ALGT - The manual

Pieter Vander Vennet

Contents

Chapter 1

Overview

This document gives an overview of the *ALGT*-tool. First, a general overview of what the tool does is given and why it was developed. Second, a hands-on tutorial develops a Simply Typed Functional Language (STFL). Thirdly, the reference manual gives an in-depth overview of the possibilities and command line flags. Fourth, some concepts and algorithms are explained more thoroughly, together with properties they use. And at last, the `dynamize` and `gradualize` options are explained in depth, as these are what the master dissertation is about.

This documentation is about version **0.1.26** (`Total Tutorial`), generated on 2017-3-14 .

Chapter 2

What is ALGT?

ALGT (*Automated Language Generation Tool* [1]) is a tool to formally specify any programming language. This is done by first declaring a syntax, using BNF, and then declaring semantics by introducing logical deduction rules, such as evaluation or typing rules. Eventually, properties can be introduced which can be tested. Of course, these can be run.

The tool is kept as general as possible, so any language can be modelled.

[1]Note the similarity of *ALGT* and *AGT* . The tool started its life as *AGT*, based on the paper *Abstracting Gradual Typing*. When it became more general, another name and acronym was chosen.

Chapter 3

Tutorial: developing a simple programming language

We will develop a programming language which can work with booleans and integers. Apart from doing basic arithmetic, we can also use anonymous functions.

Example programs are:

```
True
False
If True Then False Else True
If If True Then False Else True Then True Else False
42
20 + 22
1 + 2 + 3
(\x : Int . x + 1) 41
```

The expression `(\x : Int . x + 1)` is a *lambda expression*. This is an anonymous function, taking one argument - named x- of type `Int`. When applied (e.g. `(\x : Int . x + 1) 41`, the expression right of the . is returned, with the variable `x` substituted by the argument, becoming `41 + 1`.

3.1 Setting up a language

A language is declared inside a `.language` file [1]. Create `STFL.language`, and put a title in it:

```

 STFL
******

# A Simply Typed Functional Language
```

You can put comments wherever you want after a `#`, e.g. to give some explanation about the language

3.2 Declaring the syntax

For a full reference on syntax, see the reference manual on syntax.

3.2.1 Simple booleans

A program is nothing more then a string of a specific form. To describe strings of an arbitrary structure, *BNF* [2] can be used.

The syntax of our programming language is defined in the **Syntax**section of *STFL.language*:

[1] Actually, the extension doesn't matter at all.

[2] Backus-Naur-form, as introduced by John Backus in the ALGOL60-report.

```

 STFL
******

# A Simply Typed Functional Language

 Syntax
========
```

What do we write here? Let's start with declaring the boolean values `True` and `False`. We express how these can be parsed by writing `bool    ::= "True" | "False"`. This tells the tool that a syntactic form named `bool` exists and it is either `True` of `False`. Note the double quotes, these are important to indicate that we want this string literally. The `|` epxresses that it can choose between those forms.

STFL.language now looks like:

```

 STFL
******

# A Simply Typed Functional Language

 Syntax
========

bool    ::= "True" | "False"
```

Lets try running this! Create `examples.stfl`, which contains:

```
True
False
```

We can parse these by running (in your terminal) `./ALGT STFL.language examples.stfl bool -l`. The first argument is the language file, the second the examples, the `bool` tells ALGT what syntactic rule to parse. The `-l` flag expresses that each line should be treated individually.

If all went well, you should get the following output:

```
# "True" was parsed as:
True    bool.0
# "False" was parsed as:
False    bool.1
```

The most interesting part here is that `True` has been parsed with `bool.0`, thus the first choice of the `bool`-form, while `False` has been parsed with the second form.

3.2.2 If-statements

Now, let's add expressions of the form `If True Then False Else True`. We define a new syntactic form: `expr    ::= "If" bool "Then" bool "Else" bool`.[3] This tell *ALGT* that an expression starts with a literal `If`, is followed by a `bool` (so either `True` or `False`), is followed by a literal `Then`, ... The tool uses the double quotes `"` to distinguish between a literal string and another syntactic form.

STFL.language now looks like:

```

 STFL
******

# A Simply Typed Functional Language

 Syntax
========

bool    ::= "True" | "False"
expr    ::= "If" bool "Then" bool "Else" bool
```

This captures already some example expressions. Let's add `If True Then False Else True` to *examples.stfl*:

[3]Don't worry about spaces and tabs, we deal with them. If you want need to parse stuff like "duizendeneen" or whitespace sensitive languages, please refer to the reference manual

```
True
False
If True Then False Else True
```

Let's run our tool, this time with `./ALGT STFL.language examples.stfl` **expr** `-l`

Oops! Seems like our parser now always wants to see a `If` in the beginning, and can't handle `True` anymore. Perhaps we should tell that a `bool` is a valid expression to:

```

 Syntax
========

bool     ::= "True" | "False"
expr     ::= "If" bool "Then" bool "Else" bool
          | bool
```

Lets see what this gives:

```
+  If        expr.0
|  True      bool.0
|  Then      expr.0
|  False     bool.1
|  Else      expr.0
|  True      bool.0
```

Looks a lot better! The third example shows clearly how the expression falls apart in smaller pieces. What with a nested `If`?

`If If True Then False Else True Then True Else False` clearly can't be parsed, as the condition should be a `bool`, according to our current syntax.

Well, we can just write `expr` instead of `bool` in our syntax:

```
expr     ::= "If" expr "Then" expr "Else" expr
          | bool
```

Running this gives

```
# "If If True Then False Else True Then True Else False" was parsed as:
+  If           expr.0
|  +  If        expr.0
|  |  True      bool.0
|  |  Then      expr.0
|  |  False     bool.1
|  |  Else      expr.0
|  |  True      bool.0
|  Then         expr.0
|  True         bool.0
|  Else         expr.0
|  False        bool.1
```

This clearly shows how the parse trees are nested. This can be rendered too:[4]

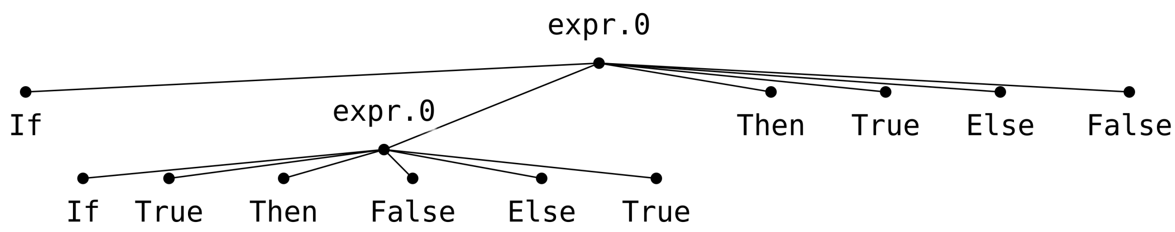


Figure 3.1: ParseTree of a nested condition[5]

[4]These images can be created with `ALGT STLF.language examples.stfl -l --ptsvg Outputname`
[5]These images can be created with `ALGT STLF.language examples.stfl -l --ptsvg Outputname`

3.2.3 Adding numbers, subtyping and forbidden left recursion

Time to spice things up with numbers. To make things easier, integers are built in as `Number`. It's good practice to introduce a new syntactic rule for them:

```
int      ::= Number
```

As an `int` is a valid expression, we add it to the `expr` form:

```
expr     ::= "If" expr "Then" expr "Else" expr
           | bool
           | int
```

Note that every `int` now also is an `expr`, just as every `bool` is an `expr`. This typing relationship can be visualized with `ALGT STFL.language -lsvg Subtyping.svg`[6] :

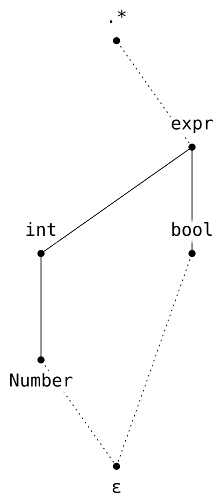


Figure 3.2: Subtyping relation of STFL.language

Now that numbers have been added, let's run this with a number as example:

```
42
```

should give

```
# "42" was parsed as:
42    Number.0
```

So far, so good! Time to add addition:

```
expr     ::= "If" expr "Then" expr "Else" expr
           | bool
           | int
```

expr "+" expr
We add some example:

```
20 + 22
1 + 2 + 3
```

And run it:

```
Error:
  While checking file STFLrec.language:
    While checking the syntax:
      Potential infinite left recursion detected in the syntax.
      Left cycles are:
          expr -> expr
```

[6]Creating this svg might take a long time for complicated syntaxes, as ALGT calculates the ordering of labels resulting in the least intersecting lines.

Oops! Looks like we did something wrong. What is this left recursion? Whenever the parser wants to parse an expression, it tries every choice from left to right. This means that whenever it tries to parse `expr`, it should first try to parse `expr`. That's not really helpfull, so the parser might get in an infinite loop then.

Not allowing left recursion also means that no loops in the subtypings occur. In other words, the subtyping relationship is a lattice.

The solution to this problem is splitting `expr` in two parts: a `term` with simple elements in front, and `expr` with advanced forms:

```
expr    ::= term
        | term "+" expr
term    ::= "If" expr "Then" expr "Else" expr
        | bool
        | int
```

Let's retry this:

```
Error:
  While checking file STFLWrongOrder.language:
    While checking the syntax:
      While checking for dead choices in expr:
        The choice 'term "+" expr' will never be parsed.
        The previous choice 'term' will already consume a part of it.
        Swap them and you'll be fine.
```

What went wrong this time? The parser tries choice after choice. When parsing `20 + 22` against `expr ::= term | term "+" term`, it'll first try `term` (and not `term "+" term`). It succesfully parses `20` against the lonely `term`, thus the input string `+ 22` is left. The parser doesn't know what to do with this leftover part, so we get an error.

