# Grammar for 2to3. This grammar supports Python 2.x and 3.x.

# NOTE WELL: You should also follow all the steps listed at

# https://devguide.python.org/grammar/

# Start symbols for the grammar:

# file\_input is a module or sequence of commands read from an input file;

# single\_input is a single interactive statement;

# eval\_input is the input for the eval() and input() functions.

# NB: compound\_stmt in single\_input is followed by extra NEWLINE!

file\_input: (NEWLINE | stmt)\* ENDMARKER

single\_input: NEWLINE | simple\_stmt | compound\_stmt NEWLINE

eval\_input: testlist NEWLINE\* ENDMARKER

decorator: '@' dotted\_name [ '(' [arglist] ')' ] NEWLINE

decorators: decorator+

decorated: decorators (classdef | funcdef | async\_funcdef)

async\_funcdef: ASYNC funcdef

funcdef: 'def' NAME parameters ['->' test] ':' suite

parameters: '(' [typedargslist] ')'

typedargslist: ((tfpdef ['=' test] ',')\*

('\*' [tname] (',' tname ['=' test])\* [',' ['\*\*' tname [',']]] | '\*\*' tname [','])

| tfpdef ['=' test] (',' tfpdef ['=' test])\* [','])

tname: NAME [':' test]

tfpdef: tname | '(' tfplist ')'

tfplist: tfpdef (',' tfpdef)\* [',']

varargslist: ((vfpdef ['=' test] ',')\*

('\*' [vname] (',' vname ['=' test])\* [',' ['\*\*' vname [',']]] | '\*\*' vname [','])

| vfpdef ['=' test] (',' vfpdef ['=' test])\* [','])

vname: NAME

vfpdef: vname | '(' vfplist ')'

vfplist: vfpdef (',' vfpdef)\* [',']

stmt: simple\_stmt | compound\_stmt

simple\_stmt: small\_stmt (';' small\_stmt)\* [';'] NEWLINE

small\_stmt: (expr\_stmt | print\_stmt | del\_stmt | pass\_stmt | flow\_stmt |

import\_stmt | global\_stmt | exec\_stmt | assert\_stmt)

expr\_stmt: testlist\_star\_expr (annassign | augassign (yield\_expr|testlist) |

('=' (yield\_expr|testlist\_star\_expr))\*)

annassign: ':' test ['=' test]

testlist\_star\_expr: (test|star\_expr) (',' (test|star\_expr))\* [',']

augassign: ('+=' | '-=' | '\*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^=' |

'<<=' | '>>=' | '\*\*=' | '//=')

# For normal and annotated assignments, additional restrictions enforced by the interpreter

print\_stmt: 'print' ( [ test (',' test)\* [','] ] |

'>>' test [ (',' test)+ [','] ] )

del\_stmt: 'del' exprlist

pass\_stmt: 'pass'

flow\_stmt: break\_stmt | continue\_stmt | return\_stmt | raise\_stmt | yield\_stmt

break\_stmt: 'break'

continue\_stmt: 'continue'

return\_stmt: 'return' [testlist]

yield\_stmt: yield\_expr

raise\_stmt: 'raise' [test ['from' test | ',' test [',' test]]]

import\_stmt: import\_name | import\_from

import\_name: 'import' dotted\_as\_names

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' | 'nonlocal') NAME (',' NAME)\*

exec\_stmt: 'exec' expr ['in' test [',' test]]

assert\_stmt: 'assert' test [',' test]

compound\_stmt: if\_stmt | while\_stmt | for\_stmt | try\_stmt | with\_stmt | funcdef | classdef | decorated | async\_stmt

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]

with\_var: 'as' expr

# NB compile.c makes sure that the default except clause is last

except\_clause: 'except' [test [(',' | 'as') test]]

suite: simple\_stmt | NEWLINE INDENT stmt+ DEDENT

# Backward compatibility cruft to support:

# [ x for x in lambda: True, lambda: False if x() ]

# even while also allowing:

# lambda x: 5 if x else 2

# (But not a mix of the two)

testlist\_safe: old\_test [(',' old\_test)+ [',']]

old\_test: or\_test | old\_lambdef

old\_lambdef: 'lambda' [varargslist] ':' old\_test

test: or\_test ['if' or\_test 'else' test] | lambdef

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)\*

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: [AWAIT] atom trailer\* ['\*\*' factor]

atom: ('(' [yield\_expr|testlist\_gexp] ')' |

'[' [listmaker] ']' |

'{' [dictsetmaker] '}' |

'`' testlist1 '`' |

NAME | NUMBER | STRING+ | '.' '.' '.')

listmaker: (test|star\_expr) ( comp\_for | (',' (test|star\_expr))\* [','] )

testlist\_gexp: (test|star\_expr) ( comp\_for | (',' (test|star\_expr))\* [','] )

lambdef: 'lambda' [varargslist] ':' test

trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME

subscriptlist: subscript (',' subscript)\* [',']

subscript: test | [test] ':' [test] [sliceop]

sliceop: ':' [test]

exprlist: (expr|star\_expr) (',' (expr|star\_expr))\* [',']

testlist: test (',' test)\* [',']

dictsetmaker: ( ((test ':' test | '\*\*' expr)

(comp\_for | (',' (test ':' test | '\*\*' expr))\* [','])) |

((test | star\_expr)

(comp\_for | (',' (test | star\_expr))\* [','])) )

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

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

# "test '=' test" is really "keyword '=' test", but we have no such token.

# These need to be in a single rule to avoid grammar that is ambiguous

# to our LL(1) parser. Even though 'test' includes '\*expr' in star\_expr,

# we explicitly match '\*' here, too, to give it proper precedence.

# Illegal combinations and orderings are blocked in ast.c:

# multiple (test comp\_for) arguments are blocked; keyword unpackings

# that precede iterable unpackings are blocked; etc.

argument: ( test [comp\_for] |

test '=' test |

'\*\*' test |

'\*' test )

comp\_iter: comp\_for | comp\_if

comp\_for: [ASYNC] 'for' exprlist 'in' testlist\_safe [comp\_iter]

comp\_if: 'if' old\_test [comp\_iter]

testlist1: test (',' test)\*

# not used in grammar, but may appear in "node" passed from Parser to Compiler

encoding\_decl: NAME

yield\_expr: 'yield' [yield\_arg]

yield\_arg: 'from' test | testlist