informative comments.

```
| Fasto.Split (num, arr, typ, pos)
  => let
       (* type of first argument, should be Int *)
      val (n_type, n_dec) = expType vs num
      (* type of second argument, should be Array *)
      val (arr_type, arr_dec) = expType vs arr
      (* type of elements in Array, if not Array an
       * exception is thrown *)
      val el_type = case arr_type of
            (* typ is Unknown, we unify with the types and
            * determine the elements types of the Array..
            * Unifying are not stricly necessary here, but
            * we do it to keep consistency in the code eg.
            * Reduce also uses this practice,
            * alternativly we could just return t
            * the first tuple element in Array *)
            Fasto.Array (t,_) => unifyTypes pos (typ, t)
          | _ => raise Error ("Split: Second argument"^
                              "not an array",pos)
     in
       (* check if first argument is type Int *)
       if
         typesEqual (n_type, Fasto.Int pos)
       then
         (Fasto.Array (arr_type, pos),
          Fasto.Split (num ,arr_dec, el_type, pos))
       else
         raise Error ("Split: First argument type is not Int " ^
                      showType n_type, pos)
     end
```

With the comments hopefully it is easy to determine how the case is handled by the type checker.

For the type checking to be successful the first expression must have the type `Fasto.Int`, the second expression must be of type `Fasto.Array`. The type of the elements in the array is not important to whether the type checking succeed. The elements could actually be `Fasto.UNKNOWN` and still pass type checking. This behavior may not be intentional and could be fixed

by a different ordering of the cases in the function `unifyTypes`, which at this point is:

```
fun unifyTypes pos (Fasto.UNKNOWN, t) =
  | unifyTypes pos (t, Fasto.UNKNOWN) =
  | unifyTypes pos (t1, t2) = (* typesEqual where
                                 unknown and unknown
                                 return false. but
                                 unifyTypes has already
                                 returned due to case
                                 ordering *)
```

For `concat` the following case was added to the type checker. The full code sample is again added below due to the comments explaining how it works.

```
| Fasto.Concat (arr1, arr2, arg1_typ, pos)
  => let
      (* type of first argument, should be Array *)
      val (arr1_type, arr1_dec) = expType vs arr1
      (* type of second argument, should be Array *)
      val (arr2_type, arr2_dec) = expType vs arr2
      (* type of elements in Array, if not Array and exception is thrown
          *)
      val el1_type = case arr1_type of
            Fasto.Array (t,_) => unifyTypes pos (arg1_typ, t)
          | other => raise Error ("Concat: First Array Argument "^
                                  "not an array",pos)
      val el2_type = case arr2_type of
            Fasto.Array (t,_) => unifyTypes pos (arg1_typ, t)
          | other => raise Error ("Concat: Second Array Argument "^
                                  "not an array",pos)
      val el_type_unified = unifyTypes pos (el2_type, el1_type)
     (* An alternative solution would be to check with typesEqual, this
      * way we could have used a if-then-else and raised a more specific
      * error. *)
     in
       (Fasto.Array (el_type_unified, pos), Fasto.Concat(arr1_dec,
       arr2_dec, el_type_unified, pos))
     end
```

First we find the types with the function `expType` of the two input expression (we hope these are arrays, therefore the naming `arr1` and `arr2` of the input parameters).

Then we pattern match on the types of the expressions `arr1` and `arr2`. If the pattern do not match the type `Fasto.Array` we raise the exception Error and inform the user with the error message.

Now we have the element types of the two arrays, thus we are able to unify these and thus check equality.

If no exception if thrown in the call to `unifyTypes`, then we can safely proceed and return a tuple holding the type of the `concat` expression and a updated `concat` expression.

3.2 Compiler

For `split` a case was added to compileExp. Below are shown some pseudo code that roughly describes the idea of the implementation.

```
split(int n, int[] lst, type eltp);
{
int *data1_adr;
int *data2_adr;

// Check bounds
if (n-1 >= 0){ THROW( E)}
if (n-len(lst) >= 0){ THROW( E)}

// Get adresses where data is already stored
data1_adr = lst[0];
data2_adr = sizeof(elty) * n + data1_adr

// Create tow new inner array (only with header information)
int arr1[1]
int arr2[1]

// Create the outer array
int arr2[1]

// Set number of elements
arr1[0] = n
arr2[0] = data2_adr-data1_adr

// Copy over adresses for existing data
arr1[1] = data1_adr
arr2[1] = data2_adr

// The outer Array has two elements, arr1 and arr2.
arr3[0] = 2
arr3[1] = &[arr1,arr2]
}
```

By this implementation we avoid allocation of memory for data already placed in memory. And thus, we only need to allocate new "data entries" in the memory for the outer array holding the two new inner arrays, but due to the way arrays work in Fasto, we still have to allocate two words of memory for every new array to hold the header information.

