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.

Read Chap 7
Cuppa2

- We extend our Cuppa1 language with variable declarations of the form

  \[
  \text{declare } x = 10;
  \]

- 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

Listing 7.1: Grammar for the Cuppa2 language.

```
stmt_list : (stmt)*
stmt : declare ID (= exp)? ;?
    | ID = exp ;?
    | get ID ;?
    | put exp ;?
    | while \( \exp \) stmt
    | if \( \exp \) stmt (else stmt)?
    | \{ stmt_list \}
exp : exp_low
exp_low : exp_med (== | =<) exp_med)*
exp_med : exp_high (+ | -) exp_high)*
exp_high : primary (* | /) primary)*
primary : INTEGER
    | ID
    | \( \exp \)
    | - primary
    | not primary
ID : <any valid variable name>
INTEGER : <any valid integer number>
```

Notice that the initializer for the declaration is optional.
def stmt(stream):
    token = stream.pointer()
    if token.type in ['DECLARE']:
        stream.match('DECLARE')
        id_tk = stream.match('ID')
        if stream.pointer().type in ['ASSIGN']:
            stream.match('ASSIGN')
            e = exp(stream)
        else:
            # if no initializer assume default value
            e = ('INTEGER', 0)
    if stream.pointer().type in ['SEMI']:
        stream.match('SEMI')
    return ('DECLARE', ('ID', id_tk.value), e)
elif token.type in ['ID']:
    ...
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

```plaintext
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

Global Scope

Current Scope Pointer

Local Scope

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

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

Local Scope

Current Scope Pointer

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))
Symbol Tables

```python
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.

cuppa2_interp_walk.py
Interpret Walker

That's it – everything else is the same as the Cuppa1 interpreter!
Syntactic vs Semantic Errors

- 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

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;