P0 Symbol Table

Emil Sekerinski, McMaster University, revised February 2022

Declarations of the source program are entered into the symbol table as the source program is parsed. The symbol detects multiple definitions or missing definitions and reports those by calling procedure `mark(msg)` of the scanner.

- classes `Var`, `Ref`, `Const`, `Type`, `Proc`, `StdProc` are for the symbol table entires
- classes `Int`, `Bool`, `Record`, `Array`, `Set` are for the types of symbol table entries
- procedures `Init()`, `newDecl(name, entry)`, `find(name)`, `openScope()`, `topScope()`, `closeScope()` are the operations of the symbol table
- procedure `printSymTab()` visualizes the symbol table in a readable textual form with indentation.

In [ ]:
```
import nbimporter, textwrap
nbimporter.options["only_defs"] = False
from SC import mark

def indent(n):
    return textwrap.indent(str(n), '  ')
```

Symbol table entries are objects of following classes:

- `Var` for global variables, local variables, and value parameters (must be `Int` or `Bool`)
- `Ref` for reference parameters (of any type)
- `Const` for constants of types `Int` or `Bool`
- `Type` for named or anonymous types
- `Proc` for declared procedures
- `StdProc` for one of `write`, `writeln`, `read`

All entries have a field `tp` for the type, which can be `None`.

In [ ]:
```
class Var:
    def __init__(self, tp):
        self.tp = tp
    def __str__(self):
        return 'Var(name = ' + str(getattr(self, 'name', '')) + ', lev = ' + \
               str(getattr(self, 'lev', '')) + ', tp = ' + str(self.tp) + ')'
    def __eq__(self, other):
        return self.reg == other.reg and self.adr == other.adr
    def __hash__(self):
        return hash(str(self))

class Ref:
    def __init__(self, tp):
        self.tp = tp
    def __str__(self):
        return 'Ref(name = ' + str(getattr(self, 'name', '')) + ', lev = ' + \
               str(getattr(self, 'lev', '')) + ', tp = ' + str(self.tp) + ')'

class Res:
    def __init__(self, tp):
        self.tp = tp
    def __str__(self):
        return 'Res(name = ' + str(getattr(self, 'name', '')) + ', lev = ' + \
               str(getattr(self, 'lev', '')) + ', tp = ' + str(self.tp) + ')'

class Const:
    def __init__(self, tp, val):
        self.tp, self.val = tp, val
    def __str__(self):
        return 'Const(name = ' + str(getattr(self, 'name', '')) + ', tp = ' + \
               str(self.tp) + ', val = ' + str(self.val) + ')'
    def __eq__(self, other):
        return self.val == other.val
    def __hash__(self):
        return hash(str(self))

class Type:
    def __init__(self, tp):
        self.tp, self.val = None, tp
    def __str__(self):
        return 'Type(name = ' + str(getattr(self, 'name', '')) + ', val = ' + \
               str(self.val) + ')'

class Proc:
    def __init__(self, par, res):
```

```
        self.tp, self.par, self.res = None, par, res
    def __str__(self):
        return 'Proc(name = ' + self.name + ', lev = ' + str(self.lev) + \
               ', par = [' + ', '.join(str(s) for s in self.par) + ']' + \
               ', res = [' + ', '.join(str(s) for s in self.res) + '])'

class StdProc:
    def __init__(self, par, res):
        self.tp, self.par, self.res = None, par, res
    def __str__(self):
        return 'StdProc(name = ' + self.name + ', lev = ' + str(self.lev) + \
               ', par = [' + ', '.join(str(s) for s in self.par) + ']' + \
               ', res = [' + ', '.join(str(s) for s in self.res) + '])'
```

- The P0 types `integer` and `boolean` are represented by the classes `Int` and `Bool`; no objects of `Int` or `Bool` are created
- Record, array, and set types in P0 are represented by objects of class `Record`, `Array`, `Set`; for records, a list of fields is kept, for arrays, the base type, the lower bound, and the length of the array is kept, for sets, the lower bound and the length (in bits) is kept.

In [ ]:
```
class Int: pass

class Bool: pass

class Record:
    def __init__(self, fields):
        self.fields = fields
    def __str__(self):
        return 'Record(fields = [' + ', '.join(str(f) for f in self.fields) + '])'

class Array:
    def __init__(self, base, lower, length):
        self.base, self.lower, self.length = base, lower, length
    def __str__(self):
        return 'Array(lower = ' + str(self.lower) + ', length = ' + \
               str(self.length) + ', base = ' + str(self.base) + ')'

class Set:
    def __init__(self, lower, length):
        self.lower, self.length = lower, length
    def __str__(self):
        return 'Set(lower = ' + str(self.lower) + ', length = ' + \
               str(self.length) + ')'
```

The symbol table is represented by a list of scopes. Each scope is a list of entries. Each entry has a name, which is assumed to be a string, and the level at which it is declared; the entries on the outermost scope are on level 0 and the level increases with each inner scope.

In [ ]:
```
def init():
    global symTab
    symTab = [[]]

def symTabStr():
    return [[str(e) for e in l] for l in symTab]

def newDecl(name, entry):
    top, entry.lev, entry.name = symTab[0], len(symTab) - 1, name
    for e in top:
        if e.name == name:
            mark("multiple definition of " + str(name)); return
    top.append(entry)

def find(name):
    for l in symTab:
        for e in l:
            if name == e.name: return e
    mark('undefined identifier ' + name)
    return Const(None, 0)

def openScope():
    symTab.insert(0, [])

def topScope():
    return symTab[0]

def closeScope():
    symTab.pop(0)
```