Specification of the Amy language

Computer Language Processing

LARA

Autumn 2019

1 Introduction

Welcome to the Amy project! This semester you will learn how to compile a simple functional Scala-like language from source files down to executable code. When your compiler is complete, it will be able to take Amy source (text) files as input and produce WebAssembly bytecode files. WebAssembly is a new format for portable bytecode which is meant to be run in browsers.

This document is the specification of Amy. Its purpose is to help you clearly and unambiguously understand what a Amy program means, and to be the Amy language reference, along with the reference compiler. It does not deal with how you will actually implement the compiler; this will be described to you as assignments are released.

1.1 Features of Amy

Let us demonstrate the basic features of Amy through some examples:

1.1.1 The factorial function

```
object Factorial {
  def fact(i: Int): Int = {
    if (i < 2) { 1 }
    else { i * fact(i-1) }
  }
}
```

Every program in Amy is contained in a module, also called `object`. A function is introduced with the keyword `def`, and all its parameters and result type must be explicitly typed. Amy supports conditional (or `if`-) expressions with

obligatory brackets. Notice that conditionals are not statements, but return a value, in this case an `Int`.

In fact, there is no distinction between expressions and statements in Amy. Even expressions that are called only for their side-effects return a value of type `Unit`.

The condition of an `if`-expression must be of type `Int` and its branches must have the same type, which is also the type of the whole expression.

1.1.2 Saying hello

```
object Hello {
  Std.printString("Hello " ++ "world!")
}
```

Amy supports compiling multiple modules together. To refer to functions (or other definitions) in another module, one must explicitly use a qualified name. There is no import statement like in Scala.

In this example, we refer to the `printString` function in the `Std` module, which contains some builtin functions to interact with the user. The string we print is constructed by concatenating two smaller strings with the `++` operator.

1.1.3 Input, local variables and sequencing expressions

```
object ReadName {
  Std.printString("What is your name?");
  val name: String = Std.readString();
  Std.printString("Hello " ++ name)
}
```

We can read input from the console with the `readX` functions provided in `Std`.

We can define local variables with `val`, which must always be typed explicitly. The value of the variable is given after "=", followed by a semicolon.

We can sequence expressions with ";". The value of the first expression is discarded, and the value of the second one is returned. Note that ";" is an *operator* and not a terminator: you are not allowed to put it at the end of a sequence of expressions.

1.1.4 Type definitions

Except for the basic types, a user can define their own types in Amy. The user-definable types in Amy come from functional programming and are called *algebraic data types*. In this case, we define a type, `List`, and two constructors `Nil` and `Cons`, which we can call to construct values of type `List`.

```
object L {
  abstract class List
  case class Nil() extends List
  case class Cons(h: Int, t: List) extends List
}
```

1.1.5 Constructing ADT values

```
def range(from: Int, to: Int): List = {
  if (to < from) { Nil() }
  else {
    Cons(from, range(from + 1, to))
  }
}
```

We can create a `List` by calling one of its two constructors like a function, as demonstrated in the `range` function.

1.1.6 Pattern matching

```
def length(l: List): Int = { l match {
  case Nil() => 0
  case Cons(h, t) => 1 + length(t)
}}
```

To use a list value in any meaningful way, we have to break it down, according to the constructor used to construct it. This is called *pattern matching* and is a powerful feature of functional programming.

In `length` we pattern match against the input value `l`. Pattern matching will check if its argument matches the pattern of the first case, and if so will evaluate the corresponding expression. Otherwise it will continue with the second case etc. If no pattern matches, the program will exit with an error. If the constructor has arguments, as does `Cons` in this case, we can bind their values to fresh variables in the pattern, so we can use them in the case expression.

1.1.7 Wildcard patterns and errors

The `error` keyword takes a string as argument, prints `Error:` and its argument on the screen, then exits the program immediately with an error code. In this function, we are trying to compute the head of a list, which should fail if the list is empty.

