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

Tuesday December 6, 2005, 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.

The exam was created by Ole Edsberg. (*date/signature*)

The quality of the exam was approved by Per Holager. (*date/signature*)

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 programming problems must be solved with `Oz`.
- You may use the following functions and procedures from the textbook, without defining them: `Append`, `Drop`, `FoldL`, `FoldR`, `ForAll`, `Length`, `Map`, `Max`, `Min`, `Member`, `Reverse`, `Take`, `Solve`, `SolveAll`.

Multiple Choice

Fill in your answers on the answer sheet on the last page.

Only one of the alternatives for each subproblem is correct.

Problem 1: (20 %)

a)

Consider the following functions:

```
fun {FoldR L F U}
   case L
   of nil then U
   [] X|L1 then {F X {FoldR L1 F U}}
   end
end

fun {IterSum Xs}
   case Xs
   of nil then 0
   [] H|T then H+{IterSum T}
   end
end
```

Which of the functions will execute with constant stack size?

1. `FoldR` but not `IterSum`.
2. `IterSum` but not `FoldR`.
3. Both.
4. Neither.

Solution: 4. Neither.

b)

Which of the following statements about *purely functional* programming in the declarative, sequential computation model is the most correct?

1. Every program component is either a function or a procedure.
2. Thread synchronization is done with dataflow variables.
3. Variables are never unbound.
4. The execution of function calls is delayed until the result is needed.

Solution: 3. Variables are never unbound.

c)

Which programming language is more expressive, pure `Prolog` or the relational computation model from chapter 9 of the textbook?

1. Pure `Prolog`.

2. The relational computation model.
3. Neither.
4. The question is not meaningful.

Solution: 2. The relational computation model.

d)

Consider the following procedure in the relational computation model:

```
proc {Member ?A B ?C}
   case B
   of nil then C=false
   [] H|T then choice H=A C=true [] {Member A T C} end
   end
end
```

Which of the following logical expressions is a correct logical semantics for the procedure?

1. $\forall a, b, c.(member(a, b, c) \leftrightarrow$
   $(b = nil \wedge c = false) \vee (\exists h, t.(b = h|t \wedge ((h = a \wedge c = true) \vee (member(a, t, c))))))$
2. $\forall a, b, c.(member(a, b, c) \leftrightarrow$
   $(b = nil \wedge c = false) \vee (\forall h, t.(b = h|t \wedge ((h = a \wedge c = true) \vee (member(a, t, c))))))$
3. $\forall a, b, c.(member(a, b, c) \leftrightarrow$
   $(b = nil \vee c = false) \wedge (\exists h, t.(b = h|t \vee ((h = a \vee c = true) \wedge (member(a, t, c))))))$
4. $\forall a, b, c.(member(a, b, c) \leftrightarrow$
   $(b = nil \vee c = false) \wedge (\forall h, t.(b = h|t \vee ((h = a \vee c = true) \wedge (member(a, t, c))))))$

Solution: 1.

e)

Consider the following producer-consumer situation:

```
fun {Buffer In N}
   End=thread {List.drop In N} end
   fun lazy {Loop In End}
      case In of I|In2 then
         I|{Loop In2 thread End.2 end}
      end
   end
in
   {Loop In End}
end

fun lazy {Produce N} N|{Produce N+1} end
```

```
proc {Consume Xs}
   case Xs of _|Xr then {Delay 10} {Consume Xr} end
end

local Xs Ys in
   thread Xs = {Produce 0} end
   thread Ys = {Buffer Xs 3} end
   {Consume Ys}
end
```

Which of the following statements are true about this situation?

1. The producer will produce too many elements, filling up the memory.
2. The producer and consumer will execute in lockstep.
3. The execution will suspend indefinately.
4. Neither 1., 2. or 3. are true.

Solution: 4. Neither . . . .

f)

Consider the following producer-consumer situation. (This is the same as the situation in the previous subproblem, except that the `lazy` keywords have been removed.):

```
fun {Buffer In N}
   End=thread {List.drop In N} end
   fun {Loop In End}
      case In of I|In2 then
         I|{Loop In2 thread End.2 end}
      end
   end
in
   {Loop In End}
end

fun {Produce N} N|{Produce N+1} end

proc {Consume Xs}
   case Xs of _|Xr then {Delay 10} {Consume Xr} end
end

local Xs Ys in
   thread Xs = {Produce 0} end
   thread Ys = {Buffer Xs 3} end
   {Consume Ys}
end
```

Which of the following statements are true about this situation?

