Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

EXAM IN COURSE TDT 4165 PROGRAMMING LANGUAGES WITH A SOLUTION

Thursday August 17, 2006, 9.00–13.00

ENGELSK

Contact during the exam:
Ole Edsberg, Tlf 952 81 586

Exam aid code: C
No written material is permitted.
The officially approved calculator is allowed.

Read all of the following before you start making your answers:

- Answer briefly and concisely. Unclear and unnecessarily long answers will receive lower grades.
- All programs must be written in `Oz`.
- You may use the following functions and procedures from the textbook, without defining them: `Append`, `Drop`, `FoldL`, `FoldR`, `ForAll`, `IsNumber`, `Length`, `Map`, `Max`, `Member`, `Min`, `Reverse`, `Take`, `Solve`, `SolveAll`.

Paradigms

Problem 1: (14 %)

You are familiar with the following two programming paradigms:

- *Purely functional programming*, as supported by the declarative computation models in the textbook.
- *Object oriented programming (with an imperative core)*, as supported by `Java`.

Explain what are the most important differences and similarities between these two paradigms. Focus on the paradigms in themselvs, not on aspects specific to programming languages or their implementations and libraries. Write no more than one page. (*Hint*: Some important concepts are: declarativity, encapsulation, abstraction, polymorphism, state, reuse.)

Solution: TODO.

Functional programming

Problem 2: (21 %)

In this problem we will work with tree-structures defined by the following grammar:

```
<Tree>    ::= leaf | tree(val:<Value> left:<Tree> right:<Tree>)
<Value>   ::= ...
```

The sentences generated by the grammar are record-expressions in `Oz`. `<Value>` stands for a value in `Oz`. In the following example `Tree1` is bound to a record that is valid according to the grammar.

```
Tree1 = tree(val:1
             left:tree(val:2
                       left:leaf
                       right:leaf)
             right:tree(val:3
                        left:leaf
                        right:tree(val:4
                                   right:leaf
                                   left:leaf)))
```

a)

Write a function `{TreeSum Tree}` that takes the tree `Tree` as input and computes the sum of all the values in the tree. (Assume for this subtask that all the values in the tree have the same type, and that the `+` operator in `Oz` is valid for that type.)

For example, the following call should return 10:

```
{TreeSum Tree1}
```

Solution:

```
declare

T = tree(val:1
             left:tree(val:2
                       left:leaf
                       right:leaf)
             right:tree(val:3
                        left:leaf
                        right:tree(val:4
                                   right:leaf
                                   left:leaf)))

{Browse T}

fun {TreeSum T}
   case T
```

```
   of leaf then 0
   [] tree(val:V left:L right:R) then V + {TreeSum L} + {TreeSum R}
   end
end

fun {BottomUp F T U}
   case T
   of leaf then U
   [] tree(val:V left:L right:R) then {F V {F {TreeFold F L U} {TreeFold F R U}}}
   end
end

{Browse {TreeFold fun {$ X Y} X * Y end T 1}}

{Browse {TreeFold fun {$ X Y} X|Y end T nil}}

{Browse {TreeFold fun {$ X Y} op(X Y) end T leaf}}
```

b)

Will your solution from the previous subproblem run with constant stack-size? Give a convincing argument for your answer.

Solution: No, because the second recursive call will remain on the stack while the first is being evaluated.

c)

Use higher-order programming to make a generic function `{BottomUp F U Tree}` that takes the tree `Tree`, the binary function `F` and the base value `U` as input and performs a computation similar to the one in the a), but with the binary function instead of `+`. For example, the call

```
{BottomUp fun {$ X Y} X + Y end 0 Tree1}
```

should return 10, and the call

```
{BottomUp fun {$ X Y} X * Y end 1 Tree1}
```

should return 24.

The call

```
{BottomUp fun {$ X Y} op(X Y) end leaf Tree1}
```

should return

```
op(1
   op(op(2 op(leaf leaf))
      op(3 op(leaf op(4 op(leaf leaf))))))
```

.

