I

Build your own language

Building a extremely simple language

Supporting

- Numbers
- Addition
- Functions
- Typechecking

Examples

```
42
41 + 1
1 + 2 + 39
(\ x : Int . x + 1) 41
(\f : Int -> Int . f 41) (\x : Int . x + 1)
```

Examples

For a first version, we'll omit all types:

```
42
41 + 1
1 + 2 + 39
(\ x . x + 1) 41
(\f . f 41) (\x . x + 1)
1 + 41
```

Tools

ALGT
github.com/pietervdvn/ALGT
All the files you need are in the **demo**-directory, download it entirely

Tools

ALGT

github.com/pietervdvn/ALGT

All the files you need are in the **demo**-directory, download it entirely

Alpha

Beware of bugs and unclear error messages

II

Declaring the syntax

What is the syntax of the language?

Syntax = *what it looks like*

Backus-Naur-Formulation

What is BNF?

```
name ::= "literal1" | "literal2" | "literal3"
```

Backus-Naur-Formulation

What is BNF?

```
name ::= "literal1" | "literal2" | "literal3"
```

Possible files:

```
literal1
```

OR

```
literal2
```

OR

```
literal3
```

Syntactic form names are written with a **lowercase letter**

Backus-Naur-Formulation

What is BNF?

```
name ::= "literal1" "literal2"
```

Backus-Naur-Formulation

What is BNF?

```
name ::= "literal1" "literal2"
```

Possible files:

```
literal1 literal2
```

Whitespace is ignored by default! See the manual for other options

Backus-Naur-Formulation

```
name                ::= "literal1"
otherName           ::= name
```

Backus-Naur-Formulation

```
name             ::= "literal1"
otherName        ::= name
```

Possible files:

```
literal1
```

Backus-Naur-Formulation

```
name             ::= "literal0"
otherName        ::= name name | "literal1" name | "literal2"
```

Backus-Naur-Formulation

```
name            ::= "literal0"
otherName       ::= name name | "literal1" name | "literal2"
```

Possible files:

```
literal0 literal0
```

OR

```
literal1 literal0
```

OR

```
literal2
```

Backus-Naur-Formulation

Defining numbers:

```
int             ::= "0" | "1" | "2" ...
```

Backus-Naur-Formulation

Defining variables:

```
var             ::= "a" | "b" | "c" | ... | "someVariableName" | ...
```

Backus-Naur-Formulation

Too much work. . .

Special rules **Number** and **Identifier** have been provided as builtin

Builtins are written with an *uppercase*

How do we define a language?

In the file *Demo.language*

```
 Demo
******

 Syntax
========

 int    ::= Number
```

Parsing!

./ALGT Demo.language demo.demo int -l

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

Addition

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

 int    ::= Number
 expr   ::= int "+" int
```

Addition

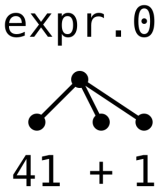
```
41 + 1
```

./ALGT Demo.language demo.demo **expr** -l

```
# "41 + 1" was parsed as:
+  41     Number.0
|  +      expr.0
|  1      Number.0
```

Addition

Use extra flag –ptsvg Name to create an image of your parsetree:



Addition

```
42
```

```
"demo.demo (line 0)" (line 1, column 3):
unexpected end of input
expecting "+"
```

Addition

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

 int    ::= Number
 expr   ::= int "+" int
        | int
```

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

Addition

What with $1 + 2 + 3$?

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

Addition

Let’s change expression to be recursive!

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

Left recursion

```
int      ::= Number
expr     ::= expr "+" expr
           | int
```

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

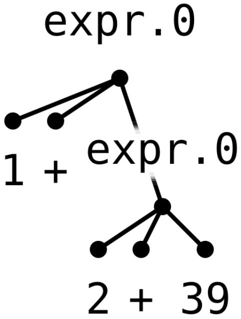
Left recursion

Simply use int for the first term:

```
int      ::= Number
expr     ::= int "+" expr
           | int
```

```
# "1 + 2 + 39" was parsed as:
+  1         Number.0
|  +         expr.0
|  +  2      Number.0
|  |  +      expr.0
|  |  39     Number.0
```

Left recursion



expr.0

1 + expr.0

2 + 39

Adding Functions

```
( \ x . x + 1)
```

Adding Functions

```
( \ x . x + 1)
```

```
expr    ::= "(" "\\" var "." expr ")"
        | ...
```

