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

Exam in TDT4165 Programming Languages

(with solutions)
Tuesday, October 16., 2007, 12:00–14:00

Questions and Answers

Question 1

Consider the following grammar for a simple programming language:

```
               ⟨program⟩  ::=  ⟨statements⟩
            ⟨statements⟩  ::=  ⟨statement⟩
                            |  ⟨statement⟩ ⟨statements⟩
             ⟨statement⟩  ::=  ⟨assignment⟩
                            |  ⟨loop⟩
                            |  ⟨io⟩
            ⟨assignment⟩  ::=  ⟨identifier⟩ : ⟨expression⟩ .
                  ⟨loop⟩  ::=  loop ⟨identifier⟩ { ⟨statement⟩ } .
                    ⟨io⟩  ::=  in ⟨identifier⟩ .
                            |  out ⟨identifier⟩ .
            ⟨identifier⟩  ::=  [A-Za-z]+
            ⟨expression⟩  ::=  ⟨additive expression⟩
                            |  ⟨multiplicative expression⟩
                            |  ⟨identifier⟩
   ⟨additive expression⟩  ::=  ⟨expression⟩ + ⟨expression⟩
                            |  ⟨expression⟩ - ⟨expression⟩
⟨multiplicative expression⟩ ::= ⟨expression⟩ * ⟨expression⟩
                            |  ⟨expression⟩ / ⟨expression⟩
```

The definition of the nonterminal ⟨*identifier*⟩ is a regular expression that matches one or more letters not separated by any non-letter character. (This has no importance for the question.)

Which of the following is correct:

1. The grammar is ambiguous and left-recursive.
2. The grammar is ambiguous but not left-recursive.
3. The grammar is left recursive but not ambiguous.
4. The grammar is neither ambiguous nor left-recursive.

Answer: 1.

A grammar is ambiguous if it is possible to construct a sentence that is valid according to the grammar and that has more than one leftmost (or rightmost) derivation (more than one parse tree). The grammar above is ambiguous because of the definition of expressions. For example, a syntactically valid program including the code `A + B * C` would have at least two parse trees, in which the expression would be parsed differently:

> `+(A, *(B,C))`, or
>
> `*(+(A,C), D)`.

A grammar is left recursive if there is a nonterminal for which it is possible to make a replacement with that same nonterminal as the left-most symbol. That is, there is a nonterminal N for which there is a sequence of rules allowing to replace N with Nα, with α any sequence of terminals and nonterminals. The grammar above is (indirectly) left-recursive because of the definition of expressions. The nonterminal $\langle$*expression*$\rangle$ can be replaced with the sequence $\langle$*expression*$\rangle$ `+` $\langle$*expression*$\rangle$, for example, where $\langle$*expression*$\rangle$ is the left-most nonterminal.

Question 2

Consider the following two pieces of code:

A.
```
loop Index {
    in Input .
    Output : Input + Index .
    out Output .
} .
```

B.
```
in Input .
loop Input { Output : Output + 1 . } .
out Output .
```

Whitespace is added for readability; assume that the amount and type of whitespace between lexemes (tokens) has no importance for the validity of a piece of code.

Given the grammar from Question 1, which of the following is true:

1. According to the grammar, A is syntactically valid but B is not.
2. According to the grammar, B is syntactically valid but A is not.
3. According to the grammar, both A and B are syntactically valid.
4. According to the grammar, neither A nor B are syntactically valid.

Answer: 4.

While reasoning about the validity of a sequence of tokens with respect to a formal grammar, you should use no additional assumptions, but rather follow the grammar in a machine-like manner.

A is not valid because a loop statement can contain only one nested statement; this is clear from the definition of $\langle$*loop*$\rangle$ and $\langle$*statement*$\rangle$.

B is not valid because the grammar does not allow to parse numerals; there is no rule which allow you to parse (or generate) sentences including the symbol `1`, and thus any sequence of tokens including that symbol is not a valid sentence according to the grammar above.

Question 3

Consider the following mathematical definition of Fibonacci numbers:

$$Fibonacci_n = \begin{cases} 1, & \text{if } n < 2 \\ Fibonacci_{n-1} + Fibonacci_{n-2}, & \text{otherwise} \end{cases}$$

The following code shows one possibility of implementing this definition in Haskell:

```
fib n | n < 2 = 1
      | otherwise = fib (n - 1) + fib (n - 2)
```

Given that Haskell is a programming language with lazy evaluation, which of the following is true:

1. The implementation above is not recursive, even though the mathematical definition is recursive.
2. The implementation above is not tail-recursive because it is not possible to implement a tail-recursive function in a language with lazy evaluation.
3. The implementation above is not tail-recursive irrespectively of the order of evaluation.
4. The implementation above is tail-recursive because of the lazy evaluation.

You do not need to know Haskell to answer correctly.

