Programming Languages
(CS 550)

Mini Language Compiler

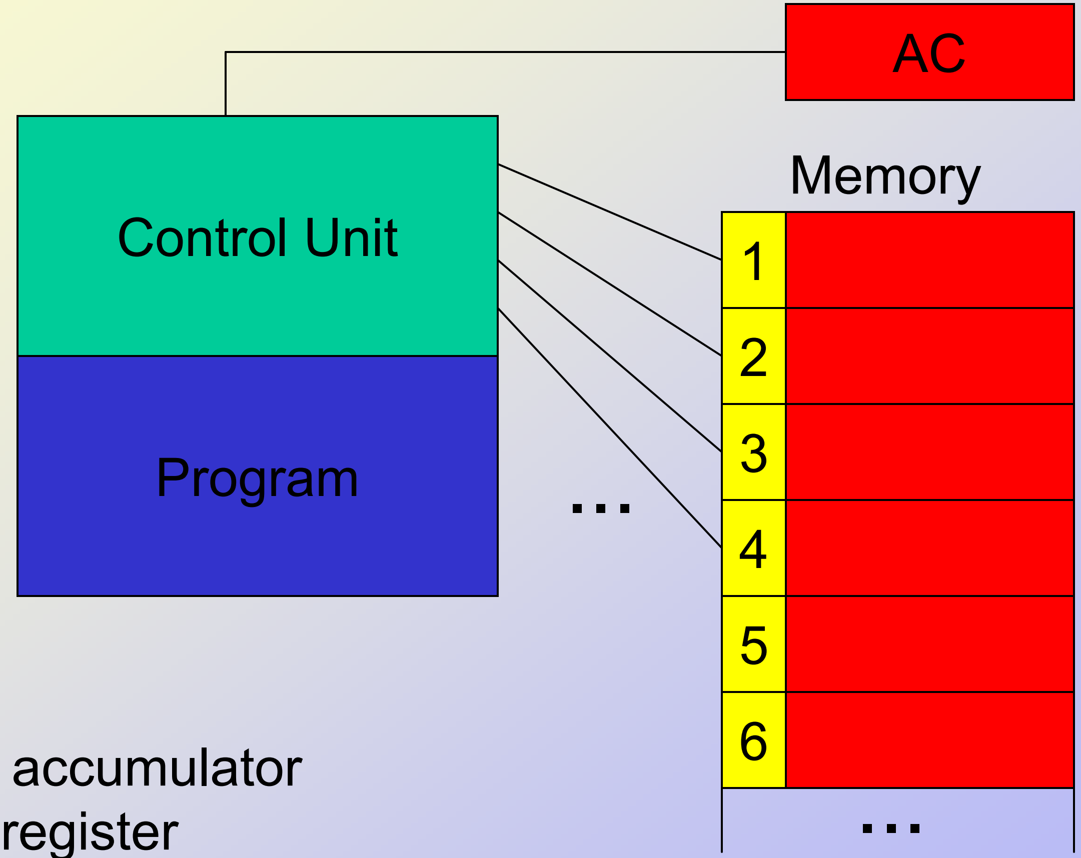
Jeremy R. Johnson

Introduction

- Objective: To illustrate how to map Mini Language instructions to RAL instructions. To do this in a systematic way that illustrates how to write a compiler to translate Mini Language programs to RAL programs. Show simple optimizations that can be used to reduce the number of instructions.

- Algorithm
  - Construct code for expressions, assignments, if, and while.
  - Concatenate statements in stmt-list
  - Allocate temporaries as needed
  - Keep track of variables, constants, and temporaries in Symbol Table
  - Use symbolic instructions and fill in absolute addresses (linking) when complete code has been constructed

