1 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 apply anonymous functions on them.

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`.

1.1 Setting up a .language

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

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

```
 STFL
******

# A Simply Typed Functional Language
```

1.2 Declaring the syntax

1.2.1 Simple booleans

TODO broken link For a full reference on syntax, see the reference manual on syntax

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. TODO reference

```
 STFL
******

# A Simply Typed Functional Language

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

What do we write here? Let's start with the boolean values `True` and `False`. We express that 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.

STFL.language now looks like:

```
 STFL
******

# A Simply Typed Functional Language

 Syntax
========

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

Lets try running this! Create `examples.stfl`, with in it:

```
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 tells that the example file should be split over each line.

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.

1.2.2 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`.[ˆspaces] 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.

TODO Broken link [ˆspaces]: Don't worry about spaces and tabs, we deal with them. If you want need to parse stuff like "duizendeneen", please refer reference manual

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 a few example programs. Let's add `If True Then False Else True` to *examples.stfl*:

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

And lets run our tool, this time with `./ALGT STFL.language examples.stfl` **expr** `-l`

:

```
  "examples.stfl" (line 1, column 1):
  unexpected "T"
  expecting "If"
  Could not parse expression of the form expr
```

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:

```
 STFL
******

# A Simply Typed Functional Language

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

Lets see what this gives:

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

Looks a lot better! The third examples shows clearly how the program 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` in our syntax instead:

```
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:
+  expr.0
|  "If": expr.0
|  +  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 indicates how the parse trees are nested. This can be rendered too:[3]

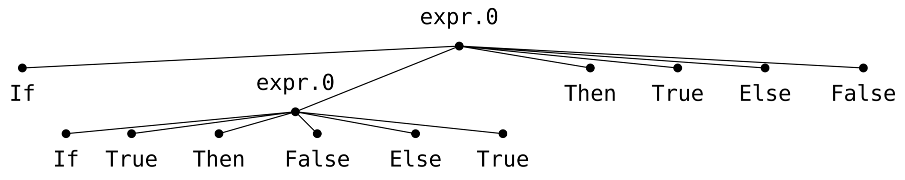


Figure 1: ParseTree of a nested condition[4]

1.2.3 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`[5] :

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

```
42
```

should give

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

So far, so good! Time to add addition:

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

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

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

[5]Creating this svg might take a long time for complicated syntaxes, as some optimization happens to figure out the lattice with the least intersections.

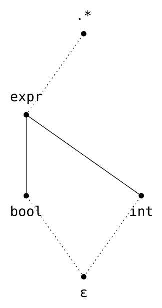


Figure 2: Subtypings of STFL

We add some example:

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

And run it:

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

Oops! Looks like we did something wrong. What is this left recursion? Whenever the parser want 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.

Also, 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:

```
                                                                                :
  "examples.stfl" (line 1, column 4):
  unexpected '+'
  expecting end of input
```

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 `term`, with the input string `+ 22` left. The parser doesn't know what to do with this leftover part, so we get an error.

To fix this, we only have to change the order:

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

When we try again, we get:

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

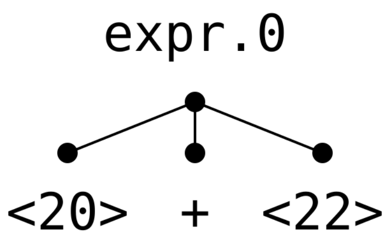


Figure 3: Parsetree of `20+22`

1.2.4 Lambda expressions

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

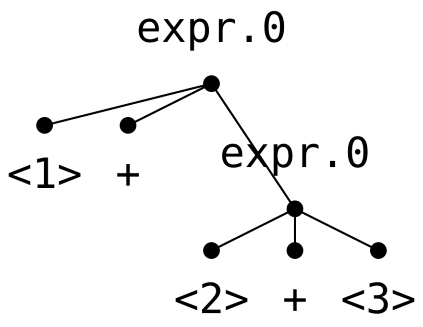


Figure 4: Parsetree of `1 + 2 + 3`

Variables The first thing we should deal with, are variables. A builtin in provided for those, namely `Identifiers`, matching all words starting with a lowercase. Let's introduce them in our syntax:

```
var     ::= Identifier
```

These are also valid in expressions, e.g. the expression `x + 1`, so we want to add it to our `term`:

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

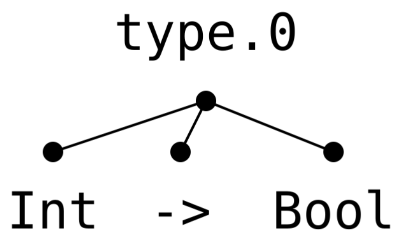
Types The second ingredient we still need, are types, to annotated the input types. Valid types, for starters, are `Bool` and `Int`. But what is the type of `(\x : Int . x + 1)`? It's something that takes an `Int` and gives back an `Int`. We type this as `Int -> Int`.

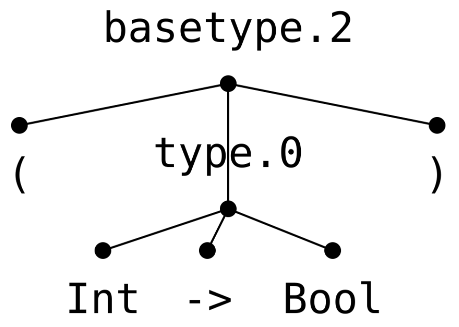
What is the type of a function, taking another function as an argument? That would be, for example, `(Int -> Int) -> Int`, meaning we need to add a form with parentheses.

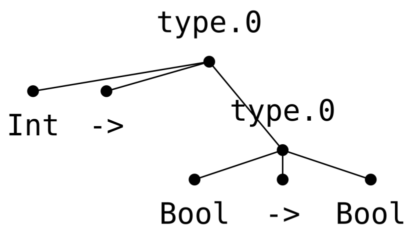
Recalling the trouble we had with left recursion and ordering, we write `type` as following:

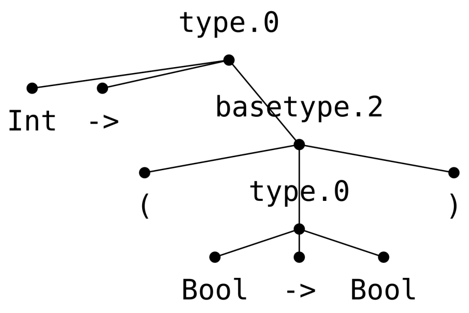
```
basetype::= "Bool" | "Int" | "(" type ")"
type    ::= basetype "->" type | basetype
```

Some examples of types are:









1.2.5 Lambda's

Now we have all we need to define lambda's. They act as a term, so 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.

As last, we can apply arguments to a lambda expression. We expand `expr`:

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

1.2.6 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.

1.2.7 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
        | term expr


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



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


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

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

