Most modern programming languages have some notion of scope.
Scope defines the “lifetime” of a program symbol.
If a symbol is no longer accessible then we say that it is “out of scope.”
The simplest scope is the “block scope.”
With scope we need a notion of variable declaration which allows us to assert in which scope the variable is visible or accessible.
We extend our Cuppa1 language with variable declarations of the form

```declare x = 10;
```

- Declares the variable `x` in the current scope and initializes it to the value 10
- If the current scope is the global (outermost) scope then we call `x` a “global” variable.
Cuppa2 Grammar

cuppa2_gram.py

program : stmt_list

stmt_list : stmt stmt_list
           | empty

stmt : DECLARE ID opt_init opt_semi
     | ID '=' exp opt_semi
     | GET ID opt_semi
     | PUT exp opt_semi
     | WHILE '( ' exp ' ' )' stmt
     | IF '( ' exp ' ' )' stmt opt_else
     | '{' stmt_list '}'

opt_init : '=' exp
        | empty

opt_else : ELSE stmt
        | empty

opt_semi : ';'
        | empty

exp : exp PLUS exp
    | exp MINUS exp
    | exp TIMES exp
    | exp DIVIDE exp
    | exp EQ exp
    | exp LE exp
    | INTEGER
    | ID
    | '( ' exp ' ' )'
    | MINUS exp %prec UMINUS
    | NOT exp
```python
def p_stmt(p):
    ...
    stmt : DECLARE ID opt_init opt_semi
        | ID '=' exp opt_semi
        | GET ID opt_semi
        | PUT exp opt_semi
        | WHILE '(' exp ')' stmt
        | IF '(' exp ')' stmt opt_else
        | '{' stmt_list '}'
    ...
    if p[1] == 'declare':
        p[0] = ('declare', p[2], p[3])
    elif p[2] == '=':
        p[0] = ('assign', p[1], p[3])
    elif p[1] == 'get':
        p[0] = ('get', p[2])
    elif p[1] == 'put':
        p[0] = ('put', p[2])
    elif p[1] == 'while':
        p[0] = ('while', p[3], p[5])
    elif p[1] == 'if':
        p[0] = ('if', p[3], p[5], p[6])
    elif p[1] == '{':
        p[0] = ('block', p[2])
    else:
        raise ValueError("unexpected symbol {}".format(p[1]))
```
We can now write properly scoped programs

Consider:

```plaintext
declare x = 1;
{
    declare x = 2;
    put x;
}
{
    declare x = 3;
    put x;
}
put x;
```
Variable Shadowing

- An issue with scoped declarations is that inner declarations can “overshadow” outer declarations.
- Consider:

```plaintext
declare x = 2;
{
    declare x = 3;
    {
        declare y = x + 2;
        put y;
    }
}
```

What is the output of the program once it is run?
Variable update

- A variable update can be outside of our current scope.
- Consider

```
declare x = 2;
{
    declare y = 3;
    x = y + x;
    put x;
}
put x;
```
Symbol Tables

- To deal with programs like that we need something more sophisticated for variable lookup than a dictionary.
- *a dictionary stack*

- This stack needs to be able to support the following functionality
  - Declare a variable (insertion)
  - Lookup a variable
  - Update a variable value
Semantic Rules for Variable Declarations

- Here are the rules which we informally used in the previous examples:
  - The ‘declare’ statement inserts a variable declaration into the current scope
    - a variable lookup returns a variable value from the current scope or the surrounding scopes
  - Every variable needs to be declared before use
  - No variable can be declared more than once in the current scope.
Symbol Tables

- **Design:**
  - we have a class `SymTab` that:
    - Holds a stack of scopes
      - `scoped_symtab`
    - Defines the interface to the symbol table
      - `push_scope`, `pop_scope`, `declare_sym`, etc
  - By default, `SymTab` is initialized with a single scope on the stack – *the global scope*. 
Symbol Tables