A Random Access Machine

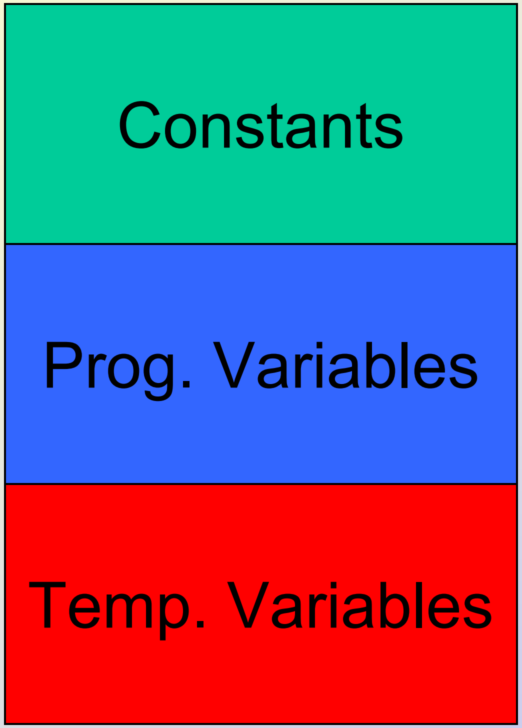


AC = accumulator register

Instruction Set

- LDA X; Load the AC with the contents of memory address X
- LDI X; Load the AC indirectly with the contents of address X
- STA X; Store the contents of the AC at memory address X
- STI X; Store the contents of the AC indirectly at address X
- ADD X; Add the contents of address X to the contents of the AC
- SUB X; Subtract the contents of address X from the AC
- MUL X; Multiply the contents of address X to the contents of the AC
- JMP X; Jump to the instruction labeled X
- JMZ X; Jump to the instruction labeled X if the AC contains 0
- JMN X; Jump to the instruction labeled X if the contents of the AC is negative
- HLT ; Halt execution

Memory Organization



Constants

Prog. Variables

Temp. Variables

- Num_Consts
- Num_Vars
- get_temp()
- Num_Temps

Symbolic Instructions

- Addresses and labels can be symbolic names
- Symbolic names are mapped to actual addresses during linking

- Example:
  - LD x
  - ST z
  - ADD y
  - JMP L
- Linked code with (x=100, y =110, z = 105, L = 20)
  - LD 100
  - ST 105
  - ADD 110
  - JMP 20

Symbol Table

- Map from identifiers → Symbol table entries

- Symbol table entries contain: address [may be unknown]

- Indicate whether entry is an constant, variable, temporary or label

Expressions

expr → $expr_1$ op $expr_2$

$Code_1$; result stored in t_1

$Code_2$; result stored in t_2

LD t_1 ; load result of exp_1

OP t_2 ; apply op to result of exp_2 and result of exp_1

ST t_3 ; store result of exp_1 op exp_2

Expressions

expr → NUMBER

; check to see if NUMBER in symbol table,
; otherwise add to symbol table

LD NUMBER ; load constant from constant table
ST t_n ; next available temporary

Expressions

expr → IDENT

; check to see if IDENT in symbol table

; otherwise add to symbol table

```
LD IDENT          ; load constant from constant table
ST t_n            ; next available temporary
```

Assignment

assign_stmt → IDENT = expr

; check to see if IDENT in symbol table

; otherwise add to symbol table

Code

```
LD t
ST IDENT
```

Conditional Statements

if_stmt → if expr then S_1 else S_2 fi

⇔ if expr > 0 then S_1 else S_2 fi

```
    Code_e    ; result stored in t
    LD t      ;
    JMN L1    ;  Jump if t ≤ 0 then
    JMZ L1
    Code_1
    JMP L2
L1: Code_2
L2:
```

While Statements

while_stmt → while expr do S od

⇔ while expr > 0 then S od

```
L1:  Code_e      ; result stored in t
     LD t        ;
     JMN L2      ; jump if t ≤ 0
     JMZ L2
     Code_S
     JMP L1
L2:
```

Statement List

stmt-list → stmt; stmt-list | stmt

code_1

code_2

…

code_n

Example

```
n := 0 - 5;
if n then i := n else i := 0 - n fi;
fact := 1;
while i do fact := fact * i; i := i - 1 od
```

Example

n := 0 - 5

```
LD ZERO
ST T1
LD FIVE
ST T2
LD T1
SUB T2
ST T3
LD T3
ST n
```

Example

if n then i := n else i := 0 - n fi;

```
    LD n
    ST T4
    LD T4
    JMN L1
    JMZ L1
    LD n
    ST T5
    LD T5
    ST i
    JMP L2
```

```
L1:  LD ZERO
     ST T6
     LD n
     ST T7
     LD T6
     SUB T7
     ST T8
     LD T8
     ST i
L2:
```

Example

```
fact := 1;

LD ONE
ST T9
LD T9
ST fact
```

Example

```
while i do
  fact := fact * i; i := i - 1
od
L3:  LD i
     ST T10
     LD T10
     JMN L4
     JMZ L4
     LD fact
     ST T11
     LD i
     ST T12
     LD T11
     MUL T12
     ST T13
     LD T13
     ST fact
     LD i
     ST T14
     LD ONE
     ST T15
     LD T14
     SUB T15
     ST T16
     LD T16
     ST i
     JMP L3
L4:
```

