October 3, 2013

The Python Programming Language

Prepared For:

Mantis Cheng, CSC 330

University of Victoria

Written By:

Jasmine Mago, Freja Kelleris

Laura Rossetto, Yuma Katayama

Samuel Navarrete, Daniel Oon

Josh Stelting, Max Gunton

Trevor Schiavone, Breanne Gruenke

# 

# Table of Contents

1..……………………….Grammar

2..……………………….Compiler Creation (T-Diagrams)

3..……………………….History

3.1………………………..…Introduction

3.2………………………..…Type Mechanism

3.3………………………..…Syntax

3.4………………………..…Semantics

3.5………………………..…Pragmatics

3.6………………………..…Implementation Techniques

4..……………………….Types

4.1…………………………..Introduction

4.2…………………………..Compiler Based Implementations

4.3…………………………..Tool/Debugger Based Implementations

4.4…………………………..Benefits of Type Inference

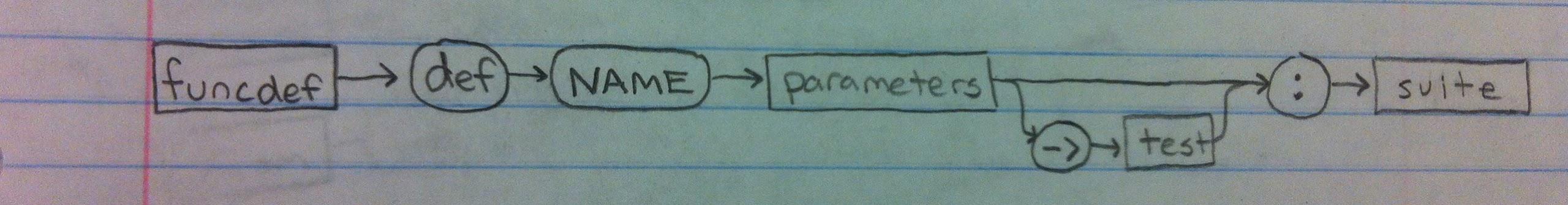
5..……………………….Appendix

6..……………………….References

**1. Grammar**

Python's grammar, along with its types, allow for programmers to make programs that are much easier to read and are generally shorter overall. While most features in Python can be reproduced in other languages, it is Python’s ease of use and power that make it stand out.

Unlike most other programming languages, Python does not use curly braces to hold code blocks, but rather just uses indentation which makes it easier for the programmer and others to read. A complete diagram of Python’s BNF grammar can be found in the Appendix.

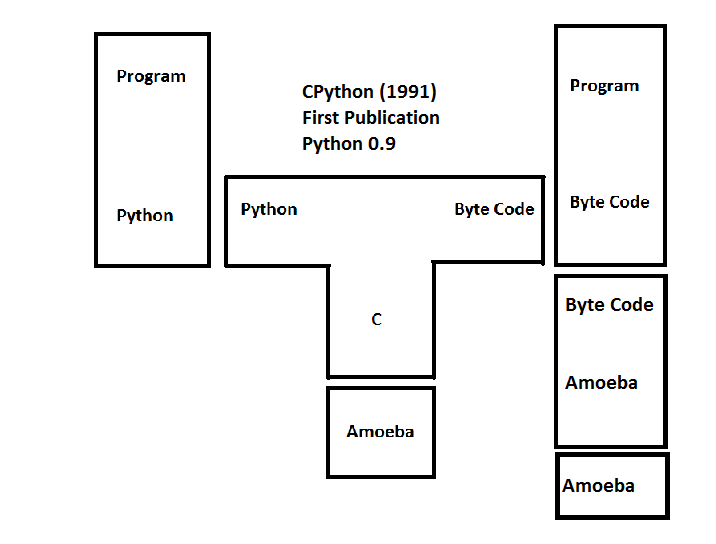


**2. Compiler Creation (T-Diagrams)**

Python has many different implementations. The original Python was the CPython interpreter written in C. Another Python compiler of interest is the PyPy compiler. The following T-Diagrams illustrate how Python programs execute with these two compilers.

## CPython Interpreter (from 1991)

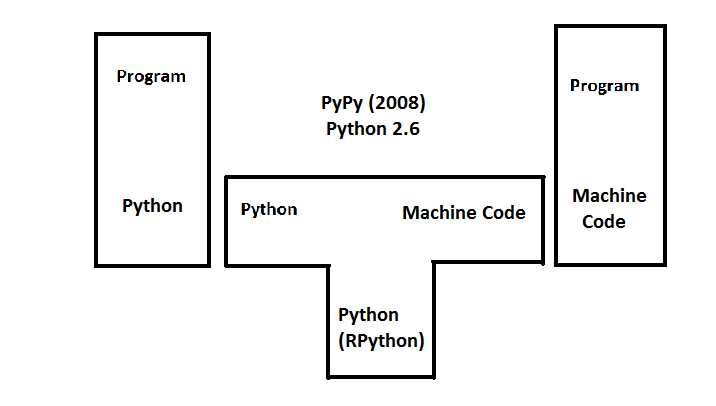
Figure 1.1: T-Diagram for CPython interpreter



CPython was the original Python interpreter. It was developed in hopes for a better alternative to the ABC programming language in the Amoeba OS. The T-Diagram above shows how CPython translates Python code into bytecode. The CPython interpreter runs on the target architecture (Amoeba in this case) and executes the bytecode. It was published in 1991.

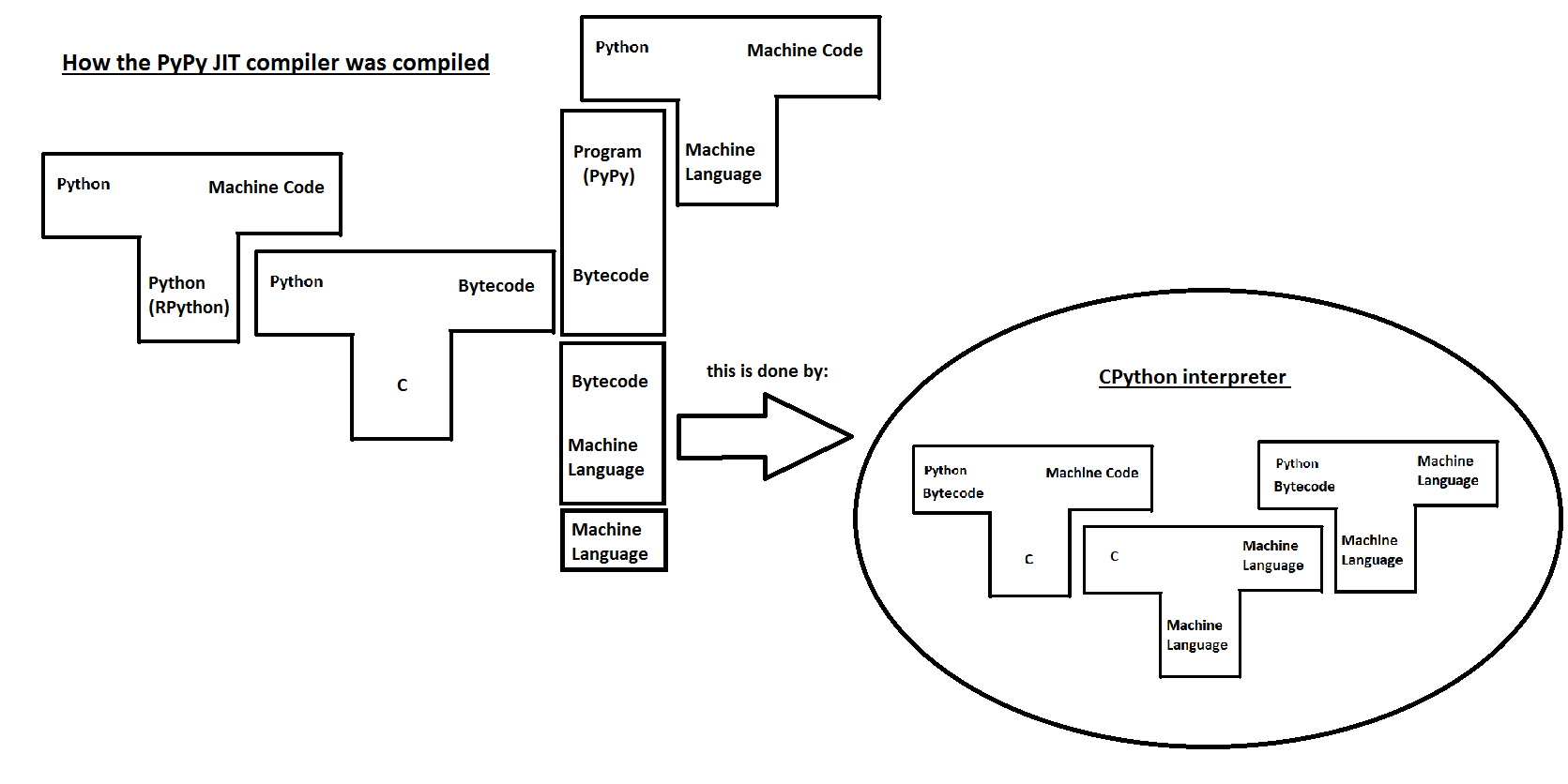
## PyPy JIT Compiler

Figure 1.2: T-Diagram for PyPy JIT Interpreter



First published in 2008, the PyPy JIT compiler is written in RPython (restricted Python) and translates python code into machine code (in a way similar to how Java apps run on Android). Unlike CPython, PyPy compiles at runtime by JIT tracing; it works by optimizing loops that are most often executed.