1. The producer will produce too many elements, filling up the memory.
2. The producer and consumer will execute in lockstep.
3. The execution will suspend indefinately.
4. Neither 1., 2. or 3. are true.

Solution: 1. …produce too many ….

g)

Consider the following inheritance situation.

```
class Parent1
   meth m1 skip end
   meth m2 skip end
end

class Parent2
   meth m1 skip end
   meth m3 skip end
end

class Parent3
   meth m2 skip end
   meth m3 skip end
end
```

```
class Child1 from Parent1 Parent2
   meth m1 skip end
   meth m2 skip end
end

class Child2 from Parent1 Parent3
   meth m1 skip end
   meth m3 skip end
end

class Child3 from Parent2 Parent3
   meth m2 skip end
   meth m3 skip end
end
```

How many of the subclasses `Child1`, `Child2` and `Child3` exhibit illegal inheritance?

1. 0
2. 1
3. 2
4. 3

Solution: 2. 1.

h)

Consider the following two functions. They are supposed to check for conflicts in the inheritance situation with the child class `Child` and the list of parent classes `Parents`, returning `true` if there are any conflicts and `false` otherwise. (The functions expect classes in the same form as in the object system in chapter 7 and exercise 8, but without wrappings.)

```
fun {Conflicts1 Child Parents}
   ConfMeth = {Set.minus
               {FoldL
                Parents
                fun {$ U X} {Set.inter U {Arity X.methods}} end
                nil}
               {Arity Child.methods}}
   ConfAttr = {Set.minus
               {FoldL
```

```
                Parents
                fun {$ U X} {Set.inter U X.attrs} end
                nil}
               Child.attrs} in
   ConfMeth\=nil orelse ConfAttr\=nil
end

fun {Conflicts2 Child Parents}
   case Parents
   of nil then false
   [] H|T then {FoldL
                T
                fun {$ U X}
                   ConfAttr={Set.minus
                             {Set.inter H.attrs X.attrs}
                             Child.attrs}
                   ConfMeth={Set.minus
                             {Set.inter {Arity H.methods} {Arity X.methods}}
                             {Arity Child.methods}} in
                   ConfAttr\=nil orelse ConfMeth\=nil orelse U
                end
                false} orelse
      {Conflicts2 Child T}
   end
end
```

Which of the functions are correct?

1. `Conflicts1` but not `Conflicts2`.
2. `Conflicts2` but not `Conflicts1`.
3. Both.
4. Neither.

Solution: 2. `Conflicts2` but not `Conflicts1`.

i)

If the class B is a subclass of the class A and the *substitution property* is satisfied, which of the following statements must be true?.

1. For each method in B, A has a method with the same label.
2. For each method in B, A does not have a method with the same label.
3. For each method in B, if A has a method with the same label, the method in B performs exactly the same operations on the object state as the method in A.
4. For each method in B, if A has a method with the same label, the method in B satisfies any invariant assertions specified for the method in A.

Solution: 4. . . . invariant assertions . . . .

j)

The following are statements about the object systems of `Java` and `Oz`. Which statement is correct?

1. `Oz` uses dynamic binding by default. `Java` uses static binding by default.
2. `Oz` uses delegation by default. `Java` uses forwarding by default.
3. `Oz` uses the type view of inheritance by default. `Java` uses the structure view by default.
4. Neither 1., 2. or 3. are correct.

Solution: 4. Neither . . . .

Functional programming

Problem 2: (15 %)

a)

You are familiar with the function `Map` as defined in the textbook. `{Map Xs F}` returns a new list calculated from the list `Xs` by applying the function `F` to each of its elements.

Write a function `TreeMap` that performs an analogous calculation for binary trees conforming to the following grammar:

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

Here, `<Value>` means any value in Oz.

