Pascal0 Compiler

Emil Sekerinski
McMaster University

Daniel Zingaro
McMaster University

revised January 2011

Abstract

Pascal0 is a subset of the Pascal programming language, suited for teaching the principles of compiler construction. This report describes a one-pass recursive descent compiler for Pascal0. The compiler is written in Pascal and generates code for RISC, a register-based instruction set, and STACK, a stack-oriented instruction set. Interpreters for RISC and STACK are included; these allow the execution of the generated code to be traced. The presentation of the compiler exemplifies modularization and documentation principles.

This document is written using the `noweb` literate programming tool. The Pascal source code is automatically extracted from this document.

1 Pascal0

1.1 Context

The Pascal0 compiler takes source text in a subset language of Pascal, and compiles it into executable code. For the most part, we try to stay in accordance with Extended Pascal (ISO 10206:1990). Where deviations are made, they will be noted in this document.

1.2 Symbols

```
ident = letter (letter | digit).
integer = digit {digit}.
```

1.3 Syntax

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

assignment = ident selector ":=" expression.
ActualParameters = "(" [expression {"," expression}] ")".
ProcedureCall = ident selector [ActualParameters].
CompoundStatement = "begin" statement {";" statement} "end".
IfStatement = "if" expression "then" Statement ["else" Statement].
WhileStatement = "while" expression "do" Statement.
Statement = [assignment | ProcedureCall | CompoundStatement |
  IfStatement | WhileStatement].

IdentList = ident {"," ident}.
ArrayType = "array" "[" expression ".." expression "]" "of" type.
FieldList = [IdentList ":" type].
RecordType = "record" FieldList {";" FieldList} "end".
type = ident | ArrayType | RecordType.
FPSection = ["var"] IdentList ":" type.
FormalParameters = "(" [FPSection {";" FPSection}] ")".
ProcedureDeclaration = "procedure" ident [FormalParameters] ";"
  declarations CompoundStatement.
declarations = ["const" {ident "=" expression ";"}]
  ["type" {ident "=" type ";"}]
  ["var" {IdentList ":" type ";"}]
```

```
    {ProcedureDeclaration ";"}.

  program = "program" ident ["(" ident {"," ident} ")"] ";"
    declarations CompoundStatement.
```

1.4 Examples

```
program multiply;

  var x,y,z: integer;

begin
  read(x); read(y); z := 0;
  while x > 0 do
    begin
      if x mod 2 = 1 then z := z + y;
      y := 2 * y; x := x div 2
    end;
  write(x); write(y); write(z); writeln
end.

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 bubblesort;

  const maximum = 20;
```

```
  var x: array [1..maximum] of integer; {array to sort}
    outer, inner, size: integer;

  procedure order (var x, y: integer);
    var h: integer;
  begin
    if x > y then begin h := x; x := y; y := h end
  end;

begin
  {Read unsorted numbers}
  read (size); outer := 1;
  while (outer <= size) and (outer <= maximum) do
    begin read (x[outer]); outer := outer + 1 end;

  {Sort the array}
  outer := 1;
  while outer < size do
    begin inner := 1;
      while inner <= size − outer do
        begin
          order (x[inner], x[inner+1]);
          inner := inner + 1
        end;
      outer := outer + 1
    end;

  {Print the sorted array}
  outer := 1;
  while outer <= size do
    begin write (x[outer]); outer := outer + 1 end
end.
```

1.5 Using the Compiler

To use the compiler to parse and interpret a Pascal0 program, you must "make" the RISC or STACK compilers. To do this with Freepascal (FPC), simply issue commands `fpc risccompiler.pas` and `fpc stackcompiler.pas`. Then, type `risccompiler <filename>` or `stackcompiler <filename>` to compile and interpret a Pascal0 program using the specified parser and interpreter.

2 Modularization

[Phases, intermediate representations, passes, all as on slides]

The Pascal-0 compiler is a *single-pass* compiler, meaning that code is generated as the source is read. This allows to simplify the structure of the compiler as intermediate tree representations need not be constructed. The structure is explained in terms of the *services* and *secrets* of each module and the module dependency diagram in Fig. **??**:

Scanner The tasks of the scanner are (1) to determine the name of the source file from the command line, (2) to recognize symbols from the characters of the source file, and (3) to print error messages. The scanner encapsulates the algorithm for recognizing symbols as well as input and output operations. The scanner is implemented as a module with procedures that are called by the parser.

Parser The tasks of the parser are (1) to determine the tree structure of the input, (2) to check its well-formedness, and (3) to select the code that needs to be generated. The parser is implemented as a main program that uses the scanner for reading symbols and for issuing error messages, uses the symbol table to maintain context dependencies, and uses a code generator to emit code. The parser encapsulates the parsing algorithm, handling of errors in the source program, and code selection.

Symbol Table The symbol table maintains for each identifier the context-dependencies given by its declarations as well as its address in memory as required by the code generator. Following the scoping rules, this information is maintained for each level. The symbol table encapsulates the data structure and algorithms for storing and finding the entry of an identifier.

Code Generator The code generator is responsible for emitting code into the code array. It encapsulates procedures for working with the instruction formats supported by RISC or stack, and can also display a text representation of the code it produces.

Interpreter The RISC and STACK interpreters take code emitted by their respective generators, and interpret (execute) it.

3 Scanner

⟨*scanner.pas*⟩ ≡
unit *scanner*;
interface

const
IdLen = 31; {number of significant characters in identifiers}

type
  Symbol = (*null*, *TimesSym*, *DivSym*, *ModSym*, *AndSym*, *PlusSym*, *MinusSym*,
    OrSym, *EqlSym*, *NeqSym*, *LssSym*, *GeqSym*, *LeqSym*, *GtrSym*, *PeriodSym*,
    CommaSym, *ColonSym*, *RparenSym*, *RbrakSym*, *OfSym*, *ThenSym*, *DoSym*,
    LparenSym, *LbrakSym*, *NotSym*, *BecomesSym*, *NumberSym*, *IdentSym*,
    SemicolonSym, *EndSym*, *ElseSym*, *IfSym*, *WhileSym*, *ArraySym*, *RecordSym*,
    ConstSym, *TypeSym*, *VarSym*, *ProcedureSym*, *BeginSym*, *ProgramSym*,
    EofSym);
  Identifier = **string**[*IdLen*];

var
  sym: *Symbol*; {next symbol}
  val: *integer*; {value of number if sym = NumberSym}
  id: *Identifier*; {string for identifier if sym = IdentSym}
  error: *Boolean*; {whether an error has occurred so far}

procedure *Mark* (*msg*: **string**);

procedure *GetSym*;

implementation

⟨*scanner implementation*⟩

The scanner procedure *GetSym* reads the next symbol into the global variable *sym* of type *Symbol*. The values of *Symbol* are the Pascal0 symbols, *EofSym* for end of the source file, or *null* when no symbol is recognized. Initially *sym* is set to the first symbol of the input. The *error* variable signals that an error has occurred. It is initially *false* and set to *true* by the first call to *Mark*. Procedure *Mark* prints an error message together with the line and position at which the error occured; in order to avoid cascading error messages, at most one message is printed at each position. All variables are only to be read by other modules.

For the implementation we anticipate the use of auxiliary procedures:

⟨*scanner implementation*⟩ ≡
  const
    ⟨*scanner constants*⟩
  type
    ⟨*scanner types*⟩
  var
    ⟨*scanner variables*⟩
  ⟨*scanner procedures*⟩
  ⟨*procedure Mark*⟩
  ⟨*procedure GetSym*⟩

```
begin
  ⟨scanner initialization⟩
end.
```

Procedure *GetSym* keeps reading the next character, *ch*, from file *source* through *GetChar* for recognizing the next symbol. The procedure first skips any white space: any characters in the ASCII characters set below the space, like tab and newline, are considered white space.

```
⟨procedure GetSym⟩ ≡
procedure GetSym;
begin {first skip white space}
  while not eof (source) and (ch <= '␣') do GetChar;
  if eof (source) then sym := EofSym
  else
    case ch of
      '*': begin GetChar; sym := TimesSym end;
      '+': begin GetChar; sym := PlusSym end;
      '-': begin GetChar; sym := MinusSym end;
      '=': begin GetChar; sym := EqlSym end;
      '<': begin GetChar;
              if ch = '=' then
                begin GetChar; sym := LeqSym end
              else if ch = '>' then
                begin GetChar; sym := NeqSym end
              else sym := LssSym
            end;
      '>': begin GetChar;
              if ch = '=' then
                begin GetChar; sym := GeqSym end
              else sym := GtrSym
            end;
      ';': begin GetChar; sym := SemicolonSym end;
      ',': begin GetChar; sym := CommaSym end;
      ':': begin GetChar;
              if ch = '=' then
                begin GetChar; sym := BecomesSym end
              else sym := ColonSym
            end;
      '.': begin GetChar; sym := PeriodSym end;
      '(': begin GetChar; sym := LparenSym end;
      ')': begin GetChar; sym := RparenSym end;
      '[': begin GetChar; sym := LbrakSym end;
      ']': begin GetChar; sym := RbrakSym end;
```

```
      '0'..'9': Number;
      'A' .. 'Z', 'a'..'z': Ident;
      '{': begin comment; GetSym end;
    otherwise
      begin GetChar; sym := null end
    end
end;
```

Procedure *GetChar* reads the next character from the source file and updates the current line and position. It also keeps the previous line and position, which are used by procedure *Mark* for checking if an error messsage was already output at that line and position.