To fix this, we change the order:

```
expr    ::= term "+" expr
        | term
```

When we try again, we get:

```
# "20 + 22" was parsed as:
+  20    Number.0
|  +     expr.0
|  22    Number.0
# "1 + 2 + 3" was parsed as:
+  1       Number.0
|  +       expr.0
|  +  2    Number.0
|  |  +    expr.0
|  |  3    Number.0
```

3.2.4 Lambda expressions

The lambda expression is the last syntactic form we'd like to add. Recall that these look like `(\x : Int . x + 1)`.

Variables

The first thing we should deal with, are variables. A builtin is provided for those, namely `Identifiers`, matching all words starting with a lowercase (matching `[a-z][a-zA-Z0-9]*`). Let's introduce them in our syntax:

```
var     ::= Identifier
```

A `var` is valid in expressions too, e.g. in the expression `x + 1`, so we want to add it to our `term`:

```
term    ::= "If" expr "Then" expr "Else" expr
        | bool
        | int
        | var
```

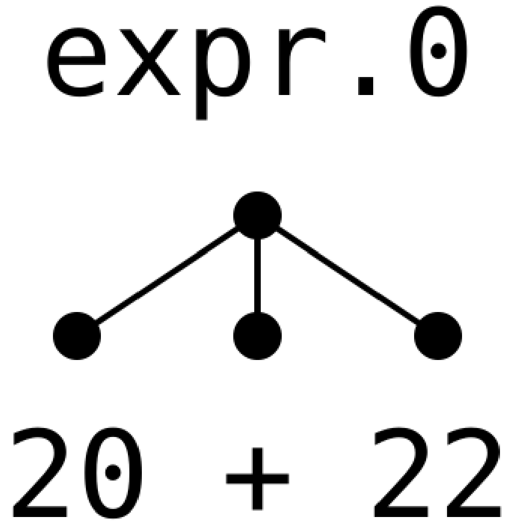


Figure 3.3: Parsetree of `20+22`

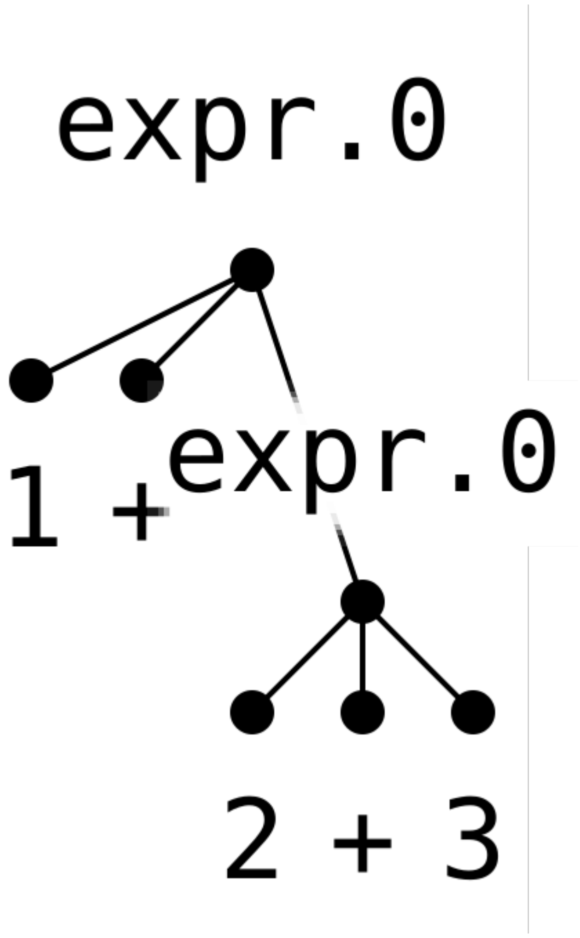


Figure 3.4: Parsetree of `1 + 2 + 3`

Types

The second ingredient we still need, are types, to annotated the input types. Valid types, for starters, are `Bool` and `Int`.

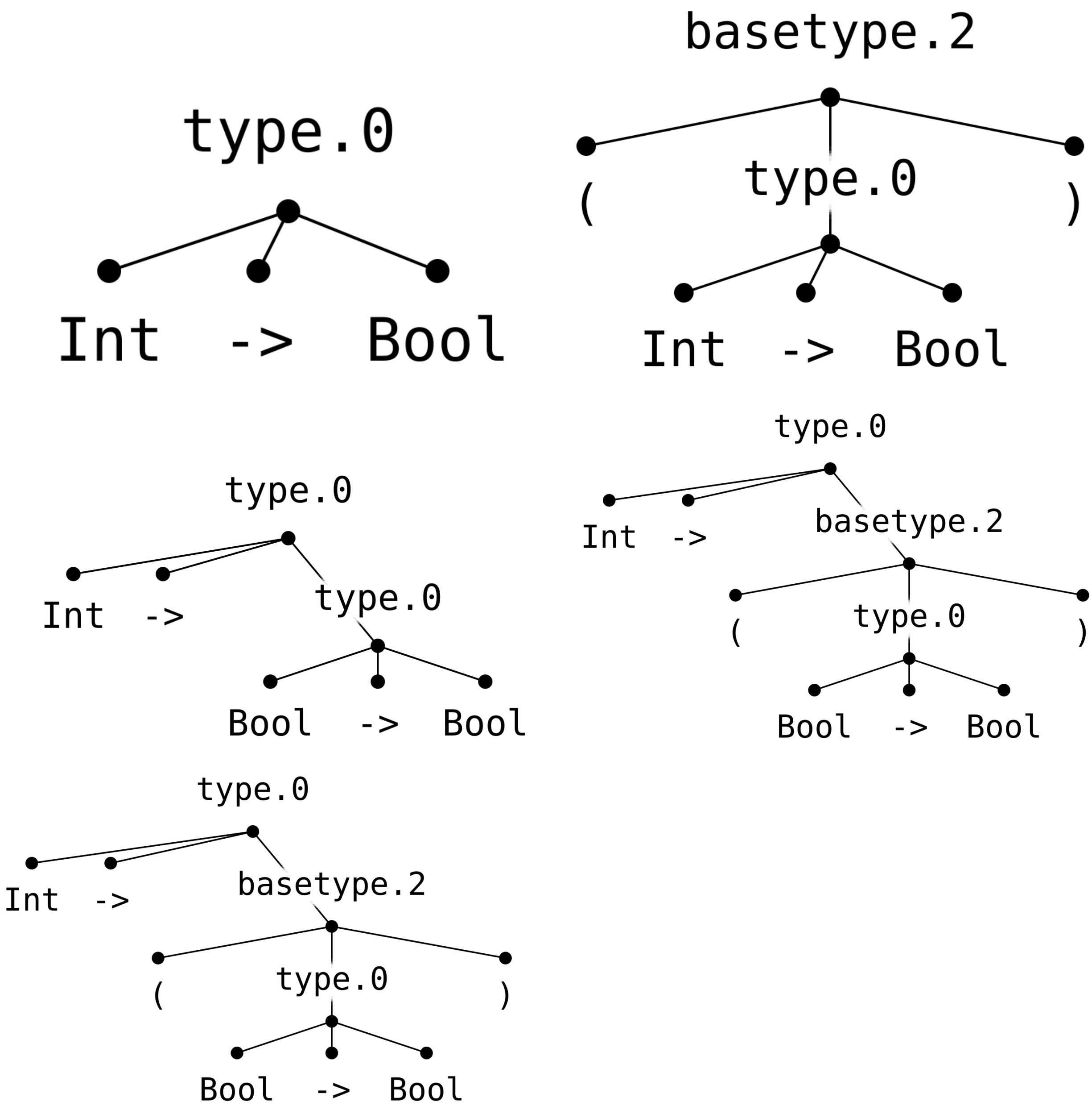
But what is the type of `(\x : Int . x + 1)`? It's something that takes an `Int` and gives back an `Int`. We type this as `Int -> Int`.

And what is the type of a function, taking another function as an argument? That would be, for example, `(Int -> Int) -> Int`, meaning we need to add a form with parentheses.

Recalling the trouble we had with left recursion and ordering, we write `type` as following:

```
basetype::= "Bool" | "Int" | "(" type ")"
type    ::= basetype "->" type | basetype
```

Some examples of types are:



3.2.5 Lambda expressions

Now we have what we need to define lambda expressions. As they act as a term, we add it there:

```
term     ::= "If" expr "Then" expr "Else" expr
         | "(" "\\" var ":" type "." expr ")"
         | bool
         | int
         | var
```

Backslash is the escape character, so use two of them to represent a single backslash.

We can also apply arguments to a lambda expression. We expand `expr`:

```
expr     ::= term "+" expr
         | term expr
         | term
```

3.2.6 What about nonsensical input?

With the current syntax, expresions as `If 5 Then True else False`, `True + 5`, `True 5` or `(\x : Int : x + 1) True` can be written. We allow these forms to be parsed, as the next stage of the compiler (the typechecker) will catch these errors. How to construct this, will be explained in a following section.

3.2.7 Recap

Our *STFL.language* contains

```

 STFL
******

# A Simply Typed Functional Language

 Syntax
========

basetype::= "Bool" | "Int" | "(" type ")"
type     ::= basetype "->" type | basetype

bool     ::= "True" | "False"
int      ::= Number
var      ::= Identifier

expr     ::= term "+" expr
         | term expr
         | term


term     ::= "If" expr "Then" expr "Else" expr
         | "(" "\\" var ":" type "." expr ")"
         | bool
         | int
         | var
```

Our *examples.stfl* contains

```
True
False
If True Then False Else True
If If True Then False Else True Then True Else False
42
20 + 22
1 + 2 + 3
(\x : Int . x + 1) 41
```

We run this with

- `ALGT STFL.language examples.stfl expr -l` to show the parsetrees
- `ALGT STFL.language examples.stfl expr -l --ptsvg SVGnames` to render the parsetrees as SVG
- `ALGT STFL.language --lsvg SVGname.svg` to visualize the subtyping relationship.

