OLD EXAM - DO NOT USE!!!

MDSDP Take-Home Exam 2022

Ulrik Pagh Schultz, Miguel Campusano
MMMI

1 Introduction

The MDSD 2022 take-home exam concerns the implementation of the IF22 language in Xtext. This document describes the language informally using EBNF and examples. The attached archive contains the source of nine IF22 programs, the corresponding generated Java code, and test programs for the generated Java code.

Your goal is to implement a compiler for IF22 using Xtext. The code generated by your compiler does not need to be identical to the provided generated code but should be similar (as described later) and should pass the test cases. The provided example programs exercise all parts of the language. Your implementation is expected to work for the supplied IF22 programs and similar programs. For a complete answer, your compiler must support validation and be able to guarantee that the generated code is without errors code. Note that the exam does not emphasize other interactive features such as content assist. Scoping rules are very relevant, and validation requires implementing type validation (among others). Still, the exam can be completed with a medium grade without any scoping or validation, as described later. The rest of this document is organized as follows. First, Section 2 introduces the IF22 language, after which Section 3 gives you hints on how you can implement your solution. Then, Section 4 describes the formal requirements on providing a solution, and last Section 5 describes how grading will be performed.

2 The IF22 Language

The IF22 language is designed to create Interactive Fiction (IF) games. An IF is a type of videogame in which players use text commands to control characters in a world represented only by words. You can see this as an interactive narration.

The basic element of a narrative is a *Scenario*. Scenarios have *announcements*, *questions* and *ends*. An announcement is a piece of text given to the player. A question is also a piece of text, but it waits for a typed input from the player. Depending on a conditional expression and the player's input, the narrative chooses how to continue by selecting another announcement, question, or end. Scenarios can specify how they end by using end statements, which may give a final text to the player.

A narrative can have several scenarios, and a scenario can instantiate other scenarios. Whenever an instantiated scenario ends, the scenario that called it can decide what to do, depending on how the instantiated scenario ended.

2.1 Scenarios

Scenarios consist of a sequence of *announcements*, *question* and *end* statements. Announcement statements show a text to the player, and Question statements show a text and wait for input from the player. Both statements should connect to one or more announcements, questions, or ends, and this connection can use a conditional

expression, allowing for one statement to *jump* to another statement if the condition returns *true*. For question statements, the conditional expressions can use the input given by the player. Finally, end statements do not connect to any other statement, and they only indicate that the scenario is over, optionally showing a final piece of text.

```
1  story Example
2  scenario House{
3    announce Enter "You see a suspicious house,
4                    and you decide to go inside"
5      to Room
6
7    question Room "In the middle of the room,
8                   there is a basket with apples.
9                   You count them...
10                  How many apples do you find?" as number
11     to TonsOfApplesEnd if this > 10
12     to FewApplesEnd
13
14   end TonsOfApplesEnd "So many apples, you take some of them
15                        wondering who needs that many..."
16   end FewApplesEnd "You leave the apples over there,
17                     somebody will notice if you take some"
18 }
```

The previous program defines a story named `Example`, which contains a scenario named `House`. Then, it defines an announcement `Enter` using the `announce` keyword. This announcement shows a string to the player. Finally, the `to` statement will call another announcement or question after this announcement is completed.

If there are several announcement and question statements, the first one defined in the scenario will start the scenario. In this case, the player receives the text from the `Enter` announcement, and then gets the text of the following question statement `Room`. When receiving this question, the player should input a number, as determined by the `as number` statement. If the user types more than 10, the narrative jumps to the end `TonsOfApplesEnd`; if not, the narrative jumps to `FewApplesEnd`, as shown in Lines 10 and 11. As you can imagine, announcements and questions may have several outputs with different conditions, and the program jumps on the first successful condition. The `this` keyword represents the value of the input received from the player for a particular question.

