The P0 Compiler

COMP SCI 4TB3/6TB3, McMaster University
Emil Sekerinski, revised February 2022

This collection of Jupyter notebooks develops a compiler for P0, a programming language inspired by Pascal, a language designed for ease of compilation. The differences to Pascal are that indentation is used for bracketing, there are only value and result parameters (no reference parameters), multiple assignments are allowed, the syntax for procedure calls, procedure declarations, array declarations, and record declarations is different, and Unicode characters are used. The compiler currently generates WebAssebmbly and MIPS code, but is modularized to facilitate other targets. WebAssembly is representative of stack-based virtual machines while the MIPS architecture is representative of Reduced Instruction Set Computing (RISC) processors.

The P0 Language

The main syntactic elements of P0 are *statements*, *declarations*, *types*, and *expressions*.

Statements

- *Assignment statement* (x_1 , x_2 , … variable identifers, d selector, e , e_1 , e_2 , … expressions): $x_1, x_2, \ldots := e_1, e_2, \ldots$ x d := e
- *Procedure call* (p procedure identifier, e_1 , e_2 , … expressions, x_1 , x_2 , … variable identifiers): $p(e_1, e_2, \ldots)$ $x_1, x_2, \ldots \leftarrow p(e_1, e_2, \ldots)$
- *Sequential composition* (S_1 , S_2 , … statements): $S_1; S_2; \ldots$
- *If-statements* (B Boolean expression, S , T statements): if B then S if B then S else T
- *While-statements* (B Boolean expression, S statement): while B do S

Declarations

- *Constant declaration* (c constant identifier, e constant expression): const c = e
- *Type declaration* (t type identifier, T type): type t = T
- *Variable declaration* (x_1 , x_2 , … variable identifiers, T type): var $x_1, x_2, \ldots$: T
- *Procedure declaration* (p procedure identifier, v_1 , v_2 , … variable identifiers, T_1 , T_2 , …, U_1 , U_2 , … types, D_1 , D_2 , … declarations, S statement): procedure p $(v_1: T_1, v_2: T_2, \ldots) \rightarrow (r_1: U_1, r_2: U_2, \ldots)$ D_1 D_2 … S

Types

- *Elementary Types:* integer, boolean
- *Arrays* (m , n integer expressions, T type): [m .. n] → T
- *Records* (f_1 , f_2 , … field identifiers, T_1 , T_2 , …, types): $(f_1: T_1, f_2: T_2, \ldots)$
- *Sets* (m , n integer expressions) set [m .. n]

Expressions:

- *Constants:* number, identifier
- *Selectors* (i index expression, f field identifier): [i] .f
- *Operators,* in order of their binding power (e, e_1, e_2 are expressions): (e), ¬ e, #e, C e $e_1 \times e_2$, e_1 div e_2, e_1 mod e_2, $e_1 \cap e_2$, e_1 and e_2 + e, – e, $e_1 + e_2$, $e_1 - e_2$, $e_1 \cup e_2$, e_1 or e_2 $e_1 = e_2$, $e_1 \neq e_2$, $e_1 < e_2$, $e_1 \leq e_2$, $e_1 > e_2$, $e_1 \geq e_2$, $e_1 \in e_2$, $e_1 \subseteq e_2$, $e_1 \supseteq e_2$

Types `integer` , `boolean` , constants `true` , `false` , and procedures `read` , `write` , `writeln` are not symbols of the grammar; they are *standard identifiers* (*predefined identifiers*).

