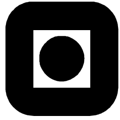


Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

EXAM IN COURSE TDT 4165
PROGRAMMING LANGUAGES

Saturday May 28, 2005, 9.00–13.00

ENGELSK

Contact during the exam:
Per Holager (lecturer), Tlf 996 17 836
Per Kristian Lehre, Tlf 913 60 757

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.

Multiple Choice

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

Only one of the alternatives in each subproblem is correct.

Problem 1: (15 %)

a) Which language is the following program fragment written in:

```
size::BinTree a -> Integer
size Empty = 0
size(Node va lt rt)=1+(size lt)+(size rt)
```

1. Prolog
2. Haskell
3. Java
4. C++

b) What is meant by the term *memoization*?

1. That the interface of a component should be independent of the computation model used to implement it.
2. To keep track of the results of calls to a function so that future calls can be handled quicker.
3. That objects can share data without any particular precautions when all objects run in the same thread.
4. Using an eager version as an optimization over a lazy version.

c) Given two Oz functions for computing the factorial function, $n!$ (recall that $0! = 1$):

```
declare
fun {Fac1 N}
   if N>1 then {Fac1 N-1}*N
   else 1
   end
end
```

```
declare
fun {Fac2 N}
   if N<1 then 1
   else N*{Fac1 N-1}
   end
end
```

Which of the following is true:

1. `Fac1` does not compute the function correctly.
2. `Fac2` does not compute the function correctly.
3. `Fac1` and `Fac2` are roughly equally fast.
4. `Fac1` is not right recursive (tail recursive), it is therefore much slower than `Fac2`.

d) Following the terminology of the textbook, what can be said of the type system of Oz?

1. Weak and dynamic
2. Weak and static
3. Strong and dynamic
4. Strong and static

e) Which is a correct *structural operational semantics* from Chapter 13 for `if`-statements for the case that the condition after `if` holds:

1. $$\frac{\texttt{if}\ X\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \;\Big\|\; S_1\{X \rightarrow x\}}{\sigma \wedge x = \texttt{true} \;\Big\|\; \sigma \wedge x = \texttt{true}}$$

2. $$\frac{\texttt{if}\ x\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{end} \;\Big\|\; S_1}{\sigma \;\Big\|\; \sigma \wedge x = \texttt{true}}$$

3. $\dfrac{\texttt{if } x \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}}{\sigma} \Bigg\| \dfrac{S_1}{\sigma}$ if $\sigma \models x = \texttt{true}$.

4. $\dfrac{\texttt{if } X \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ end}}{\sigma \wedge X = \texttt{true}} \Bigg\| \dfrac{S_1\{X \rightarrow x\}}{\sigma \wedge X = \texttt{true}}$

f) The structural operational semantics for failed values does not comprise the reduction:

$\dfrac{\{\texttt{FailedValue } x\ x_f\}}{\sigma} \Bigg\| \dfrac{\texttt{skip}}{\sigma \wedge x_f = failed(x)}$, because

1. that would not give a good error reaction if x_f is bound already.
2. execution would just stop, the store would not change.
3. then, execution would not stop immediately.
4. the result of function *failed* cannot be assigned to x_f because it is not a boolean.

g) What is the effective data type (in the “general” type notation of the lectures) of the parameters in the method in the following Java code fragment:

```
public class Cc { public double pr(double p1, Cc p2) ... }
```

1. `p1: ref double, and p2: ref Cc`
2. `p1: ref double, and p2: Cc`
3. `p1: double, and p2: ref Cc`
4. `p1: double, and p2: Cc`

h) What is the effective data type (in the “general” type notation of the lectures) of variable `x` declared by following C++:

```
int *x(char*)
```

1. `ref func(ref char)int`
2. `ref [0..255] int`
3. `[ref char]ref int`
4. `func(ref char)ref int`

i) Which of the following statements about threaded state (i.e. accumulator parameter) is the least correct?

1. often reduces the number of procedure arguments.
2. is an alternative to explicit state.
3. is a declarative programming technique.
4. is useful for making efficient programs.

j) Which language is the following program fragment written in:

```
nyrootsyn(BM,NN,Gender):-nrootsyn(BM,NN),nyroot(NN,Gender),!.
```

1. Prolog
2. Haskell
3. Java
4. C++

Programming

Problem 2: (8 %)

Given a Java class for a singly linked list of Strings,

```
public class LiStr { String str; LiStr next; ... }
```

Write a Java method (`public void printList()`) for this class, that traverses the list and writes out the Strings (using `System.out.println(str)`). Limit yourself to the single assignment paradigm, in particular: do not use any `for`- or `while`-loops.

(If you don't know Java, use some other well known imperative language except Oz. Remember to tell which.)

Relational Model

Problem 3: (25 %)

Write a function in the relational model that produces all permutations of a given input list of any size. Show how your function should be called with the `SolveAll`-function defined in the text book.

Remark: You may use the built-in function `{Append List1 List2}`.