```
⟨scanner variables⟩ ≡
ch: char;
line, lastline, errline: integer;
pos, lastpos, errpos: integer;
```

```
⟨scanner procedures⟩ ≡
procedure GetChar;
begin
  lastpos := pos;
  if eoln (source) then begin pos := 0; line := line + 1 end
  else begin lastline:= line; pos := pos + 1 end;
  read (source, ch)
end;
```

```
⟨procedure Mark⟩ ≡
procedure Mark (msg: string);
begin
  if (lastline > errline) or (lastpos > errpos) then
    writeln ('error: line ', lastline:1, ' pos ', lastpos:1, ' ', msg);
  errline := lastline; errpos := lastpos; error := true
end;
```

```
⟨scanner initialization⟩ ≡
line := 1; lastline := 1; errline := 1;
pos := 0; lastpos := 0; errpos := 0;
error := false;
```

Procedure *Number* reads a sequence of digits and converts them to a number that is stored in *val*. When reading each digit, we also have to check for overflow, which occurs when $(val * 10) + newDigit > maxint$.

⟨*scanner procedures*⟩+ ≡

```
procedure Number;
begin val := 0; sym := NumberSym;
  repeat
    if val <= maxint − (ord (ch) − ord ('0')) div 10 then
      val := 10 ∗ val + (ord (ch) − ord ('0'))
    else
      begin Mark ('number␣too␣large'); val := 0 end;
    GetChar
  until not (ch in ['0'..'9'])
end;
```

Procedure *Ident* reads letters into variable *id* and looks up if the result is a keyword. This is done by a linear search of the array *keyTab* which contains the strings of the keywords and the symbol they represent. The entries in *keyTab* are in order of decreasing frequency of occurrence in typical Pascal programs, for speeding up the linear search.

```
⟨scanner constants⟩ ≡
KW = 20; {number of keywords}

⟨scanner types⟩ ≡
KeyTable = array [1..KW] of
  record sym: Symbol; id: Identifier end;

⟨scanner variables⟩+ ≡
keyTab: KeyTable;

⟨scanner procedures⟩+ ≡

procedure Ident;
  var len, k: integer;
begin len := 0;
  repeat
    if len < IdLen then begin len := len + 1; id[len] := ch; end;
    GetChar
  until not (ch in ['A'..'Z', 'a'..'z', '0'..'9']);
  setlength(id, len); k := 1;
  while (k <= KW) and (id <> keyTab[k].id) do k := k + 1;
  if k <= KW then sym := keyTab[k].sym else sym := IdentSym
end;

⟨scanner initialization⟩+ ≡
keyTab[1].sym := DoSym; keyTab[1].id := 'do';
keyTab[2].sym := IfSym; keyTab[2].id := 'if';
keyTab[3].sym := OfSym; keyTab[3].id := 'of';
```

```
keyTab[4].sym := OrSym; keyTab[4].id := 'or';
keyTab[5].sym := AndSym; keyTab[5].id := 'and';
keyTab[6].sym := NotSym; keyTab[6].id := 'not';
keyTab[7].sym := EndSym; keyTab[7].id := 'end';
keyTab[8].sym := ModSym; keyTab[8].id := 'mod';
keyTab[9].sym := VarSym; keyTab[9].id := 'var';
keyTab[10].sym := ElseSym; keyTab[10].id := 'else';
keyTab[11].sym := ThenSym; keyTab[11].id := 'then';
keyTab[12].sym := TypeSym; keyTab[12].id := 'type';
keyTab[13].sym := ArraySym; keyTab[13].id := 'array';
keyTab[14].sym := BeginSym; keyTab[14].id := 'begin';
keyTab[15].sym := ConstSym; keyTab[15].id := 'const';
keyTab[16].sym := WhileSym; keyTab[16].id := 'while';
keyTab[17].sym := RecordSym; keyTab[17].id := 'record';
keyTab[18].sym := ProcedureSym; keyTab[18].id := 'procedure';
keyTab[19].sym := DivSym; keyTab[19].id := 'div';
keyTab[20].sym := ProgramSym; keyTab[20].id := 'program';
```

Procedure *comment* scans passed Pascal comments.

⟨*scanner procedures*⟩+ ≡

```
procedure comment;
begin GetChar;
  while (not eof (source)) and (ch <> '}') do GetChar;
  if eof (source) then Mark ('comment␣not␣terminated')
  else GetChar;
end;
```

Opening source file ...

⟨*scanner variables*⟩+ ≡

```
fn: string[255]; {name of source file}
source: text; {source file}
```

⟨*scanner initialization*⟩+ ≡

```
if paramcount > 0 then
  begin fn := paramstr (1); assign (source, fn); reset (source);
    GetChar
  end
else Mark ('name␣of␣source␣file␣expected')
```

4 Symbol Table

```
⟨symboltable.pas⟩ ≡
unit symboltable;
interface

uses scanner;

  type
    Class = (HeadClass, VarClass, ParClass, ConstClass, FieldClass, TypeClass,
      ProcClass, SProcClass, RegClass, EmitClass, CondClass);
    Form = (Bool, Int, Arry, Rcrd);

    Objct = ^ObjDesc;
    Typ = ^TypeDesc;

    Item = record
      mode: Class;
      lev: integer;
      tp: Typ;
      a, b, c, r, o: integer;
      indirect: boolean; {requires indirect addressing?}
      parSize: integer {parameter size, if procedure}
    end;

    ObjDesc = record
      cls: Class;
      lev: integer;
      next, dsc: Objct;
      tp: Typ;
      name: Identifier;
      val: integer;
      isAParam: boolean;
      parSize: integer
    end;

    TypeDesc = record
      form: Form;
      fields: Objct; {for records}
      base: Typ; {for arrays}
      lower, size, len: integer {for arrays}
    end;

  var
```

topScope: *Objct*; {current scope, where search for an identifier starts}
guard: *Objct*; {topScope and universe are linked lists, end with guard}
boolType, *intType*: *Typ*; {predefined types}

procedure *NewObj* (**var** *obj*: *Objct*; *cls*: *Class*);
procedure *Find* (**var** *obj*: *Objct*);
procedure *FindField* (**var** *obj*: *Objct*; *list*: *Objct*);
function *IsParam* (*obj*: *Objct*): *boolean*;
procedure *OpenScope*;
procedure *CloseScope*;
procedure *PreDef* (*cl*: *Class*; *n*: *integer*; *name*: *Identifier*; *tp*: *Typ*);

implementation
⟨*symboltable implementation*⟩

The *NewObj* procedure adds a new object to the linked list rooted at *topScope*, whereas the *Find* procedure looks up the implicit parameter *id* in the stack of linked lists. Procedure *FindField* searches an field list of a record type for the implicit *id* parameter, and returns its associated object. The *IsParam* function returns *true* if the object passed to it represents a parameter of a procedure; *false* otherwise. Procedures *OpenScope* and *CloseScope* open and close new levels (scopes) in the symbol table. Finally, procedure *PreDef* is used to add predefined identifiers to the beginning of the list at *TopScope*.

⟨*symboltable implementation*⟩ ≡
var
⟨*symboltable variables*⟩
⟨*procedure NewObj*⟩
⟨*procedure Find*⟩
⟨*procedure FindField*⟩
⟨*function IsParam*⟩
⟨*procedure OpenScope*⟩
⟨*procedure CloseScope*⟩
⟨*procedure PreDef*⟩
begin
⟨*symboltable initialization*⟩
end.

NewObj looks for an object whose name is in the implicit parameter *id*. If it doesn't find one, an object with that name can be added; if it does, it is a Multiple Definition error. A guard is used to signal the end of the list:

⟨*symboltable initialization*⟩ ≡
new (*guard*); *guard*ˆ.*cls* := *VarClass*; *guard*ˆ.*tp* := *intType*; *guard*ˆ.*val* := 0;

⟨*procedure NewObj*⟩ ≡

```
procedure NewObj (var obj: Objct; cls: Class);
  var n, x: Objct;
begin x := topScope; guard^.name := id; {set sentinel for search}
  while x^.next^.name <> id do x := x^.next;
  if x^.next = guard then
    begin
      new (n); n^.name := id; n^.cls := cls; n^.next := guard;
      x^.next := n; obj := n
    end
  else begin obj := x^.next; Mark ('mult␣def') end
end;
```

Find is the counterpart of *NewObj*, as it locates the name of an object entered by *NewObj*. However, *Find* must continue searching lower levels until some stop condition is met. This will be handled by an outermost scope called *universe*.

⟨*symboltable variables*⟩ ≡

```
universe: Objct; {final scope with only predefined identifiers}
```

⟨*procedure Find*⟩ ≡

```
procedure Find (var obj: Objct);
  var s, x: Objct;
begin s := topScope; guard^.name := id;
  while true do
    begin x := s^.next;
      while x^.name <> id do x := x^.next;
      if x <> guard then begin obj := x; break end;
      if s = universe then
        begin obj := x; Mark ('undef'); break end;
      s := s^.dsc
    end
end;
```

FindField is similar to *Find*, except that it only searches one linked list, which is passed in *list*. It is used to search the list of fields of a record.

⟨*procedure FindField*⟩ ≡

```
procedure FindField (var obj: Objct; list: Objct);
begin guard^.name := id;
  while list^.name <> id do list := list^.next;
  obj := list
end;
```

The *IsParam* function:

⟨*function IsParam*⟩ ≡

```
function IsParam (obj: Objct): boolean;
begin IsParam := obj^.isAParam
end;
```

⟨*procedure OpenScope*⟩ ≡

```
procedure OpenScope;
  var s: Objct;
begin new (s); s^.cls := HeadClass; s^.dsc := topScope;
  s^.next := guard; topScope := s
end;
```

⟨*procedure CloseScope*⟩ ≡

