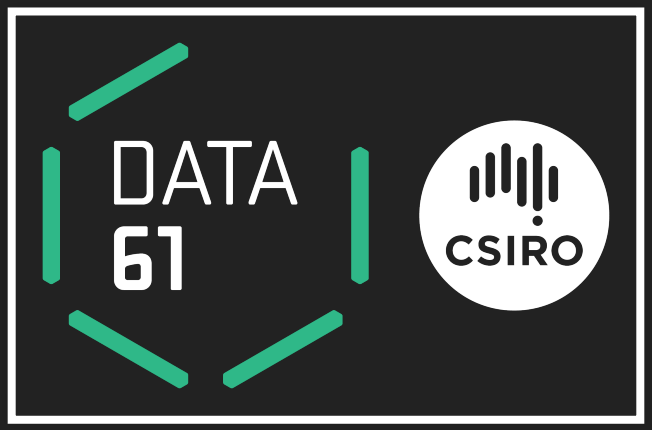
SNAKE WRANGLING

Isaac Elliott



How can we bring the benefits of better languages to existing codebases?

Language tooling

Language tooling for Python

```
append_to :: Statement
append_to =
  def_ "append_to" [ p_ "element", k_ "to" (list_ []) ]
    [ expr_ $ call_ ("to" /> "append") [ "element" ]
    , return_ "to"
    ]
```

```
def append_to(element, to=[]):
    to.append(element)
    return to
```

```
def append_to(element, to=None):
    if to is None:
        to = []
    to.append(element)
    return to
```

```
fixMutableDefaultArguments :: Statement -> Maybe Statement
```

```
rewriteOn _Statements fixMutableDefaultArguments :: Module -> Module
```

DESIGN

Parse & Print

```
(print . parse) :: String -> String

                =

               id
```

Write & Check

Optics

PROPERTY TESTING

Property of `plus`:

```
for all inputs A and B: A + B == B + A
```

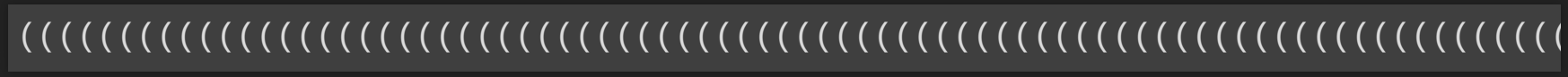
Parsing, printing, and validating Python source have useful properties

```
print . parse = id
```

Shrinking can find minimal counter-examples

You don't need to remember the whole language

Random generation is great for poking programming languages



CORRECTNESS BY CONSTRUCTION

If we can construct some data, then that data is correct (by some measure)

Incorrect = type error

Syntactically correct by construction

Python syntax isn't very straightforward

```
data Expr
  = Int Int
  | Bool Bool
  | Var String
  | ...

data Statement
  = Assign Expr Expr
  | ...
```

```
Assign (Int 1) (Int 2)
```

$$1 = 2$$

```
data Expr
  = Int Int
  | Bool Bool
  | Var String
  | ...

data AssignableExpr
  = AEVar String
  | ...

data Statement
  = Assign AssignableExpr Expr
  | ...
```

```
{-# language GADTs, DataKinds, KindSignatures #-}
```

```
data Assignable = IsAssignable | NotAssignable

data Expr :: Assignable -> * where
  Int :: Int -> Expr 'NotAssignable
  Bool :: Bool -> Expr 'NotAssignable
  Var :: String -> Expr a
  ...

data Statement
  = Assign (Expr 'IsAssignable) (Expr 'NotAssignable)
  | ...
```

```
expr :: Parser (Expr ??)
```

```
exprAssignable :: Parser (Expr 'Assignable)
exprNotAssignable :: Parser (Expr 'NotAssignable)
```

```
data ExprU
  = IntU Int
  | BoolU Bool
  | VarU String
  | ...

data StatementU
  = AssignU ExprU ExprU
  | ...
```

```
expr :: Parser ExprU
statement :: Parser StatementU
```

```
validateExprAssignable
  :: ExprU
  -> Either SyntaxError (Expr 'Assignable)

validateExprNotAssignable
  :: ExprU
  -> Either SyntaxError (Expr 'NotAssignable)

validateStatement
  :: StatementU
  -> Either SyntaxError Statement
```

Rinse and repeat

But it (mostly) worked... until...

```
not(condition)
```

```
data Expr :: type_stuff -> * where
  Not
    :: {- not -}
       NonEmpty Whitespace
    -> Expr type_stuff
    -> Expr type_stuff
  Parens
    :: Expr type_stuff
    -> Expr type_stuff
  ...
```

```
NonEmpty Whitespace -> [Whitespace]
```

```
data Expr :: type_stuff -> * where
  Not
    :: {- not -}
       [Whitespace]
    -> Expr type_stuff
    -> Expr type_stuff
  Parens
    :: Expr type_stuff
    -> Expr type_stuff
  ...
```

```
Not [] (Parens condition)
```

not(condition)

```
Not [] (Not [] (Parens condition))
```

notnot(condition)

Spaces are required between tokens when their concatenation would give a single token

```
mkNot
  :: {- not -}
     [Whitespace]
  -> Expr type_stuff
  -> Either SyntaxError (Expr type_stuff)
```

```
_Not :: Prism' Expr ([Whitespace], Expr)
```

```
_Not :: Prism Expr ExprU ([Whitespace], Expr) ([Whitespace], ExprU)
```

```
Expr -> ExprU
```