Complete Example

```
LD ZERO
ST T1
LD FIVE
ST T2
LD T1
SUB T2
ST T3
LD T3
ST n
LD n
ST T4
LD T4
JMN L1
JMZ L1
LD n
ST T5
LD T5
ST i
JMP L2
```

```
L1:  LD ZERO
     ST T6
     LD n
     ST T7
     LD T6
     SUB T7
     ST T8
     LD T8
     ST i
L2:  LD ONE
     ST T9
     LD T9
     ST fact
```

```
L3:  LD i
     ST T10
     JMN L4
     JMZ L4
     LD fact
     ST T11
     LD i
     ST T12
     LD T11
     MUL T12
     ST T13
     LD T13
     ST fact
```

```
     LD i
     ST T14
     LD ONE
     ST T15
     LD T14
     SUB T15
     ST T16
     LD T16
     ST i
     JMP L3
L4: HLT
```

Symbol Table

Name	Value	Type	addr
ZERO	0	const	?
FIVE	5	const	?
n	u	var	?
T1	u	temp	?
T2	u	temp	?
T3	u	temp	?
T4	u	temp	?
T5	u	temp	?
i	u	var	?
T6	u	temp	?
T7	u	temp	?
T8	u	temp	?
ONE	1	const	?
T9	u	temp	?
fact	u	var	?
T10	u	temp	?
T11	u	temp	?
T12	u	temp	?
T13	u	temp	?
T14	u	temp	?
T15	u	temp	?
T16	u	temp	?

Symbol Table and Label Summary

Num_Vars = 3

Num_Consts = 3

Num_Temps = 16

Constants

- ZERO -> addr 1
- FIVE -> addr 2
- One -> addr 3

Variables

- n -> addr 4
- i -> addr 5
- fact -> addr 6

Temporaries

- T1 -> addr 7
- T2 -> addr 8
- …
- T16 -> addr 22

L1 = 20

L2 = 29

L3 = 33

L4 = 55

Linked Example

```
LD 1
ST 7
LD 2
ST 8
LD 7
SUB 8
ST 9
LD 9
ST 4
LD 4
ST 10
LD 10
JMN 20
JMZ 20
LD 4
ST 11
LD 11
ST 5
JMP 29
```

```
L1:  LD 1
     ST 12
     LD 4
     ST 13
     LD 12
     SUB 13
     ST 14
     LD 14
     ST 5
L2:  LD 3
     ST 15
     LD 15
     ST 6
```

```
L3:  LD 5
     ST 16
     JMN 55
     JMZ 55
     LD 6
     ST 17
     LD 5
     ST 18
     LD 17
     MUL 18
     ST 19
    LD 19
    ST 6
```

```
    LD 5
    ST 20
    LD 3
    ST 21
    LD 20
    SUB 21
    ST 22
    LD 22
    ST 5
    JMP 33
L4: HLT
```

Optimizations

- Peephole optimization
  - Remove LD immediately following by ST

- Commute (expr$_1$,expr$_2$) in expr → expr$_1$ op expr$_2$ to allow additional peephole optimizations

- Constant folding

- Common subexpression elimination

Complete Example (after peephole optimization)

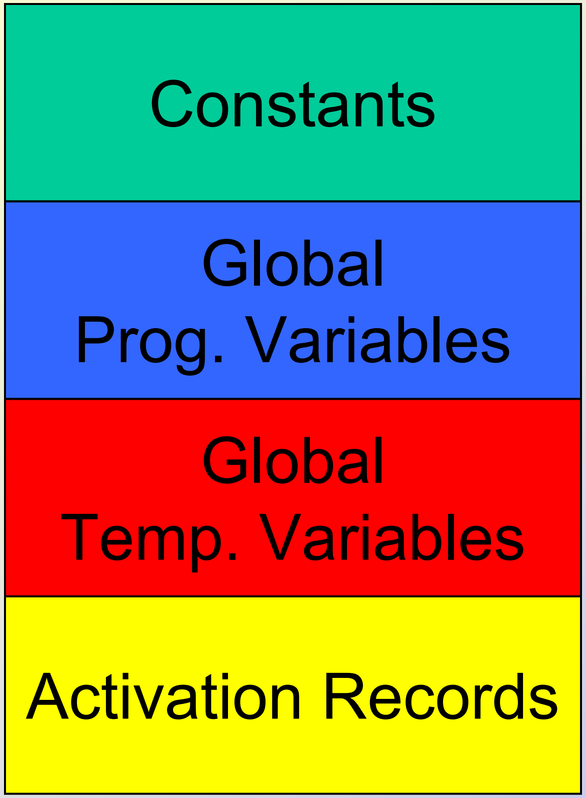
```
    LD FIVE
    ST T2
    LD ZERO
    ST T1
    SUB T2
    ST T3
    ST n
    ST T4
    JMN L1
    JMZ L1
    LD n
    ST T5
    ST i
    JMP L2
L1: LD ZERO
    ST T6
    LD n
    ST T7
    LD T6
    SUB T7
    ST T8
    ST i
L2: LD ONE
    ST T9
    ST fact
L3: LD i
    ST T10
    JMN L4
    JMZ L4
    LD i
    ST T12
    LD fact
    ST T11
    MUL T12
    ST T13
    ST fact
    LD i
    ST T14
    LD ONE
    ST T15
    LD T14
    SUB T15
    ST T16
    ST i
    JMP L3
L4: HLT
```

