Csci465
Principle of Translation

Dr. Hassan Reza

A Simple P-Code Compiler

In this handout we describe a simple compiler which translates the statement portion of Pascal-like programs into code for an imaginary stack-based machine. It is based on predictive parsing. The only data types are **Integer** and **Boolean**. There are no procedures (and hence no I/O!), but there are arrays. Not all details are described; this is mostly pseudocode.

Symbols

Ultimately, a compiler views its input program not as a stream of characters but as a stream of **symbols**, or **tokens**. A typical programming language contains symbols in some or all of the following classes:

Punctuation symbols: ( ) [ ] = , ; . := < <= + * /
Reserved words: **begin end case while do if mod var div**
Numbers (e.g. strings of digits)
Strings
Identifiers
    CONST, TYPE, VAR, PROCEDURE, FUNCTION names
    parameter names (usually several types, e.g. value & **var** params)
    record field names
    element names within an enumerated type (red, yellow, blue, green)
End-of-file and end-of-line symbols

This list is sufficient for Pascal and most other languages. Note that two-character punctuation strings such as := and <> are treated as single symbols. In general, comments do **not** appear in the list; the compiler proper does not see them. An end-of-line 'symbol' is needed only in languages such as Fortran that treat line-ends as special; Pascal and other free-form languages treat line-ends like any other white space. An end-of-file symbol is needed for the compiler to check that the program does not continue past where the compiler thinks it ends. A Pascal compiler would thus expect that the end-of-file symbol would be next after seeing the final 'end' and then '.'; anything else would be an error.

Our first problem is to **tokenize** the input; that is, convert from the original character stream to a stream of symbols. Symbols can be represented in a variety of ways. For instance, reserved words and punctuation can be represented by strings or by a large enumerated type, e.g. (assign, lessthan, greaterthan, ... , beginsym, endsym, whilesym). In what follows, we are going to take an abstract approach to tokenization compatible with any of these representations. We do this partly to maintain a clear modular design, but also because this lets us make up additional details as we go along. The basic procedure is **GETSYM**(symbol), which returns the next symbol of the input file. We suppose that variables representing all the standard symbols have been pre-defined; thus we write things like

    while symbol = semicolon do begin getsym(symbol); ... end

Note that we cannot, in Pascal, use 'while' as a variable name, because it is a reserved word; hence the use of 'whilesym' instead. The only thing we lose with our abstract approach, compared to the enumerated type approach, is that we cannot use the Pascal **set** structure; that is, we cannot write "**while** symbol **in** [plus, minus, orsym] **do**".

As equal partners in the GETSYM module, we will also suppose the existence of appropriate 'inquiry' functions and procedures to find out about the properties of various symbols. For instance, **IsIdent**(symbol) might return TRUE if the symbol in question were an identifier. More complex inquiries might be about how an identifier was declared; e.g. is it a variable or a constant, what is its type, what is its size and allocated address. For instance, we will make use of a **getaddress**(symbol, var address) to retrieve addresses.

Another concern is how we are initially to give identifiers their "properties", as must occur while processing the declaration sections of a program. For the time being, as we deal only with the statement section of programs, we simply ignore this question. Later, we will further extend our abstract GETSYM as seems appropriate.

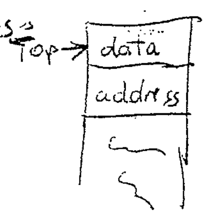
The P-machine

We will generate code for a hypothetical ("Pseudo"; the P does not stand for Pascal) machine that has two data spaces: a region where variables are stored, and a stack for evaluating expressions. Variables can be loaded to or saved from the stack, and arithmetic can be performed on the top one or two quantities on the stack (binary operations like +,-,/,* use the top two quantities; unary minus as in x:= -x would use only one. We will assume that all variables are assigned addresses as the type declarations are processed, and that this address is available from the procedure **getaddr**(symbol, **var** **address**), in which **address** is a var parameter.