Notice that in the second case, we don't really care what the tail of the list is. Therefore, we use a *wildcard pattern* (`_`), which matches any value without binding it to a name.

```scala
def head(l: List): Int = {
  l match {
    case Cons(h, _) => h
    case Nil() => error("head(Nil)")
  }
}
```

1.2   Relation to Scala

Amy is designed to be as close to a simple subset of Scala as possible. However, it is not a perfect subset. You can easily come up with Amy programs that are not legal in Scala. However, many "reasonable" programs will be compilable with `scalac`, provided you provide an implementation of the Amy standard library along with your code. This should not be required however, as we are providing a reference implementation of Amy.

2   Syntax

The syntax of Amy is given formally by the context-free grammar of Figure 1. Everything spelled in *italic* is a nonterminal symbol of the grammar, whereas the terminal symbols are spelled in `monospace` font. * is the Kleene star, s^+ stands for one or more repetitions of s, and ? stands for optional presence of a symbol (zero or one repetitions). The square brackets [] are not symbols of the grammar, they merely group symbols together.

Before parsing an Amy program, the Amy *lexer* generates a sequence of terminal symbols (*tokens*) from the source files. Some notnerminal symbols mentioned, but not specified, in Figure 1 are also represented as a single token by the lexer. They are lexed according to the rules in Figure 2. In Figure 2, we denote the range between characters α and β (included) with $[\alpha - \beta]$.

The syntax in Figure 1 is an *overapproximation* of the real syntax of Amy. This means that it allows some programs that should not be allowed in Amy. To get the real syntax of Amy, there are some additional restrictions presented (among other things) in the following notes:

- The reserved words of Amy are the following: `abstract`, `Boolean`, `case`, `class`, `def`, `else`, `error`, `extends`, `false`, `if`, `Int`, `match`, `object`, `String`, `true`, `Unit`, `val`.

  Identifiers are not allowed to coincide with a reserved word.

- The operators and language constructs of Amy have the following precedence, starting from the *lowest*:

  (1) `val`, ; (2) `if`, `match` (3) `||` (4) `&&` (5) `==` (6) `<`, `<=` (7) `+`, `-`, `++` (8) `*`, `/`, `%` (9) Unary `-`, `!` (10) `error`, calls, variables, literals, parenthesized expressions.