Here is an example of this kind of tree:

```
T1 = tree(val:false
          left:tree(val:true
                    left:leaf
                    right:leaf)
          right:tree(val:true
                     left:leaf
                     right:leaf))
```

`{TreeMap T F}` should return a new tree calculated from the tree `T` by applying the function `F` to the `val` field of each node in `T`.

For example, `{TreeMap T1 fun {$ X} X==false end}` should return the following tree:

```
tree(val:true
     left:tree(val:false
               left:leaf
               right:leaf)
     right:tree(val:false
                left:leaf
                right:leaf))
```

Solution:

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

b)

Will your solution for a) execute with constant stack size? Give a convincing argument for your answer.

Solution: Our solution for a) will not execute with constant stack size, since there are two recursive calls, and when the first recursive call is executing, the second call will remain on the stack.

Relational programming

Problem 3: (20 %)

Write a function `{Palindrome Len Alphabet}` that returns a list containing all palindromes of length `Len` over the alphabet `Alphabet`. A palindrome is a sequence that is identical to itself when reversed. Use the relational computation model.

For example, the function call `{Palindrome 3 [a b c]}` should return the following list of palindromes. (The order of the list is not important.)

```
[[a a a] [a b a] [a c a]
 [b a b] [b b b] [b c b]
 [c a c] [c b c] [c c c]]
```

Solution:

```
fun {Letter Alphabet}
   case Alphabet
   of nil then fail
   [] H|T then
      choice H [] {Letter T} end
   end
end


fun {Palindrome Len Alphabet}
   fun {PalindromeRel Len Alphabet}
      if Len==0 then nil
      elseif Len==1 then [{Letter Alphabet}]
      elseif Len>1 then L={Letter Alphabet} in
         L|{Append {PalindromeRel Len-2 Alphabet} [L]}
      else fail
      end
   end in
   {SolveAll fun {$} {PalindromeRel Len Alphabet} end}
end
```

Grammars and parsing

Problem 4: (10 %)

Consider the following grammar for arithmetic expressions with postfix operators. (A postfix operator is an operator that is written after the operands.)

```
<Expr> ::= <Expr> <Expr> <Op> | (Int)
<Op>   ::= '+' | '-' | '*' | '/'
```

`(Int)` is a terminal symbol that stands for any integer. The tokenizer gives `(Int)` tokens as `Oz` records with the label `int` and with an Oz integer as its only content. For example, the expression `1 1 + 2 2 + *` will be processed by the tokenizer into the following token sequence: `[int(1) int(1) '+' int(2) int(2) '+' '*']`.

a)

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

Solution: The grammar is not ambiguous. An expression is either an integer or an expression consisting of two subexpressions and an operator. In the first case, it is obvious that there is only one derivation tree. In the second case, the operator is the rightmost symbol in the expression, and it is obvious that it has only one derivation tree. To the right of the operator comes the two expressions, and we can know unambigously where the first one ends and the second begins by counting the number of operators and integers, since an expression must have one more integers than operators. (The derivation for an expression contains n applications of the operator-introducing rule, where n is zero or greater. Each application increases both the number of operators and the number of `<Expr>`-symbols by one. Each `<Expr>` can only disappear by changing it into an integer. Since the derivation starts with an `<Expr>`-symbol, the number of integers must be one greater than the number of operators.) If we assume that the two expressions each only have one derivation tree, then the combined expression will only have one derivation tree also, since there is only one way to combine the subexpressions. So we have proved by induction that the grammar is not ambiguous. FIXME: clear this up.

b)

What is the significance of the concepts *precedence* and *associativity* for this grammar?

Solution: The concepts precedence and associativity have no significance for this grammar, since with postfix operators there is no ambiguity as to the order that the operations should be evaluated in.

c)

Does the grammar have any properties that make it unsuitable for parsing with left-right recursive descent? If so, give an example of such a property. If not, give an example of a property that the grammar doesn't have but that would have made the grammar unsuitable. For the property you gave as the example, explain why it makes grammars unsuitable for parsing by left-right recursive descent.

Solution: The grammar has left-recursion in the rule for `<Expr>`. This makes it unsuitable for parsing with left-right recursive descent, since when the parser tries to parse an instance of the non-terminal with the left-recursive rule, it will first try to parse an instance of the same non-terminal, and so on in unending recursion.