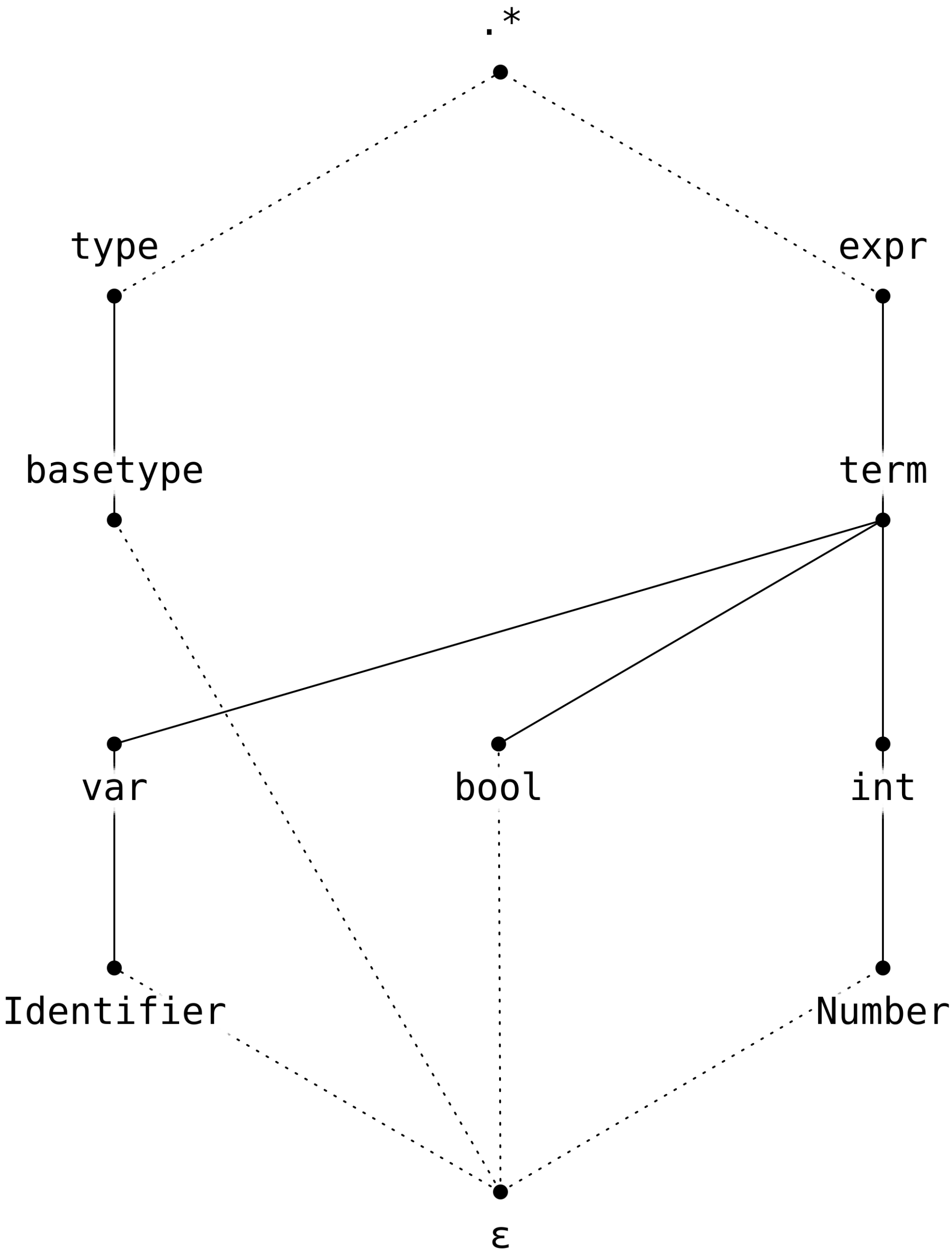


Figure 3.5: The final subtyping relationship of STFL.language

3.3 Functions

For a full reference, see the reference manual on functions.

3.3.1 Domain and codomain

It'll come in handy later on to be able to calculate the *domain* and *codomain* of a function type. The *domain* of a function is the type it can handle as input. The *codomain* of a function is the type it gives as output.

Table 3.1: Examples of domain and codomain

Function type	dom	cod
`Int -> Bool`	`Int`	`Bool`
`(Int -> Bool)`	`Int`	`Bool`
`Int -> Bool -> Bool`	`Int`	`Bool -> Bool`
`Int -> (Bool -> Bool)`	`Int`	`Bool -> Bool`
`(Int -> Bool) -> Bool`	`Int -> Bool`	`Bool`
`Int`	*Undefined*	*Undefined*
`Bool`	*Undefined*	*Undefined*

Now, let's define these functions!

3.3.2 The function section

We add a new header to *STFL.language*:

```
 Functions
===========
```

In this function section, we can define the function *domain* in the following way:

```
domain                            : type -> type
domain(T1 "->" T2)                = T1
```

So, what is going on here? Let's first take a look to the first line:

```
domain                            : type -> type
```

The `domain` is the name of the function. The `type -> type` indicates what syntactic form is taken as input (a `type`) before the `->` and what is given as output (again a `type`). You probably noticed the similarity between the types declared in our own STFL and this declaration. This is intentional. This is quite meta, don't get confused!

Figure 3.6: Relevant XKCD (by Randall Munroe, #917)

3.3.3 Pattern matching

Let's have look at the body of the function:

```
domain(T1 "->" T2)                = T1
```

What happens if we throw in `Int -> Bool`? Remember that this is parsed as a tree, with three leafs. Notice that there are three elements in the pattern match too: a variable `T1`, a literal `->` and a variable `T2`. These leafs are matched respectively:

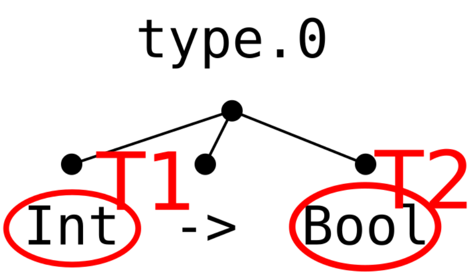


Figure 3.7: Pattern matching in action

The same principle applies with more advanced inputs:

We thus always bind `T1` to the part before the top-level `->`, in other words: we always bind the input type (or domain type) to `T1`. As that is exactly what we need, we return it!

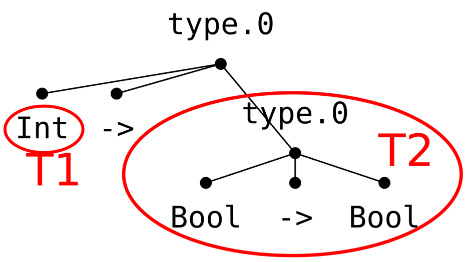


Figure 3.8: Pattern matching in action (more advanced)

3.3.4 Missing cases

This already gives us the most important part. However, what should we do if the first argument is a function type (e.g. `(Int -> Bool) -> Bool`)?

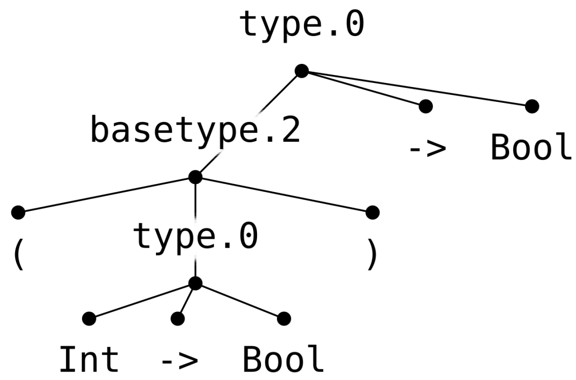


Figure 3.9: Typetree of function argument

This will match the pattern `T1 "->" T2` as following:

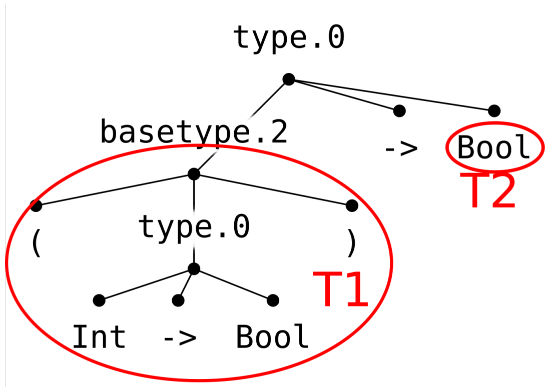


Figure 3.10: Typetree of function argument, matched

We see that `T1` includes the `(` and `)`, which we don't want. We simply solve this by adding a extra clause:

```
domain                          : type -> type
domain(("(" T1 ")") "->" T2)    = T1
domain(T1 "->" T2)              = T1
```

We expect that the part *before* the arrow now is surrounded by parens. Note that we put parens once without double quotes and once with. These parens capture this part of the parsetree and match it against the patterns inside the parens, as visible in the green ellipse:

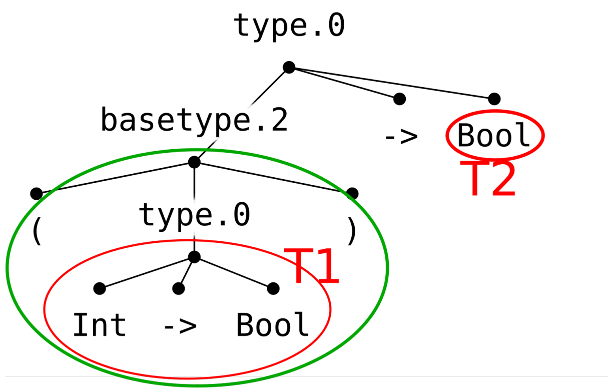


Figure 3.11: Recursive pattern matching

3.3.5 Recursion

There is a last missing case, namely if the entire type is wrapped in parens.