Solution:

```
declare

T = tree(val:1
              left:tree(val:2
                        left:leaf
                        right:leaf)
              right:tree(val:3
                         left:leaf
                         right:tree(val:4
                                    right:leaf
                                    left:leaf)))

{Browse T}

fun {TreeSum T}
   case T
   of leaf then 0
   [] tree(val:V left:L right:R) then V + {TreeSum L} + {TreeSum R}
   end
end

fun {BottomUp F T U}
   case T
   of leaf then U
   [] tree(val:V left:L right:R) then {F V {F {TreeFold F L U} {TreeFold F R U}}}
   end
end

{Browse {TreeFold fun {$ X Y} X * Y end T 1}}

{Browse {TreeFold fun {$ X Y} X|Y end T nil}}

{Browse {TreeFold fun {$ X Y} op(X Y) end T leaf}}
```

Grammars and parsing

Problem 3: (30 %)

Here follows a grammar G_E for power expressions.

```
<Expr>    ::= <Expr> '**' <Expr> | <Integer>
<Integer> ::= ...
```

`<Integer>` stands for an integer in `Oz`.

a)

Is the grammar ambiguous? Give a convincing argument for your answer.

Solution: It is ambiguous, because for example the token list `[1 '**' 1 '** 1]` will have more than one derivation tree. (Draw the trees.)

b)

Give a short definition of the terms *precedence* and *associativity*. Explain the significance of these terms in relation to parsing of language expressions generated by G_E.

Solution: Definitions: TODO. Significance: Only one operator, so there can only be one precedence level. Stating the associativity of the operator, or changing the grammar to account for it, would resolve the ambiguity problem.

c)

Define a grammar G_T to represent abstract syntax trees for these expressions. Use BNF or EBNF. The trees should be valid record expressions in Oz.

Solution: See appendix.

d)

(Counts as three subtasks.)

Write a parser for G_E. The parser should be callable as a function `{Parse Tokens}` and should return a list of all possible abstract syntax trees for the expression given as `Tokens`. The abstract syntax trees should conform to G_T. `Tokens` is a list of terminal symbols in G_E, for example `[2 '**' 3 '**' 4]`.

Solution:

```
\insert Solve.oz

declare

Tokens = [1 '**' 2 '**' 3]

fun {Parse Tokens}
   fun {Expr Before Rest}
      case Rest
      of [X] then Before=nil {IsNumber X}=true X
      [] H|T then
         choice
            H='**' power({Expr nil Before} {Expr nil T})
         []
            {Expr {Append Before [H]} T}
         end
      else fail
      end
   end in
      {SolveAll fun {$} {Expr nil Tokens} end}

end

{Browse {Parse Tokens}}
```

Computation models

Problem 4: (14%)

In this problem we will extend a computation model to give it needed expressive power. The starting point is the data-driven, concurrent computation model (defined in chapter 4.1 of the textbook), hereafter called M.

We consider a situation where a server process handles requests from two clients. The clients send requests to the server through separate streams. We don't concern ourselves about what the server does with the requests, except that it handles them with the procedure `ProcessRequest`. We don't concern ourselves about what the clients are doing, except that they send requests to the server. The clients are operating independently from each other and from the server. We have made the following attempt to implement the server in M. (The server and each client runs in its own thread.)

```
proc {Server FirstStream SecondStream}
   case FirstStream
   of X|Xr then {ProcessRequest X}
      case SecondStream
      of Y|Yr then {ProcessRequest Y}
         {Server Xr Yr}
      end
   end
end
```

This implementation does not work correctly. The server attempts to alternatively read from each stream, but this will not guarantee that all the requests are handled, or even that they are handled in the order in which they were written to the streams. It is not possible to make a satisfying solution in M because it cannot handle componentss that behave non-deterministically in relation to each other.

a)

Add one or more new constructions to the computation model, so that you get a computation model M' that is able to solve the problem. You can choose constructions defined in the textbook, or define some by yourself. Use M' to make a server implementation that can handle the requests in the order in which they were written to the streams.

Solution: Add WaitTwo from the textbook.

b)

Which consequences will the chosen extension have for the declarativity of the computation model? Give a convincing argument for your answer.

Solution: It will not be declarative. FIXME: Show standard example.

Changing the language `P`

Problem 5: (21%)

This problem is concerned with the toy langauge `P`, for wich we wrote grammars, a parser and an interpreter in the project.

