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

Exam in TDT4165
Programming Languges

(with solutions)
Monday, December 3., 2007

Prepared by: Wacek Kuśnierczyk

Reviewed by: Ole Edsberg

General Comments and Hints

Read the following general comments and the rest of the exam text **before** you begin to answer.

- Throughout the text, the acronym 'CTMCP' is used to refer to the pensum book (P. van Roy and S. Haridi: Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004).
- The exam is composed of 5 tasks, each task contains one or more questions. To achieve the maximal score, you need to correctly answer all questions in all tasks. If you skip or answer incorrectly a question, your score will be reduced correspondingly. Different tasks contribute differently to the total score; see each task for details.
- All code examples are given in Oz. Where you are asked to write code, the code should be in Oz as well.
- You answers should be concise, without text that is irrelevant or does not contribute to the answer.
- Each answer should include a brief explanation; rather than simply 'yes', you should answer 'yes, because ...'. Correct answers with no justification will score less than answers with justifications. Incorrect answers with coherent explanations may still give you some points. You may also disagree with what is stated in CTMCP or with what was explained during the lectures, but you should give convincing arguments in such cases.

Task 1 (10 points)

Consider an execution of the following program:

```
local X Y Z in
   X = Y
   try
      X = 1 Y = 2 Z = 3
   catch Exception then
      skip
   end
   {Browse X#Y#Z}
end
```

Question 1-1. Will there be any output printed in the browser window, and if yes, what will be displayed?

Question 1-2. Explain how the try-catch statement is executed on the abstract machine, and use this explanation to justify your answer to **Question 1-1**. (You may provide a formal specification of the abstract machine semantics for try-catch statements, but a clear free-text description will be sufficient).

Solutions

Question 1-1. There will be output displayed in the browser window: `1#1#Z` (or `1#1#_`) will be displayed.

Question 1-2. For a detailed description of the try-catch statement, see Sec. 2.6 in CTMCP. In general, if an exception is thrown, the stack corresponding to the current thread (there is just one thread in the example above) is unwinded, and the first catch statement met is executed; however, bindings in the single assignment store are *not* undone. Therefore, in the program above, `X` becomes bound with `Y`, then `X` becomes bound to `1`, then the unification failure while trying to bind `Y` to `2` causes an exception to be thrown so that `Z = 3` is not reached. The exception is caught and processed (i.e., the `skip` statement is executed). `X` remains bound to `1`, `Y` to `X`, and `Z` remains unbound when the `Browse` statement is reached.

Task 2 (30 points)

Memoization is a technique used to improve the efficiency of programs. A memoized function remembers the results of the computation for all those inputs it has already been called with earlier. When it is called again with an input which it has already been called with earlier, the function returns the already computed result from memory instead of calculating it anew.

Consider the function `Fibonacci`, implemented as follows:

```
fun {Fibonacci N}
   if N<3 then 1
   else {Fibonacci N-1} + {Fibonacci N-2} end
end
```

Question 2-1. Show how to implement a memoized version of the function (name it `FibonacciM`). Your implementation should differ from the above just in that memoization is added. You should use the same algorithm, rather than optimize the computation with, e.g., tail-recursion. `FibonacciM` should be a one-parameter function, to be called in exactly the same way as `Fibonacci` (i.e., `{FibonacciM N}`).

`FibonacciM` should behave as follows:

- For any positive integer argument, FibonacciM computes the correct result (see the implementation above; for any valid N, {FibonacciM N} == {Fibonacci N} is true). For any other input, the behaviour is unspecified (you may simply ignore invalid arguments, as it is done in Fibnonacci above).
- When FibonacciM is applied to some argument N for the first time, the execution of {FibonacciM N} takes approximately as much time as an execution of {Fibonacci N} would take.
- With a sufficiently large input value N (say, 40), when {FibonacciM N} is executed for the second (and any subsequent) time the runtime is considerably shorter than that of an execution of {Fibonacci N}.