Declarativity and computation models

Problem 5: (10 %)

In this task, we will consider the consequences of adding the statement `IsDet` to various computation models. `IsDet` has the following syntax:

```
<s> ::= {IsDet <x> <y>}
```

The semantic rule for `IsDet` is as follows:

The semantic statement is `({IsDet <x> <y>}, E)`

Execution consists of the following actions:

If $E(\langle x \rangle)$ is determined (i.e. bound to a value), bind $E(\langle y \rangle)$ to `true`. otherwise, bind $E(\langle y \rangle)$ to `false`.

For each of the following computation models, state what would be the consequence for the declarativeness of the model if the model was extended with the `IsDet` statement. (None of the models have exceptions.) Give convincing arguments for your answers.

a)

The declarative, sequential computation model from chapter 2.

Solution: Adding the `IsDet` statement to the declarative, sequential computation model from chapter 2 will have no effect on the declarativeness of the model. Since the model is sequential, the statements in a program will always be executed in the same order. Consider a sequence of statements, some of which are `IsDet` statements. Since the model without `IsDet` is declarative, all executions with the same initial binding of the variables will result in the same binding of the variables right before the first `IsDet` statement. Therefore, the result of `IsDet` will be the same in each of these executions. By the same reasoning, the results of the remaining `IsDet` statements will also be the same. Therefore, the resulting model is declarative.

b)

The data-driven concurrent computation model from chapter 4.1.

Solution: The data-driven concurrent model is declarative. If we add the IsDet statement, we can write the following program:

```
local X in
   thread if {IsDet X} then skip else X=true end end
   thread if {IsDet X} then skip else X=false end end
end
```

This program will give a different binding of X depending on the scheduling of the threads. Therefore, we can see that adding IsDet has had the consequence of making the model lose its declarativeness.

c)

The demand-driven concurrent computation model from chapter 4.5.

Solution: The demand-driven concurrent model is declarative and is also a superset of the data-driven concurrent model, so the result will be the same as for b).

d)

The stateful, sequential computation model from chapter 6.

Solution: The stateful, sequential model is not declarative, and adding IsDet will not change that, since we can still write the same non-declarative programs as before.

Extending P

Problem 6: (25%)

Remember the toy language P that we wrote grammars, a parser and an interpreter for in the project.

We want to extend P with lists. A list is a composite value that contains a sequence of zero or more values. A list expression can be written with the keyword `list`, followed by an opening paranthesis, followed by a comma-separated sequence of zero or more expressions, followed by a closing paranthesis. A list expression can stand alone as a program. The value of a list expression is the list of the values of the expression the list expression consists of. The list `list(2, 4, 6)` is the value of the following list expression.

```
list(1+1, 2+2, 3+3)
```

We also need to add some operations for working with lists. The operation `head` gives the first element of a list. A `head` expression is written with the keyword `head`, followed by an opening parenthesis, followed by an expression, followed by a closing parenthesis. `head` gives a runtime failure if the value of the expression between the parentheses is an empty list or not a list. The following expression has the number 1 as its value:

```
head(list(1, 2, 3))
```

The operation `tail` gives the result of removing the first element from a list. A `tail` expression is written with the keyword `tail`, followed by an opening parenthesis, followed by an expression, followed by a closing parenthesis. `tail` gives a runtime failure if the value of the expression between the parantheses is an empty list or not a list. The following expression has as its value the list `list(2, 3)`.

```
tail(list(1, 2, 3))
```

The operation `cons` gives the result of constructing a list from two expressions, with the value of the first becoming the head and the value of the second becoming the tail. A `cons` expression is written with the keyword `cons`, followed by an opening parenthesis, followed by an expression, followed by a comma, followed by an expression, followed by a closing parenthesis. `cons` gives a runtime failure if the value of the second expression is not a list. The following expression has as its value the nested list `list(list(1, 2) 3, 4)`.

```
cons(list(1, 2) list(3, 4))
```

Here is an example program that finds the length of a list. The program returns the number 3.

```
functions
   length(xs)
      if xs==list() then 0
      else 1+call length(tail(xs))
      end
   end
in
   call length(list(1,2,3))
end
```

Here is an example program that appends two lists. The program returns the list `list(1, 2, 3, 4, 5, 6)`.

```
functions
   append(xs, ys)
      if xs==list() then ys
      else cons(head(xs), call append(tail(xs), ys))
      end
   end
in
   call append(list(1,2,3), list(4,5,6))
end
```

In the subproblems a)-c), you will modify the grammars, parser and interpreter to make it able to handle lists, list expressions and the list operations `head`, `tail` and `cons`. In the appendix you will find the suggested solution from the project. Give references to the line numbers where you will make a modification or an addition. Make reasonable assumptions where necessary.

a)

Write the modifications and additions you will make to the grammars for the concrete and abstract syntax.

