P0

March 12, 2018

0.1 The P0 Compiler

COMP SCI 4TB3/6TB3, McMaster University

Original Author: Emil Sekerinski, February 2017 This collection of *jupyter notebooks* develops a compiler for P0, a subset of Pascal. The compiler generates MIPS code, but is modularized to facilitate other targets. Pascal is a language that was designed with ease of compilation in mind; the MIPS architecture is representative of Reduced Instruction Set Computing (RISC) processors.

0.1.1 The P0 Language

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

Statements

- *Assignment statement* (`x` variable identifer, `f` field identifier, `i`, `e` expressions): x := e x.f := e x[i] := e
- *Procedure call* (`p` procedure identifier, `e`, `e`, … expressions): p(e, e, …)
- *Compound statement* (`S`, `S`, … statements): begin S; S; … end
- *If-statements* (`c` Boolean expression, `S`, `T` statements): if c then S if c then S else T
- *While-statements* (`c` Boolean expression, `S` statement): while c 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`, `x`, … variable identifiers, `T` type): var x, x, …: T;
- *Procedure Declaration* (`p` procedure identifier, `v`, `v`, …, `r`, `r`, … variable identifiers, `T`, `T`, … types, `D`, `D`, … declarations, `S`, `S`, … statements): procedure p (v, v, …:T; var r, r, …: T; …) D, D, … begin S, S; … end;

Types

- *Elementary Types:* integer, boolean
- _Arrays (`m`, `n` expressions, `T` type): array [m .. n] of T
- _Records (`f`, `f`, g, g, … identifiers, `T`, `U`, … types): record f, f, … : T; g, g, … : U; … end

Expressions:

- *Constants:* number, identifier
- *Selectors* (x, f identifiers, i expression): x[i], x.f
- *Operators,* in order of their binding power (e, e, e are expressions): (e), not e e * e, e div e, e mod e, e and e
  - e, – e, e + e, e – e, e or e e = e, e <> e, e < e, e <= e, e > e, e >= e

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

0.1.2 P0 Examples

```
program arithmetic;
  var x, y, q, r: integer;
  procedure QuotRem(x, y: integer; var q, r: integer);
    begin q := 0; r := x;
      while r >= y do { q*y+r=x and r>=y }
        begin r := r - y; q := q + 1
        end
    end;
  begin
    read(x); read(y);
    QuotRem(x, y, q, r);
    write(q); write(r); writeln
  end.

program factorial;
  var y, z: integer;
  procedure fact(n: integer; var f: integer);
    begin
      if n = 0 then f := 1
      else
        begin fact(n - 1, f); f := f * n end
    end;
  begin
    read(y);
    fact(y, z);
    write(z)
  end.
```

0.1.3 The P0 Grammar

```
selector ::= {"." ident | "[" expression "]"}.
factor ::= ident selector | integer | "(" expression ")" | "not" factor.
term ::= factor {("*" | "div" | "mod" | "and") factor}.
simpleExpression ::= ["+" | "-"] term {("+" | "-" | "or") term}.
expression ::= simpleExpression
    {("=" | "<>" | "<" | "<=" | ">" | ">=") simpleExpression}.
```

```
compoundStatement = "begin" statement {";" statement} "end"
statement ::=
    ident selector ":=" expression |
    ident "(" [expression {"," expression}] ")" |
    compoundStatement |
    "if" expression "then" statement ["else"statement] |
    "while" expression "do" statement.
type ::=
    ident |
    "array" "[" expression ".." expression "]" "of" type |
    "record" typedIds {";" typedIds} "end".
typedIds ::= ident {"," ident} ":" type.
declarations ::=
    {"const" ident "=" expression ";"}
    {"type" ident "=" type ";"}
    {"var" typedIds ";"}
    {"procedure" ident ["(" [["var"] typedIds {";" ["var"] typedIds}] ")"] ";"
        declarations compoundStatement ";"}.
program ::= "program" ident ";" declarations compoundStatement.
```

0.1.4 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 two code generators: `CGmips` generates MIPS code and `CGast` generates only an abstract syntax tree.

