SSE2-PLDE_4

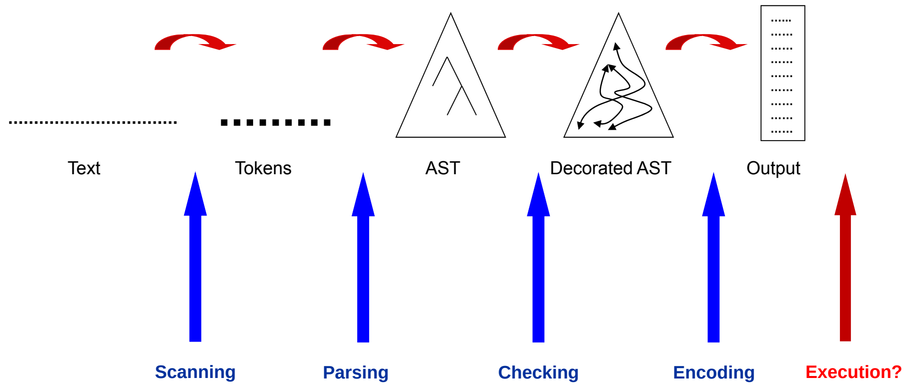
Contents:

- Interpretation, recursive or iterative
- Triangle Abstract Machine (TAM), TAM interpreter

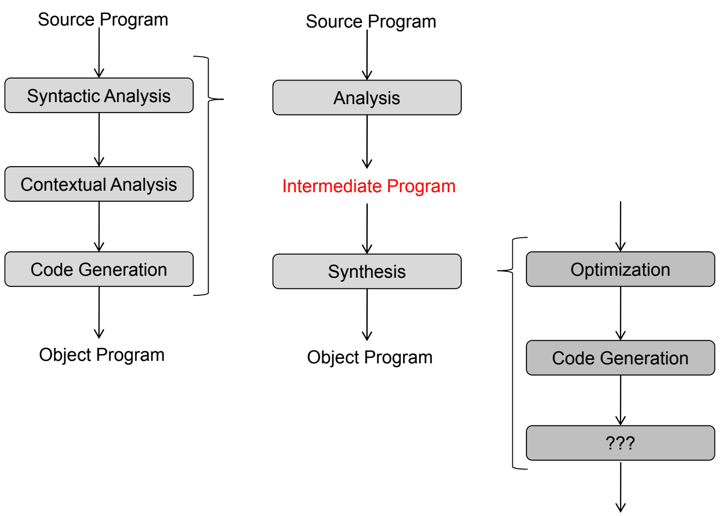
Literature:

- Watt & Brown:
  - 8.2, 8.3
  - C.1-C.3

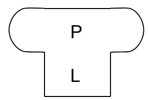
Translation

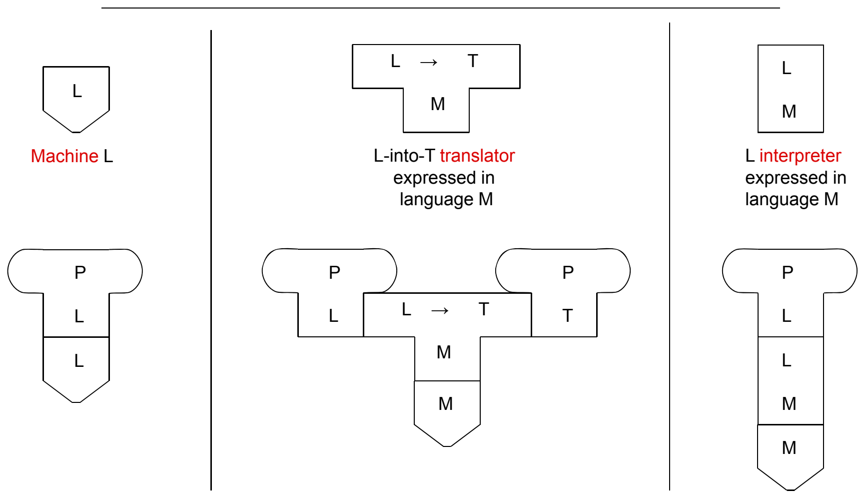


Compiler Structure



Translation & Interpretation





Interpretation

- Iterative interpretation scheme

- Recursive interpretation scheme

```
fetch and analyze the program
execute the program

// both analysis and execution are recursive)
```

