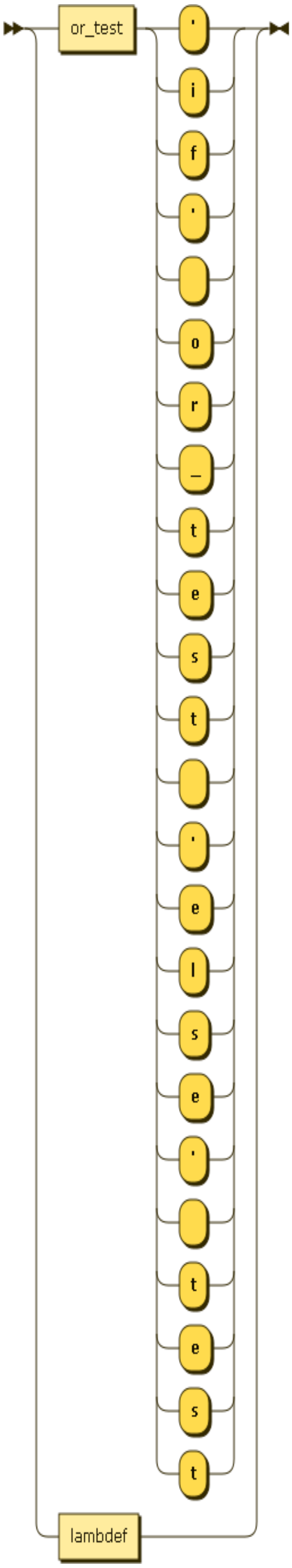
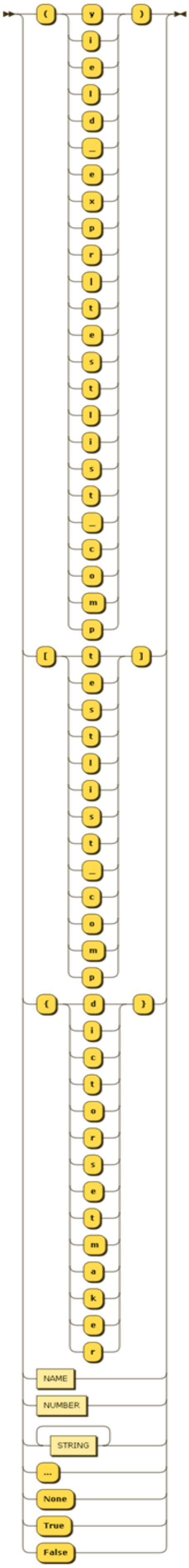
Python; Dissected

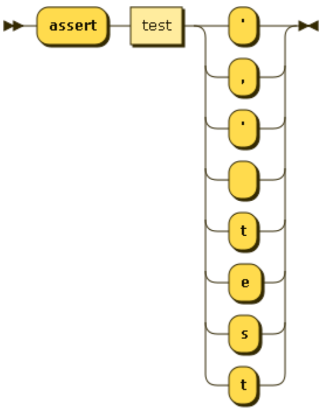
I. Syntax

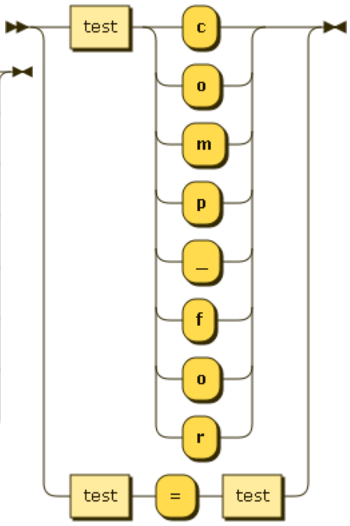
Python continues to captivate novice and adept programmers alike. The simplistic beauty of the language lies in its resemblance to spoken English. The syntax relies heavily upon whitespace for proper formatting, disposing of the classic use of semicolons to denote statement completion; however, Python also implements foundational knowledge of programming that stands the test of time.