Figure 2: How the PyPy JIT compiler was compiled



PyPy was written in a restricted version of Python called RPython. The example above illustrates how a Python program (in this case the PyPy compiler) is interpreted by the CPython interpreter.

The RPython (that PyPy is written in) is translated into byte code as we showed in the previous example, and the CPython interpreter translates the program.

**3. History of Python**

## 3.1 Introduction

The Python programming language was designed in 1990 by Guido van Rossum. It was created with the intent of becoming a scripting language on the Amoeba OS. The design goals for Python were to create a simple but powerful modular programming language which allowed for easy embedding and extension with other languages as well as remaining extremely readable.In order to achieve his design speciations, Rossum drew influence from many other languages. The most prominent of these was ABC, however, languages C and Modular 3 were also important contributors in the implementation of Python. In this brief overview of Python’s history, we explore how these languages contributed to the syntax, semantics, implementation techniques, pragmatics, typing mechanisms, and overall development of the python language.

## 3.2 Type Mechanism

The first aspect we wish to discuss is the type mechanisms of Python. Python is a dynamically typed language. This means that similarly to Lisp and Smalltalk, instead of type checking during compilation, type checking occurs during run-time.While influenced by C in many areas, this is one case in which the two languages differ.

Like most of the features of Python, ABC influenced this decision. Whereas ABC uses dynamic type checking for global variables and static typing for local variables, Python uses all dynamic type checking.

## 3.3 Syntax

Guido van Rossum “didn’t like Perl’s syntax”. Python’s syntax was instead inspired by van Rossum’s first learned languages Algol 60, Pascal, Algol 68, C and of course ABC. Many of the keywords in Python are borrowed from C, such as if, else, and while, and the naming rules for identifiers are the same.

What sets Python apart from these languages is its goal for readability, thus keywords are used where other languages use punctuation, and it has fewer syntactic constructions.

## 3.4 Semantics

As previously mentioned, Python was designed to be a language that could be easily used as an extension of another language. Therefore, as with many of its other features, its semantics were strongly influenced by various other languages. Examples of this include Python strings, which were designed to be very similar to ABC strings; the exception being the uses of different notation and 0-based indexing. Another example is Python’s class mechanisms. These mechanisms add classes with minimal use of new syntax and semantics. To accomplish this, a mixture of C++ and Modula-3’s class mechanisms was applied to Python. Similar to C, class members in Python are public while member functions are virtual. Like Modula-3, Python does not allow shorthand reference to object members from a method. It’s final significant influence came from SmallTalk; like SmallTalk, Python’s classes are themselves objects.

## 3.5 Pragmatics

Like the other aspects of Python, its pragmatics were based on the pros and cons of ABC. ABC was designed as an educational language with the goal of isolating the user. User-friendly terminology was used, and the language was intended to be flawless.

Guido van Rossum strived to make Python completely different in this regard. He designed Python as a language that would evolve with user input, and be as simple to use as possible without ‘dumbing it down’ for the user. Because of this, Python has lended itself to many open source projects.

## 3.6 Implementation Techniques

The final thing we wish to discuss is the implementation techniques of Python.

Once again, Python’s implementation techniques are based on ABC’s. While the major data types are closely derived, some changes have been made. Examples of these changes include a hash table implementation instead of a B-tree implementation, and the implementation of numbers, specifically Python’s two integer types.

Some implementation techniques are also influenced by functional languages such as Lisp and Scheme. Several operations with lists and functions that are common in these languages were not originally part of Python. Users began implementing such functions, and in January 1994 map(), filter() and reduce() were added to the standard library.

Types

## 4.1 Introduction

In the past there have been two primary algorithms used for type inference of programming languages: the Hindley-Milner algorithm and the Cartesian Product algorithm. In general, the Hindley-Milner algorithm has been used for functional languages, whereas the Cartesian product algorithm is best suited for object-oriented programming. Since Python is an object-oriented program, it would seem that the Cartesian product algorithm could be used; however, due to Python’s dynamic type declarations this proves to be difficult in practice. In Python, all the control flow information needed for an accurate type inference may not be available at compile-time. As well, there is no guarantee that the compile and run-time codes will be the same. (B. Cannon, 2005**)**

Several alternate implementations have been proposed for python, specifically to address some of the problems of dynamic typing. We have divided these into compiler and tool based implementations. Details of a few of these projects are described below followed by a summary of the advantages and disadvantages of using type inference for Python.

## 4.2 Compiler Based Implementations

**Psyco** (Note: Psyco compiler is no longer supported<http://psyco.sourceforge.net/>)

Psyco is a just-in-time (JIT) compiler that is a re-implementation of the main

eval loop for Python. It tries to detect ints and strings that are consistent from compile-time to run-time. With this information, the new eval loop emits x86 assembly to perform calculations on those variables. Specifically, it attempted to involve elements of static inference in Python. Psyco infers locally deﬁned ints and strings directly and does not modify any other types making it only a very primitive example of type inferencing. (B. Cannon, 2005)

**Starkiller**

Starkiller is a type inferencer and compiler that analyzes Python source programs and converts them into equivalent C++ programs. Starkiller uses a modified version of the Cartesian Product Algorithm. The type inference algorithm handles both data and parametric polymorphism, thus improving precision. Starkiller supports the entire Python language except for dynamic code insertion features such as eval and dynamic module loading.

The type inferencer correctly analyzes programs that change an instance’s class at runtime as well as programs that change inheritance relationships at runtime but cannot understand programs that use for loops or tuple unpacking as many simple and uninteresting constructs were passed over in the interest of time. The compiler itself supports only a very small subset of the language, consisting of basic arithmetic, simple functions, while loops, conditionals, and primitive forms of IO. (M. Saleb, 2004)

## 

## 4.3 Tool/Debugger Based Implementations

**PyCharm (2.7 or newer)**

“PyCharm is a Python IDE with unique code assistance and analysis for productive Python development. Its debugger is able to infer type information at runtime, and use it to improve code completion suggestions, code insight, and quick documentation popups. PyCharm remembers the different types encountered during runtime.” (<http://blog.jetbrains.com/pycharm/2013/02/dynamic-runtime-type-inference-in-pycharm-2-7/>)

**Aggressive Type Inference (ATI)**

Aggressive Type Inference assumes that in general people do not write dynamically-typed programs even though it is possible. For a program to be used with ATI, it must be written in such a way that ATI can infer an exact type for all variables.

Implementation:

1. The input Python program is scanned and parsed. (No semantic checks are performed) All information relevant to ATI is distilled into summary information and saved. All information about control flow (ie. branches and loops) is discarded. Information that *is* kept includes: the scope of classes, methods, and functions (scope), variable assignments (assign) operations on variables (op), method/function return types (return), the types of names (type), equivalences between names (equ), import statements (import), and global declarations (global).
2. The summary information is repeatedly examined in order to propagate type information.