Answer: 3.

A function (procedure) is recursive if it calls itself inside its body. A function (procedure) is tail-recursive if it calls itself in its own body (i.e., if it is recursive) and if there is just one recursive call which is the last statement in the body.[1]

Question 4

Consider the following function:

```
fun {Switch Number Threshold Result Rescue}
   if Number<Threshold then Result
   else Rescue end
end
```

Suppose that `Switch` is applied as follows:

```
Result = {Switch N 5 {Fibonacci N} {Fibonacci 5}} +
         {Switch 5 N {Fibonacci 5} {Fibonacci N}}
```

where `Fibonacci` is a function that given a number n returns the n-th Fibonacci number. You do not know how `Fibonacci` is implemented, but assume that it computes Fibonacci numbers with runtime linear in terms of the input, i.e., a computation of `{Fibonacci 10}` takes $10t$, `{Fibonacci 50}` takes $50t$, etc., for some constant t.[2]

Assuming that the computation cost added by `Switch` is constant and negligible, which of the following is true:

[1] True for direct recursion. Functions can also be indirectly recursive (and tail-recursive), if they do not call themselves, but call other functions (procedures) that call the first again. Also, a tail-recursive procedure can actually include more than one call ti itself in the body, provided that the calls are placed, e.g., in different branches of a conditional statement.

[2] The assumption is reasonable for an appropriate implementation and sufficiently small inputs.

1. For any input n, it is guaranteed that an execution of this code will take $(10 + 2n)t$, irrespectively of how `Fibonacci` is implemented.
2. It is guaranteed that an execution of this code can't take more than $10t$, irrespectively of how `Fibonacci` is implemented and of the input n.
3. It is guaranteed that an execution of this code can't take more than $2nt$, irrespectively of how `Fibonacci` is implemented and of the input n.
4. None of the above.

(Note: n is the value of `N`.)

Answer: 4.

`Fibonacci` may be an eager or a lazy function.

- If `Fibonacci` is eager, both `{Fibonacci 5}` and `{Fibonacci N}` will be evaluated twice, with runtime $2(5+n)t$. (If `Fibonacci` were memoized, the runtime would rather be $\max(5t, nt)$, assuming constant-time cost of memoization. However, if `Fibonacci` were memoized, the computation of `{Fibonacci n}` would take nt on the first occurrence, but constant time on every other occurrence, which does not meet the specification above.)
- If `Fibonacci` is lazy, only one of `{Fibonacci 5}` and `{Fibonacci N}` will be evaluated (but necessarily twice), with runtime $2\times5t$ or $2nt$, respectively. (If `Fibonacci` were memoized, the runtime would be $5t$ or nt, respectively.) Which of them will be evaluated depends on the value of n, and the computation will never take more than $2 \times 5t$ and $2nt$, for any n.

Since it is not known how `Fibonacci` is implemented, none of the above can be guaranteed.

Question 5

Consider the following two definitions:

A.
```
fun {Reverse List}
   fun {Reverse List Accumulator}
      Head|Tail = List in
      {Reverse Tail Head|Accumulator}
   end
in
   {Reverse List nil}
end
```

B.
```
fun {Reverse List}
   case List of Head|Tail
   then {Reverse Tail}|Head
   else nil
end
```

Which of the following is true:

1. A defines constant-stack computation, but B defines linear-stack computation.
2. Both A and B define constant-stack computation.
3. Both A and B define linear-stack computation.
4. B defines constant-stack computation, but A defines linear-stack computation.

(Here 'constant' and 'linear' refer to the length of the input list.) Hint: translate these definitions into the kernel language.

Answer: 2.

In Oz, an execution of a tail-recursive procedure (or function) leads to an iterative process, a computation with constant stack size. A is explicitly tail-recursive. B seems non-tail-recursive, but its translation to the kernel language is tail-recursive:

```
proc {Reverse List ?Result}
   case List of Head|Tail then
      local Rest in
         Result = Rest|Head
         {Reverse Rest}
      end
   else nil end
end
```

Note that even though A is not a correct implementation of `Reverse` and it will never return a correct result (see below), for any input list there will be an actual linear-stack computation done before a unification failure is met.

Question 6

Consider the two definitions of `Reverse` above. Which of the following is true:

1. A correctly computes the reverse of an input list, but B does not.
2. Both A and B correctly compute the reverse of an input list.
3. B correctly computes the reverse of an input list, but A does not.
4. Neither A nor B correctly compute the reverse of an input list.

(Here, 'reverse' means a list with the same elements as the original list, but in the opposite order, e.g., `{Reverse [1 2 3]} == [3 2 1]`.)

Answer: 4.

A is an almost correct implementation, except for that it attempts to unify `List` with a record `Head|Tail`, rather than match `List` against the pattern `Head|Tail`. When `List` is `nil`, this results in a unification failure. Since every list is a sequence of nested records ending in the empty list `nil`, will `Reverse` fail with any input list.