2.2 Variables and Types

The language supports three basic types: text, number, and boolean. The text type is the same as Java Strings, the number type is the same as Java integers, and the boolean type is represented with `true` and `false` keywords.

Scenarios can define variables that can be filled using question statements, for example:

```
story House
scenario House{
  var apples : number

  /* ...Skip Code... */

  question Room "...How many apples do you find?"
```

```
                    as number in apples
    to TonsOfApplesEnd if apples > 10
    to FewApplesEnd

  /* ...Skip Code... */
}
```

This program defines a variable `apples`, specifying its type. Then, in the question `Room` the statement at the end `in apples` stores the input from the player into the variable `apples`. This variable can be used in any expression of this scenario, as shown in the condition of the same question. In particular, in the conditions from question `Room` the variable `apples` has the same value as `this`.

2.3 Expressions

Expressions in IF22 include ordinary binary operations for arithmetic, text concatenation using `&`, and conditional expressions such as `&&` *and*, `||` *or*, `<=` *less than equal*, `!` *negation*, etc. We provide the full extent of the operations in the EBNF in Section 2.7. The following program show the use of several expressions:

```
story Apples
scenario PickApples{
  var name : text
  var groups : number
  question AskName "What is your name?"
                    as text in name
    to NoName if name == ""
    to SpotBasket

  announce NoName "Come on, do not be shy!"
    to AskName

  announce SpotBasket "So "&name&"... you go into the room
                       and spot a basket with 50 apples..."
    to GroupApples

  question GroupApples "You decide to carry them in groups.
                        How many groups will you make?"
                          as number in groups
    to TonsOfApples if (50 / groups < 50 + 1) &&
                       (50 / groups >= 50 / 10)
    to TonsOfGroups if 50 / groups < 50 / 10

  end TonsOfApples "That is a lot of apples per group,
                    you will need more than your hands..."
  end TonsOfGroups "So many groups! That will require
                    a lot of trips to take them all..."
}
```

2.4 Question Input Validation

When using questions, programs do not only have the choice of validating the type of the input from the players, but also extra validation can be done. If the input fails the validation, then the question is *asked again*. For example, when a question ask for a number, it can validate that the number is positive and not 0. This way,

there is no problem if we want to use this input as a denominator of a division (*i.e.,* we do not divide by zero and the division is always positive):

```
story Apples
scenario PickApples{
  var name : text
  var groups : number

  /* ...Skip Code... */

  question GroupApples "You decide to carry them in groups.
                        How many groups will you make?"
                          as number > 0 in groups
    to TonsOfApples if (50 / groups < 50 + 1) &&
                       (50 / groups >= 50 / 10)
    to TonsOfGroups if 50 / groups < 50 / 10

  /* ...Skip Code... */
}
```

Of course, this means that the validation can support any type of expression, including the keyword for the type of the question (`number`, `text`).

2.5 Multiple Scenarios

IF22 programs can define multiple scenarios, choosing the first defined one as the starting scenario. Then, announces and questions can call to other scenarios, and, depending on how these other scenarios end, the original caller can choose what to do.

```
1  story House
2  scenario House{
3    announce Enter "You see a suspicious house,
4                    and you decide to go inside"
5      to Room
6
7    question Room "In the middle of the room,
8                   there is a basket with apples.
9                   Would you take some apples?" as text
10     to TakeApples if this == "yes" {
11       on TonsOfApplesEnd to TonsOfApples
12       on FewApplesEnd to FewApples
13     }
14     to NoPick
15
16   end TonsOfApples "You must really love apples"
17   end FewApples "Better pick a few at a time"
18   end NoPick "Maybe it is better not to pick any..."
19 }
20
21 scenario TakeApples {
22   question Take "How many apples do you take?"
23                   as number > 0
24   to TonsOfApplesEnd if this > 10
25   to FewApplesEnd
```

```
26
27    end TonsOfApplesEnd
28    end FewApplesEnd
29  }
```