P0 Examples

```
procedure quotrem(x, y: integer) → (q, r: integer)
    q, r := 0, x
    while r ≥ y do // q × y + r = x ∧ r ≥ y
        r, q := r - y, q + 1

program arithmetic
    var x, y, q, r: integer
      x ← read(); y ← read()
      q, r ← quotrem(x, y)
      write(q); write(r)

procedure fact(n: integer) → (f: integer)
    if n = 0 then f := 1
    else
        f ← fact(n - 1); f := f × n

program factorial;
```

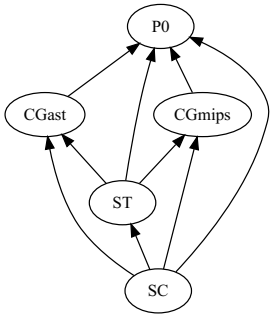
```
    var y, z: integer
        y ← read(); z ← fact(y); write(z)

const N = 10
var a: [0 .. N - 1] → integer

procedure has(x: integer) → (r: boolean)
    var i: integer
        i := 0
        while (i < N) and (a[i] ≠ x) do i := i + 1
        r := i < N
```

The P0 Grammar

```
selector ::= { "[" expression "]" | "." ident}
factor ::= ident selector | integer | "(" expression ")" | "{" [expression {"," expression}]
"}" | ("¬" | "#" | "C") factor
term ::= factor {("×" | "div" | "mod" | "∩" | "and") factor}
simpleExpression ::= ["+" | "-"] term {("+" | "-" | "∪" | "or") term}
expression ::= simpleExpression
    {("=" | "≠" | "<" | "≤" | ">" | "≥" | "∈" | "⊆" | "⊇") simpleExpression}
statementList ::= statement {";" statement}
statementBlock ::= statementList {statementList}
statementSuite ::= statementList | INDENT statementBlock DEDENT
statement ::=
    ident selector ":=" expression |
    ident {"," ident} (":=" expression {"," expression} |
        "←" ident "(" [expression {"," expression}] ")") |
    "if" expression "then" statementSuite ["else" statementSuite] |
    "while" expression "do" statementSuite
type ::=
    ident |
    "[" expression ".." expression "]" "→" type |
    "(" typedIds ")" |
    "set" "[" expression ".." expression "]"
typedIds ::= ident {"," ident} ":" type {"," ident {"," ident} ":" type}.
declarations ::=
    {"const" ident "=" expression}
    {"type" ident "=" type}
    {"var" typedIds}
    {"procedure" ident "(" [typedIds] ")" [ "→" "(" typedIds ")" ] body}
body ::= INDENT declarations (statementBlock | INDENT statementBlock DEDENT) DEDENT
program ::= declarations "program" ident body
```

Modularization

- The parser, `P0`, parses the source text, type-checks it, evaluates constant expressions, and generates target code, in one pass over the source text.
- The scanner, `SC`, reads characters of the source text and provides the next symbol to the parser; it allows errors to be reported at the current position in the source text.
- The symbol table, `ST`, stores all currently valid declarations, as needed for type-checking.
- The code generator, `CG`, provides the parser with procedures for generating code for P0 expressions, statements, and variable declarations, and procedure declarations.



The parser is the main program that calls the scanner, symbol table, and code generator.
All call the scanner for error reporting. The code generator augments the entries in the the symbol table, for example with the size and location of variables. There are three code generators: `CCGwat` generates WebAssembly code, `CGmips` generates MIPS code, and `CGast` generates only an abstract syntax tree.

The Parser

The scanner and symbol table are always imported. Depending on the selected target, a different code generator is imported when compilation starts.

In [ ]:
```
import nbimporter; nbimporter.options["only_defs"] = False
import SC  #  used for SC.init, SC.sym, SC.val, SC.error
from SC import IDENT, NUMBER, TIMES, DIV, MOD, PLUS, MINUS, AND, OR, \
    EQ, NE, LT, GT, LE, GE, SEMICOLON, COMMA, COLON, BECOMES, PERIOD, DOTDOT, \
    NOT, LPAREN, RPAREN, LBRAK, RBRAK, LBRACE, RBRACE, LARROW, RARROW, CARD, \
    COMPLEMENT, UNION, INTERSECTION, ELEMENT, SUBSET, SUPERSET, IF, THEN, ELSE, \
    WHILE, DO, CONST, TYPE, VAR, SET, PROCEDURE, PROGRAM, INDENT, DEDENT, EOF, \
    getSym, mark
```

```
import ST  #  used for ST.init
from ST import Var, Ref, Const, Type, Proc, StdProc, Int, Bool, \
    Record, Array, Set, newDecl, find, openScope, topScope, closeScope
```

The parser type-checks the source code; the code generators are supposed to compile all type-correct code, but may impose restrictions. Procedure `compatible` checks for structural compatibility. In case of sets, lower and upper bound of sets are not checked.