B is incorrect: except for the trivial case where the input is `nil`, it does not produce a list but rather a series of head-nested records, each with an element of the input list in the tail. It will return a list if the input is a nested list, but this still will not be the correct result:

`{Reverse [1 2 3]}` returns `((nil|3)|2)|1` rathen than `[3 2 1]`;

`{Reverse [[1] [2] [3]]}` returns `[[[nil 3] 2] 1]` rather than `[[3] [2] [1]]`.

Question 7

Consider the following program:

```
local
   fun {MakeSender Message Stream}
      proc {Send Stream} Rest in
         Stream = Message|Rest
         thread {Send Rest} end
      end
   in
      sender(start:proc {$} {Send Stream} end)
   end
   fun {MakeReceiver Stream}
      proc {Receive Stream}
         case Stream of Message|Stream
         then {Browse Message} {Receive Stream} end
      end
   in
      receiver(start:proc {$} {Receive Stream} end)
   end
   Stream
   [S1 S2 R] = [{MakeSender s1 Stream}
                {MakeSender s2 Stream}
                {MakeReceiver Stream}]
in
   {R.start}
   {S1.start}
   {S2.start}
end
```

Which of the following is true about an execution of this program:

1. None of the answers below is correct.
2. The execution will crash due to a unification failure.
3. The execution will repeatedly print `s1` and `s2` in the browser window, in an apparently random order.
4. The execution will repeatedly print `s1` in the browser window.

Answer: 1.

The receiver is started first, and freezes over the unbound variable `Stream`. Since the receiver runs in the main thread, it suspends the whole program — none of the senders is started.

The situation would be different if the receiver were started in a separate thread, e.g.,

- if `{R.start}` were replaced with `thread {R.start} end`, or
- if `receiver(start:proc {$} {Receive Stream} end)` were replaced with `receiver(start:proc {$} thread {Receive Stream} end end)`, etc.

then both senders would have a chance to start. One of them would add an element to the stream, and the other would cause a unification failure. The failure would stop the main thread, since the first sending of each sender is performed in that thread. Depending on the scheduling, the following could happen:

- If the second sender causes the failure after the first has started a new thread, the first sender will keep sending its message, which will be then printed by the receiver (running in a separate thread — see the assumption above).
- If the second sender causes the failure before the first has started a new thread, the first sender will stop as well (it still runs in the main thread). Only the first message will be displayed.

Question 8

Let us extend Oz with a new type of statement with the following syntax:

`{Bound` $\langle id\rangle_1$ $\langle id\rangle_2$`}`

and the following semantics:

The semantic statement is:

(`{Bound` $\langle id\rangle_1$ $\langle id\rangle_2$`}`, `E`)

The execution rule is:.

- If `E`($\langle id\rangle_1$) is a bound variable, then bind `E`($\langle id\rangle_2$) to `true`.
- If `E`($\langle id\rangle_1$) is an unbound variable, then bind `E`($\langle id\rangle_2$) to `false`.

Consider the declarative sequential model of computation (DS), the declarative concurrent model with data-driven computation (DCDta), and the declarative concurrent model with demand-driven computation (DC-Dem). Which of the following is true:

1. Adding this statement type has no influence on the declarativeness of any of the models.
2. Adding this statement type will cause DCDta and DCDem, but not DS, to become non-declarative.
3. Adding this statement type will cause DS, DCDta, and DCDem to become non-declarative.
4. Adding this statement type will cause DS, but not DCDta and DCDem, to become non-declarative.

Answer: 2.

Adding this statement type has no influence on the declarativeness of the sequential model:

- Any program without any `Bound` statement will give the same result in DS and DS+`Bound`, and thus DS+`Bound` must be declarative.
- Any program with a `Bound` statement will not be valid in DS, but will give the same result on any execution in DS+`Bound` – if `E`($\langle id\rangle_1$) is bound, then it is bound on any execution, and thus `E`($\langle id\rangle_2$) is `true` on every execution; analogously if `E`($\langle id\rangle_1$) is unbound.