```
procedure CloseScope;
begin topScope := topScope^.dsc
end;
```

Initializing the symbol table requires that *universe* points to the outermost (and only) scope:

⟨*symboltable initialization*⟩+ ≡

```
topScope := nil; OpenScope; universe := topScope
```

Procedure *PreDef* accepts parameters of default identifiers in the symbol table, and adds them to the outermost scope.

⟨*procedure PreDef*⟩ ≡

```
procedure PreDef (cl: Class; n: integer; name: Identifier; tp: Typ);
  var obj: Objct;
begin new (obj);
  obj^.cls := cl; obj^.val := n; obj^.name := name;
  obj^.tp := tp; obj^.dsc := nil;
  obj^.next := topScope^.next; topScope^.next := obj
end;
```

5 Stack Architecture Description

The stack architecture is a *Virtual Machine* implemented as a Pascal module which *interprets* the code. The architecture borrows concepts from the Java Virtual Machine, in its use of many no-operand instructions (see Appendix I). Like JVM, it has different instructions for pushing and popping values on or off the stack, based on whether the address being supplied is for a global or local variable. This precludes the possibility of having nested procedures with variables at intermediate levels, since there would be no way to access them. The parser therefore emits an error if a nested procedure is encountered.

5.1 Machine Components

- IR: instruction register - holds current instruction. 32 bits wide.

- PC: program counter - holds word address of next instruction. 32 bits wide.
- ST: 32-bit memory space, where values are pushed and popped
- FP: base register - points to base of current stack frame
- SP: top register - points to current top-of-stack
- M: 32-bit words, byte addressable, of main memory

5.2 Instruction Formats

Most instructions are just 8-bits (one byte) long; they are 0-op (*ADD*, *SUB*, *CMP*). The remainder are one-op instructions (*APUSH*, *APOP*), and require two extra bytes for the argument. Note that Freepascal decides to treat *integer* as only 16 bits on some architectures, so *longint* is used where 32 bits are necessary. GPC treats *longint* as *at least* 32 bits, which allows for compatibility between the compilers.

The stack interpreter uses a code array of bytes, which efficiently packs operators (and, when appropriate, operands). As not to get carried away, the memory for the stack is word-addressible (by 32-bits) to simplify it's manipulation.

5.3 Operators

Let *arg* represent the operand passed to an instruction, if any.

Arithmetic and Comparison—No Operand

Instruction	Action
ADD	SP := SP + 1; M[SP] := M[SP] + M[SP - 1]
SUB, CMP	SP := SP + 1; M[SP] := M[SP] - M[SP - 1]
MUL	SP := SP + 1; M[SP] := M[SP] * M[SP - 1]
DIV	SP := SP + 1; M[SP] := M[SP] div M[SP - 1]
MOD	SP := SP + 1; M[SP] := M[SP] mod M[SP - 1]
AND	SP := SP + 1; M[SP] := M[SP] and M[SP - 1]
OR	SP := SP + 1; M[SP] := M[SP] or M[SP - 1]
XOR	SP := SP + 1; M[SP] := M[SP] xor M[SP - 1]
CHK	SP := SP + 2; trap if (M[SP - 1] $<$ 0) or (M[SP] $\geq$ M[SP - 2])

Branching—One Operand

Instruction	Action
BEQ	if M[SP] = 0 then nxt := PC + arg; SP := SP + 1
BNE	if M[SP] $\neq$ 0 then nxt := PC + arg; SP := SP + 1
BLT	if M[SP] < 0 then nxt := PC + arg; SP := SP + 1
BGE	if M[SP] $\geq$ 0 then nxt := PC + arg; SP := SP + 1
BLE	if M[SP] $\leq$ 0 then nxt := PC + arg; SP := SP + 1
BGT	if M[SP] > 0 then nxt := PC + arg; SP := SP + 1

Push and Pop—One Operand

Variable *glob* is the top of the global activation record; *FP* is the frame pointer.

Instruction	Action
IPUSH	SP := SP - 1; M[SP] := arg
ABSADRG, ABSADRL	M[SP] := absolute address of M[SP]
APUSHG	M[SP] := M[(glob - (M[SP] div 4))]
APUSHL	M[SP] := M[(fp - (M[SP] div 4))]
AAPUSH	M[SP] := value at M[SP]
APOPG	M[(glob - (M[SP + 1] div 4))] := M[SP]; SP := SP + 2
APOPL	M[(fp - (M[SP + 1] div 4))] := M[SP]; SP := SP + 2
AAPOP	M[M[SP + 1]] := M[SP]; SP := SP + 2

Other Instructions

Instruction	Action
INT	SP := SP - arg
DUP	SP := SP - 1; M[SP] := M[SP + 1]
RD	SP := SP - 1; read (M[SP])
WRD	write (' ', M[SP]); SP := SP + 1
WRL	writeln

Blk and Ret

The *BLK* instruction is a one-argument instruction, used to set up a new execution block. Its argument gives the size of the parameter section of the procedure, in bytes. At the bottom of the stack, it expects the return address from the procedure; at the top of the procedure space, it expects two words of data it can use for storing the *Base Address* and *Return Address*, the latter copied from the top of stack. It first stores the old base address, then the return address. It then sets the new base address to the top of the procedure block.

The *RET* operator undoes the work of *BLK*: it sets the top-of-stack to point to the top of the procedure block, and copies out the return and base addresses.

6 Stack Interpreter

⟨*stack.pas*⟩ ≡
unit *stack*;
interface

const
$ADDOP = 0$; $SUBOP = 1$; $MULOP = 2$; $DIVOP = 3$; $MODOP = 4$; $CMPOP = 5$;
$OROP = 6$; $ANDOP = 7$; $XOROP = 8$; $LSHOP = 9$; $ASHOP = 10$;
$CHKOP = 11$; $APUSHGOP = 12$; $IPUSHOP = 13$; $APOPGOP = 14$;
$BEQOP = 15$; $BNEOP = 16$; $BLTOP = 17$; $BGEOP = 18$; $BLEOP = 19$; $BGTOP = 20$;
$BLKOP = 21$; $RETOP = 22$; $RDOP = 23$; $WRDOP = 24$; $WRLOP = 25$; $INTOP = 26$;
$DUPOP = 27$; $ABSADRGOP = 28$; $AAPUSHOP = 29$; $AAPOPOP = 30$; $SWAPOP = 31$;
$APUSHLOP = 32$; $APOPLOP = 33$; $ABSADRLOP = 34$;

$MemSize = 4096$; {in bytes}

type *codeMem* = **array** [0 .. *MemSize*] **of** *byte*; {byte−addressed}
stackMem = **array** [0 .. *MemSize*] **of** *longint*; {word−addressed}

procedure *Execute* (*pc0*: *longint*);

procedure *LoadCode* (**var** *code*: *codeMem*; *len*: *longint*);

implementation
⟨*stack implementation*⟩

The *LoadCode* procedure loads the code created by the generator, into code array *M*. The *Execute* procedure interprets the afforementioned code, and follows closely from the description of the stack architecture above.

⟨*stack implementation*⟩ ≡
var
⟨*stack variables*⟩
⟨*stack procedures*⟩
⟨*procedure LoadCode*⟩
⟨*procedure Execute*⟩
end.

Loading the code is a simple array copy...

⟨*procedure LoadCode*⟩ ≡
procedure *LoadCode* (**var** *code*: *codeMem*; *len*: *longint*);
var *i*: *longint*;
begin $i := 0$;

```
  while i < len do
    begin M[i] := code[i]; i := i + 1 end
end;
```

To interpret the code, the necessary machine features are first introduced.

⟨*stack variables*⟩ ≡

```
PC, FP, SP: longint; {program, base, and stack pointer registers}
IR: longint; {instruction register}
M: codeMem; ST: stackMem;
```

When executing the code, it would be nice to optionally produce an execution trace. The variables *chk_ok* and *done* are used to signal that code interpretation should hault, because of a problem, or because the program has terminated. Normal program termination is detected when a *RET* is encountered at top-of-stack. The procedure requires a parameter indicating the *word* to begin execution at. Notice how the *Absolute Address* operators work. They store the global frame pointer, or current (local frame pointer), whichever is appropriate, in the stack location along with the address that was already there. This allows the address to be increased, but leaving the base-FP of the address intact, for retrieval by *AAPOP* and *AAPUSH*.

⟨*procedure Execute*⟩ ≡

```
procedure Execute (pc0: longint);
  var opc, arg1, nxt, topProc, temp, glob, ext1, ext2, atSP, atSP1: longint;
    p1, p2: longint; {for extracting operands}
    done: boolean;
begin
  PC := pc0 div 4;
  SP := MemSize − 1;
  FP := SP − 1;
  glob := FP;
  done := false;
  repeat
    if paramcount >= 3 then State;
    IR := M[PC]; opc := IR;
    nxt := PC + 1;
    p1 := M[PC + 1] and $000000FF;
    p2 := (M[PC + 2] shl 8) and $0000FF00;
    arg1 := (p1 + p2) and $0000FFFF;
    if arg1 >= $8000 then arg1 := arg1 − $10000; {sign extension}
    ext1 := (ST[SP] shr 16) and $0000FFFF;
    atSP := ST[SP] and $0000FFFF;
    ext2 := (ST[SP+1] shr 16) and $0000FFFF;
    atSP1 := ST[SP+1] and $0000FFFF;
    if opc in [IPUSHOP, BLTOP, BLEOP, BNEOP, BEQOP, BGTOP, BGEOP, INTOP,
```