For example, given the summary information:

|  |
| --- |
| assign x = y  assign y = #t1  type #t1 is string |
|  |

ATI would discover on the first pass that there are three names: x, y, and #t1 (a temporary name generated by phase 1). It would also note that #t1 has the type *string*. On the second pass, ATI would find that y has the type *string* too. Finally, ATI would conclude on the third pass that x is a *string*. ATI can be given either partial or whole-program information, as appropriate. [(http://www.python.org/workshops/2000-01/proceedings/papers/aycock/aycock.html](http://www.python.org/workshops/2000-01/proceedings/papers/aycock/aycock.html))

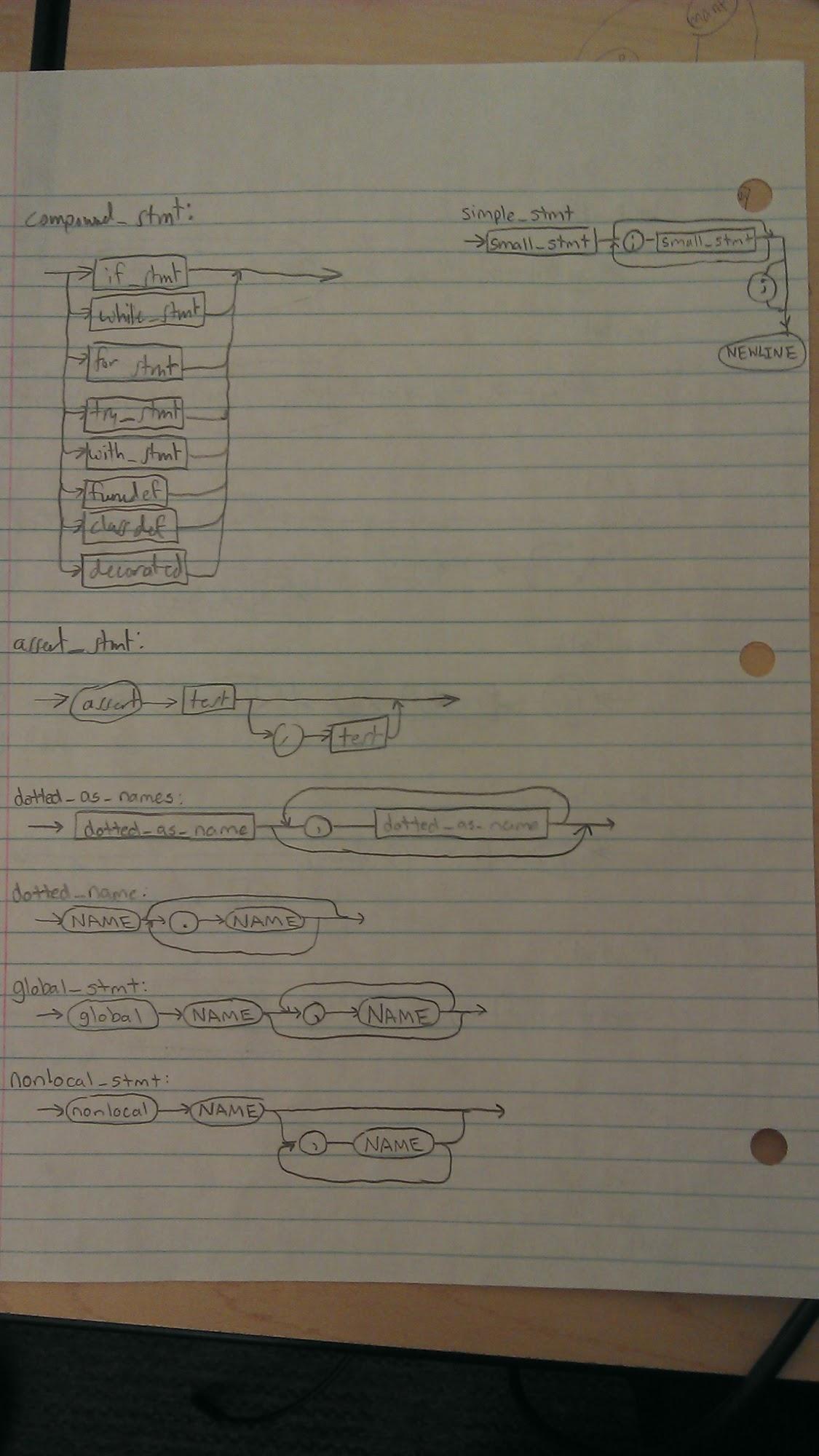
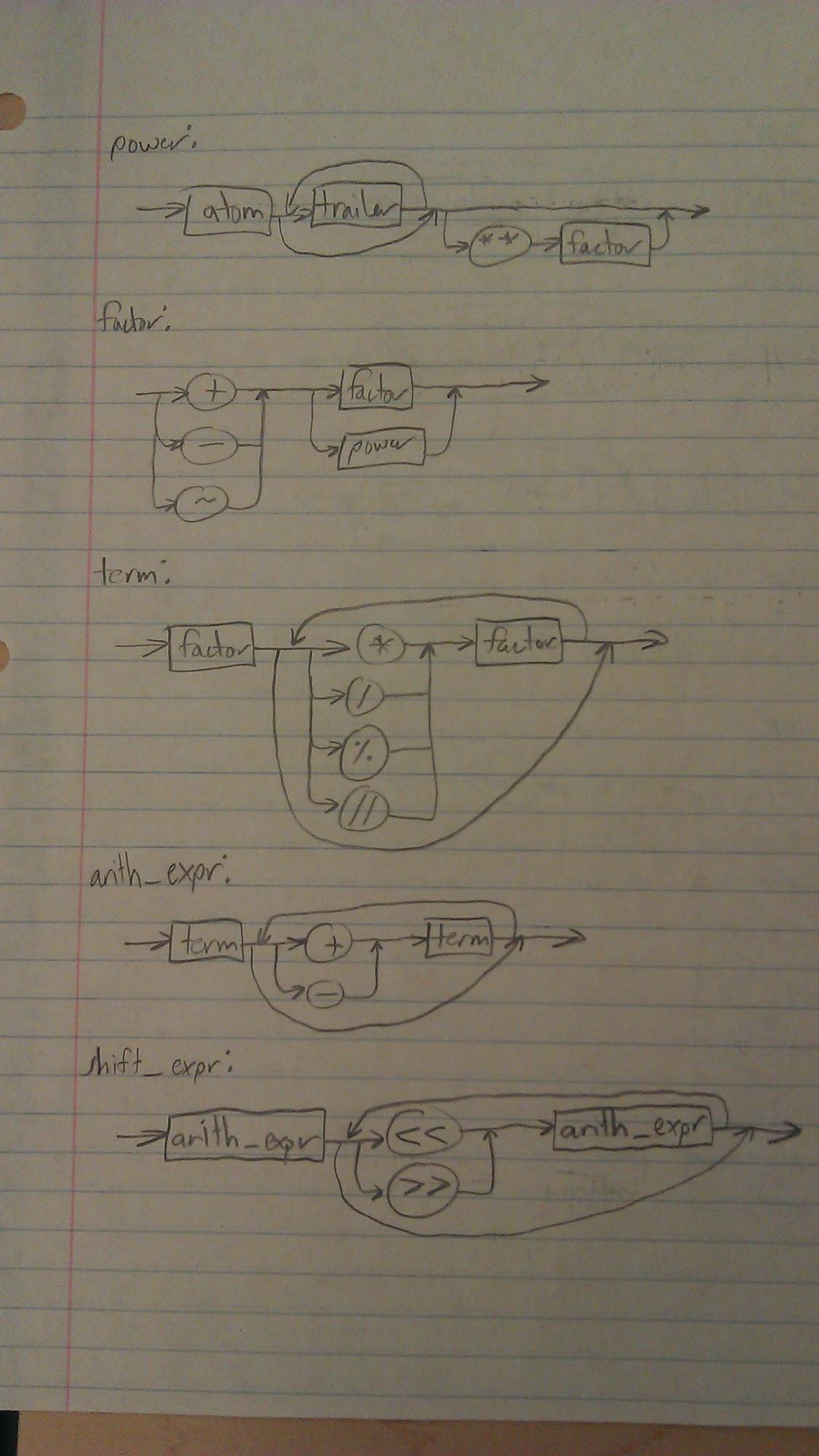
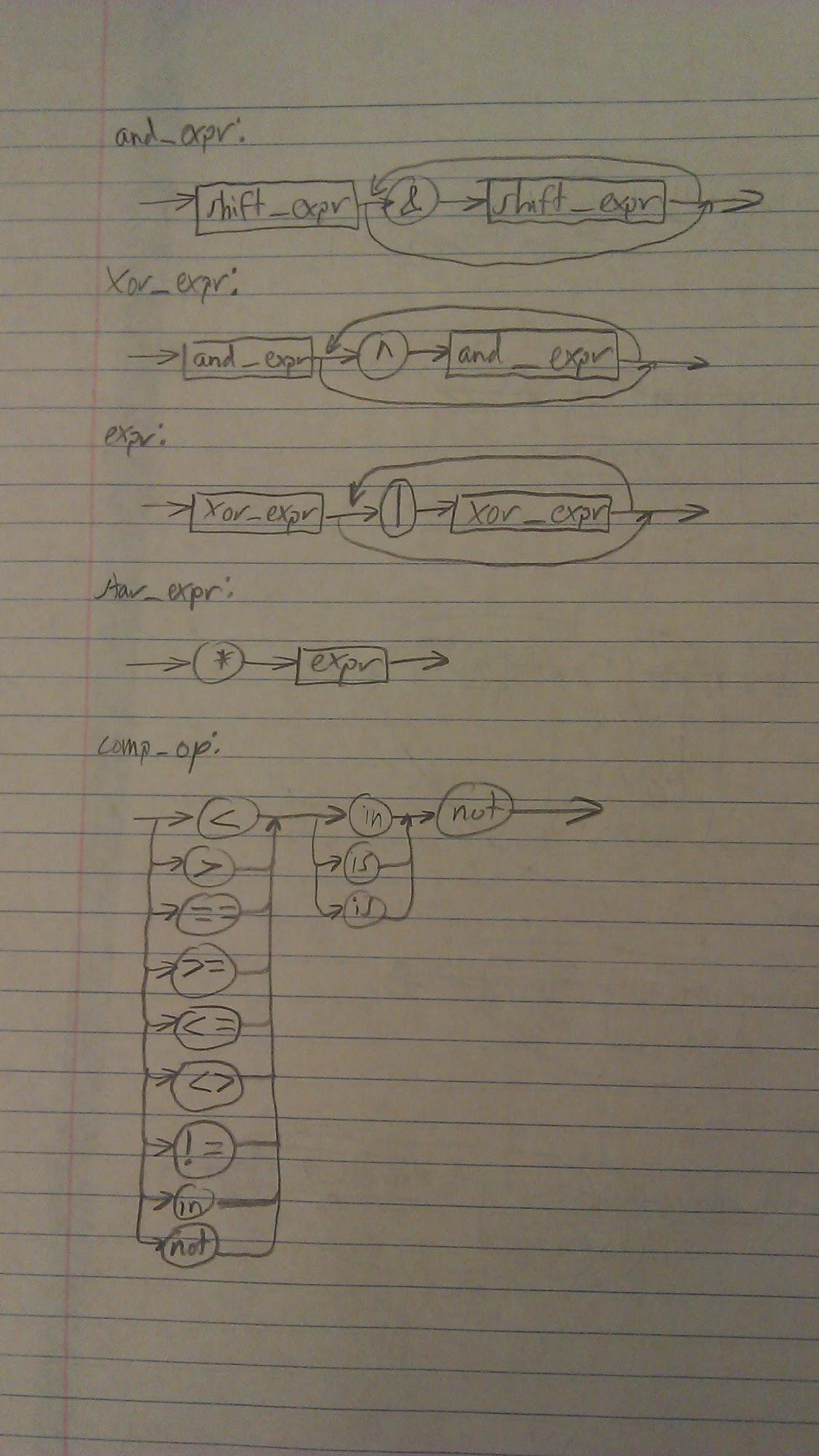
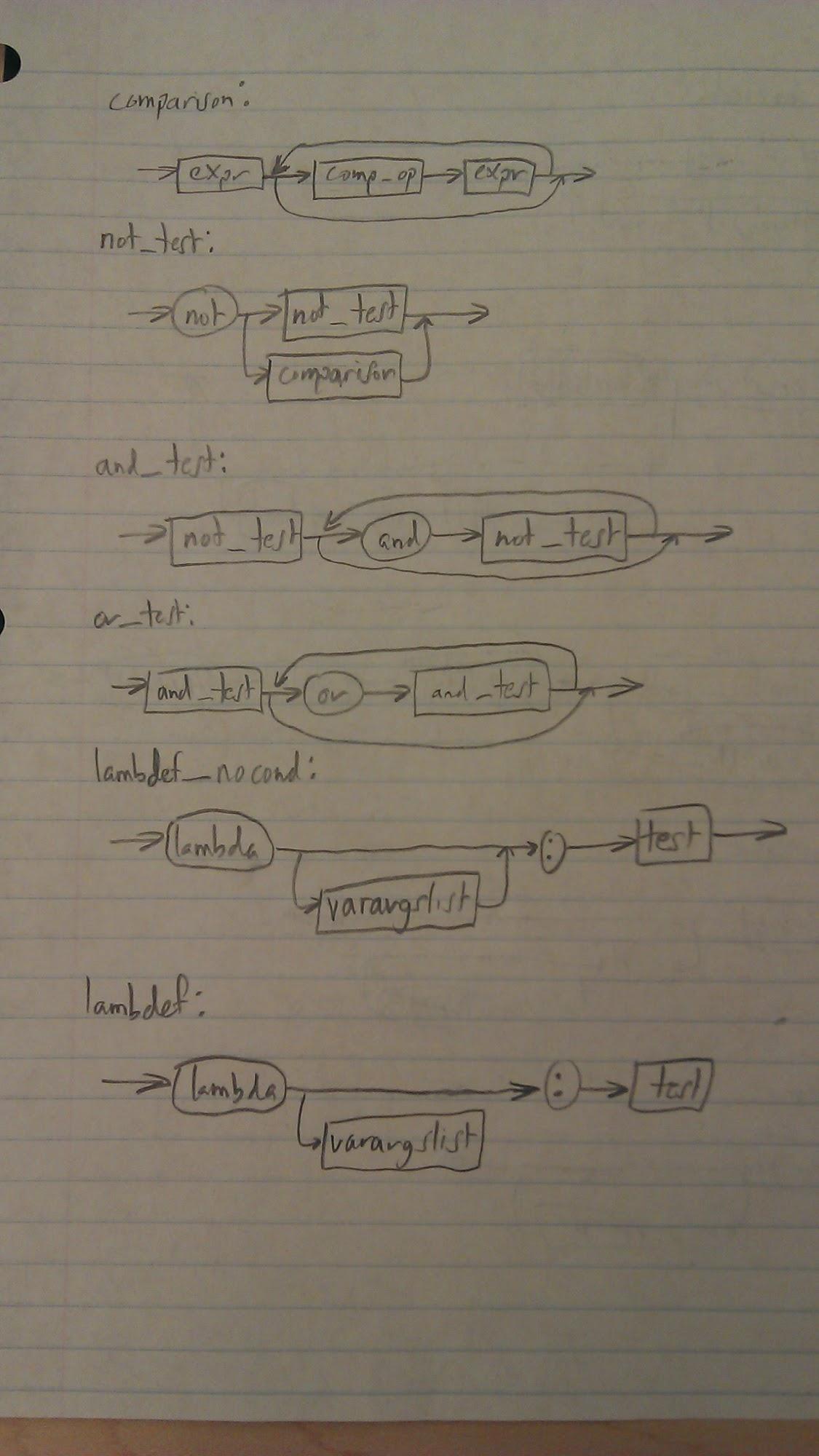
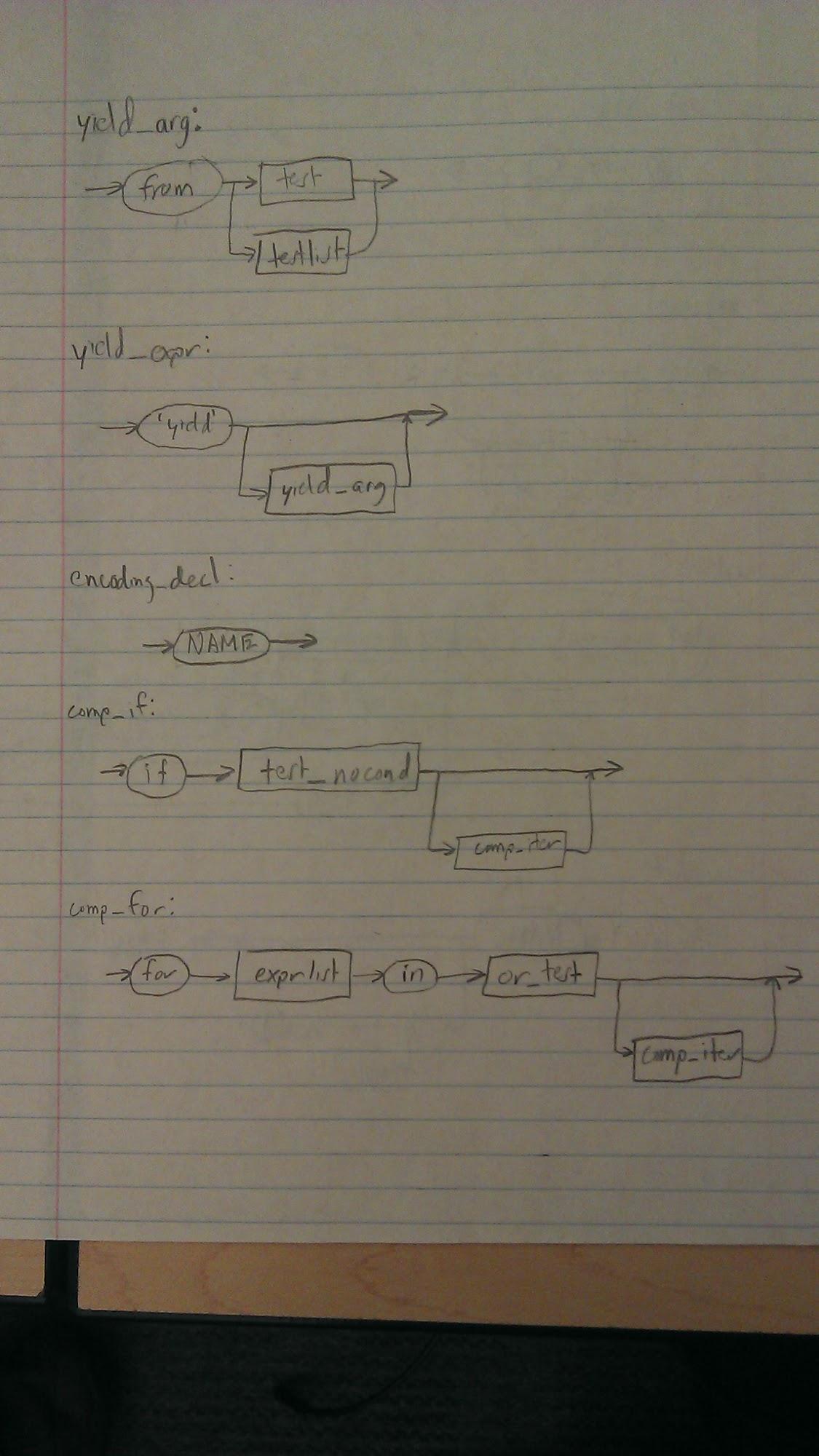
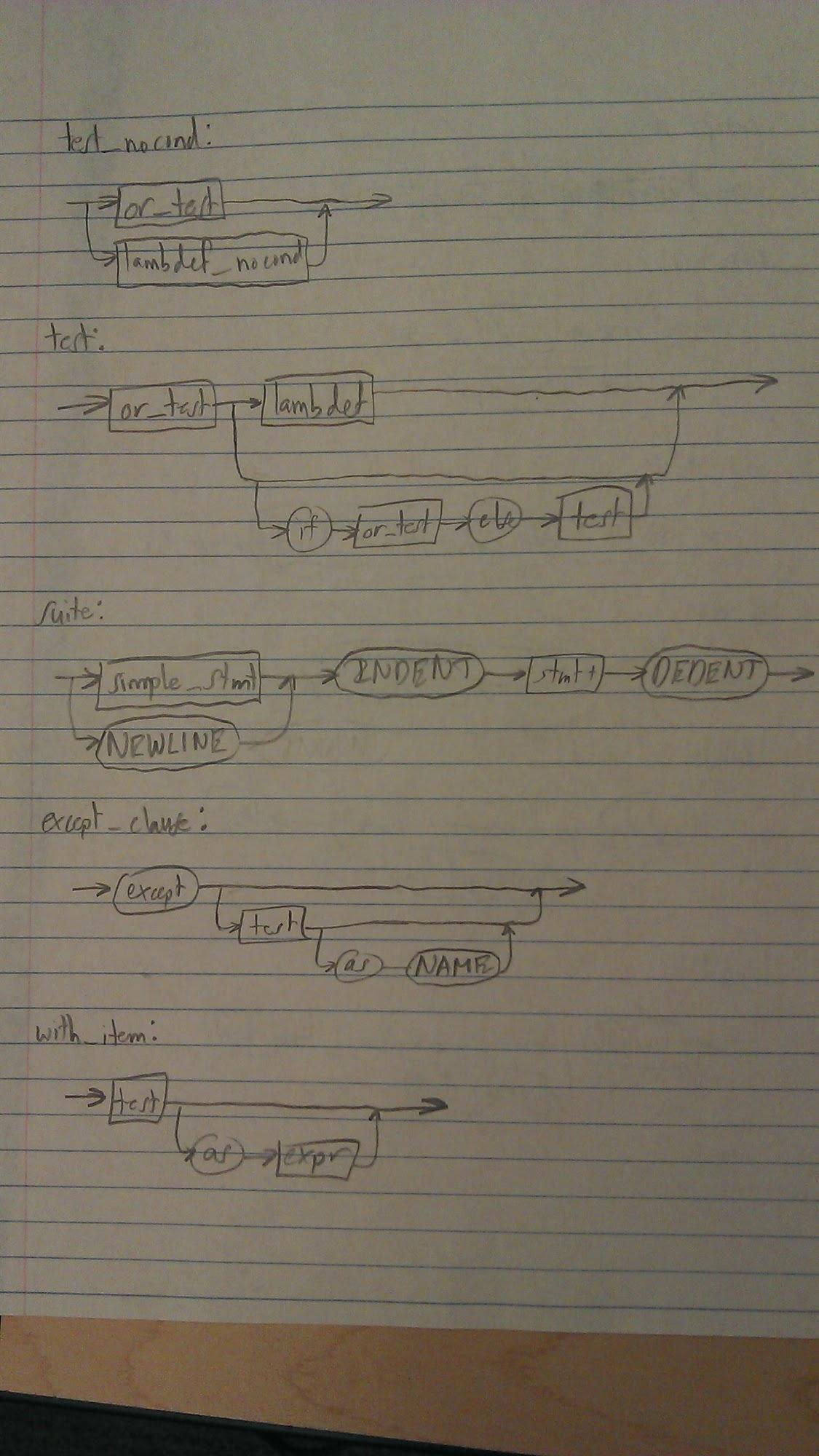
|  |
| --- |
|  |