Adding Variables

```
var      ::= Identifier
```

Expression

```
expr    ::= "(" "\\" var "." expr ")" arg
          | int
          | var
          | int "+" expr
          | var "+" expr
          | "(" \\ var "." expr ")" arg "+" expr
          | ...
```

Adding Terms

We’ll want to introduce a syntactic form **term** , for variables, ints and functions:

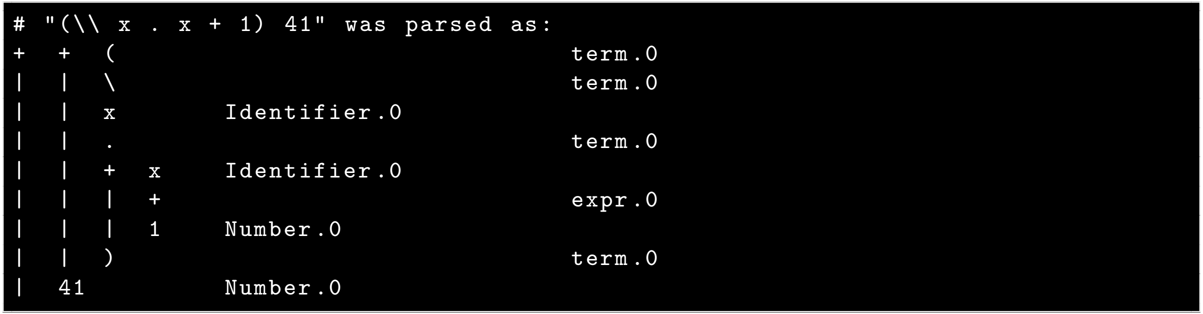
```
term    ::= "(" "\\" var "." expr ")"
          | "(" expr ")"
          | int
          | var
```

Adding Terms

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

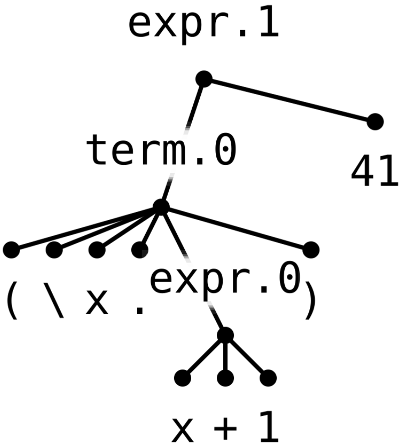
Parsing stuff

Still done with "./ALGT Demo.language demo.demo expr -l"



```
# "(\\ x . x + 1) 41" was parsed as:
+  +  (                              term.0
|  |  \                              term.0
|  |  x       Identifier.0
|  |  .                              term.0
|  |  +  x    Identifier.0
|  |  |  +                           expr.0
|  |  |  1    Number.0
|  |  )                              term.0
|  41         Number.0
```

Parsing stuff



III

Building the evaluator

Function or natural deduction?

ALGT supports two ways to perform computations:

Functions and natural deduction

For the evaluator, we'll use natural deduction

Declaring the relation

In a new section in the *.language*

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

(⟶) : expr (in), expr (out)     Pronounced as "Smallstep"
```

Declaring the relation

In a new section in the *.language*

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


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

This relation tells us *e0 becomes e1*

1 + 1 → 2

(\x . x + 1) 41 → 41 + 1

Defining →

Relations are defined in another section:

```
 Rules
=======
```

Defining EvalPlus

```
--------------------------- [EvalPlus]
 ... → ...
```

The conclusion goes beneath the line

The rulename goes on the right

Defining EvalPlus

```
-------------------------- [EvalPlus]
 i0 "+" i1 → ...
```

Defining EvalPlus

```
--------------------------- [EvalPlus]
 i0 "+" i1 → !plus(i0, i1)
```

Builtin functions do have an exclamation mark

Defining EvalPlus

```
 i0:int i1:int
--------------------------- [EvalPlus]
 i0 "+" i1 → !plus(i0, i1)
```

Running →

```
 i0:int i1:int
-------------------------- [EvalPlus]
 i0 "+" i1 → !plus(i0, i1)
```

./ALGT DemoDyn.language demodyn.demo expr -l -r →

```
# 41 + 1 applied to →
# Proof weight: 3, proof depth: 2



41 : int     1 : int
------------------- [EvalPlus]
41 + 1 → 42
```

Function application

```
------------------------------------------------------------ [EvalApp]
 function arg → ...
```

Function application

```
------------------------------------------------------------ [EvalApp]
 ("(" "\\" x "." expr ")") arg → ...
```

Extra parentheses around function, to group the subterm!

Function application

