UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 164 **P. N. Hilfinger**
Spring 2005

Project #1: Lexer and Parser for Pyth

Due: Monday, 2 March 2005

This first project calls for writing a lexical analyzer and parser for our Python dialect (Pyth) that produces abstract syntax trees (ASTs). The semantic actions of your parser will build up these trees, and the main program will print them in a Lisp-like format. So that you can see whether you're getting things right, we will supply a sample back-end that accepts these ASTs and prints out corresponding Python translations. The resulting programs will not, of course, be identical to the input, but should have the same effects. Part of your job is to test systematically that this is so.

When the directory `~cs164/hw/proj1` becomes readable, you can copy the files we have placed there to help with this project. We will provide here some classes for creating, reading, and printing abstract syntax trees for the Pyth dialect. Alternatively (although we certainly do not recommend this) you can throw them away and work from scratch. Your task is to fulfill a specification, not to do it in any particular way.

The homework page will contain pointers to current descriptions of the Pyth subset you are to implement. We'll stick to electronic form, at least for now, since the description is subject to change if we discover things that we decide are more trouble for you to implement than they're worth.

Abstract Syntax Trees

Also look at the homework page for links to the AST description. For this assignment, it is only necessary for you to produce the external (printed) form of the AST. Nevertheless, we'll provide a convenient library for building these trees internally, which you'll probably want to use.

We have considerably simplified the description of the AST by relying on an interesting feature of the Python language: most operators correspond to special methods. For example, the `+` operator corresponds to the method `__add__`, so that `3+x` is equivalent to `(3).__add__(x)`. This eliminates the need for individual node types for all the operators; we can make almost everything a method call with the right identifier for the method name.

1 Output and Testing

The output of the program is a textual representation of the AST. This design is not optimal for a real compiler, since there is considerable overhead involved in converting back and forth between the internal tree form and text. For our purposes, however, it is useful that the output of our compiler modules has this representation- independent textual form. It frees us from dependence on a particular implementation language, for one thing, and minimizes interactions between compiler phases (particularly useful when we plug our backend modules into your front ends).

Naturally, this raises the question of how you check that your compiler's output is correct. It's all very well to decide that you are going to test your results, but just what does this mean? Presumably, you must start with a body of test cases: Pyth programs that systematically exercise all constructs of the language and all parts of your compiler. You'll need a convenient way to run all your tests (your *test suite*) against the latest version of your compiler, and to mechanically check the result. One possibility is to look at the actual ASTs produced and make sure they are "right," which implies that you figure out by hand what your test cases are supposed to produce. This can be rather tedious and also has the problem that there may be alternative equivalent ASTs for any given program, and your compiler is not necessarily wrong for picking something different from your canonical solution.

Therefore, we're looking for a slightly different approach. We will provide a back end that "compiles" your ASTs into standard Python, suitable for execution with a Python interpreter. By making each of your test cases "self-testing"—by having them be little programs each of which prints something—you can compare the outputs of these programs with what they are supposed to produce.

Testing is important and, quite frankly, we don't have you do enough of it. To try to change that, 6 points of your project grade will come from the quality and completeness of your test cases.

2 What to turn in

You will be turning in three things:

- Source files (in flex, jflex, bison, jbison, C++, or Java).

- A testing subdirectory containing Pyth source files and corresponding files with the correct output.
- A Makefile that provides (at least) two targets:
  - The default target (built with a plain `gmake` command) should compile your program, producing an executable program called `pythc` (we will provide instructions for how to accomplish this in Java, so that you don't need the `java` command to run your program.
  - The command `gmake check` should run all your tests against your compiler and check the results.

3 Assorted Advice

First, get started as soon as possible. Second, don't *ever* waste time beating your head against a wall. If you come to an impasse, recognize it quickly and come see one of us or, if we are not immediately available, work on something else for a while (you can never have enough test cases, for example). Third, keep track of your partner. If possible, schedule time to do most of your work together. I've seen all too many instances of the Case of the Flaky Partner.

Learn your tools. You should be doing all of your compilations using `gmake`. Get to know this tool and try to understand the "makefiles" we give you. It really does make life much easier for you. Learn to use it from within Emacs. Learn to use the `gdb` and gjdb debuggers (also usable from within Emacs). In most cases, if your C++ program blows up, you should be able to at least tell me *where*, it blew up (even if the error that caused it is elsewhere). I do not look kindly on those who do not at least make that effort before consulting me. Use CVS or PRCS to store a history of versions of your program and to share development with your partner (we'll have a session or two on this subject). If you are using some other IDE at home (like Eclipse), make sure you learn how to use the analogous features in it (if it doesn't have such features, it is not a proper IDE; find another).