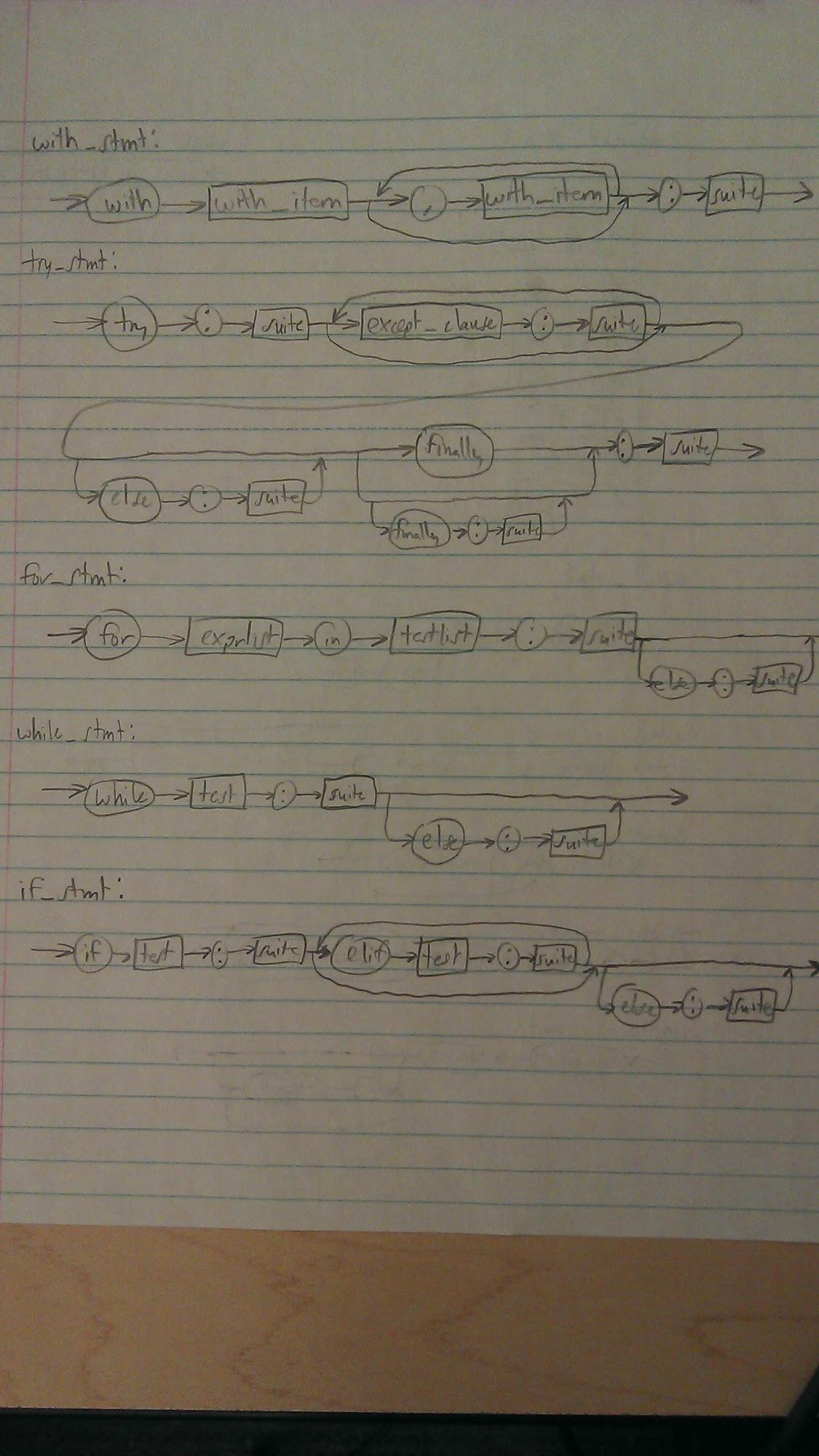
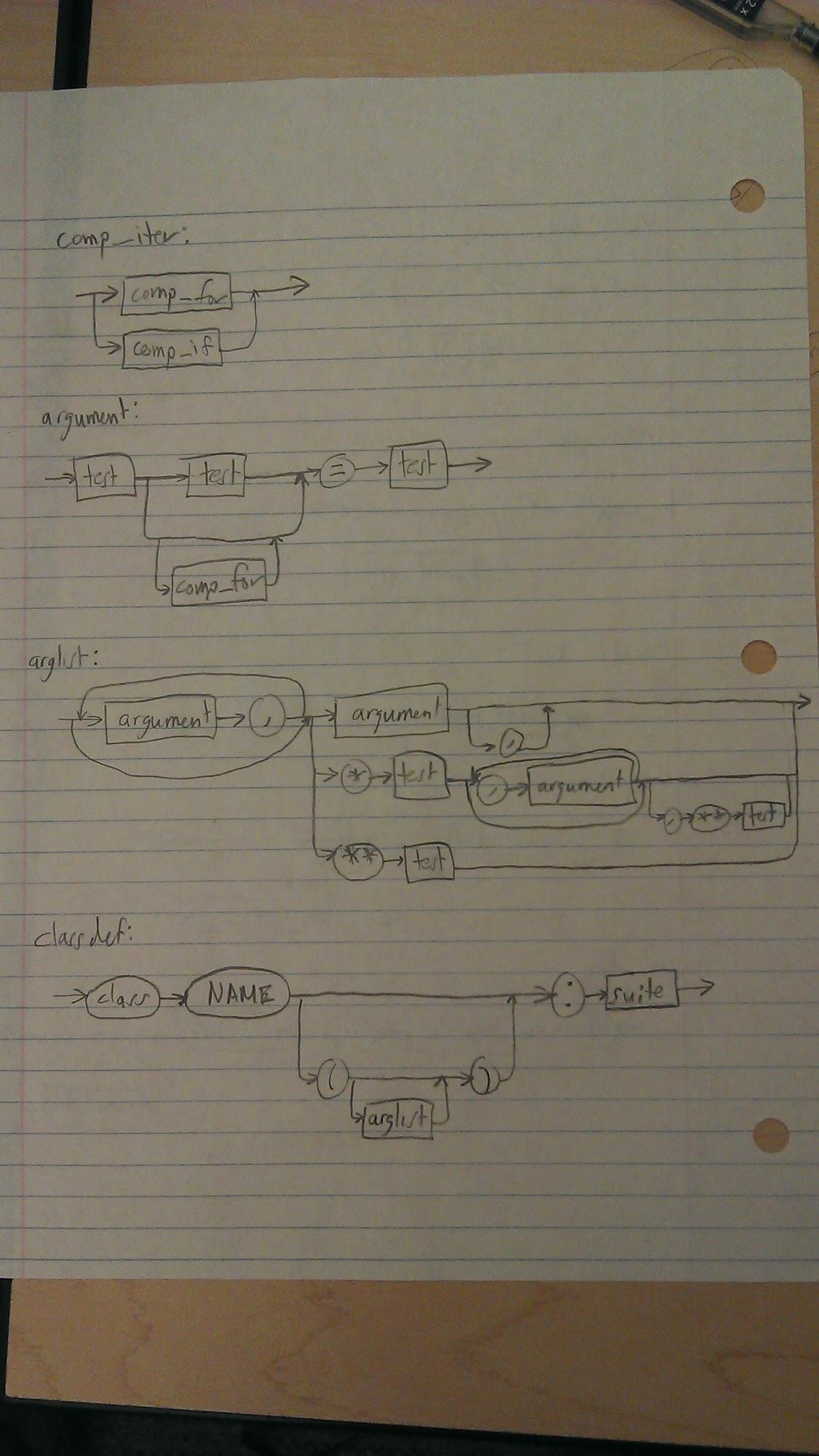
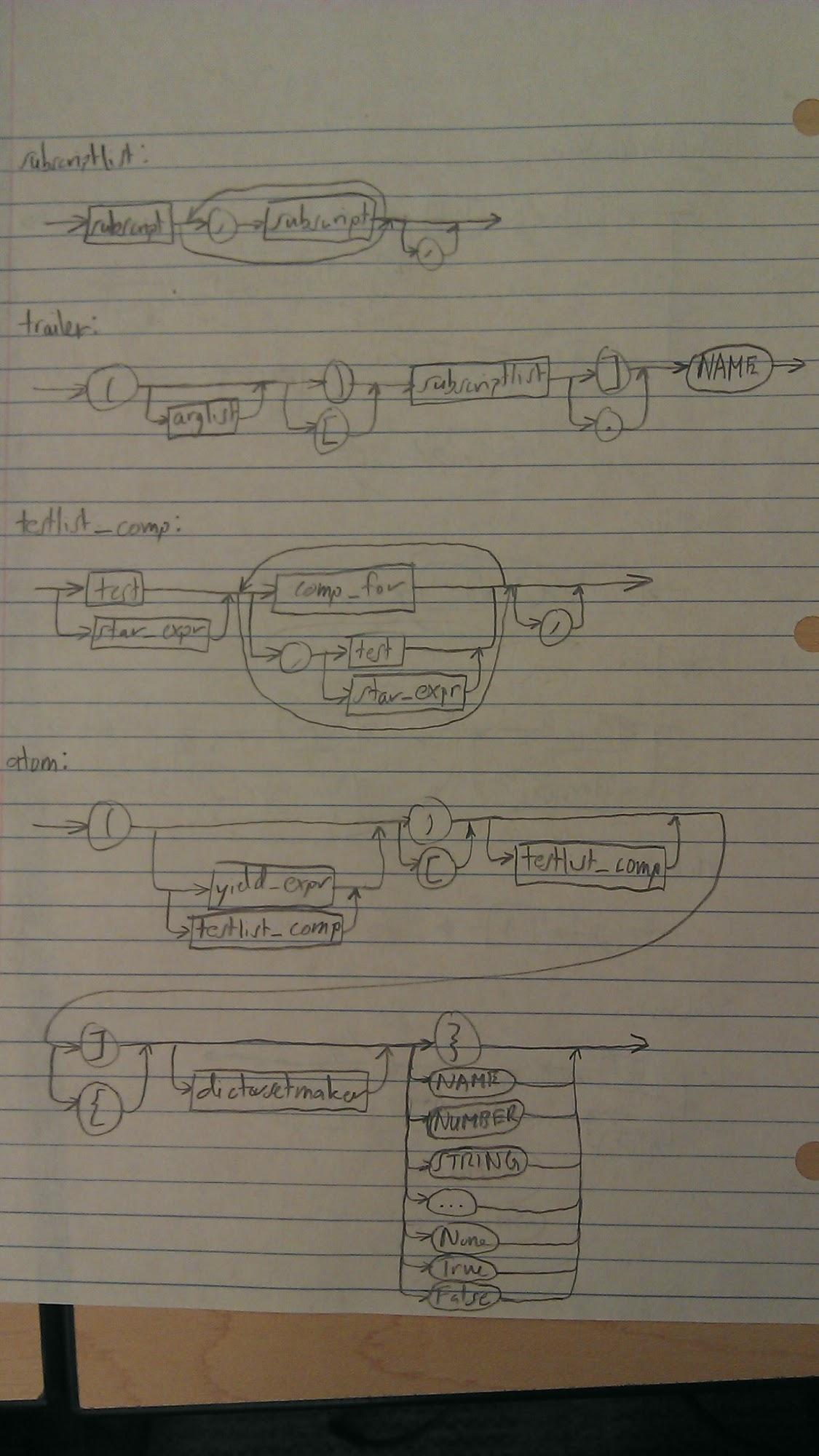