In this example, the scenario `TakeApples` is called from the question `Room` from scenario `House`. In the `on` statement for this question, instead of pointing to an announce, question or end, it points directly to the `TakeApples` scenario (in this case, using a condition). The scenario `TakeApples` has two endings without text, which means no text is displayed when arriving at any of the endings. Nevertheless, these endings are used when deciding what to do after the scenario `TakeApples` finishes. If the `TonsOfApplesEnd` ending is reached, then the next part of the story will be `TonsOfApples` ending from `House` scenario, as shown in Line 11. If the `FewApplesEnd` is reached, then the next part will be `FewApples` ending from `House` scenario, as shown in Line 12.

Moreover, scenarios can be initialized with variables (similar to class constructors):

```
story House
scenario House{
  var name : text

  /* Skip Code: filled name with the name of the player */

  question Room "How many apples will you take"
                  as number > 0
    to TakeApples(name, this){
      on TonsOfApplesEnd to TonsOfApples
      on FewApplesEnd to FewApples
    }
    to NoPick

  end TonsOfApples "You must really love apples"
  end FewApples "Better pick a few at a time"
  end NoPick "Maybe it is better not to pick any..."
}

scenario TakeApples(name: text, numApples: number) {
  announce Start "So you take "&numApples&" of apples, "&name
    to TonsOfApplesEnd if numApples > 10
    to FewApplesEnd

  end TonsOfApplesEnd
  end FewApplesEnd
}
```

In this example, `TakeApples` scenario requires two arguments, `name` and `numApples`. The call to this scenario only makes sense when filled with correctly typed arguments. Moreover, here we show two other aspects of the program: (i) concatenation can also be between strings and numbers, and (ii) announcements can also have multiple paths, even when they cannot receive input from players.

2.6 External Functions

The program also allows the definition of external functions:

```
IF22: 'story' ID Function* Scenario*
Function: 'function' ID '(' (Type (',' Type)*)? ')' ':' Type
Scenario: 'scenario' ID ('('(Parameter (',' Parameter)*)?')')?
          '{' (VariableDef | Statement)* '}'
Statement: Announce | Question | End
VariableDef: 'var' ID ':' Type
Announce: 'announce' ID Exp Target+
Question: 'question' ID Exp 'as' Exp ('in' ID)? Target+
End: 'end' ID Exp?
Target: 'to' ID ('('(Exp (',' Exp)*)?')')? ('if' Exp)? ('{' EndingTarget+ '}')?
EndingTarget: 'on' ID Target
Parameter: ID ':' Type
Type: 'boolean' | 'text' | 'number'
Exp: LogicExp | MathExp | TextExp
    | '(' Exp ')'
    | Type
    | 'this'
    | 'true' | 'false'
    | STRING
    | NUMBER
    | ID '('(Exp (',' Exp)*)?')'
    | ID
LogicExp: Exp ('==' | '!=' | '<' | '>' | '<=' | '>=') Exp
        | Exp ('&&' | '||') Exp
        | '!' Exp
MathExp: Exp ('+' | '-' | '*' | '/') Exp
TextExp: Exp '&' Exp
```

Figure 1: EBNF of IF22

```
story Example
function textLength(text) : number
scenario Example{
  question AskText "Type a text"
                   as textLength(text) > 0
    to End1 if textLength(this) > 10
    to End2
  /* Skip Code */
}
```

External functions are defined at the beginning of the program using the `function(type, type, type, ...) : returnType` expression. Developers can define as many external functions as they want, with any number of parameters and only one return value. Types are defined for the parameters and the return value.

2.7 Syntax

We provide you the EBNF of the syntax of IF22 in Figure 1. A program defines its name, external functions, and scenarios. External functions are defined with type on their parameters and a return type. Scenarios can have a constructor, and they have a list of statements: variable definition, announcements, questions, and ends. Announcements and questions must have at least one target, as the story should

progress until it encounters an end statement that does not have any target. The text displayed by an end statement is optional.