We wish to change `P` to give it dynamic scope. (Previously, the scope was lexical/static.) Lexical/static scope means that the bindings for free identifiers in a function body will be taken form the environment at the *definition* of the function. Dynamic scope means that the bindings for free identifiers in a function body will be taken from the environment at the *call* of the function.

In the subtasks a)-c) you will modify the grammars, the parser and the interpreter to make the scope dynamic. In the appendix, you can find the suggested solution from the project. Give references to the line numbers where you will make a change or an addition. Make reasonable assumptions were necessary.

a)

Write and explain the changes and additions you will make in the grammars for the abstract and the concrete syntax.

Solution: Nothing.

b)

Write and explain the changes and additions you will make in the parser.

Solution: Nothing.

c)

Write and explain the changes and additions you will make in the interpreter.

Solution: Change CEnv to Env in one place.

Appendix

Concrete and abstract syntax for P

```
Concrete syntax (epsilon means nothing)

<Expr>              ::= <ExprP2> | <Expr> <COP> <ExprP2>
<ExprP2>            ::= <ExprP3> | <ExprP2> <EOP> <ExprP3>
<ExprP3>            ::= <ExprP4> | <ExprP3> <TOP> <ExprP4>
<ExprP4>            ::= <LetExpr>
                      | <Functions>
                      | <IfExpr>
                      | <FunApp>
                      | (Ident) | (Num) | (Bool) | '(' <Expr> ')'
<LetExpr>           ::= let <LetItems> in <Expr> end
<LetItems>          ::= <LetItem> | <LetItem> ',' <LetItems>
<LetItem>           ::= (Ident) '=' <Expr>
<Functions>         ::= functions <FunDefs> in <Expr> end
<FunDefs>           ::= <FunDef> | <FunDef> ',' <FunDefs>
<FunDef>            ::= (Ident) '(' <FormalParamList> ')' <Expr> end
<FormalParamList> ::= epsilon | <FormalParams>
<FormalParams>      ::= (Ident) | (Ident) ',' <FormalParams>
<IfExpr>            ::= if <Expr> then <Expr> else <Expr> end
<FunApp>            ::= call (Ident) '(' <ActualParamList ')'
<ActualParamList> ::= epsilon | <ActualParams>
<ActualParams>      ::= <Expr> | <Expr> ',' <ActualParams>
<COP>               ::= '==' | '!=' | '>' | '<' | '=<' | '>='
<EOP>               ::= '+' | '-'
<TOP>               ::= '*' | '/'

Abstract syntax

<Expr>          ::= op( <OP> <Expr> <Expr> )
                  | <LetExpr>
                  | <Functions>
                  | <IfExpr>
                  | <FunApp>
                  | <Ident>
                  | <Number>
                  | <Bool>
<LetExpr>       ::= letexpr( <LetItems> <Expr> )
<LetItems>      ::= <LetItem> '|' nil | <LetItem> '|' <LetItems>
<LetItem>       ::= letitem( <Ident> <Expr> )
<Functions>     ::= functions( <FunDefs> <Expr> )
<FunDefs>       ::= <FunDef> '|' nil | <FunDef> '|' <FunDefs>
<FunDef>        ::= fundef( <Ident> <FormalParams> <Expr> )
<FormalParams> ::= nil | <Ident> '|' <FormalParams>
<IfExpr>        ::= ifexpr( <Expr> <Expr> <Expr> )
<FunApp>        ::= funapp( <Ident> <ActualParams> )
<ActualParams> ::= nil | <Expr> '|' <ActualParams>
```

```
<OP>            ::= ’==’ | ’!=’ | ’>’ | ’<’ | ’=<’ | ’>=’ | ’+’ | ’-’ | ’*’ | ’/’
<Ident>         ::= <OzAtom>
<Num>           ::= <OzInt>
<Bool>          ::= <OzBool>
```