We match a type between parens, and then calculate its domain simply by calling the domain function again.

```
domain                          : type -> type
domain("(" T ")")               = domain(T)
domain(("(" T1 ")") "->" T2)    = T1
domain(T1 "->" T2)              = T1
```

3.3.6 Clause determination

As you can see, there are three clauses now. What clause is executed?

Simply put, when the function is evaluated, the arguments are pattern matched against the first clause. If this pattern match succeeds, the expression on the right hand side is returned. Do the arguments not match the patterns? Then the next clause is considered.

In other words, clauses are tried **from top to bottom** and tested. The first clause of which the patterns match, is executed.

When no clauses match, an error message is given.

3.3.7 Executing functions

Let's put this to a test. We can calculate the domain of our examples.

Create a file `typeExamples.stfl`, with contents

```
Int -> Bool
(Int -> Bool)
Int -> Bool -> Bool
Int -> (Bool -> Bool)
(Int -> Bool) -> Bool
Int
Bool
```

Run this with `ALGT STFL.language typeExamples.stfl type -l -f domain`:

```
While checking file STFL.language:
  Warning:
    While checking the totality of function "domain":
      Following calls will fall through:
        domain("Bool")
        domain("Int")
```

```
# "Int -> Bool" applied to domain
Int

# "(Int -> Bool)" applied to domain
Int

# "Int -> Bool -> Bool" applied to domain
Int

# "Int -> (Bool -> Bool)" applied to domain
Int

# "(Int -> Bool) -> Bool" applied to domain
Int -> Bool

# "Int" applied to domain
Not a single clause of domain matched:
  While pattern matching clause 0:
       In Pattern matching clause 0 with arguments: (Int)
       In domain(Int)
    :
      FT: Could not pattern match '"Int"' over '"(" T ")"'
  While pattern matching clause 1:
       In Pattern matching clause 1 with arguments: (Int)
       In domain(Int)
    :
      FT: Could not pattern match '"Int"' over '("(" T1 ")") "->" T2'
  While pattern matching clause 2:
       In Pattern matching clause 2 with arguments: (Int)
       In domain(Int)
    :
      FT: Could not pattern match '"Int"' over 'T1 "->" T2'

# "Bool" applied to domain
Not a single clause of domain matched:
  While pattern matching clause 0:
       In Pattern matching clause 0 with arguments: (Bool)
       In domain(Bool)
    :
      FT: Could not pattern match '"Bool"' over '"(" T ")"'
  While pattern matching clause 1:
       In Pattern matching clause 1 with arguments: (Bool)
       In domain(Bool)
    :
      FT: Could not pattern match '"Bool"' over '("(" T1 ")") "->" T2'
  While pattern matching clause 2:
       In Pattern matching clause 2 with arguments: (Bool)
       In domain(Bool)
    :
      FT: Could not pattern match '"Bool"' over 'T1 "->" T2'
```

What does this output tell us?

For starters, we get a warning that we forgot two cases, namely `Bool` and `Int`. For some functions this is a problem, but domain is not defined for those values. Note that a bunch of other tests are builtin as well.[7]

Then, we see an overview for each function what result it gives (or an error message if the pattern matches failed).

If you want more information about the behaviour of a function, specify `--ia` or `--ifa FUNCTION-TO-ANALYZE` to get a clause-per-clause overview:

```


 Analysis of domain : type -> type
===================================



    Analysis of clause 0
```

[7]We'll silently ignore these warnings for the rest of the tutorial with flag `--no-checks`

```
  .....................

  Clause:
    domain("(" T ")")         = domain(T)

  Possible inputs at this point:
  #  (type)


  Possible results:
  0   ("(" type(0/1/2:1) ")"): basetype/2     --> type( (Function call - ID not retrieva




   Analysis of clause 1
  .....................

  Clause:
    domain(("(" T1 ")") "->" T2)
                              = T1

  Possible inputs at this point:
  #  ("Bool")
  #  ("Int")
  #  ((basetype "->" type))


  Possible results:
  1   (("(" type(0/0:0/2:1) ")"): basetype/2 "->" type(0/0:2)): type/0        --> type((




   Analysis of clause 2
  .....................

  Clause:
    domain(T1 "->" T2)        = T1

  Possible inputs at this point:
  #  ("Bool")
  #  ("Int")
  #  (("Bool" "->" type))
  #  (("Int" "->" type))


  Possible results:
  2   ("Bool" "->" type(0/0:2)): type/0       --> "Bool"          : "basetype"
  2   ("Int" "->" type(0/0:2)): type/0        --> "Int"           : "basetype"




 Falthrough
------------

("Bool")
("Int")
```

3.3.8 Codomain

`codomain` can be implemented analogously:

```
codomain                        : type -> type
codomain("(" T ")")             = codomain(T)
```

```
codomain(T1"->" ("(" T2 ")"))    = T2
codomain(T1 "->" T2)             = T2
```

3.4 Relations and Rules: building the evaluator

While we could build a function which evaluates our programming languages, language designers love *natural deduction* more. Don't worry if you never heard about that before, we'll explain it right away!

3.4.1 Natural deduction

Natural deduction is about defining **relations**. After declaring a relation, the elements that are part of the relation are defined by an **inference rule**, which looks like:

```
 Given predicates
------------------ [Inference Rule Name]
 Given conclusion
```

A conclusion would be, in our case, that certain elements are part of a relation.

This is Turing complete just as well. We yield this power to build the evaluator.

3.4.2 Declaring evaluation

First, we declare a new section inside our *STFL.language*, with a relation declaration inside:

```
 Relations
===========

(→)     : expr (in), expr (out)         Pronounced as "small step"
```

Let us break this line down.

The first part, `(→)`, says that we declare a relation with name →. Except from some builtin symbols, you can use whatever string you want, including unicode[8]. If you don't want to use the unicode-arrow for this tutorial, you can replace → by `->`.

The second part, `: expr (in), expr (out)` states that this is a relation between two `expr`. As example, `2 + 3 , 5` will be in `(→)` or written more conventionally `2 + 3 → 5`.

What about the `(in)` and `(out)` parts? These are called the **mode** of the argument and are to help the computer. Given `2 + 3`, it's pretty easy for the computer to calculate `5`. Given `5`, the computer can't magically deduce that this was computed by calculating `2 + 3`, especially because an infinite amount of possible calculations lead to the result `5`.

The last part, `Pronounced as "evaluation"` defines a name for the relation. It's documentation, to help users of your language to know what a relation is supposed to do or to help them searching it on a search engine.

Note that the goal of → is to make a small, fundamental step - just one addition or simplification, e.g. `1 + 2 + 3 → 1 + 5`. We'll design another relation later on which will give us the end result immediatly, giving us `6`.

3.4.3 Defining evaluation

Simple deduction rules: If

Defining relations works with one or more rules.

We start with a simple one:

```

----------------------------------------          [EvalIfTrue]
 "If" "True" "Then" e1 "Else" e2 → e1
```

[8] to enter an unicode character on a linux machine, type `Left-Ctrl + Shift + U`, release, and type the hexcode of the desired character, e.g. `2192` to get the right-arrow. On windows, hold down `Alt` and type `+ 2192` (thus: type a plus, followed by typing the number).

How should you read this rule? The part under the line says that this is part of a relation, namley →; in other words; `"If" "True" "Then" e1 "Else" e2` will evaluate to `e1`. This is equivalent to writing the function clause `eval("If" "True" "Then" e1 "Else" e2) = e1`.

The part right of the line (namely `[EvalIfTrue]`) gives the name of the rule. You can use whatever you want, it is documentation as well.

Analogously, you can add a rule for `If False`:

```

----------------------------------------        [EvalIfFalse]
 "If" "False" "Then" e1 "Else" e2 → e2
```

This is already enough to run our third example. To run a relation, specify `-r <name-of-relation>`, thus `./ALGT STFL.language examples.stfl expr -l -r →`:

```
# If True Then False Else True applied to →
# Proof weight: 1, proof depth: 1


------------------------------------ [EvalIfTrue]
If True Then False Else True → False
```

Deduction rules with predicates: plus

How do we evaluate expressions with `+`? We can add a deduction rule for addition too:

```
 n1:Number              n2:Number
------------------------------------    [EvalPlus]
 n1 "+" n2 → !plus(n1, n2)
```

First, take a look at the bottom line. The left part is straightforward; we match a parsetree with form `n1 + n2`. But what is `!plus(n1, n2)`? It's a function call with arguments `n1` and `n2`. The exclamation mark `!` indicates that this is a builtin function[9].

In other words, this rule indicates that `1 + 2` should be evaluated with `!plus(1,2)`, giving `3`.

There is a catch, though. `!plus` has type `Number -> Number -> Number` (recall, this means that plus takes two `Numbers` and gives us a `Number` in return). We can't pass in other types, or it would fail. We thus have to check that we get correct input for this rule. To do this, we have those predicates on top: `n1:Number` and `n2:Number`. Read `n1:Number` as *`n1` is of syntactic form `Number`*.

Let give this a run!

```
# 20 + 22 applied to →
# Proof weight: 3, proof depth: 2


20 : Number    22 : Number
-------------------------- [EvalPlus]
20 + 22 → 42




# Could not apply relation → to the input "1 + 2 + 3", because:
While trying to proof that (→) is applicable to "1 + 2 + 3":
  Not a single rule matched:
  While trying to intepret the rule EvalPlus with 1 + 2 + 3:
    n2 = 2 "+" 3 is not a "Number" but a "expr"
  While trying to intepret the rule EvalIfTrue with 1 + 2 + 3:
    Sequence lengths are not the same: "If" "True" "Then" e1 "Else" e2 /= 1 "+" (2 "+" 3)
  While trying to intepret the rule EvalIfFalse with 1 + 2 + 3:
    Sequence lengths are not the same: "If" "False" "Then" e1 "Else" e2 /= 1 "+" (2 "+" 3
```

We can see that our simple example, `20 + 22` neatly gives us the answer[10]. The other example, `1 + 2 + 3`, fails, giving a detailed overview of what rules it attempted to apply and why those rules failed.