Mini Triangle (Recursive) Interpretation

```
public Object visitAssignCommand (AssignCommand com, Object arg) {
   Value val = (Value) com.E.visit (this, null);
   assign(com.V, val);
   return null
}

public Object visitSequentialCommand (SequentialCommand com, Object arg) {
   com.C1.visit (this, null);
   com.C2.visit (this, null);
   return null;
}

public Object visitWhileCommand (WhileCommand com, Object arg) {
   for (;;)  {
      BoolValue val = (BoolValue) com.E.visit (this, null);
      if (! Val.b) break;
      com.C.visit (this, null);
   }
   return null;
}

public Object visitIfCommand (IfCommand com, Object arg) {
   BoolValue val = (BoolValue) com.E.visit (this, null);
   if (Val.b) com.C1.visit (this, null);
   else       com.C2.visit (this, null);
   return null;
}
```

Mini Triangle (Recursive) Interpretation

```
public class MiniTriangleState {
   …
   Program program; //decorated AST
   …
   Value[] data = new Value(DATASIZE];
   …
}
```

```
public interface Visitor {
  …
}
```

```
public class MiniTriangleProcessor extends MiniTriangleState implements Visitor {

   public void fetchAnalyze ()  {
      // parse … check … allocate …
   }

   public void run ()  {
      program.C.visit(this, null);
   }

   // Visitor/interpreting methods …
   …
   // Other methods …
   …
}
```

```
Public abstract class Value  {  }

Public class IntValue extends Value  {
   public short int;
}

Public class BoolValue extends Value  {
   public boolean b;
}

Public class UndefinedValue extends Value  {
}
```

Mini Triangle (Recursive) Interpretation

```
public Object visitVarDeclaration (VarDeclaration decl, Object arg) {
   KnownAdress entity = (KnownAddress) delc.entity;
   data[entity.address] = new UndefinedValue();
   return null
}

public Object visitVnameExpression (VnameExpression expr, Object arg) {
   return fetch(expr.V);
}

public Object visitBinaryExpression (BinaryExpression expr, Object arg) {
   Value val1 = (Value) expr.E1.visit (this, null);
   Value val2 = (Value) expr.E2.visit (this, null);
   return applyBinary (expr.O, val1, val2);
}
```

LISP Interpreter

```
evalquote[fn; x] ::= apply[fn; x; NIL]

apply[fn; x; a] ::= [atom[fn] → [eq[fn; CAR] → caar[x];
                                 eq[fn; CDR] → cdar[x];
                                 eq[fn; CONS] → cons[car[x]; cadr[x]];
                                 eq[fn; ATOM] → atom[car[x]];
                                 eq[fn; EQ] → eq[car[x]; cadr[x]];
                                 T → apply[eval[fn; a]; x; a]];
            eq[car[fn]; LAMBDA] → eval[caddr[fn]; pairlis[cadr[fn]; x; a]]
             eq[car[fn]; LABEL] → apply[caddr[fn]; x; cons[cons[cadr[fn];
                                                         caddr[fn]]; a]]]

eval[e; a] ::= [atom[e] → cdr[assoc[e; a]];
               atom[car[e]] → [eq[car[e]; QUOTE] → cadr[e];
                               eq[car[e]; COND] → evcon[cdr[e]; a];
                               T → apply[car[e]; evlis[cdr[e]; a]; a]];
               T → apply[car[e]; evlis[cdr[e]; a]; a]]

pairlis[x; y; a] ::= [null[x] → a;
                      T → cons[cons[car[x]; car[y]]; pairlis[cdr[x]; cdr[y]; a]]]

assoc[x; a] ::= [eq[caar[a]; x] → car[a];  T → assoc[x; cdr[a]]]

evcon[c; a] ::= [eval[caar[c]; a] → eval[cadar[c]; a];
                 T → evcon[cdr[c]; a]]

evlis[m; a] ::= [null[m] → NIL;
                 T → cons[eval[car[m]; a]; evlis[cdr[m]; a]]]
```

Table 12-1: LISP Interpreter

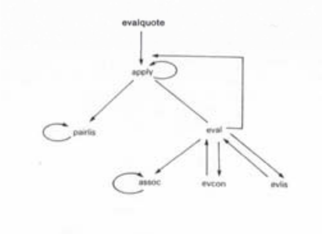


Figure 12-2: Calling Hierarchy of the LISP Interpreter

Interpretation

- Iterative interpretation scheme

```
Initialize
do {
    fetch the next instruction
    analyze this instruction
    execute this instruction
} while (still running) ;
```

- Recursive interpretation scheme

TAM Expression Evaluation

Register Machine:

```
(a * b) + (1 – (c * 2))

LOAD R1 a
MULT R1 b
LOAD R2  #1
LOAD R3 c
MULT R3  #2
SUB R2 R3
ADD R1 R2
```

