

# COMP 520 - Compilers

Lecture 18 (April 21, 2022)

## *Register Code Generation*

- PA4 comments
- Lecture covers material not in the text
  - No additional reading beyond slides

# Topics

---

1. mJAM CG for local variable declarations in block scopes
  - where and how to store local variable declarations in a stack machine
  
2. Code generation for RISC architectures
  - RISC: Reduced Instruction Set Architecture
    - register machine, not a stack-machine
  
  - Register use in expression evaluation
    - temporary values
    - lifetime of temporaries
    - expression evaluation order and number of temporaries
  
  - Basic optimization steps
    - tuple code
    - common subexpression elimination
    - register allocation
    - spill code
    - linkage optimization



# PA4: A tricky point in codegen

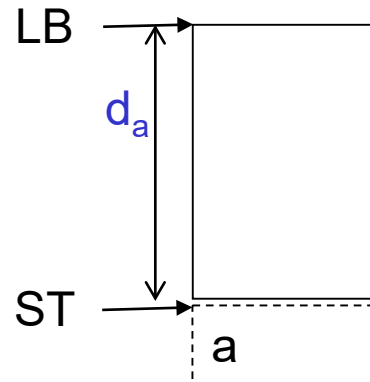
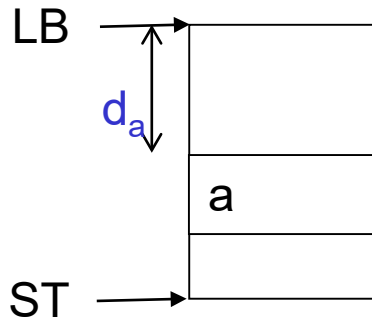
- what's the difference between these two examples?

```
a = new A();
```

```
LOADL -1  
LOADL SA  
CALL newobj  
STORE da[LB]
```

```
A a = new A();
```

```
LOADL -1  
LOADL SA  
CALL newobj  
STORE da[LB]
```



# Frame maintenance with local VarDecls

- What does Triangle do?

```
let
  const b ~ 10;
  var i: Integer;
in
  i := i * b
```

```
LOAD 4[SB]
LOADL 10
mult
STORE 4[SB]
```

(b) Decorated AST with attached entity descriptions:

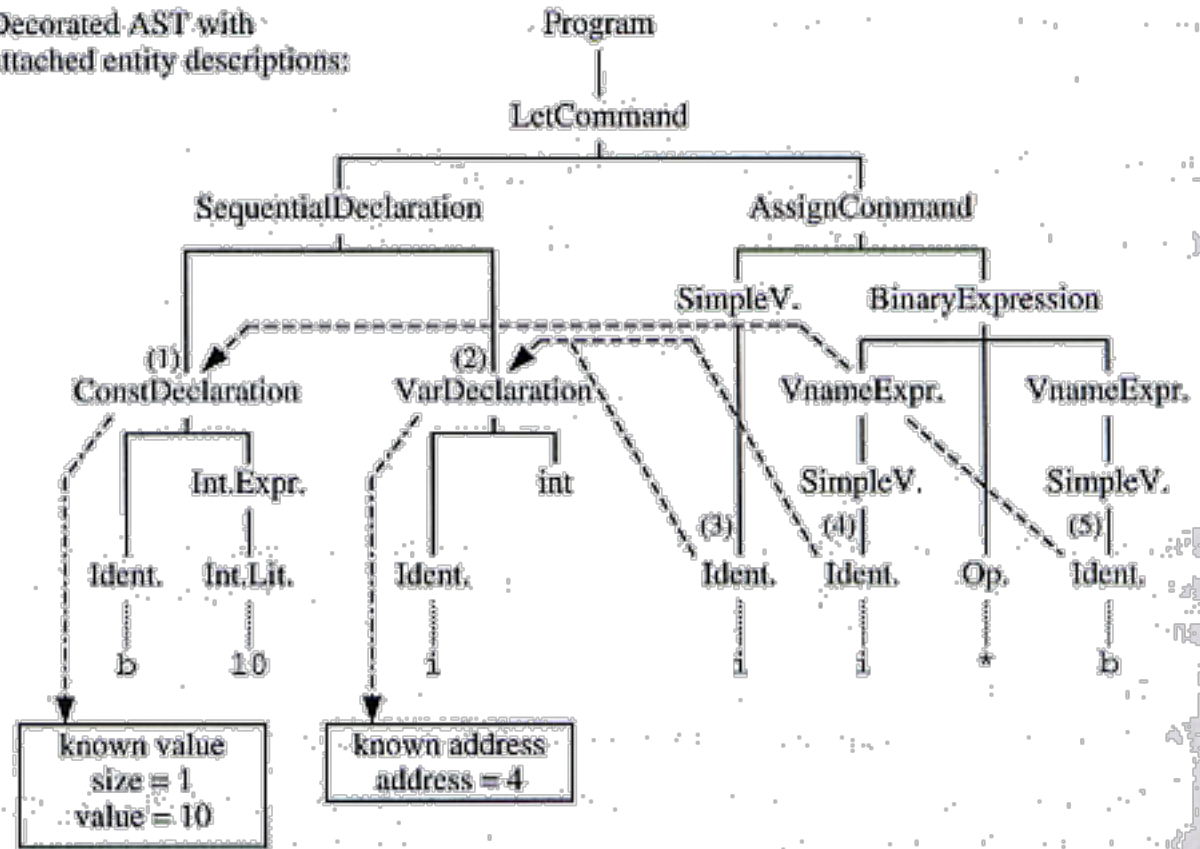
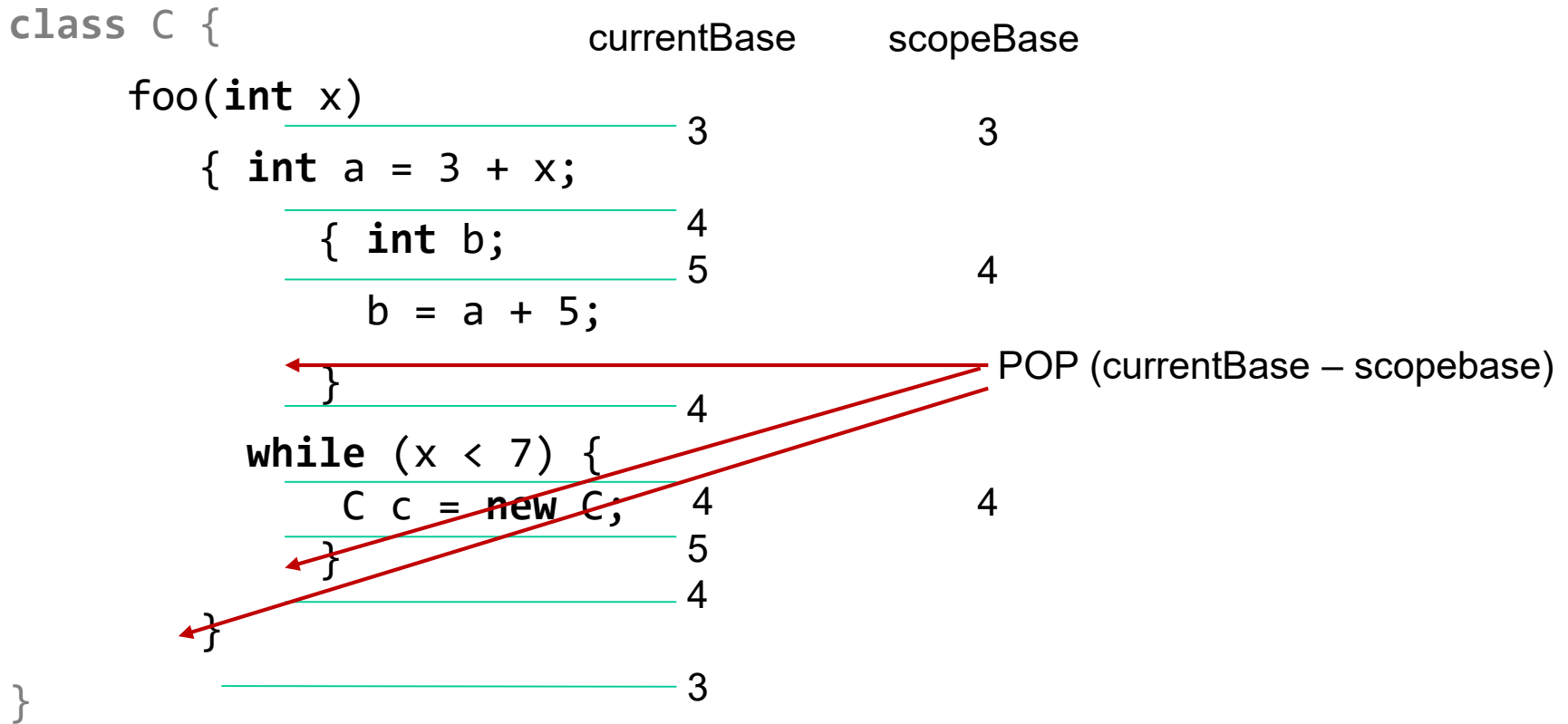


Figure 7.1 Entity descriptions for a known value and a known address.