46 vs. 56 instructions

Supporting Procedures

- Fully static environment
  - No recursion
  - Activation record
    - Parameters
    - Local variables (keep count)
    - Return address (indirect jump needed)
    - Can be statically allocated
- Dynamic environment
  - Allow recursion
  - Call stack (dynamic allocation)
  - Use stack pointer (sp) and frame pointer (fp) access stack
  - Indirect load and store needed

Memory Organization



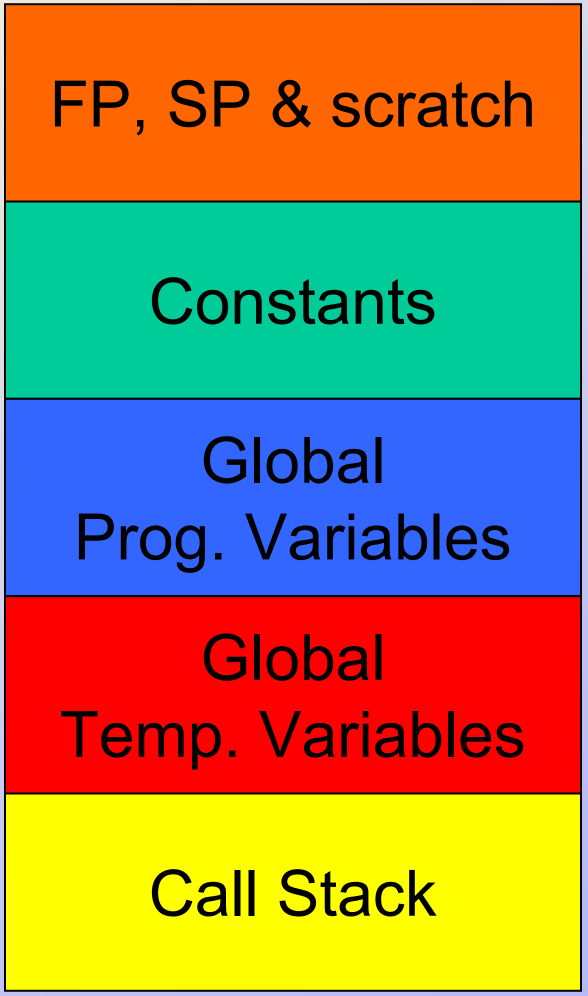
