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)
Wednesday, 15. October, 2008

Task 1

Solutions

Subtask 1-1. A recursive procedure is a procedure that contains a call to itself within its own body.

The definition can be extended to include procedures that do not call themselves, but rather call other procedures that make back-calls to the original procedures (which is typically called 'mutual recursion'). We might also want to include procedures that do not call themselves, but which call other procedures that are recursive (by any of the definitions above; we could call this pattern 'forwarding recursion').

The first definition above demands solely that the procedure in question contains a call to itself in its own body (and analogously for the other two versions). You can always ask if the definition is not too permissive; for example, is the following a recursive procedure?

```
proc {JudgeMe}
   if false then {JudgeMe}
   else skip end
end
```

Subtask 1-2. Both recursive and iterative processes are processes that are (potentially) composed of subprocesses that are identical modulo the particular input and output values, i.e., they are processes that have repetitively executed subprocesses as parts.

According to CTMCP (and many other books in the field), a *recursive process* is a process in which each subsequent repetition of a subprocess must return the control of execution to its caller, the preceding subprocess, while an *iterative process* is a process in which each subsequent subprocess is potentially able to return the control to the original caller that started the whole repetitive process.

In practice, this means that a recursive process requires each subprocess to store a pointer to its parent caller, and that there must thus be a complete chain of such pointers (and thus also call frames for all respective calls) stored until the last subprocess in the sequence will actually return, and the chain will be 'consumed' backwards. In an iterative process, as soon as a subprocess is started the call frame of parent subprocess can be removed from memory, since the child subprocess has a back-pointer to the original caller rather than to its parent.

This means, in turn, that the length (in terms of the number of recursive calls) of a recursive process is bounded by the available memory, while an iterative process may, in principle, run forever. Another consequence is that an iterative process can be interrupted at any time, and it requires only a fixed number of parameters to be stored in order to restart the process from where it was stopped. For a recursive process this would be much more complicated, and the amount of information that would have to be stored has size linear in terms of the recursive calls invoked so far.

Subtask 1-3. We could think of an iterative procedure as a procedure that either a) is recursive and the process it specifies is iterative; or b) as a procedure that is not recursive but rather uses an iterative control structure such as a `for` or `while` loop with explicit update of a set of mutable state variables. (The term 'iterative procedure' is not very common in the literature, however.)

Subtask 1-4.

```
fun {Iterate State IsFinal Update}
   if {IsFinal State}
   then State
   else {Iterate {Update State} IsFinal Update}
   end
end
```

(source file: `code/iterate.oz`)

Task 2

Solutions

Subtask 2-1.

```
fun {RunningSum List}
   fun {RunningSum List Sum}
      case List
      of nil then nil
      [] Head|Tail then
         (Sum+Head)|{RunningSum Tail Sum+Head} end
   end
in
   {RunningSum List 0}
end
```

(source file: `code/runningsum-recursive.oz`)

Subtask 2-2.

```
fun {RunningSum List}
   nil#_#Sums#nil =
   {Iterate
    local End in List#0#End#End end
    fun {$ List#_#_#_}
       List == nil
    end
    fun {$ (Head|Tail)#Sum#Start#End}
       NewEnd in
       End = (Sum+Head)|NewEnd
       Tail#(Sum+Head)#Start#NewEnd
    end}
in
   Sums
end
```

(source file: `code/runningsum-iterative.oz`)

Note: your solutions did not have to be identical with the ones above. You can execute the file `code/runningsum-test.oz` to perform a simple test of the implementation.

Task 3

Solutions

Subtask 3-1. (a) Procedural abstraction: an approach to programming in which recurring patterns are wrapped into procedure definitions and can then be reused by making procedure calls instead of repeating the code, which is a tedious and error-prone activity.

(b) Genericity: a property that characterises a procedure that can be used to do a number of different, though related tasks; part of the functionality is specified in the procedure body, another part is specified as arguments at the time the procedure is called. In higher-order programming this typically means passing procedures (e.g., comparators) as arguments to other, generic procedures (e.g., sorters).

(c) Instantiation: a way to create multiple procedures with related, but possibly different in details functionality, by means of applying a generic procedure (a procedure-factory) to different arguments and having it return more specific procedures (which are then regarded its instances, hence 'instantiation').

(d) Embedding: an approach to programming in which procedures are stored as elements of data structures just like any other kind of first-class objects. Embedding is useful for conditional application of a large number of different procedures that otherwise would have to be called manually.

Task 4

Solutions

Subtask 4-1. A closure is an object that consists of a procedure (a function) code and an environment which contains mappings for the variables referenced to within the procedure body, but not introduced (declared) there. When the closure's procedure is called, the variables that are not declared within the procedure body are looked-up through the closure's environment rather than through the environment in which the procedure is called. (At least, this is how things are in lexically scoped languages.)