# Frame maintenance in miniJava nested scopes

- miniJava nested scopes



## What do we need to know?

- BlockStmt RED scopeBase: offset from LB at start of BlockStmt
- currentBase: current offset from LB



# Static variable management in mJAM

---

- what is the lifetime of a static variable?
  - lifetime of the program
- where can we allocate them?
  - stack? heap?
- how much space is needed?
  - # static vars
- when must we allocate them?
  - before the program starts!
- how?
  - in the preamble (before calling main)



# Code generation for RISC instruction sets

---

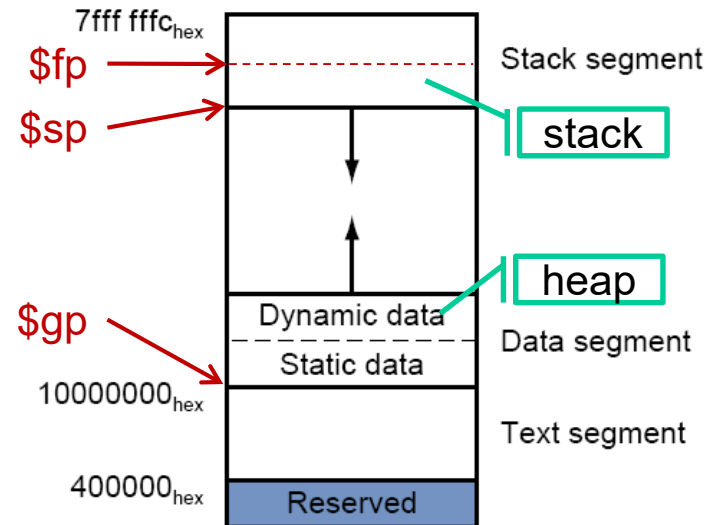
- How does a RISC machine compare to a stack-oriented machine
  - all arithmetic and logical functions operate on registers
    - no explicit stack
  - minimal support for procedure/function call
    - no frame maintenance
  - registers are a compiler-managed cache
- Compiler challenges related to a RISC instruction set
  - large but finite set of registers (32 – 128)
    - register allocation – a problem for the compiler
      - which values should be placed in registers?
      - what should we do when we run out of registers?
  - procedure and function calling conventions
    - no direct support for execution stack
      - frame pointer, stack pointer, and other linkage conventions must be defined
    - flexible strategy for register use
      - needed to limit register save/restore work in call/return
      - parameters and results passed in registers instead of on the stack



# MIPS memory organization

- Key areas

- Reserved
  - for use by operating system
- Text segment
  - compiler-generated program is loaded here
- Stack segment
  - procedure invocation stack
    - expands downwards
    - holds some local variables, some linkage information
- Data segment
  - static constants and variables are placed at the bottom
    - their locations are known by the compiler
  - dynamically allocated values are placed above the static data
    - this area is the heap
    - heap allocation addresses cannot be predicted by the compiler
    - heap expands upwards
    - memory for deleted values can be reused/compacted





# MIPS register conventions

---

Register	Symbolic name	Value	Set by
0	\$zero	Constant zero	Not set
1	\$at	ASM use only	ASM
2-3	\$v0, \$v1	Function results	Callee
4-7	\$a0, \$a3	Functions args 1-4	Caller
8-15	\$t0, \$t7	Temp – not preserved	
16-23	\$s0, \$s7	Temp – preserved	
24-25	\$t8, \$t9	Temp – not preserved	
26-27	\$k0, \$k1	OS use only	OS
28	\$gp	Global pointer	OS
29	\$sp	Stack pointer	Callee
30	\$fp	Frame pointer	Callee
31	\$ra	Return address	Caller

- \$s0 - \$s7 are expected to be preserved across a function call
- \$t0 - \$t9 are not expected to be preserved across a function call



