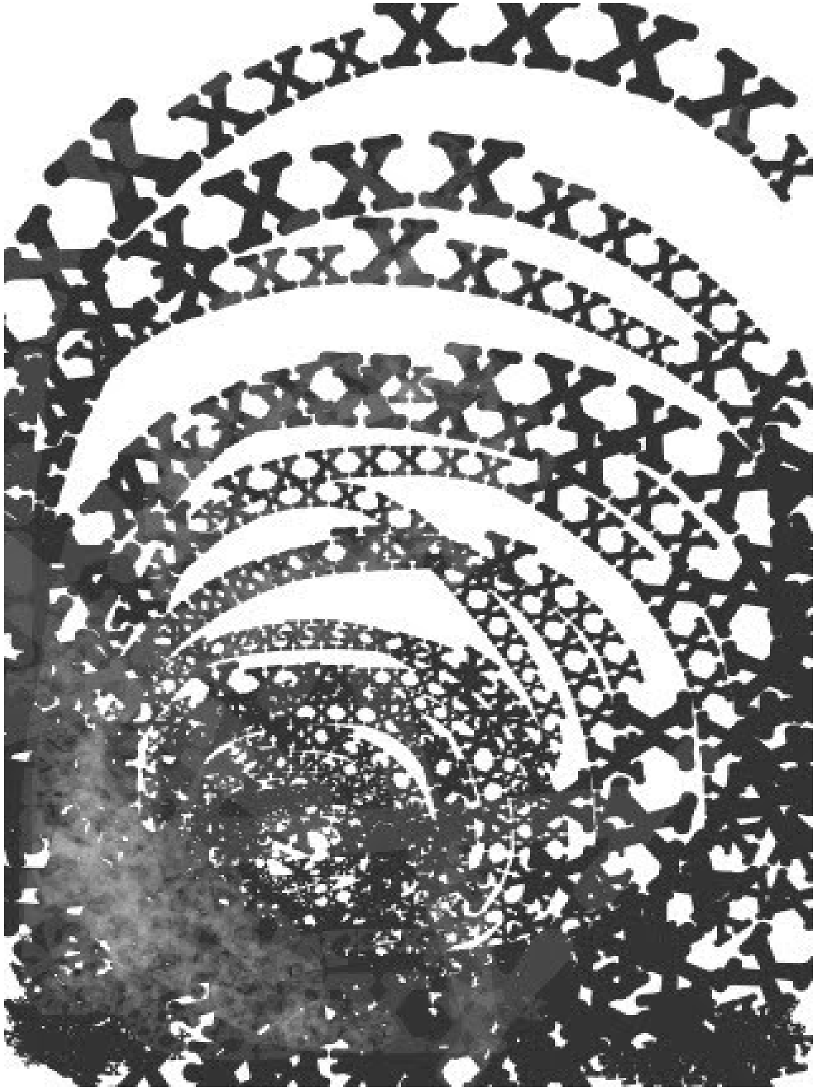


Sourcetools

Kay Schlühr
(‘kaɪ ʃlyɐ)

kay@fiber-space.de

Sourcetools - or ...

... how to script source code with little effort.

Sourcetools

Big source tools (full language view):

- `Compilers ( lexer, parser )`
- `Class browsers, refactoring IDEs`

Small source tools (little language view)?

Sourcetools

Big source tools (full language view):

- Compilers (lexer, parser)
- Class browsers, refactoring IDEs

Small source tools (little language view):

- Search & Replace in source code
- String templates with safety guards
- Expression generation

Nesting - a first example

Nested Function Calls

How to detect nested function calls in C code?

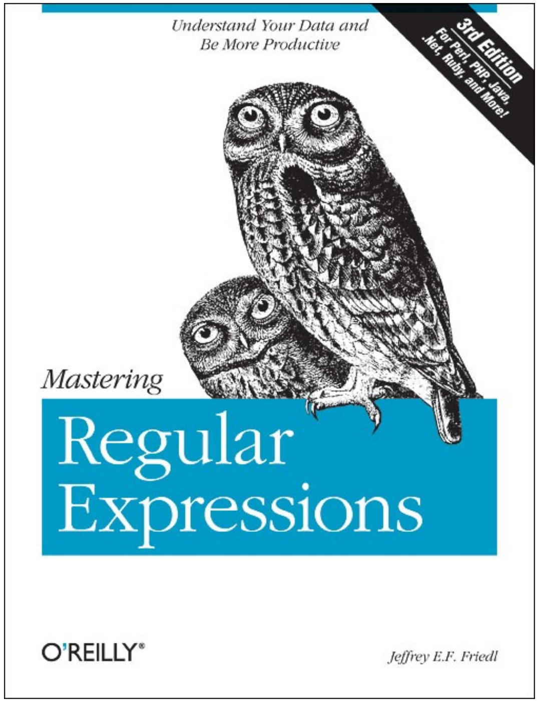
```
f((x+1)+(int*)(a)>>3)+g(3)

f(a,b,((float)c[i]*(g(3)-1)),d)

...
```

1st try : regexps

That's easy! - use regular expressions :)



```
\w+\s*\(  ...
```

“You work on Unix? Just buy it!” (reader opinion)

Problem: nested parentheses

Superduper regexp: `\w+\s*\(   ...`

How to find the terminating parenthesis?

This expression shall match:

```
F(.. G(...) ..)
```

but not this:

```
F(...) + G(...)
```

2nd try : C grammar

Building C grammar ...

2nd try : C grammar

Building C grammar is far too complicated !

