The Vassar Interpreter (CS331)

Keith Suderman
Vassar College
suderman@cs.vassar.edu

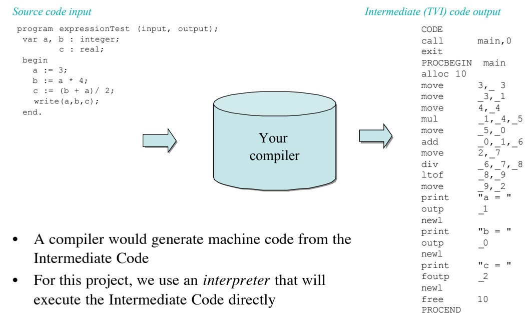
Compiling

- No "real" compiler takes source code and converts it directly into machine code.
  - compile source into intermediate code
  - perform optimizations on intermediate code
  - generate CPU specific opcodes

Your Task

- You will take a Pascal like language and generate intermediate code.
  - intermediate code looks a lot like machine code
  - the interpreter takes your intermediate code an executes it
  - eliminates many ugly hardware details

The Overview



- A compiler would generate machine code from the Intermediate Code
- For this project, we use an *interpreter* that will execute the Intermediate Code directly

A closer look at TVI code

```
program expressionTest (input, output);
 var a, b : integer;
        c : real;
 begin
   a := 3;
   b := a * 4;
   c := (b + a)/ 2;
   write (a,b,c);
 end.
```

Symbol table

name	type	offset
a	int	1
b	int	0
c	real	2
temp1	int	3
temp2	int	4
temp3	int	5
temp4	int	6
temp5	int	7
temp6	int	8
temp7	real	9

Temp variables are created as needed for expressions

```
CODE
call      main,0
exit
PROCBEGIN  main
alloc 10
move      3,_3
move      _3,_1
move      4,_4
mul       _1,_4,_5
move      _5,_0
add       _0,_1,_6
move      2,_7
div       _6,_7,_8
ltof      _8,_9
move      _9,_2
print     "a = "
outp      _1
newl
print     "b = "
outp      _0
newl
print     "c = "
foutp     _2
newl
free      10
PROCEND
```

What does the interpreter need to do?

- Read in the tokens
  - Easy because of fixed field format
- Does *not* enter IDs in symbol table
  - There are no IDs!
  - Variables are now just addresses (offsets)
- Allocate storage
  - Our intermediate code will include "alloc" and "free" statements at beginning/end of procedures (and the main routine)
- Execute the statements
  - Access memory, perform operations

The Interpreter

- Written in C++ (originally)
- Bison generated parser (Yacc clone)
- Flex generated scanner (Lex clone)
  - Using automated tools makes a lot of the work go away

Writing TVI Code

- TVI code is broken into two types of sections
  - DATA sections
    - contain global variable declarations and initialization
    - Your TVI code does not deal with this
  - code sections
    - contain code…
    - execution will start at first instruction in the *first* code segment and *flow through* in linear order

CODE Sections

- scanner/parser take all code segs and link into one long list
- How does main() get called???
  - Either the first statement of main is the first statement of the first code segment, or
  - Compiler adds a "stub" that calls main
    - This is most likely and the way most compilers do it.

Data Types

- Two simple types
  - STR (strings)
  - LONG (4 bytes)
- Can't do anything with strings other than print them
- NO TYPE CHECKING!!!

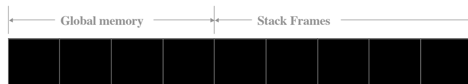
Data Types

- There is a difference between
  - integer operations
  - floating point operations
  - separate op codes for each type
- The interpreter doesn't care what you do with memory locations

```
add     %0, %1, _0
fadd    %0, %1, _1
```

Memory Model

- Series of contiguous memory locations
  - Anything declared in a DATA section is in global memory
  - User defined variables are in a stack frame



Memory Access

- Each memory location is accessed through an offset
  - _0 is offset 0 from the *base* of memory
  - %0 is offset 0 in the current stack frame

```
move 5,  _0
move 5, %0
add   %1, %0, %0
```

  - There are also @ and ^ modifiers.

Writing TVI Code

```
Program Sum (input, output);
Var
     i,j,k: Integer;
begin
     i := 5;
     j := 10;
     k := i + j;
     write(k);
end.
```

Writing TVI Code

```
CODE
        alloc       12
        move        5, %0
        move        10, %4
        add         %0, %4, %8
        outp        %8
        newl
        free        12
        exit
```

Writing TVI Code

- Three address opcodes are used:
  - *opcode operand1, operand2, destination*

```
add   i, j, k      ; k := i + j
sub   i, j, k      ; k := i - j
```

Exceptions
print, move, goto, param, call, etc.

Input

- There is are two input instructions

  inp dest
  - Read a number and store in integer format

  finp dest
  - Read a number and store in floating point format in dest

Output

- There are four output statements
  - newl ; prints a newline
  - print "Print a literal string.\n"
  - outp _0 ; print as int
  - foutp _0 ; print as float

Moving Memory

- Use the *move* instruction to move contents of one location to another

```
move    s, d        ; d = s
move    _1, %0
```

- No floating point version required
  - These simply copy bits

Array Style Memory Access

- Very often we want to access memory as an array
- Two opcodes to use
  - load array, offset, dest
    - load A, i, j ; j := a[i]
  - store value, offset, array
    - store j, i, A ; a[i] := j