$$
\begin{array}{rcl}
Program & ::= & Module^* \\
Module & ::= & \texttt{object}\ Id\ \texttt{\{}\ Definition^*\ Expr?\ \texttt{\}} \\
Definition & ::= & AbstractClassDef\ |\ CaseClassDef\ |\ FunDef \\
AbstractClassDef & ::= & \texttt{abstract}\ \texttt{class}\ Id \\
CaseClassDef & ::= & \texttt{case}\ \texttt{class}\ Id\ \texttt{(}\ Params\ \texttt{)}\ \texttt{extends}\ Id \\
FunDef & ::= & \texttt{def}\ Id\ \texttt{(}\ Params\ \texttt{)}\ \texttt{:}\ Type\ \texttt{=}\ \texttt{\{}\ Expr\ \texttt{\}} \\
Params & ::= & \epsilon\ |\ ParamDef\ [\ \texttt{,}\ ParamDef\ ]^* \\
ParamDef & ::= & Id\ \texttt{:}\ Type \\
Type & ::= & \texttt{Int}\ |\ \texttt{String}\ |\ \texttt{Boolean}\ |\ \texttt{Unit}\ |\ [\ Id\ \texttt{.}\ ]?\ Id \\
Expr & ::= & Id \\
 & | & Literal \\
 & | & Expr\ BinOp\ Expr \\
 & | & UnaryOp\ Expr \\
 & | & [\ Id\ \texttt{.}\ ]?\ Id\ \texttt{(}\ Args\ \texttt{)} \\
 & | & Expr\ \texttt{;}\ Expr \\
 & | & \texttt{val}\ ParamDef\ \texttt{=}\ Expr\ \texttt{;}\ Expr \\
 & | & \texttt{if}\ \texttt{(}\ Expr\ \texttt{)}\ \texttt{\{}\ Expr\ \texttt{\}}\ \texttt{else}\ \texttt{\{}\ Expr\ \texttt{\}} \\
 & | & Expr\ \texttt{match}\ \texttt{\{}\ MatchCase^+\ \texttt{\}} \\
 & | & \texttt{error}\ \texttt{(}\ Expr\ \texttt{)} \\
 & | & \texttt{(}\ Expr\ \texttt{)} \\
Literal & ::= & \texttt{true}\ |\ \texttt{false}\ |\ \texttt{(}\ \texttt{)} \\
 & | & IntLiteral\ |\ StringLiteral \\
BinOp & ::= & \texttt{+}\ |\ \texttt{-}\ |\ \texttt{*}\ |\ \texttt{/}\ |\ \texttt{\%}\ |\ \texttt{<}\ |\ \texttt{<=} \\
 & | & \texttt{\&\&}\ |\ \texttt{||}\ |\ \texttt{==}\ |\ \texttt{++} \\
UnaryOp & ::= & \texttt{-}\ |\ \texttt{!} \\
MatchCase & ::= & \texttt{case}\ Pattern\ \texttt{=>}\ Expr \\
Pattern & ::= & [\ Id\ \texttt{.}\ ]?\ Id\ \texttt{(}\ Patterns\ \texttt{)}\ |\ Id\ |\ Literal\ |\ \texttt{\_} \\
Patterns & ::= & \epsilon\ |\ Pattern\ [\ \texttt{,}\ Pattern\ ]^* \\
Args & ::= & \epsilon\ |\ Expr\ [\ \texttt{,}\ Expr\ ]^* \\
\end{array}
$$

Figure 1: Syntax of Amy

$$
\begin{array}{rcl}
IntLiteral & ::= & Digit^+ \\
Id & ::= & Alpha\ AlphaNum^*\ \text{(and not a reserved word)} \\
AlphaNum & ::= & Alpha\ |\ Digit\ |\ \texttt{\_} \\
Alpha & ::= & [\texttt{a} - \texttt{z}]\ |\ [\texttt{A} - \texttt{Z}] \\
Digit & ::= & [\texttt{0} - \texttt{9}] \\
StringLiteral & ::= & \texttt{"}\ StringChar^*\ \texttt{"} \\
StringChar & ::= & \text{Any character except newline and "} \\
\end{array}
$$

Figure 2: Lexical rules for Amy

5

For example,
`1 + 2 * 3` means `1 + (2 * 3)` and
`1 + 2 match {...}` means `(1 + 2) match {...}`.

A little more complicated is the interaction between `;` and `val`: the definition part of the `val` extends only as little as the first semicolon, but then the variable defined is visible through any number of semicolons. Thus `(val x: Int = y; z; x)` means `(val x: Int = y; (z; x))` and not `(val x: Int = (y; z); x)` or `((val x: Int = y; z); x)` (i.e. x takes the value of y and is visible until the end of the expression).

All operators are left-associative. That means that within the same precedence category, the leftmost application of an operator takes precedence. An exception is the sequence operator, which for ease of the implementation (you will understand during parsing) can be considered right-associative (it is an associative operator so it does not really matter).

- A `val` definition is not allowed directly in the value assigned by an enclosing `val` definition. E.g. `(val x: Int = val y: Int = 0; 1; 2)` is not allowed. On the other hand, `(val x: Int = 0; val y: Int = 1; 2)` is allowed.

- It is not allowed to use a `val` as a (second) operand to an operator. E.g. `(1 + val x: Int = 2; x)` is not allowed.

- A unary operator is not allowed as a direct argument of another unary operator. E.g. `--x` is not allowed.

- It is not allowed to use `match` as a first operand of any binary operator, except `;`. E.g. `(x match { ... } + 1)` is not allowed. On the other hand `(x match { ... }; x)` is allowed.