[9] An overview for all builtin functions can be found in the reference manual.

[10] Luckily, your computer didn't have to run for 10 million years. And it conveniently gave the question too, so that we wouldn't forget it.

If with complicated conditions

But what with our fourth example, `If If True Then False Else True Then True Else False`? The condition itself as an `If`-expression as well.

Herefore we introduce a more complicated rule:

```

 cond0 → cond1
----------------------------------------------------- [EvalIfCond]
 "If" cond0 "Then" e1 "Else" e2 → "If" cond1 "Then" e1 "Else" e2
```

This rule states that, whenever `cond0` evaluates to `cond1`, then we can evaluate the bigger expression.

We might also introduce two similar rules for, for evaluating the arguments of `+`, so that `1 + (2 + 3)` can be evaluated too. However, it is cumbersome to add all these extra rules for each syntactic form.

Evaluation contexts for congruence rules

Luckily, there is a way to write all those rules even shorter:

```

 expr0 → expr1
----------------------------                [EvalCtx]
 expr[expr0] → expr[expr1]
```

The part `expr[expr0]` will search, within the expression we want to evaluate, a nested expression that satisfies the conditions. In other words, it will search in the parsetree (e.g. `If (If True Then False Else False) Then True Else` a part that can be evaluated (e.g. `If True Then False Else False`). This part will be evaluated (to `False`) and plugged back in the bigger expression.

The evaluated expression will then be put back in the original, bigger parsetree at the same location.

Make sure to name the nested expr `expr0`, thus *syntactic-form-name* followed by a number. That's how the tool figures out what kind of parsetree to search for.

```
# If If True Then False Else True Then True Else False applied to →
# Proof weight: 2, proof depth: 2


------------------------------------
If True Then False Else True → False
------------------------------------------------------------------------------------
If If True Then False Else True Then True Else False → If False Then True Else False
```

As expected, this rule also solves our `1 + 2 + 3`!

```
# 1 + 2 + 3 applied to →
# Proof weight: 4, proof depth: 3


2 : Number    3 : Number
------------------------ [EvalPlus]
2 + 3 → 5
------------------------- [EvalCtx]
1 + 2 + 3 → 1 + 5
```

These proofs are getting a bit harder to read. If you get lost, remember to always start from the bottom.

This proof states that `1 + (2 + 3)` makes a single step to `1 + 5`, because of rule `EvalCtx`; this rule could be used because `2 + 3` evaluates to `5`.

The proof for that part of the evaluation is given on top, by rule `EvalPlus`, which could be invoked because both `2` and `3` are `Numbers`.

Application

As last, we'd like to apply functions, such as `(\\x : Int . x + 1) 41`.

Our intution is that, given something as `(\\x : someType . someExpr) someArg`, we want to evaluate this to `someExpr`, where we replace every `x` in `someExpr`. Luckily, a builtin function does the hard part of replacing for us: `!subs`. This gives us the following rule:

```
----------------------------------------------------- [EvalLamApp]
 ("(" "\\" var ":" type "." e ")") arg → !subs:expr(var, arg, e)
```

```
# (\x : Int . x + 1) 41 applied to →
# Proof weight: 1, proof depth: 1


--------------------------------- [EvalLamApp]
( \ x : Int . x + 1 ) 41 → 41 + 1
```

3.4.4 Evaluation-relation: recap

Our evaluation rule is defined as:

```

 Relations
===========

(→)      : expr (in), expr (out)         Pronounced as "small step"


 Rules
=======


 expr0 → expr1
----------------------------            [EvalCtx]
 expr[expr0] → expr[expr1]



 n1:Number               n2:Number
------------------------------------    [EvalPlus]
 n1 "+" n2 → !plus(n1, n2)




---------------------------------------         [EvalIfTrue]
 "If" "True" "Then" e1 "Else" e2 → e1


----------------------------------------        [EvalIfFalse]
 "If" "False" "Then" e1 "Else" e2 → e2


----------------------------------------------------- [EvalLamApp]
 ("(" "\\" var ":" type "." e ")") arg → !subs:expr(var, arg, e)
```

3.4.5 Is canonical

It is usefull to known when an expression is *canonical*, thus is fully evaluated. This can be simply stated by a relation taking just one argument and *giving no output*[11]. It should contain exactly the `Int`s, `True` and `False` (the `bool`s for short).

First, the declaration in the `Relations`-section:

```
(✓)      : expr (in)                     Pronounced as "is canonical"
```

And the implementation in the `Rule`-section:

```
 i:int
-----------                     [CanonInt]
 (✓) i


 b:bool
```

[11] Mathematicians would call this a *set*.

```
-----------                          [CanonBool]
 (✓) b
```

3.4.6 Bigstep

Of course, when we input `1 + 2 + 3`, we would like te get `6`, and not `1 + 5`. For this, we can define a third relation,

First, let us declare `bigstep`:

```
(→*)    : expr (in), expr (out)          Pronounced as "big step"
```

The implementation is based on recursion. If something is canonical, we are done and just return the unchanged value. We express this basecase as following:

```
 (✓) e
---------                            [BigStepBase]
 e →* e
```

What if we are not done? That means that we can make a single step, `e0 →e1` and that we calculate this `e1` to its canonical from `e2` with `bitstep` itself!

```
 e0 → e1         e1 →* e2
-------------------------            [BigStepRec]
 e0 →* e2
```

So, we finally did it! Time to see our examples in all their glory!

```
# True applied to →*
# Proof weight: 3, proof depth: 3


True : bool
----------- [CanonBool]
(✓) True
----------------------- [BigStepBase]
True →* True
```

```
# False applied to →*
# Proof weight: 3, proof depth: 3


False : bool
------------ [CanonBool]
(✓) False
------------------------ [BigStepBase]
False →* False
```

```
# If True Then False Else True applied to →*
# Proof weight: 5, proof depth: 4


                                        False : bool
                                        ------------
                                        (✓) False
------------------------------------    --------------
If True Then False Else True → False    False →* False
------------------------------------------------------
If True Then False Else True →* False
```

```
# 42 applied to →*
# Proof weight: 3, proof depth: 3


42 : int
-------- [CanonInt]
(✓) 42
------------------- [BigStepBase]
42 →* 42
```

```
# 20 + 22 applied to ->*
# Proof weight: 7, proof depth: 4


                              42 : int
                              --------
20 : Number    22 : Number    (✓) 42
--------------------------    --------
20 + 22 -> 42                 42 ->* 42
---------------------------------------
20 + 22 ->* 42
```

```
# 1 + 2 + 3 applied to ->*
# Proof weight: 12, proof depth: 5


                                                      6 : int
                                                      -------
2 : Number    3 : Number    1 : Number    5 : Number  (✓) 6
------------------------    ------------------------  -------
2 + 3 -> 5                  1 + 5 -> 6                6 ->* 6
------------------------    ---------------------------------
1 + 2 + 3 -> 1 + 5          1 + 5 ->* 6
-------------------------------------------------------------
1 + 2 + 3 ->* 6
```

```
# (\x : Int . x + 1) 41 applied to ->*
# Proof weight: 9, proof depth: 5


                                                                  42 : int
                                                                  --------
                                    41 : Number    1 : Number     (✓) 42
                                    -------------------------     --------
                                    41 + 1 -> 42                  42 ->* 42
---------------------------------   --------------------------------------
( \ x : Int . x + 1 ) 41 -> 41 + 1  41 + 1 ->* 42
--------------------------------------------------------------------------
( \ x : Int . x + 1 ) 41 ->* 42
```

3.5 Building the typechecker

Now that we have some experience with natural deduction, we can slay the next dragon: the typechecker!

For those unfamiliar, the typechecker looks at the expression and determines the type of it and halts on inconsistencies, such as `1 + True` or `If True Then 0 Else False`, ...

we will build a single rule for each syntactic choice of `expr`; thu a rule for:

- The constants `True` and `False`
- The constant `Numbers`
- Typing plus
- Typing `If ... Then ... Else ...`
- Typing lambda's `(\\x : T . e)`
- Typing variables `x`
- Typing application

3.5.1 The typing environment

Before where start, how should we type a variable, such as `x`? Of course, this depends on the environment. In the lambda `(\\x : Int . x)`, `x` should be typed as a `Int`, while in the lambda `(\\ x : Bool . x)`, this `x` clearly is a `Bool`.

We could type the inner expressions by substituting a simple default value in the expression, and then typing it. However, this doesn't scale to more advanced languages.

The other, more general solution is keeping track of the type of each variable. We declare a simple list to keep track of the types in the `Syntax Section`:

```
typing                    ::= var ":" type
typingEnvironment         ::= typing "," typingEnvironment | "{}"
```

The `typing` represents a data entry, whereas the `typingEnvironment` can contain zero or more of these data entries, thus keeping track of the variable types.

The typing relation is often denoted with a uppercase gamma, Γ. We will follow this convention[12].

3.5.2 The typing relation

We're all set now! What should our typing relation look like? First, we'll want to take a `typingRelation` as input, together with an `expr`. This should be enough to calculate the `type` of the expression.

In other words, the type of the relation is `typingRelation (in), expr (in), type (out)`. In the acadical world, this is often given the symbol ⊢ [13], pronounced `entails` or `out of this environment follows this typing`.

So, our declaration becomes:

```
(⊢)      : typingEnvironment (in), expr (in), type (out) Pronounced as "entails typing"
```

3.5.3 Typing constants `True` and `False`

It's pretty easy to type constants, such as `42` and `True`.