For example, the following might be observed:

```
{Browse {Fibonacci 40}}    % takes 20 seconds
{Browse {Fibonacci 40}}    % takes 20 seconds
{Browse {FibonacciM 40}}   % takes 20 seconds
{Browse {FibonacciM 40}}   % takes <1 second
```

Hint: use explicit state.

Question 2-2. What are the benefits of using memoized functions?

Question 2-3. Are there any drawbacks of memoizing functions? Can there be situations in which a memoized function performs slower than a corresponding non-memoized version? Can there be situations in which using memoized functions may lead to wrong (unexpected, undesirable) computations?

The following is a skeleton code for the implementation of a function that takes as an argument a one-parameter function and returns its memoized version:

```
fun {Memoize Function}
   Memory = ...
   ...
in
   fun {$ Argument}
      ...
   end
end
```

For any one-parameter function F and any input X, both {F X} and {{Memoize F} X} return the same value (or both calls fail). For example, the call {Memoize Factorial} returns a memoized version of the function Factorial; for any non-negative N, {Factorial N} == {{Memoize Factorial} N} is true.

Question 2-4. Implement Memoize.

Question 2-5. Compare FibonacciM as defined above with the function returned by {Memoize Fibonacci}. Does the latter perform in the same way as the former? If not, why?

Question 2-6. Suppose that the execution of {Fibonacci 40} takes 20 seconds, and also that subsequent calls to FibonacciM with the same argument take less than 1 second. In the example below, the execution times of some of the calls are given in the comments. Give approximate execution times for the remaining calls (replace the question marks with your estimates).

```
{Browse {Fibonacci 40}}                               % takes 20 seconds
{Browse {Fibonacci 40}}                               % takes 20 seconds
{Browse {Fibonacci 39}}                               % takes 13 seconds

{Browse {FibonacciM 40}}                              % takes 20 seconds
{Browse {FibonacciM 40}}                              % takes <1 second
```

```
{Browse {FibonacciM 39}}                               % takes ?? seconds

{Browse {{Memoize Fibonacci} 40}}                      % takes ?? seconds
{Browse {{Memoize Fibonacci} 40}}                      % takes ?? seconds
{Browse {{Memoize Fibonacci} 39}}                      % takes ?? seconds

FibonacciMemoized = {Memoize Fibonacci}
{Browse {FibonacciMemoized 40}}                        % takes ?? seconds
{Browse {FibonacciMemoized 40}}                        % takes ?? seconds
{Browse {FibonacciMemoized 39}}                        % takes ?? seconds
```

Your answer should be consistent with your implementation of `FibonacciM` and `Memoize`.

Question 2-7. What is the result of the call `{Memoize Memoize}`?

Solutions

Question 2-1. Different solutions were possible. A simple solution following the specifications is:

```
FibonacciM =
local
   Memory = {NewCell nil}
   fun {Recall N Result|Results}
      case Result of !N#Result then Result
      else {Recall N Results} end
   end
in
   fun {$ N}
      try {Recall N Memory}
      catch _ then
         Result = {Fibonacci N}
         Memory := (N#Result)|@Memory
         Result
      end
   end
end
```

This solution uses explicit state (a cell) to keep a list of argument-result pairs from precious computations. (The use of exceptions is inessential to the solution; since the result of `Fibonacci` is always a number, `Recall` might return `nil` if the argument is seen for the first time, rather than cause an exception to be thrown.) Note that when the result has to be actually computed, `FibonacciM` calls `Fibonacci`. A better solution (which would not, however, fully realize the specifications)[1] would be to have `FibonacciM` call itself recursively. In that case, `FibonacciM` would memoize not only the result for the particular input with which it is called, but also results for all smaller inputs, e.g.:

```
FibonacciM =
local
   Memory = {NewCell nil}
   fun {Recall N Results}
      case Results of nil then nil
      [] (!N#Result)|_ then Result
      [] _|Results then {Recall N Results} end
   end
```