Solution: Concrete syntax:

- Add the following after line 10:

```
| <List> | <Head> | <Tail> | <Cons>
```

- Add the following after line 25 (line numbers before the above addition):

```
<List>            ::= list '(' <ListElements> ')'
<ListElements>    ::= epsilon | <Expr> | <Expr> ',' <ListElements>
<Head>            ::= head '(' <Expr> ')'
<Tail>            ::= tail '(' <Expr> ')'
<Cons>            ::= cons '(' <Expr> <Expr> ')'
```

Abstract syntax:

- Add the following after line 36 (line numbers before the above additions):

```
| <List>
| <Head>
| <Tail>
| <Cons>
```

- Add the following after line 50 (line numbers before the above additions):

```
<List>         ::= nil | <Expr> '|' <List>
<Head>         ::= head(<Expr>)
<Tail>         ::= tail(<Expr>)
<Cons>         ::= cons(<Expr> <Expr>)
```

b)

Write the modifications and additions you will make to the parser.

Solution:

- Add the following after line 44:

```
[] list then S3 in
   S2 = '('|S3
   case S3
   of ')'|S4 then Sn=S4 nil
   else{SeqAsList Expr Comma S3 ')'|Sn}
   end
[] head then S3 in
   S2 = '('|S3
   head({Expr S3 ')'|Sn})
[] tail then S3 in
   S2 = '('|S3
   tail({Expr S3 ')'|Sn})
[] cons then S3 S4 in
   S2 = '('|S3
   cons({Expr S3 ','|S4} {Expr S4 ')'|Sn})
```

c)

Write the modifications you will make to the interpreter. When a program has a list as its value, the interpreter should return that value as an `Oz` list.

Solution:

- Add the following after line 60:

```
[] head(E) then H|_={Eval E Env} in H
[] tail(E) then _|T={Eval E Env} in T
[] cons(E1 E2) then {Eval E1 Env}|{Eval E2 Env}
```

- Change lines 61-63 (line numbers before the above addition) to:

```
else if {IsInt AST} orelse {IsBool AST} orelse {IsList AST} then AST
     elseif {IsAtom AST} then {Lookup AST Env}
     end
```

d)

We now want to add *static typechecking* to `P`. Explain as concretely as possible what changes we can make to the grammars, the parser, the interpreter and to the system as a whole in order to accomplish this. Write no more than one page.

Solution: (The language will have four types: numbers, booleans, lists and functions. In addition there can be different types of lists and functions depending on the type of the contents of a list and the type on the parameters and return value of a function.) We need to extend the concrete syntax to include specifications of the type of each identifier introduced in a let expression or as a formal argument to a function. We also need to extend the concrete syntax to include specifications for the return values of functions and the type of the contents of a list. The abstract syntax must be extended to hold these type specification in the nodes for let expressions, function declarations and list expressions. The parser must be extended to parse the new concrete syntax and build the new abstract syntax trees.

The type checking itself will happen in a new function that takes an AST as input and returns the type of the AST. If the AST is misstyped, the function raises an exception. For a node in the AST, the function finds the types of the children and from them calculates the type of the node. At the bottom of the tree we find number and boolean literals, whose types can be found without recursion.

The interpreter only needs to be changed to ignore the type information in the AST.

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
```

Answer sheet for Problem 1, Multiple Choice.

Fill in your student number and answers to Problem 1 on this page.

Student number: ....

	1.	2.	3.	4
a)				
b)				
c)				
d)				
e)				
f)				
g)				
h)				
i)				
j)				

Remember to hand in this page along with the rest of your answers!

END OF EXAM