3rd try : crafting a parser

Building C grammar is far too complicated !

Instead:

create a parentheses counting state machine, which

- **avoids looking into strings and comments**
- **avoids looking at type declarations etc.**

Then use this engine and write a scanner for the code.

3rd try : crafting a parser

1 hour later ...

Is building a C grammar really too complicated?

3rd try : still crafting a parser

2 hours later ...

Do I need a full C grammar?

4th try : minimalist grammars

```
grammar: funcall
funcall: NAME '(' [expr] (',' expr)* ')'
expr: binary_expr [ '?' binary_expr ':' binary_expr]
binary_expr: ([operator] (subscript_expr|funcall)
              ((operator|'.') (subscript_expr|funcall))*
              [operator])
subscript_expr: cast_expr ('[' expr (',' expr)* ']')*
cast_expr: '(' expr ')' [ expr ] | literal
literal: NAME | STRING | NUMBER
operator: OPERATOR
```

4th try: minimalist grammars

Small grammar - sufficient for pattern matching!

Still needs a Lexer for `NAME`, `STRING`, `IGNORE` etc. but this looks doable and can be re-used.

Matching grammars

Need a `re.search()` like function for our little language.

Need to search for a `funcall` within a `funcall`.

Parser Generators

pyparsing

pyPeg

SPARK

Yappy

Yapps

aperiot

DParser

ANTLR

PyGgy

mxTextTools

Parsing

PyBison

SimpleParse

Martel

Wisent

PLY

yeanpypa

LEPL

modparser

ZestyParser

ToyParserGenerator

Langscape

`langscape.sourcetools`

Langscape

- Powerful parser generator (trace based parsing)
- Languages as reusable components (langlets)
- Rich API
- Pure Python implementation (portable)

Langscape

- Powerful parser generator (trace based parsing)
- Languages as reusable components (langlets)
- Rich API
- Pure Python implementation (portable)

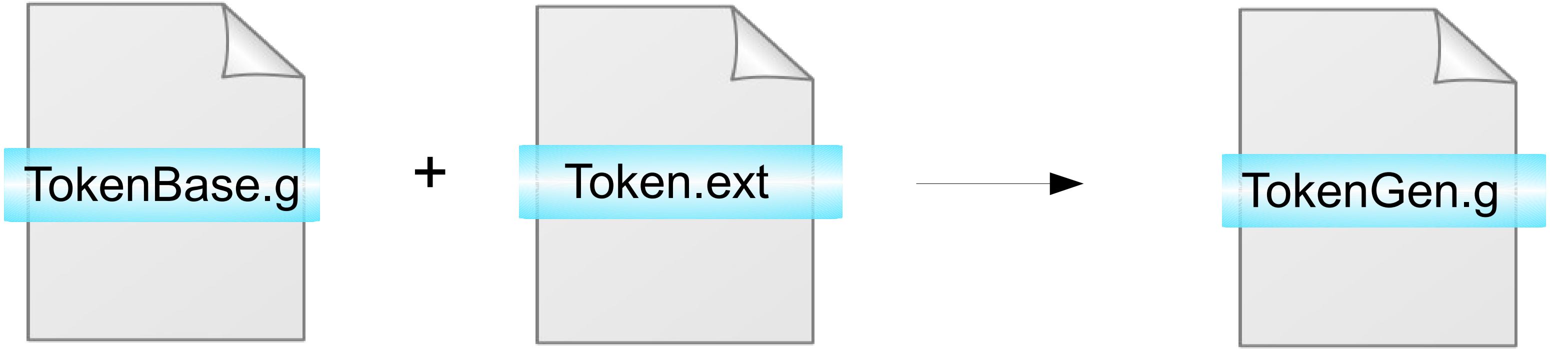
- Pure Python implementation (slow lexer/parser)
- Has β feeling

Langscape – create_langlet

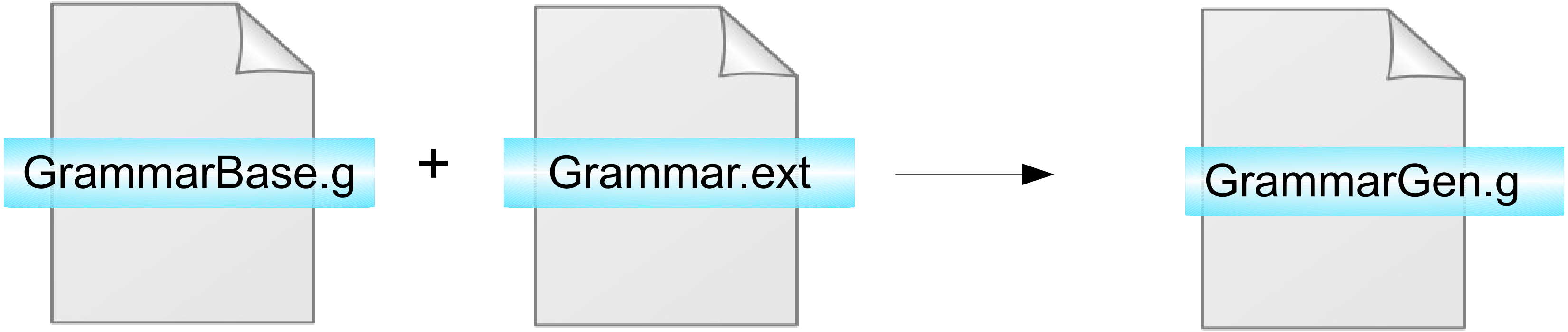
```
>>> import langscape
>>> langscape.create_langlet(“cfuncall”, prompt=”c>“)
```