- The syntax $[\ Id\ .\ ]?\ Id$ refers to an optionally qualified name, for example either `MyModule.foo` or `foo`. If the qualifier is included, the qualified name refers to a definition `foo` in another module `MyModule`; otherwise, `foo` should be defined in the current module. Since Amy does not have the import statement of Scala or Java, this is the only way to refer to definitions in other modules.

- One line comments are introduced with "`//`": `//This is a comment`. Everything until the end of the line is a comment and should be ignored by the lexer.

- Multiline comments can be used as follows: `/*This is a comment */`. Everything between the delimiters is a comment, notably including newline characters and `/*`. Nested comments are not allowed.

- Escaped characters are not recognised inside string literals. I.e. `"\n"` stands for a string literal which contains a backspace and an "n".

3 Semantics

In this section we will give the semantics of Amy, i.e. we will systematically explain what a Amy represents, as well as give the restrictions that a legal Amy program must obey. The discussion will be informal, except for the typing rules of Amy.

3.1 Program Structure

A Amy program consists of one or more source files. Each file contains a single module (`object`), which in turn consists of a series of type and function definitions, optionally followed by an expression. We will use the terms object and module interchangeably.

3.2 Execution

When a Amy program is executed, the expression at the end of each module, if present, is evaluated. The order of execution among modules is the same that the user gave when compiling or interpreting the program. Each module's definitions are visible within the module automatically, and in all other modules provided a qualified name is used.

3.3 Naming rules

In this section, we will give the restrictions that a legal Amy program must obey with regard to naming or referring to entities defined in the program. Any program not following these restrictions should be rejected by the Amy name analyzer.

- Amy is case-sensitive.

- No two modules in the program can have the same name.

- No two classes, constructors, and/or functions in the same module can have the same name.

- No two parameters of the same function can have the same name.

- No two local variables of the same function can have the same name if they are visible from one another. This includes binders in patterns of match-expressions. Variables that are not mutually visible can have the same name. E.g. the program
  `val x = 0; val x = 1; 2` is not legal, whereas
  `(val x = 0; 1); (val x = 1; 2)` is.

- A local variable can have the same name as a parameter. In this case, the local variable definition shadows the parameter definition.

- Every variable encountered in an expression has to refer to a function parameter or a local variable definition.

- All case classes have to extend a class in the same module.

- All function or constructor calls or type references have to refer to a function/constructor/type defined in the same module, or another module provided a qualified name is given. It is allowed to refer to a constructor/type/function before declaring it.

- All calls to constructors and functions have to have the same number of arguments as the respective constructor/function definition.

3.4 Types and Classes

Every expression, function parameter, and class parameter in Amy has a *type*. Types catch some common programming errors by introducing *typing restrictions*. Programs that do not obey these restrictions are illegal and will be rejected by the Amy type checker.

The built-in types of Amy are `Int`, `String`, `Boolean` and `Unit`.

`Int` represents 32-bit signed integers. `String` is a sequence of characters. Strings have poor support in Amy: the only operations defined on them are are concatenation and conversion to integer. In fact, not even equality is "properly" supported (see Section 3.5). `Boolean` values can take the values `true` and `false`. `Unit` represents a type with a single value, `()`. It is usually used as the result of a computation which is invoked for its side-effects only, for example, printing some output to the user. It corresponds to Java's `void`.

In addition to the built-in types, the programmer can define their own types. The sort of types that are definable in Amy are called Algebraic Data Types (ADTs) and come from the functional programming world, but they have also been successfully adopted in Scala.

An ADT is a *type* along with several *constructors* that can create values of that type. For example, an ADT defining a list of integers in pseudo syntax may look like this: `type List = Nil() | Cons(Int, List)`, which states that a `List` is either `Nil` (the empty list), or a `Cons` of an integer and another list. We will say that `Cons` has two *fields* of types `Int` and `List`, whereas `Nil` has no fields. Inside the program, the only way to construct values of the `List` type is to call one of these constructors, e.g. `Nil()` or `Cons(1, Cons(2, Nil()))`. You can think of them as functions from their field types to the `List` type.