```
------------------------------------------------------------ [EvalApp]
 ("(" "\\" x "." expr ")") arg → !subs:expr(x, arg, expr)
```

Builtin function **!subs**: replace this, with that, everywhere in

For **!subs** is an explicit type needed

Function application

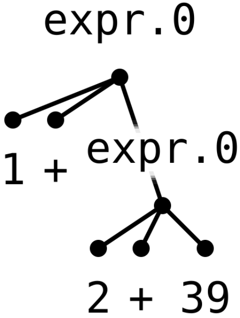
```
# (\ x . x + 1) 41 applied to →
# Proof weight: 1, proof depth: 1


--------------------------- [EvalApp]
( \ x . x + 1 ) 41 → 41 + 1
```

Contexts

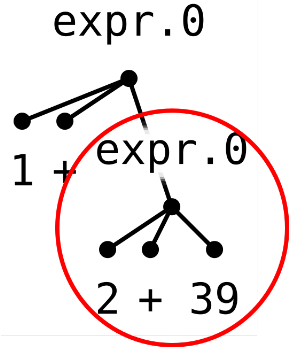
What with $1 + 2 + 39$?

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

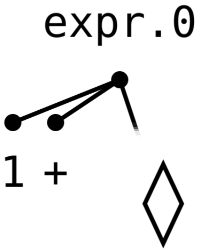
Contexts



Contexts



Contexts

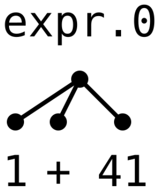


expr.0

1 +

◊

Contexts



expr.0

1 + 41

Contexts

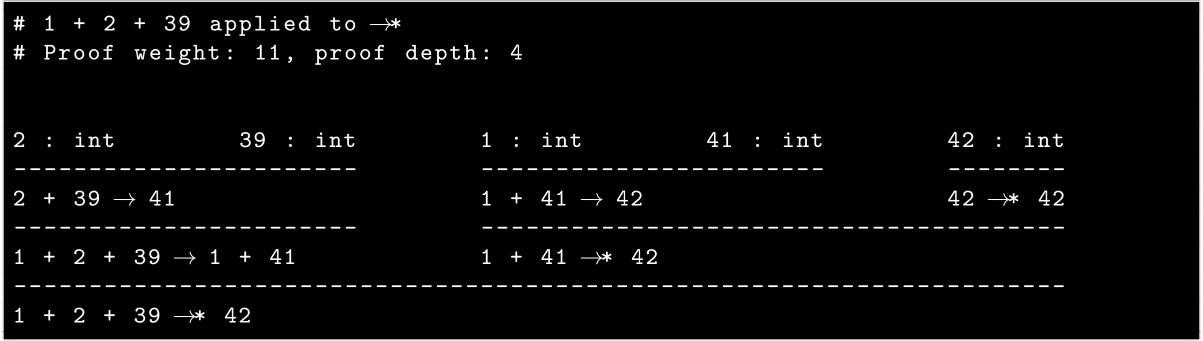
```
# 1 + 2 + 39 applied to →
# Proof weight: 4, proof depth: 3


2 : int    39 : int
------------------- [EvalPlus]
2 + 39 → 41
-------------------- [EvalCtx]
1 + 2 + 39 → 1 + 41
```

Bigstep

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

```
 i:int
--------- [BigStep Base]
 i →* i


 e0 → e1        e1 →* e2
-------------------------- [BigStep recursive]
 e0 →* e2
```

Bigstep



```
# 1 + 2 + 39 applied to →*
# Proof weight: 11, proof depth: 4


2 : int        39 : int        1 : int        41 : int        42 : int
-----------------------        -----------------------        --------
2 + 39 → 41                    1 + 41 → 42                    42 →* 42
-----------------------        ---------------------------------------
1 + 2 + 39 → 1 + 41            1 + 41 →* 42
----------------------------------------------------------------------
1 + 2 + 39 →* 42
```

IV

Your turn!

Your turn

Now it's your turn to give these a try.

- github.com/pietervdvn/ALGT
- Download the **demo**-directory
- Overview of commands and usefull stuff in **readme.md**

If there is still time, we'll also build a typechecker for the demo language...

V

The typechecker

VI

Introducing types

Introducing types

We’ll need syntactic forms for types

Introducing types

We'll need syntactic forms for types

```
typeTerm          ::= "Int" | "(" type ")"
type              ::= typeTerm "->" type | typeTerm
```

Introducing types

Explicit type tags on the input arguments:

```
term   ::= "(" "\\" var ":" type "." expr ")"
       | "(" expr ")"
       | int | var
expr   ::= term "+" expr | term expr | term
```

Introducing types

Which also means we'll have to update **EvalApp**

Introducing types

Which also means we'll have to update **EvalApp**

```
  -------------------------------------------------------------------- [EvalApp]
   ("(" "\\" x ":" TArg "." expr ")") arg → !subs:expr(x, arg, expr)
```

VII

Functions

Domain and codomain

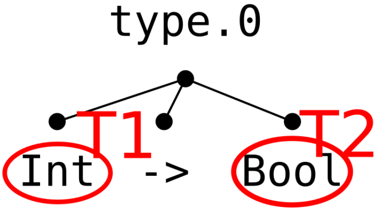
Function type	dom	cod
`Int -> Int`	`Int`	`Int`
`Int -> (Int -> Int)`	`Int`	`Int -> Int`
`(Int -> Int) -> Int`	`Int -> Int`	`Int`

Domain

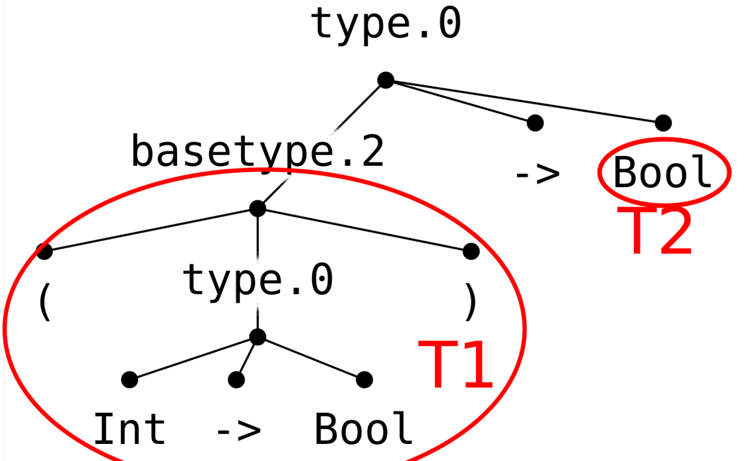
```
 Functions
===========


dom              : type -> type

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

Domain

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

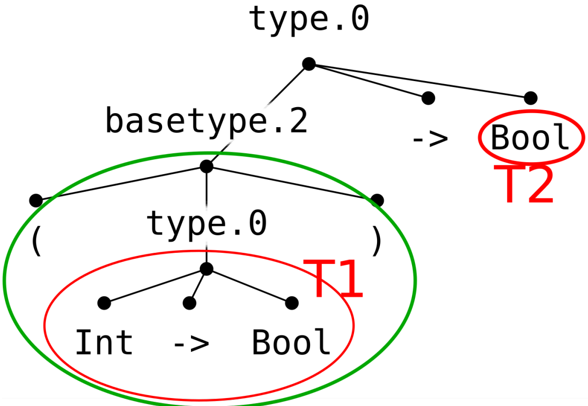


Domain



Domain

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



Domain

```
dom("(" T ")")   = dom(T)
```

Domain

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

Domain

Built-in totality check

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

Domain

Built-in totality check

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

```
dom("Int")          = !error("Undefined")
```

Codomain

```
cod             : type -> type
cod("(" T ")")  = cod(T)
cod(T1 "->" ("(" T2 ")"))
                = T2
cod(T1 "->" T2) = T2
cod("Int")      = !error("Undefined")
```

VIII

The typing environment

Variables

```
(\x : Int . x + 1) 41
```

How to keep track of what type a variable (such as **x**) has?

Typing environment

```
typing          ::= var ":" type
typings         ::= typing typings | "{}"
```

The typing environment will be denoted with Γ (U+393)

IX

Typing

The typing relation

```
(⊢) : typings (in), expr (in), type (out)         Pronounced as "Is typed as"
```

(⊢ is pronounced *entails*; U+22a2)

Typing constants

```
 i:int
-------------------- [TConstant]
 Γ ⊢ i, "Int"
```

Typing constants

```
 i:int
-------------------- [TConstant]
 Γ ⊢ i, "Int"
```

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


42 : int
------------ [TConstant]
{} ⊢ 42, Int
```

Typing plus

```
----------------------------------------- [TPlus]
Γ ⊢ i0 "+" i1 , "Int"
```

Typing plus

```
Γ ⊢ i0, "Int"           Γ ⊢ i1, "Int"
---------------------------------------- [TPlus]
Γ ⊢ i0 "+" i1  , "Int"
```

Typing plus