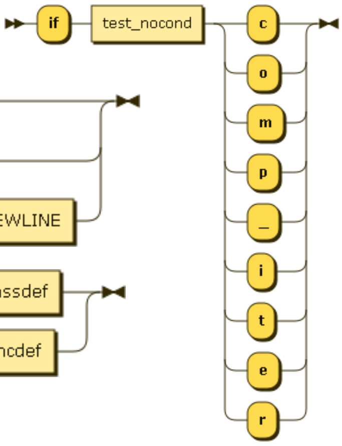
The Python Syntax Diagram that follows was assembled using the most recent stable release of Python's BNF, Python 3.3.2.(1)

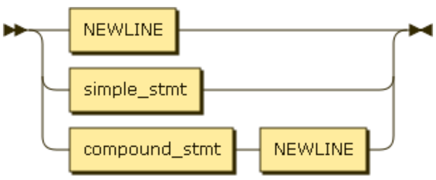


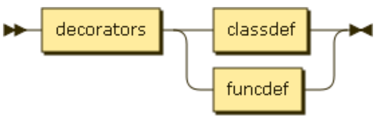


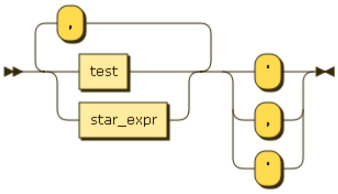


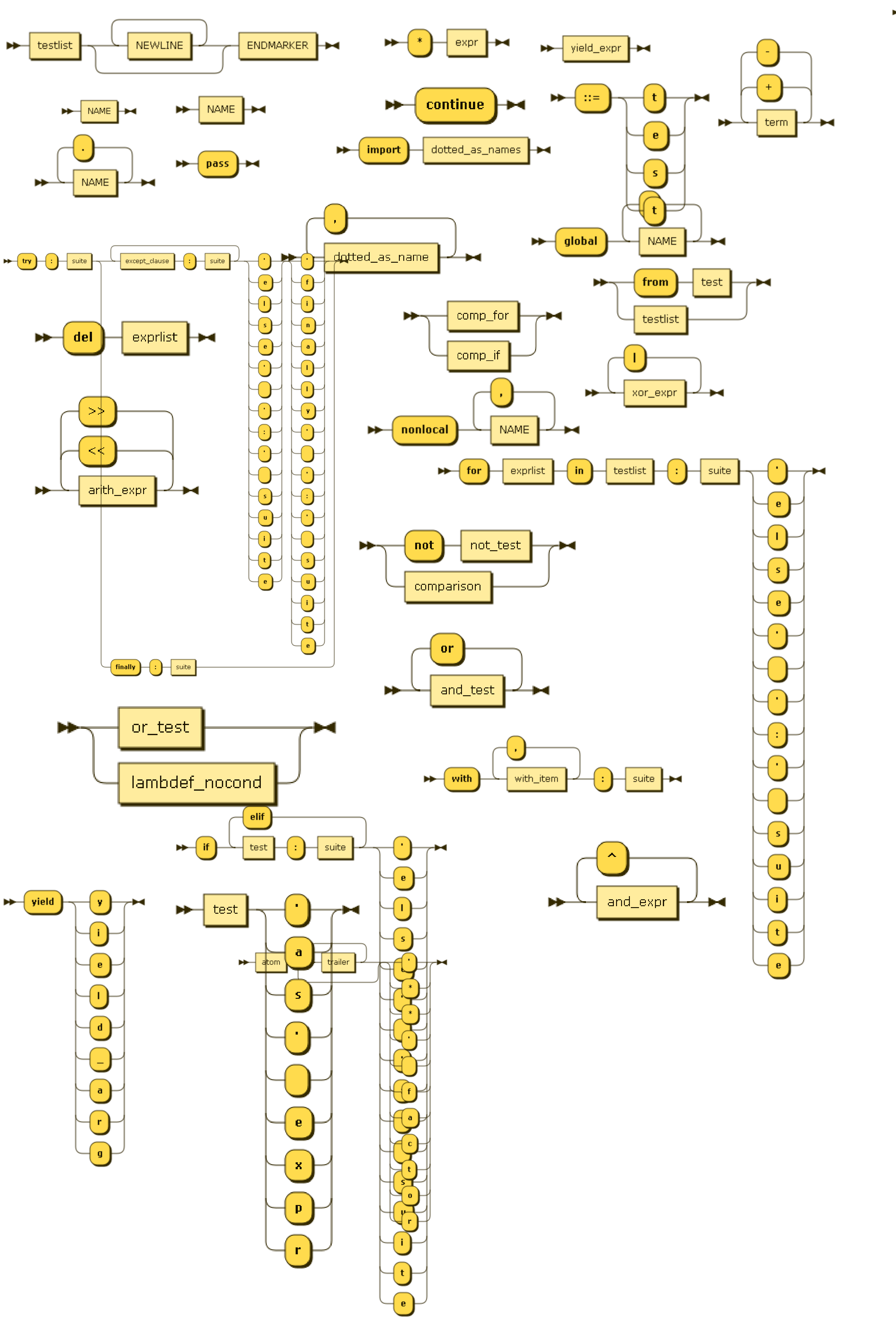




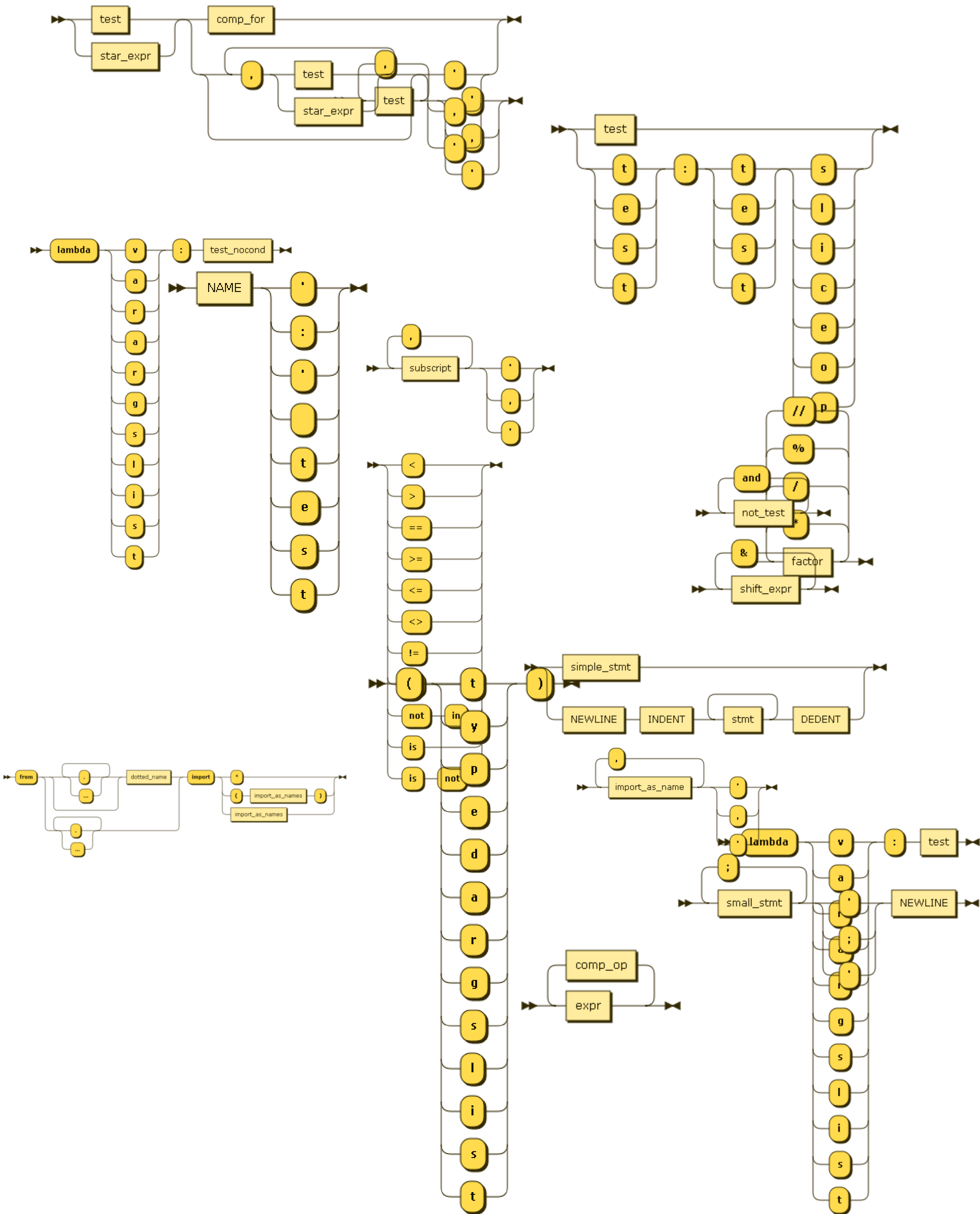




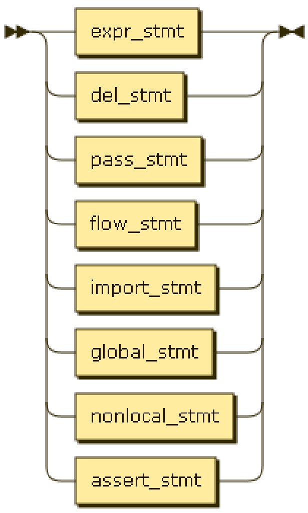


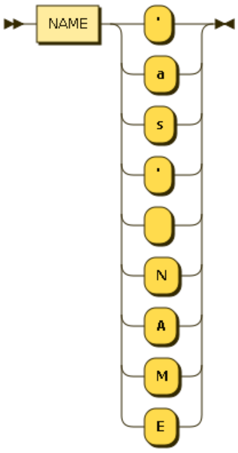