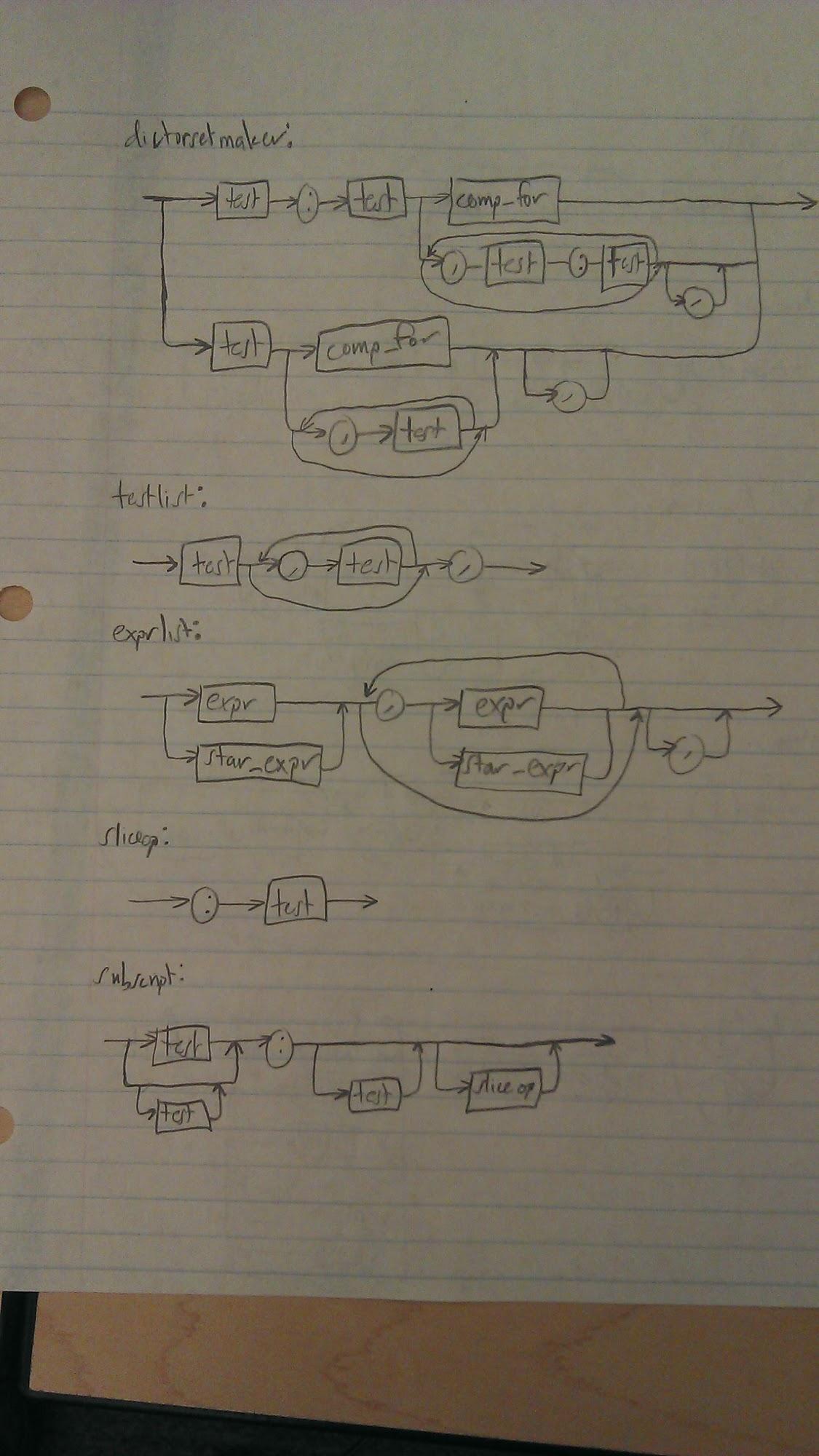
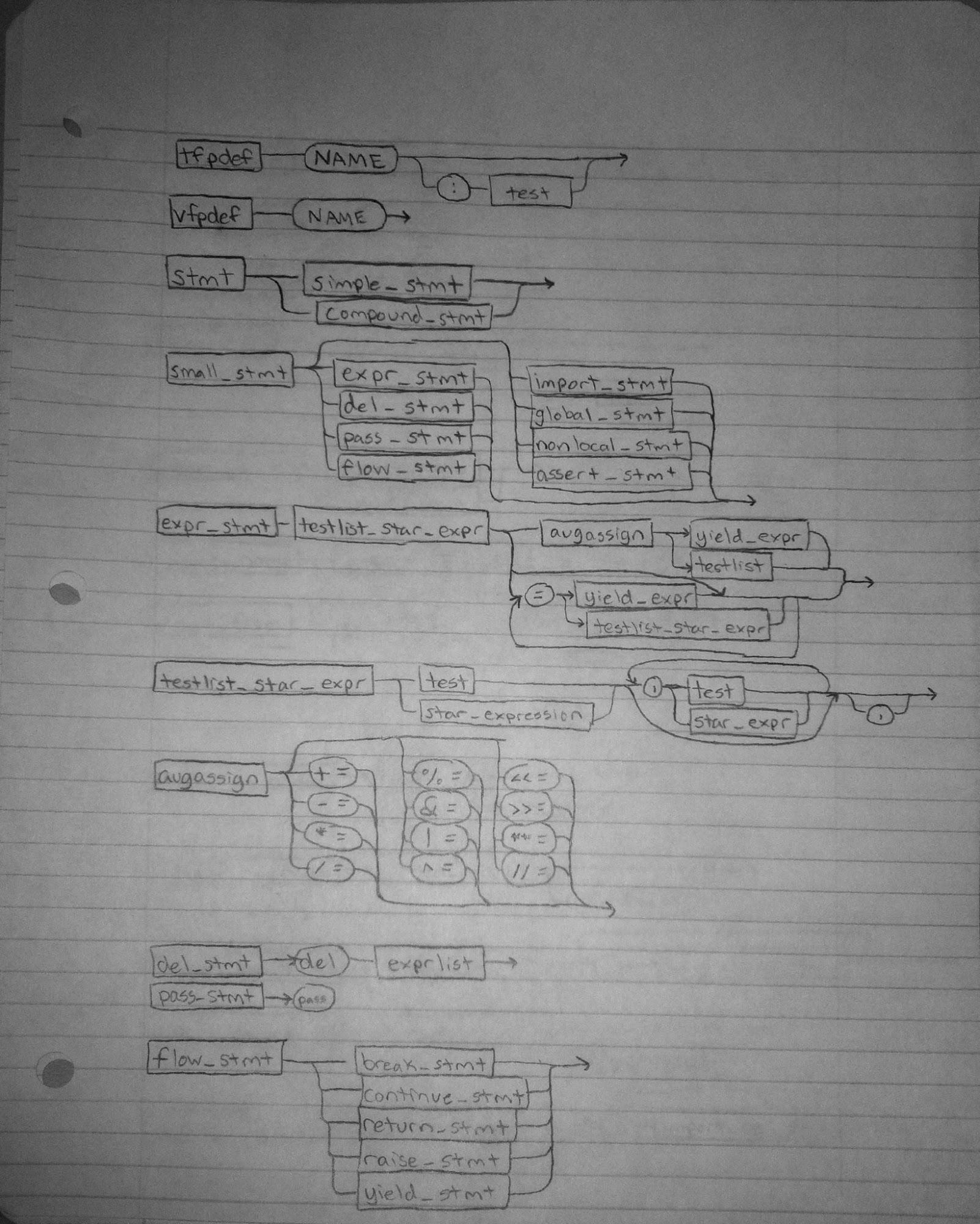