Langscape – grammar defintions

langscape/langlets/cfuncall/lexdef


TokenBase.g + Token.ext → TokenGen.g

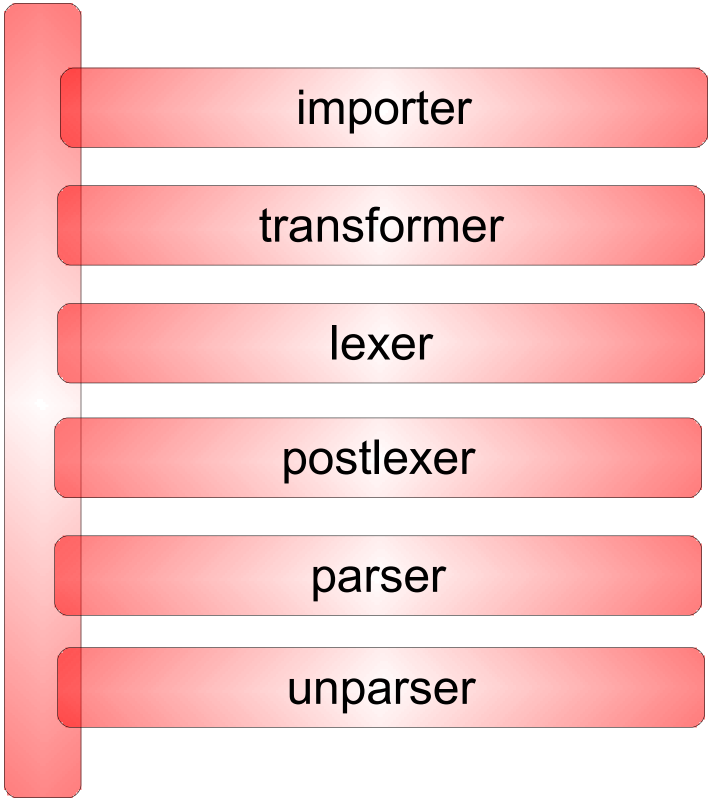
langscape/langlets/cfuncall/parsedef


GrammarBase.g + Grammar.ext → GrammarGen.g

Langscape – load_langlet

```
>>> cfuncall = langscape.load_langlet(“cfuncall”)
```

cfuncall-langlet



Langlet API

```
>>> cfuncall.tokenize(“foo(*x)”)
<TokenStream: [[130003, 'foo', 1, (0, 3)], [130025, '(', 1, (3, 4)],
[130040, '*', 1, (4, 5)], [130003, 'x', 1, (5, 6)], [130026, ')', 1, (6,
7)], [130002, '', 2, (0, 0)]] >
>>> cfuncall.parse(“foo(*x)”)
[131000, [131001, [130003, 'foo', 1, (0, 3)], [130025, '(', 1, (3, 4)],
[131002, [131003, [131007, [130040, '*', 1, (4, 5)]], [131004, [131005,
[131006, [130003, 'x', 1, (5, 6)]]]]]], [130026, ')', 1, (6, 7)]], [130002,
'', 2, (0, 0)]]
>>> cfuncall.untokenize(cfuncall.tokenize(“foo(*x)”))
'foo(*x)'
>>> cfuncall.unparse(cfuncall.parse(“foo(*x)”))
'foo(*x)'
```

Langlet – interactive console

```
>>> cfuncall.console().interact()
___________________________________________________________

 cfuncall

 On Python 2.7.1 (r271:86832, Nov 27 2010, 18:30:46)
___________________________________________________________

c> foo(*x)
←
c> quit
___________________________________________________________

>>>
```

Langlet – parse failure

```
c> 1+foo(*x)

Traceback (most recent call last):
   ...
ParserError: Failed to parse input '1' at (line 1 , column 1).

    line[1]: '1'

              ^
Failed to apply grammar rule:

    grammar:  NUMBER

              ^^^^^^
One of the following symbols must be used:

    Symbols
             NAME

c>
```

Search

Search – Prepare Source

```
url = ”http://codespeak.net/svn/xpython/trunk/dist/src/Objects/cobject.c”

import urllib2

f = urllib2.urlopen(url)

source = f.read()
```

Search – match code

```
>>> from langscape.sourcetools.codesearch import*
>>> cs = CSearchObject(cfuncall, cfuncall.symbol.funcall)
>>> for i, m in enumerate(cs.finditer(source)):
...     print i, m.matched
0 class_lookup(PyClassObject *, PyObject *,
                  PyClassObject **)
1 instance_getattr1(PyInstanceObject *, PyObject *)
2 instance_getattr2(PyInstanceObject *, PyObject *)
3 PyClass_New(PyObject *bases, PyObject *dict, PyObject *name)
4 if (docstr == NULL)
               . . .
970 PyMethod_Fini(void)
971 while (free_list)
972 PyObject_GC_Del(im)
```

Search – crap filter I

```
>>> from langscape.csttools.cstsearch import find_node, find_all
>>> for i, m in enumerate(cs.finditer(source)):
...     cst = cfuncall.parse(m.matched)
...     if len(find_all(cst, cfuncall.symbol.funcdef))>1:
...         print i, m.matched
```

```
17 if (PyDict_GetItem(dict, docstr) == NULL)
18 if (PyDict_SetItem(dict, docstr, Py_None) < 0)
19 if (PyDict_GetItem(dict, modstr) == NULL)
24 if (PyDict_SetItem(dict, modstr, modname) < 0)
28 if (!PyTuple_Check(bases))
32 if (!PyClass_Check(base))
33 if (PyCallable_Check(
                    (PyObject *) base->ob_type))
   . . .
```