```
    BLKOP] then nxt := PC + 3; {skip operand}
  case opc of
    ADDOP: begin SP := SP + 1; ST[SP] := ST[SP] + ST[SP − 1] end;
    SUBOP, CMPOP: begin SP := SP + 1; ST[SP] := ST[SP] − ST[SP − 1] end;
    MULOP: begin SP := SP + 1; ST[SP] := ST[SP] ∗ ST[SP − 1] end;
    DIVOP: begin SP := SP + 1; ST[SP] := ST[SP] div ST[SP − 1] end;
    MODOP: begin SP := SP + 1; ST[SP] := ST[SP] mod ST[SP − 1] end;
    ANDOP: begin SP := SP + 1; ST[SP] := ST[SP] and ST[SP − 1] end;
    OROP: begin SP := SP + 1; ST[SP] := ST[SP] or ST[SP − 1] end;
    XOROP: begin SP := SP + 1; ST[SP] := ST[SP] xor ST[SP − 1] end;
    CHKOP:
      begin SP := SP + 2;
        if (ST[SP − 1] < 0) or (ST[SP] >= ST[SP − 2]) then
          begin writeln ('Trap␣at␣', PC:2); done := true end;
      end;
    IPUSHOP: begin SP := SP − 1; ST[SP] := arg1 end;
    ABSADRGOP: ST[SP] := (glob shl 16) + ((ST[SP]) and $0000FFFF);
    ABSADRLOP: ST[SP] := (FP shl 16) + ((ST[SP]) and $0000FFFF);
    APUSHGOP: ST[SP] := ST[(glob − (ST[SP] div 4))];
    AAPUSHOP: ST[SP] := ST[(ext1 − (atSP div 4))];
    APOPGOP:
      begin ST[(glob − (ST[SP + 1] div 4))] := ST[SP]; SP := SP + 2 end;
    AAPOPOP:
      begin ST[(ext2 − (atSP1 div 4))] := ST[SP]; SP := SP + 2 end;
    APUSHLOP: ST[SP] := ST[(FP − (ST[SP] div 4))];
    APOPLOP:
      begin ST[(FP − (ST[SP + 1] div 4))] := ST[SP]; SP := SP + 2 end;
    BEQOP: begin if ST[SP] = 0 then nxt := PC + arg1; SP := SP + 1 end;
    BNEOP: begin if ST[SP] <> 0 then nxt := PC + arg1; SP := SP + 1 end;
    BLTOP: begin if ST[SP] < 0 then nxt := PC + arg1; SP := SP + 1 end;
    BGEOP: begin if ST[SP] >= 0 then nxt := PC + arg1; SP := SP + 1 end;
    BLEOP: begin if ST[SP] <= 0 then nxt := PC + arg1; SP := SP + 1 end;
    BGTOP: begin if ST[SP] > 0 then nxt := PC + arg1; SP := SP + 1 end;
    BLKOP:
      begin topProc := SP + (arg1 div 4) + 1;
        ST[topProc − 1] := FP; {Store base and return address}
        ST[topProc − 2] := ST[SP];
        FP := topProc
      end;
    RETOP:
      begin SP := FP;
        if SP = MemSize then done := true
```

```
        else begin nxt := ST[SP − 2]; FP := ST[SP − 1] end
      end;
    INTOP: SP := SP − arg1;
    DUPOP: begin SP := SP − 1; ST[SP] := ST[SP + 1] end;
    SWAPOP:
      begin temp := ST[SP + 1]; ST[SP + 1] := ST[SP]; ST[SP] := temp end;
    RDOP: begin SP := SP − 1; read (ST[SP]) end;
    WRDOP: begin write (' ', ST[SP]); SP := SP + 1 end;
    WRLOP: writeln
  end;
  PC := nxt
until done;
if paramcount >= 3 then State;
end;
```

The *State* procedure simply prints a snapshot of the stack architecture.

⟨*stack procedures*⟩ ≡

```
procedure State;
begin
  writeln ('PC=', PC, ' SP=', SP, ' FP=', FP, ' Top=', ST[SP], ' Over=',
    ST[SP − 1]);
end;
```

7 Stack Compiler

⟨*stackcompiler.pas*⟩ ≡

```
program stackCompiler (input, output);

uses scanner, symboltable, stackgenerator, stack;

  const
    WordSize = 4;

    {first/follow sets}
    MoreExp = [EqlSym, NeqSym, LssSym, GeqSym, LeqSym, GtrSym];
    MoreSimpleExp = [PlusSym, MinusSym, OrSym];
    MoreTerm = [TimesSym, DivSym, ModSym, AndSym];
    FirstFactor = [IdentSym, NumberSym, LparenSym, NotSym];
    FollowFactor = [TimesSym, DivSym, ModSym, AndSym, OrSym, PlusSym,
      MinusSym, EqlSym, NeqSym, LssSym, LeqSym, GtrSym, GeqSym, CommaSym,
      SemicolonSym, ThenSym, ElseSym, RparenSym, DoSym, PeriodSym, EndSym];
    DeclSyms = [ConstSym, TypeSym, VarSym, ProcedureSym];
```

```
    StrongSyms = [ConstSym, TypeSym, VarSym, ProcedureSym, WhileSym, IfSym,
      BeginSym, EofSym];
    FirstStatement = [IdentSym, IfSym, WhileSym, BeginSym];
    FollowStatement = [SemicolonSym, EndSym, ElseSym, BeginSym];
    FirstType = [IdentSym, RecordSym, ArraySym];
    FollowType = [SemicolonSym];
    FollowDecl = [BeginSym, EndSym, ProcedureSym, EofSym];
    FollowProcCall = [SemicolonSym, EndSym, ElseSym, IfSym, WhileSym];

  procedure expression (var x: Item); forward;
  procedure statement; forward;
  procedure selector (var x: Item; LeftSide: boolean); forward;

  ⟨S expressions⟩
  ⟨S statements⟩
  ⟨S declarations⟩
  ⟨S parser procedures⟩
  ⟨S compile⟩
begin
  ⟨S parser initialization⟩
end.
```

The development of the parser is broken into main components of expressions, statements, declarations, and initialization. The *compile* procedure starts the compiling, by reading a symbol and invoking a procedure to deal with the entire program.

```
⟨S compile⟩ ≡
procedure Compile;
begin GetSym; MainProgram
end;
```

The first thing to parse is the **program** statement. For this, a routine to skip over optional identifiers is useful.

```
⟨S parser procedures⟩ ≡
procedure SkipIdents; {skip optional identifiers in program clause}
begin
  if sym = IdentSym then
    begin GetSym;
      while sym = CommaSym do
        begin GetSym;
          if sym = RparenSym then break;
          if sym = IdentSym then GetSym else Mark ('ident?')
        end
    end
```

```
  else Mark ('ident?')
end;
```

Using the above, the **program** statement can be parsed. Then, declarations and procedures must be handled, followed by the main compound statement. This results in the following outline.

⟨*S parser procedures*⟩+ ≡

```
procedure MainProgram;
  var progid: Identifier; varsize: integer;
begin write ('␣␣compiling␣');
  if sym = ProgramSym then
    begin GetSym; Open; OpenScope; varsize := 16;
      if sym = IdentSym then
        begin progid := id; GetSym; writeln (progid) end
      else Mark ('ident?');
      if sym = LparenSym then
        begin GetSym; SkipIdents;
          if sym = RparenSym then GetSym else Mark (')?␣')
        end;
      if sym = SemicolonSym then GetSym else Mark (';?');
      declarations (varsize);
      while sym = ProcedureSym do
        begin ProcedureDecl;
          if sym = SemicolonSym then GetSym else Mark (';?')
        end;
      Header (varsize);
      if sym = BeginSym then begin GetSym; CompoundStatement end;
      if sym = EndSym then GetSym else Mark ('end?');
      if sym <> PeriodSym then Mark ('.␣?');
      CloseScope;
      if not error then
        begin Close; writeln ('␣␣code␣generated', pc :6) end
    end
  else Mark ('program?')
end;
```

Following the **program** statement are the type, variable, and procedure declarations. In Pascal, identifiers can be seperated by a comma, in order to facilitate simple setting of types, and so an accessory procedure will be defined for this.

⟨*S declarations*⟩ ≡
⟨*S IdentList*⟩
⟨*S TypeDecl*⟩

⟨*S other Declarations*⟩
⟨*S ProcedureDecl*⟩

The *IdentList* procedure simply scans a set of comma-delimited identifiers, and stops once it reaches a :, after which its type should be found.

⟨*S IdentList*⟩ ≡

```
procedure IdentList (cls: Class; var first: Objct);
  var obj: Objct;
begin
  if sym = IdentSym then
    NewObj (first, cls); GetSym;
    while sym = CommaSym do
      begin GetSym;
        if sym = IdentSym then
          begin NewObj (obj, cls); GetSym end
        else Mark ('ident?')
      end;
    if sym = ColonSym then GetSym else Mark (':?')
end;
```

TypeDecl parses the type of the preceeding variable definitions. There are three possibilities. If the type is a named type, previously defined, it is looked up in the symbol table and assigned to the type. If the type is an array, the lower and upper indices are extracted and checked, and the procedure is called recursively for the base type. For records, identifiers are read, and their associated types are assigned to them.

⟨*S TypeDecl*⟩ ≡

