Bachelor Assignment 3
Group 5

Andreas Abild Mortensen
anmor16@student.sdu.dk

Samuel Valdemar Grange
sagra16@student.sdu.dk

Mads Grau Kristensen
mkris16@student.sdu.dk

March 2019

1 Introduction

In this part of the project we have created a weeder and a type checker for our compiler. The purpose of the weeder is to check the abstract syntax tree for invalid syntax that the parser and scanner can not catch like how the identifier in a function's tail must be the same as in the function's head. The type checker's job is to find the type of expressions and make sure expressions has the expected type. For example the plus operator expects expressions of type `int` on both sides so the type checker finds the type of both sides and then gives an error if any of the side's types are not and `int`. This enforces some correctness in the logic of the program.

2 Design philosophy behind features

We will start by discussing the design philosophy behind the val type. The val type allows us to write less verbose code, where much redundant explicit code can be left out. A combination of val's and lambda's can be used to create a powerful declarative standard library. Concepts such as map, filter and reduce can be implemented using val's and lambda's.
For classes we have implemented basic classes on the frontend as just "namespaces". Furthermore the generics for classes and type constraints are mostly handled on the frontend, which also allows us to not perform much extra work to implement it in the backend phase. These provide flexibility for our language, and we feel that it is a nice addition to the language.

3 Design and Implementation

3.1 Basic requirements

Our weeder simply traverses the abstract syntax tree twice, once checking if the identifier in a functions tail is the same as the on in the head. The second time we check if all functions have an return statement.

After the weeding phase, we decorate the abstract syntax tree. The decoration phase makes sure to setup symbol table pointers and put the symbols for declarations and lambda bodies inside of the symbol tables. Additionally this phase also changes id types into generic types that should be converted, and evaluates the types for automatic value declarations.

The type checker is a collection of functions, each function used to type check some part of the grammar (declaration, statement, expression, etc.). All functions take some part of the abstract syntax tree and an expected type returning an Error. If there is an error while type checking a statement or declaration like using a symbol that is not declared or an unexpected type an Error is returned otherwise `NULL` is returned telling us that there was no error. The flow of type checking using the functions are shown on figure 1. The type checker is very complex and catches many corner cases. Although it catches many of the corner cases, it does not completely pass all of the tests given.

3.2 Additions

Since we have a substantial amount of additions to the basic grammar, explaining the implementation in detail would require a lot of time. Instead we will give an overview of how we conceptually built the different additions.
For a simple thing like auto deriving types. The val auto type can only be created by assigning it to a value, hence how it can be constrained to only be automatic. In a bit more detail, the automatic type uses a utility function to unwrap an expressions type and then binding the type to the declaration.
For lambdas we had to implement a new lambda type which represents the parameters and the return type. We had to both implement a new term for the lambda as a right hand side value and as the previously mentioned type. The new lambda term can have a body and thus must be type checked as a function. Since we now also have the lambda type, type checking in some cases has gotten significantly more complex, like for instance when checking for type equivalences.
Classes are a beast of their own. Classes themselves are simply implemented as namespaces. They can contain declarations only, since statements inside of a class scope does not make sense. Function declarations are not allowed, and all "methods" should be defined with lambdas instead, as they are much more powerful.
The classes become significantly more complex and useful when adding inheritance. Inheritance is implemented by using a "mixin" technique. A mixin is simply, when we extend a class, we just prepend all of its content to the class we're currently in. This may seem like a hack, but it is actually a real technique and

it can be called inheritance.
But having inheritance cannot be enough right? Therefore we have implemented generic types also. The generic types work like any other languages generic types, they are just placeholders for when the classes have actually been instantiated and bound with types. The implementation of this was very hard, since there were many corner cases. It was implemented by allowing everything in the class scope to point to a generic type as a type id, then later these type ids that were converted to a new generic type type. When the classes were actually instantiated, the class's generic types had to be bound as well, so we could actually interact with the class fields as the correct types. This got very complicated with classes extending classes and such.
A frontend only addition for the classes is type constraining. Type constraining means that we can guarantee that a generic type of the class is at least of some other class. This means that we can access the generic type as if it were a type of whatever type constraint it has. Implementing this was not in one sense, since it at the first point only required to look through the bound class type and its extended classes recursively. The real difficulty was when we had to account for pulling fields out of the class that were generic and constrained, this required many small tweaks over the whole typechecker.
Finally we have class down-casting. This is useful for when you have a function for instance that takes a more primitive class that what you have in scope, then down-casting it to the more primitive type would provide the tools to call the function. Obviously down-casting requires a class to extend the class to be down-casted to.