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 show the parsetrees as SVG
- `ALGT STFL.language --lsvg SVGname.svg` to visualize the subtyping relationship.

1.3 Functions

1.3.1 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 output. The *codomain* of a function is the type it gives as output.

Table 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 -> Bool`
`Int`	*Undefined*	*Undefined*
`Bool`	*Undefined*	*Undefined*

Now, let's define these functions!

1.3.2 The function section

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

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

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

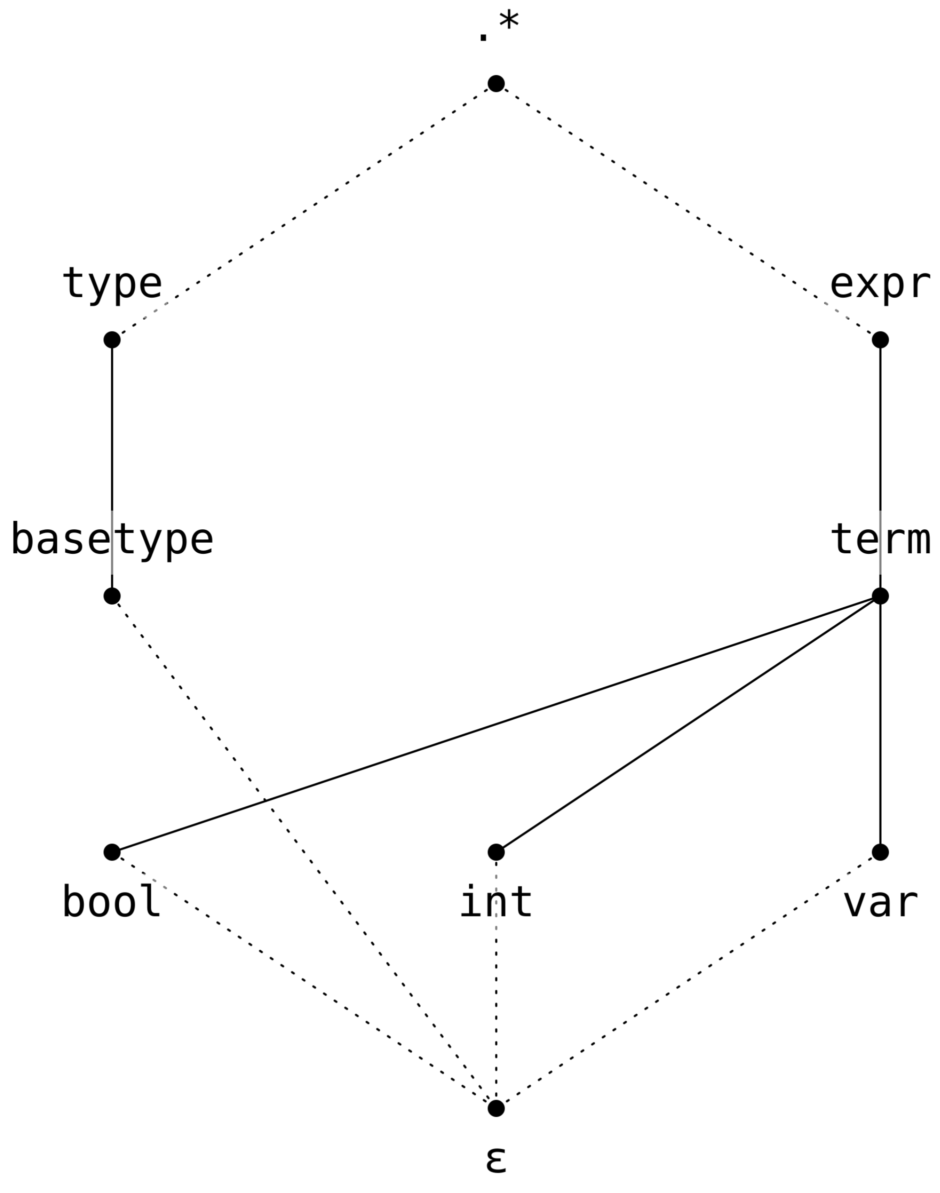


Figure 5: The final subtyping relationship of STFL.language

```
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 functions. The `type -> type` indicates what syntactic form is taken as input (a `type`) before the `->` and what is given as output (again a `type`). [6]

WHAT'S THIS?
DOUGLAS HOFSTADTER'S SIX-WORD AUTOBIOGRAPHY. AFTER ALL THOSE 700-PAGE TOMES, I GUESS HE WANTED TO TRY FOR BREVITY.
HUH. LET'S SEE...
I'M SO META, EVEN THIS ACRONYM
...WHOA.
I THINK HE NAILED IT.

Figure 6: Relevant XKCD (by Randall Munroe)

1.3.3 Pattern matching

Now, let's have look at the line of the function:

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

1.4 Relations and Rules

1.5 Properties

1.6 Recap: used command line arguments

[6] You probably noticed the similarity between the types declared in our own STFL and this declaration. Also see Randall Munroe's work on meta stuff

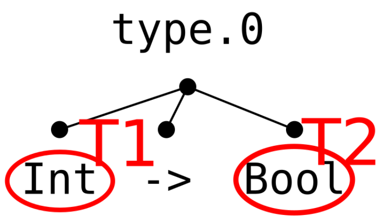


Figure 7: Pattern matching in action