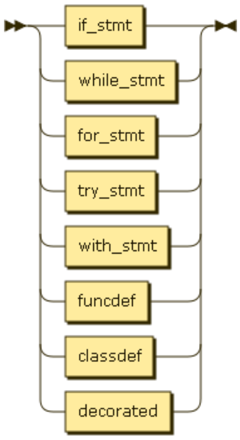
testlist
NEWLINE
ENDMARKER
*
expr
yield_expr
NAME
NAME
continue
::=
t
e
s
t
-
+
term
.
NAME
pass
import
dotted_as_names
global
NAME
,
dotted_as_name
try
:
suite
except_clause
:
suite
finally
:
suite
from
test
testlist
del
exprlist
comp_for
comp_if
|
xor_expr
,
nonlocal
NAME
>>
<<
arith_expr
for
exprlist
in
testlist
:
suite
not
not_test
comparison
or
and_test
or_test
lambdef_nocond
,
with
with_item
:
suite
elif
if
test
:
suite
^
and_expr
yield
test
atom
trailer

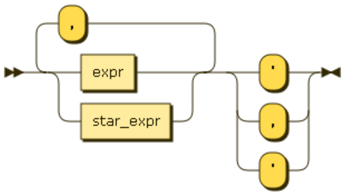


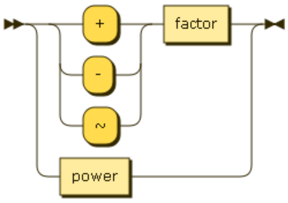
test
star_expr
comp_for
test_nocond
lambda
NAME
subscript
not_test
and
factor
shift_expr
simple_stmt
NEWLINE
INDENT
stmt
DEDENT
import_as_name
small_stmt
comp_op
expr
from
dotted_name
import
import_as_names

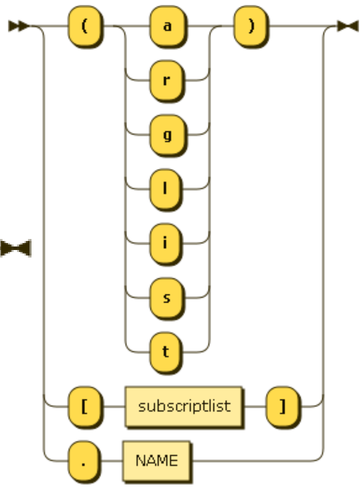

expr_stmt
del_stmt
pass_stmt
flow_stmt
import_stmt
global_stmt
nonlocal_stmt
assert_stmt