Notice that in the above syntax, `Nil` and `Cons` are **not** types. More specifically, they are not subtypes of `List`: in fact, there is no subtyping in Amy. Only `List` is a type, and values such as `Nil()` or `Cons(1, Cons(2, Nil()))` have the type `List`.

In Amy, we use Scala syntax to define ADTs. A type is defined with an abstract class and the constructors with case classes. The above definition in Amy would be

```scala
abstract class List
case class Nil() extends List
case class Cons(h: Int, t: List) extends List
```

Notice that the names of the fields have no practical meaning, and we only use then to stay close to Scala.

We will sometimes use the term abstract class for a type and case class for a type constructor.

The main programming structure to manipulate class types is *pattern matching*. In Section 3.5 we define how pattern matching works.

3.5   Typing Rules and Semantics of Expressions

Each expression in Amy is associated with a *typing rule*, which constrains and connects its type and the types of its subexpressions. A Amy program is said to *typecheck* if (1) all its expressions obey their respective typing rules, and (2) the body of each function corresponds to its declared return type. A program that does not typecheck will be rejected by the compiler.

In the following, we will informally give the typing rules and explain the semantics (meaning) of each type of expression in Amy. We will use function type notation for typing of the various operators. For example, $(A, B) \Rightarrow C$ denotes that an operator takes arguments of types A and B and returns a value of type C.

When talking about the semantics of an expression we will refer to a *context*. A context is a mapping from variables to the values that have been assigned to them.

- Literals of Amy are expressions of the base types that are *values*, i.e. they cannot be evaluated further. The literals `true` and `false` have type `Boolean`. `()`, the unit literal, has type `Unit`. String literals have type `String` and integer literals have type `Int`.

- A variable has the type of the corresponding definition (function parameter or local variable definition). Its value is the value assigned to it in the current context.

- `+`, `-`, `*`, `/` and `%` have type $(\texttt{Int}, \texttt{Int}) \Rightarrow \texttt{Int}$, and are the usual integer operators.

- Unary `-` has type $(\texttt{Int}) \Rightarrow \texttt{Int}$ and is the integer negation.

- `<` and `<=` have type $(Int, Int) \Rightarrow Boolean$ and are the usual arithmetic comparison operators.

- `&&` and `||` have type $(\texttt{Boolean}, \texttt{Boolean}) \Rightarrow \texttt{Boolean}$ and are the boolean conjunction and disjunction. *Notice that these operators are short-circuiting.* This means that the second argument does not get evaluated if the result

is known after computing the first one. For example, `true || error("")` will yield `true` and not result in an error, whereas `false || error("")` will result in an error in the program.

- `!` has type (`Boolean`) $\Rightarrow$ `Boolean` and is the boolean negation.

- `++` has type (`String`, `String`) $\Rightarrow$ `String` and is the string concatenation.

- `==` is the equality operator. It has type $(A, A) \Rightarrow$ `Boolean` for every type A. Equality for values of the `Int`, `Boolean` and `Unit` types is defined as *value equality*, i.e. two values are equal if they have the same representation. E.g. `0 == 0`, `() == ()` and `(1 + 2) == 3`. Equality for the *reference types* `String` and all user-defined types is defined as *reference equality*, i.e. two values are equal only if they refer to the same object. I.e. `"" == ""`, `"a" ++ "b" == "ab"` and `Nil() == Nil()` all evaluate to `false`, whereas `(val s = "Hello"; s == s)` evaluates to `true`.

- `error()` has type (`String`) $\Rightarrow$ A for any type A, i.e. `error` is always acceptable, regardless of the expected type. When a program encounters `error`, it needs to print something like `Error: <msg>`, where `<msg>` is its evaluated argument, and then exit immediately.