Targets define how to progress with the story. They can call parametrized scenarios, and when this happens, they can also define how to proceed when the called scenario ends by using as many `on` statements as necessary. These statements are defined inside brackets { }.

Expressions include basic boolean and arithmetic expressions and an operation over text with the concatenation operator. Expressions can also use the keywords for Type (`text`, `number`), as shown when defined an input validation from a question. Moreover, expressions can also use the keyword `this` when they are defined in Target rules inside Question rules. In addition, expression support basic values, such as numbers `NUMBER`, strings defined with quotes `STRING` and boolean values `true|false`. Finally, expressions can call external functions or reference defined variables.

The EBNF does not show precedence or associativity for expressions. Moreover, all binary operators are required to be left-associative. The precedence for expressions, from lower to greater, is:

1. And, Or boolean operators: `&&`, `||`
2. Equality comparison: `==`, `!=`
3. Less/greater comparison: `<`, `<=`, `>`, `>=`
4. Concatenation: `&`
5. Addition and subtraction
6. Multiplication and division
7. Primitives (same priority):
    - Constant: Number, String, Boolean
    - Not boolean expression: `!`
    - Keyword: type keywords (`number`, `text`) and `this`
    - References: Variable reference and function calls
    - Parenthesis

2.8 Scope

It is not mandatory to apply Xtext scope mechanism, and some of the scoping rules can be implemented using validation rules. However, we encourage you to use Xtext scope when needed, as it will help in the implementation of your solution.

In your solution, you should take into consideration:

- Check if names are undefined.
    - An announce/question pointing to a non-existing announce/question/end from the same scenario or pointing to a non-existing scenario.
    - Call to undefined variables/parameters
    - Call to undefined functions
- When calling an external scenario, the ending condition should point to an end statement from the called scenario

While every point from this list can be implemented using validation, we recommend you to use Xtext Scope. Let us exemplify the last point from the previous list:

```
1  story Example
2  scenario S1 {
3    announce A "Announce 1"
4      to S2 {
5        on End21 to End11
6        on End11 to End12
7      }
8
9    end End11
10   end End12
11 }
12
13 scenario S2 {
14   announce A "Announce 2"
15     to End21 if 1 > 0
16     to End22
17   end End21
18   end End22
19 }
```

In this program, Line 4 calls to scenario `S2`. When `S2` finishes with ending `End21`, Line 5 connects this end with `End11` of `S1`. The defined scope should take into consideration that, in this same line, `End21` is an end statement defined in `S2`. With this into consideration, Line 6 is invalid, due to `End11` is not an end statement of `S2`.

2.9 Validation

Validation should check for a range of different properties, including:

- Scenarios must have unique names. This also applies if scenarios have different parameters (we do not support method overload)
- Function definition must have unique names. As with scenarios, method overload is not supported
- Duplicated names of announces, questions, and ends. For example, if you have the same name for a question and an end statement, then the program would not be able to know which one the developer is referring to
- An scenario should have at least one end statement (Then, the smallest possible program is a scenario with only one end defined)
- Only use type keywords (`number`, `text`, `boolean`) in an input validation expression. It is illegal to use `number > 0` in another parts of the program.
- Only use *one* type keyword in a question input validation expression. It is illegal to use `number > 0 && text == "yes"` as input validation.
- Only use `this` keyword inside expression defined in Target rule (see EBNF) when the target is defined inside a Question rule. Using `this` only makes sense when a player inputs data