In [ ]:
```
def compatible(xt, yt):
    return xt == yt or \
           type(xt) == Set == type(yt) or \
           type(xt) == Array == type(yt) and xt.length == yt.length and \
               compatible(xt.base, yt.base) or \
           type(xt) == Record == type(yt) and \
               all(compatible(xf.tp, yf.tp) for xf, yf in zip(xt.fields, yt.fields))
```

The first sets for recursive descent parsing are:

In [ ]:
```
FIRSTSELECTOR = {LBRAK, PERIOD}
FIRSTFACTOR = {IDENT, NUMBER, LPAREN, NOT, CARD, COMPLEMENT}
FIRSTEXPRESSION = {PLUS, MINUS, IDENT, NUMBER, LPAREN, NOT, CARD, COMPLEMENT}
FIRSTSTATEMENT = {IDENT, IF, WHILE}
FIRSTTYPE = {IDENT, LPAREN}
FIRSTDECL = {CONST, TYPE, VAR, PROCEDURE}
```

Procedure `selector()` parses

```
selector ::= { "[" expression "]" | "." ident }
```

and generates code for the selector if not error is reported.

In [ ]:
```
def selector(x):
    while SC.sym in {LBRAK, PERIOD}:
        if SC.sym == LBRAK:  #  x[y]
            getSym(); y = expression()
            if type(x.tp) == Array:
                if y.tp == Int:
                    if type(y) == Const and (y.val < x.tp.lower or y.val >= x.tp.lower + x.tp.length):
                        mark('index out of bounds')
                    else: x = CG.genIndex(x, y)
                else: mark('index not integer')
            else: mark('not an array')
            if SC.sym == RBRAK: getSym()
            else: mark("] expected")
        else:  #  x.f
            getSym()
            if SC.sym == IDENT:
                if type(x.tp) == Record:
                    for f in x.tp.fields:
                        if f.name == SC.val:
                            x = CG.genSelect(x, f); break
                    else: mark("not a field")
                    getSym()
                else: mark("not a record")
            else: mark("identifier expected")
    return x
```

Procedure `factor()` parses

```
factor ::= ident selector | integer | "(" expression ")" | "{" [expression {"," expression}]
"}" | ("¬" | "#" | "C") factor
```

and generates code for the factor if no error is reported. If the factor is a constant, a `Const` item is returned (and code may not be generated); if the factor is not a constant, the location of the result is returned.

In [ ]:
```
def factor():
    if SC.sym == IDENT:
        x = find(SC.val)
        if type(x) == Var: x = CG.genVar(x); getSym()
        elif type(x) == Const: x = Const(x.tp, x.val); x = CG.genConst(x); getSym()
        else: mark('variable or constant identifier expected')
        x = selector(x)
    elif SC.sym == NUMBER:
        x = Const(Int, SC.val); x = CG.genConst(x); getSym()
    elif SC.sym == LPAREN:
        getSym(); x = expression()
        if SC.sym == RPAREN: getSym()
```

```
        else: mark(') expected')
    elif SC.sym == LBRACE:
        getSym()
        if SC.sym in FIRSTEXPRESSION:
            y = expression()
            if y.tp == Int: x = CG.genUnaryOp(SET, y)
            else: mark('not integer')
            while SC.sym == COMMA:
                getSym(); y = expression()
                if y.tp == Int: y = CG.genUnaryOp(SET, y)
                else: mark("not integer")
                x = CG.genBinaryOp(UNION, x, y)
        else: x = Const(Set(0, 32), 0); x = CG.genConst(x)
        if SC.sym == RBRACE: getSym()
        else: mark('} expected')
    elif SC.sym == NOT:
        getSym(); x = factor()
        if x.tp != Bool: mark('not boolean')
        elif type(x) == Const: x.val = 1 - x.val # constant folding
        else: x = CG.genUnaryOp(NOT, x)
    elif SC.sym in {CARD, COMPLEMENT}:
        op = SC.sym; getSym(); x = factor()
        if type(x.tp) == Set: x = CG.genUnaryOp(op, x)
        else: mark('set expected')
    else: mark('expression expected')
    return x
```

