Last Time: Code Generation

- Over the past few lectures we've seen how to translate abstract machine code, such as T-code and JVM code, to the structures and instructions of a real machine.

- This is one way to implement a programming language, but it has its drawbacks:
  - the generated code is machine-dependent, and runs only on that operating system and target machine.
  - we need to make a new code generator for each operating system and target machine.

- In the modern interweb-based world, these are serious problems. We need language implementations that are independent of their environment, and can run programs anywhere.
Abstract Machine Interpreters

• An alternative method for implementing a language is an abstract machine interpreter - a program that, given the abstract machine code (T-code or JVM code) for a program, simulates the virtual machine's actions to run the program.

• We have already seen one such interpreter - the S/SL "walker" program itself, which simulates the S/SL "machine" to run our S/SL programs.

• The basic idea of an abstract machine interpreter is the same - we put the abstract machine code in a big code array (e.g. S/SL table) then "walk" the array from the beginning, fetching each abstract machine instruction and doing what it specifies, using global variables to represent the Memory, ES, Display and other structures of the abstract machine.

• The interpreter itself is a very simple program that can be implemented in any language, and is easily ported from machine to machine.
Abstract Machine Interpreters

var code : array [0 .. maxcode] of integer;
var es : array [1 .. maxdepth] of integer;
var estop : integer;
var pc : integer;

begin
  estop := 0;  pc := 0;
  while (processing) do
    begin
      case code[pc] of
        tLiteralInteger:
          begin
            pc := pc + 1;
            estop := estop + 1;
            es[estop] := code[pc];
            pc := pc + 1;
          end;
    end;
end;
end.
Interpreters vs. Compilers

• Because they must be precisely specified in order to understand the language well enough to implement it, abstract machines have a very detailed specification, and so interpreters are relatively easy to program quickly and accurately.

• By contrast, as we have seen, creating a code generator requires a lot of new design decisions, involving many choices for which there is no clear correct answer.

• Thus, if we want our new language to be quickly portable between platforms, interpreting is the preferred way.
The PT Abstract Machine

- The PT abstract machine is based on a model very similar to real hardware units, and is fully specified in terms of transfers of data between these units.

  Stack:  
  expression stack

  L, R:  
  temporary registers

  IP:  
  instruction pointer

  Memory:  
  data area

  Stack <- (expn):  
  push expn value onto Stack

  var <- Stack:  
  pop Stack and assign popped value to var

  tLiteralAddress(addr)  
  Stack <-- (addr)

  tFetchAddress  
  L <-- Stack; Stack <-- Memory[L]

  tAssignAddress  
  R <-- Stack; L <-- Stack; Memory[L] <-- R
Interpreters in S/SL

• The boxes and arrows on the previous slide should remind you of S/SL - boxes being semantic mechanisms, and the arrows being semantic operations.

• S/SL is a very good vehicle for implementing abstract machine interpreters, separating the higher level logic (the specification of the interpreter as interactions of the units) and the lower level data manipulation (the implementation of the units).

• The PT abstract machine interpreter, PTAM, is implemented in this way.

• This is much like the microprogramming of complex instruction sets in real machines such as the Intel Core processors, where a higher level instruction set is implemented by internal sequences of lower level operations.
type INSTRUCTION : % corresponds to semantic.ssl output
tMultiply
tDivide
tModulus
tAdd
tSubtract
  . . .
tFetchAddress
tFetchInteger
tFetchChar
tFetchBoolean
tAssignBegin
  . . .
tLiteralAddress
tLiteralInteger
tLiteralChar
tLiteralBoolean
  . . .
tTrap
tWriteEnd
tReadEnd
;
mechanism Memory:
% The memory mechanism transfers data to or from memory.
% The memory address register (MAR) points to the target
% of the fetch or store. The memory data register (MDR)
% holds the data to be stored or the data fetched.
%
% MDR := Memory[MAR] 'fetch'
% Memory[MAR] := MDR 'store'

oMemoryLoadCode % Load the code memory from
% the T-code input file

oMemoryFetchInstruction % MDR <- CodeMemory[MAR]
% Fetch the next instruction from
% the code memory at address MAR.