[1]Specifically, the application of `FibonacciM` to 40 would not take 20 seconds even on the first occasion if `FibonacciM` called itself recursively; such an implementation would run linearly (at least on the first occasion) in terms of the input, since all results of recusrive calls are computed just once, due to recursive memoization. Such a solution was not considered incorrect.

```
in
   fun {$ N}
      if N < 3 then 1
      else Retrieved = {Recall N Memory} in
         if Retrieved == nil
         then Result in
            Result = {FibonacciM N-1} + {FibonacciM N-2}
            Memory := (N#Result)|@Memory
            Result
         else Retrieved
         end
      end
   end
end
```

The use of a plain list was not mandatory, you could have used your own implementation of a tree or a hash table, or the built-in `Dictionary` structure (in this case you were not required to remember the exact syntax), etc. (See below for an example using a dictionary).

An interesting and elegant alternative is to use lazy computation:

```
FibonacciM =
local
   fun lazy {MakeFibs N1 N2}
      N1|{MakeFibs N2 N1+N2}
   end
   Fibs = {MakeFibs 1 1}
in
   fun {$ N}
      {Nth Fibs N}
   end
end
```

Note that here again all recursive calls are memoized.

Question 2-2. The most important benefit of using a memoized function is that, for a given input, the computation necessary to obtain the result is done just once, on the first occasion. This may help in reducing the runtime of programs.

Question 2-3. If memoization is not directly supported in the language, using memoization requires the programmer to actually implement it, either for each individual function (as above), or, better, in the form of a generic memoizer (as below; implementing a memoizer for functions of arbitrary arity is a bit more involved in Oz, but not impossible). In some languages you can use libraries providing the memiozation functionality (e.g., `use Memoize` in Perl).

Memoization has its cost related to acessing the underlying data structure used to keep track of the results of previous computations. If the memoized function is called with a large number of different inputs and the actual computation quite simple, searching the result memory may take more time than the actual computation. If the memoized function is seldom called with the same argument more than once, memoization makes little sense. In some cases, the first call to a memoized function may take more time than a call to the non-memoized version; this is the case with the first implementation of `FibonacciM` above (it caches the result *in addition* to the usual computation), though not with the latter two implementations (these do recursive memoization which spares a lot of redundant computation).

Memoization makes sense if the function is declarative (it is an implementation of a function in the mathematical sense); it makes no sense to memoize non-declarative functions, e.g., a function that returns a random number given some specifications of the distribution (e.g., the minimum and maximum). Another example is a function that should return a new entity each time it is called, such as `New`: in the call `{New Class init}`, one would normally expect a *distinct* object to be returned each time `New` is called with the same class and initialization message.

Question 2-4. Here is a possible implementation of `Memoize`, using a dictionary:

```
fun {Memoize Function}
   Memory = {Dictionary.new}
in
   fun {$ Argument}
      if {Dictionary.member Memory Argument}
      then {Dictionary.get Memory Argument}
      else Result = {Function Argument} in
         {Dictionary.put Memory Argument Result}
         Result
      end
   end
end
```

Question 2-5. `FibonacciM` and the result of `{Memoize Fibonacci}` may, in general, differ in at least two respects:

- the efficiency of the underlying data structure — here the dictionary used in `Memoize` is much more efficient than the plain list in `FibonacciM` as above;
- the recursive call — here the first `FibonacciM` and the result of `{Memoize Fibonacci}` call the non-memoized `Fibonacci` for the actual computation, but the two other implementations of `FibonacciM` make better use of recursion. Note that `Memoize` does not know anything about the function it gets as an argument (beyond that it should be a one-parameter function), and it can return a function which memoizes only the arguments it was explicitly called with, and not the arguments used in recursive calls (since the recursion, if any, is realized by the original function).