0.1.5 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 [1]: import nbimporter
        import SC  #  used for SC.init, SC.sym, SC.val, SC.error
        from SC import TIMES, DIV, MOD, AND, PLUS, MINUS, OR, EQ, NE, LT, GT, \
            LE, GE, PERIOD, COMMA, COLON, RPAREN, RBRAK, OF, THEN, DO, LPAREN, \
            LBRAK, NOT, BECOMES, NUMBER, IDENT, SEMICOLON, END, ELSE, IF, WHILE, \
            ARRAY, RECORD, CONST, TYPE, VAR, PROCEDURE, BEGIN, PROGRAM, EOF, \
            getSym, mark
        import ST  #  used for ST.init
        from ST import Var, Ref, Const, Type, Proc, StdProc, Int, Bool, Enum, \
            Record, Array, newDecl, find, openScope, topScope, closeScope
```

```
importing Jupyter notebook from SC.ipynb


<IPython.core.display.HTML object>


importing Jupyter notebook from ST.ipynb
```

The first and follow sets for recursive descent parsing.

```
In [2]: FIRSTFACTOR = {IDENT, NUMBER, LPAREN, NOT}
        FOLLOWFACTOR = {TIMES, DIV, MOD, AND, OR, PLUS, MINUS, EQ, NE, LT, LE, GT, GE,
                        COMMA, SEMICOLON, THEN, ELSE, RPAREN, RBRAK, DO, PERIOD, END}
        FIRSTEXPRESSION = {PLUS, MINUS, IDENT, NUMBER, LPAREN, NOT}
        FIRSTSTATEMENT = {IDENT, IF, WHILE, BEGIN}
        FOLLOWSTATEMENT = {SEMICOLON, END, ELSE}
        FIRSTTYPE = {IDENT, RECORD, ARRAY, LPAREN}
        FOLLOWTYPE = {SEMICOLON}
        FIRSTDECL = {CONST, TYPE, VAR, PROCEDURE}
        FOLLOWDECL = {BEGIN}
        FOLLOWPROCCALL = {SEMICOLON, END, ELSE}
        STRONGSYMS = {CONST, TYPE, VAR, PROCEDURE, WHILE, IF, BEGIN, EOF}
```

Procedure `selector(x)` parses

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

Assuming `x` is the entry for the identifier in front of the selector, generates code for the selector or reports error.

```
In [3]: def selector(x):
            while SC.sym in {PERIOD, LBRAK}:
                if SC.sym == PERIOD:  #  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")
                else:  #  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")
            return x
```

Procedure `factor()` parses

```
factor ::= ident selector | integer | "(" expression ")" | "not" 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 need to be generated); if the factor is not a constant, the location of the result is returned as determined by the code generator.

```
In [4]: def factor():
            if SC.sym not in FIRSTFACTOR:
                mark("expression expected"); getSym()
                while SC.sym not in FIRSTFACTOR | STRONGSYMS | FOLLOWFACTOR:
                    getSym()
            if SC.sym == IDENT:
                x = find(SC.val)
                if type(x) in {Var, Ref}: x = CG.genVar(x)
                elif type(x) == Const: x = Const(x.tp, x.val); x = CG.genConst(x)
                else: mark('expression expected')
                getSym(); 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 == 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)
            else: x = Const(None, 0)
            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 need to be generated); if the term is not a constant, the location of the result is returned as determined by the code generator.

```
In [5]: def term():
            x = factor()
            while SC.sym in {TIMES, DIV, MOD, AND}:
                op = SC.sym; getSym();
                if op == AND and type(x) != Const: x = CG.genUnaryOp(AND, x)
                y = factor() # x op y
                if x.tp == Int == y.tp and op in {TIMES, DIV, MOD}:
                    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 x.tp == Bool == y.tp and op == AND:
                    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 need to be generated); the simple expression is not constant, the location of the result is returned as determined by the code generator.

```
In [6]: 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, OR}:
                op = SC.sym; getSym()
                if op == OR and type(x) != Const: x = CG.genUnaryOp(OR, x)
                y = term() # x op y
                if x.tp == Int == y.tp and op in {PLUS, MINUS}:
                    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 x.tp == Bool == y.tp and op == OR:
                    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: mark('bad type')
            return x
```

Procedure `expression()` parses

```
expression ::= simpleExpression
             {("=" | "<>" | "<" | "<=" | ">" | ">=") simpleExpression}.
```

and generates code for the term if no error is reported. The location of the result is returned as determined by the code generator.

```
In [7]: def expression():
            x = simpleExpression()
            while SC.sym in {EQ, NE, LT, LE, GT, GE}:
                op = SC.sym; getSym(); y = simpleExpression() # x op y
                if x.tp == Int == y.tp:
                    x = CG.genRelation(op, x, y)
                else: mark('bad type')
            return x
```