Search – crap filter II

```
>>> from langscape.csttools.cstsearch import find_node, find_all
>>> for i, m in enumerate(cs.finditer(source)):
...     cst = cfuncall.parse(m.matched)
...     if len(find_all(cst, cfuncall.symbol.funcdef))>1:
...         name = find_node(cst, cfuncall.token.NAME)[1]
...         if name not in (“if”, “for”, “while”):
...             print i, m.matched
83 class_lookup(
            (PyClassObject *)
            PyTuple_GetItem(cp->cl_bases, i), name, pclass)
98 PyErr_Format(PyExc_AttributeError,
                 "class %.50s has no attribute '%.400s'",
                 PyString_AS_STRING(op->cl_name), sname)
107 set_slot(&c->cl_getattr, class_lookup(c, getattrstr, &dummy))
108 set_slot(&c->cl_setattr, class_lookup(c, setattrstr, &dummy))
   . . .
```

Code search

Extending the scripting toolbox:

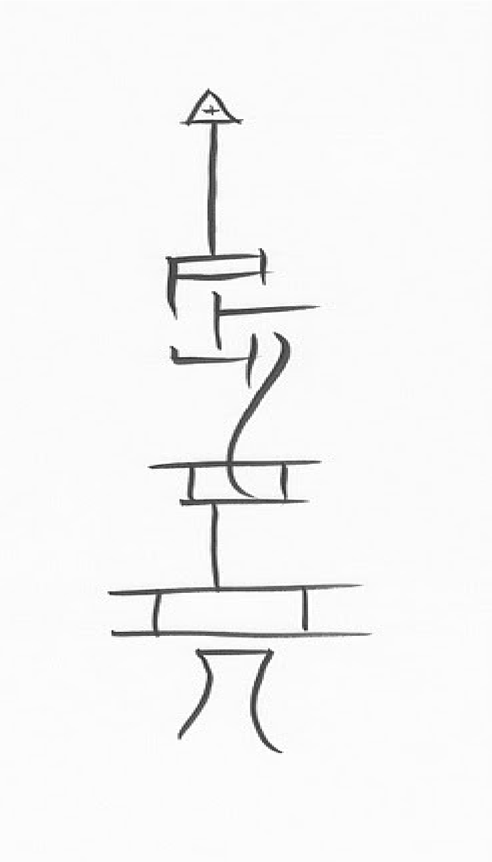
Small and shallow grammars
(declarative)

+

Postprocessing and filtering
(evil imperative or whatever)

Code Templates

Advanced textual substituation



Code Templates

A CodeTemplate is like a StringTemplate but

- there is no meta-syntax not even a metacharacter such as '%'
- transformations are checked for syntactical correctness
- support for gensyms to avoid name capturing

Code Templates

Initialization and parametrization

```
if_stmt = '''
if TEST:
    BLOCK
'''

from langscape.sourcetools.codetemplate import*
python = langscape.load_langlet(“python”)

ct = CodeTemplate(python, if_stmt)
```

Code Templates – bind & subst

```
>>> ct.bind(t = “TEST”, b = “BLOCK”)
>>> tree = ct.substitute(t=”f(x) == 0”, b = if_stmt)
>>> print python.unparse(tree)
if f(x) == 0:
    if TEST:
        BLOCK
```

```
>>> print if_stmt.replace(“BLOCK”, if_stmt)

if TEST:

if TEST:
    BLOCK
```

Code Templates – bind & subst

```
>>> ct.bind(t = “TEST”, b = “BLOCK”)
>>> tree = ct.substitute(t=”f(x) == 0”, b = if_stmt)
>>> print python.unparse(tree)
if f(x) == 0:
    if TEST:
        BLOCK

>>> print if_stmt.replace(“BLOCK”, if_stmt)

if TEST:

if TEST:
    BLOCK
```

Code Templates – internal representation

```
>>> ct.tokstream
<TokenStream: [[4, '\n', 1, (0, 1)], [516, 'if', 2, (0,
2)], [1, 'TEST', 2, (3, 7)], [11, ':', 2, (7, 8)], [4, '\
n', 2, (8, 9)], [5, '    ', 3, (0, 4)], [1, 'BLOCK', 3,
(4, 9)], [4, '\n', 3, (9, 10)], [6, '', 4, (0, 0)], [0,
'', 5, (0, 0)]] >
```