Question 2-6.

```
{Browse {{Memoize Fibonacci} 40}}                 % takes 20 seconds
{Browse {{Memoize Fibonacci} 40}}                 % takes 20 seconds
{Browse {{Memoize Fibonacci} 39}}                 % takes 13 seconds

FibonacciMemoized = {Memoize Fibonacci}
{Browse {FibonacciMemoized 40}}                   % takes 20 seconds
{Browse {FibonacciMemoized 40}}                   % takes <1 second
{Browse {FibonacciMemoized 39}}                   % takes 13 seconds
```

Each call `{Memoize Fibonacci}` results in a *new* memoized version of `Fibonacci`; each such memoized function has its own memory, and does not reuse the results of calls to the other memoized functions. Furthermore, a function returned by `Memoize` remembers only results for the arguments with which it has previously been called, but not results for the recursive calls (in an actual computation), since the recursive calls are realized using the non-memoized version (see above). The call `{Browse {FibonacciMemoized 39}}` takes as much time as a call to the non-memoized version would take, despite `FibonacciMemoized` having previously been called with the input `40`.

Question 2-7. `{Memoize Memoize}` returns a memoized version of `Memoize`. Note that each call to `Memoize` with the same function as the argument returns a *distinct* memoized version of the function. That is,

```
{Memoize F} == {Memoize F}
```

is false for any function `F`, and thus the behaviour above. To the contrary, a memoized version of memoize will always return *the same* memoized version when the same function is passed to it as an argument. That is, in

```
MemoizedMemoize = {Memoize Memoize}
{Memoize F} == {Memoize F}
```

the equality test will be true for any function F. In this case (which was not covered by the question) we would thus have:

```
{Browse {{MemoizedMemoize Fibonacci} 40}}               % takes 20 seconds
{Browse {{MemoizeMemoized Fibonacci} 40}}               % takes <1 second
{Browse {{MemoizeMemoized Fibonacci} 39}}               % takes 13 seconds
```

Task 3 (25 points)

Consider the following code:

```
fun {Reactive Procedure}
   proc {$ Message}
      {Procedure Message}
   end
end

fun {Active Procedure}
   Stream
   Port = {NewPort Stream}
   proc {Process Message|Messages}
      {Procedure Message}
      {Process Messages}
   end
in thread {Process Stream} end
   proc {$ Message}
      {Send Port Message}
   end
end

fun {Hyperreactive Procedure}
   proc {$ Message}
      thread {Procedure Message} end
   end
end
```

The three functions can be described as follows:

- `Reactive` takes as an argument a procedure `Procedure`, and returns a *reactive object*[2] implemented as a procedure. The object, when called with a message `Message`, immediately applies `Procedure` to `Message`.
- `Active` takes as an argument a procedure `Procedure`, and returns an *active object* implemented as a procedure, a stream, and a threaded stream-processor. The object, when called with a message `Message`, appends `Message` to the stream by sending it to the port; messages are retrieved from the stream and `Procedure` is applied to them in a unique, separate thread.
- `Hyperreactive` takes as argument a procedure `Procedure`, and returns a *hyperreactive object* implemented as a procedure. The object, when called with a message `Message`, immediately starts a new thread in which the `Procedure` is applied to `Message`.

[2]The terminology is partially invented for this example, and may not correspond to any widely accepted nomenclature.

Suppose you have an object that you want to use to ping[3] two NTNU servers as follows:

```
Furu = 'furu.idi.ntnu.no'       % server names
Selje = 'selje.idi.ntnu.no'

{Browse pinging(Furu)}
{Pinger Furu}                   % start pinging furu
{Browse pinging(Selje)}
{Pinger Selje}                  % start pinging selje
```

Here, `Pinger` is a reactive, an active, or a hyperreactive object defined, respectively, as follows:

1. `Pinger = {Reactive Ping}`,
2. `Pinger = {Active Ping}`,
3. `Pinger = {Hyperreactive Ping}`.

The actual pinging is done by an application of the procedure `Ping` to a host name. `Ping`, given a host name, displays the name three times in the browser window, in intervals of one second:

```
proc {Ping Host}
   proc {Do Attempt}
      if Attempt < 3 then
         {Delay 1000}
         {Browse ping(Host)}
         {Do Attempt+1} end
   end
in {Do 0} end
```

Question 3-1. Predict the output in the case when `Ping` is reactive.