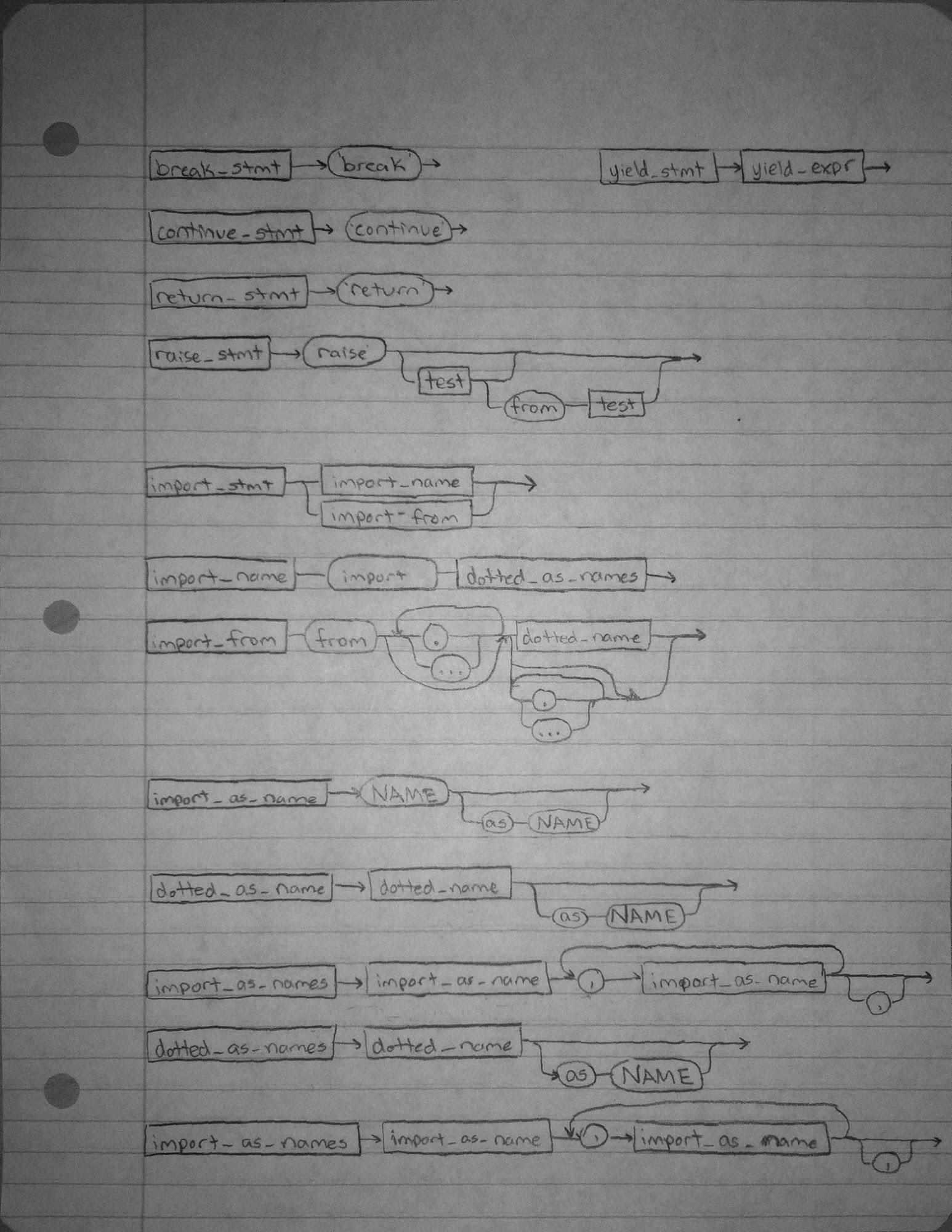
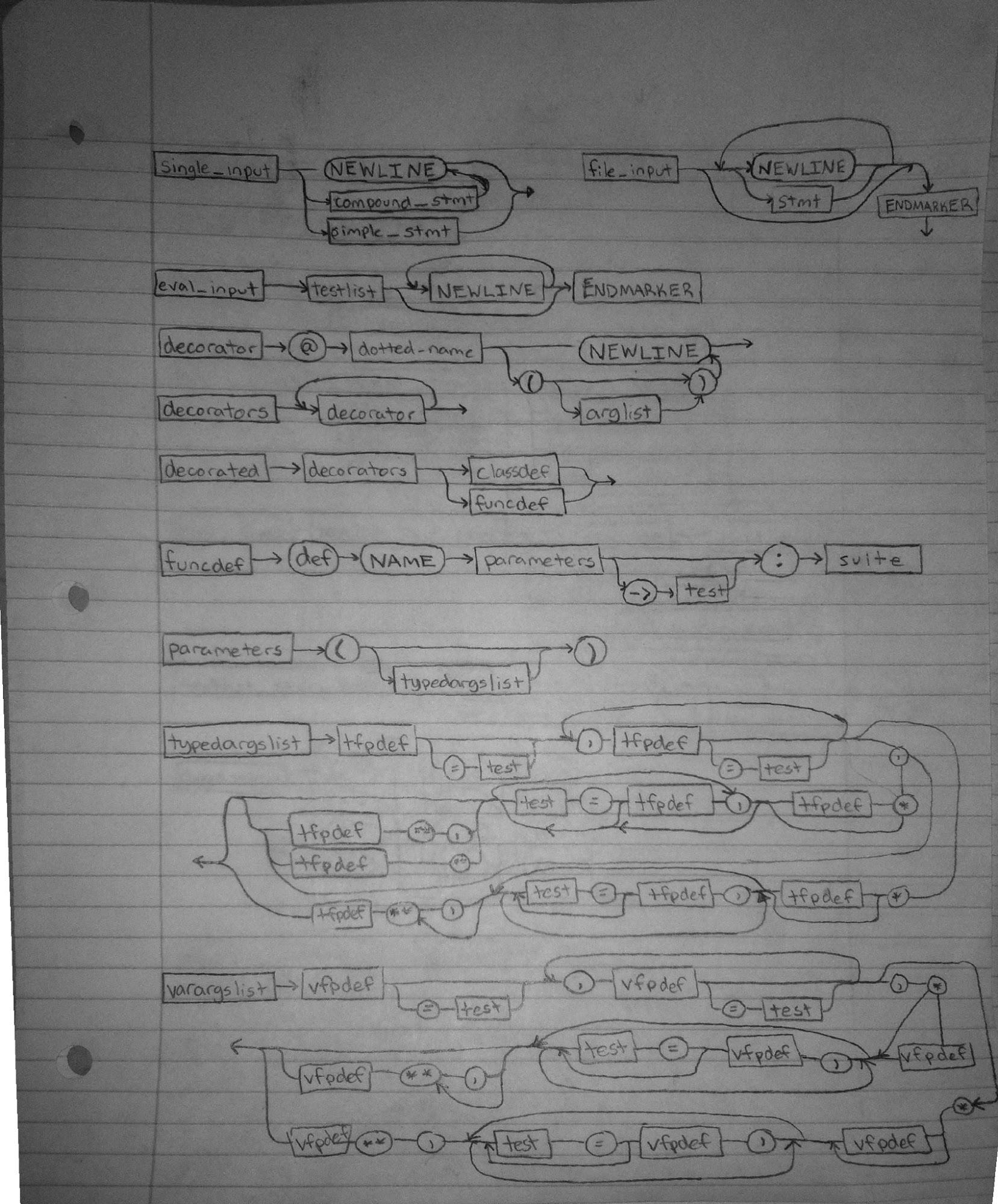
## 4.4 Benefits of Type Inference

