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

Exam for the course
TDT4165 / SIF8028
Programming Languages
with Proposed Solution

Monday August 8. 2005, 0900 – 1300 hrs.

ENGLISH

Hjelpemidler: C – An officially approved calculator is permitted.
No printed or hand written material is allowed.

Read all of the following before you start making a solution. Note that the last page is a form for answering task 1.
For each task and subtask, a percentage is given for the weight this part has in the final score used for grading.
Answer briefly, comprehensibly and legibly: Unclear and unnecessarily long answers will result in lower grades.
If you find some task unclear or incompletely specified, write down the assumptions you make in your soolution.
Unless some other language is specified for a task, all programming is to be done in oz.

Task 1 (20%) Multiple Choice/ Theory

(Each question/ subtask has equal weight in the final grade.) Fill in your answers in the form on page 8.
Only one of the alternative answers for each question is meant to be correct. If in doubt, choose the alternative you think the teaching staff had in mind.. Unless something else is implied, code in the following is in oz.

a)

A Lisp macro essentially serves to -

1: generate lists of values

2: generate Lisp code ← ***Solution***

3: optimize functional programming

4: encapsulate a group of function definitions

b)

Which of the following operations is not done in the single assignment store?

1: creating values ← ***Solution***

2: binding values to variables

3: binding variables to variables

4: creating variables

c)

Which of the following constructs is in the declarative kernel language?

1: **thread** <***s***>$_1$ **end**

2: **case** <***x***> **of** <***pattern***> **then** <***s***>$_1$ **else** <***s***>$_2$

3: **for** <***x***> **in** <***y***>$_1$**..**<***y***>$_2$ **do** <***s***>$_1$ **end**

4: **{SolveAll** <***f***>$_1$**}** ← ***Solution***

d)

Which language is the following program fragment written in:

```
class Expr {public: static Expr* primary(void);
```

1: Prolog

2: Haskell

3: Java

4: C++ ← ***Solution***

e)

The term *procedural abstraction* means:

1: procedures that can be used for many purposes, in different contexts

2: object oriented

3: any statement can equivalently be written in a procedure (function) which is then called ← ***Solution***

4: procedures can (should) be made so that many internal details can be left out of their specification

f)

In case you do not remember them:

```
fun{FoldR X F S}
      case X
      of E|Xr then {F E {FoldR Xr F S}}
      else S
      end
end
fun{FoldL X F Ac}
      case X
      of E|Xr then {FoldL Xr F {F Ac E}}
      else Ac
      end
end
```

Given two functions

```
declare fun{G1 L R} L|R end
declare fun{G2 L R} R|L end
```

Which of the following calls will give the result **[3 2 1]**:

1: **{FoldL [1 2 3] G1 nil}**

2: **{FoldL [1 2 3] G2 nil}** ← ***Solution***

3: **{FoldR [1 2 3] G1 nil}**

4: **{FoldR [1 2 3] G2 nil}**

g)

Given

```
declare fun{H L R} R-L end
```

What is the value of

{FoldL [1 2 3] H 1}

1: -5

2: -1

3: 1 ← ***Solution***

4: 2

h)

Purely functional languages do not have -

1: partial values ← *Solution*

2: more than 1 parameter in function calls

3: static, strong typing

4: lazy evaluation

i)

Given

```
declare E F G H
thread
        proc{X1 P Q S T}
            case T of more|Tr then
                R=P+Q  Sr in
                S=R|Sr
                {X1 Q R Sr Tr}
            end
        end
    in
        {X1 H 1 E F}
end
thread
        proc{X2 N P Q}
            Qr in
            Q=more|Qr
            case P of R|Pr then
                if N>0
                then {X2 N-1 Pr Qr}
                end
            end
        end
    in
        {X2 G E F}
end
G=11
H=1
for D in E do
    {Show D}
end
```

This code will write out a series of numbers which is

1: part of an arithmetic series

2: part of a geometric series

3: part of the Fibonacci series ← *Solution*

4: a line from Pascal's triangle

j)

What does the following construct do

{Exchange <***x***> <***y***> <***z***>**}**

1: binds *E*(<***z***>) to the old value of *E*(<***x***>), sets new value of *E*(<***x***>) to *E*(<***y***>)

2: binds *E*(<***x***>) to the old value of *E*(<***y***>), sets new value of *E*(<***y***>) to *E*(<***z***>)

3: binds *E*(<***z***>) to the old value of *E*(<***y***>), sets new value of *E*(<***y***>) to *E*(<***x***>)