Let us start with typing the constant `True`.

```

----------------         [TboolTrue]
 Γ ⊢ "True", "Bool"
```

If this looks magical: we take the typing environment as input (but don't use it), pattern match on a literal `True` and return the known type `Bool`.

We can do the same for `False`:

```

----------------         [TboolFalse]
 Γ ⊢ "False", "Bool"
```

3.5.4 Typing constant `Numbers`

Our next challenge is giving a type to `Int`s. Making a single rule for each number is a bit hard, especially because there is an infinite amount of them…

However, we can simply fix this by adding a predicate, checking that our input is a number:

```
 n:int
---------------          [Tnumber]
 Γ ⊢ n, "Int"
```

Remember that our predicates are written above the line.

3.5.5 Typing against an empty environment

Let's try to run our typing relation, with `./ALGT STFL.language expr -l -r ⊢`

```
# Could not apply relation ⊢ to the input "True", because:
While trying to proof that (⊢) is applicable to "True":
  Expected 2 arguments to relation ⊢, but only got 1
```

Well, that didn't work. The tool expects two arguments; but only one is provided…

To solve this, we declare yet another relation, in which we type an expression against an empty environment:

[12] Type `Ctrl+Shift+U 0393` on linux to input `Gamma`. On Windows, hold down `Alt` and type `+ 0393`.
[13] Type `Ctrl+Shift+U 22a2` on linux to input `entails`. On Windows, hold down `Alt` and type `+ 22a2`.

```
 n:int
---------------           [Tnumber]
 Γ ⊢ n, "Int"
```

Retrying with the new relation (`./ALGT STFL.language examples.stfl expr -l -r ::`) gives us:

```
# True applied to ::
# Proof weight: 2, proof depth: 2


--------------- [TboolTrue]
{} ⊢ True, Bool
--------------------------- [TEmptyCtx]
True :: Bool




# False applied to ::
# Proof weight: 2, proof depth: 2


---------------- [TboolFalse]
{} ⊢ False, Bool
----------------------------- [TEmptyCtx]
False :: Bool
```

```
# If True Then False Else True applied to ::
# Proof weight: 6, proof depth: 3


---------------  ----------------  ---------------
{} ⊢ True, Bool  {} ⊢ False, Bool  {} ⊢ True, Bool  T1 = Bool = Tr
-----------------------------------------------------------------
{} ⊢ If True Then False Else True, Bool
-----------------------------------------------------------------
If True Then False Else True :: Bool
```

3.5.6 Typing plus

Another expression we'll want to type are additions.

Of course, this will always return an `Int`, but there is more. `True + False` is not valid, whereas `1 + 1` can be typed. In other words, we have to check that the arguments to `+` are both numbers.

We could thus type `+` as following:

```
 n1:Number         n2:Number
------------------------------                          [TPlus]
 Γ ⊢ n1 "+" n2, "Int"
```

This works for `1 + 2`, but not for `1 + (2 + 3)`, as `(2 + 3)` is *not* a syntactic form that is a literal `Number`. It can be typed as `Int` though, so we can generalize our predicates by using the typing relationship recursively:

```
 Γ ⊢ n1, "Int"  Γ ⊢ n2, "Int"
------------------------------                          [TPlus]
 Γ ⊢ n1 "+" n2, "Int"
```

Looks good! Time to give this a try:

```
# 1 + 2 + 3 applied to ::
# Proof weight: 9, proof depth: 5


               2 : int        3 : int
               -----------    -----------
1 : int        {} ⊢ 2, Int    {} ⊢ 3, Int
-----------    --------------------------
{} ⊢ 1, Int    {} ⊢ 2 + 3, Int
-----------------------------------------
```

```
{} ⊢ 1 + 2 + 3, Int
-----------------------------------------
1 + 2 + 3 :: Int
```

3.5.7 Typing `If`

It is pretty straightforward that the condition should be a `Bool`, which already gives a draft of the rule:

```
 Γ ⊢ cond, "Bool"
----------------------------------------------------- [TIf]
 Γ ⊢ "If" cond "Then" e1 "Else" e2,  ???
```

We also want to make sure that both `e1` and `e2` are correctly typed, so we recursively typecheck them:

```
 Γ ⊢ cond, "Bool"       Γ ⊢ e1, T1       Γ ⊢ e2, T2
----------------------------------------------------- [TIf]
 Γ ⊢ "If" cond "Then" e1 "Else" e2,  ???
```

But what type should we return? The type of `e1` or `e2`?

Consider expression `If True Then 0 Else False`. Evaluating this yields `0`. However, expression `If False Then 0 Else False` would yield `False`. In other words, depending on the runtime value of the condition, we might get a different type.

That's not behaviour we want. The types of `e1` and `e2` should be the same to function correctly. We add a predicate to check this:

```
 Γ ⊢ cond, "Bool"      Γ ⊢ e1, T1     Γ ⊢ e2, T2     T1 = T2
------------------------------------------------------------- [TIf]
 Γ ⊢ "If" cond "Then" e1 "Else" e2,  ???
```

Now we can also return a type; as `T1` and `T2` are the same, we just pick one:

```
 Γ ⊢ c, "Bool"  Γ ⊢ e1, Tl      Γ ⊢ e2, Tr      Tl = Tr : type
--------------------------------------------------------------- [TIf]
 Γ ⊢ "If" c "Then" e1 "Else" e2, Tl
```

All done! Time to give it a try:

```
# If True Then False Else True applied to ::
# Proof weight: 6, proof depth: 3


---------------  ----------------  ---------------
{} ⊢ True, Bool  {} ⊢ False, Bool  {} ⊢ True, Bool  Tl = Bool = Tr
-------------------------------------------------------------------
{} ⊢ If True Then False Else True, Bool
-------------------------------------------------------------------
If True Then False Else True :: Bool
```

3.5.8 Typing lambda's

Typing lambda's is a bit complicated. Remember that a lambda such as `(\\x : T . e)` means that, *given `x` of type `T` as input argument, it gives back expression `e` with `x` replaced.*

This gives us quite some clues about what to do.

Let's start with the skeleton of the rule:

```
 ???
---------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" T1 "." e ")", ???
```

Of course, we'll want to type `e` jus as well. Not only to check wether it is correct, but also because we'll need it's type later on:

```
 Γ ⊢ e, T1
---------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" T1 "." e ")", ???
```

This is close to what we want, but there is a catch though: in the expression `e`, we know that `x` has the type `T1`. We should pass this knowledge to the typing of `e`, by adding it to the typing environment `Γ`:

```
 (x ":" T1) "," Γ ⊢ e, T2
------------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" T1 "." e ")", ???
```

So far, so good! The only question remaining is what type we should return. We know we have input of type `T1` and output of type `T2`. Thats where our `->` comes into play: the entire lambda has type `T1 -> T2`!

There is a little technicality into play here though: `T1` might be some complicated type, such as `Int -> Bool` - meaning we expect a *function* as input argument. If we would write `Int -> Bool -> T2`, that would be read as funcion taking *two* arguments: first a `Int`, followed by a `Bool`. Not quite the same thus.

The fix for this is simple: add parentheses. The type of a lambda is `(T1) -> T2`.

Typing this out as rule yields:

```
 (x ":" T1) "," Γ ⊢ e, T2
---------------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" T1 "." e ")", ( "(" T1 ")") "->" T2
```

Note that an extra pair of parentheses was added; one pair is between double quotes, denoting that this should be added in the parse tree; the other pair just groups them together to help the tool build the parsetree.

3.5.9 Typing variables

Before we can see typing of lambda's live, we need to take another hurdle: typing variables.

You'll probably think it'll be a lot of work to design the searching behaviour, but luckily, there is a special construction that does exactly that. Remember the evaluation context? We can use this builtin here too:

```

--------------------    [Tx]
 Γ[ x ":" T ] ⊢ x, T
```

Quite succint! If you're a bit puzzled about its workings, we stated to ALGT that it should search a typing of `x`, where `x` is exactly the name of the variable we want to type.

At this point, we can finally type a single lambda:

```
# (\x : Int . x + 1) applied to ::
# Proof weight: 6, proof depth: 5


                               1 : int
--------------------- [Tx]    --------------------- [Tnumber]
x : Int , {} ⊢ x, Int         x : Int , {} ⊢ 1, Int
--------------------------------------------------- [TPlus]
x : Int , {} ⊢ x + 1, Int
------------------------------------------------- [TLambda]
{} ⊢ ( \ x : Int . x + 1 ), ( Int ) -> Int
----------------------------------------------- [TEmptyCtx]
( \ x : Int . x + 1 ) :: ( Int ) -> Int
```

3.5.10 Typing application

The typechecker is nearly complete, only a single syntactic form can't be typed yet: application of the form `function argument`.

How can we tackle this problem? For starters, we'll have to type the `function` and `argument`

```
 Γ ⊢ func, TFunc     Γ ⊢ arg, TArg
----------------------------------------------------- [TApp]
 Γ ⊢ func arg, ???
```

A function has a determined input argument, also known as the *domain* of the function. To be well typed, the domain should match the type of the argument exactly. But how can we get this argument?

Luckily, we created a function earlier on that calculates exactly that! We can simply use `domain` and check that it's result equals `TArg`:

```
 Γ ⊢ func, TFunc      Γ ⊢ arg, TArg      domain(TFunc) = TArg
----------------------------------------------------- [TApp]
 Γ ⊢ func arg, ???
```

Nearly done! Only question left is what type we should return.