Parser for P

```
% Grammar transformation.
%
% To enable parsing with left-right recursive descent, the first three
% lines of the grammar have been changed to the following. (The
% operators are still parsed left-assosiatively.)
%
% <Expr>             ::= <ExprP2> | <ExprP2> <COP> <Expr>
% <ExprP2>           ::= <ExprP3> | <ExprP3> <EOP> <ExprP2>
% <ExprP3>           ::= <ExprP4> | <ExprP4> <TOP> <ExprP3>

functor
export parse:Parse
define

   fun {Expr S1 Sn}
      {OpSeq ExprP2 COP S1 Sn}
   end

   fun {ExprP2 S1 Sn}
      {OpSeq ExprP3 EOP S1 Sn}
   end

   fun {ExprP3 S1 Sn}
      {OpSeq ExprP4 TOP S1 Sn}
   end

   fun {ExprP4 S1 Sn}
      T|S2=S1 in
      case T
      of let then {LetExpr S1 Sn}
      [] functions then {Functions S1 Sn}
      [] ’if’ then {IfExpr S1 Sn}
      [] call then {FunApp S1 Sn}

      [] ’(’ then E S3 in
         E = {Expr S2 S3}
         S3=’)’|Sn
         E
      [] ident(X) then Sn=S2 X
      [] num(X) then Sn=S2 X
      [] bool(X) then Sn=S2 case X
                            of ’true’ then true
                            [] ’false’ then false
```

```
                              end
    end
  end

  fun {LetExpr S1 Sn}
    S2 S3 X1 X2 in
    S1 = let|S2
    X1 = {SeqAsList LetItem Comma S2 'in'|S3}
    X2 = {Expr S3 'end'|Sn}
    letexpr(X1 X2)
  end

  fun {Functions S1 Sn}
    S2 S3 X1 X2 in
    S1 = functions|S2
    X1 = {SeqAsList FunDef Comma S2 'in'|S3}
    X2 = {Expr S3 'end'|Sn}
    functions(X1 X2)
  end

  fun {LetItem S1 Sn}
    S2 S3 I E in
    S1 = ident(I)|S2
    S2 = '='|S3
    E = {Expr S3 Sn}
    letitem(I E)
  end

  fun {FunDef S1 Sn}
    I FParams Body S2 S3 S4 in
    ident(I)|S2=S1
    S2='('|S3
    FParams = {FormalParamList S3 ')'|S4}
    Body = {Expr S4 'end'|Sn}
    fundef(I FParams Body)
  end

  fun {FormalParamList S1 Sn}
    case S1
    of [')'] then S1=Sn nil
    [] ')'|_ then S1=Sn nil
    else {SeqAsList
          fun {$ S1 Sn}
             case S1 of ident(I)|S2 then Sn=S2 I end
          end
          Comma S1 Sn}
    end
  end

  fun {IfExpr S1 Sn}
```

```
      X1 X2 X3 S2 S3 S4 in
      S1 = 'if'|S2
      X1 = {Expr S2 'then'|S3}
      X2 = {Expr S3 'else'|S4}
      X3 = {Expr S4 'end'|Sn}
      ifexpr(X1 X2 X3)
   end

   fun {FunApp S1 Sn}
      I AParams S2 S3 in
      S1 = call|S2
      S2 = ident(I)|'('|S3
      AParams = {ActualParamList S3 ')'|Sn}
      funapp(I AParams)
   end

   fun {ActualParamList S1 Sn}
      case S1
      of [')'] then S1=Sn nil
      [] ')'|_ then S1=Sn nil
      else {SeqAsList Expr Comma S1 Sn}
      end
   end

   fun {SeqAsList NonTerm Sep S1 Sn}
      X1 S2 in
      X1 = {NonTerm S1 S2}
      case S2
      of nil then S2=Sn [X1]
      [] T|S3 then if {Sep T} then X1|{SeqAsList NonTerm Sep S3 Sn}
                   else S2=Sn [X1]
                   end
      end
   end

   fun {OpSeq NonTerm Sep S1 Sn}
      fun {Loop Prefix S2 Sn}
         case S2 of T|S3 andthen {Sep T} then Next S4 in
            Next={NonTerm S3 S4}
            {Loop op(T Prefix Next) S4 Sn}
         else
            Sn=S2 Prefix
         end
      end
      First S2
   in
      First={NonTerm S1 S2}
      {Loop First S2 Sn}
   end