- `if(..) {..} else {..}` has type (`Boolean`, A, A) $\Rightarrow$ A for any type A, and has the following meaning: First, evaluate the condition of `if`. If it evaluates to `true`, evaluate and return the then-branch; otherwise, evaluate and return the else-branch. Notice that the value that is not taken is not evaluated.

- `;` is the *sequence* operator. It has type $(A, B) \Rightarrow B$ for any types A and B. Notice that the first expression has to be well typed, although its precise type does not matter. `;` evaluates and discards its first argument (which we will usually invoke just for its side-effects) and then evaluates and returns its second argument.

- `val n = e; b` defines a local variable with name `n` and adds it to the context, mapped to the value of `e`. It is visible in `b` but not in `e`. `n` has to obey the name restrictions described in Section 3.3.

- An expression `f(..)` or `m.f(..)` denotes either a function call, or an invocation of a type constructor. `f` has to be the name of a function/constructor defined in the program. The types of the real arguments of the function/constructor invocation have to match the corresponding types of the formal arguments in the definition of the function/constructor. The type of a function/constructor call is the return type of the function, or the parent type of the constructor respectively.

  Evaluating a function call means evaluating its body in a new context, containing the function's formal arguments mapped to the values of the real arguments provided at the function call. Evaluating a call to a constructor

means generating and returning a fresh object containing (a reference to) the constructor and the arguments passed to it.

Notice that an invocation of a type constructor *on values* is itself a value, i.e. cannot be evaluated further. It corresponds to literals of the other types.

- `match` is the pattern-matching construct of Amy. It corresponds to Scala's pattern matching. Java programmers can think of it as a generalized switch-statement. `match` is the only way to access the structure of a value of a class type. It also happens to be the most complicated structure of Amy.

Terminology: To explain how the match-expression works, let us first establish some terminology. A match case has a *scrutinee* (the first operand, which gets pattern matched on), and a number of *match cases* (or simply cases). A case is introduced with the keyword `case`, followed by the *(case) pattern*, then the symbol `=>` and finally an expression, which we will call the *case expression*.

As seen in Section 2, a pattern comes in four different forms, which in the grammar are denoted as (1) $Id(Patterns)$, (2) Id, (3) *Literal* and (4) `_`. We will call those forms *case class pattern*, *identifier pattern*, *literal pattern* and *wildcard pattern* respectively. The identifier at the beginning of case class pattern is called the *constructor* of the pattern, and its arguments are called its *subpatterns*.

Typing rules: For the match-expression to typecheck, two conditions have to hold:

- All its case expressions have the same type, which is also the type of the whole match expression.
- All case patterns have to *follow* the type of the scrutinee. For a pattern to follow a type means the following, according to its form:
    * Each literal pattern follows exactly the type of its literal.
    * Wildcard and identifier patterns follow any type.
    * A case class pattern follows only the resulting type of its constructor, if and only if all its subpatterns follow the types of the respective fields of the constructor.
      For example, `Nil() match { case Cons(_, t) => () }` type-checks, whereas `Nil() match { case 0 => () }` does not.

Semantics: The semantics of pattern matching are as follows: First, the scrutinee is evaluated, then cases are scanned one by one until one is found whose pattern *matches* the scrutinee value. If such case is found, its

11

case expression is evaluated, after adding to the environment the variables bound in the case pattern (see below). The value produced in this way is returned as the value of the match-expression. If none is found, the program terminates with an error.

We say that a pattern *matches* a value when the following holds:

- A wildcard pattern _ or an identifier pattern `x` match any value. In the second case, `x` is bound to that value when evaluating the case expression.

- A literal pattern matches exactly the value of its literal. Notice that string literals are compared by reference, so they can never match.

- A case class pattern `case C(..)` matches a value v, if and only if v has been constructed with the same constructor `C` and every subpattern of the pattern matches the corresponding field of v. Notice that we have to recursively bind identifiers in subpatterns.