This is pretty straighforward too, as we earlier made the function `codomain` which exactly calculates the return type of `TFunc`

```
 Γ ⊢ e1, Tfunc   Γ ⊢ e2, Targ    Targ = domain(Tfunc) : type
--------------------------------------------------------------   [Tapp]
 Γ ⊢ e1 e2, codomain(Tfunc)
```

All finished now, except for trying of course:

```
# (\x : Int . x + 1) 41 applied to ::
# Proof weight: 10, proof depth: 6


                         1 : int
---------------------    ---------------------
x : Int , {} ⊢ x, Int    x : Int , {} ⊢ 1, Int
----------------------------------------------
x : Int , {} ⊢ x + 1, Int                          41 : int
----------------------------------------------     ------------
{} ⊢ ( \ x : Int . x + 1 ), ( Int ) -> Int         {} ⊢ 41, Int    Targ = Int = domain(Tfu
------------------------------------------------------------------------------------------
{} ⊢ ( \ x : Int . x + 1 ) 41, Int
------------------------------------------------------------------------------------------
( \ x : Int . x + 1 ) 41 :: Int
```

3.6 Evaluator and typechecker: recap

Our declared relations are:

```
 Relations
===========

(→)     : expr (in), expr (out)          Pronounced as "small step"
(→*)    : expr (in), expr (out)          Pronounced as "big step"
(✓)     : expr (in)                      Pronounced as "is canonical"

(⊢)     : typingEnvironment (in), expr (in), type (out) Pronounced as "entails typing"

(::)    : expr (in), type (out) Pronounced as "type in empty context"
```

The definitiion of those relations are:

```
 Rules
=======





 expr0 → expr1
----------------------------            [EvalCtx]
 expr[expr0] → expr[expr1]



 n1:Number              n2:Number
------------------------------------    [EvalPlus]
 n1 "+" n2 → !plus(n1, n2)





----------------------------------------         [EvalIfTrue]
 "If" "True" "Then" e1 "Else" e2 → e1

```

```

----------------------------------------        [EvalIfFalse]
 "If" "False" "Then" e1 "Else" e2 → e2


------------------------------------------------------  [EvalLamApp]
 ("(" "\\" var ":" type "." e ")") arg → !subs:expr(var, arg, e)




 i:int
-----------                     [CanonInt]
 (✓) i


 b:bool
-----------                     [CanonBool]
 (✓) b




 (✓) e
---------                       [BigStepBase]
 e →* e

 e0 → e1        e1 →* e2
-------------------------       [BigStepRec]
 e0 →* e2





 "{}" ⊢ e, T
-----------     [TEmptyCtx]
 e :: T




----------------        [TboolTrue]
 Γ ⊢ "True", "Bool"


----------------        [TboolFalse]
 Γ ⊢ "False", "Bool"


 n:int
---------------         [Tnumber]
 Γ ⊢ n, "Int"








--------------------    [Tx]
 Γ[ x ":" T ] ⊢ x, T


 Γ ⊢ n1, "Int"  Γ ⊢ n2, "Int"
------------------------------                          [TPlus]
 Γ ⊢ n1 "+" n2, "Int"


 Γ ⊢ c, "Bool"  Γ ⊢ e1, Tl      Γ ⊢ e2, Tr      Tl = Tr : type
```

```
------------------------------------------------------------- [TIf]
 Γ ⊢ "If" c "Then" e1 "Else" e2, T1



 (x ":" T1) "," Γ ⊢ e, T2
----------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" T1 "." e ")", ( "(" T1 ")") "->" T2



 Γ ⊢ e1, Tfunc  Γ ⊢ e2, Targ    Targ = domain(Tfunc) : type
-----------------------------------------------------------   [Tapp]
 Γ ⊢ e1 e2, codomain(Tfunc)
```

Running the relation :: on all our examples with flags `./ALGT STFL.language True False If True Then False Else True If If` gives:

```
# True applied to ::
# Proof weight: 2, proof depth: 2


---------------
{} ⊢ True, Bool
---------------
True :: Bool




# False applied to ::
# Proof weight: 2, proof depth: 2


----------------
{} ⊢ False, Bool
----------------
False :: Bool




# If True Then False Else True applied to ::
# Proof weight: 6, proof depth: 3


---------------  ----------------  ---------------
{} ⊢ True, Bool  {} ⊢ False, Bool  {} ⊢ True, Bool  T1 = Bool = Tr
-------------------------------------------------------------------
{} ⊢ If True Then False Else True, Bool
-------------------------------------------------------------------
If True Then False Else True :: Bool




# 42 applied to ::
# Proof weight: 3, proof depth: 3


42 : int
------------
{} ⊢ 42, Int
------------
42 :: Int




# 20 + 22 applied to ::
# Proof weight: 6, proof depth: 4
```

```
20 : int      22 : int
------------  ------------
{} ⊢ 20, Int  {} ⊢ 22, Int
--------------------------
{} ⊢ 20 + 22, Int
--------------------------
20 + 22 :: Int




# 1 + 2 + 3 applied to ::
# Proof weight: 9, proof depth: 5


             2 : int      3 : int
             -----------  -----------
1 : int      {} ⊢ 2, Int  {} ⊢ 3, Int
-----------  ------------------------
{} ⊢ 1, Int  {} ⊢ 2 + 3, Int
-------------------------------------
{} ⊢ 1 + 2 + 3, Int
-------------------------------------
1 + 2 + 3 :: Int




# (\x : Int . x + 1) 41 applied to ::
# Proof weight: 10, proof depth: 6


                       1 : int
---------------------  ---------------------
x : Int , {} ⊢ x, Int  x : Int , {} ⊢ 1, Int
--------------------------------------------
x : Int , {} ⊢ x + 1, Int                     41 : int
--------------------------------------------  ------------
{} ⊢ ( \ x : Int . x + 1 ), ( Int ) -> Int    {} ⊢ 41, Int  Targ = Int = domain(Tfunc)
--------------------------------------------------------------------------------------
{} ⊢ ( \ x : Int . x + 1 ) 41, Int
--------------------------------------------------------------------------------------
( \ x : Int . x + 1 ) 41 :: Int
```

3.7 Properties

The last fundamental part of our language are it's properties.

Arbitrary properties can be stated just like rules can.

ALGT will check that these properties hold, by trying a bunch of random examples, effectively quickchecking your language implementation.

As most languages are concerned about two important properties, *preservation* and *progress*, we will work these out too.

3.7.1 Stating Preservation

Preservation is the property that, when an expression `e0` of type `T` is evaluated, the new expression `e1` is still of type `T`. Without this property, our typechecker would be useless...

Now, we can state this property in the `Properties`-section of the file:

```
 Properties
============

 e0 :: T         e0 → e1
-------------------------- [Preservation]
```

```
          e1 :: T
```

This is the same as a rule in a relation: given the predicates (`e0 :: T` and `e0 →e1`), the consequent (`e1 :: T`) can always be proven.

As our implementation is correct, we can just run our program as following, and see it did try to proof us wrong:

```
Done quickchecking property Preservation with 8 examples
 # Language file parsed. No action specified, see -h or --manual to specify other options
```

We can ask ALGT to try more examples, with the `--quickcheck-runs NUMBER-OF-RUNS` flag:

```
Done quickchecking property Preservation with 25 examples
 # Language file parsed. No action specified, see -h or --manual to specify other options
```

Note that we can also run these properties on our own examples, with `--test-property NAME OF PROPERTY --ppp` or `--test-all-properties --ppp`. By default, property proofs are not printed (as they tend to be long), `--ppp` says to Print the Property Proofs.

```
Property Progress holds for given examples
Property successfull
# Property Progress statisfied with assignment {T --> Bool, e0 --> True}
  # Predicate satisfied:
  # e0 :: T

    -------------- [TboolTrue]
    {} ⊢ True, Bool
    -------------------------- [TEmptyCtx]
    True :: Bool



  # Satisfies a possible conclusion:
  # (✓) e0

  True : bool
  ---------- [CanonBool]
  (✓) True
```

3.7.2 Stating progress

Another important property is *progress*. This states that, if an expression is well-typed, it either is in canonical form (a simple value) or we can evaluate for another step.

This is quite an important check to, as it means we can't have a *stuck* state, in which the evaluation doesn't know how to progress. Turing complete programming language (which **STFL** is *not*) might get stuck in a infinite loop though, but it is because a certain expression would yield exactly the same expression, but with at least some steps in between.

This property can be stated as following:

```
          e0 :: T
------------------------- [Progress]
 (✓) e0    |      e0 → e1
```

Here, *choice* is used in the consequent: given the predicates (`e0 :: T`), at least one of the choices must be proven:

```
Property Progress holds for given examples
Property successfull
# Property Progress statisfied with assignment {T --> Bool, e0 --> True}
  # Predicate satisfied:
  # e0 :: T

    -------------- [TboolTrue]
    {} ⊢ True, Bool
    -------------------------- [TEmptyCtx]
    True :: Bool
```

```
  # Satisfies a possible conclusion:
  # (✓) e0

  True : bool
  ---------- [CanonBool]
  (✓) True
```

```
Property Progress holds for given examples
Property successfull
# Property Progress statisfied with assignment {T --> Bool, e0 --> If True Then False Else
  # Predicate satisfied:
  # e0 :: T

    ---------------   ----------------   ---------------
    {} ⊢ True, Bool   {} ⊢ False, Bool   {} ⊢ True, Bool   T1 = Bool = Tr
    ---------------------------------------------------------------------
    {} ⊢ If True Then False Else True, Bool
    ---------------------------------------------------------------------
    If True Then False Else True :: Bool



  # Satisfies a possible conclusion:
  # e0 → e1

  -----------------------------------
  If True Then False Else True → False
```

Does ALGT find counterexamples for this property?

```
Done quickchecking property Progress with 8 examples
 # Language file parsed. No action specified, see -h or --manual to specify other options
```

Phew! No counterexamples found! Our language is probably *sound*.

3.7.3 Disabling quickchecks and symbolic checks

If the quickchecks take to long, you can disable them, using `--quickcheck-runs 0`. When the other checks takes to long (such as minimal typing of functions, liveability and totality), add `--no-check` to disable them all.