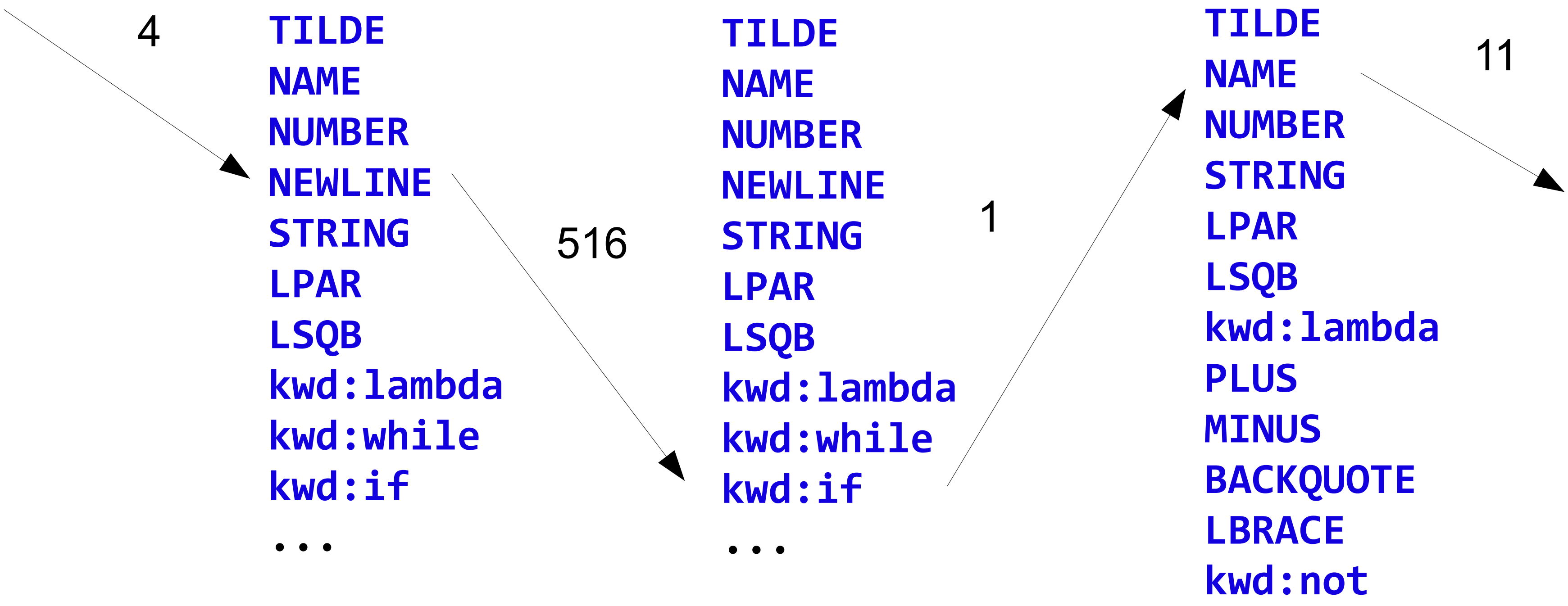
Code Templates – internal representation

```
>>> ct.tokstream
<TokenStream: [[4, '\n', 1, (0, 1)], [516, 'if', 2, (0,
2)], [1, 'TEST', 2, (3, 7)], [11, ':', 2, (7, 8)], [4, '\
n', 2, (8, 9)], [5, '    ', 3, (0, 4)], [1, 'BLOCK', 3,
(4, 9)], [4, '\n', 3, (9, 10)], [6, '', 4, (0, 0)], [0,
'', 5, (0, 0)]] >
```

Code Templates – TokenTracer

```
<TokenStream: 4, 516, 1, 11, 4, 5, 1, 4, 6, 0 >
```

```
from langscape.trail.tokentracer import TokenTracer
tracer = TokenTracer(python, python.symbol.file_input)
tracer.select(4)
tracer.select(516)
...
```



4

TILDE
NAME
NUMBER
NEWLINE
STRING
LPAR
LSQB
kwd:lambda
kwd:while
kwd:if
...

516

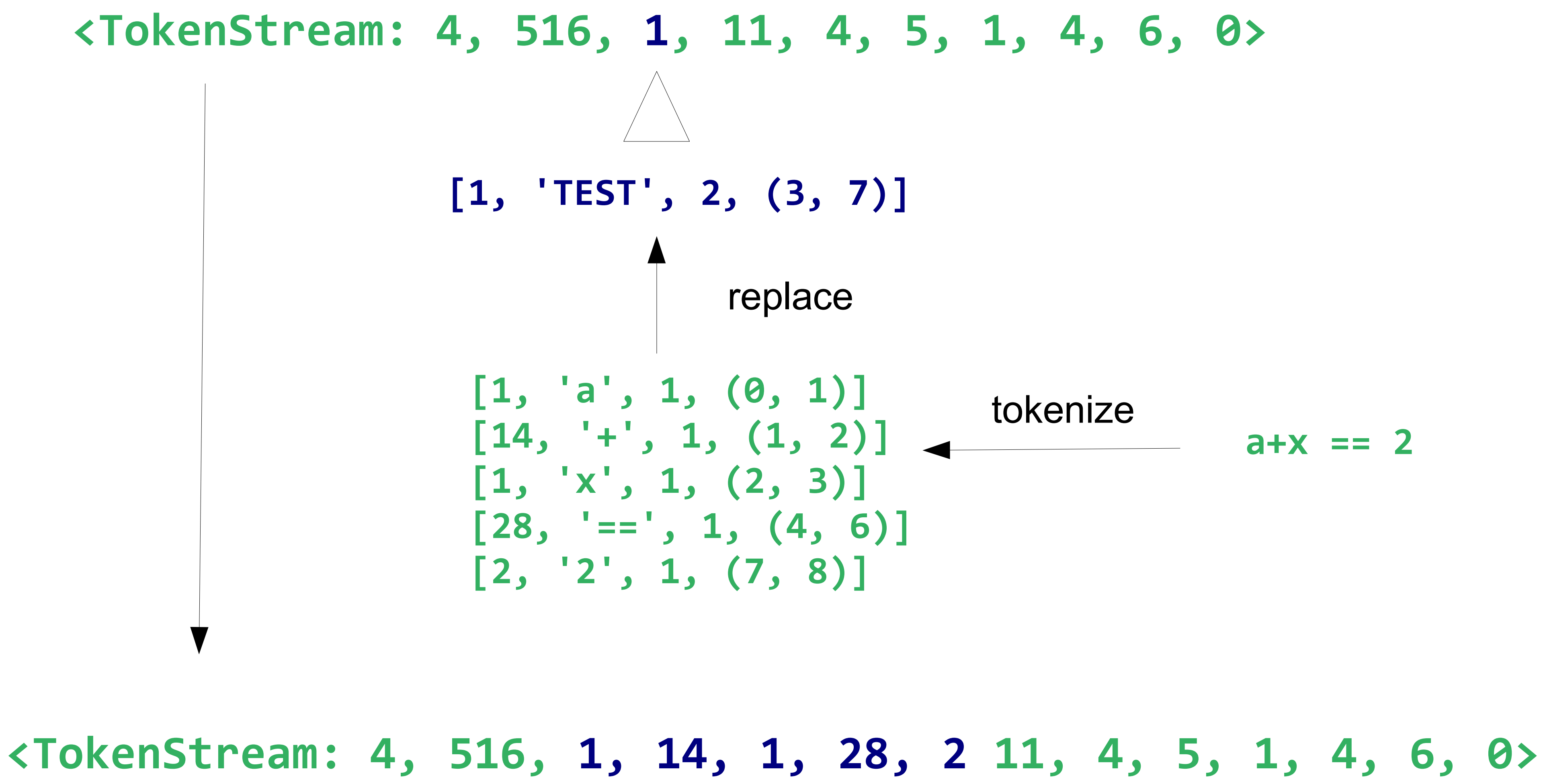
TILDE
NAME
NUMBER
NEWLINE
STRING
LPAR
LSQB
kwd:lambda
kwd:while
kwd:if
...