Question 3-2. Predict the output in the case when `Ping` is active.

Question 3-3. Predict the output in the case when `Ping` is hyperreactive.

In each of the above cases,

- if there is only one output possible, write down the output;
- if more than one output is possible (hint: look for concurrency), specify how an output may look and how it can't look, and give a valid example.

In each case, concisely justify your answer.

Solutions

Question 3-1. A reactive object performs the whole message handling in the thread in which it is called. Therefore, the main thread (the *only* thread in this case) cannot proceed before the object is ready with pinging. Thus, the output *must* be:

[3] In networking, to ping a host is to send to the host a simple request to check whether the host is reachable across the network. The example above is a rather unrealistic simulation.

```
pinging('furu.idi.ntnu.no')
ping('furu.idi.ntnu.no')
ping('furu.idi.ntnu.no')
ping('furu.idi.ntnu.no')
pinging('selje.idi.ntnu.no')
ping('selje.idi.ntnu.no')
ping('selje.idi.ntnu.no')
ping('selje.idi.ntnu.no')
```

Question 3-2. An active object collects the messages it is called with on a stream, and processes them, in a separate thread, in the order they have arrived. The following *must* hold:

- the `pinging('furu.idi.ntnu.no')` printout must appear in the output before the `pinging('selje.idi.ntnu.no')` printout;
- the `pinging` printout corresponding to a server (furu or selje) must appear in the output before any `ping` printout corresponding to the same server;
- all `ping('furu.idi.ntnu.no')` printouts must appear in the output before any of the `pinging('selje.idi.ntnu.no')` printouts;

Beyond that, the order is unspecified. Thus, the output *could* be:

```
pinging('furu.idi.ntnu.no')
ping('furu.idi.ntnu.no')
pinging('selje.idi.ntnu.no')
ping('furu.idi.ntnu.no')
ping('furu.idi.ntnu.no')
ping('selje.idi.ntnu.no')
ping('selje.idi.ntnu.no')
ping('selje.idi.ntnu.no')
```

It is perhaps, depending on the scheduller, most likely that the two first printouts in the output are the `pinging` printouts; the question, however, was *what* outputs are possible, not which are *more or less likely* than others.

Question 3-3. A hyperreactive object performs all message handling in separate threads, one thread per message. The following *must* hold:

- the `pinging('furu.idi.ntnu.no')` printout must appear in the output before the `pinging('selje.idi.ntnu.no')` printout;
- the `pinging` printout corresponding to a server (furu or selje) must appear in the output before any `ping` printout corresponding to the same server;

Beyond that, the order is unspecified. Thus, the output *could* be:

```
pinging('furu.idi.ntnu.no')
ping('furu.idi.ntnu.no')
pinging('selje.idi.ntnu.no')
ping('selje.idi.ntnu.no')
ping('furu.idi.ntnu.no')
ping('selje.idi.ntnu.no')
ping('furu.idi.ntnu.no')
ping('selje.idi.ntnu.no')
```

Both the active and the hyperreactive objects can produce an output identical to that of the reactive object. The hyperreactive object can produce all outputs that the active object can produce, but the inverse is not true.

Task 4 (15 points)

Question 4-1. Explain what it means for a program to be *declarative*. Interpret the term 'declarative' in a broad sense — describe more than one way in which it is used and understood.

Question 4-2. Give at least two examples of language features, or combinations of more than one feature, that cannot be included in a declarative model of computation. Explain why it is that they cannot.

Question 4-3. Explain what the benefits and drawbacks of using a declarative model of computation are.

Solutions

Question 4-1. Different answers were possible. According to CTMCP, a component (e.g., a function) is declarative if on every logically equivalent occasion of use (e.g., application to the same arguments) it performs in the same way (e.g., returns the same value). A program is declarative if it specifies and uses only declarative components. Another view on declarativity is that a program is declarative if it specifies *what* should be done (e.g., the results of a computation) rather than *how* to do it (e.g., the algorithm). See Sec. 3.1 and preface to Ch. 6 in CTMCP for details on how the books defines declarativeness.