CONCRETE SYNTAX TREE

Have the data structures mirror the syntax

```
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test ['from' test]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from: ('from' (('.' | '...')* dotted_name | ('.' | '...')+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
assert_stmt: 'assert' test [',' test]

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef | classdef | decc
async_stmt: ASYNC (funcdef | with_stmt | for_stmt)
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
while_stmt: 'while' test ':' suite ['else' ':' suite]
for_stmt: 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
try_stmt: ('try' ':' suite
           ((except_clause ':' suite)+
            ['else' ':' suite]
            ['finally' ':' suite] |
           'finally' ':' suite))
with_stmt: 'with' with_item (',' with_item)*  ':' suite
with_item: test ['as' expr]
# NB compile.c makes sure that the default except clause is last
except_clause: 'except' [test ['as' NAME]]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT

test: or_test ['if' or_test 'else' test] | lambdef
test_nocond: or_test | lambdef_nocond
lambdef: 'lambda' [varargslist] ':' test
lambdef_nocond: 'lambda' [varargslist] ':' test_nocond
or_test: and_test ('or' and_test)*
and_test: not_test ('and' not_test)*
not_test: 'not' not_test | comparison
comparison: expr (comp_op expr)*
# <> isn't actually a valid comparison operator in Python. It's here for the
# sake of a __future__ import described in PEP 401 (which really works :-)
comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'
star_expr: '*' expr
expr: xor_expr ('|' xor_expr)*
xor_expr: and_expr ('^' and_expr)*
and_expr: shift_expr ('&' shift_expr)*
shift_expr: arith_expr (('<<'|'>>') arith_expr)*
arith_expr: term (('+'|'-') term)*
term: factor (('*'|'@'|'/'|'%'|'//') factor)*
factor: ('+'|'-'|'~') factor | power
power: atom_expr ['**' factor]
atom_expr: [AWAIT] atom trailer*
atom: ('(' [yield_expr|testlist_comp] ')' |
       '[' [testlist_comp] ']' |
       '{' [dictorsetmaker] '}' |
       NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')
testlist_comp: (test|star_expr) ( comp_for | (',' (test|star_expr))* [','] )
trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME
subscriptlist: subscript (',' subscript)* [',']
subscript: test | [test] ':' [test] [sliceop]
sliceop: ':' [test]
exprlist: (expr|star_expr) (',' (expr|star_expr))* [',']
testlist: test (',' test)* [',']
dictorsetmaker: ( ((test ':' test | '**' expr)
                   (comp_for | (',' (test ':' test | '**' expr))* [','])) |
                  ((test | star_expr)
                   (comp_for | (',' (test | star_expr))* [',']) )

classdef: 'class' NAME ['(' [arglist] ')'] ':' suite

arglist: argument (',' argument)*  [',']
```

Syntax is not code

THE DRAWING BOARD

~~Correct by Construction~~

~~Concrete Syntax Tree~~

~~Validated/Unvalidated Trees~~

```
data Expr (ts :: [*])
  = Int Int
  | Bool Bool
  | Var String
  | Not (Expr ts)
  | ...

data Statement (ts :: [*])
  = Assign (Expr ts) (Expr ts)
  | ...
```

```
-- raw, unvalidated
Expr '[]

-- syntax validated
Expr '[Syntax]

-- indentation & syntax validated
Statement '[Indentation, Syntax]
```

```
data Indentation

validateStatementIndentation
  :: Statement ts
  -> Either SyntaxError (Statement (Nub (Indentation ': ts)))
```

```
_Not
  :: Prism
       (Expr ts)
       (Expr '[])
       ([Whitespace], Expr ts)
       ([Whitespace], Expr '[])
```

```
import Data.Coerce

unvalidateStatement :: Statement ts -> Statement '[]
unvalidateStatement = coerce
```

Making 'incorrect' things impossible

vs.

Making 'correct' things trivial

Modelling how things appear

vs.

Modelling what things mean

SUMMARY

Property testing is great

We can get pretty far with type-level programming...

...But it's better to err on the side of usability

Instead of making the incorrect impossible, make the correct trivial

Figure out abstractions, rather than sticking to appearances

Mistakes are probably necessary

COOL STUFF

```
fact_tr :: Statement '[]
fact_tr =
  def_ "fact" [p_ "n"]
  [ def_ "go" [p_ "n", p_ "acc"]
    [ ifElse_ ("n" .== 0)
      [return_ "acc"]
      [return_ $ call_ "go" [p_ $ "n" .- 1, p_ $ "n" .* "acc"]]
    ]
  , return_ $ call_ "go" [p_ "n", p_ 1]
  ]
```

```
def fact(n):
    def go(n, acc):
        if n == 0:
            return acc
        else:
            return go(n - 1, n * acc)
    return go(n, 1)
```

```
optimizeTailRecursion :: Statement '[] -> Maybe (Statement '[])
```

```
rewrite optimizeTailRecursion fact_r :: Statement '[] -> Statement '[]
```

```
def fact(n):
    def go(n, acc):
        n__tr = n
        acc__tr = acc
        __res__tr = None
        while True:
            if n__tr == 0:
                __res__tr = acc__tr
                break
            else:
                n__tr__old = n__tr
                acc__tr__old = acc__tr
                n__tr = n__tr__old - 1
                acc__tr = n__tr__old * acc__tr__old
        return __res__tr
    return go(n, 1)
```