A summary of simplified P-code instructions follows:

Loading and storing, between stack and variable space:

LOAD address	loads word at address to top of stack
ILOAD	"index" load: pops address from top of stack, and loads the word at that address. For arrays.
STOR address	Stores word at top of stack in address
ISTOR	pops data from stack, then pops address, then stores word at address. i.e. memory
DLOAD (number or address)	Load number or address directly to top of stack



Branching instructions:

JUMP location	unconditional jump
JTRUE loc	pop top word; jump if it is true (≠0)
JFALSE	

Arithmetic:

ADD	pop two words, push their sum
MUL	
DIV, MOD, SUB, AND, OR	
CMP =	pop two words, push 1 if equal, otherwise 0.
CMP >, >=, etc	similar
NEG	change sign of topmost word
NOT	change 1 to 0 & vice-versa

We will suppose there is a procedure EMIT(instruction, params) that takes care of the actual output of code; we will assume that, like writeln, EMIT can take a variable number of parameters. We also suppose, for the generation of labels for destinations of branch instructions, that we have a procedure

GENLABEL(**var** label) to return a unique label. To label a specific instruction, we use the SPC assembler directive LOC ⟨label⟩. We allow multiple labels to the same point; thus, LOC L1 works more like the IBM 370's

```
   L1    EQU   *
```

than like

```
   L1    ⟨instruction⟩
```

Syntax

The following EBNF productions describe the syntax of the fragment of pascal we will consider. The E of EBNF stands for Extended; the extension is the use of [ ] to enclose optional segments (parts of a production that may appear 0 or 1 times) and { } to enclose parts that may be repeated 0 or more times.

```
number   ::= digit { digit }   (thus, this is an unsigned integer number.)
variable  ::= variable-identifier [ '[' expression ']' ]
        /* e.g. a[2]       */

factor     ::= number | '(' expression ')' | 'not' factor | variable
term       ::= factor { mulop factor }
mulop      ::= '*' | 'div' | 'mod' | 'and'
simple-expr ::= [ '+' | '-' ] term { addop term }
addop      ::= '+' | '-' | 'or'
expression ::= simple-expr [ relop simple-expr ]
relop      ::= '=' | '<' | '>' | '<>' | '<=' | '>='

statement ::= assign-stmt | empty-stmt | while-stmt | if-stmt | for-stmt |
              begin-stmt
assign-stmt   ::= variable ':=' expression
empty-stmt    ::=                              /* that's it!! */
while-stmt    ::= 'while' expression 'do' statement
if-stmt       ::= 'if' expression 'then' statement [ 'else' statement ]
for-stmt    ::= 'for' identifier ':=' expression 'to' expression 'do' statement
begin-stmt ::= 'begin' statement { ';' statement } 'end'   ← we will just compile the main begin block.
```

Our approach is to define a separate procedure to handle compilation of each of these productions. These procedures, like the productions themselves, will be mutually recursive. In many cases (e.g. terms and expressions) we cannot tell that a production has ended until after we have read one symbol too far. We could define an UNgetsym that puts a symbol back into the getsym stream, but it is simpler to adopt the one-symbol-lookahead approach. At all times, the global variable **symbol** will contain the current symbol. Every compiling procedure will be called with **symbol** containing the first symbol of the corresponding production, and every procedure will exit with **symbol** containing the first symbol <u>following</u> that production. Thus, in compiling a statement, **symbol** is either whilesym, ifsym, forsym, beginsym, or is a procedure or variable identifier, or else the statement is empty. On exit from any statement-compiling routine, **symbol** must be either semicolon or endsym; these are the only two symbols that can follow statements. For instance, if CEXPR compiles expressions, we suppose it does the following:

1. It respects the one-symbol-lookahead conventions
2. The code it generates does not change or examine any existing cells on the stack.
3. The code it generates causes the value of the expression to be left on the top of the stack, one cell past the previous top-of-stack.