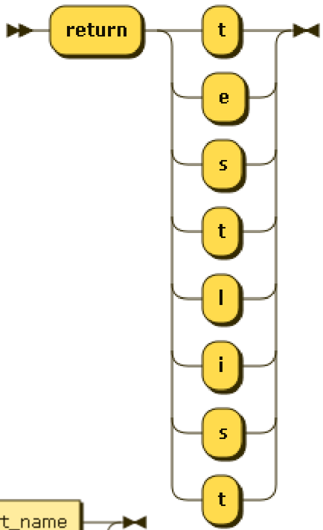

NAME
'
a
s
'
N
A
M
E

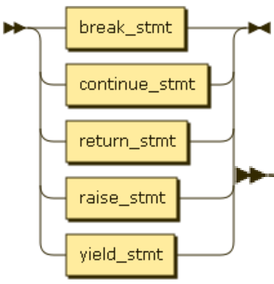

if_stmt
while_stmt
for_stmt
try_stmt
with_stmt
funcdef
classdef
decorated

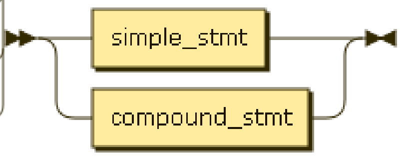

,
expr
star_expr
'
,
'