1

TILDE
NAME
NUMBER
STRING
LPAR
LSQB
kwd:lambda
PLUS
MINUS
BACKQUOTE
LBRACE
kwd:not

11

Code Templates – replacements



```
<TokenStream: 4, 516, 1, 11, 4, 5, 1, 4, 6, 0>
```

```
[1, 'TEST', 2, (3, 7)]
```

replace

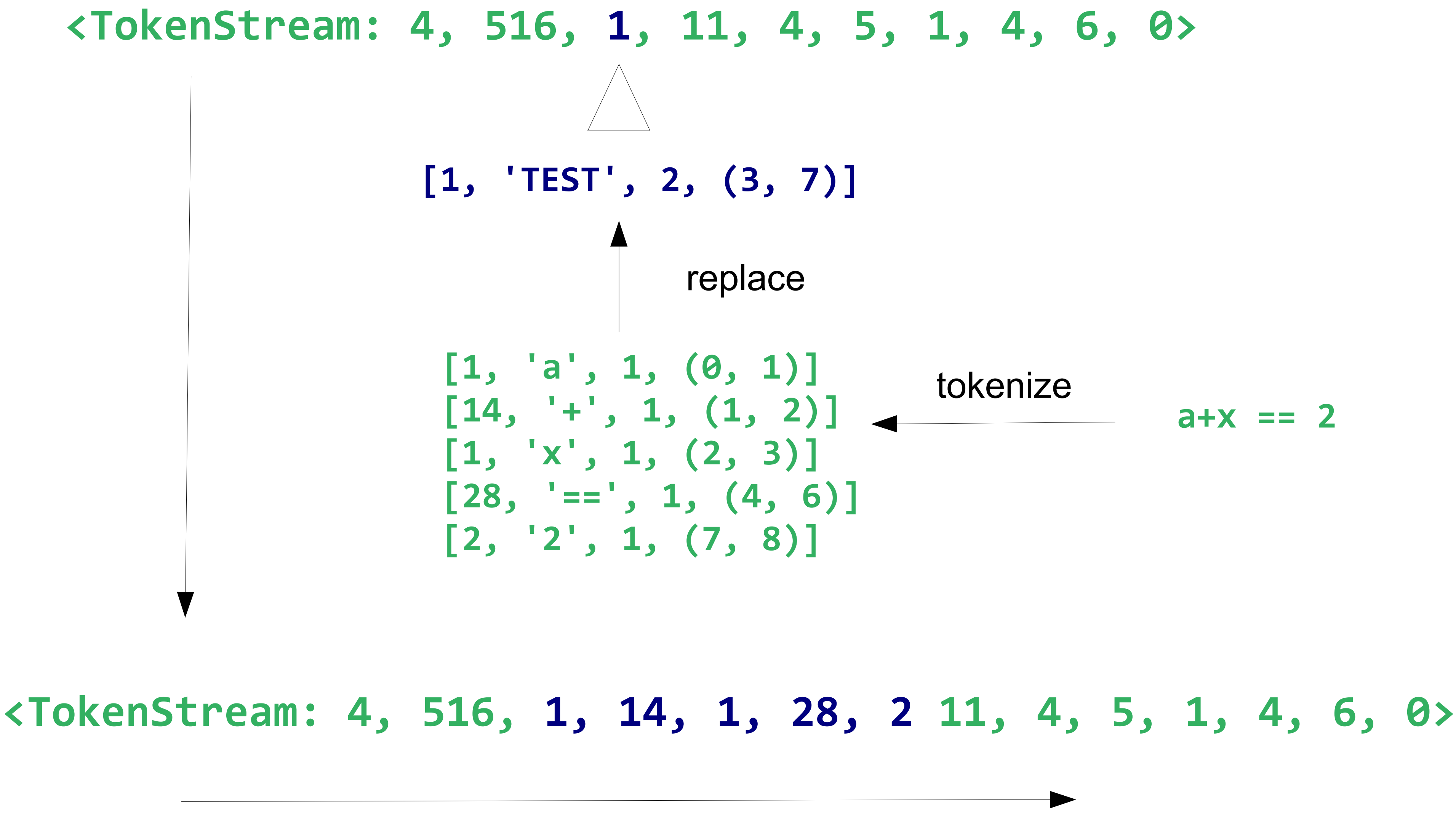
```
[1, 'a', 1, (0, 1)]
[14, '+', 1, (1, 2)]
[1, 'x', 1, (2, 3)]
[28, '==', 1, (4, 6)]
[2, '2', 1, (7, 8)]
```