Question 4-2. Features that make a language non-declarative include explicit state (e.g., in the form of cells or ports), non-deterministic choice (in principle), and the combination of exceptions with concurrency. (If you claimed that concurrency leads to non-declarativeness and provided good arguments, the answer was considered correct.) See the corresponding chapters in CTMCP for further details.

Question 4-3. Different answers were possible. Easier reasoning about programs and robustness are among the major features of a declarative model of computation. Unnatural code and difficulties in simulation of real-life stateful objects are among the major drawbacks of declarativeness. See the corresponding chapters in CTMCP for further details.

Task 5 (20 points)

Question 5-1. Define the terms 'syntax' and 'semantics'.

Examine the following implementation of a *syntax checker*:[4]

```
\insert Solve.oz

declare

fun {CheckSyntax Tokens}
   {SolveAll fun {$} {CheckProgram Tokens nil} true end} \= nil
end

proc {CheckProgram Tokens Rest}
   {CheckInstructions Tokens Rest}
end

proc {CheckInstructions Tokens Rest}
   choice
      Tokens = nil
      Rest = nil
```

[4]A syntax checker parses code just as a parser does, but unlike a parser, it returns the Boolean value `true` rather a structured representation of the code if the code is a valid program.

```
      [] Instructions in
         {CheckInstruction Tokens Instructions}
         {CheckInstructions Instructions Rest} end
   end

   proc {CheckInstruction Tokens Rest}
      choice
         {CheckDefinition Tokens Rest}
      [] {CheckExpression Tokens Rest} end
   end

   proc {CheckDefinition Tokens Rest}
      choice
         {CheckVariableDefinition Tokens Rest}
      [] {CheckFunctionDefinition Tokens Rest} end
   end

   proc {CheckExpression Tokens Rest}
      choice
         {CheckIdentifier Tokens Rest}
      [] {CheckNumber Tokens Rest}
      [] {CheckApplication Tokens Rest} end
   end

   proc {CheckVariableDefinition Tokens Rest}
      Identifier Expression in
      Tokens = '<'|'define'|Identifier
      {CheckIdentifier Identifier Expression}
      {CheckExpression Expression '>'|Rest}
   end

   proc {CheckFunctionDefinition Tokens Rest}
      FunctionIdentifier ArgumentIdentifier Expression in
      Tokens = '<'|'define'|'<'|FunctionIdentifier
      {CheckIdentifier FunctionIdentifier ArgumentIdentifier}
      {CheckIdentifier ArgumentIdentifier '>'|Expression}
      {CheckExpression Expression '>'|Rest}
   end

   proc {CheckIdentifier Tokens Rest}
      Tokens = identifier(_)|Rest
   end

   proc {CheckNumber Tokens Rest}
      Tokens = number(_)|Rest
   end

   proc {CheckApplication Tokens Rest}
      Identifier Expression in
      Tokens = '<'|Identifier
      {CheckIdentifier Identifier Expression}
      {CheckExpression Expression '>'|Rest}
   end
```

Question 5-2. Using a BNF or EBNF notation, specify the *syntax* of the language recognized by the above syntax checker.

Question 5-3. Define the terms ‘abstract syntax’, ‘abstract syntax tree’, and ‘derivation’.

Question 5-4. Describe a typical pipeline (sequence of steps) in which a program is processed from program text to an internal structured representation. Name the components of this pipeline.

Solutions

Question 5-1. *Syntax* is a specification of the *form* of programs in a language. That is, the syntax of a language specifies which sequences of tokens are valid programs in the language (and which are not).

Semantics is a specification of the *meaning* of programs in a language. That is, the semantics of a language specifies what happens during an execution of instructions in a valid program.