Chapter 4

Reference manual

4.1 General

A language is defined in a `.language`-file. It starts (optionally) with a title:

```

 Language Name
***************
```

Comments start with a `#` and can appear quasi everywhere.

```

# This is a comment
```

Syntax, functions, relations, ... are all defined in their own sections:

```

 Syntax
========
```

A section header starts with an upper case, is underlined with `=` and followed by a blank line.

4.2 Syntax

All syntax is defined in the `Syntax` section. It consists out of `BNF`-rules, of the form

```
name    ::= "literal" | choice | seq1 seq2
```

Choices might be written on multiple lines, as long as at least one tab precedes them:

```
name    ::= choice1 | choice2
        | choice3
```

4.2.1 Literals

A string that should be matched exactly, is enclosed in `"` (double quotes). Some characters can be escaped with a backslash, namely:

Table 4.1: Escape sequences

Sequence	Result
`\n`	newline
`\t`	tab
`\"`	double quote
`\\`	backslash

4.2.2 Parsing order

Rules are parsed **left to right**, in other words, choices are tried in order. No backtracking happens when a choice is made; the parser is a *recursive descent parser*. This has two drawbacks: left recursion results in an infinite loop and the ordering of choices does matter.

```
# Left recursion, error message.
expr    ::= expr "+" int

# ‘int‘ already consumes a part of ‘int + int‘, error message
expr    ::= int | int "+" int

# Common part extraction, error message

# Correct
expr    ::= int "+" int | int
```

4.2.3 Builtin syntactic forms

Some syntactic forms are already provided for your convenience, namely:

Table 4.2: Builtin syntax

<table>
<tr><th>Builtin</th><th>Meaning</th><th>Regex</th></tr>
<tr><td><code>Identifier</code></td><td>Matches an identifier</td><td><code>[a-z][a-zA-Z0-9]*</code></td></tr>
<tr><td><code>Number</code></td><td>Matches an (negative) integer. Integers parsed by this might be passed into the builtin arithmetic functions.</td><td><code>-?[0-9]*</code></td></tr>
<tr><td><code>Any</code></td><td>Matches a single character, whatever it is, including newline characters</td><td><code>.</code></td></tr>
<tr><td><code>Lower</code></td><td>Matches a lowercase letter</td><td><code>[a-z]</code></td></tr>
<tr><td><code>Upper</code></td><td>Matches an uppercase letter</td><td><code>[A-Z]</code></td></tr>
<tr><td><code>Digit</code></td><td>Matches an digit</td><td><code>[0-9]</code></td></tr>
<tr><td><code>Hex</code></td><td>Matches a hexadecimal digit</td><td><code>[0-9a-fA-F]</code></td></tr>
<tr><td><code>String</code></td><td>Matches a double quote delimted string, returns the value including the double quotes</td><td><code>"([^"\]|\"|\\)*"</code></td></tr>
<tr><td><code>StringUnesc</code></td><td>Matches a double quote delimeted string, returns the value without the double quotes</td><td><code>"([^"\]|\"|\\)*"</code></td></tr>
<tr><td><code>LineChar</code></td><td>Matches a single character that is not a newline. This includes ◦:</td><td><code>[^\n]</code></td></tr>
<tr><td><code>ParO</code></td><td>Matches a ‘(’, which will dissapear in the parsetree</td><td><code>(</code></td></tr>
<tr><td><code>ParC</code></td><td>Matches a ‘)’, which will dissapear in the parsetree</td><td><code>)</code></td></tr>
</table>

4.2.4 Subtyping relationship

A syntactic form equals a (possibly infinite) set of strings. By using a syntactic form `a` as choice in other syntactic form `b`, `a` will be a subset of be, giving the natural result that `a` is a subtype of `b`.

In the following examle, `bool` and `int` are both subsets of `expr`. This can be visualised with the `--lsvg Output.svg`-flag.

```
bool    ::= "True" | "False"
int     ::= Number
expr    ::= ... | bool | int
```

4.2.5 Whitespace in sequences

Whitespace (the characters `" "`,`"\t"`), is parsed by default (and ignored completely). If you want to parse a whitespace sensitive language, use other symbols to declare the rule:

Table 4.3: Whitespace modes

Operator	Meaning
`::=`	Totally ignore whitespace
`~~=`	Parse whitespace for this rule only
`//=`	Parse whitespace for this rule and all recursively called rules

This gives rise to the following behaviour:

Table 4.4: Whitespace mode examples

Syntax	Matching String
`a ::= "b" "c" x`	`b c x y`
`x ::= "x" "y"`	`bcxy`
	`b\tc\tx\ty`
	`b c\txy`
	`...`
`a ~~= "b" "c" x`	`bcx y`
`x ::= "x" "y"`	`bcxy`
	`bcx\ty`
`a //= "b" "c" x`	`bcxy`
`x ::= "x" "y"`	

4.2.6 Grouping sequences

Sometimes, you'll want to group an entire rule as a token (e.g. comments, an identifier, ...)

Add a `$` after the assignment to group it.

```
text                    ::= LineChar line | LineChar
commentLine             ::= $ "#" text "\n"

customIdentifier        ::= $ Upper Number
```

When such a token is used in a pattern or expression, the contents of this token are parsed against this rule:

```
f               : customIdentifier -> statement
f("X10")        = "X9" "# Some comment"
```

4.3 Functions

4.3.1 Patterns and expressions

Functions transform their input. A function is declared by first giving its type, followed by one or more clauses:

```
f               : a -> b
f(a, "b")       = "c"
f(a, b)         = "d"
```

When an input is given, arguments are pattern matched against the patterns on between parentheses. If the match succeeds, the expression on the right is given. If not, the next clause is given.

Note that using the same variable multiple times is allowed, this will only work if these arguments are the same:

```
f(a, a)         = ...
```

Recursion can be used just as well:

```
f("a" a)        = f(a)
```

This is purely functional, heavily inspired on Haskell.

Possible expressions

Expr	Name	As expression
`x`	Variable	Recalls the parsetree associated with this variable
`_`	Wildcard	*Not defined*
`42`	Number	This number
`"Token"`	Literal	This string
`func(arg0, arg1, ...)`	Function call	Evaluate this function
`!func:type(arg0, ...)`	Builtin function call	Evaluate this builtin function, let it return a `type`
`(expr or pattern:type)`	Ascription	Checks that an expression is of a type. Bit useless
`e[expr or pattern]`	Evaluation context	Replugs `expr` at the same place in `e`. Only works if `e` was created with an evaluation context
`a "b" (nested)`	Sequence	Builds the parse tree

Possible patterns

Expr	As pattern
`x`	Captures the argument as the name. If multiple are used in the same pattern, the captured arguments should be the same or the match fails.
`_`	Captures the argument and ignores it
`42`	Argument should be exactly this number
`"Token"`	Argument should be exactly this string
`func(arg0, arg1, ...)`	Evaluates the function, matches if the argument equals the result. Can only use variables which are declared left of this pattern
`!func:type(arg0, ...)`	Evaluates the builtin function, matches if the argument equals the result. Can only use variables which are declared left of this pattern
`(expr or pattern:type)`	Check that the argument is an element of `type`
`e[expr or pattern]`	Matches the parsetree with `e`, searches a subpart in it matching `pattern`
`a "b" (nested)`	Splits the parse tree in the appropriate parts, pattern matches the subparts

4.3.2 Typechecking

Equality

```

 Syntax
========

a        ::= ...
b        ::= ...
c        ::= a | b | d
```

```

 Functions
===========

f       : a -> b -> c
f(x, x) = x
```

When equality checks are used in the pattern matching, the variable will be typed as the smallest common supertype of both types. If such a supertype does not exist, an error message is given.

Note that using a supertype might be a little *too* loose, but won't normally happen in real-world examples.

In the given example, `x` will be typed as `c`, the common super type. In this example, `x` might also be a `d`, while this is not possible for the input. This can be solved by splitting of `a | b` as a new rule.

4.3.3 Totality- and liveabilitychecks

Can be disabled with `--no-check`, when they take to long.

4.3.4 Higher order functions and currying?

Are not possible for now (v 0.1.26). Perhaps in a future version or when someone really needs it and begs for it.

4.3.5 Builtin functions

name	Descr	Arguments
`plus`	Gives a sum of all arguments (0 if none given)	`Number* -> Number`
`min`	Gives the first argument, minus all the other arguments	`Number -> Number* -> Number`
`mul`	Multiplies all the arguments. (1 if none given)	`Number* -> Number`
`div`	Gives the first argument, divided by the product of the other arguments. (Integer division, rounded down))	`Number -> Number* -> Number`
`mod`	Gives the first argument, module the product of the other arguments.	`Number -> Number* -> Number`
`neg`	Gives the negation of the argument	`Number -> Number`
`equal`	Checks that all the arguments are equal. Gives 1 if so, 0 if not.	`.* -> .* -> Number`
`error`	Stops the function, gives a stack trace. When used in a rule, this won't match a predicate	`.** ->` ε
`subs`	(expression to replace, to replace with, in this expression) Replaces each occurence of the first expression by the second, in the third argument. You'll want to explictly type this one, by using `subs:returnType("x", "41", "x + 1")`	`.* -> .* -> .* -> .*`
`group`	Given a parsetree, flattens the contents of the parsetree to a single string	`.* -> StringUnesc`

4.4 Relations and Rules

4.5 Properties

4.6 Command line flags

Chapter 5

Used concepts and algorithms

Chapter 6

Dynamization and gradualization

Chapter 7

Thankword

Thanks to

- Christophe Scholliers
- Ilion Beyst, for always being up to date, giving ideas, spotting bugs at first sight, implementing `Nederlands` and his enthousiasm in general
- Isaura Claeys, for proofreading and finding a lot of typos