- On the Target rule, allow `on` keywords only when an external scenario is called. These `on` statements only makes sense when connecting the ending of the called scenario to another part of the story defined in the caller scenario
- The arguments for function calls and scenario calls should match their parameters definition. The same amount of arguments and correspondent type.
- The assignation of an input into a variable should be consistent with typing. Same type for question input and variable.
- Type of expressions
  - Input validation expression of a question should return boolean
  - The condition for going to an announce/question/end/scenario should return boolean
  - The type of `this` keyword should work consistently. `this>0` only makes sense if the input was validated as a `number`
  - The use of a variable in an expression should be consistent. `aVar > 0` only makes sense if the declaration of the variable is `var aVar : number`
  - General expression rules. For example: equality checks expressions of the same type, and/or expressions work over booleans, mathematical operations work over numbers, concatenation works when at least one of the expressions is a text (`"Hi "&1` and `100&" apples"` are legal, but `42&24` is illegal)

3 Implementation Hints

3.1 Material provided

The zip-file included with this document contains the response template (see Section 4) and the directory of the runtime eclipse instance. Inside this directory are nine IF22 programs, the corresponding generated programs, and testing programs that your generated programs should pass. See the `README.txt` file for specific details.

3.2 Concrete example

Provided generated programs are *examples* of how code can be generated, you are not required to precisely match this code, as described in more detail in Section 4.

From the files, you can see that each scenario generates its own class, which are subclasses of a common `Scenario` class. Moreover, an extra class called `Game` is generated per each IF22 file, which instantiates the starting scenario. Programs using the standard input/output Java mechanism For the standard input, we heavily recommend using the static variable `br` from `Scenario` class, as we modify this variable when executing the tests.

Every IF22 file generates a Java package containing all the classes. This package is of the form `interactive_fiction.<name_of_story>`. Moreover, an extra `interactive_fiction.common` package is created, for the `Scenario` class.

For files with external functions, an extra `External` interface is generated, with the definition of every function method. Remember, when using external functions, it is the developer's job to connect this interface with an actual implementation. For testing purposes, we provided the implementation of the interfaces for programs P7 and P9.

For testing purposes, we use the same technique as the one used during the third assignment of the course. For showing the text to users, we assume the program uses the standard output, which the tests redefine to check the actual text provided by the programs. For sending inputs to the program, tests use the `br` variable, as explained before.

Take into considerations that every String used in the examples presented in this PDF are written to be easily readable in this file, but they should not necessarily compile into correct Java code. For example, the following IF22 code:

```
story Example
scenario S1 {
  end End "I have a break line...
           ...here!"
}
```

May compile to the following Java code:

```
/* Skip code */
System.out.println("I have a break line...
           ...here!");
/* Skip code */
```

This is not a valid Java code because Java does not allow multi-line String by default. To solve this, you could use the new Java 15 Text block feature, use a StringBuilder capturing the new line character, etc. Nevertheless, we do not expect your solution to handle multi-line String, and you can just remove the break lines from the Strings of the examples provided in this text.

3.3 Relevant material

Completely solving this exam requires working with validation in general and type validation in particular, and some of the analysis needed can be done as scoping rules. While we have covered scoping and validation during the course and the assignments, we did not cover type validation. Thankfully Bettini's book has excellent coverage of these topics as part of the course curriculum. Since your time is short, here are a few hints.

First, type validation is introduced in Chapter 8 "Typing expressions" with additional details in Chapter 9 "Type checking". In general, Chapter 9 on SmallJava contains many concepts that you may find useful in solving the exam.

Second, the scoping rule for a local variable and the call to an announce/question/end/scenario is more complex than a simple cross-reference. Luckily, we covered this during assignment 2 with local and global variables (let and var binding). Nevertheless, if you forgot how to do this, the standard approach of a variable simply being an Xtext cross-reference to an element of the grammar does not work. In SmallJava, the solution is to introduce a new rule that only is used for cross-referencing, not for the grammar (page 210 of Bettini):

```
SJSymbol: SJVariableDeclaration | SJParameter ;
```

Variables can now be described as a cross-reference:

```
{SJSymbolRef} symbol=[SJSymbol]
```

The use of the "SJSymbol" technique will most likely make it easier for you to solve the basic task of referencing variables and announce/question/end/scenario.

