MDSD Take-Home Exam 2024

Miguel Campusano
MMMI, Software Engineering Section

1 Introduction

The MDSD 2024 take-home exam concerns implementing the X24 language in Xtext. This document describes the language informally using EBNF and examples. The attached archive contains the source of 6 X24 programs, the corresponding generated Java code, and test programs for the generated Java code.

Your goal is to implement a compiler for X24 programs 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 work with the test scenarios. The provided example programs exercise all parts of the language. Your implementation is expected to work for the supplied X24 and similar programs. For a complete answer, your compiler must support validation and be able to guarantee that the Java-generated code does not have errors. Note that the exam does not emphasize other interactive features, such as content assistance. 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 X24 language, after which Section 3 gives hints on how to implement your solution. Then, Section 4 describes the formal requirements for providing a solution, and Section 5 describes how grading will be performed.

2 The X24 Language

The X24 language is designed to express simple computations over mathematical sets. The language supports the following data types: integers, sets, tuples, and records.

2.1 Set and minimal program

Sets are built using square brackets (*i.e.,* { }), where elements of **the same type** are defined inside them. Moreover, sets can only contain elements of type Int and Tuple. The statement `compute` is used over sets, giving the result of the current computation. In X24, programs start with the statement `program`, which gives a name to the program and will be used to generate valid Java programs.

```
program SimpleSet
compute {1,2,3}
```

2.2 Basic data types

Besides sets and integers, X24 programs also support tuples, which have a fixed size. Moreover, every element of a tuple should be of type Int.

```
program SimpleSetWithTuples
compute {[1,2], [2,3], [4,5]}
```

2.3 Set operations

X24 programs support several operations over Sets and Tuples: union, intersection, multiplication, and projections. Union and intersection are binary operations between sets represented with the symbols `U` and `&`, respectively:

```
compute {1,2,3} U {4,5,6} & {0,2,4,6,8}
```

This computes first the union `U` of two sets and intersects `&` the result with the third set. The result is {`2,4,6`}. These operations can also be performed over a set of tuples:

```
compute {[1,2],[1,3],[2,3]} U {[2,1],[1,3],[2,3]}
```

This computes the union of two sets of pairs. The results is {`[1,2],[1,3],[2,3],[2,1]`}, since `[1,2]` and `[2,1]` are different values.

Multiplication over sets is another binary operation that computes all possible combinations of the two sets. For example, the following statement:

```
compute {1,2,3}*{4,5}
```

Results in the value {`[1,4],[1,5],[2,4],[2,5],[3,4],[3,5]`}.

Finally, projections allow the selection of specific elements in each tuple in a set:

```
compute {[1,2,3],[4,5,6],[7,8,9]}#<1,3>
```

This computation picks the first and third element of each tuple, resulting in the value {`[1,3],[4,6],[7,9]`}. Note that there is an implicit conversion of single-element tuples to the corresponding elements, so:

```
compute {[1,2],[3,4]}#<1>
```

Computes the value {`1,3`}, not {`[1],[3]`}.

2.4 Records

Expressions in X24 can be organized into *records* that contain values. One record can extend another record, similarly to single inheritance of classes. Records have parameters similar to constructor parameters on classes. Here is a simple program written using two records:

```
program Records
record r1 (a: Int) { // Constructor parameter
    b: 2 // Value of type Int
    c: {a,b} // Value of type Set
}
record r2: r1 (d: Int) {
    e: c U {d,b,3} // Value as operation
}
compute !r1(1)->c
compute !r2(1,0)->e
```

The record `r1` has a single parameter and declares two values. The second record `r2` extends `r1`, meaning it has access to all its values. The two computations yield {`1,2`} and {`0,1,2,3`} respectively.

```
Program: 'program' ID ExternalDef* Record* Compute*
ExternalDef: 'external' ID '('Type')'
Record: 'record' ID (':' ID:Record)?
    ('(' Parameter (',' Parameter)* ')')? '{' Member* '}'
Parameter: ID ':' Type
Type: 'Int' | 'Tuple' | 'Set' '(' Type ')'
Member: ID : Exp
Compute: 'compute' Exp
Exp: Exp 'U' Exp | Exp '&' Exp | Exp '*' Exp
    | Exp '#' '<' INT (',' INT)* '>'
    | Exp '#' '[' ID:ExternalDef ']'
    | '!' ID:Record ('(' Exp (',' Exp)* ')')? '->' ID:Member
    | '[' Exp (',' Exp)* ']'
    | '{' (Exp (',' Exp)*)? '}'
    | INT | ID:(Parameter|Member) | '(' Exp ')'
```

Figure 1: EBNF of X24

2.5 External Functions

The program also allows the definition of external functions:

```
program ExternalFunctions
external isEven(Int)
external areSequential(Tuple)
compute {1, 2, 3, 4, 5, 6, 7, 8, 9}#[isEven]
compute {[1,2], [2,3], [5,7]}#[areSequential]
```

External functions are defined at the beginning of the program using the `function(type)` expression. Developers can define as many external functions as they want. External functions are defined with only one parameter that must be typed. External function calls work as filters over a single set. This means functions are called over every single element of a set, computing a new set with the elements that pass the function. In the previous example, the computation yields {`2,4,6,8`} and {`[1,2], [2,3]`} respectively.

2.6 Syntax

Figure 1 shows the EBNF of the X24 syntax. A program defines its name, external functions, records, and computation statements. Records can extend other records, can take parameters, and contain members. Members have names and values. Parameters have names and types. Types are integers, sets, and tuples.