tokenize

```
a+x == 2
```

```
<TokenStream: 4, 516, 1, 14, 1, 28, 2 11, 4, 5, 1, 4, 6, 0>
```

Code Templates – replacements



```
<TokenStream: 4, 516, 1, 11, 4, 5, 1, 4, 6, 0>
```

```
[1, 'TEST', 2, (3, 7)]
```

replace

```
[1, 'a', 1, (0, 1)]
[14, '+', 1, (1, 2)]
[1, 'x', 1, (2, 3)]
[28, '==', 1, (4, 6)]
[2, '2', 1, (7, 8)]
```

tokenize

```
a+x == 2
```

```
<TokenStream: 4, 516, 1, 14, 1, 28, 2 11, 4, 5, 1, 4, 6, 0>
```

```
Verify: tr.select(4),tr.select(516),tr.select(1),tr.select(14),...
```

Code Templates – name capture

```
swap = '''
temp = b
b = a
a = temp
'''
```

```
>>> ct = CodeTemplate(python, swap)
>>> ct.bind(x = ”a”, y = “b”)
>>> cst = ct.substitute(x=”temp”, y = “b”)
>>> print python.unparse(cst)

temp = b
b = temp
temp = temp
```

Code Templates – name capture

```
swap = '''
temp = b
b = a
a = temp
'''
```

```
>>> ct = CodeTemplate(python, swap)
>>> ct.bind(x = ”a”, y = “b”)
>>> cst = ct.substitute(x=”temp”, y = “b”)
>>> print python.unparse(cst)
```

```
temp = b
b = temp
temp = temp
```

Code Templates – name capture

```
swap = '''
temp = b
b = a
a = temp
'''

>>> ct = CodeTemplate(python, swap)
>>> ct.bind(x = ”a”, y = “b”)
>>> ct.local_names(”temp”)
>>> cst = ct.substitute(x=”temp”, y = “b”)
>>> print python.unparse(cst)

temp_00001 = b
b = temp
temp = temp_00001
```

Generative grammars

“The language speaks” (M.Heidegger)

Generative grammars

An obvious idea:

Grammars as expression generators

Generative grammars

An obvious idea:

Grammars as expression generators

YES!

but which ones?

Generative grammars - approximation

```
grammar: funcall
funcall: NAME '(' [expr] (',' expr)* ')'
expr: binary_expr [ '?' binary_expr ':' binary_expr]
binary_expr: ([operator] (subscript_expr|funcall)
              ((operator|'.') (subscript_expr|funcall))*
              [operator])
subscript_expr: cast_expr ('[' expr (',' expr)* ']')*
cast_expr: '(' expr ')' [ expr ] | literal
literal: NAME | STRING | NUMBER
operator: OPERATOR
```

Generative grammars - approximation

Eliminate loops:

```
grammar: funcall
funcall: NAME '(' [expr] [',' expr] ')'
expr: binary_expr [ '?' binary_expr ':' binary_expr]
binary_expr: ([operator] (subscript_expr|funcall)
              [(operator|'.') (subscript_expr|funcall)]
              [operator])
subscript_expr: cast_expr ['[' expr [',' expr] ']']
cast_expr: '(' expr ')' [ expr ] | literal
literal: NAME | STRING | NUMBER
operator: OPERATOR
```

Generative grammars - approximation

But allow rewriting of rules and add multiplicities:

```
grammar: funcall
funcall: NAME '(' [expr] (',' expr){0,3} ')'
expr: binary_expr [ '?' binary_expr ':' binary_expr]
binary_expr: ([operator] (subscript_expr|funcall)
              [(operator|'.') (subscript_expr|funcall)]
              [operator])
subscript_expr: cast_expr ['[' expr [',' expr] ']']
cast_expr: '(' expr ')' [ expr ] | literal
literal: NAME | STRING | NUMBER
operator: OPERATOR
```

Generative grammars - insertion

Expansion of rules can create new cycles:

```
grammar: funcall
funcall: NAME '(' [expr] [',' expr] ')'
expr: binary_expr [ '?' binary_expr ':' binary_expr]
binary_expr: ([operator] (subscript_expr|funcall)
              [(operator|'.') (subscript_expr|funcall)]
              [operator])
subscript_expr: cast_expr ['[' expr [',' expr] ']']
cast_expr: '(' expr ')' [ expr ] | literal
literal: NAME | STRING | NUMBER
operator: OPERATOR
```

Generative grammars - reduction

But there are also nice contractions:

```
grammar: funcall
funcall: NAME '(' [expr] [',' expr] ')'
expr: binary_expr [ '?' binary_expr ':' binary_expr]
binary_expr: ([operator] (subscript_expr|funcall)
              [(operator|'.') (subscript_expr|funcall)]
              [operator])
subscript_expr: cast_expr ['[' expr [',' expr] ']']
cast_expr: '(' expr ')' [ expr ] | literal
literal: NAME | STRING | NUMBER
operator: OPERATOR
```