Nevertheless, this is not enough for the case of referencing end statements from a called scenario (Section 2.8). For this case, you need to modify the default Xtext scope by implementing the method:

```
getScope(EObject context, EReference reference)
```

When you determine which `context` and `reference` you need to change the scope, you can use the utility methods:

```
Scopes.scopeFor(Iterable<? extends EObject> elements, IScope outer)
Scopes.scopeFor(Iterable<? extends EObject> elements)
```

What these methods let you do is essentially propose elements as the scope in which to search for the name we are resolving, but if the name is not found there, search instead in the `outer` scope. If you consider that there is no outer scope, you can skip that argument. Naturally, these can be nested to have many layers of scoping.

4 Solution

4.1 General principles

You will be evaluated on your ability to use Xtext to implement an IF22 compiler performing similarly to the system described in this document and the included source code. Ideally, you will implement a compiler that can:

1. Parse the IF22 source code provided.
2. Generate Java code that allows a person to play the game, showing the text from announcements, questions, and ends, allows players to input data, and passes the provided unit tests.
3. Compile programs similar to those provided (i.e., adding more declarations to one of the provided IF22 programs does not break your compiler).
4. Use validation to check if IF22 programs are legal before attempting to generate code, as presented in Section 2.8 and Section 2.9

This means that: (i) If you need to make changes to the IF22 syntax to make your compiler work, for example, by using syntactic modifiers to indicate if variables are parameters or local variables, or require the use of parentheses because you did not implement operator precedence, it is still a solution (just not as good as if you did not need to make changes). Moreover, (ii) there is no requirement that the code you generate exactly matches the provided code, just that it is similar in functionality. In addition, (iii) hard-coded solutions tailored to the precise programs that are provided are not considered acceptable. Last, (iv) validation should catch all kinds of errors outlined in Section 2.8 and Section 2.9 inside the eclipse DSL environment, such that the user does not have to resolve typing errors or illegal identifiers in the generated Java code.

This assignment thus includes Xtext-style validation but does not otherwise include anything regarding interactive usage, i.e., content assist. Type inference is needed to implement the validation, i.e., the syntax allows dividing a number by a string or adding a boolean to a number. This would not be legal in Java, and thus it is not legal in IF22 either (in other words, these illegal programs should be caught during validation).

Please note that it is possible to generate code that partially satisfies the requirements (i.e., can compile and passes the test programs provided) without using scoping or type validation. This can be done by treating all identifiers as simple `ID` types and not implementing any validation. In this case, errors in the IF22 program will manifest in the Java program but provided correct IF22 programs

generate valid Java code, and this is still a meaningful solution. An answer without scoping or validation will, of course, not be given full credit but will earn you partial credit (see next section for details on grading). For this reason, you are strongly encouraged to work in an iterative fashion, where you first produce something that works and then subsequently extend this to a more complete functionality. *Note, however, that if you are unable to implement an IF22 compiler that can generate legal Java code (code that can compile using a standard Java compiler), you are in danger of not passing the course!*

4.2 Specific requirements

You are required to hand in two different files: a pdf file with a short written description of how you have solved the different parts of the exam and a zip archive containing your solution.

The pdf file must follow the template in Figure 2. You are expected to provide short and concise answers to the questions. The response to questions 1–5 is not expected to take up more than 3 pages of text (this is not a hard requirement, but if you're writing 6 pages, you're not being concise, if you only have 1 page, then there are not enough details). The response to question 6 is your Xtext grammar file and all of your (manually written) Xtend files, which must be included at the end as part of the pdf (which will then be rather long, but that is the intention).

