322 Compilers: Assignment 1b
A Packrat Tiger Parser

For this assignment, you must:

- Build infrastructure for a Packrat parser (including memoization) as we did in class.
- Design a packrat parser for the Tiger language, with the exceptions in assignment 1a.
- Design a test case harness that consumes a file name *file*`.tig` as input. If the input is well-formed, it should print out the matching expression tree (from the left-hand side of assignment 1a); if the input is not well-formed, it should print out the string `#illegal`. Nothing else should be printed out (ie, debugging output will be considered a failed test case).
- Produce a parser that runs under linux on one of the machines in the TLAB. (If this is a hardship for you, please get in touch with me and we'll see if we can work something out.)
- Hand in a `.zip` file containing two directories: `1a` with your current set of test cases (the ones from the previous assignment if you did not add to them or change them), and `1b` with your parser implementation. It should contain a script or executable file named `parse` that runs your test harness.

Recommendations:

- Test the parser combinators carefully before starting the Tiger parser itself.
- Build the parser incrementally, testing carefully after each addition to the language (and save all your test cases!).
- Avoid adding too many primitive parsers, intead use helper functions to set up common parsing patterns. You will need at least
  - knots,
  - atoms (consider generalizing this one to support character ranges, not just characters),
  - a parser that accepts any character (but there must be at least one character),
  - alternation,
  - sequencing,
  - negation, and
  - a parser that transforms the results of another parser (but doesn't change the language it accepts).

  Build up other common parsing patters as helper functions using these basic parsers.
- The parser's result should be a tree that matches the structure of the trees in the previous assignment.

  Use whatever implementation of trees you'd like, presumably one that matches your programming language well. If you are using a class-based OO language, you probably want one interface (or abstract class) for each non-terminal in that grammar, one concrete class for each production, and one field for each piece of each production.

  For example, you would have a `dec` interface with three classes that implement it, `vardec`, `varwithtypedec`, and `typedec`. The `vardec` class would have two fields, a `id` and an `exp`.

  The only method that these class would have is `print` that would simply print out the expression tree (i.e., for `vardec`, it would print an open paren, print `var`, print a space, print the identifier (via the identifier class's `print` method), print a space, print the initialization expression (as before via the `print` method, but this time the expression's), and then print the final close paren.