Here is a summary of the procedures we have:
Getsym(symbol), getaddress(symbol, var address), (*from "declarations"*)
Functions: IsIdent(symbol), IsVariable(symbol), IsArray(sym)

type checking

Types are stored in variables of type **typetype**. Special values are inttype, Booltype. Arrays have a subtype.
gettype(symbol; var t:typetype); getsubtype(origtype; var subtype)
Tsize(t:typetype):integer; (*returns size of a variable of type t, in bytes. Usually 4.*)

code generation

emit(inst, param);
inst has type insttype= (LOAD addr, ILOAD, DLOAD (num or addr), STOR addr, ISTOR, JUMP, JTRUE, JFALSE, ADD, MUL, DIV, MOD, SUB, AND, OR, CMP operand, NEG, NOT). Procedure genlabel(var l: labeltype) creates a new label. Emit(LOC, label) then introduces that label into the code.

Now we start the compilation. We assume that we are faced with the "begin" of the main "begin...end" pair. However, we'll start the procedures with CSIMEXPR. THis is taken pretty much from Dragon. It compiles "Simple Expressions", which is what the "expr"s of Dragon were. (The EXPRS here allow comparisons). Stack notation, used here, *is* postfix (used in Dragon).

Although we have both Boolean and Integer types, I have omitted type checking. It is not difficult to add it. In fact, it will be an assignment.
<u>**HINT:** Compare each of these with its EBNF, above.</u>

```
procedure CSIMEXPR;              (* cf Dragon, expr, p. 75 *)
var signflag: boolean;           (* true if we have a minus in front *)
    addsym: symtype;
begin
    signflag := false;
    if symbol = plus then
        match (plus)
    else if symbol = minus then
        begin match (minus); signflag := true  end;
    (* now the sign is stripped *)
    CTERM;
    while IsAddOp(symbol) do begin        (* is symbol = plus or minus? *)
        addsym := symbol;
        match(addsym);           (* same as getsym here; match must succeed *)
        CTERM;
        case addsym of
            plus:     Emit(ADD);
            minus:    Emit(SUB);
            orsym:    Emit(OR);
        end (* case *)
    end (* while *)
end; (* CSIMEXPR *)

procedure CTERM;                 (* cf. Dragon, p. 75 *)
var mulsym : symtype;
begin
    CFACTOR;
```

```
    while IsMulOp(symbol) do begin;
        mulsym:= symbol; match(mulsym);
        CFACTOR;
        case mulsym of
            times:    Emit(MUL);
            divsym:   Emit(DIV);
            modsym:   Emit(MOD);
            andsym    Emit(AND);
        end; (*case*)
    end; (*while*)
end; (* CTERM *)
```

```
procedure CFACTOR;          (* number | '(' expr ')' | 'not' factor | variable *)
    (* uses CVARIABLE, below, which loads address, not data! *)
begin
    case symbol of
        number:  begin Emit(DLOAD, value(symbol)); match(number) end;
        lparen:  begin match(lparen); CEXPR; match(rparen); end;
        notsym:  begin match(notsym); CFACTOR; Emit(NOT); end;
        ident:   begin CVARIABLE; emit(ILOAD) (*why?*); end;
    end; (* case *)
end; (*CFACTOR *)
```

Finally, CEXPR; recall expr ::= simple-expr [relop simple-expr], where relop (relational operator) is =, ≠, <, >, etc. We assume emit works so that Emit(CMP, relop) emits the correct one of CMP =, CMP ≠, CMP <, etc. This saves a big case statement.

```
procedure CEXPR;
var relop: symtype;   (* to save the operator *)
begin
    CSIMEXPR;
    if IsRelOp(symbol) then begin
        relop := symbol; match(relop);
        CSIMEXPR;
        EMIT(CMP, relop);         (* shortcut mentioned above *)
    end; (* if relop *)
end; (*CEXPR*)
```