```
Γ ⊢ i0, "Int"             Γ ⊢ i1, "Int"
------------------------------------- [TPlus]
Γ ⊢ i0 "+" i1  , "Int"
```

```
# 41 + 1 applied to ::
# Proof weight: 6, proof depth: 4


41 : int                        1 : int
------------ [TConstant]        ----------- [TConstant]
{} ⊢ 41, Int                    {} ⊢ 1, Int
------------------------------------------- [TPlus]
{} ⊢ 41 + 1, Int
```

Typing variables

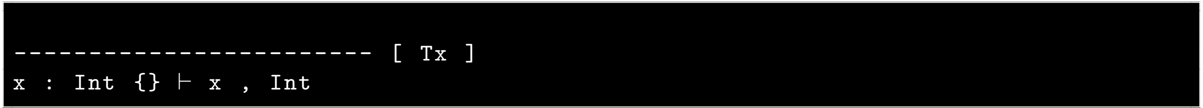
We lookup the variable in the **typingEnvironment**:

```

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

Typing variables

We lookup the variable in the **typingEnvironment**:

```

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



```

------------------------ [ Tx ]
x : Int {} ⊢ x , Int
```

Typing functions

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

What type do we return?

Typing functions

```
 Γ ⊢ expr, TExpr
----------------------------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" TArg "." expr ")" , ???
```

Hmm, something is missing here...

Typing functions

```
 (x ":" TArg) Γ ⊢ expr, TExpr
------------------------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" TArg "." expr ")" , ???
```

Typing environment syntax

```
 typing          ::= var ":" type
 typings         ::= typing typings | "{}"
```

Nearly done. . .

Typing functions

```
 (x ":" TArg) Γ ⊢ expr, TExpr
------------------------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" TArg "." expr ")" , TArg "->" TExpr
```

There is a catch. . .

Typing functions

```
 (x ":" TArg) Γ ⊢ expr, TExpr
----------------------------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" TArg "." expr ")" , TArg "->" TExpr
```

There is a catch. . .

TArg = **Int** -> **Int**

TExpr = **Int**

Typing functions

```
 (x ":" TArg) Γ ⊢ expr, TExpr
------------------------------------------------------------ [TLambda]
 Γ ⊢ "(" "\\" x ":" TArg "." expr ")" , TArg "->" TExpr
```

There is a catch...

TArg = **Int -> Int**

TExpr = **Int**

TArg "->" TExpr = **Int -> Int -> Int**

Typing functions

```
 (x ":" TArg) Γ ⊢ expr, TExpr
------------------------------------------------------------------------ [TLambda]
 Γ ⊢ "(" "\\" x ":" TArg "." expr ")" , ( "(" TArg ")") "->" TExpr
```

Typing functions

```
 (x ":" TArg) Γ ⊢ expr, TExpr
------------------------------------------------------------------- [TLambda]
 Γ ⊢ "(" "\\" x ":" TArg "." expr ")" , ( "(" TArg ")") "->" TExpr
```

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



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

Typing application

```
-----------------------------------------------------------   [Tapp]
 Γ ⊢ e1 e2, ???
```

Typing application

```
 Γ ⊢ e1, Tfunc
-------------------------------------------------------------    [Tapp]
 Γ ⊢ e1 e2, ???
```

Typing application

```
 Γ ⊢ e1, Tfunc  Γ ⊢ e2, Targ
-------------------------------------------------------------    [Tapp]
 Γ ⊢ e1 e2, ???
```

Typing application

```
 Γ ⊢ e1 , Tfunc  Γ ⊢ e2 , Targ
-------------------------------------------------------------   [Tapp]
 Γ ⊢ e1 e2 , cod(TFunc)
```

Typing application

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

Typing application

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

```
                    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 = dom(Tfunc)
-------------------------------------------------------------------------------
{} ⊢ ( \ x : Int . x + 1 ) 41, Int
```

Typing: practically

Define relation :: to type in an empty environment

```
(::) : expr (in), type (out)
```

```
"{}" ⊢ e, T
------------- [Typing in empty]
e :: T
```

./ALGT Demo.language demo.demo expr -l -r ::

One more thing

X

Syntax coloring

Syntax coloring

Extra section, just under the syntax

```
 Syntax Style
==============

 Number -> "constant"
 type -> "type"
 typeTerm -> "type"
 Identifier -> "identifier"
```

Syntax coloring

–style Terminal

{(\ x : **Int** . x + 1) 41}

–style White

{(\ x : **Int** . x + 1) 41}