Generative grammars - reduction

```
expr → binary_expr →
       subscript_expr →
       cast_expr →
       literal →
       NAME
```

Generative grammars - reduction

Idea: replace rule by terminal of the same name

```
expr → binary_expr →
       subscript_expr →
       cast_expr →
       literal →
       NAME →
       'expr'
```

Generative grammars - reduction

Expressions that look like rules:

```
grammar: funcall
funcall: NAME '(' ['expr'] [',' 'expr'] ')'
expr: 'binary_expr' [ '?' 'binary_expr' ':' 'binary_expr']
binary_expr: ([operator] ('subscript_expr'|funcall)
              [(operator|'.') ('subscript_expr'|funcall)]
              [operator])
subscript_expr: 'cast_expr' ['[' 'expr' [',' 'expr'] ']']
cast_expr: '(' 'expr' ')' [ 'expr' ] | literal
literal: NAME | STRING | NUMBER
operator: OPERATOR
```

LangletExpr - API

```
>>> from langscape.sourcetools.langletexpr import*
>>> le = LangletExpr(cfuncall)
>>> for expr in le.expressions(cfuncall.symbol.funcall):
...     if expr[0] == cfuncall.symbol.funcall:
...         print expr[1:]

('funcall', 'b(expr)')
('funcall', 'c(expr, expr)')
('funcall', 'd()')
('funcall', 'e(, expr)')
```

LangletExpr - API

```
>>> from langscape.sourcetools.langletexpr import*
>>> le = LangletExpr(cfuncall)
>>> for expr in le.expressions(cfuncall.symbol.funcall):
...     if expr[0] == cfuncall.symbol.funcall:
...         print expr[1:]

('funcall', 'b(expr)')
('funcall', 'c(expr, expr)')
('funcall', 'd()')
('funcall', 'e(, expr)')
```

LangletExpr - funcall

```
>>> from langscape.sourcetools.langletexpr import*
>>> le = LangletExpr(cfuncall)
>>> for expr in le.expressions(cfuncall.symbol.funcall):
...     if expr[0] == cfuncall.symbol.funcall:
...         print expr[1:]

('funcall', 'b(expr)')
('funcall', 'c(expr, expr)')
('funcall', 'd()')
('funcall', 'e(, expr)')
~~~~~~~~~~~~~~~~~~~~~~~~
```

Grammar - cleanup the mess

```
funcall: NAME '(' [expr] (',' expr)* ')'
```

↓

```
funcall: NAME '(' [expr (',' expr)*] ')'
```

LangletExpr - funcall

```
>>> from langscape.sourcetools.langletexpr import*
>>> le = LangletExpr(cfuncall)
>>> for expr in le.expressions(cfuncall.symbol.funcall):
...     if expr[0] == cfuncall.symbol.funcall:
...         print expr[1:]

('funcall', 'b(expr)')
('funcall', 'c(expr, expr)')
('funcall', 'd()')
```

LangletExpr - binary_expr

```
>>> from langscape.sourcetools.langletexpr import*
>>> le = LangletExpr(cfuncall)
>>> for expr in le.expressions(cfuncall.symbol.funcall):
...     if expr[0] == cfuncall.symbol.binary_expr:
...         print expr[1:]

('binary_expr', '== subscript_expr.subscript_expr~')
('binary_expr', 'funcall()~funcall()')
('binary_expr', 'funcall()~funcall()~')
('binary_expr', 'funcall()~funcall().subscript_expr')
...
```

Grammar - cleanup the mess II

```
binary_expr: ([operator] (subscript_expr|funcall)
              ((operator|'.') (subscript_expr|funcall))*
              [operator])
```

```
binary_expr: ([prefix_perator] (subscript_expr|funcall)
              ((infix_perator|'.') (subscript_expr|funcall))*
              [postfix_perator])
```

LangletExpr - binary_expr

```
>>> from langscape.sourcetools.langletexpr import*
>>> le = LangletExpr(cfuncall)
>>> for expr in le.expressions(cfuncall.symbol.funcall):
...     if expr[0] == cfuncall.symbol.binary_expr:
...         print expr[1:]

('binary_expr', '+subscript_expr.subscript_expr')
('binary_expr', '++subscript_expr.subscript_expr')
('binary_expr', '~subscript_expr.subscript_expr')
('binary_expr', '--subscript_expr.subscript_expr')
('binary_expr', '+subscript_expr.subscript_expr != funcall()')
...
```

LangletExpr - refinements

```
class CFuncallExpr(LangletExpr):
    @refine
    def funcall(self):
        '''
        funcall: NAME '(' [expr (',' expr){0,3}] ')'
        '''
```

specialization

```
funcall: NAME '(' [expr (',' expr)*] ')'
```

LangletExpr - refinements

More expressions through refinement

```
>>> le = CFuncallExpr(cfuncall)
>>> for expr in le.expressions(cfuncall.symbol.funcall):
...     if expr[0] == cfuncall.symbol.funcall:
...         print expr[1:]

('funcall', 'b(expr, expr)')
('funcall', 'c(expr, expr, expr)')
('funcall', 's(expr, expr, expr, expr)')
('funcall', 'x(expr)')
('funcall', 'y()')
```

Thanks for coming!

www.fiber-space.de

code.google.com/p/langscape/