# P0 Code Generator for AST Only

**Emil Sekerinski, McMaster University, revised February 2022**

The AST has facilities for pretty-printing.

In [ ]:
```
import nbimporter
nbimporter.options["only_defs"] = False
from SC import TIMES, DIV, MOD, PLUS, MINUS, AND, OR, EQ, NE, LT, GT, LE, GE, \
     NOT, CARD, COMPLEMENT, UNION, INTERSECTION, ELEMENT, SUBSET, SUPERSET, \
     mark
from ST import indent, Bool

class UnaryOp:
    def __init__(self, op, arg):
        self.tp, self.op, self.arg = arg.tp, op, arg
    def __str__(self):
        o = '+' if self.op == PLUS else \
            '-' if self.op == MINUS else \
            '¬' if self.op == NOT else \
            '#' if self.op == CARD else \
            'C' if self.op == COMPLEMENT else 'op?'
        return o + '\n' + indent(self.arg)

class BinaryOp:
    def __init__(self, op, tp, left, right):
        self.tp, self.op, self.left, self.right = tp, op, left, right
    def __str__(self):
        o = '×' if self.op == TIMES else \
            'div' if self.op == DIV else \
            'mod' if self.op == MOD else \
            'and' if self.op == AND else \
            '+' if self.op == PLUS else \
            '-' if self.op == MINUS else \
            'or' if self.op == OR else \
            '=' if self.op == EQ else \
            '≠' if self.op == NE else \
            '<' if self.op == LT else \
            '>' if self.op == GT else \
            '≤' if self.op == LE else \
            '≥' if self.op == GE else \
            '∪' if self.op == UNION else \
            '∩' if self.op == INTERSECTION else \
            '∈' if self.op == ELEMENT else \
            '⊆' if self.op == SUBSET else \
            '⊇' if self.op == SUPERSET else 'op?'
        return o + '\n' + indent(self.left) + '\n' + indent(self.right)

class Assignment:
    def __init__(self, left, right):
        self.left, self.right = left, right
    def __str__(self):
        return ':=\n' + indent(self.left) + '\n' + indent(self.right)

class Call:
    def __init__(self, res, ident, param):
        self.res, self.ident, self.param = res, ident, param
    def __str__(self):
        return 'call ' + (str(self.res) if self.res else '') + ' ' + \
               str(self.ident) + ('\n' if len(self.param) > 0 else '') + \
               indent('\n'.join([str(x) for x in self.param]))

class Seq:
    def __init__(self, first, second):
        self.first, self.second = first, second
    def __str__(self):
        return 'seq\n' + indent(self.first) + '\n' + indent(self.second)

class IfThen:
    def __init__(self, cond, th):
        self.cond, self.th = cond, th
    def __str__(self):
        return 'ifthen\n' + indent(self.cond) + '\n' + indent(self.th)

class IfElse:
    def __init__(self, cond, th, el):
        self.cond, self.th, self.el = cond, th, el
    def __str__(self):
        return 'ifelse\n' + indent(self.cond) + '\n' + indent(self.th) + \
               '\n' + indent(self.el)
```

```
class While:
    def __init__(self, cond, bd):
        self.cond, self.bd = cond, bd
    def __str__(self):
        return 'while\n' + indent(self.cond) + '\n' + indent(self.bd)

class ArrayIndexing:
    def __init__(self, arr, ind):
        self.tp, self.arr, self.ind = arr.tp.base, arr, ind
    def __str__(self):
        return str(self.arr) + '[]\n '+ indent(self.ind)

class FieldSelection:
    def __init__(self, rec, fld):
        self.tp, self.rec, self.fld = fld.tp, rec, fld
#        self.tp, self.rec, self.fld = rec.tp.fields[fld].tp, rec, fld
    def __str__(self):
        return str(self.rec) + '.' + str(self.fld)

# public functions

def genBool(b):
    b.size = 4; return b

def genInt(i):
    i.size = 4; return i

def genRec(r):
    """Assuming r is Record, determine fields offsets and the record size"""
    s = 0
    for f in r.fields:
        f.offset, s = s, s + f.tp.size
    r.size = s
    return r

def genArray(a):
    """Assuming r is Array, determine its size"""
    # adds size
    a.size = a.length * a.base.size
    return a

def genSet(s):
    s.size = 4; return s

def genGlobalVars(sc, start):
    pass

def genLocalVars(sc, start):
    pass

def genProgStart():
    pass

def genProgEntry(ident):
    pass

def genProgExit(x):
    return x

def genProcStart():
    pass

def genProcEntry(ident, parsize, localsize):
    pass

def genProcExit(x, parsize, localsize):
    pass

def genActualPara(ap, fp, n):
    pass

def genSelect(x, f):
    # x.f, assuming f is ST.Field
    return FieldSelection(x, f)

def genIndex(x, y):
    # x[y], assuming x is ST.Var, x.tp is ST.Array, y is Const or Reg integer
    return ArrayIndexing(x, y)

def genVar(x):
    # assuming x is ST.Var, ST.Ref, ST.Const
    return x
```

```
def genConst(x):
    return x

def genUnaryOp(op, x):
    return UnaryOp(op, x) if op in \
           {PLUS, MINUS, NOT, CARD, COMPLEMENT} else x

def genBinaryOp(op, x, y):
    return BinaryOp(op, x.tp, x, y)

def genRelation(op, x, y):
    return BinaryOp(op, Bool, x, y)

def genLeftAssign(x):
    return x

def genRightAssign(x):
    return x

def genAssign(x, y):
    return Assignment(x, y)

def genActualPara(ap, fp, n):
    return ap

def genCall(r, pr, ap):
    return Call(r, pr.name, ap)

def genRead(x):
    return Call(x, 'read', [])

def genWrite(x):
    return Call(None, 'write', [x])

def genWriteln():
    return Call(None, 'writeln', [])

def genSeq(x, y):
    return Seq(x, y)

def genThen(x):
    return x

def genIfThen(x, y):
    return IfThen(x, y)

def genElse(x, y):
    return y

def genIfElse(x, y, z):
    return IfElse(x, y, z)

def genWhile():
    pass

def genDo(x):
    return x

def genWhileDo(t, x, y):
    return While(x, y)
```