Subtask 4-2. The purpose of the closure environment is to preserve the bindings available in the environment where the procedure was defined, and which might no longer be accessible otherwise (e.g., when the procedure is called outside of the environment in which it was defined).

Subtask 4-3. In Oz, closure environments are created when closure are created, i.e., when a procedure value-creation expression (`proc ... end`) or a function value-creation expression (`fun ... end`) is evaluated. A closure environment will then contain mappings for those variable names which appear, in at least one place, in the procedure body (but not in the parameter list) as free (undeclared) identifiers.

Task 5

Solutions

Subtask 5-1. (a) An ADT can be implemented as a declarative (immutable) or a non-declarative (mutable) data structure. The former will always have the same content, the latter may change the content while preserving its own identity.

(b) An ADT can be implemented as a bundled (object-like) or unbundled data structure. The former will contain both data and methods to access the data, the latter will contain only the data and the ADT's operations will be implemented as separate procedures to be applied from outside.

(c) An ADT can be implemented as a secure (closed) or as an insecure (open) data structure. The former will not allow any unauthorised (not specified by the ADT and implemented accordingly) procedure to access the data, while the latter will expose its content so that any procedure can be applied to it.

Subtask 5-2. (a) Unbundled and insecure data structures are commonly used in languages such as C, where the data are contained within arrays and structs, and operations are implemented as separate procedures.

(b) Unbundled but secure data structures are an improvement over unbundled insecure d.s., but are more difficult to implement and are rather uncommon, especially where there's a need for quick prototyping (because security may be inessential) or for large-scale implementations (because object-oriented programming with bundled and secure d.s. is much more convenient).

(c) Bundled but insecure data structures are convenient for quick prototyping of applications with larger amounts of code (because of better support for modularity) where security is inessential (e,g., in scientific applications). Python is an excellent example of a language with support for bundled but insecure d.s.

(d) Bundled and secure d.s. are most useful in production-quality code, where security is an essential property of the implemented system.

All these variants can be implemented with or without mutable state; if mutable state is not included, security is often inessential in that the data cannot be corrupted (except for inappropriate binding of unbound variables in Oz).

Task 6

Solutions

Subtask 6-1. A language is lexically scoped if free identifiers inside a procedure's body at the time of its being called are looked up in the environment in which the procedure was defined (which is typically done via a closure environment).

A language is dynamically scoped if free identifiers inside a procedure's body at the time of its being called are looked up the environment in which it is being called. A language can be said to be dynamically scoped also when variable declarations inside a procedure body temporarily change the values of external variables looked up in the usual way. The former is typical of languages such as Emacs Lisp and newLISP, while the latter is typical of Perl's dynamic scoping with the `local` variable qualifier or Scheme's `fluid-let`.

For your interest, execute the following Perl code[1] and try to interpret the results:

```
#!/usr/bin/perl

$global = 1;
$hard = \$global;
$soft = "global";

{
   my $global = 2;
   print "[my]\thard: $$hard, soft: $$soft, global: $global\n";
}
print "[out]\tglobal: $global\n";

{
   local $global = 2;
```

[1] `perl scope.pl`.

```
   print "[local]\thard: $$hard, soft: $$soft, global: $global\n";
}
print "[out]\tglobal: $global\n";

{
   our $global = 2;
   print "[our]\thard: $$hard, soft: $$soft, global: $global\n";
}
print "[out]\tglobal: $global\n"
```

(source file: `code/references.pl`)

Subtask 6-2. The answer may depend on how the dynamic scoping is designed and implemented, but we can take a guess. We would have `P` be a procedure with a free identifier inside its body which would be looked-up where `P` is called. When `P` is called, `X` is 3, and `Y` is 4, but inside `P`'s body `Y` is not free, so that only `X` would be looked-up, and `P` would print 5 (3+2).

To convince yourself, execute the following newLISP code[2] and note the output:

```
(let ((p (let ((x 1))
            (lambda ()
               (let ((y 2)) (println (+ x y)))))))
   (let ((x 3) (y 4))
      (p)))
```

(source file: `code/scope.lsp`)

(You can try it in some variant of Scheme, just replace `println` with `display`; compare the results.)

Here's a similar piece of code in Perl, where we have to enforce dynamic typing:

```
#!/usr/bin/perl

{
   local $p;
   {
      local $x = 1;
      $p = sub { local $y = 2; print $x+$y }
   }
   {
      local $x = 3;
      local $y = 3;
      &$p;
   }
}
```

(source file: `code/scope.pl`)

Task 7

Solutions

Subtask 7-1. The code doesn't work because the call to `Integers` inside the `Browse` statement keeps producing the stream, and the `Browse` cannot be applied to the result before `{Integers}` returns (and it won't return).