STORE R*i a*	Store the value in register *i* in address a
LOAD R*i* x	Fetch the value of *x* and place it in register *i*
ADD R*i x*	Fetch the value of *x* and add it to the value in register *i*
SUB R*i x*	Fetch the value of *x* and subtract it from the value in register *i*
MULT R*i x*	Fetch the value of *x* and multiply it to the value in register *i*

Stack Machine:

```
(a * b) + (1 – (c * 2))

LOAD a
LOAD b
MULT
LOADL 1
LOAD c
LOADL 2
MULT
SUB
Add
```

STORE *a*	Pop the value off the stack and store it in address *a*
LOAD *a*	Fetch the value from address *a* and push it on to the stack
LOADL *n*	Push the literal value *n* on to the stack
ADD	Replace the top two values on the stack by their sum
SUB	Replace the top two values on the stack by their difference
MULT	Replace the top two values on the stack by their product

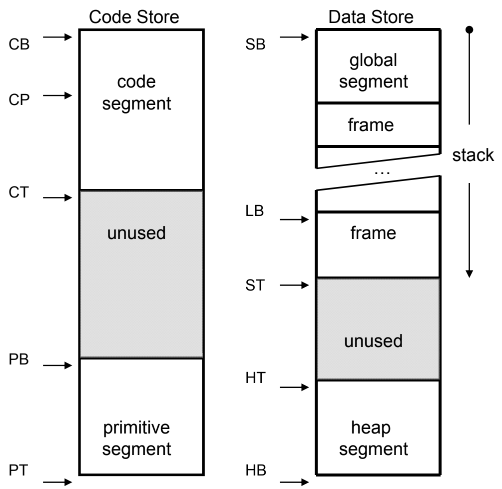
TAM Instructions

op	*r*	*n*	*d*
4 bits	4bits	8 bits	16 bits (signed)

TAM instruction format

Op-code	Instruction mnemonic	Effect
0	`LOAD (n) d[r]`	Fetch an *n*-word object from the data address (*d* + register r), and push it on to the stack
…	…	…

TAM Organization: Code & Data Store



TAM Registers

Register number	Register mnemonic	Register name	Behavior
0	CB	Code Base	constant
1	CT	Code Top	constant
2	PB	Primitives Base	constant
3	PT	Primitives Top	constant
4	SB	Stack Base	constant
5	ST	Stack Top	changed by most instructions
6	HB	Heap Base	constant
7	HT	Heap Top	changed by most instructions
8	LB	Local Base	changed by call and return instructions
9	L1	Local Base 1	L1 = *content* (LB)
10	L2	Local Base 2	L2 = *content* (*content* (LB))
11	L3	Local Base 3	L3 = *content* (*content* (*content* (LB)))
12	L4	Local Base 4	L4 = *content* (*content* (*content* (*content* (LB))))
13	L5	Local Base 5	L5 = *content* (*content* (*content* (*content* (*content* (LB)))))
14	L6	Local Base 6	L6 = *content* (*content* (*content* (*content* (*content* (*content* (LB))))))
15	CP	Code Pointer	changed by all instructions

TAM

```
public class Instruction {
  …
  public int op; // OpCode
  public int r;  // RegisterNumber
  public int n;  // Length
  public int d;  // Operand
  …
}
```

```
public final class Machine {
  …
  public final static int
    LOADop = 0,   LOADAop = 1,   LOADIop = 2,
    LOADLop = 3,  STOREop = 4,   STOREIop = 5,
    CALLop = 6,   CALLIop = 7,   RETURNop = 8,
    PUSHop = 10,  POPop = 11,    JUMPop = 12,
    JUMPIop = 13, JUMPIFop = 14, HALTop = 15;

  public static Instruction[] code = new Instruction[1024];

  public final static int
    CBr = 0, CTr = 1, PBr = 2, PTr = 3, SBr = 4,
    STr = 5, HBr = 6, HTr = 7, LBr = 8, L1r = LBr + 1,
    L2r = LBr + 2, L3r = LBr + 3, L4r = LBr + 4,
    L5r = LBr + 5, L6r = LBr + 6, CPr = 15;
  …
}
```