- Parentheses `(e)` can be used freely around an expression `e`, mainly to override operator precedence or to make the program more readable. `(e)` is equivalent to `e`.

3.6 Formal discussion of types

In this section, we give a formal (i.e. mathematically robust) description of the Amy typing rules. A typing rule will be given as

$$\text{R{\scriptsize ULE} N{\scriptsize AME}}$$
$$\frac{P_1 \qquad \dots \qquad P_n}{C}$$

where P_i are the rule *premises* and C is the rule *conclusion*. A typing rule means that the conclusion is true under the premises.

Conclusions and most premises will be *type judgements* in an *environment*. A type judgement $\Gamma \vdash e : T$ means that an expression (or pattern) e has type T in environment Γ. Environments Γ are mappings from variables to types and will be written as $\Gamma = v_1 : T_1, \dots, v_n : T_n$. We can add a new pair to an environment Γ by writing $\Gamma, v_{n+1} : T_{n+1}$. We will also sometimes write a type judgement of the form $\Gamma \vdash p$. This means that p typechecks, but we don't assign a type to it. *Type checking will try to typecheck a program under the empty environment, and reject it if it fails to do so.*

The *initial environment* $\Gamma_0(p)$ of a program p is one that contains the types of all functions and constructors in p, where a constructor is treated as a function from its fields to its parent type (see Section 3.4). The initial environment is used to kickstart typechecking at the function definition level.

Figure 3 lists typing rules for expressions. Figure 4 lists typing rules for patterns, functions and programs. In the typing rule for pattern matching, *bindings*(p) refers to the variable bindings implied by a pattern as explained in

Section 3.5. Rules for literal patterns are omitted because they are the same as literal expressions.

4   The standard library of Amy

Amy comes with a library of predefined functions, which are accessible in the `Std` object. Some of these function implement functionalities that are not expressible in Amy, e.g. printing to the standard output. These *built-in functions* are implemented in JavaScript and WebAssembly in case of compilation, and in Scala in the interpreter. Built-in functions have stub implementations in the Amy `Std` module for purposes of name analysis and type checking.

The Amy compiler will not automatically include `Std` to the input files. If you want them included, you have to provide them manually.

The signature of the `Std` module is shown in Figure 5.

VARIABLE
$v : T \in \Gamma$

$\overline{\Gamma \vdash v : T}$

INT LITERAL
i is an integer literal

$\overline{\Gamma \vdash i : \mathtt{Int}}$

STRING LITERAL
s is a string literal

$\overline{\Gamma \vdash s : \mathtt{String}}$

UNIT

$\overline{\Gamma \vdash \mathtt{()} : \mathtt{Unit}}$

BOOLEAN LITERAL
$b \in \{\mathtt{true}, \mathtt{false}\}$

$\overline{\Gamma \vdash b : \mathtt{Boolean}}$

ARITH. BIN. OPERATORS
$\Gamma \vdash e_1 : \mathtt{Int} \qquad \Gamma \vdash e_2 : \mathtt{Int} \qquad op \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$

$\overline{\Gamma \vdash e_1 \; op \; e_2 : \mathtt{Int}}$

ARITH. COMP. OPERATORS
$\Gamma \vdash e_1 : \mathtt{Int} \qquad \Gamma \vdash e_2 : \mathtt{Int} \qquad op \in \{\texttt{<}, \texttt{<=}\}$

$\overline{\Gamma \vdash e_1 \; op \; e_2 : \mathtt{Boolean}}$

ARITH. NEGATION
$\Gamma \vdash e : \mathtt{Int}$

$\overline{\Gamma \vdash \texttt{-}e : \mathtt{Int}}$

BOOLEAN BIN. OPERATORS
$\Gamma \vdash e_1 : \mathtt{Boolean} \qquad \Gamma \vdash e_2 : \mathtt{Boolean} \qquad op \in \{\texttt{\&\&}, \texttt{||}\}$