Parser

Problem 4: (26 %)

Given a simplified fragment from the syntax of Java:

```
<term>       ::= <field term> | <call term> .
<field term> ::= 'id' | <term> '.' 'id' .
<call term>  ::=  <field term> '(' <param> ')' .
<param>      ::=  ε | <term> .
```

Here, the terminal symbol `'id'` represents an identifier, the lexical analyzer will also provide the actual name in the form of an atom (in the obvious way). The symbol ϵ represents the empty string.

The following is a syntax for a semantic tree (in the book: abstract syntax tree) for a parser in Oz for the syntax above:

```
<term>       ::= <field term> | <call term> | nil .
<field term> ::= field(id: <id's text>  nxt: <term> ) .
<call term>  ::= call(id: <id's text>  prm:<term>  nxt: <term> )
```

That is, if input to the lexical analyser is

```
Nils.venn(Jens).farge
```

then the list of tokens input to the parser will be

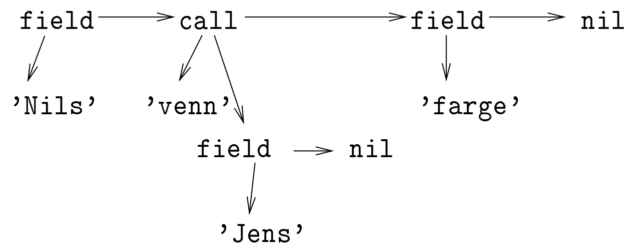
```
[ 'id'('Nils') '.' 'id'('venn') '(' 'id'('Jens') ')'
 '.' 'id'('farge') ]
```

Except for the order of the fields, the output should be

```
field(id:'Nils'
     nxt:call(id:'venn'
             prm:field(id:'Jens' nxt:nil)
             nxt:field(id:'farge'  nxt:nil)
   )       )
```

Alternatively,

PSfrag replacements



Here, edges to the next are drawn horizontally to the right.

a) Is this syntax fragment a suitable starting point for writing a recursive descent parser? Explain your reasoning. If it is not suitable, set up a suitable equivalent grammar. Try to make it as simple as possible.

b) Write a recursive descent parser in Oz for this language. The parser shall return `true` when parsing succeeds and `false` when parsing fails.

c) Extend your answer to **b)** above, to make it produce semantic trees of the specified form.

Interpreter

Problem 5: (26 %)

Introduction

In this problem we will consider an interpreter for a very simple imperative language. The concrete syntax, i.e. the syntax for the input to the parser, doesn't matter for this exercise. We assume that there exists a parser that generates semantic trees (in the book: abstract syntax trees) of the form defined by the syntax in Figure 1. As you see, the semantic trees are structures of nested records in Oz. The Appendix contains an interpreter for these trees. Your task will be to extend and discuss extending the semantic tree and the interpreter.

An informal description of the semantic tree and its semantics

A program (<Program> in Fig. 1) is represented as a list of statements (<Statement> in Fig. 1). The statements are executed in their order of appearance the list. There are two types of statements: assign(I E) makes a binding in the store between the variable with indentifier I (<Identifier> in Fig. 1) and the value of the expression E (<Expression> in Fig. 1), and print(E) prints the value of the expression E to the screen. A variable is created in the store when it is first bound to a value. The relationships between identifiers and variables are permanent. A variable can be bound to a value any number of times, even to values of differing types. The possible types of values are integers (<Integer> in Fig. 1) and booleans (<Boolean> in Fig. 1). When a variable is re-bound, the new value replaces the old. The semantics of expressions should be evident from the syntax.

The interpreter

The interpreter can be seen as the formal definition of the semantics. It consists of the main function Execute and some help functions.

The function Execute takes as input an execution state and returns the list of execution states resulting from executing that state one step at a time until the execution is finished. Each execution state is represented with a record as defined in Figure 2.

An execution state consists of the list of remaining statements and the current store. When interpretation starts, Execute is called with a state containing the whole program and an empty store. As execution proceeds, Execute will be called recursively with the execution state resulting from carrying out the preceding single execution step. When the current list of statements is nil, execution is finished.

The function Evaluate takes as input an expression and a store and returns the value of the expression with respect to the store.

The function Bind takes as input an identifier, a value and a store and returns the store resulting from binding the identifier to the value in the input store.

The function Lookup takes as input an identifier and a store, and returns the value to which the identifier is bound in the store, if it is bound.