* Understanding code can be hard without type inference since it is often difficult to understand what type of object a particular variable might hold at any given time (<http://dj1.willowmail.com/~jeske/Projects/PyLint/>)
* Types can be used to check that illegal operations do not occur between disparate values. Type information can also be used to improve performance, such as identifying integers and performing integral math at the assembly level. (B. Cannon, 2005)
* Type inference allows one to have the beneﬁts of type information at compile-time without the programming overhead of explicitly specifying types.
* Type inference can allow for early error detection, making code optimization and certification easier (<http://www.dcc.fc.up.pt/~nam/publica/artigoDYLA.pdf>)

Overall, it is very difficult to implement an accurate and robust type inference for Python and requires a high level of sophistication if the language is to remain unchanged. Many believe that the benefits of type inference is not worth the difficulty and compromises the flexibility of Python as a programming language. (M. Saleb, 2004)

Appendix





References

[B. Cannon, “ Localized Type Inference of Atomic Types in Python,” M.S. Thesis, Dept. Comp. Sci., California Polytechnic State Univ., San Luis Obispo, CA, 2005.](http://docs.python.org/2/tutorial/index.html)

E. Maia, N. Moreira and R. Reis, “A Static Type Inference for Python,” Universidade du Porto Portugal, n.d.

G. van Rossum (2009, January). *The History of Python* [Online]. Available: <http://python-history.blogspot.ca>

G. van Rossum. (2006, October 9). *Python: Design and Implementation* [Online]. Available: <http://www.stanford.edu/class/cs242/slides/2006/python-vanRossum.pdf>

JetBrains (2013, February 13). *Dynamic RunTime Interface in PyCharm 2.7* [Online]. Available:<http://blog.jetbrains.com/pycharm/2013/02/dynamic-runtime-type-inference-in-pycharm-2-7/>

J. Aycock, “Aggressive Type Inference,” Dept. Comp. Sci., University of Victoria, Victoria, BC, 1999.

L. Tratt. (2009, March 13). *Dynamically Typed Languages* [Online]. Available: <http://tratt.net/laurie/research/pubs/html/tratt__dynamically_typed_languages/>

M. Salib, “Starkiller: A Static Type Inferencer and Compiler for Python,” M.S. Thesis, Dept. Comp. Sci. and Dept. Elect. Eng., Massachusetts Institute of Technology., 2004.

Python Software Foundation. (2013, October 2). *The Python Tutorial* [Online]. Available: <http://docs.python.org/2/tutorial/index.html>

Python v2.7.5 Documentation. (2013, Oct 2). *Why was Python created in First Place?*[Online].Available: <http://docs.python.org/2.7/faq/general.html>

Python v2.7.5 Documentation. (2013, Oct 2). *Alternative Implementations*[Online]. Available:

<http://docs.python.org/2/reference/introduction.html#alternate-implementations>

SourceForge.Net (2010, July 16). *Psyco* [Online]. Available:<http://psyco.sourceforge.net/>

S. Hassan and D. Jeske. *Pylint* [Online]. Available:<http://dj1.willowmail.com/~jeske/Projects/PyLint/>