$\overline{\Gamma \vdash e_1 \; op \; e_2 : \mathtt{Boolean}}$

BOOLEAN NEGATION
$\Gamma \vdash e : \mathtt{Boolean}$

$\overline{\Gamma \vdash \texttt{!}e : \mathtt{Boolean}}$

STRING CONCATENATION
$\Gamma \vdash e_1 : \mathtt{String} \qquad \Gamma \vdash e_2 : \mathtt{String}$

$\overline{\Gamma \vdash e_1 \; \texttt{++} \; e_2 : \mathtt{String}}$

EQUALITY
$\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T$

$\overline{\Gamma \vdash e_1 \; \texttt{==} \; e_2 : \mathtt{Boolean}}$

SEQUENCE
$\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2$

$\overline{\Gamma \vdash e_1 \; \texttt{;} \; e_2 : T_2}$

LOCAL VARIABLE DEFINITION
$\Gamma \vdash e_1 : T_1 \qquad \Gamma, n : T_1 \vdash e_2 : T_2$

$\overline{\Gamma \vdash \texttt{val} \; n \; \texttt{:} \; T_1 \; \texttt{=} \; e_1 \; \texttt{;} \; e_2 : T_2}$

FUNCTION/CLASS CONSTRUCTOR INVOCATION
$\Gamma \vdash e_1 : T_1 \qquad \ldots \qquad \Gamma \vdash e_n : T_n \qquad \Gamma \vdash f : (T_1, \ldots, T_n) \Rightarrow T$

$\overline{\Gamma \vdash f(e_1, \ldots, e_n) : T}$

IF-THEN-ELSE
$\Gamma \vdash e_1 : \mathtt{Boolean} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T$

$\overline{\Gamma \vdash \texttt{if} \; \texttt{(}e_1\texttt{)} \; \{e_2\} \; \texttt{else} \; \{e_3\} : T}$

ERROR
$\Gamma \vdash e : \mathtt{String}$

$\overline{\Gamma \vdash \texttt{error}(e) : T}$

PATTERN MATCHING
$\Gamma \vdash e : T_s \qquad \forall i \in [1, n].\ \Gamma \vdash p_i : T_s \qquad \forall i \in [1, n].\ \Gamma, bindings(p_i) \vdash e_i : T_c$

$\overline{\Gamma \vdash e \; \texttt{match} \; \{ \; \texttt{case} \; p_1 \; \texttt{=>} \; e_1 \; \ldots \; \texttt{case} \; p_n \; \texttt{=>} \; e_n \; \} : T_c}$

Figure 3: Typing rules for expressions

WILDCARD PATTERN

$$\overline{\Gamma \vdash \_ : T}$$

IDENTIFIER PATTERN

$$\overline{\Gamma \vdash v : T}$$

CASE CLASS PATTERN

$$\frac{\Gamma \vdash p_1 : T_1 \qquad \ldots \qquad \Gamma \vdash p_n : T_n \qquad \Gamma \vdash C : (T_1, \ldots, T_n) \Rightarrow T}{\Gamma \vdash C(p_1, \ldots, p_n) : T}$$

FUNCTION DEFINITION

$$\frac{\Gamma, v_1 : T_1, \ldots, v_n : T_n \vdash e : T}{\Gamma \vdash \ \texttt{def} \ f(v_1 : T_1, \ldots, v_n : T_n) : T \ = \ \{ \ e \ \}}$$

PROGRAM

$$\frac{\forall f \in p. \ \Gamma_0(p) \vdash f}{\vdash p}$$

Figure 4: Typing rules for patterns, functions and programs

```scala
object Std {
  // Output
  def printString(s: String): Unit = ...
  def printInt(i: Int): Unit = ...
  def printBoolean(b: Boolean): Unit = ...

  // Input
  def readString(): String = ...
  def readInt(): Int = ...

  // Conversions
  def intToString(i: Int): String = ...
  def digitToString(i: Int): String = ...
  def booleanToString(b: Boolean): String = ...
}
```

Figure 5: The `Std` module