Example

Suppose we use the interpreter to execute:

```
SimpleProgram = [assign('x' 2) assign('x' add('x' 2)) print('x')]
Trace = {Execute state(SimpleProgram empty)}
```

Then the browser will show the answer `4`, `Trace` will contain the following:

```
[state([assign(x 2) assign(x add(x 2)) print(x)] empty)
 state([assign(x add(x 2)) print(x)] bind(x 2 empty))
 state([print(x)] bind(x 4 empty))
 state(nil bind(x 4 empty))]
```

Subproblems

a) We want to extend the language with `if`-constructions. An `if`-statement (`<Statement>`) contains a conditional expression (`<Expression>`), a `then`-statement (`<Statement>`) and an `else`-statement (`<Statement>`). When it is executed, the conditional is evaluated. Then, if the result is true, the `then`-statement is executed. Otherwise, the `else`-statement is executed. Invent a suitable extension to the semantic tree to handle `if`-constructions. Describe it as an extension to the syntax. Write Oz code that will enable the interpreter to properly execute your `if`-constructions. Indicate by line numbers where in the syntax (Fig. 1) and interpreter your additions should be placed.

b) We want to extend the language with `while`-loops. A `while`-statement (`<Statement>`) contains a conditional expression (`<Expression>`) and a body statement (`<Statement>`). When it is executed, the conditional expression is evaluated. If the result is true, the body statement is executed and then the `while`-statement is executed again. Otherwise, execution proceeds with the rest of the program. Invent a suitable extension to the semantic tree to represent `while`-loops and show the extension to the syntax. Write Oz code that will make the interpreter properly execute your `while`-loops. Indicate by line numbers where in the syntax and interpreter your additions should be placed.

c) Explain with words, not code or grammar rules, how you would extend the semantic tree and interpreter with the following features:

- Dynamic type checking
- Static type checking

d) The operation of the interpreter in this exercise resembles that of the abstract machine used to describe semantics in Chapter 2 of the textbook. That machine uses environments, but this interpreter seems to function properly without them. Why?

```
1  <Program>      ::= "["{<Statement>}"]"
2  <Statement>    ::= assign(<Identifier> <Expression>)
3                   | print(<Expression>)
4  <Expression>   ::= add(<Expression> <Expression)
5                   | subtract(<Expression> <Expression>)
6                   | multiply(<Expression> <Expression>)
7                   | divide(<Expression> <Expression>)
8                   | equals(<Expression> <Expression>)
9                   | lessthan(<Expression> <Expression>)
10                  | greaterthan(<Expression> <Expression>)
11                  | <Identifier>
12                  | <Value>
13                  | 'not'(<Expression>)
14 <Value>        ::= <Integer>|<Boolean>
15 <Identifier>   ::= <Letter>{<Letter>}
16 <Letter>       ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
17 <Integer>      ::= <NonzeroDigit>{<Digit>}
18 <Digit>        ::= 0|<NonzeroDigit>
19 <NonzeroDigit> ::= 1|2|3|4|5|6|7|8|9
20 <Boolean>      ::= true|false
```

Figure 1: Syntax for the semantic tree.

```
<State> ::= state(<Program> <Store>)
<Store> ::= bind(<Identifier> <Value> <Store>)|empty
```

Figure 2: Syntax for representing execution states.

Appendix : Interpreter for the semantic trees

```
declare

fun {Execute State}
   case State
   of state(nil _) then [State]
   [] state(Statement|RestOfStack Store) then
      case Statement
      of assign(Identifier Expression) then
         State|{Execute state(RestOfStack
                             {Bind Identifier {Evaluate Expression Store} Store})}
      [] print(Expression) then
         {Browse {Evaluate Expression Store}}
         State|{Execute state(RestOfStack Store)}
      else raise invalidStatement(Statement) end
      end
   else raise invalidState(State) end
   end
end

fun {Evaluate Expr Store}
   case Expr
   of add(A B) then {Evaluate A Store}+{Evaluate B Store}
   [] subtract(A B) then {Evaluate A Store}-{Evaluate B Store}
   [] multiply(A B) then {Evaluate A Store}*{Evaluate B Store}
   [] divide(A B) then {Evaluate A Store} div {Evaluate B Store}
   [] equals(A B) then {Evaluate A Store}=={Evaluate B Store}
   [] greaterthan(A B) then {Evaluate A Store}>{Evaluate B Store}
   [] lessthan(A B) then {Evaluate A Store}<{Evaluate B Store}
   [] 'not'(A) then {Evaluate A Store}==false
   else if {IsAtom Expr} then {Lookup Expr Store}
        elseif {IsNumber Expr} orelse Expr==false orelse Expr==true then Expr
        else raise couldntEvaluate(Expr Store) end
        end
   end
end
```

```
44 fun {Bind Identifier Value Store}
45    case Store
46    of empty then bind(Identifier Value Store)
47    [] bind(OldIdentifier OldValue RestOfStore)
48    then if OldIdentifier==Identifier then bind(Identifier Value RestOfStore)
49         else bind(OldIdentifier OldValue {Bind Identifier Value RestOfStore})
50         end
51    else raise invalidStore(Store) end
52    end
53 end
54
55 fun {Lookup Identifier Store}
56    case Store
57    of empty then raise identifierNotInStore(Identifier Store) end
58    [] bind(BoundIdentifier Value RestOfStore) then
59       if BoundIdentifier==Identifier then Value
60       else {Lookup Identifier RestOfStore}
61       end
62    else raise invalidStore(Store) end
63    end
64 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