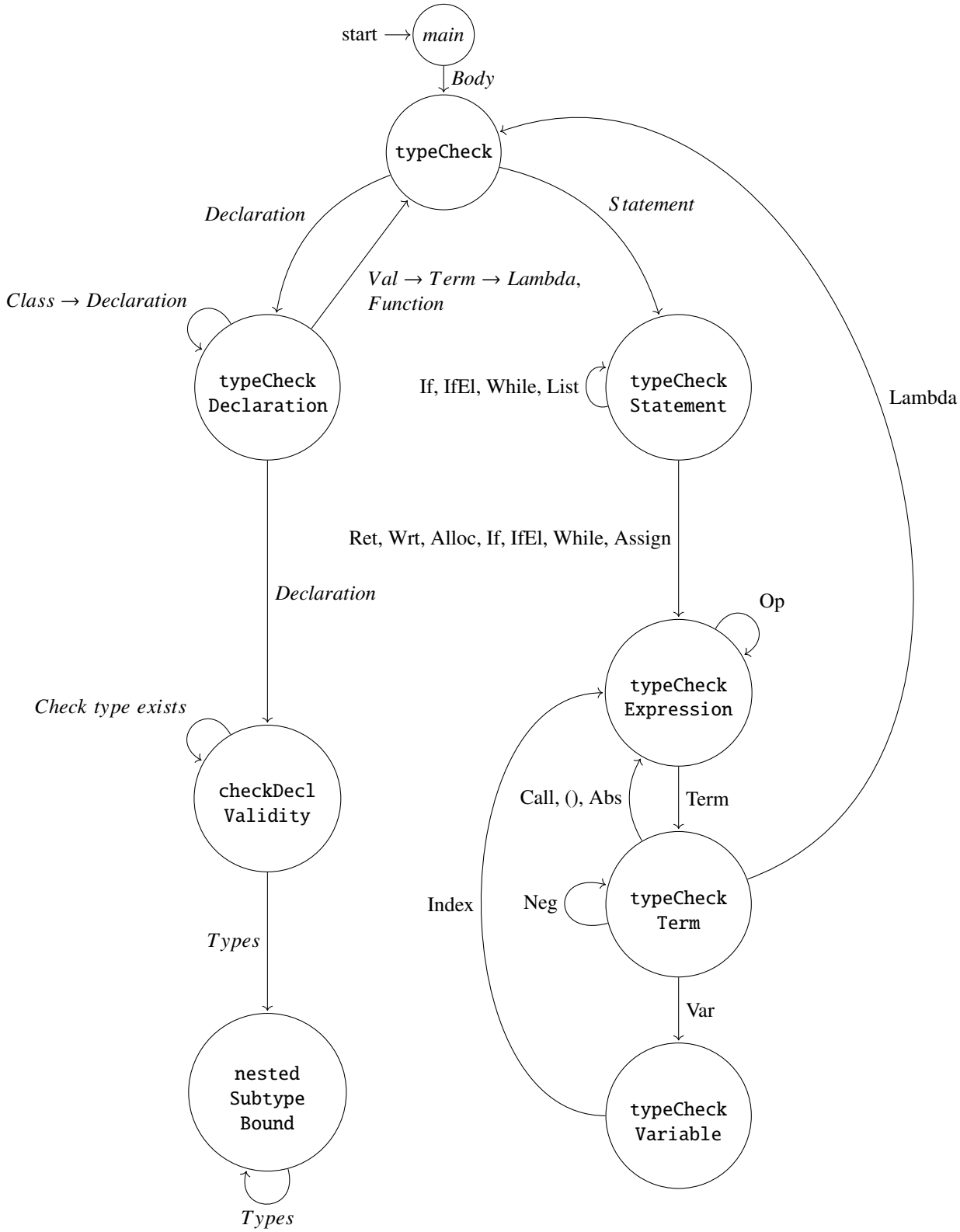

Figure 1: How the different functions of the type checker interact.

4 Grammar additions

4.1 New grammar

Before delving too deep into the new grammar and special additions, a quick roundup of the new syntax will be listed. This new grammar and conceptual syntax is type checked.

- Recursive types, a->b->c->a
- Lambdas, lambda & function types, and R-value lambdas
- Auto derived types (which will be const checked in the weeder later)
- Classes
- Classes with extended types (inheritance)
- Classes with generic types
- Classes with generic types which have subtype constraints
- Class downcasting

4.2 Examples and descriptions

In this part of the project we have extended the grammar for our language. The entire grammar is shown on figure 2 and 3. Our new additions adds lambda functions, functions as values, auto derived types at declaration, and classes with generics.