4: binds *E*(<*y*>) to the old value of *E*(<*x*>), sets new value of *E*(<*x*>) to *E*(<*z*>) ***← Solution***

Task 2 (20%) Declarative programming

Write a function **Pasc** which returns Pascal's triangle in the form of a list of lists. (Make auxiliary functions as needed.) The function should take the size of the triangle as parameter. The result should be a little bit rotated compared to what we are used to: **{Pasc 8}** shall give the result

```
[[1   1   1   1   1   1   1   1]
 [1   2   3   4   5   6   7]
 [1   3   6  10  15  21]
 [1   4  10  20  35]
 [1   5  15  35]
 [1   6  21]
 [1   7]
 [1]]
```

***Solution (with example of use)*:**

```
declare
fun{PasLine Up Pr N}
  if N>0 then U Rup Th in
    Up = U|Rup
    Th =Pr+U
    Th|{PasLine Rup Th N-1}
  else nil
  end
end

declare
fun{Ones N}
  if N>0 then 1|{Ones N-1} else nil end
end

declare
fun{Pasc Up N}
  Up|if N>0 then
    {Pasc {PasLine Up 0 N} N-1}
  else nil
  end
end


{Browse {Pasc {Ones 8} 7}}
```

Task 3 (15%) Higher Order & Relational Model

Make a filter in the relational model. More concretely, write a function that takes a boolean function and a list as parameters, and that produces a function without parameters. This resulting function will then be given as parameter to **SolveAll**, it should comprise **choice**. It shall return the elements of the list that produce **true** when given as parameter to the boolean function.

***Solution (with example of use)*:**

```
declare
fun{Source F L}
   fun{Each L}
      case L
      of E|Lt then
         choice E
         [] {Each Lt}
         end
      else fail
      end
   end in
   fun{$} E={Each L} in {F E}=true E end
end

declare
G={Source fun{$ V} (V mod 3)==0 end [1 10  3  5  7  9  2  6  4  8]}
{Browse {SolveAll G}}
```

Task 4 (25%) Parser

Given a syntax for lists of: lists and integer literals:

<*list*> ::= '**[**' <*seq*> '**]**' .
<*seq*> ::= | <*seq*> <*list*> | <*seq*> '**num**'.

Here, '**num**' should be understood as a natural number. In the usual way, we assume that the input to the lexer module at this point should be a series of digits; out of the lexer and into the parser one will have a tuple **num(V)** where **V** is the value of the natural number. An input string to the lexical analyzer may e.g. be

[2 [3 []] 4 [[5] 6]]

the resulting input to the parser may then be the following list of terminal symbols:

['[' num(2) '[' num(3) '[' ']' ']' num(4) '[' '[' num(5) ']' num(6) ']' ']']

The result from the parser shall be a semantic tree. Our example shall give the value:

```
num(   con:2
       nxt:lis(     con:num( con:3      nxt:lis(    con:nil      nxt:nil ))
                    nxt:num( con:4
                             nxt:lis(    con:lis(    con:num(    con:5    nxt:nil )
                                                     nxt:num(    con:6    nxt:nil ))
                                         nxt:nil ))))
```

Alternatively, we can draw this:

```
→ num  → lis  → num  → lis →/
  ↓      ↓      ↓      ↓
  2      num → lis →/  4      lis  →  num →/
         ↓     ↓              ↓       ↓
         3     /              num →/  6
                              ↓
                              5
```

Here we have **/** in stead of **nil**.

a) (5%)

The syntax above is not suitable when making the usual kind of top-down, left-to-right parser. Why not? Find an equivalent syntax that is suited. Try to make it as simple as possible.

***Solution*:**

The problem is that <*seq*> has two alternative productions that are left-recursive. According to the rule for

removing left-recursion, it can be rewritten as

<seq> **::=** *<seq rest>*
<seq rest> **::=** | *<list>* *<seq rest>* | **'num'** *<seq rest>*

Here, the non-terminal *<seq rest>* **is obviously superfluous. The complete grammar can then be written:**

<list> **::= '['** *<seq>* **']' .**
<seq> **::=** | *<list>* *<seq>* | **'num'** *<seq>*

b) (20%)
Make a recursive descent parser that accepts the syntax you found in the preceding subtask, and which produces the semantic tree described above.. (The semantic tree is ordered such that it should match nicely the syntax you arrived at.)