TAM Interpreter

```
static void interpretProgram() {
  …
  ST = SB; HT = HB; LB = SB; CP = CB;
  status = running;
  do {
    currentInstr = Machine.code[CP];
    op = currentInstr.op; r = currentInstr.r;
    n = currentInstr.n; d = currentInstr.d;
    switch (op) {
      case Machine.LOADop: ...
      case Machine.LOADAop: ...
      case Machine.LOADIop: ...
      case Machine.LOADLop: ...
      case Machine.STOREop: ...
      case Machine.STOREIop: ...
      case Machine.CALLop: ...
      case Machine.CALLIop: ...
      case Machine.RETURNop: ...
      case Machine.PUSHop: ...
      case Machine.POPop: ...
      case Machine.JUMPop: ...
      case Machine.JUMPIop: ...
      case Machine.JUMPIFop: ...
      case Machine.HALTop: status = halted; … break;
  } while (status == running);
}
```

TAM Instruction Execution

```
case Machine.LOADLop:
  …
  data[ST] = d;
  ST = ST + 1; CP = CP + 1;
  break;
case Machine.STOREop:
  addr = d + content(r);
  ST = ST - n;
  for (index = 0; index < n; index++)
    data[addr + index] = data[ST + index];
  CP = CP + 1;
  break;
case Machine.PUSHop:
  …
  ST = ST + d;
  CP = CP + 1;
  break;
case Machine.JUMPop:
  CP = d + content(r);
  break;
case Machine.JUMPIFop:
  ST = ST - 1;
  if (data[ST] == n)
    CP = d + content(r);
  else
    CP = CP + 1;
  break;
```

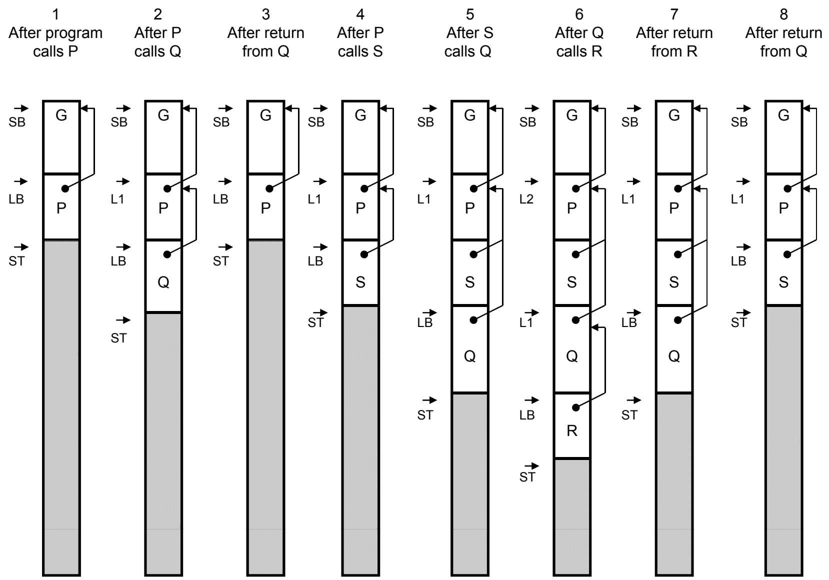
TAM Instruction Execution

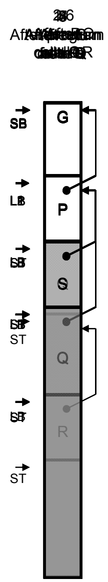
```
static void callPrimitive (int primitiveDisplacement) {
    …
    switch (primitiveDisplacement) {
      case Machine.andDisplacement:
        ST = ST - 1;
        data[ST - 1] = toInt(isTrue(data[ST - 1]) & isTrue(data[ST]));
        break;
      case Machine.negDisplacement:
        data[ST - 1] = -data[ST - 1];
        break;
      case Machine.addDisplacement:
        ST = ST - 1;
        accumulator = data[ST - 1];
        data[ST - 1] = overflowChecked(accumulator + data[ST]);
        break;
      case Machine.multDisplacement:
        ST = ST - 1;
        accumulator = data[ST - 1];
        data[ST - 1] = overflowChecked(accumulator * data[ST]);
        break;
      case Machine.ltDisplacement:
        ST = ST - 1;
        data[ST - 1] = toInt(data[ST - 1] < data[ST]);
        break;
    }
}
```

Routine levels:

- routine level 0
- routine level 1
- routine level 2
- routine level 3

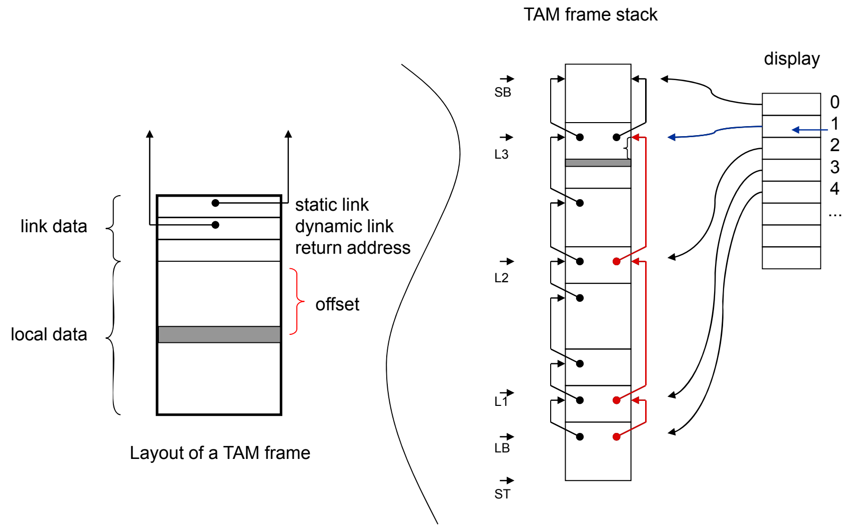
```
let
   var g1: Integer;
   var g2: array 3 of Boolean

   Proc P () ~
     let
        var p1: Boolean;
        var p2: Integer;

        proc Q () ~
          let
             var q: array 3 of Char;

             proc R () ~
               let
                  var r: Boolean
               in
                  begin … end !R!
          in begin … end !Q!

        proc S () ~
          let
             var s: array 4 of Char
          in
             begin … end !S!
     in
        begin … end !P!
in
   begin … end
```

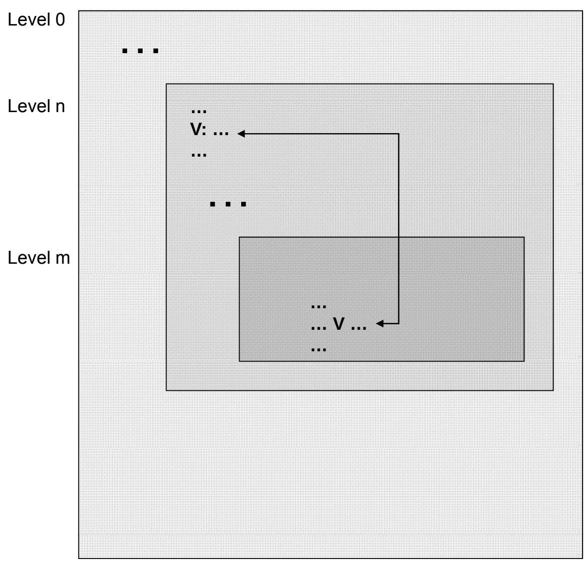





```
let                                           Level Offset
   var g1: Integer;                             0      1
   var g2: array 3 of Boolean                   0      2

   Proc P () ~
     let
        var p1: Boolean;                        1      1
        var p2: Integer;                        1      2

        proc Q () ~
          let
             var q: array 3 of Char;            2      1

             proc R () ~
               let
                  var r: Boolean                3      1
               in
                  begin … end !R!
          in begin … end !Q!

        proc S () ~
          let
             var s: array 4 of Char             2      1
          in
             begin … end !S!
     in
        begin … end !P!
in
   begin … end
```

TAM: Frame & Stack





Declaration:
V has level n and a unique offset for V (within this level n):
(V.level, V.offset)

Application
V is applied at level m

In order to access V:

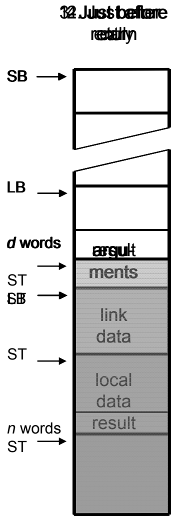
1) Use display[V.level] + V.offset

2) For this application of V the difference m-n is constant:
Follow static link m-n times and then add V.offset

TAM Instruction Execution

```
case Machine.LOADop:
  short addr = relative(d, r);
  data[ST++] = data[addr];
  break;

case Machine.STOREop:
  short addr = relative(d, r);
  data[addr] = data[--ST];
  break;
```

```
Private static short relative (short d, byte r)  {
  switch (r)  {
    …
    case SBr: return d + SB;
    case LBr: return d + LB;
    case L1r: return d + data[LB];
    case L2r: return d + data[data[LB]];
    …
  }
}
```



1. Just before call

...

Procedure … call (arguments)

4. Just after return

...

Procedure … (parameters) {

2. Just after entry

...

3. Just before return

}