Array Style Memory Access

```
DATA
        LONG            A[5], i, j
CODE
        move    0, i            ; i:= 0
0:      stor    i, i, A         ; A[i] := i
        add     i, 4, i         ; i := i + 4
        blt     i, 20, 0        ; if i < 20 goto 0
        move    0, i            ; i := 0
1:      load    A, i,  j        ; j := A[i]
        outp    j               ; write(j)
        newl                    ; newline
        add     i, 4, i         ; i := i + 1
        blt     i, 20, 1        ; if i < 5 goto 1
        exit
```

Array Style Memory Access

```
CODE
        alloc   28              ; array at %0 to %16
        move    0, %20          ; index at %20
0:      store   %20, %20, %0    ; A[i] := i
        add     %20, 4, %20     ; i := i + 4
        blt     %20, 20, 0      ; if i < 20 goto 0
        move    0, %20          ; i := 0
1:      load    %0, %20, %24    ; j := A[i]
        outp    %24             ; write(j)
        newl                    ; newline
        add     %20, 4, %20     ; i := i + 4
        blt     %20, 20, 1      ; if i < 20 goto 1
        free    28
        exit
```

Flow of Control (Branching)

- goto label
  - where "label" is an integer followed by a colon

```
CODE
1:   ...
     goto 99
     ...
99:  ...
     goto 1
```

Flow of Control (Branching)

- labels do not need to be in numerical order
- do not need to start at a particular value
- best ides is to write a simple function to generate labels as needed

Generating Labels

```
int GenerateLabel(void)
{
     static int nextLabel = 0;
     return nextLabel++;
}
```

Conditional Branching

- Six opcodes used to test conditionals
  - beq (branch if equal)
  - bne (" if not equal)
  - blt (" if less than)
  - ble (" if less than or equal)
  - bgt (" if greater than)
  - bge (" if greater than or equal)

Conditional Branching

- All branching opcodes take the same form
  - opcode operand1, operand2, label

```
CODE
      ...
      blt   _0, %0, 99   ; if i < j then
                         ;      goto 99
```

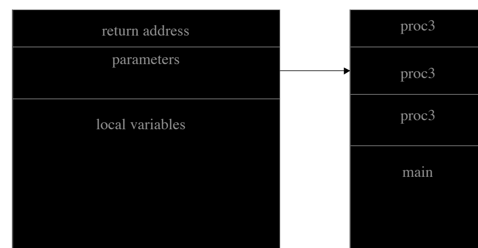
Addressing Modes

- Different ways to access memory
  - load from the memory location
  - treat as a pointer
  - use the address of the location
- move i, j ; moves value of i
- move @i, j ; moves address of i
- move ^j, k ; dereference j

Stack Frames

- Created by the compiler every time a function or procedure is invoked
- Used to store
  - return address
  - any parameters passed to the procedure
  - local variables
  - etc.

Stack Frames



Stack Frames

- TVI uses very simple stack frames
  - don't need a return address
  - holds the parameters
  - compiler must add space for local variables
- Use addressing mode %n
  - '%' indicates this is an offset into the current stack frame
  - 'n' is the offset in the frame

Using Local Variables

- The compiler must increase the size of the stack frame to hold local variables
  - up to the compiler to keep track of the offsets of local variables in the frame
- Must also release the extra space before the procedure returns!!!!
  - "alloc" adds space for variables
  - "free" releases space

Using Local Variables

```
CODE
      param      number
      call       foo
      exit
PROCBEGIN   Foo
      alloc      12
      move       %0, %4
      add        %0, %4, %8
      free       12
PROCEND
```

Calling Procedures

```
Program Test(input, output);
Var
      i, j, k: Integer;
Procedure Sum(a, b: Integer);
begin
      k := a + b;
end;

begin
      i := 5; j := 10;
      Sum(i, j);
      write(k);
end.
```

Calling Procedures

```
CODE
      alloc 12
      move  5, %0
      move  10, %4
      param %0
      param %4
      call  Sum
      outp  %8
      exit
PROCBEGIN   Sum
      alloc 12
      add   %0, %4, _8
      free  12
PROCEND
```

Calling Functions

```
Program Test(input, output);
Var
      i, j, k: Integer;
Function Sum(a, b: Integer);
begin
      Sum := a + b;
end;

begin
      i := 5; j := 10;
      k := Sum(i, j);
      write(k);
end.
```

Calling Functions

```
CODE
      alloc 12
      move  5, %8
      move  10, %4
      param %8
      param %4
      param @%0
      call  Sum
      outp  %8
      exit
PROCBEGIN   Sum
      alloc 12
      add   %0, %4, ^%8
      free  12
PROCEND
```

Using the Interpreter

- "run program.a"
  - extension doesn't matter
- command line parameters
  - /p:[onloff] (print instructions)
  - /d:[onloff] (parser output)
  - /s:[onloff] (scanner output)
- if onloff omitted then "on" is assumed

Using the Interpreter

```
run /p program.a
run /p:on /d:on /s:off program.a
run /p /d /s program.a
etc.
```

The End

The Parser

```
instruction   : ADD op ',' op ',' dest_op
                  {
                     $$ = new CQuadruple(ADD, $2, $4, $6);
                  }
              | SUB op ',' op ',' dest_op
                  {
                     $$ = new CQuadruple(SUB, $2, $4, $6);
                  }
              | DIV op ',' op ',' dest_op
                  {
                     $$ = new CQuadruple(DIV, $2, $4, $6);
                  }
              | ...
```

The Scanner

```
param                      { return PARAM; }
alloc                      { return LOCAL; }
print                      { return OUTPUT; }
goto                       { return GOTO; }
DATA                       { return DATA; }
CODE                       { return CODE; }
call                       { return CALL; }
LONG                       { return LONG; }
[a-zA-Z][_a-zA-Z0-9]*      { yylval.string =
                           duplicate(yytext);
                             return IDENT; }
```