Adding this statement type to a declarative concurrent model (DCDta or DCDem) will cause the model to become non-declarative. With `Bound`, it is possible to write a program that will give different results on different executions. For example:

```
local X in
   thread if {Bound X} then skip else X = 1 end
   thread if {Bound X} then skip else X = 2 end
   ...  % X is bound to 1 or 2 (or even unbound), depending on the schedulling
end
```

Question 9

C is a language with strict-order evaluation: expressions given as arguments in a function call are evaluated before the resulting values are used in the function body. To simulate non-strict evaluation in which expressions are not evaluated unless their values are needed, C progammers use *macros*. The following are two ways of computing the cube of a number, one using a macro and one using a function:

```
...
#define cubem(a) a*a*a

long cubef(long a) {
   return a*a*a;
}

...
```

For example, both `cubem(2)` and `cubef(2)` result in the number 8. If included in a program's code, the application `cubem(`⟨*expression*⟩`)` is replaced by the expression ⟨*expression*⟩`*`⟨*expression*⟩`*`⟨*expression*⟩:

```
...
long y = cubem(1234567);
...
```

is replaced by the code:

```
...
long y = 1234567*1234567*1234567;
...
```

Given that C is a language with strict evaluation, which of the following is true:

1. For any argument, an application of `cubem` will always return the same value as the application of `cubef`.
2. For some arguments, an application of `cubem` will return the same value as an application of `cubef`, for all other arguments at least one of the applications will fail (there will be an error condition reported in some way).
3. For some arguments, an application of `cubem` will return the same value as an application of `cubef`, for all other arguments the results will differ or at least one of the applications will fail.
4. None of the above statements is correct.

Consider only declarative programs, i.e., programs without side effects (e.g., changing the value of a variable), so that `++x`, `x+=1`, etc., are not valid arguments, but `x`, `x+1`, etc., are.

Answer: 3.

Arguments which are arithmetic expressions including operators other than `*` may lead to different results of an evaluation of expressions involving `cubem` and `cubef`.

It is possible for both `cubef` and `cubem` to give the same result for some arguments, e.g.,

> `cubef(2)` evaluates to 8, as `cubem(2)` does;
>
> `cubef(2*1)` evaluates to 8, as `cubem(2*1)` does, etc.

It is possible for `cubem` and `cubef` to give different results for some arguments, e.g., `cubef(1+1)` evaluates to 8 (because `1+1` is evaluated before it is passed to `cubef`), while `cubem(1+1)` evaluates to 4 (`cubem(1+1)` is replaced with `1+1*1+1*1+1`).

It is also possible that for some arguments one of or both of `cubef` and `cubem` will lead to a failure, e.g., due to type conversion error or value overflow, depending on the context and compilation. For example, `cubem(10000LL)` will compute the cube of the number 10000 represented as a `long long` value, while `cubef(10000LL)` will compute the cube of the number 10000 represented as a `long` value, which will either give an incorrect value (since 10000^3 does not fit into a `long`) or result in a failure (if overflow detection is turned on).

Note that this has no influence on which answer is correct.

Question 10

Most programming languages include facilities for reporting and recovering from abnormal conditions. Typically, they provide some form of the try-catch-finally statement type, where the content of the `finally` clause is executed irrespectively of whether any statement inside the `try` clause raises an exception, and irrespectively of whether such an exception is caught by the `catch` clause. For example, the following program:

```
try
   {Browse raising(exception)}
   raise exception end
catch Exception then
   {Browse caught(Exception)}
finally
   {Browse finalizing}
end
```

results in `raising(exception)`, `caught(exception)`, and `finalizing` displayed in the browser window.

Let us consider the Oz kernel language that provides the try-catch form, with no `finally` clause. Which of the following is true:

1. It is impossible to achieve the effect of the full try-catch-finally statement directly in the kernel language, and it is impossible to extend the kernel language to provide support for the `finally` clause.
2. It is impossible to achieve the effect of the full try-catch-finally statement directly in the kernel language as it is defined, and thus it is impossible to define a practical language with the `finally` clause, unless one modifies the kernel language itself.
3. It is impossible to achieve the effect of the full try-catch-finally statement directly in the kernel language as it is defined, but it is possible to define a practical language by means of translation into the kernel language, such that the practical language provides the `finally` clause.
4. It is possible to achieve the effect of the full try-catch-finally statement directly in the kernel language, without the need for any extension of the syntax and semantics.

Answer: 4.

The try-catch-finally statement of the practical langauge:

```
try ⟨statement⟩_try
catch ⟨id⟩_exception then ⟨statement⟩_catch
finally ⟨statement⟩_finally
end
```

can be translated into the kernel language as:

```
local ⟨id⟩tag in
   try
      try ⟨statement⟩try catch ⟨id⟩exception then ⟨statement⟩catch end
      ⟨id⟩tag = clean
   catch ⟨id⟩exception then ⟨id⟩tag = ⟨id⟩exception end
   ⟨statement⟩finally
   if ⟨id⟩tag \= clean then raise ⟨id⟩tag end
end
```

Note: you were not required to know how this can be achieved. To answer this question correctly it was enough to know that the kernel language does not include the `finally` keyword (this was explicit in the text), that the practical language does allow for try-catch-finally statements, and that the practical language is defined by means of translation to the kernel language without extending its expressivity. Since the practical language has the same expressivity as the kernel language, the kernel language *must* be expressive enough for the desired effect without additional syntactic or semantic extensions.