Constants

Global
Prog. Variables

Global
Temp. Variables

Activation Records

Static



FP, SP & scratch

Constants

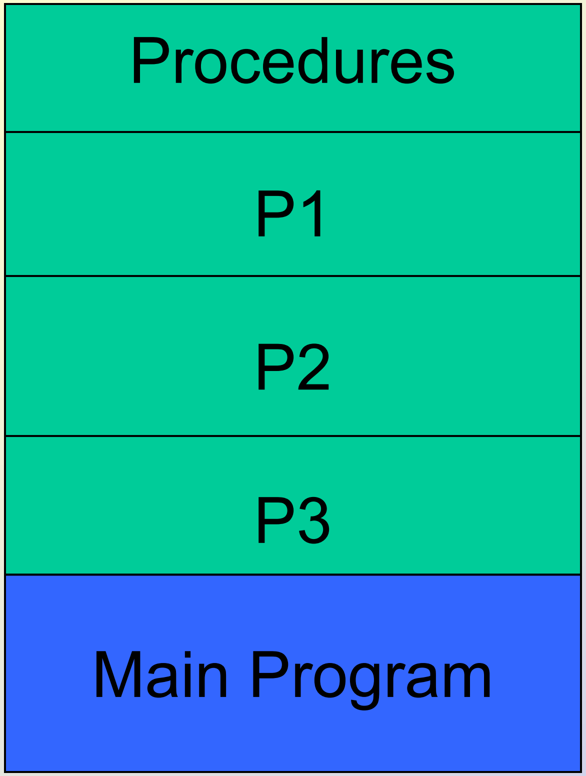
Global
Prog. Variables

Global
Temp. Variables

Call Stack

Dynamic

Program Memory Organization



Procedures

P1

P2

P3

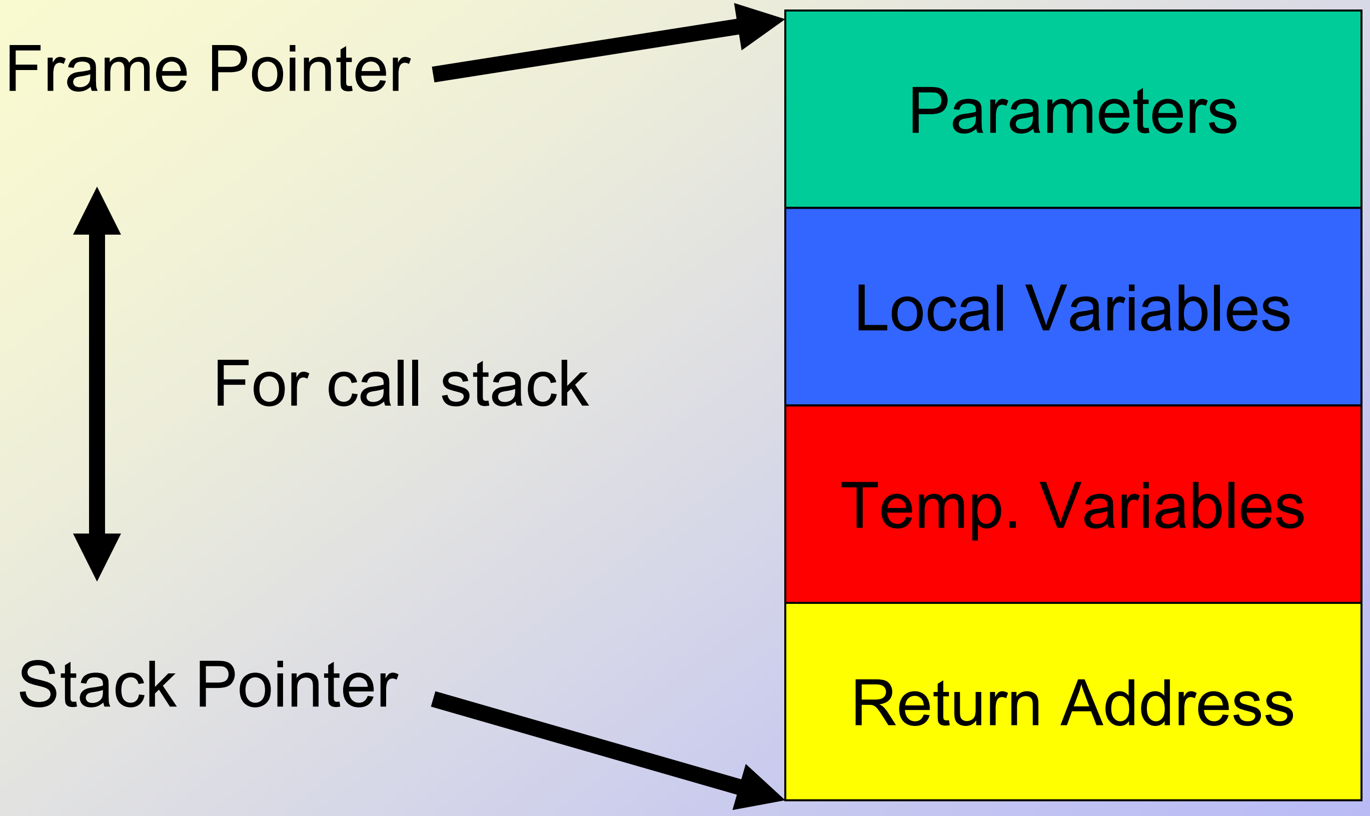
Main Program

Procedure Entry in Function Table

- Number of parameters
- Number of local/temp variables
- Starting address
- Number of instructions