The zip archive file must contain files precisely matching the following structure (failure to follow this structure will result in a lower grade):

```
<SDU username>/if22cc.jar
<SDU username>/ExamEclipse/...
<SDU username>/ExamRuntime/...
```

Here, `<SDU username>` is the part of your SDU student email address that comes before `@`, so your zip file contains a single directory holding all of the contents. The file `if22.jar` must be a stand-alone version of your compiler generated using the approach described by Bettini in Chapter 5, "Standalone command-line compiler" (pages 94–97): an Xtend main is generated and exported as "Runnable jar file". The directory `ExamEclipse/` must contain your complete Xtext project, everything included (i.e., the eclipse project containing your Xtext and Xtend files). The directory `ExamRuntime/` must contain your version of the runtime directory used by the runtime eclipse instance (so primarily IF22 source files that work with your compiler and the corresponding generated Java files).

5 Grading

Maximal grade is awarded for a response that:

- Generates legal (i.e., can be compiled using the Java compiler) and readable code for all nine provided examples, using scoping rules and validation (including validating types), and signals errors for all incorrect programs.
- Is implemented in an understandable way and appropriately uses Xtext/Xtend features (such as cross-references in the grammar file and type-switch or dispatch methods in the Xtend files). This also implies that using scoping rules to find missing identifiers is preferred to validation rules since this is the "Xtext way" of solving the problem.
- Is robust in the sense that the compiler always generates correct code for legal IF22 programs, and illegal IF22 programs do not generate Java programs.

1. Name, email
2. Xtext Grammar:
   (a) Does your grammar support all of the example programs? If not what are the limitations?
   (b) How did you implement operator precedence and associativity?
   (c) How did you implement the syntax of variables (`ID` in rule `Exp` of the IF22 BNF) such that they can refer both to parameters of scenarios and local variables?
   (d) How did you implement the syntax of `to` statements, such that they refer to announcement/question/end/scenario? (first `ID` in rule `Target` of the IF22 BNF)
3. Scoping Rules
   (a) Did you implement scoping rules that allow variables to refer to variable definition and scenario parameters? If not, what are the limitations?
   (b) Did you implement scoping rules that allow announcement and question statements to call scenarios and reference the called scenario end statements? If not, what are the limitations?
   (c) Describe your implementation of any scoping rules included with your system.
4. Validation
   (a) What validation rule did you implement, if any?
   (b) Did you implement validation for the correct use of keywords such as `this` and Type keywords (`number`, `text`)? If not, what are the limitations? If yes, briefly describe your approach
   (c) Did you implement validation with type inferencing? If yes, does it correctly check the types for all expressions (including input validation and conditions) in the provided IF22 examples? If not, what are the problems?
   (d) If you implemented validation with type inferencing, briefly describe your approach.
5. Generator
   (a) Does your code generator correctly generate code for all of the examples provided, and are you confident that it will also work for "similar" programs? If not, then what limitations are there?
   (b) Briefly describe how your code generator works.
6. Implementation: include your Xtext grammar file and all implemented Xtend files (scoping, type validation, generator, and any additional file).

Figure 2: Outline of the response template

- Provides clear and concise answers to questions 1–5 in the response template.
- Follows the guidelines outlined in the previous section.

The minimally acceptable response is one that:

- Generates legal (i.e., can be compiled using the Java compiler) code for a minimal set of slightly modified IF22 examples included in the response.
- Is implemented in Xtext.
- Provides answers to questions 1–5 in the response template.
- Manages to include enough material that it is possible to evaluate the answer.

Anything in between is assessed in terms of how well you have managed to solve the problem. A compiler that correctly generates code for all of the example programs but does not include any scoping rules or validation will be awarded a grade of "7". The assessment includes running your compiler (using the provided jar file), inspecting your code (Xtext and Xtend) and the generated code, and may also include interactively inspecting your project from within eclipse.

Last, keep in mind that this is an *individual* exam. Therefore, you are not allowed to interact with other students regarding this exam; this will be considered cheating. The sanctions put in place by SDU against cheating in online exams such as this one are quite severe. You are, however, allowed to contact Ulrik and/or Miguel and make use of all materials (i.e., information on itslearning, the curriculum, material found on the Internet). Please see itslearning for additional details on how to contact Ulrik and/or Miguel during the exam.