Procedure `term()` parses

```
term ::= factor {("×" | "div" | "mod" | "∩" | "and") factor}
```

and generates code for the term if no error is reported. If the term is a constant, a `Const` item is returned (and code may not be generated); if the term is not a constant, the location of the result is returned.

In [ ]:
```
def term():
    x = factor()
    while SC.sym in {TIMES, DIV, MOD, INTERSECTION, AND}:
        op = SC.sym; getSym();
        if op == AND and type(x) != Const: x = CG.genUnaryOp(AND, x)
        y = factor() # x op y
        if op in {TIMES, DIV, MOD} and x.tp == Int == y.tp:
            if type(x) == Const == type(y): # constant folding
                if op == TIMES: x.val = x.val * y.val
                elif op == DIV: x.val = x.val // y.val
                elif op == MOD: x.val = x.val % y.val
            else: x = CG.genBinaryOp(op, x, y)
        elif op == INTERSECTION and type(x.tp) == Set == type(y.tp):
            x = CG.genBinaryOp(op, x, y)
        elif op == AND and x.tp == Bool == y.tp:
            if type(x) == Const: # constant folding
                if x.val: x = y # if x is true, take y, else x
            else: x = CG.genBinaryOp(AND, x, y)
        else: mark('bad type')
    return x
```

Procedure `simpleExpression()` parses

```
simpleExpression ::= ["+" | "-"] term {("+" | "-" | "∪" | "or") term}
```

and generates code for the simple expression if no error is reported. If the simple expression is a constant, a `Const` item is returned (and code may not be generated); if the simple expression is not constant, the location of the result is returned.

In [ ]:
```
def simpleExpression():
    if SC.sym == PLUS:
        getSym(); x = term()
    elif SC.sym == MINUS:
        getSym(); x = term()
        if x.tp != Int: mark('bad type')
        elif type(x) == Const: x.val = - x.val # constant folding
        else: x = CG.genUnaryOp(MINUS, x)
    else: x = term()
    while SC.sym in {PLUS, MINUS, UNION, OR}:
        op = SC.sym; getSym()
        if op == OR and type(x) != Const: x = CG.genUnaryOp(OR, x)
        y = term() # x op y
        if op in {PLUS, MINUS} and x.tp == Int == y.tp:
            if type(x) == Const == type(y): # constant folding
                if op == PLUS: x.val = x.val + y.val
```

```
                elif op == MINUS: x.val = x.val - y.val
            else: x = CG.genBinaryOp(op, x, y)
        elif op == UNION and type(x.tp) == Set == type(y.tp):
            x = CG.genBinaryOp(UNION, x, y)
        elif op == OR and x.tp == Bool == y.tp:
            if type(x) == Const: # constant folding
                if not x.val: x = y # if x is false, take y, else x
            else: x = CG.genBinaryOp(OR, x, y)
        else: print(x, y); mark('bad type')
    return x
```

Procedure `expression()` parses

```
expression ::= simpleExpression
    {("=" | "≠" | "<" | "≤" | ">" | "≥" | "∈" | "⊆" | "⊇") simpleExpression}
```

and generates code for the expression if no error is reported. If the expression is a constant, a `Const` item is returned (and code may not be generated); if the expression is not constant, the location of the result is returned.