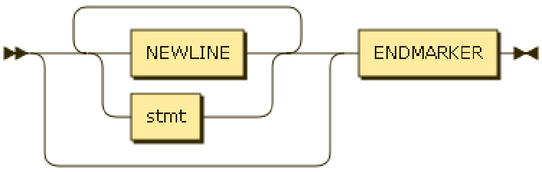

+
-
~
factor
power

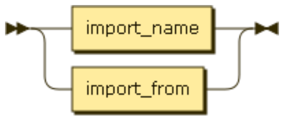

(
a
r
g
l
i
s
t
)
[
subscriptlist
]
.
NAME

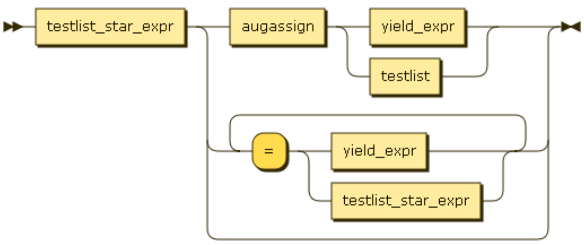

return
t
e
s
t
l
i
s
t

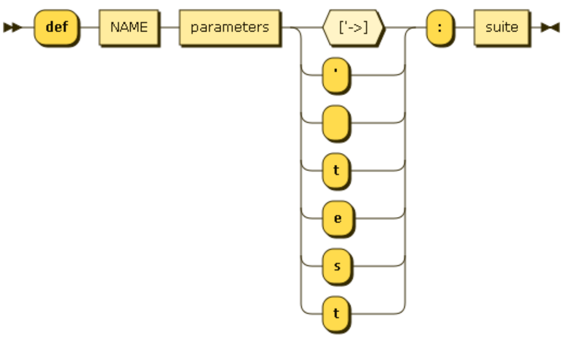

break_stmt
continue_stmt
return_stmt
raise_stmt
yield_stmt

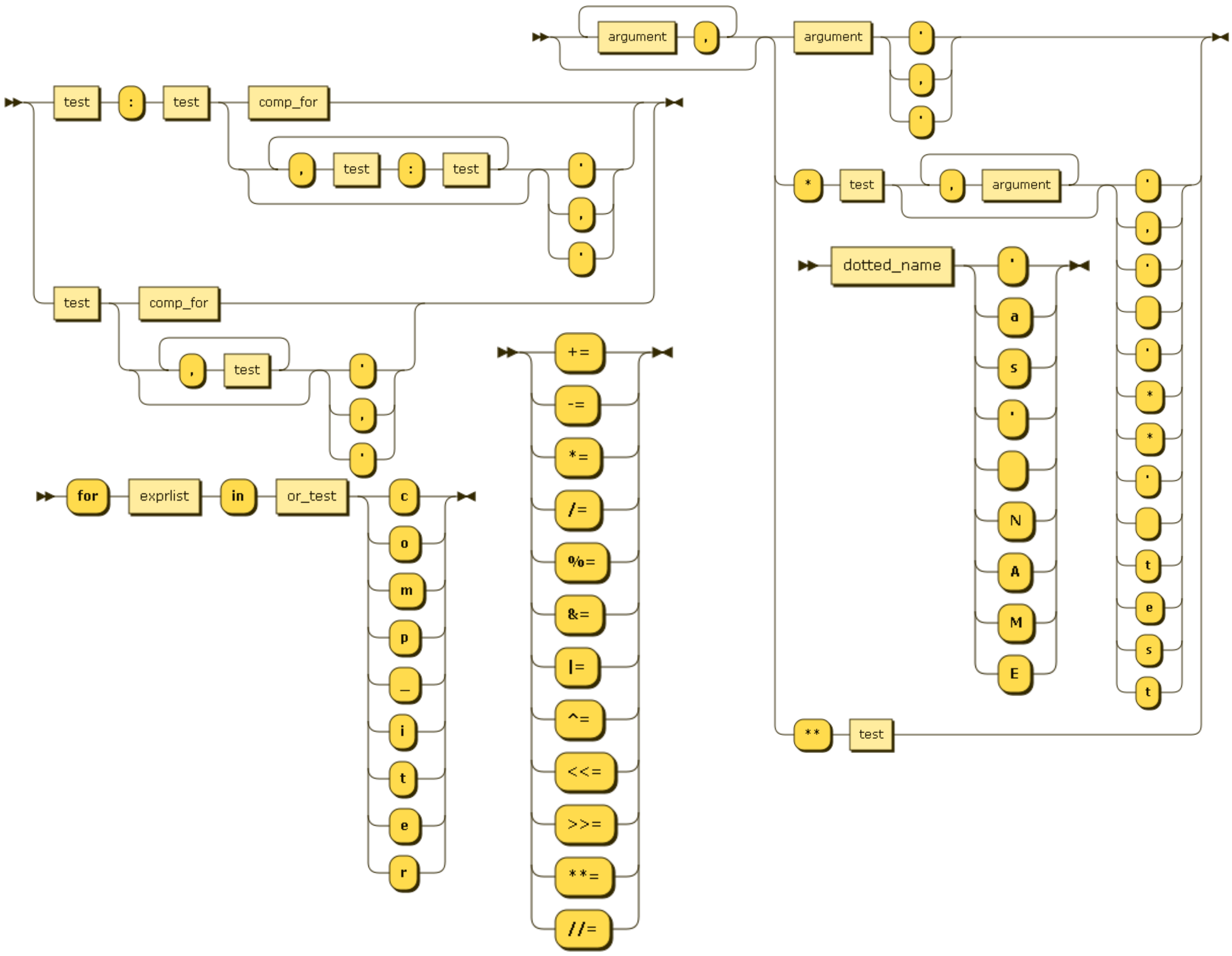

simple_stmt
compound_stmt


NEWLINE
stmt
ENDMARKER


import_name
import_from


testlist_star_expr
augassign
yield_expr
testlist
=
yield_expr
testlist_star_expr


def
NAME
parameters
['->]
'
t
e
s
t
:
suite



II. ToolChain

Python is an interpretative language, so it is not compiled like other popular languages like Java and C. Python source code is automatically compiled into Python bytecode by the CPython interpreter; this is usually stored in PYC (or PYO) file types and is regenerated when the source is updated, or when otherwise necessary.

To distribute a program to the people who already have Python installed, the programs can be shipped either as PY or PYC file types. As Python source code does not need to be compiled, programs are portable under the assumption that all are running the same version of Python. In recent versions, a ZIP archive can be created with PY or PYC files and use a small “bootstrap script” to add that ZIP archive to the path.

Earliest Python’s implementation was a simple LL(1) parser generator called “pgen.c". This parser generator is still part of the Python source distribution and probably the least changed of all the code.

Progress of Python versions:

The earliest version of Python 0.9 Beta had a really small grammar set written in C language.

Historically between 0.9 and 2.4 Python used, compilation from source code to bytecode involved two steps:

1. Parse the source code into a parse tree (Parser/pgen.c)
2. Emit bytecode based on the parse tree (Python/compile.c)

Python 2.5 and later uses these steps for compilation:

1. Parse source code into a parse tree (Parser/pgen.c)
2. Transform parse tree into an Abstract Syntax Tree (Python/ast.c)
3. Transform AST into a Control Flow Graph (Python/compile.c)
4. Emit bytecode based on the Control Flow Graph (Python/compile.c)