Here is procedure CVARIABLE, that generates code to push the <u>address</u> of the variable onto the stack. The procedure **getaddr** looks up the allocated of a given variable; the result is returned in a variable of type **addrtype**. At this point we begin to diverge substantially from the Dragon book; we are no longer dealing with postfix alone.

```
procedure CVARIABLE (var vartype : typetype);
(* pushes address of current variable onto top of stack. *)
var
    addr   : addrtype;             (* representation of address *)
begin
    assert( isIdent(symbol));    (* error if symbol is not an identifier*)
    getaddr(symbol, addr);
```

```
        gettype(symbol, vartype);   (* type of variable *)
        Emit(DLOAD, addr);          (* load address on top of stack *)
        match(ident);               (* get next symbol *)
        if symbol = lbracket do begin         (* we have an array *)
            Assert(IsArray(vartype));     (* is variable declared as an array?*)
            getsubtype(vartype, vartype);  ? Why does this work?
            match(lbracket);
            CEXPR(itype);           (* compile the index expression *)
            match(rbracket);
            Assert(iscompatible(itype, inttype));  index type must be integer!
            (* now compute the address of the array component *)
            emit(DLOAD ,4);         (* 4 = tsize(vartype) for all integer type *)
            emit(MUL);                    (* compute offset in array *)
            emit(ADD);                    (* address is now computed *)
          end;
        end; (* while *)
    end; (*CVARIABLE*)
```

The next, CASSIGN, is a bit awkward. We push the address of the left side onto the stack, and then push the value of the right side, and then emit the ISTOR instruction. This stores the contents of stack[top] in the location pointed to by stack[top-1]. Realistically, for assignment to a simple variable we should just evaluate the expression and emit a STOR instruction to that variable. This is not that hard to do, but I grew tired. The present assign does not allow for assignment between arrays. Finally, I stuck in some type checking. This means we now assume procedures return the type of the object just compiled in a parameter; e.g. CEXPR(var etype).

```
procedure CASSIGN;
var
    addr : addrtype;
    etype, vtype : typetype;       (* type of expression and variable *)
begin
    CVARIABLE(vtype);              (* leaves address of left side on stack *)
    (* note this is not how cvariable was written above! *)
    match(assign);
    CEXPR(etype);                  (* leaves value of right side on stack*)
    assert(IsCompatible(etype, vtype));   (*assert prints error msg if false*)
    emit(ISTOR);                   (*store one machine word at stack addr*)
end; (* CASSIGN *)


procedure CWHILE;   (* compile a while statement *)
var
    label1,label2 : labeltype;
    exprtype : typetype; (* type of expression encountered *)
begin
    match(whilesym);
    genlabel(label1);
    genlabel(label2);
    emit(LOC, label1);             (* branch to here on body exit *)
    CEXPR(exprtype);  (*compile the loop expression, & leave result on stack*)
    Assert(exprtype = booltype);                    (* error otherwise *)
    emit(JFALSE, label2);          (* jump on false *)
```

```
    match(dosym);
    CSTATEMENT;                  (* compile loop body *)
    emit(JUMP, label1)           (* always jump back to label 1 *)
    emit(LOC, label2);           (*target for jump if test fails*)
end;


procedure CIF;
(* compiles if-then and if-then-else *)
var
    label1,label2 : labeltype;
    exprtype : typetype; (* type of expression encountered *)
begin
    match(ifsym);
    genlabel(label1);
    CEXPR(exprtype); (*compile the if expression, & leave result on stack*)
    Assert(exprtype = boolean);              (* error otherwise *)
    match(thensym);
    emit(JFALSE, label1);        (* jump on false *)
    CSTATEMENT;                  (* compile loop body *)
    if symbol <> elsesym then begin              (* else part coming *)
        emit(LOC, label1);       (*target for jump *)
    end
    else begin                   (* ELSE clause is present *)
        match(elsesym);
        genlabel(label2);
        emit(JUMP, label2);      (* unconditional jump *)
        emit(LOC, label1);       (*target for jump if condition was false *)
        CSTATEMENT;
        emit(LOC, label2);
    end;
end;
```

BEGIN is last; this is what we assume the program starts with.

```
procedure CBEGIN;
(* compiles begin <statement> { ';' <statement> } end  *)
begin
    CSTATEMENT;
    while symbol = semicolon do begin
        match(semicolon);                  (* read the semicolon *)
        CSTATEMENT;
    end;
    match(endsym); (* if symbol ≠ end, the most likely error *)
end;               (* is that the user omitted a semicolon *)
```

Oops! I forgot CSTATEMENT:

```
    case symbol of
        ident :        CASSIGN;              basic idea
        whilesym:      CWHILE;
        ifsym:         CIF;
        beginsym:      CBEGIN;
        none of the above: ; (* compiles the empty statement!*)
```