***Solution (with example of use)*:**

```
declare Seq
fun{List In Ut} X in
   In = '['|X
   {Seq X ']'|Ut}
end
fun{Seq In Ut} X in
   case In
   of '['|_ then lis(con:{List In X} nxt:{Seq X Ut})
   [] 'num'(X)|It then num(con:X nxt:{Seq It Ut})
   else Ut=In nil
   end
end

declare R
In=[ '[' num(2) '[' num(3) '[' ']' ']' num(4) '[' '[' num(5) ']' num(6) ']' ']' ]
{Browse {List In R}}
{Browse R}
```

Task 5 (20%) Type Checking

In this task, we shall consider type checking for a strictly typed language. Correct use of data types shall be checked after parsing and before the program is executed. The concrete syntax, i.e. the detailed syntax for input to the lexer and parser, is not important in this task: We presume the parser produces error free semantic trees (in the textbook: *abstract syntax trees*) of the form defined by the following EBNF syntax:

<*Program*> ::= '**[**' { <*Stmt*> } '**]**'

<*Stmt*> ::= **decl(ident** <*Type*> **)**
| **asg(ident** <*Expr*> **)**

<*Expr*> ::= **plus(**<*Expr*> <*Expr*>**)**
| **val(**<*Type*>**)**
| **ident**

<*Type*> ::= **int** | **bool**

Explanation:

- **decl(ident** <***Type***> **)** means, a variable named **ident** is declared with type <*Type*>.
- **ident** is an identifier, i.e. a variable name, represented as an atom in oz. When an identifier is used in an <*Expr*> or **asg(...)**, it must have occurred earlier in a declaration such that its <*Type*> is known.
- **asg(ident** <***Expr***> **)** means that when this is executed, the value of <*Expr*> will be assigned to the variable **ident**. This is only allowed if the types of the variable and the <*Expr*> are the same.
- **plus(**<***Expr***> <***Expr***>**)** means that when this is executed, a value is produced by adding the values of the two <Expr>-s. This is only allowed if both have type **int**, and the type of the result will be **int**..
- **val(**<***Type***> **V)** represents a literal of <***Type***> with value **V.** The latter is not needed for type checking.

An example of such a semantic tree:

[decl(a int) decl(b int) asg(b plus(a val(int 3)))]

This first declares the variables **a** and **b**, both of type **int**. Then the value of **a + 3** is assigned to variable **b**.
For keeping track of identifiers and types, you may use the following from the standard library:

initiation: **declare D={NewCell nil}**
D:={NewDictionary}

to record a new variable: **{Dictionary.put @D Ident Type}**

to find the type of a variable: **Type={Dictionary.get @D Ident}**

a) (5%)
Explain how you will organize the code of the following subtask..

***Solution*:**

We need a function Make that creates a dictionary, which will keep track of all mappings from identifiers to their types.

We need a function Type that takes the representation of an expression and the dictionary as parameters, and returns the type of the expression: If it is a plus ..., check that both parts have type int, if so – return int. If it is a literal, return the type provided. If it is an identifier, find the type in the dictionary.

Finally, we need a procedure Check that runs through the semantic tree: For declarations, it should record the identifier – type pairs in the dictionary. For assignments, it should check that the type of the identifier is the same as the type of the expression.

b) (15%)
Make a program that checks that types are used correctly in such semantic trees. In particular, make a procedure **Check** which takes the tree and possibly a 'dictionary' or whatever as parameters, and which raises an exception decribing the first error (an error message in the form of an atom or tuple label) if any.

Solution (with example of use):

```
declare
fun{MakeD}
        D={NewCell nil} in
        D:={NewDictionary}
        D
end
fun{Type E D}
        case E
        of plus(L R) then
                if {Type L D}==int andthen {Type R D}==int
                then int else raise badPlus(L R) end end
        [] val(T _) then T
        [] Id then {Dictionary.get @D Id}
        end
end
proc{Check S D}
        case S
```

```
        of decl(Id Ty)|R then {Dictionary.put @D Id Ty} {Check R D}
        [] asg(Id Expr)|R then
                if {Type Expr D}=={Dictionary.get @D Id}
                then {Check R D} else raise badAsg(Id Expr) end
                end
        [] nil then skip
        end
end


{Check [decl(a int) decl(b int) asg(b plus(a val(int 3)))] {MakeD}}
```

Student number: . . . . . .

Date:. . . . . Page nr.:. . . Pages in all: . . .

Form for answering Task 1: Multiple Choice

	answer:	1	2	3	4
subtask					
a)			X		
b)		X			
c)					X
d)					X
e)				X	
f)			X		
g)				X	
h)		X			
i)				X	
j)					X

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