In theory we could avoid allocation any new memory by placing the data in the registers. But it seams like a bad solution because registers are spare and a single call to the `split` method, would take $2*2 = 4$ registers to hold the information about the new inner arrays. And $2+2 = 4$ registers to hold information and data of the outer array.

Another solution to avoid memory allocation, would present itself if the array to be split was not used anywhere else in the program. Then we could be smart and "reuse" the array and thus saving the allocation of 2 words for the header information. But it would be essential for this to work, that the array we reuse 'dead'

For `concat` I also added a new case. `concat` first checks, whether the two lists to be joined are adjacent. This is done by Mips code with the same approach as in the pseudo code below

```
// data_addr holds the memory address's where data start for the
// two arrays to be joined. If adj is 0, the two arrays can be joined
// without copy over data

int adj
adj = (data_addr1 - data_addr2) - (len(lst1) * sizeof(lst1_eltp))
```

If the arrays are adjacent we use `dynalloc` to create an 'empty' array only containing header information(thus avoiding memory allocation).

If the two lists are not adjacent, then a new array are created with the size len(arr1) + len(arr2). And the data is copied over.

```
// n is size of final array
int n
n = len(arr1) + len(arr2)

// pointers to exsiting data
eltp* data_addr1;
eltp* data_addr2;

//final array
eltp final[n]

// Loop n times to copy all elements to new array
for (int i=0; i >= n; i++)
{
// if i is less than len(arr1) copy
// data from arr1 else copy data from arr2

if (i < len(arr1)
{
final[i] = *data_addr1
data_addr1 = data_addr1 + sizeof(eltp)
}
final[i] = *data_addr2
data_addr2 = data_addr2 + sizeof(eltp)
}
```

3.2.1 Optimizer

Cases for `concat` and `split` has been added to the existing functions in the optimizer as well, as to the functions performing constant folding and

constant propagation.

3.3 Memory allocation in the example

The `splitting` will make an outer array of size 2 holding the inner arrays, thus 2 words for data plus 2 words for the header information.

The data itself will be reused, but the two inner arrays has to each allocate 2 words for header information, thus at this point all in all 8 words has been allocated.

The `concat`enation will reuse the data(because it is adjacent), but also allocate 2 words for header information. Thus, 10 words - also known as 40 bytes - will be allocated.

3.4 Correctness of the code

To ensure correctness of `concat` and `split` the following tests has been added to the `DATA` folder.

3.4.1 Tests for `concat`

testConcatIntCharBool.fo
: Testing the correctness of `concat` when applied to arrays of bool, char and int.

testConcatArrays.fo
: Testing the correctness of `concat` when applied to two dimentional array.

testConcatFstArgBoolErr.fo
: Type testing with boolean as first argument..

testConcatFstArgCharErr.fo
: Type testing with char as first argument..

testConcatFstArgIntErr.fo
: Type testing with int as first argument..

testConcatSecndArgBoolErr.fo
: Type testing with boolean as second argument..

testConcatSecndArgCharErr.fo
: Type testing with char as second argument..

testConcatSecndArgIntErr.fo
: Type testing with int as second argument..

3.4.2 Tests for `split`

testSplitIntBoolChar.fo
Testing the correctness of `split` when applied to arrays of boolean, char and int.

testSplitTwoDimArray.fo
Testing the correctness of `concat` when applied to two dimensional array.

testSplitLowerBound.fo
Testing correctness of lower bound handling.

testSplitUpperBound.fo
Testing correctness of upper bound handling.

testSplitFstArgumentArrayErr.fo
Type testing with array as first argument.

testSplitFstArgumentBoolErr.fo
Type testing with boolean as first argument.

testSplitFstArgumentCharErr.fo
Type testing with char as first argument.

testSplitSecndArgumntBoolErr.fo
Type testing with boolean as second argument.

testSplitSecndArgumntChar.fo
Type testing with char as second argument.

testSplitSecndArgumntInt.fo
Type testing with int as second argument.