Lambda functions provide a more concise way to declare functions. This also allows us to give functions as arguments to functions allowing for callbacks and more. The following would not be possible in standard Kitty syntax.

```
1  func slowOperation(a: int): int
2      return a*2;
3  end slowOperation
4
5  func doSlowThing(a: int, callback: (int) -> int): bool
6      val x = slowOperation(a); (* Some function that takes a long time *)
7      val res = callback(x);
8      return true;
9  end doSlowThing
10
11 (* The 'callback' argument is given using a lambda function *)
12 val r = doSlowThing(2, (a: int): int -> {
13     write a;
14     return a;
15 });
```

This would be more useful if we had a way to run code asynchronously, but it is still useful for higher-order functions like map, filter and reduce.

Functions as values allows assigning functions to identifiers like variables using lambda functions. For example this piece of code:

```
1  func add(a: int, b: int): int
2      return a + b;
3  end add
```

```
func addBy(a: int): (int) -> int
    return (a1: int): int -> {
        return a1 + a;
    };
end addBy

var addTwo: (int) -> int;

addTwo = addBy(2);

write addTwo(3);

(* Would write '5' *)
```

Can now be written as the following

```
var add: (int, int) -> int;
add = (a: int, b: int): int -> {
    return a + b;
};
```

This is a bit more verbose but the following piece of code would not be possible without functions as values:

```
func f(a: int, g: (int) -> int): int
    return g(a);
end f

var double: (int) -> int;
double = (a: int): int -> {
    return a + a;
};

write f(2, double);
```

And we are even able to return functions from functions like this example:

Auto derived types at assignment allows one to not write the types of a variable when declaring it, however it must be given a value at declaration. So the following standard Kitty code:

```
var a: int;
var b: bool;
var f: (int, bool) -> int;

a = 2;
b = false;
f = (a: int, b: bool): int -> {
    if (b) then
        return a;
    else
        return 0;
};
```

Can now be written as:

```
val a = 2; (* The expression '2' is evaluated to be a int, so 'a' is a int*)
val b = false;
val f = (a: int, b: bool): int -> {
    if (b) then
        return a;
    else
        return 0;
};
```

This makes functions as value a lot more elegant. Our plan is to have the 'val' keyword used for this to also make variables immutable.

We also support classes with generics and mix-ins. Classes are declared using the following syntax:

```
class A {
    val MAX_SIZE = 256;

    val test = (a: int): bool -> {
        return a < MAX_SIZE;
    };
};

var instance: class A;
val t = instance.test;

write instance.MAX_SIZE;
write t(20);
(* Would write '256' and 'true' *)
```

Mix-ins allows a form of sub typing. Classes can mix-in other classes giving them the same declarations.

```
class A {
    val MAX_SIZE = 256;

    val test = (a: int): bool -> {
        return a < MAX_SIZE;
    };
};

class B with A {
    val testWithDouble = (a: int): bool -> {
        return test(a * 2);
    };
};

var instance: class B;

val twd = instance.testWithDouble;

write instance.MAX_SIZE;
write twd(129);
(* Would write '256' and 'false' *)
```

Classes can have generic parameters which also support subtype contraints.

```
class A {
```

```
    var a : int;
};

class B [T : A] {
    var h: T;
    val wrt = (): int -> {
        write h.a;
        return 0;
    };
};

class C with A {};

var b: class B[C];
var c: class C;

val writer = b.wrt;

c.a = 3;
b.h = c;

return writer();
(* Would write '3' *)
```