Python 3.0 had the first bootstrapped version of the Python compiler written entirely in python.
The process for CPython Compilation:
Source Code → Parse Tree → Abstract Syntax Tree → Control Flow Graph → Byte Code
AST → CFG →Byte Code:
1. Create AST
2. Create CFG
3. Convert AST to Python bytecode
4. Output as bytecode

The core of Python interpreter does have the structure of a classic compiler, because it is thought to be a scripting language. When we invoke the “python” command, our source code is scanned for tokens and the are parsed into a tree representing the logical structure of the program, then it is transformed into bytecode. Finally the bytecode is executed by the virtual machine. It is the job of Python’s parser to take the tokens as an input and produce a Syntax Tree(AST). Python’s custom parser generator automatically generates its parser from a grammar description, then the parse tree is generated. The parse tree is a low level representation of the parsed program in the structure defined by the grammar description.

Python’s bytecode interpreter is a stack-based virtual machine. This means that the process of bytecode execution manipulates a data stack, with instructions adding, removing and operating upon the top couple of stack elements. The execution of Python bytecode is handled by the bytecode interpreter and the interpreter is a stack-based virtual machine that executes Python bytecode.

III. History

Python is a general-purpose, object-oriented scripting language that was written by Guido van Rossen at CWI (Centrum Wislunde & Informatica or National Research Institute for Mathematics and Computer Science) in the Netherlands. Python was created by Rossen in order to address issues with the ABC programming language, but Rossen decided to make a language that was generally extensible. Python was created with its core syntax directly from the ABC programing language, but some of its syntax was also provided from C. The Bourne shell was used as the interpreter model that became interactive when run without arguments. Languages like LISP and Haskell provided the list comprehensions, lexical closures, and more for Python. The Icon programming language was used as inspiration for Python's generators and iterators. Python's exception model and module system were based on the Modula-3 programming language. Perl's regular expressions were borrowed by Python for its string manipulation and its standard library were influenced by Java(11).

Version Release Dates:

- Python 1.0 - January 1994
    - Python 1.5 - December 31, 1997
    - Python 1.6 - September 5, 2000
- Python 2.0 - October 16, 2000
    - Python 2.1 - April 17, 2001
    - Python 2.2 - December 21, 2001
    - Python 2.3 - July 29, 2003
    - Python 2.4 - November 30, 2004
    - Python 2.5 - September 19, 2006
    - Python 2.6 - October 1, 2008
    - Python 2.7 - July 3, 2010
- Python 3.0 - December 3, 2008
    - Python 3.1 - June 27, 2009
    - Python 3.2 - February 20, 2011
    - Python 3.3 - September 29, 2012
    - Python 3.4 - Still in beta

Python's typing mechanism that of a dynamically typed language, but it's also moderately type-checked at run-time. Dynamically-typed refers to the fact that types aren't assigned until a value is given to the variable. Python is also strongly typed, which means that it prevents operations that aren't well defined rather than attempting to make sense of them. Python has an implicit conversion defined for numeric types, which means for example, that it can do arithmetic operations on an integer and a complex number without type casting. However, no there is no implicit conversion between numbers and strings(12).

Python's syntax was based mainly on the ABC programming language, with some syntax also taken from C. However, like most languages, Python also has many keywords and reserved words that can't be used as identifiers (continue, break and etc.). Python's syntax uses whitespaces to delimit program and function blocks, but it also follows the off-side rule. This means that blocks in a programming language are expressed by their indentation. This feature is taken from Python's processor ABC.

Semantics defines the relationship between the syntax of a language and the computational model. The semantics of Python are split into two types, static semantics and dynamic semantics. The static semantics define restrictions on the structure of statements that are difficult to express; these are rules that need to be checked during compile time. The dynamic semantics or execution semantics refer to the fact that once data has been specified, the machine needs to be instructed on how to perform tasks on data(13).

Pragmatics mainly defines the usability of the language, the application areas and its ease of implementation. However, in order to accomplish this, there needs to less ambiguity in Python because there is still some ambiguity in Python. For example, the '/' symbol can mean either the divide operation or the floor function. The structure of Python's control statements reduces some of the ambiguity of statements, but it doesn't remove all of it. This is major issue with many programming languages.