Procedure `compoundStatement()` parses

```
compoundStatement ::= "begin" statement {";" statement} "end"
```

and generates code for the term if no error is reported. A result is returned as determined by the code generator.

```
In [8]: def compoundStatement():
            if SC.sym == BEGIN: getSym()
            else: mark("'begin' expected")
            x = statement()
            while SC.sym == SEMICOLON or SC.sym in FIRSTSTATEMENT:
                if SC.sym == SEMICOLON: getSym()
                else: mark("; missing")
                y = statement(); x = CG.genSeq(x, y)
            if SC.sym == END: getSym()
            else: mark("'end' expected")
            return x
```

Procedure `statement()` parses

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

and generates code for the statement if no error is reported. A result is returned as determined by the code generator.

```
In [9]: def statement():
            if SC.sym not in FIRSTSTATEMENT:
                mark("statement expected"); getSym()
                while SC.sym not in FIRSTSTATEMENT | STRONGSYMS | FOLLOWSTATEMENT:
                    getSym()
            if SC.sym == IDENT:
                x = find(SC.val); getSym()
                if type(x) in {Var, Ref}:
                    x = CG.genVar(x); x = selector(x)
                    if SC.sym == BECOMES:
                        getSym(); y = expression()
                        if x.tp == y.tp in {Bool, Int}: x = CG.genAssign(x, y)
                        else: mark('incompatible assignment')
                    elif SC.sym == EQ:
                        mark(':= expected'); getSym(); y = expression()
                    else: mark(':= expected')
                elif type(x) in {Proc, StdProc}:
                    fp, ap, i = x.par, [], 0   #  list of formals, list of actuals
                    if SC.sym == LPAREN:
                        getSym()
                        if SC.sym in FIRSTEXPRESSION:
                            y = expression()
                            if i < len(fp):
                                if (type(fp[i]) == Var or type(y) == Var) and \
                                   fp[i].tp == y.tp:
                                    if type(x) == Proc:
                                        ap.append(CG.genActualPara(y, fp[i], i))
                                else: mark('illegal parameter mode')
                            else: mark('extra parameter')
                            i = i + 1
                            while SC.sym == COMMA:
                                getSym()
                                y = expression()
                                if i < len(fp):
                                    if (type(fp[i]) == Var or type(y) == Var) and \
                                       fp[i].tp == y.tp:
                                        if type(x) == Proc:
                                            ap.append(CG.genActualPara(y, fp[i], i))
                                    else: mark('illegal parameter mode')
                                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(x) == StdProc:
```

```
                    if x.name == 'read': x = CG.genRead(y)
                    elif x.name == 'write': x = CG.genWrite(y)
                    elif x.name == 'writeln': x = CG.genWriteln()
                else: x = CG.genCall(x, ap)
            else: mark("variable or procedure expected")
        elif SC.sym == BEGIN: x = compoundStatement()
        elif SC.sym == IF:
            getSym(); x = expression();
            if x.tp == Bool: x = CG.genCond(x)
            else: mark('boolean expected')
            if SC.sym == THEN: getSym()
            else: mark("'then' expected")
            y = statement()
            if SC.sym == ELSE:
                if x.tp == Bool: y = CG.genThen(x, y);
                getSym(); z = statement();
                if x.tp == Bool: x = CG.genIfElse(x, y, z)
            else:
                if x.tp == Bool: x = CG.genIfThen(x, y)
        elif SC.sym == WHILE:
            getSym(); t = CG.genTarget(); x = expression()
            if x.tp == Bool: x = CG.genCond(x)
            else: mark('boolean expected')
            if SC.sym == DO: getSym()
            else: mark("'do' expected")
            y = statement()
            if x.tp == Bool: x = CG.genWhile(t, x, y)
        else: x = None
        return x
```

Procedure typ parses

```
type ::= ident |
         "array" "[" expression ".." expression "]" "of" type |
         "record" typedIds {";" typedIds} "end"
```

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 [10]: def typ():
             if SC.sym not in FIRSTTYPE:
                 getSym(); mark("type expected")
                 while SC.sym not in FIRSTTYPE | STRONGSYMS | FOLLOWTYPE:
                     getSym()
             if SC.sym == IDENT:
                 ident = SC.val; x = find(ident); getSym()
                 if type(x) == Type: x = Type(x.val)
                 else: mark('not a type'); x = Type(None)
             elif SC.sym == ARRAY:
                 getSym()
                 if SC.sym == LBRAK: getSym()
                 else: mark("'[' expected")
                 x = expression()
                 if SC.sym == PERIOD: getSym()
                 else: mark("'.' expected")
                 if SC.sym == PERIOD: getSym()
                 else: mark("'.' expected")
                 y = expression()
                 if SC.sym == RBRAK: getSym()
                 else: mark("']' expected")
                 if SC.sym == OF: getSym()
                 else: mark("'of' expected")
                 z = typ().val;
                 if type(x) != Const or x.val < 0:
                     mark('bad lower bound'); x = Type(None)
                 elif type(y) != Const or y.val < x.val:
                     mark('bad upper bound'); x = Type(None)
                 else: x = Type(CG.genArray(Array(z, x.val, y.val - x.val + 1)))
             elif SC.sym == RECORD:
                 getSym(); openScope(); typedIds(Var)
                 while SC.sym == SEMICOLON:
                     getSym(); typedIds(Var)
                 if SC.sym == END: getSym()
                 else: mark("'end' expected")
                 r = topScope(); closeScope()
                 x = Type(CG.genRec(Record(r)))
             else: x = Type(None)
             return x
```

Procedure typeIds(kind) parses

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

and updates the top scope of symbol table; an error is reported if an identifier is already in the top scope. The parameter kind is assumed to be callable and applied to the type before an identifier and its type are entered in the symbol table.