- Need to know starting address of main program

Activation Record



Frame Pointer

For call stack

Stack Pointer

Parameters

Local Variables

Temp. Variables

Return Address

Example: fact(n)

```
define fact
  proc(n)
    i := n;
    f := 1;
       while i do
           f := f * i;
           i := i - 1
       od;
       return := f
end
```

fact(n)

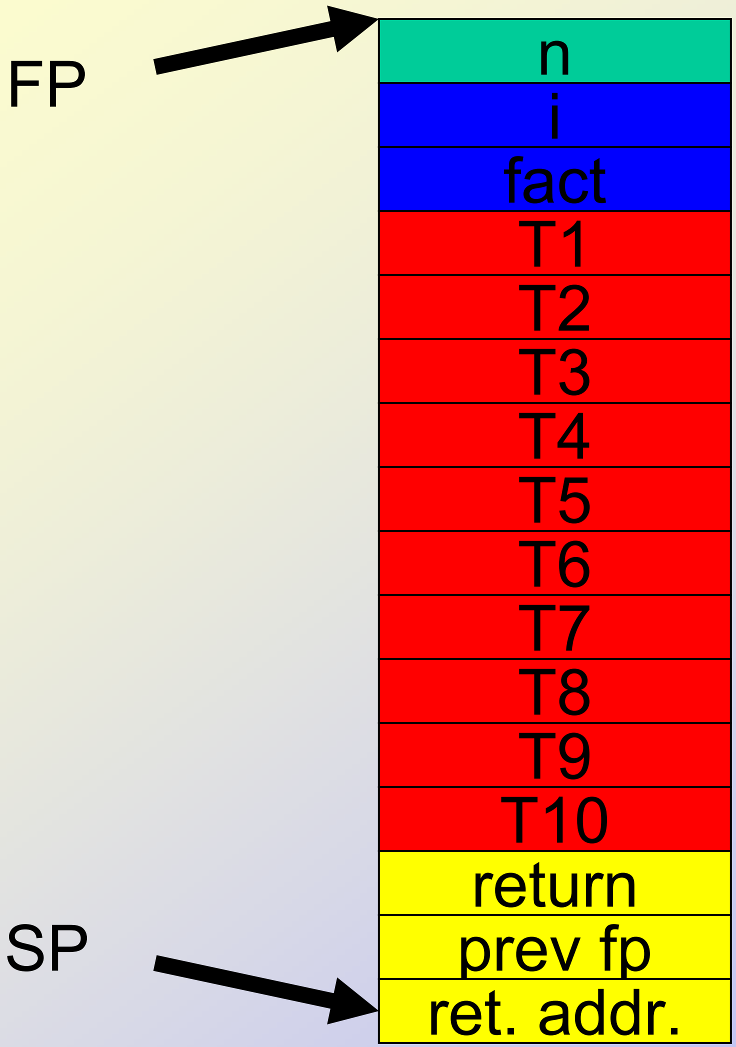
```
LD n
ST T1
LD T1
ST i
LD ONE
ST T2
LD T2
ST fact
```

```
L1:  LD i
     ST T3
     JMN L2
     JMZ L2
     LD fact
     ST T4
     LD i
     ST T5
     LD T4
     MUL T5
     ST T6
     LD T6
     ST fact
```

```
     LD i
     ST T7
     LD ONE
     ST T8
     LD T7
     SUB T8
     ST T9
     LD T9
     ST i
     JMP L1
L2:  LD fact
      ST T10
      LD T10
      ST return
```

Note that addressing (LD/ST) different than main program.
If main were a function the code would be uniform.

Activation Record



Accessing AR

LD n ⇔ LDI FP

ST n ⇔ STI FP

```
LD i  ⇔ LD FP
          ADD ONE
          ST FPB
          LDI FPB
      ⇔ LDO FP[1]
ST i ⇔ STO FP[1]
LD Tj ⇔
LDO FP[j+Num_Param+Num_Vars]
```

Calling Sequence

Initiate call

1. Create activation record
    1. Update FP and SP
2. Store parameters in activation record
3. Store return address (RA)
4. Jump to starting address of procedure code
    1. Introduce call instruction (can place RA relative to SP)
    2. Can compute RA from PC

Return from call

1. Store return value in activation record (when return is assigned)
2. Jump to RA
    1. Introduce ret instruction (jmp indirect)
3. Retrieve return value from activation record
4. Update FP and SP