In [ ]:
```
def expression():
    x = simpleExpression()
    while SC.sym in {EQ, NE, LT, LE, GT, GE, ELEMENT, SUBSET, SUPERSET}:
        op = SC.sym; getSym()
        if op in (EQ, NE, LT, LE, GT, GE):
            y = simpleExpression() # x op y
            if x.tp == y.tp in (Int, Bool):
                if type(x) == Const == type(y): # constant folding
                    if op == EQ: x.val = int(x.val == y.val)
                    elif op == NE: x.val = int(x.val != y.val)
                    elif op == LT: x.val = int(x.val < y.val)
                    elif op == LE: x.val = int(x.val <= y.val)
                    elif op == GT: x.val = int(x.val > y.val)
                    elif op == GE: x.val = int(x.val >= y.val)
                    x.tp = Bool
                else: x = CG.genRelation(op, x, y)
            else: mark('bad type')
        elif (op == ELEMENT and x.tp == Int) or \
            (op in (SUBSET, SUPERSET) and type(x.tp) == Set):
            x = CG.genUnaryOp(op, x); y = simpleExpression()
            if type(y.tp) == Set: x = CG.genRelation(op, x, y)
            else: mark('set expected')
        else: mark('bad type')
    return x
```

Procedure `statementList()` parses

```
statementList ::= statement {";" statement}
```

and generates code for the statement list if no error is reported.

In [ ]:
```
def statementList():
    x = statement()
    while SC.sym == SEMICOLON:
        getSym(); y = statement(); x = CG.genSeq(x, y)
    return x
```

Procedure `statementBlock()` parses

```
statementBlock ::= statementList {statementList}
```

and generates code for the statement block if no error is reported. Each statement list has to start on a new line.

In [ ]:
```
def statementBlock():
    x = statementList()
    while SC.sym in FIRSTSTATEMENT:
        if not SC.newline: mark('new line expected')
        y = statementList(); x = CG.genSeq(x, y)
    return x
```

Procedure `statementSuite()` parses

```
statementSuite ::= statementList | INDENT statementBlock DEDENT
```

and generates code for the statement suite if no error is reported.

In [ ]:
```
def statementSuite():
    if SC.sym in FIRSTSTATEMENT: x = statementList()
    elif SC.sym == INDENT:
        getSym(); x = statementBlock()
        if SC.sym == DEDENT: getSym();
        else: mark('dedent or new line expected')
    else: mark("indented statement expected")
    return x
```

Procedure `statement()` parses

```
    statement ::=
        ident selector ":=" expression |
        ident "." ident
        ident {"," ident} (":=" expression {"," expression} |
            "←" ident "(" [expression {"," expression}] ")") |
        "if" expression "then" statementSuite ["else" statementSuite] |
        "while" expression "do" statementSuite
```

and generates code for the statement if no error is reported.

