# ST

March 12, 2018

## 0.1 P0 Symbol Table

**Original Author: Emil Sekerinski, February 2017** 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` 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 [2]: import nbimporter, textwrap
        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 [2]: class Var:
            def __init__(self, tp):
                self.tp = tp
            def __str__(self):
                return 'var ' + str(getattr(self, 'name', '')) + ' lev ' + \
                       str(getattr(self, 'lev', '')) + ':\n' + indent(self.tp)

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

        class Const:
```

```
        def __init__(self, tp, val):
            self.tp, self.val = tp, val
        def __str__(self):
            return 'const ' + str(getattr(self, 'name', '')) + ': ' + \
                   str(self.tp) + ' = ' + str(self.val)

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

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

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

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

```
In [3]: class Int: pass

    class Bool: pass

    class Enum: pass # for adding enumeration types

    class Record:
        def __init__(self, fields):
            self.fields = fields
        def __str__(self):
            return 'record\n' + \
                   indent('\n'.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\n' + indent(self.base)
```

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 [4]: def init():
            global symTab
            symTab = [[]]

        def printSymTab():
            for l in symTab:
                for e in l: print(e)
                print()

        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"); 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)
```