Python was created by Guido van Rossen, in order to deal with issues with the ABC programming language. Python is a dynamically and strongly typed language and its syntax is based on ABC's syntax features. Python is a general-purpose, object-oriented programming language that uses both static semantics and dynamic semantics.

IV. Inference

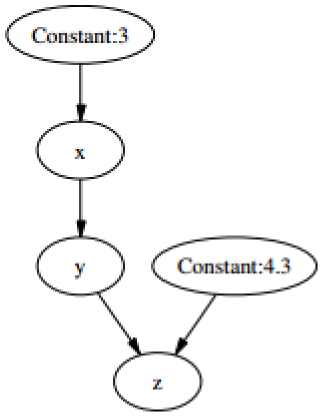
Python is a dynamic language; that is, the entire code written for Python is interpreted at run time. Python only checks for syntax or semantic errors while it's running. Therefore, Python is forced to check types and type operations as the interpreter runs through the code, effectively slowing down the overall runtime. Therefore, if we eliminate the need to type check at run time, we can significantly improve the program's performance. One way to achieve this optimization is to precompile our Python code. This can be done by introducing type inference.[14]

The main advantage of implementing type inference is that it allows the code to be precompiled. Precompiling the code with type inference can greatly improve performance of Python code[14]. Other advantages of type inference include avoiding needless and superfluous annotations and in some cases making code easier to read. Also, without the need of most annotations, programmers can focus programming methods and algorithms instead of focusing on conditions to meet the type-checker. However, with its various advantages also come many disadvantages, such as, errors due to unification failure and error messages due to actual type error[15].

There is a static type inferencer, called Starkiller, that was developed by Michael Salib[14]. Starkiller takes in Python source code and infers the information needed to make native code used for compilation . In order to do this, the Starkiller algorithm constructs a dataflow network for types that models the runtime behavior of values in the input program. The dataflow network is created with nodes linked together with constraints. The nodes correspond to variables and expressions in the input program, and they contain a set of types that the variable or expression can achieve at runtime. Constraints link nodes together based on data flow between two nodes. The constraint is a unidirectional link and forces the receiving node to contain all the elements in the sender node. This allows types to flow along constraints, so when a node gets a new type the type is passed along to all other nodes connected to it. Since Starkiller does not have runtime information, it has to make approximations and will have an overly broad type set for expressions. It will, however, always infer every type that appears at runtime[14].

Ex:

```
x = 3
y = x
z = y
z = 4.3
```



Variable	Type Set
x	{int}
y	{int}
z	{int, float}

Works Cited

1. Unknown (2013). *Full grammar specification*. Retrieved from http://docs.python.org/3.3/reference/grammar.html

2. Unknown (2013). Railroad Diagram Generator. Retrieved from http://railroad.my28msec.com/rr/ui

3. Hansper, George (2000). *Yacc, a parser generator*. Retrieved from http://luv.asn.au/overheads/lex_yacc/yacc.html

4. Jinks, Pete (2004). *Notations for context-free grammars*. Retrieved from http://www.cs.man.ac.uk/~pjj/bnf/bnf.html

5. Unknown, (1997) *The Python Programming Language* <http://groups.engin.umd.umich.edu/CIS/course.des/cis400/python/python.html>

6. R. Guido Van (2009) *The History of Python* <http://python-history.blogspot.ca/2009/02/early-language-design-and-development.html>

7. Lundh, Fredrik (2003). *Compiling Python Code.* <http://effbot.org/zone/python-compile.htm>

8. Unknown (2013). Python:Using the Python Interpreter. <http://docs.python.org/2/tutorial/interpreter.html>

9. Unknown (2012). Python Developer's Guide: Design of CPython's compiler. <http://docs.python.org/devguide/compiler.html>

10. Cottingham, Matt (2012). *Static Modification of Python with Python: the AST Module*. <http://blueprintforge.com/blog/2012/02/27/static-modification-of-python-with- python-the-ast-module/>

11. http://www.manuinfo.com/syntax-and-semantics-of-python/

12. ttp://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/intro.htm

13. http://python-history.blogspot.ca/
14. Salib, Michael (2004). *Faster than C: Static Type Inference with Starkiller*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.3786&rep=rep1&type=pdf
15. Unknown (2010). *Type Inference*. Retrieved from http://c2.com/cgi/wiki?TypeInference