```
procedure TypeDecl (var t: Typ);
  var obj, first: Objct; x, y: Item; tp: Typ;
begin t := intType; {sync}
  if not (sym in FirstType) then
    begin Mark ('type?');
      repeat GetSym until sym in FirstType + FollowType + StrongSyms
    end;
  if sym = IdentSym then
    begin Find (obj); GetSym;
      if obj^.cls = TypeClass then t := obj^.tp else Mark ('type?')
    end
  else if sym = ArraySym then
    begin GetSym;
      if sym = LbrakSym then GetSym else Mark ('[?');
      expression (x); {lower bound}
      if x.mode <> ConstClass then Mark ('bad␣index');
      if sym = PeriodSym then GetSym else Mark ('.?');
```

```
        if sym = PeriodSym then GetSym else Mark ('.?');
        expression (y); {upper bound}
        if (y.mode <> ConstClass) or (y.a < x.a) then Mark ('bad␣index');
        if sym = RbrakSym then GetSym else Mark (']?');
        if sym = OfSym then GetSym else Mark ('of?');
        TypeDecl (tp); new (t); t^.form := Arry; t^.base := tp;
        t^.lower := x.a; t^.len := (y.a − x.a) + 1; t^.size := t^.len ∗ tp^.size
      end
  else if sym = RecordSym then
      begin GetSym;
        new (t); t^.form := Rcrd; t^.size := 0; OpenScope;
        repeat
            if sym = IdentSym then
              begin IdentList (FieldClass, first); TypeDecl (tp); obj := first;
                while obj <> guard do
                  begin obj^.tp := tp; obj^.val := t^.size;
                    t^.size := t^.size + obj^.tp^.size; obj := obj^.next
                  end
              end;
            if sym = SemicolonSym then GetSym
            else if sym = IdentSym then Mark (';␣?')
        until not (sym in [SemicolonSym, IdentSym]);
        t^.fields := topScope^.next; CloseScope;
        if sym = EndSym then GetSym else Mark ('end?')
      end
  else Mark ('ident?')
end;
```

Parsing constant, variable, and type declarations...

```
⟨S other Declarations⟩ ≡
procedure declarations (var varsize: integer);
  var obj, first: Objct;
    x: Item; tp: Typ;
begin {sync}
  if not (sym in DeclSyms + FollowDecl) then
    begin Mark ('dah␣declaration?');
      repeat GetSym until sym in DeclSyms + FollowDecl
    end;
  repeat
    if sym = ConstSym then
      begin GetSym;
        while sym = IdentSym do
```

```
            begin NewObj (obj, ConstClass); GetSym;
              if sym = EqlSym then GetSym else Mark ('=?');
              expression (x);
              if x.mode = ConstClass then
                begin obj^.val := x.a; obj^.tp := x.tp end
              else Mark ('expression␣not␣constant');
              if sym = SemicolonSym then GetSym else Mark (';?')
            end
        end;
    if sym = TypeSym then
        begin GetSym;
          while sym = IdentSym do
            begin NewObj (obj, TypeClass); GetSym;
              if sym = EqlSym then GetSym else Mark ('=?');
              TypeDecl (obj^.tp);
              if sym = SemicolonSym then GetSym else Mark (';?')
            end
        end;
    if sym = VarSym then
        begin GetSym;
          while sym = IdentSym do
            begin IdentList (VarClass, first); TypeDecl (tp); obj := first;
              while obj <> guard do
                begin obj^.tp := tp; obj^.lev := curlev;
                  obj^.val := varsize; obj^.isAParam := false;
                  varsize := varsize + obj^.tp^.size;
                  obj := obj^.next;
                end;
              if sym = SemicolonSym then GetSym else Mark (';␣?')
            end
        end;
    if sym in [ConstSym, TypeSym, VarSym] then
        Mark ('illegal␣declaration␣order')
  until not (sym in [ConstSym, TypeSym, VarSym])
end;
```

ProcedureDecl parses a complete procedure, including nested procedures. It keeps track of the parameter size, and local variable size, storing offset addresses in the symbol table as it goes. Auxiliary procedure FPSection is used to scan parameter sections in the procedure's declaration. To make identifiers in the procedure local, a new scope is started in the symbol table.

⟨*S ProcedureDecl*⟩ ≡

```
procedure ProcedureDecl;
  var proc, obj: Objct;
    locblksize, parblksize: integer;

  procedure FPSection;
    var obj, first: Objct; tp: Typ;
  begin
    if sym = VarSym then
      begin GetSym; IdentList (ParClass, first) end
    else IdentList (VarClass, first);
    if sym = IdentSym then
      begin Find (obj); GetSym;
        if obj^.cls = TypeClass then tp := obj^.tp
        else begin Mark ('type?'); tp := intType end
      end
    else begin Mark ('ident?'); tp := intType end;
    if first^.cls = VarClass then
        if tp^.form in [Arry, Rcrd] then Mark ('no␣struct␣params');
    obj := first;
    while obj <> guard do
      begin obj^.tp := tp;
        obj := obj^.next
      end
  end;

begin { ProcedureDecl }
  GetSym;
  if sym = IdentSym then
    begin
      NewObj (proc, ProcClass); GetSym; parblksize := 0;
      IncLevel (1); OpenScope; proc^.val := −1; proc^.lev := curlev;
      if sym = LparenSym then
        begin GetSym;
          if sym = RparenSym then GetSym
          else
            begin FPSection;
              while sym = SemicolonSym do
                begin GetSym; FPSection end;
              if sym = RparenSym then GetSym else Mark (')?')
            end
        end;
      obj := topScope^.next; parblksize := 8;
```

```
      while obj <> guard do
        begin
          obj^.lev := curlev;
          if obj^.cls = ParClass then parblksize := parblksize + WordSize
          else parblksize := parblksize + obj^.tp^.size;
          obj^.val := parblksize; obj^.isAParam := true; obj := obj^.next
        end;
       proc^.parSize := parblksize;
      proc^.dsc := topScope^.next;
      if sym = SemicolonSym then GetSym else Mark (';?');
      parblksize := parblksize + 4;
      locblksize := parblksize;
      declarations (locblksize);
      while sym = ProcedureSym do
        begin ProcedureDecl;
          if sym = SemicolonSym then GetSym else Mark (';?')
        end;
      proc^.val := pc; Enter (locblksize − parblksize);
      if sym = BeginSym then begin GetSym; CompoundStatement end;
      if sym = EndSym then GetSym else Mark ('end?');
      Return; CloseScope; IncLevel (−1)
    end
end;
```

Expressions are broken down into simple expressions (i.e. those with no relational operators), terms, and factors. A standard application of the EBNF rules yields the necessary procedures. Parameters are also defined, which are simply expressions that are passed to a *parameter* procedure in the generator. *Factor* calls *Selector*, which indexes arrays and updates address values for records.

⟨*S expressions*⟩ ≡

```
procedure factor (var x: Item);
  var obj: Objct;
begin {sync}
  if not (sym in FirstFactor) then
    begin Mark ('factor?');
      repeat GetSym until sym in FirstFactor + StrongSyms + FollowFactor
    end;
  if sym = IdentSym then
    begin Find (obj); GetSym; MakeItem (x, obj); selector (x, false);
      if x.mode <> ConstClass then LoadItem (x, false)
    end
  else if sym = NumberSym then
```

```
    begin MakeConstItem (x, intType, val); GetSym end
  else if sym = LparenSym then
    begin
      GetSym; expression (x);
      if sym = RparenSym then GetSym else Mark (')?')
    end
  else if sym = NotSym then
    begin GetSym; factor (x); Op1 (NotSym, x) end
  else begin Mark ('factor?'); MakeItem (x, guard) end
end;

procedure term (var x: Item);
  var y: Item; op: Symbol;
begin factor (x);
  while sym in MoreTerm do
    begin
      op := sym; GetSym;
      if op = AndSym then Op1 (op, x);
      factor (y); Op2 (op, x, y)
    end
end;

procedure SimpleExpression (var x: Item);
  var y: Item; op: Symbol;
begin
  if sym = PlusSym then begin GetSym; term (x) end
  else if sym = MinusSym then
    begin GetSym; term (x); Op1 (MinusSym, x) end
  else term (x);
  while sym in MoreSimpleExp do
    begin op := sym; GetSym;
      if op = OrSym then Op1 (op, x);
      term (y); Op2 (op, x, y)
    end
end;

procedure expression (var x: Item);
  var y: Item; op: Symbol;
begin
  SimpleExpression (x);
  if sym in MoreExp then
    begin op := sym; GetSym;
```

```
      SimpleExpression (y); Relation (op, x, y)
    end;
end;

procedure param (var fp: Objct);
  var x: Item;
begin
  expression (x);
  if IsParam (fp) then
    begin Parameter (x, fp^.tp, fp^.cls); fp := fp^.next end
  else Mark ('too many parameters')
end;
```

Statements are handled via two *mutually recursive* procedures, one for dealing with statements, and one for dealing with compound statements. Of course, a compound statement is a statement, hence the interaction.

⟨*S statements*⟩ ≡
⟨*S compound statement*⟩
⟨*S regular statement*⟩

Compound statements can be empty, in which case *statement* should not be called. Otherwise, the compound statement contains statement, and *statement* is called until an **end** is encountered in the source.

⟨*S compound statement*⟩ ≡

```
procedure CompoundStatement;
begin
  if sym <> EndSym then {don't try when CompoundStatement is empty}
    begin statement;
      while sym <> EndSym do
        begin
          if sym = SemicolonSym then {skip}
            repeat GetSym until sym <> SemicolonSym
          else Mark (';?');
          if sym <> EndSym then statement;
          if sym in DeclSyms then
            begin Mark ('end?'); break end
        end;
    end;
end;
```

Statements include the assignment, procedure call, if, while, and compound statements. When an identifier starts a statement, it may be a procedure call, or the start of an assignment statement. Resolving this is done via context-dependencies, found in the symbol table.

The *Selector* procedure is called again, but this time it must leave an address to assign to as a result (CF: procedure factor).

⟨*S regular statement*⟩ ≡