Subtask 7-2. There are two ways of fixing the problem:

[2]`newlisp scope.lsp.`

(a) we can use concurrency and a few `thread ... end` statements to make things happen in separate thread, so that `Browse` can start displaying as soon as something has been produced;

(b) we can use lazy evaluation to make the involved functions produce the streams only when needed, and have the production enforced inside the `Browse` statement.

Subtask 7-3. Here is how `Integers` could be redefined with concurrency:

```
fun {Integers}
   fun {Generate Start Step}
      {Delay 1000}
      Start|{Generate Start+Step Step}
   end
in
   thread {Map thread {Generate 1 1} end
           fun {$ Start}
              thread {Generate Start Start} end
           end}
   end
end
```

(source file: `code/integers-concurrent.oz`)

You can test the code by executing the file `code/integers-test.oz`. (The `{Delay 1000}` statement above is needed if you actually want to see the output; without it, it is theoretically possible for `Browse` to display some output, but there are so many threads being created that `Browse` has no chance to start displaying. However, the `Delay` statement is not considered an essential part of the solution.)

While it is possbile to make `Generate` a lazy function and thus avoid including its applications in separate threads, we would need to force it to produce the nested streams by, e.g., enclosing it in a non-transforming application of `Map`, which then would have to run in separate threads. Here's how this could be done:

```
fun {Integers}
   fun lazy {Generate Start Step}
      Start|{Generate Start+Step Step}
   end
in
   thread {Map {Generate 1 1}
           fun {$ Start}
              thread {Map {Generate Start Start}
                          fun {$ X} X end}
              end
           end}
   end
end
```

(source file: `code/integers-lazy.oz`)

Subtask 7-4. The minimal number of modifications is 3: we need to add three `thread ... end` statements as in the first solution above. (The other solution is more invasive.) We can't drop the outermost `thread` statement, because `Browse` would not be able to start. We can't drop the `thread` statement encolsing `{Generate 1 1}`, because `Map` would not be able to start. We can't drop the third `thread` statement, because `Map` would not be able to proceed to the second element of its input. No more modifications are needed to make the code work.

Task 8

Solutions

Subtask 8-1. Interpretation is the process of executing a program directly from the source code, in the sense that no other code is produced before the program can be executed. (Clearly, the program may be executed statement by statement, or may be tokenized and parsed into an intermediate representation and only then executed, but the intermediate representation is internal to the interpreter and is not output as a translated program.)

Subtask 8-2. Compilation is the process of translating a program from one form to another, typically to the native code of the underlying architecture or the byte code of some virtual machine interpreter.

Subtask 8-3. Typically, both interpretation and compilation require that the program be:

(a) analyzed lexically, i.e., split into a sequence of lexemes;
(b) tokenized, i.e., have all the lexemes classified;
(c) analyzed syntactically, i.e., be parsed into an intermediate representation.

In addition, both interpreted and compiled programs may be first preprocessed, i.e., have comments removed, macros expanded, etc.

Task 9

Solutions

Subtask 9-1. A grammar is context-free if all its rules are of the form $\alpha ::= \gamma$, where α is a single non-terminal and γ is any sequence of non-terminals and terminals.

Subtask 9-2. The grammar above is context free, since all four rules, when expanded to the BNF form, comply with the specification for context-free grammars:

⟨*statement*⟩	::=	`NOOP`
⟨*statement*⟩	::=	⟨*if*⟩
⟨*if*⟩	::=	`IF` ⟨*expression*⟩ `THEN` ⟨*statement*⟩
⟨*if*⟩	::=	`IF` ⟨*expression*⟩ `THEN` ⟨*statement*⟩ ⟨*else*⟩
⟨*expression*⟩	::=	`YES`
⟨*expression*⟩	::=	`NO`
⟨*else*⟩	::=	`ELSE` ⟨*statement*⟩

Subtask 9-3. A grammar is unambiguous if each sequence of terminals that can be generated by the grammar has exactly one parse tree (exactly one left- or right-most derivation).

Subtask 9-4. The grammar above is not unambiguous; for example, the followin are two ways of parsing the same sequence `IF YES THEN IF NO THEN NOOP ELSE NOOP`:

-
  ```
  statement(IF expression(YES)
            THEN statement(IF expression(NO)
                           THEN statement(NOOP)
                           ELSE statement(NOOP)))
  ```

-
  ```
  statement(IF expression(YES)
            THEN statement(IF expression(NO)
                           THEN statement(NOOP))
            ELSE statement(NOOP))
  ```

— End —