3.5 Source code

The full source code for `split` and `concat` can be found in the `SRC`. Search for 'msn378' to easily find the code blocks I have added or modified.

4 Implementing Constant Folding

In this sub assignment constant folding have been implemented into `Optimization.sml`. The idea of constant folding is simple, if you have an expression e holding other expression of constants, fold e to a simpler expression if possible. In the optimizer all expression are uniquely named as a variable and put into a `Fasto.Let` expression. Thus, constant folding does nothing unless also constant propagation is used. Please note, that I have also completed the function performing constant propagation.

The two optimization methods go hand in hand, in the way that a constant propagation run through, may lead to new possible optimizations with constant folding, and visa versa.

A lot of new code has been implemented, but instead of showing the rather uniform code here, I will instead summarize the cases where I have found to be able to make optimizations, and let you have a look at the code yourself. Note that much of the code implemented in the optimized relies on proper type checking, so when er get a expression of type `Fasto.Plus`, we assume that the types of the sub-expressions are of `Fasto.Int`.

4.1 Cases of optimization

Below is mentioned the folding optimizations I have done.

`Fasto.Plus`
: If both input expressions are a `Num`, a `Fasto.Num` is returned. If one of the expression is a `Num` with the value 0, the other expression is returned.

Fasto.Minus
: If both expressions are a `Num`, a `Fasto.Num` is returned. If $e1$ is a `Num` with the value 0, then Negate($e2$) is returned. If $e2$ is a number with the value 0, then $e2$ is returned.

Fasto.Less
: If $e1$ and $e2$ are constants they are fold with op$<$, and a `Log` expression is returned.

Fasto.If
: If $e1$ (the condition) is a constant, $e2$ is returned if $e1$ is true, otherwise $e3$ is returned as $e1$ is false.

Fasto.Times
: If $e1$ and $e2$ are both constants, they are multiplied and a `Num` expression is returned. If $e2$ is a number-constant with the value 0 a `Num` expression with value 0 is return. If $e1$ is a number-constant with the value 0 a `Num` expression with value 0 is return. If $e1$ is a number-constant with the value 1, $e2$ is returned. If $e2$ is a number-constant with the value 1, $e2$ is returned.

Fasto.Divide
: If $e1$ and $e2$ are both number-constants, they are divided and a `Num` expression is returned. If $e1$ is a number-constant with the value 0 a `Num` with value 0 is return. If $e2$ is a number-constant with the value 0 a exception is thrown.(you cannot divide with zero) If $e2$ is a number-constant with the value 1, $e1$ is returned

Fasto.Band
: If $e1$ and $e2$ are constants, they are evaluated and a `Num` expression is returned. If either $e1$ or $e2$ is a number-constant with the value 0, a `Num` with value 0 is returned.

Fasto.And
: If $e1$ and $e2$ are both constants with the value true, a `Log` expression with value `true` is returned. If either $e1$ or $e2$ are false, a `Log` expression is returned with the value `false`. If $e1$ is `true`, $e2$ is returned If $e2$ is `true`, $e1$ is returned

Fasto.Or
: If either $e1$ or $e2$ is a boolean-constant with the value `true`, a `Log` expression with value `true` is returned. if $e1$ is a boolean-constant with the value `false`, $e2$ is returned if $e2$ is a boolean-constant with the value `false`, $e1$ is returned

Fasto.Not
: if $e1$ is a boolean-constant with the value `false`, it is folded to a `Log` with value `true` if $e1$ is a boolean-constant with the value `true`, it is folded to a `Log` with value `false`

Fasto.Negate
: if $e1$ is a number-constant, a `Num` expression with the negated constant is returned.

4.2 Correctness of Constant folding

The way I have tested constant folding was by using the already exsiting test files in `DATA`. By implementing constant propagation I was was able to see the code folding when compiled with the FastoC -o, and by removing the print statements from the optimizer and adding `-o` as argument in `testg.sh`, I was able to run all the test with optimization enabled and check correctness.

4.3 Source code

The full source code can be found in the `SRC`, in the file `Optimization.sml`