```
procedure statement;
  var par, obj: Objct; x, y: Item; L: integer;

  procedure sparam (var x: Item; WhichCall: integer);
  begin
    if sym = LparenSym then GetSym else Mark ('(?');
    expression (x);
    if WhichCall = 1 then pc := pc − 1;
    if sym = RparenSym then GetSym else Mark (')?')
  end;

begin { statement }
  obj := guard; {sync}
  if not (sym in FirstStatement) then
    begin Mark ('statement?');
      repeat GetSym
      until sym in FirstStatement + StrongSyms + FollowStatement
     end;
  if sym = IdentSym then
    begin Find (obj); GetSym; MakeItem (x, obj);
      if x.mode in [VarClass, ParClass, FieldClass] then
        begin selector (x, true);
          if sym = BecomesSym then
            begin GetSym; expression (y); Store (x, y) end
          else if sym = EqlSym then
            begin Mark (':=␣?'); GetSym; expression (y) end
          else Mark (':=?')
        end
      else if obj^.cls = ProcClass then
        begin LeaveRoom; par := obj^.dsc;
          if sym = LparenSym then
            begin GetSym;
              if sym = RparenSym then GetSym
              else
                while true do
                  begin param (par);
                    if sym = CommaSym then GetSym
                    else if sym = RparenSym then
                      begin GetSym; break end
```

```
                    else if sym in FollowProcCall + StrongSyms then break
                    else Mark (')␣or␣,␣?')
                  end
              end;
            if obj^.val < 0 then Mark ('forward␣call')
            else if not IsParam (par) then Call (x)
            else Mark ('too␣few␣parameters')
          end
        else if obj^.cls = SProcClass then
          begin MakeItem (x, obj);
            if obj^.val <= 2 then sparam (y, obj^.val);
            IOCall (x, y)
          end
        else Mark ('invalid␣assignment␣or␣statement')
      end
  else if sym = IfSym then
    begin GetSym; expression (x); CJump (x);
      if sym = ThenSym then GetSym else Mark ('then?');
      statement; L := 0;
      if sym = ElseSym then
        begin GetSym; FJump (L); FixLink (x.a); statement end
      else FixLink (x.a);
      FixLink (L)
    end
  else if sym = WhileSym then
    begin GetSym; L := pc; expression (x); CJump (x);
      if sym = DoSym then GetSym else Mark ('do␣?');
      statement; BJump (L); FixLink (x.a)
    end
  else if sym = BeginSym then
    begin GetSym; CompoundStatement;
      if sym = EndSym then GetSym else Mark ('end?')
    end;
end;
```

Selector can now be written. As evident in *factor* and *statement*, it must take a flag indicating whether it should leave an address, or the value at that address, as its result.

⟨*S regular statement*⟩+ ≡

```
procedure selector (var x: Item; LeftSide: boolean);
  var y: Item; obj: Objct;
begin
```

```
  while (sym = LbrakSym) or (sym = PeriodSym) do
    if sym = LbrakSym then begin
      if x.indirect = false then
        begin x.indirect := true; PlaceHolder end;
      GetSym; expression (y);
      if x.tp^.form = Arry then Index (x, y) else Mark ('not␣an␣array');
      if sym = RbrakSym then GetSym else Mark (']?')
      end
    else
      begin GetSym;
        if sym = IdentSym then
          if x.tp^.form = Rcrd then
            begin FindField (obj, x.tp^.fields); GetSym;
              if obj <> guard then Field (x, obj) else Mark ('undef')
            end
          else Mark ('not␣a␣record')
        else Mark ('ident?')
      end;
  if LeftSide then LoadItem (x, true)
end;
```

The final step in the development of the parser is its initialization. This mainly consists of adding predefined identifiers to the symbol table.

⟨*S parser procedures*⟩+ ≡

```
procedure Init;
  begin writeln ('Pascal0␣Compiler');
    OpenScope;
    PreDef (TypeClass, 1, 'boolean', boolType);
    PreDef (TypeClass, 2, 'integer', intType);
    PreDef (ConstClass, 1, 'true', boolType);
    PreDef (ConstClass, 0, 'false', boolType);
    PreDef (SProcClass, 1, 'read', nil);
    PreDef (SProcClass, 2, 'write', nil);
    PreDef (SProcClass, 3, 'writeln', nil);
  end;
```

⟨*S parser initialization*⟩ ≡

```
Init; Compile; if not error then Load
```

8 Stack Generator

⟨*stackgenerator.pas*⟩ ≡

```
unit stackgenerator;

interface

uses scanner, symboltable, stack;

  const OpSize = 2; {operand size}

  var curlev, pc: integer;

  procedure LeaveRoom;

  procedure PlaceHolder;

  procedure LoadItem (var x: Item; LeaveAddress: boolean);

  procedure FixLink (L: integer);

  procedure IncLevel (n: integer);

  procedure MakeConstItem (var x: Item; tp: Typ; val: integer);

  procedure MakeItem (var x: Item; y: Objct);

  procedure Field (var x: Item; y: Objct);

  procedure Index (var x, y: Item);

  procedure Op1 (op: Symbol; var x: Item);

  procedure Op2 (op: Symbol; var x, y: Item);

  procedure Relation (op: Symbol; var x, y: Item);

  procedure Store (var x, y: Item);

  procedure Parameter (var x: Item; ftyp: Typ; cls: Class);

  procedure CJump (var x: Item);

  procedure BJump (L: integer);
```

procedure *FJump* (**var** *L*: *integer*);

procedure *Call* (**var** *x*: *Item*);

procedure *IOCall* (**var** *x*, *y*: *Item*);

procedure *Header* (*size*: *integer*);

procedure *Enter* (*size*: *integer*);

procedure *Return*;

procedure *Open*;

procedure *Close*;

procedure *Load*;

procedure *Decode*;

implementation

⟨*S generator implementation*⟩

procedure *LeaveRoom* is used to leave space at the start of a procedure call, for the architecture to insert the base and return addresses. *PlaceHolder* is used to leave room for an array index, known only at runtime. *LoadItem* loads an item onto the stack, so that it can be used in operations. *MakeItem* and *MakeConstItem* make items representing objects in the symbol table. *Field* and *Index* are used to access records, and arrays, respectively. *Op1*, *Op2*, and *Relation* are used to generate code for a one-op instruction, two-op instruction, and relational operator, respectively. *Store* places the top of the stack in a variable, for an assignment statement. *Parameter* pushes procedure parameters on the stack. *CJump*, *BJump*, and *FJump* are used to perform conditional, backward, and forward jumps. *Call* and *IOCall* call procedures, and *Enter* and *Return* enter and exit their scopes. *Open* initializes the generator, and *Close* terminates code generation.

⟨*S generator implementation*⟩ ≡
  var
    ⟨*S generator variables*⟩
  ⟨*S generator procedures*⟩
  ⟨*S procedure LeaveRoom*⟩
  ⟨*S procedure PlaceHolder*⟩
  ⟨*S procedure LoadItem*⟩

$\langle$*S procedure FixLink*$\rangle$
$\langle$*S procedure IncLevel*$\rangle$
$\langle$*S procedure MakeConstItem*$\rangle$
$\langle$*S procedure MakeItem*$\rangle$
$\langle$*S procedure Field*$\rangle$
$\langle$*S procedure Index*$\rangle$
$\langle$*S procedure LoadBool*$\rangle$
$\langle$*S procedure PutOp*$\rangle$
$\langle$*S procedure Op1*$\rangle$
$\langle$*S procedure Op2*$\rangle$
$\langle$*S procedure Relation*$\rangle$
$\langle$*S procedure Store*$\rangle$
$\langle$*S procedure Parameter*$\rangle$
$\langle$*S procedure CJump*$\rangle$
$\langle$*S procedure BJump*$\rangle$
$\langle$*S procedure FJump*$\rangle$
$\langle$*S procedure Call*$\rangle$
$\langle$*S procedure IOCall*$\rangle$
$\langle$*S procedure Header*$\rangle$
$\langle$*S procedure Enter*$\rangle$
$\langle$*S procedure Return*$\rangle$
$\langle$*S procedure Open*$\rangle$
$\langle$*S procedure Close*$\rangle$
$\langle$*S procedure Load*$\rangle$
$\langle$*S procedure Decode*$\rangle$
begin
$\langle$*S generator initialization*$\rangle$
end.

The code will be stored in *code*.

$\langle$*S generator variables*$\rangle \equiv$
code: *codeMem*;
entry: *integer*;

An instruction to emit code is necessary. It takes the opcode, and one argument, and puts the opcode in the current location of the code array. If the operator requires an argument, it is itself stored in the following two bytes. The necessary auxiliary procedures are defined first.

$\langle$*S generator procedures*$\rangle \equiv$
procedure *putBytes* (*at*: *longint*; *a*: *longint*);
begin {store a at bytes starting from at}
code[*at* + 1] := *a* **and** *$000000FF*;
code[*at* + 2] := ((*a* **shr** 8) **and** *$000000FF*)

```
end;

function getBytes (at: integer): longint;
  var p1, p2, temp: longint;
begin
  p1 := code[at + 1] and $000000FF;
  p2 := (code[at + 2] shl 8) and $0000FF00;
  temp := (p1 + p2) and $0000FFFF;
  getBytes := temp
end;

procedure put (op, a: longint);
begin {emit instruction}
  code[pc] := op;
  if op in [IPUSHOP, BLTOP, BLEOP, BNEOP, BEQOP, BGTOP, BGEOP, INTOP, BLKOP]
  then
    begin
      a := a and $0000FFFF; putBytes (pc, a);
      pc := pc + 1 + OpSize
    end
  else pc := pc + 1
end;
```

Several trivial procedures can now be written with the assistance of *put*.