In [ ]:
```
def statement():
    if SC.sym == IDENT: # x := y, y(...), x ← y(...)
        # type(x) == Proc, StdProc: check no result parameters needed; call, y := true, x
        # type(x) ≠ Proc, StdProc: x := selector():
        #   sym == BECOMES: assignment; call := false
        #   sym == LARROW: check result paramter match, type(y) is Proc, StdProc,
        x = find(SC.val)
        if type(x) in {Proc, StdProc}: # procedure call without result
            if x.res != []: mark('variable for result expected')
            getSym(); call, xs, y = True, [], x
        elif type(x) == Var: # assignment or procedure call with result
            x = find(SC.val); x = CG.genVar(x); getSym()
            if SC.sym in FIRSTSELECTOR:
                xs = [CG.genLeftAssign(selector(x))] # array or field update
            else: # multiple assignment or procedure call with result
                xs = [CG.genLeftAssign(x)]
                while SC.sym == COMMA:
                    getSym();
                    if SC.sym == IDENT:
                        x = find(SC.val)
                        if x.name not in {x.name for x in xs}:
                            if type(x) == Var:
                                xs += [CG.genLeftAssign(CG.genVar(x))]; getSym()
                            else: mark('variable identifier expected')
                        else: mark('duplicate variable identifier')
                    else: mark('identifier expected')
            if SC.sym == BECOMES:
                getSym(); call = False; ys = [CG.genRightAssign(expression())] # xs := ys
                while SC.sym == COMMA: getSym(); ys += [CG.genRightAssign(expression())]
                if len(xs) == len(ys):
                    for x, y in zip(reversed(xs), reversed(ys)):
                        if compatible(x.tp, y.tp): x = CG.genAssign(x, y)
                        else: mark('incompatible assignment')
                else: mark('unbalanced assignment')
            elif SC.sym == LARROW:
                getSym()
                if SC.sym == IDENT: y = find(SC.val); getSym(); call = True
                else: mark('procedure identifier expected')
                if type(y) in {Proc, StdProc}:
                    if len(xs) == len(y.res):
                        for x, r in zip(xs, y.res):
                            if not compatible(x.tp, r.tp): mark('incompatible call')
                    else: mark('unbalanced call')
                else: mark('procedure expected')
            else: mark(':= or ← expected')
        else: mark("variable or procedure expected")
        if call: # call y(ap) or xs ← y(ap)
            fp, ap, i = y.par, [], 0   #  list of formals, list of actuals
            if SC.sym == LPAREN: getSym()
            else: mark("'(' expected")
            if SC.sym in FIRSTEXPRESSION:
                a = expression()
                if i < len(fp):
                    if compatible(fp[i].tp, a.tp):
                        ap.append(CG.genActualPara(a, fp[i], i))
                    else: mark('incompatible parameter')
                else: mark('extra parameter')
                i = i + 1
```

```
                while SC.sym == COMMA:
                    getSym()
                    a = expression()
                    if i < len(fp):
                        if compatible(fp[i].tp, a.tp):
                            ap.append(CG.genActualPara(a, fp[i], i))
                        else: mark('incompatible parameter')
                    else: mark('extra parameter')
                    i = i + 1
            if SC.sym == RPAREN: getSym()
            else: mark("')' expected")
            if i < len(fp): mark('too few parameters')
            elif type(y) == StdProc:
                if y.name == 'read': x = CG.genRead(x)
                elif y.name == 'write': x = CG.genWrite(a)
                elif y.name == 'writeln': x = CG.genWriteln()
            else: x = CG.genCall(xs, y, ap)
    elif SC.sym == IF:
        getSym(); x = expression();
        if x.tp == Bool: x = CG.genThen(x)
        else: mark('boolean expected')
        if SC.sym == THEN: getSym()
        else: mark("'then' expected")
        y = statementSuite()
        if SC.sym == ELSE:
            getSym()
            y = CG.genElse(x, y)
            z = statementSuite()
            x = CG.genIfElse(x, y, z)
        else:
            x = CG.genIfThen(x, y)
    elif SC.sym == WHILE:
        getSym(); t = CG.genWhile(); x = expression()
        if x.tp == Bool: x = CG.genDo(x)
        else: mark('boolean expected')
        if SC.sym == DO: getSym()
        else: mark("'do' expected")
        y = statementSuite()
        x = CG.genWhileDo(t, x, y)
    else: mark('statement expected')
    return x
```

Procedure `typ` parses

```
type ::=
    ident |
    "[" expression ".." expression "]" "→" type |
    "(" typedIds ")" |
    "set" "[" expression ".." expression "]"
```

and returns a type descriptor if not error is reported. The array bound are checked to be constants; the lower bound must be smaller or equal to the upper bound.