... 

oMemoryFetchChar % MDR <- Memory[MAR]
oMemoryFetchInteger % Fetch the specified type from
oMemoryFetchAddress % the data memory at address MAR.
oMemoryFetchBoolean... ;
mechanism ALU:
% The ALU mechanism manipulates the two general purpose
% resisters L and R, and performs all logical and
% arithmetic operations. The expression stack, ES,
% is part of this mechanism.

oALUpushChar         % Stack <- R
oALUpushInteger      % Push the contents of the R register
oALUpushAddress      % onto the expression stack.
oALUpushBoolean

oALUpopL             % L <- Stack
% Pop the top element of the ES
% into the L register.

oALUpopR             % R <- Stack
% Pop the top element of the ES
% into the R register.

oALUpushLplusR       % Stack <- L + R
oALUpushLminusR      % Stack <- L - R
oALUpushLtimesR      % Stack <- L * R

;
mechanism CU:

% The CU mechanism implements the control unit, which
% manipulates the control registers PC, IR, MAR and MDR
% as well as the return stack.

oCUchooseInstruction >> INSTRUCTION
  % Returns the contents of the IR;
  % the instruction to be executed.

oCUzeroPC      % PC <- 0
oCUincPC       % PC <- PC + 1
              % Increment the PC

oCUmoveMDRtoR  % R <- MDR
oCUmoveMDRtoT  % T <- MDR
oCUmoveMDRtoZ  % Z <- MDR
oCUmoveMDRtoIR % IR <- MDR
oCUmoveMDRtoPC % PC <- MDR
oCUmoveMDRtoSCR% SCR <- MDR

  % (Load the specified register with
  % the contents of the MDR)

  ;
PTAM

rules

PTmachine :  
  oMemoryLoadCode % load the code memory  
  % from the T-code file  
  % set PC to 0

  oCUzeroPC

  {
    oCUmovePCtoMAR % address of next instruction
    oMemoryFetchInstruction % fetch next instruction...
    oCUmoveMDRtoIR % ...and put it into the IR
  
  [ oCUchooseInstruction % CHOOSE on contents of IR

    | tAssignInteger: % integer assignment
      oALUpopL % L <- pop
      oALUpopR % R <- pop
      oCUmoveRtoMAR
      oCUmoveLtoMDR
      oMemoryStoreAddress % Memory[R] <- L
      oCUincPC % next instruction

    ...
  }

  };

• While interpreters are easy and portable, they do give away performance - a simulation of the abstract machine cannot possibly run the program as fast as native compiled code

• A just-in-time (JIT) compiler is a hybrid between an interpreter and a code generator designed to address this issue

• JIT compilers work by interpreting abstract machine code in the usual way, but additionally try to optimize by running the code generator for the machine it is running on at run time

• The idea is to generate code for a class or method when it is called for the first time, and then execute the compiled code the next time(s) it is called

• This slows the first execution of each class or method enormously, but pays off in compiled speed for subsequent calls
Summary

• As an alternative to code generation, we can implement an abstract machine interpreter that executes programs directly by simulating the abstract machine.

• Interpreters have several advantages over code generation:
  – fully specified by abstract machine definition
  – simple and easy to implement correctly
  – easily portable to new machines

• S/SL provides a good way to implement abstract machine interpreters such as PTAM.

• Just-in-time (JIT) compilers are interpreters that run a code generator for the machine they are run on the first time a section of code is interpreted, then run the compiled code.
... and that is ...

THE END

... an S/SL Production
Next time ...

• Course summary and study guide

• Exam hints and walk-through of a previous exam

• Awarding of the famous CISC 458 Survival Certificates
  – A 25-year-old tradition!