Symbol Table

Global Scope

Current Scope Pointer

Local Scope

declare x = 2;
{
    declare y = 3;
    x = y + x;
    put x;
}
put x;
Symbol Tables

Global Scope

Local Scope

Current Scope Pointer

declare x;
get x;
If (0 <= x)
{
    declare i = x;
    put i;
}
else
{
    declare j = -1 * x;
    put j;
}
put x;
Symbol Tables

Symbol Table

Global Scope

Current Scope Pointer

Local Scope

Local Scope

declare x = 2;
{
    declare x = 3;
    {
        declare y = x + 2;
        put y;
    }
}
CURR_SCOPE = 0

class SymTab:

    #------
    def __init__(self):
        # global scope dictionary must always be present
        self.scoped_symtab = [[]]

    #------
    def push_scope(self):
        # push a new dictionary onto the stack - stack grows to the left
        self.scoped_symtab.insert(CURR_SCOPE, {})

    #------
    def pop_scope(self):
        # pop the left most dictionary off the stack
        if len(self.scoped_symtab) == 1:
            raise ValueError("cannot pop the global scope")
        else:
            self.scoped_symtab.pop(CURR_SCOPE)

    #------
    def declare_sym(self, sym, init):
        # declare the symbol in the current scope: dict @ position 0
        ...

    #------
    def lookup_sym(self, sym):
        # find the first occurrence of sym in the symtab stack
        # and return the associated value
        ...

    #------
    def update_sym(self, sym, val):
        # find the first occurrence of sym in the symtab stack
        # and update the associated value
        ...
def declare_sym(self, sym, init):
    # declare the symbol in the current scope: dict @ position 0
    # first we need to check whether the symbol was already declared
    # at this scope
    if sym in self.scoped_symtab[CURR_SCOPE]:
        raise ValueError("symbol {} already declared".format(sym))

    # enter the symbol in the current scope
    scope_dict = self.scoped_symtab[CURR_SCOPE]
    scope_dict[sym] = init

def lookup_sym(self, sym):
    # find the first occurrence of sym in the symtab stack
    # and return the associated value

    n_scopes = len(self.scoped_symtab)

    for scope in range(n_scopes):
        if sym in self.scoped_symtab[scope]:
            val = self.scoped_symtab[scope].get(sym)
            return val

    # not found
    raise ValueError("{} was not declared".format(sym))
def update_sym(self, sym, val):
    # find the first occurrence of sym in the symtab stack
    # and update the associated value

    n_scopes = len(self.scoped_symtab)

    for scope in range(n_scopes):
        if sym in self.scoped_symtab[scope]:
            scope_dict = self.scoped_symtab[scope]
            scope_dict[sym] = val
            return

    # not found
    raise ValueError("{} was not declared").format(sym))
Interpret Walker

Note: Same as Cuppa1 interpreter except for the addition of the declaration statement and additional functionality in block statements and variable expressions.
Interpret Walker

That's it – everything else is the same as the Cuppa1 interpreter!
Grammars allow us to construct parsers that recognize the syntactic structure of languages.

Any program that does not conform to the structure prescribed by the grammar is rejected by the parser.

We call those errors “syntactic errors.”
Syntactic vs Semantic Errors

- Semantic errors are errors in the behavior of the program and cannot be detected by the parser.
- **Programs with semantic errors are usually syntactically correct**

- A certain class of these semantic errors can be caught by the interpreter/compiler. Consider:
  ```
  declare x = 10;
  put x + 1;
  declare x = 20;
  put x + 2;
  ```

- Here we are redeclaring the variable ‘x’ which is not legal in many programming languages.

- Many other semantic errors cannot be detected by the interpreter/compiler and show up as “bugs” in the program.
Symbol Tables

Symbol Table

Global Scope

Current Scope Pointer

```
declare x = 10;
put x + 1;
declare x = 20;
put x + 2;
```
Symbol Tables

Symbol Table

Global Scope

Current Scope Pointer

x = x + 1;
put x;