In [ ]:
```
def typ():
    if SC.sym == IDENT:
        ident = SC.val; x = find(ident)
        if type(x) == Type: x = Type(x.val); getSym()
        else: mark('type identifier expected')
    elif SC.sym == LBRAK:
        getSym(); x = expression()
        if SC.sym == DOTDOT: getSym()
        else: mark("'..' expected")
        y = expression()
        if SC.sym == RBRAK: getSym()
        else: mark("']' expected")
        if SC.sym == RARROW: getSym()
        else: mark("'→' expected")
        z = typ().val;
        if type(x) != Const or x.val < 0: mark('bad lower bound')
        elif type(y) != Const or y.val < x.val: mark('bad upper bound')
        else: x = Type(CG.genArray(Array(z, x.val, y.val - x.val + 1)))
    elif SC.sym == LPAREN:
        getSym(); openScope(); typedIds()
        if SC.sym == RPAREN: getSym()
        else: mark("')' expected")
        r = topScope(); closeScope()
        x = Type(CG.genRec(Record(r)))
    elif SC.sym == SET:
        getSym();
        if SC.sym == LBRAK: getSym()
```

```
        else: mark ("'[' expected")
        x = expression()
        if SC.sym == DOTDOT: getSym()
        else: mark("'..' expected")
        y = expression()
        if SC.sym == RBRAK: getSym()
        else: mark("']' expected")
        if type(x) != Const: mark('bad lower bound')
        elif type(y) != Const or y.val < x.val: mark('bad upper bound')
        else: x = Type(CG.genSet(Set(x.val, y.val - x.val + 1)))
    else: mark('type expected')
    return x
```

Procedure `typeIds()` parses

```
typedIds ::= ident {"," ident} ":" type {"," ident {"," ident} ":" type}.
```

and updates the top scope of symbol table; an error is reported if an identifier is already in the top scope.

In [ ]:
```
def typedIds():
    if SC.sym == IDENT: tid = [SC.val]; getSym()
    else: mark("identifier expected")
    while SC.sym == COMMA:
        getSym()
        if SC.sym == IDENT: tid.append(SC.val); getSym()
        else: mark('identifier expected')
    if SC.sym == COLON: getSym()
    else: mark("':' expected")
    tp = typ().val
    for i in tid: newDecl(i, Var(tp))
    while SC.sym == COMMA:
        getSym()
        if SC.sym == IDENT: tid = [SC.val]; getSym()
        else: mark("identifier expected")
        while SC.sym == COMMA:
            getSym()
            if SC.sym == IDENT: tid.append(SC.val); getSym()
            else: mark('identifier expected')
        if SC.sym == COLON: getSym()
        else: mark("':' expected")
        tp = typ().val
        for i in tid: newDecl(i, Var(tp))
```

Procedure `declarations(allocVar)` parses

```
declarations ::=
    {"const" ident "=" expression}
    {"type" ident "=" type}
    {"var" typedIds}
    {"procedure" ["(" ident ":" type ")"] ident "(" [typedIds] ")" [ "→" "(" typedIds ")" ]
body}
```

and updates the top scope of symbol table; an error is reported if an identifier is already in the top scope. An error is also reported if the expression of a constant declarations is not constant. For each procedure, a new scope is opened for its formal parameters and local declarations, the formal parameters and added to the symbol table, and code is generated for the body. The size of the variable declarations is returned, as determined by calling paramater `allocVar`.

In [ ]:
```
def declarations(allocVar):
    while SC.sym == CONST:
        getSym()
        if SC.sym == IDENT: ident = SC.val; getSym()
        else: mark("constant name expected")
        if SC.sym == EQ: getSym()
        else: mark("= expected")
        x = expression()
        if type(x) == Const: newDecl(ident, x)
        else: mark('expression not constant')
    while SC.sym == TYPE:
        getSym()
        if SC.sym == IDENT: ident = SC.val; getSym()
        else: mark("type name expected")
        if SC.sym == EQ: getSym()
        else: mark("= expected")
        x = typ(); newDecl(ident, x)  #  x is of type ST.Type
    start = len(topScope())
    while SC.sym == VAR:
        getSym(); typedIds()
```

```
    var = allocVar(topScope(), start)
    while SC.sym == PROCEDURE:
        getSym()
        if SC.sym == LPAREN:
            getSym()
            if SC.sym == IDENT: r = SC.val; getSym()
            else: mark("identifier expected")
            if SC.sym == COLON: getSym()
            else: mark("':' expected")
            tp = typ().val
            if SC.sym == RPAREN: getSym()
            else: mark(") expected")
        else: r = None
        if SC.sym == IDENT: ident = SC.val; getSym()
        else: mark("procedure name expected")
        newDecl(ident, Proc([], [])) #  entered without parameters
        sc = topScope(); openScope() # new scope for parameters and body
        if r: newDecl(r, Var(tp))
        if SC.sym == LPAREN: getSym()
        else: mark("( expected")
        if SC.sym == IDENT: typedIds()
        fp = topScope()
        if SC.sym == RPAREN: getSym()
        else: mark(") expected")
        d = len(fp)
        if SC.sym == RARROW:
            getSym()
            if SC.sym == LPAREN: getSym()
            else: mark('( expected')
            typedIds()
            if SC.sym == RPAREN: getSym()
            else: mark(') expected')
        sc[-1].par, sc[-1].res = fp[:d], fp[d:] #  procedure parameters updated
        para = CG.genProcStart(ident, fp[:d], fp[d:])
        body(ident, para); closeScope() #  scope for parameters and body closed
    return var
```