Expressions are set union, intersection, product, projection, external function call, record creation and access, as well as set constants, tuple constants, integer constants, and variable use. Variables can refer to constructor parameters of the enclosing record and members from the the enclosing record and its super-records.

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. Union and intersection
2. Product
3. Projection and function filter (both using `#`)

4. Primitives (same priority):
    - Constant: integers, sets and tuples
    - Record initialization record access
    - Parenthesis (which changes priority of operation)
    - Variable reference

2.7 Scope

Applying the Xtext scope mechanism is not mandatory, 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 implement your solution. In particular, every ID reference should point to the right definition, which may be:

- Reference to a record inside a compute statement
- Reference to members or parameters inside a record, considering inheritance
- Reference to a member from a record inside a compute statement, considering inheritance

2.8 Validation

Validation should check for a range of different type properties. From the following list, you should pick any 2 of them:

- Compute statements should always return a Set (Remember that operations also return sets: `{1,2,3}` and `{1,2,3} U {4,5}` are both valid expressions for the compute statement).
- Binary set operations and projection operations should only be applied to sets and tuples, respectively.
- Filter function call should be applied to the correct type of argument: *e.g.,* a function that accepts Int cannot be applied to a set of tuples.
- Set must contain elements of the same type. While `{1,2,3}` is valid, the set `{1,2,[3,4]}` is not.
- The use of a variable in an expression should be consistent. `{1,2,a}` is only valid if `a` is an integer (remember, elements inside a set must have the same type). This also includes record access when used inside a set.

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 6 X24 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.

The files show that each program generates its own class, which are subclasses of a common `AbstractRecordProgram` class. This common class defines several util methods for set operations, within the `utils` package. Moreover, the class `Tuple` is defined inside the same util package, which makes easier to create and work with tuples. Finally, every X24 file generates a Java class inside a Java package called `setdsl`.

For X24 programs with external function definitions, an extra `External` interface is generated inside the same Java class, 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 P5 and P6.

3.3 Relevant material

Completely solving this exam requires working with scope and type validation. 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 member and a parameter 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]
```

Nevertheless, this is insufficient to reference members on a record access statement. 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)
```

These methods let you propose elements as the scope 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 there is no outer scope, you can skip that argument or use the null scope `IScope.NULLSCOPE`. 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 X24 compiler that performs similarly to the system described in this document and the included source code. Ideally, you will implement a compiler that can:

1. Parse the X24 source code provided.
2. Generate Java code that passes the provided tests.
3. Compile programs similar to those provided (i.e., adding more declarations to one of the provided X24 programs does not break your compiler).
4. Use validation and scope to check if X24 programs are legal before attempting to generate code.

This means that: (i) If you need to make changes to the X24 syntax to make your compiler work, for example, by using syntactic modifiers to indicate if variables are parameters or members, 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) the code you generate does not need to match the provided code exactly, just that it is similar in functionality. In addition, (iii) hard-coded solutions tailored to the precise programs provided are not considered acceptable (and you will most likely fail). Last, (iv) validation should catch errors inside the Eclipse DSL environment so that the user does not have to resolve typing errors or illegal identifiers in the generated Java code. This assignment thus includes Xtext-style scope and validation but does not otherwise include anything regarding interactive usage, i.e., content assist.

Note that it is possible to generate code that partially satisfies the requirements without using scoping or validation. This can be done by treating all identifiers as simple `ID` types and not implementing any validation. In this case, errors in the X24 program will manifest in the Java program, but provided that the correct X24 programs generate valid Java code, 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 iteratively**, where you first produce something that works and then subsequently extend this to a complete functionality (this is why we provided several programs, of which P1 is the simplest one, and P6 is the most difficult one). *Note, however, that if you are unable to implement an X24 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 are writing 6 pages, you are 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).

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 X24 BNF) such that they can refer both to record's parameters and members?
3. Scoping Rules
   (a) Did you implement scoping rules that allow variables to refer to members and parameters? If not, what are the limitations?
   (b) Did you implement scoping rules that allow for correct record member access? 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) Describe the implementation of any validation rule.
   (c) For the non-implemented validation rules, describe the problems of not having them.
5. Generator
   (a) Does your code generator correctly generate code for all 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

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>/x24cc.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 the contents. The file `x24cc.jar` must be a standalone version of your compiler generated using the approach described by Bettini in Chapter 5, "Standalone command-line compiler" (pages 94–97). The directory `ExamEclipse/` must contain your complete Xtext project and everything included (i.e., the eclipse project containing your Xtext and Xtend/Java files). The directory `ExamRuntime/` must contain your version of the runtime directory used by the runtime eclipse instance (primarily X24 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 provided examples using scoping rules and validation.
- Is implemented in an understandable way and appropriately uses Xtext/Xtend features (such as cross-references and validation checks). 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 X24 programs, and illegal X24 programs do not generate Java programs.
- 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 X24 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". Remember to work in an iterative way using the X24 example programs as a base. Build your grammar and generator to work first with P1, then move on to P2, and so on.

The assessment includes running your compiler (using the provided jar file), inspecting your code (Xtext and Xtend/Java) 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 pretty severe. You are, however, allowed to contact Miguel and use all materials (i.e., information on itslearning, the curriculum, and material found on the Internet), except for AI tools. Please see itslearning for additional details on how to contact Miguel during the exam.