```
⟨S procedure PlaceHolder⟩ ≡
procedure PlaceHolder;
begin put (IPUSHOP, 0)
end;

⟨S procedure LeaveRoom⟩ ≡
procedure LeaveRoom;
begin
  put (IPUSHOP, 0); put (IPUSHOP, 0)
end;
```

Call, as required by the virtual machine, must place the return address on the stack before branching to the procedure.

```
⟨S procedure Call⟩ ≡
procedure Call (var x: Item);
begin
  put (IPUSHOP, pc + 4 + OpSize*4);
  put (BLKOP, x.parSize);
  put (IPUSHOP, 0);
```

```
  put (BEQOP, x.a − pc) {unconditional branch to procedure}
end;
```

Header initializes the entry, and sets up the outermost block.

⟨*S procedure Header*⟩ ≡

```
procedure Header (size: integer);
begin entry := pc;
  put (BLKOP, 0);
  put (INTOP, size)
end;
```

⟨*S procedure Enter*⟩ ≡

```
procedure Enter (size: integer);
begin put (INTOP, size)
end;
```

⟨*S procedure Return*⟩ ≡

```
procedure Return;
begin put (RETOP, 0)
end;
```

Open initializes *PC* and the procedure nesting level.

⟨*S procedure Open*⟩ ≡

```
procedure Open;
begin curlev := 0; pc := 0
end;
```

⟨*S procedure Close*⟩ ≡

```
procedure Close;
begin put (RETOP, 0)
end;
```

When loading items on the stack, it would be helpful to first test if they are in a valid range for 16-bit entities. This is necessary, because *integer* may be treated as a 32-bit value by GNU Pascal, but only a maximum of 16 bits are allowed as a component of an operator in the stack VM.

⟨*S generator procedures*⟩+ ≡

```
procedure TestRange (x: integer);
begin {16−bit entity}
  if (x >= $8000) or (x < −$8000) then Mark ('value␣too␣large')
end;
```

LoadItem checks the type of the item to load, and loads its value (or its address) on the stack. Constants are easiest: their value is tested at compile-time, it is loaded on the stack, and its class is set to *EmitClass* to denote that it is a constant that has been emitted (for use with constant folding). Variables and parameters are loaded similarly, except there is the option to leave an address instead of a value, and parameters require one indirection to get at the value stored at the address.

⟨*S procedure LoadItem*⟩ ≡

```
procedure LoadItem (var x: Item; LeaveAddress: boolean);
begin
  if x.mode = ParClass then
    begin
      put (IPUSHOP, x.a);
      if x.lev = 0 then put (APUSHGOP, 0) else put (APUSHLOP, 0);
      if x.indirect then put (ADDOP, 0);
      if x.o <> 0 then
        begin put (IPUSHOP, x.o); put (ADDOP, 0) end;
      if not LeaveAddress then put (AAPUSHOP, 0);
    end;
  if x.mode = VarClass then
    begin
      put (IPUSHOP, x.a);
      if x.indirect then put (ADDOP, 0);
      if x.o <> 0 then
        begin put (IPUSHOP, x.o); put (ADDOP, 0) end;
        if not LeaveAddress then
      if x.lev = 0 then put (APUSHGOP, 0) else put (APUSHLOP, 0);
    end
  else if x.mode = ConstClass then
    begin TestRange (x.a);
      put (IPUSHOP, x.a);
      x.mode := EmitClass
    end
end;
```

To load an item, one must have been created in the first place. This is accomplished by *MakeItem* and *MakeConstItem*. These procedures work by basically copying the relevant fields from a symbol table object.

⟨*S procedure MakeConstItem*⟩ ≡

```
procedure MakeConstItem (var x: Item; tp: Typ; val: integer);
begin
  x.mode := ConstClass; x.tp := tp; x.a := val
end;
```

⟨*S procedure MakeItem*⟩ ≡

```
procedure MakeItem (var x: Item; y: Objct);
begin
  x.mode := y^.cls; x.tp := y^.tp; x.lev := y^.lev; x.a := y^.val;
  x.indirect := false; x.parSize := y^.parSize; x.o := 0
end;
```

Determining the address of a field in a record can easily be done at compile-time:

⟨*S procedure Field*⟩ ≡

```
procedure Field (var x: Item; y: Objct); { x := x.y }
begin
  x.o := x.o + y^.val; x.tp := y^.tp
end;
```

Arrays, on the other hand, may have an expression as its index, and so cannot always be determined at compile-time. It is also necessary to subtract the lower bound from the index, to offset the array correctly in memory.

⟨*S procedure Index*⟩ ≡

```
procedure Index (var x, y: Item); { x := x[y] }
begin
  if y.tp <> intType then Mark ('index not integer');
  if y.mode = ConstClass then
    begin
      if (y.a < x.tp^.lower) or (y.a >= x.tp^.len + x.tp^.lower) then
        Mark ('bad index');
      x.o := x.o + ((y.a - x.tp^.lower) * x.tp^.base^.size)
    end
  else
    begin
      put (IPUSHOP, x.tp^.lower);
      put (SUBOP, 0);
      put (DUPOP, 0); {duplicate 0-based index}
      put (IPUSHOP, x.tp^.len);
      put (CHKOP, 0);
      put (IPUSHOP, x.tp^.base^.size);
      put (MULOP, 0);
      put (ADDOP, 0)
    end;
  x.tp := x.tp^.base
end;
```

Op1 and *Op2* are used to process operators in expressions. In order for this to work, there should be a mechanism for loading boolean values (which have been ignored thus far), and emitting the code for the operator.

```
⟨S procedure LoadBool⟩ ≡
procedure LoadBool (var x: Item);
begin
  if x.tp^.form <> Bool then Mark ('Boolean?');
  x.mode := CondClass; x.a := 0; x.b := 0; x.c := 1
end;
```

If one of the two values being compared is not loaded yet, it must first be loaded.

```
⟨S procedure PutOp⟩ ≡
procedure PutOp (cd: integer; var x, y: Item);
  var sw: boolean;
begin
  sw := (x.mode = ConstClass) and (y.mode <> ConstClass);
  if x.mode = ConstClass then begin TestRange (x.a); LoadItem (x, false) end;
  if y.mode = ConstClass then begin TestRange (y.a); LoadItem (y, false) end;
  if sw then put (SWAPOP, 0);
  put (cd, 0)
end;
```

```
⟨S procedure Op1⟩ ≡
procedure Op1 (op: Symbol; var x: Item); { x := op x }
  var t: integer;
begin
  if op = MinusSym then
    if x.tp^.form <> Int then Mark ('bad type')
    else if x.mode = ConstClass then x.a := −x.a
    else begin put (IPUSHOP, −1); put (MULOP, 0) end
  else if op = NotSym then
    begin
      if x.mode <> CondClass then LoadBool (x);
      x.c := negated (x.c); t := x.a; x.a := x.b; x.b := t
    end
  else if op = AndSym then
    begin
      if x.mode <> CondClass then LoadBool (x);
      put (BEQOP + negated (x.c), x.a);
      x.a := pc − (1 + OpSize); FixLink (x.b); x.b := 0
    end
  else if op = OrSym then
    begin
      if x.mode <> CondClass then LoadBool (x);
      put (BEQOP + x.c, x.b);
      x.b := pc − (1 + OpSize); FixLink (x.a); x.a := 0
```

```
    end
end;
```

The *Negated* function negates the condition of a boolean (for example, from < to >=). It uses the ordering of the symbols to do this efficiently, instead of a case statement for every condition.

⟨*S generator procedures*⟩+ ≡

```
function negated (cond: integer): integer;
begin
  if odd (cond) then negated := cond − 1 else negated := cond + 1
end;
```

For *Op2*, note how, if both operands are constant, no code is emitted, but the constants are folded during compile-time.

⟨*S procedure Op2*⟩ ≡

```
procedure Op2 (op: Symbol; var x, y: Item); { x := x op y }
begin
  if (x.tp^.form = Int) and (y.tp^.form = Int) then
    if (x.mode = ConstClass) and (y.mode = ConstClass) then
      if op = PlusSym then x.a := x.a + y.a
      else if op = MinusSym then x.a := x.a − y.a
      else if op = TimesSym then x.a := x.a ∗ y.a
      else if op = DivSym then x.a := x.a div y.a
      else if op = ModSym then x.a := x.a mod y.a
      else Mark ('bad␣type')
    else
      if op = PlusSym then PutOp (ADDOP, x, y)
      else if op = MinusSym then PutOp (SUBOP, x, y)
      else if op = TimesSym then PutOp (MULOP, x, y)
      else if op = DivSym then PutOp (DIVOP, x, y)
      else if op = ModSym then PutOp (MODOP, x, y)
      else Mark ('bad␣type')
  else if (x.tp^.form = Bool) and (y.tp^.form = Bool) then
    begin
      if y.mode <> CondClass then LoadBool (y);
      if op = OrSym then
        begin x.a := y.a; x.b := merged (y.b, x.b); x.c := y.c end
      else if op = AndSym then
        begin x.a := merged (y.a, x.a); x.b := y.b; x.c := y.c end
    end
  else Mark ('bad␣type')
end;
```

⟨*S generator procedures*⟩+ ≡