⟨function⟩ : ⟨head⟩ ⟨body⟩ ⟨tail⟩
⟨head⟩ : **func id (** ⟨par_decl_list⟩ **) :** ⟨type⟩
⟨tail⟩ : **end id**
⟨type_list⟩ : ⟨type⟩ **,** ⟨type_list⟩
| ⟨type⟩
⟨type⟩ : **id**
| **int**
| **bool**
| **array of** ⟨type⟩
| **record of** { ⟨var_decl_list⟩ }
| **class id**
| **class id** ⟨type_list⟩**]**
| **(** ⟨type_list⟩ **)** − > ⟨type⟩
⟨par_decl_list⟩ : ⟨var_decl_list⟩
| ε
⟨var_decl_list⟩ : ⟨var_type⟩ **,** ⟨var_decl_list⟩
| ⟨var_type⟩
⟨var_type⟩ : **id :** ⟨type⟩
⟨body⟩ : ⟨decl_list⟩ ⟨statement_list⟩
⟨decl_list⟩ : ⟨declaration⟩ ⟨decl_list⟩
| ε
⟨declaration⟩ : **type id =** ⟨type⟩ **;**
| **val id =** ⟨expression⟩ **;**
| ⟨function⟩
| **var** ⟨var_decl_list⟩ **;**
| **class id {** ⟨decl_list⟩ **};**
| **class id [** ⟨generic_type_list⟩ **] {** ⟨decl_list⟩ **};**
| **class id with** ⟨class_extension_list⟩ **{** ⟨decl_list⟩ **};**
| **class id [** ⟨class_extension_list⟩ **] with** ⟨class_extension_list⟩ **{** ⟨decl_list⟩ **};**
⟨statement_list⟩ : ⟨statement⟩
| ⟨statement⟩ ⟨statement_list⟩
⟨statement⟩ : **return** ⟨expression⟩ **;**
| **write** ⟨expression⟩ **;**
| **allocate** ⟨variable⟩ **;**
| **allocate** ⟨variable⟩ **of length** ⟨expression⟩ **;**
| ⟨variable⟩ **=** ⟨expression⟩ **;**
| **if** ⟨expression⟩ **then** ⟨statement⟩
| **if** ⟨expression⟩ **then** ⟨statement⟩ **else** ⟨statement⟩
| **while** ⟨expression⟩ **do** ⟨statement⟩
| { ⟨statement_list⟩ }
⟨variable⟩ : **id**
| ⟨variable⟩ **[** ⟨expression⟩ **]**
| ⟨variable⟩ **. id**
⟨expression⟩ : ⟨expression⟩ **op** ⟨expression⟩
| ⟨term⟩

Figure 2: Our new grammar pt. 1

⟨term⟩ : ⟨variable⟩
| **id (** ⟨act_list⟩ **)**
| **(** ⟨expression⟩ **)**
| **!** ⟨term⟩
| **|** ⟨expression⟩ **|**
| **num**
| **true**
| **false**
| **null**
| ⟨lambda⟩
| **id : id**

⟨act_list⟩ : ⟨exp_list⟩
| ε

⟨exp_list⟩ : ⟨expression⟩
| ⟨expression⟩ **,** ⟨exp_list⟩

⟨lambda⟩ : **(** ⟨par_decl_list⟩ **) :** ⟨type⟩ **− > {** ⟨body⟩ **}**

⟨class_extension_list⟩ : **id**
| **id [** ⟨type_list⟩ **]**
| **id** ⟨class_extension_list⟩
| **id** ⟨class_extension_list⟩ **[** ⟨type_list⟩ **]**

⟨generic_type_list⟩ : **id : id,** ⟨generic_type_list⟩
| **id : id**
| **id,** ⟨generic_type_list⟩
| **id**

Figure 3: Our new grammar pt. 2

5 How to build and run

To build you must go into the `build-files` folder and run `make compiler`. After building, you can run either with types displaying by running `./runCompiler` and without (types can make it very unreadable with lambdas) by running `./runCompilerNoTypes.sh`. You can run your own kitty program though the frontend by supplying the absolute path as a command line argument;
`./Compiler /home/user/kitty/program.kitty`.

6 Testing

Test File Name	Test Performed	Motivation for test	Passed / Failed
class.kitty	Tests class with generics, sub type constraints	We want to check if our syntax for classes, generics and sub type constraints passes.	passed
decl_list.kitty	Type checking declerations and some statements. All standard Kitty syntax	We want to make sure that standard Kitty code is parsed and type checked correctly	passed
declara-tive_programming.kitty	Functions as values and lambda functions	We want to be able to return functions from functions and give functions as parameters to functions. Here we test if that parses and type checks correctly	passed
downcast.kitty	Tests down casting a class to one of its mix-ins	A simple test for seeing that the syntax for down casting is parsed and type checked correctly.	passed
functions_only.kitty	Test function syntax	We see if functions are parsed and type checked correctly with nested functions and overriding parent function.	passed
inheritance.kitty	Test inheritance of fields and downcasting of classes.	Check if classes can inherit fields from its superclass and check if classes can be downcasted to its superclass.	passed

lambda.kitty	Do we parse and type check lambdas correctly?	We want to make sure our new grammar does not break the parser and to make sure that our frontend supports these new grammar additions	passed
list_monad.kitty	Tests a lot of the new grammar additions like lambda functions classes with generics and functions as values	We want to see if our grammar supports lists with higher-order functions like map and filter.	passed
lots_of_nests.kitty	Tests nested and recursive types	We want our type checker to support recursive and nested types which we tests if we do when type checking this file.	passed
val.kitty	Test auto deriving types and calling lambdas within lambdas.	We want to test if type declaration can be omitted from vals and if lambdas accept other lambdas as a parameter.	passed