Question 5-2. The language recognized by the syntax checker can be specified using the following EBNF grammar:

$$\begin{array}{rcl}
\langle \textit{program} \rangle & ::= & \{\ \langle \textit{instruction} \rangle\ \}^* \\
\langle \textit{instruction} \rangle & ::= & \langle \textit{variable definition} \rangle \mid \langle \textit{function definition} \rangle \mid \langle \textit{expression} \rangle \\
\langle \textit{variable definition} \rangle & ::= & \texttt{'<'}\ \texttt{'define'}\ \langle \textit{identifier} \rangle\ \langle \textit{expression} \rangle\ \texttt{'>'} \\
\langle \textit{function definition} \rangle & ::= & \texttt{'<'}\ \texttt{'define'}\ \texttt{'<'}\ \langle \textit{identifier} \rangle\ \langle \textit{identifier} \rangle\ \texttt{'>'}\ \langle \textit{expression} \rangle\ \texttt{'>'} \\
\langle \textit{expression} \rangle & ::= & \langle \textit{identifier} \rangle \mid \langle \textit{number} \rangle \mid \langle \textit{application} \rangle \\
\langle \textit{identifier} \rangle & ::= & \texttt{identifier}(\lambda) \\
\langle \textit{number} \rangle & ::= & \texttt{number}(\lambda) \\
\langle \textit{application} \rangle & ::= & \texttt{'<'}\ \langle \textit{identifier} \rangle\ \langle \textit{expression} \rangle\ \texttt{'>'}
\end{array}$$

Here, λ is any lexeme recognized by the tokenizer as an identifier or a numeral. (The specification of ⟨*identifier*⟩ and ⟨*number*⟩ was not essential for a correct answer).

Question 5-3. A syntax is a specification of the form of a language. A program in a language is essentially a (linear) sequence of characters which are processed by a tokenizer and a parser into a (usually) non-linear complex data structure. *Abstract syntax* is a specification of a notation used to represent the data structures output by a parser. Abstract syntax may, but does not have to, resemble the concrete syntax. For example, the program

```
if X == 1 then Y = 2 else skip end
```

could be parsed into a hierarchical (non-linear) structure (e.g., nested records), which could be printed, according to some abstract syntax, as follows:

if (*equal* (*id* (`X`), 1), *assign* (*id* (`Y`), 2), *noop*)

A grammar can be used to specify the (concrete) syntax in which the first line above is valid; another, different grammar can be used to specify the (abstract) syntax in which the second line above is valid.

An *abstact syntax tree* (an AST, a syntax tree) is a graph-theoretic, non-linear representation of a particular output from a parser, in the form of a tree (a connected, non-cyclic graph).

A derivation is a process in which a sequence of tokens (in this context, a program in a programming language) can be constructed by iteratively applying the rules (productions) of a grammar. The rules specify how to replace a nonterminal with a sequence of nonterminals and terminals. Starting from the initial nonterminal and stopping when no nonterminals are left in the sequence, one should be able (though not necessarily in finite time) to derive all programs in a language. A derivation is typically represented as a sequence of steps, each step being a transition from one sequence of nonterminals and terminals to another sequence, according to some rule from the grammar.

(The terminology varies; the terms 'syntax tree', 'parse tree', and 'abstract syntax tree' may or may not, depending on the context, be considered synonyms.)

Question 5-4. The typical pipeline used to process a program from its linear representation as a sequence of symbols to an internal, structured representation (a syntax tree) contains the following elements:

(a) a *tokenizer* (including a lexical analyzer, a lexer) that takes a sequence of symbols (characters) as input and returns a sequence of tokens (each token being a classified lexeme) as output;

(b) a *parser* that takes a sequence of tokens as input and returns a structured representation of the program as output.

The pipeline may also include a *preprocessor* (which removes comments, expands macros, etc.), an *optimizer* (which improves the intermediate representation),[5] etc. The parser may also be seen as composed of two elements: one that parses a sequence of tokens and returns a parse tree, a hierarchical structure including the tokens as nodes, and another that performs semantic analysis of the parse tree and returns an (abstract) syntax tree, which abstracts from the concrete tokens used in the program. The terminology varies.

[5]Not to be confused with a target code optimizer.