```
function merged (L0, L1: integer): integer;
  var L2, L3: integer;
begin
  if L0 <> 0 then
    begin
      L2 := L0;
      while true do
        begin
          L3 := code[L2] and $0000FFFF;
          if L3 = 0 then break;
          L2 := L3
        end;
      code[L2] := code[L2] + L1; merged := L0
    end
  else merged := L1
end;
```

Relation compares the top two values on the stack, represented by items x and y.

⟨*S procedure Relation*⟩ ≡

```
procedure Relation (op: Symbol; var x, y: Item); { x := x ? y }
begin
  if (x.tp^.form <> Int) or (y.tp^.form <> Int) then Mark ('bad type')
  else
    begin
      PutOp (CMPOP, x, y);
      x.c := ord (op) − ord (EqlSym)
    end;
  x.mode := CondClass; x.tp := boolType; x.a := 0; x.b := 0
end;
```

Procedure *Store* stores the value at top-of-stack into the specified variable.

⟨*S procedure Store*⟩ ≡

```
procedure Store (var x, y: Item); { x := y }
begin
  if (x.tp^.form in [Bool, Int]) and (x.tp^.form = y.tp^.form) then
    begin
      if y.mode = CondClass then
        begin
          put (BEQOP + negated (y.c), y.a);
          y.a := pc − (1 + OpSize);
          FixLink (y.b);
```

```
        put (IPUSHOP, 1); {push true}
        put (IPUSHOP, 0);
        put (BEQOP, 2 + OpSize); {skip over false assignment}
        FixLink (y.a); put (IPUSHOP, 0) {put false}
      end
    else if y.mode = ConstClass then LoadItem (y, false);
    if x.mode = VarClass then
      if x.lev = 0 then put (APOPGOP, 0) else put (APOPLOP, 0)
    else if x.mode = ParClass then
      put (AAPOPOP, 0)
    else Mark ('illegal␣assignment')
  end
  else Mark ('incompatible␣assignment')
end;
```

Parameter pushes parameters on the stack, prior to *Call* calling a procedure. If a reference parameter is to be pushed, and it is already in the form of a reference parameter (I.E. passed by reference from another procedure), all that has to be done is decrement *PC* to undo the final pop instruction emitted by *expression*, since that would leave the address on the stack. If it is a standard variable, then, additionally, the *ABSADR* instruction is emitted, to get its absolute address on the stack.

⟨*S procedure Parameter*⟩ ≡

```
procedure Parameter (var x: Item; ftyp: Typ; cls: Class);
begin
  if x.tp = ftyp then
    begin
      if cls = ParClass then {VAR parameter}
        if x.mode = VarClass then
          begin pc := pc − 1;
            if x.lev = 0 then put (ABSADRGOP, 0) else put (ABSADRLOP, 0)
          end
        else if x.mode = ParClass then pc := pc − 1
        else Mark ('illegal␣parameter␣mode')
      else { value parameter }
        if x.mode = ConstClass then LoadItem (x, false)
    end
  else Mark ('bad␣parameter␣type')
end;
```

The various jumps...

⟨*S procedure CJump*⟩ ≡

```
procedure CJump (var x: Item);
begin
```

```
  if x.tp^.form = Bool then
    begin
      if x.mode <> CondClass then LoadBool (x);
      put (BEQOP + negated(x.c), x.a);
      FixLink (x.b); x.a := pc − (1 + OpSize)
    end
  else begin Mark ('Boolean?'); x.a := pc end
end;
```

⟨*S procedure BJump*⟩ ≡

```
procedure BJump (L: integer);
begin
  put (IPUSHOP, 0); put (BEQOP, L − pc)
end;
```

⟨*S procedure FJump*⟩ ≡

```
procedure FJump (var L: integer);
begin
  put (IPUSHOP, 0);
  put (BEQOP, L); L := pc − (1 + OpSize)
end;
```

FixLink fixes a chain of code lines, ending at value 0.

⟨*S procedure FixLink*⟩ ≡

```
procedure FixLink (L: integer);
  var L1: integer;
begin
  while L <> 0 do
    begin L1 := getBytes (L); fix (L, pc − L); L := L1 end
end;
```

⟨*S generator procedures*⟩+ ≡

```
procedure fix (at, fixwith: integer);
begin putBytes (at, fixwith) end;

procedure FixWith (L0, L1: integer);
  var L2: integer;
begin
  while L0 <> 0 do
    begin L2 := code[L0] and $0000FFFF; fix (L0, L1 − L0); L0 := L2 end
end;
```

The *IOCall* procedure handles built-in procedures for reading and writing.

⟨*S procedure IOCall*⟩ ≡

```
procedure IOCall (var x, y: Item);
  var z: Item;
begin
  if x.a < 3 then
    if y.tp^.form <> Int then Mark ('Integer?');
  if x.a = 1 then {read}
    begin
      put (RDOP, 0);
      z.mode := EmitClass; z.tp := intType;
      Store (y, z)
    end
  else if x.a = 2 then {write}
    begin
      if y.mode = ConstClass then LoadItem (y, false);
      put (WRDOP, 0)
    end
  else put (WRLOP, 0) {writeln}
end;
```

Decode prints a text representation of the code. It requires the list of *mnemonics* to be available in an array.

⟨*S generator variables*⟩+ ≡

```
mnemo: array [0..35, 0..5] of char; {for decoder}
```

The variable i runs through the code array until it reaches the end, signalled by *PC*.

⟨*S procedure Decode*⟩ ≡

```
procedure Decode;
  var i, a: longint; cd: byte;
begin
  writeln ('entry', entry); i := 0;
  while i < pc do
    begin
      cd := code[i]; a := getBytes (i);
      if a >= $8000 then a := a − $10000; {sign extension}
      write (i, '    ', mnemo[(cd)]);
      if cd in [IPUSHOP, BLTOP, BLEOP, BNEOP, BEQOP, BGTOP, BGEOP, INTOP,
          BLKOP] then
          begin writeln (',', a :8); i := i + 2 end
      else writeln;
      i := i + 1
    end;
  writeln;
```

end;

Initialization of the mnemonics, and built-in types:

```
⟨S generator initialization⟩ ≡
new (boolType); boolType^.form := Bool; boolType^.size := 4;
new (intType); intType^.form := Int; intType^.size := 4;
mnemo[ADDOP] := 'ADD␣';
mnemo[SUBOP] := 'SUB␣';
mnemo[MULOP] := 'MUL␣';
mnemo[DIVOP] := 'DIV␣';
mnemo[MODOP] := 'MOD␣';
mnemo[CMPOP] := 'CMP␣';
mnemo[OROP] := 'OR␣␣';
mnemo[ANDOP] := 'AND␣';
mnemo[XOROP] := 'XOR␣';
mnemo[LSHOP] := 'LSH␣';
mnemo[ASHOP] := 'ASH␣';
mnemo[CHKOP] := 'CHK␣';
mnemo[APUSHGOP] := 'APUSHG␣';
mnemo[IPUSHOP] := 'IPUSH␣';
mnemo[APOPGOP] := 'APOPG␣';
mnemo[BEQOP] := 'BEQ␣';
mnemo[BNEOP] := 'BNE␣';
mnemo[BLTOP] := 'BLT␣';
mnemo[BGEOP] := 'BGE␣';
mnemo[BLEOP] := 'BLE␣';
mnemo[BGTOP] := 'BGT␣';
mnemo[BLKOP] := 'BLK␣';
mnemo[RETOP] := 'RET␣';
mnemo[RDOP] := 'READ';
mnemo[WRDOP] := 'WRD␣';
mnemo[WRLOP] := 'WRL␣';
mnemo[INTOP] := 'INT␣';
mnemo[DUPOP] := 'DUP␣';
mnemo[ABSADRGOP] := 'ABSADRG␣';
mnemo[ABSADRLOP] := 'ABSADRL␣';
mnemo[AAPUSHOP] := 'AAPUSH␣';
mnemo[AAPOPOP] := 'AAPOP␣';
mnemo[SWAPOP] := 'SWAP␣';
mnemo[APUSHLOP] := 'APUSHL␣';
mnemo[APOPLOP] := 'APOPL␣';
```

⟨*S procedure Load*⟩ ≡

```
procedure Load;
begin
  LoadCode (code, pc);
  writeln ('␣␣code␣loaded');
  if paramcount > 2 then Decode;
  Execute (entry)
end;
```

```
⟨S procedure IncLevel⟩ ≡
procedure IncLevel (n: integer);
begin curlev := curlev + n
end;
```

9 Appendix I: Further Machine Information and References

For information on the Java Virtual Machine, consult the reference below.

Java Virtual Machine - Online Instruction Reference. http://cat.nyu.edu/ meyer/jvmref/ref-Java.html

The java VM is stack-based, as is the Pascal stack interpreter. There are some important similarities and differences:

- They both refer to variables by number, and have different operators for referring to global and local data.
- Java VM directly supports arrays, and has operators for array load and store operations. Pascal0 stack is unaware of arrays, and so the parser must correctly index them
- Java VM has single-byte operators for loading the first four variables. To load variables up to variable 255 takes two bytes, the load/store opcode followed by the variable number. To reference variables after 255, a *wide* opcode is used to signify that two bytes represent the variable number, following the load/store instruction. Pascal0 stack simplifies this, and always uses the top-of-stack (4 bytes) for the variable number
- Java VM has built-in support for integers, longs, booleans, and other object types. Pascal0 stack supports only integers and booleans, but booleans are just represented as 1 for *true* and 0 for *false*, unknown to the architecture