```

```
   fun {Comma X} X==',' end
   fun {COP Y}
      Y=='<' orelse Y=='>' orelse Y=='=<' orelse
      Y=='>=' orelse Y=='==' orelse Y=='!='
   end
   fun {EOP Y} Y=='+' orelse Y=='-' end
   fun {TOP Y} Y=='*' orelse Y=='/' end

   fun {Parse Tokens}
      {Expr Tokens nil}
   end

end
```

Interpreter for P

```
functor
export Interpret
define

   fun {Interpret AST}
      {Eval AST nil}
   end

   fun {Eval AST Env}
      case AST
      of op(Op E1 E2) then V1 V2 in
         V1 = {Eval E1 Env}
         V2 = {Eval E2 Env}
         case Op
         of '==' then V1==V2
         [] '!=' then V1\=V2
         [] '>' then V1>V2
         [] '<' then V1<V2
         [] '=<' then V1=<V2
         [] '>=' then V1>=V2
         [] '+' then V1+V2
         [] '-' then V1-V2
         [] '*' then V1*V2
         [] '/' then V1 div V2
         end
      [] letexpr(LetItems E) then NewEnv in
         NewEnv = {FoldL
                   LetItems
                   fun {$ U X} I E in
                      X = letitem(I E)
                      {Bind I {Eval E Env} U}
                   end
                   Env}
         {Eval E NewEnv}
```

```
      [] functions(FunDefs E) then CEnv in
         CEnv = {FoldL FunDefs
                   fun {$ U X} I FParams Body in
                      X = fundef(I FParams Body)
                      {Bind I funval(FParams Body CEnv) U}
                   end Env}
         {Eval E CEnv}
      [] ifexpr(E1 E2 E3) then case {Eval E1 Env}
                               of true then {Eval E2 Env}
                               [] false then {Eval E3 Env}
                               end
      [] funapp(I ActualParamList) then FParams Body CEnv ParamPairs in
         funval(FParams Body CEnv) = {Lookup I Env}
         ParamPairs = local fun {MakePairs L1 L2}
                               case L1#L2 of nil#nil then nil
                               [] (H1|T1)#(H2|T2) then (H1#H2)|{MakePairs T1 T2}
                               end
                            end in
                         {MakePairs FParams ActualParamList}
                      end
         {Eval Body {FoldL ParamPairs
                     fun {$ U X}
                        Formal Actual in
                        Formal#Actual = X
                        {Bind Formal {Eval Actual Env} U}
                     end CEnv}}
      else if {IsAtom AST} then {Lookup AST Env}
           elseif {IsInt AST} orelse {IsBool AST} then AST
           end
      end
   end

   fun {Bind Ident Value Env}
      case Env
      of nil then [bind(Ident Value)]
      [] bind(I V)|Rest then
         if Ident==I then bind(Ident Value)|Rest
         else bind(I V)|{Bind Ident Value Rest}
         end
      end
   end

   fun {Lookup Ident Env}
      case Env
      of nil then raise lookupFailure(Ident Env) end
      [] bind(I V)|Rest then
         if Ident==I then V
         else {Lookup Ident Rest}
         end
      end
```

```
   end

end
```

Example programs in P

Simple.p

```
let X = 1 in X end
```

Max.p

```
functions
   max(x, y)
      if x>y then x else y end
   end
in
   call max(3, 4)
end
```

Fact.p

```
functions
   fact(n)
      if n==0 then 1
      else n*call fact(n-1)
      end
   end
in
   call fact(3)
end
```

Fib.p

```
functions
   fib(x)
      if x==0 then 0 else
         if x==1 then 1 else
            call fib(x-1) + call fib(x-2)
         end
      end
   end
in
   call fib(12)
end
```

Fibacc.p

```
functions
   fib(x)
      functions
         fibacc(x, n, mem1, mem2)
            let
               fn = if n==0 then 0
                    else if n==1 then 1
                         else mem1+mem2
                         end
                    end
            in
               if x==n then fn
               else call fibacc(x, n+1,
                                fn, mem1)
               end
            end
         end
      in
         call fibacc(x, 0, 0, 0)
      end
   end

in

   call fib(12)

end
```

Oddeven.p

```
functions
   odd(x)
      if x==0 then false else call even(x-1) end
   end,
   even(x)
      if x==0 then true else call odd(x-1) end
   end
in
   call odd(3)
end
```

END OF EXAM