Procedure `body` parses

```
body ::= INDENT declarationBlock (statementBlock | INDENT statementBlock DEDENT) DEDENT
```

and returns the generated code if no error is reported.

In [ ]:
```
def body(ident, para):
    if SC.sym == INDENT: getSym()
    else: mark('indent expected')
    start = len(topScope())
    local = declarations(CG.genLocalVars)
    CG.genProcEntry(ident, para, local)
    if SC.sym in FIRSTSTATEMENT: x = statementBlock()
    elif SC.sym == INDENT:
        getSym(); x = statementBlock()
        if SC.sym == DEDENT: getSym()
        else: mark('dedent or new line expected')
    else: mark('statement expected')
    CG.genProcExit(x, para, local)
    if SC.sym == DEDENT: getSym()
    else: mark('dedent or new line expected')
    return x
```

Procedure `program` parses

```
program ::= declarations "program" ident body
```

and returns the generated code if no error is reported. The standard identifiers are entered initially in the symbol table.

In [ ]:
```
def program():
    newDecl('boolean', Type(CG.genBool(Bool)))
    newDecl('integer', Type(CG.genInt(Int)))
    newDecl('true', Const(Bool, 1))
    newDecl('false', Const(Bool, 0))
    newDecl('read', StdProc([], [Var(Int)]))
    newDecl('write', StdProc([Var(Int)], []))
    newDecl('writeln', StdProc([], []))
    CG.genProgStart()
    declarations(CG.genGlobalVars)
    if SC.sym == PROGRAM: getSym()
    else: mark("'program' expected")
```

```
    ident = SC.val
    if SC.sym == IDENT: getSym()
    else: mark('program name expected')
    openScope(); CG.genProgEntry(ident); x = body(ident, 0)
    closeScope(); x = CG.genProgExit(x)
    return x
```

Procedure `compileString(src, dstfn, target)` compiles the source as given by string `src`; if `dstfn` is provided, the code is written to a file by that name, otherwise printed on the screen. If `target` is omitted, MIPS code is generated.

In [ ]:
```
def compileString(src, dstfn = None, target = 'wat'):
    global CG
    if target == 'wat': import CGwat as CG
    elif target == 'mips': import CGmips as CG
    elif target == 'ast': import CGast as CG
    else: print('unknown target'); return
    try:
        SC.init(src); ST.init(); p = program()
        if dstfn == None: return str(p)
        else:
            with open(dstfn, 'w') as f: f.write(p)
    except Exception as msg:
        raise Exception(str(msg))
```

Procedure `compileFile(srcfn, target)` compiles the file named `scrfn`, which must have the extension `.p`, and generates assembly code in a file with extension `.s`. If `target` is omitted, MIPS code is generated.

In [ ]:
```
def compileFile(srcfn, target = 'wat'):
    if srcfn.endswith('.p'):
        with open(srcfn, 'r') as f: src = f.read()
        dstfn = srcfn[:-2] + '.s'
        compileString(src, dstfn, target)
    else: print("'.p' file extension expected")
```

Sample usage (in code cell):

```
cd /path/to/my/prog
compileFile('myprog.p')
```