```
In [11]: def typedIds(kind):
             if SC.sym == IDENT: tid = [SC.val]; getSym()
             else: mark("identifier expected"); tid = []
             while SC.sym == COMMA:
                 getSym()
                 if SC.sym == IDENT: tid.append(SC.val); getSym()
                 else: mark('identifier expected')
             if SC.sym == COLON:
                 getSym(); tp = typ().val
                 if tp != None:
                    for i in tid: newDecl(i, kind(tp))
                else: mark("':' expected")
```

Procedure declarations(allocVar) parses

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

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

```
            varsize = allocVar(topScope(), start)
            while SC.sym == PROCEDURE:
                getSym()
                if SC.sym == IDENT: getSym()
                else: mark("procedure name expected")
                ident = SC.val; newDecl(ident, Proc([])) #  entered without parameters
                sc = topScope()
                CG.genProcStart(); openScope() # new scope for parameters and body
                if SC.sym == LPAREN:
                    getSym()
                    if SC.sym in {VAR, IDENT}:
                        if SC.sym == VAR: getSym(); typedIds(Ref)
                        else: typedIds(Var)
                        while SC.sym == SEMICOLON:
                            getSym()
                            if SC.sym == VAR: getSym(); typedIds(Ref)
                            else: typedIds(Var)
                    else: mark("formal parameters expected")
                    fp = topScope()
                    sc[-1].par = fp[:] #  procedure parameters updated
                    if SC.sym == RPAREN: getSym()
                    else: mark(") expected")
                else: fp = []
                parsize = CG.genFormalParams(fp)
                if SC.sym == SEMICOLON: getSym()
                else: mark("; expected")
                localsize = declarations(CG.genLocalVars)
                CG.genProcEntry(ident, parsize, localsize)
                x = compoundStatement(); CG.genProcExit(x, parsize, localsize)
                closeScope() #  scope for parameters and body closed
                if SC.sym == SEMICOLON: getSym()
                else: mark("; expected")
            return varsize
```

Procedure program parses

```
program ::= "program" ident ";" declarations compoundStatement
```

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

```
In [13]: 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([Ref(Int)]))
            newDecl('write', StdProc([Var(Int)]))
            newDecl('writeln', StdProc([]))
```

```
           CG.genProgStart()
           if SC.sym == PROGRAM: getSym()
           else: mark("'program' expected")
           ident = SC.val
           if SC.sym == IDENT: getSym()
           else: mark('program name expected')
           if SC.sym == SEMICOLON: getSym()
           else: mark('; expected')
           declarations(CG.genGlobalVars); CG.genProgEntry(ident)
           x = compoundStatement()
           return CG.genProgExit(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 [14]: def compileString(src, dstfn = None, target = 'mips'):
             global CG
             if target == 'mips': import CGmips as CG
             elif target == 'ast': import CGast as CG
             else: print('unknown target'); return
             SC.init(src)
             ST.init()
             p = program()
             if p != None and not SC.error:
                 if dstfn == None: print(p)
                 else:
                     with open(dstfn, 'w') as f: f.write(p);
```

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 [15]: def compileFile(srcfn, target = 'mips'):
             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')
```

0.2 Appendix

```
In [16]: %%HTML
         <style>
         div.prompt {display:none}
         </style>
```

```
<IPython.core.display.HTML object>
```

```
In [17]: from graphviz import Source
         src = Source('digraph { rankdir=BT; SC -> ST; ST -> CGast; SC -> CGast; \
         ST -> CGmips; SC -> CGmips; SC -> PO; ST -> PO; CGmips -> PO; CGast -> PO;}')
         src.format = 'svg'; src.render('modularization')

Out[17]: 'modularization.svg'
```