# Example Code 1 – simple statement

*miniJava*

```
x = y + z;
```

*MIPS assembler*

```
lw    $t0, @y
lw    $t1, @z
add   $t2, $t0, $t1
sw    $t2, @x
```

- **Notes**

- x, y, z are local int vars
  - represented as 32-bit binary values
  - word-aligned
  - @y stands for address of y in memory
    - typically some offset added to \$fp (local vars) or \$gp (global vars)
- all operations take place between registers
  - values of y, z are fetched from frame
  - value of result is stored into x in frame



# Example Code 2 – multiple statements

```
x = y + z;  
w = y - z;
```

```
lw    $t0, @y  
lw    $t1, @z  
add   $t2, $t0, $t1  
sw    $t2, @x  
lw    $t0, @y  
lw    $t1, @z  
sub   $t2, $t0, $t1  
sw    $t2, @w
```

register allocation  
within a statement

```
lw    $t0, @y  
lw    $t1, @z  
add   $t2, $t0, $t1  
sw    $t2, @x  
sub   $t3, $t0, $t1  
sw    $t3, @w
```

register allocation  
across two statements



# Code Example 3 – multiple execution paths

```
if (x < y)
    x = y + 1;
```

```
lw    $t0, @x
lw    $t1, @y
bge   $t0, $t1, L10
addi  $t0, $t1, 1
sw    $t1, @x
L10:
```

register allocation  
across execution  
paths

```
while (x != y) {
    if (x > y)
        x = x - y;
    else
        y = y - x;
}
```

```
lw    $t0, @x
lw    $t1, @y
Ltop:
beq   $t0, $t1, Lend
ble   $t0, $t1, Lelse
Lthen:
sub   $t0, $t0, $t1
b     Ltop
Lelse:
sub   $t1, $t1, $t0
b     Ltop
Lend:
sw    $t0, @x
sw    $t1, @y
```

register allocation  
across loops –  
large payoff



# Example Code 4 – function call

```
x = x + f(x) + x;
```

```
lw    $s0, @x
mv    $a0, $s0
jal   @f
aw    $t0, $s0, $v0
aw    $t0, $t0, $s0
sw    $t0, @x
```

Function invocation in an expression

```
int f(x){ return 3 + x;}
```

```
f:
subiu $sp, $sp, 8
sw    $ra, 8($sp)
sw    $fp, 4($sp)
addiu $fp, $sp, 8

addiu $v0, $a0, 3

lw    $ra, 8($sp)
lw    $fp, 4($sp)
addiu $sp, $sp, 8
jr    $ra
```

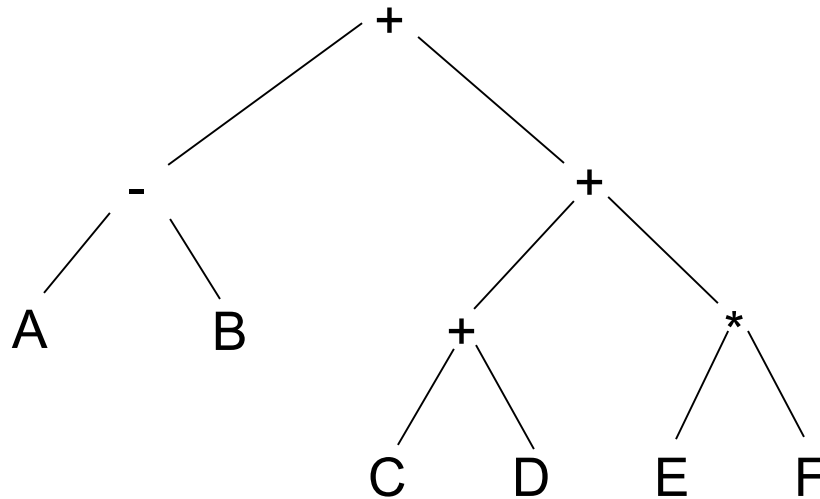
Function body  
explicit frame/stack management  
can be omitted in this case!



# Expression evaluation

- How many registers are needed to evaluate an expression?
  - using our stack-based expression evaluation strategy we could use a new register for each temporary value

$(A - B) + ((C + D) + (E * F))$



```
t1 := A
t2 := B
t3 := t1 - t2
t4 := C
t5 := D
t6 := t4 + t5
t7 := E
t8 := F
t9 := t7 * t8
t10 := t6 + t9
t11 := t3 + t10
```

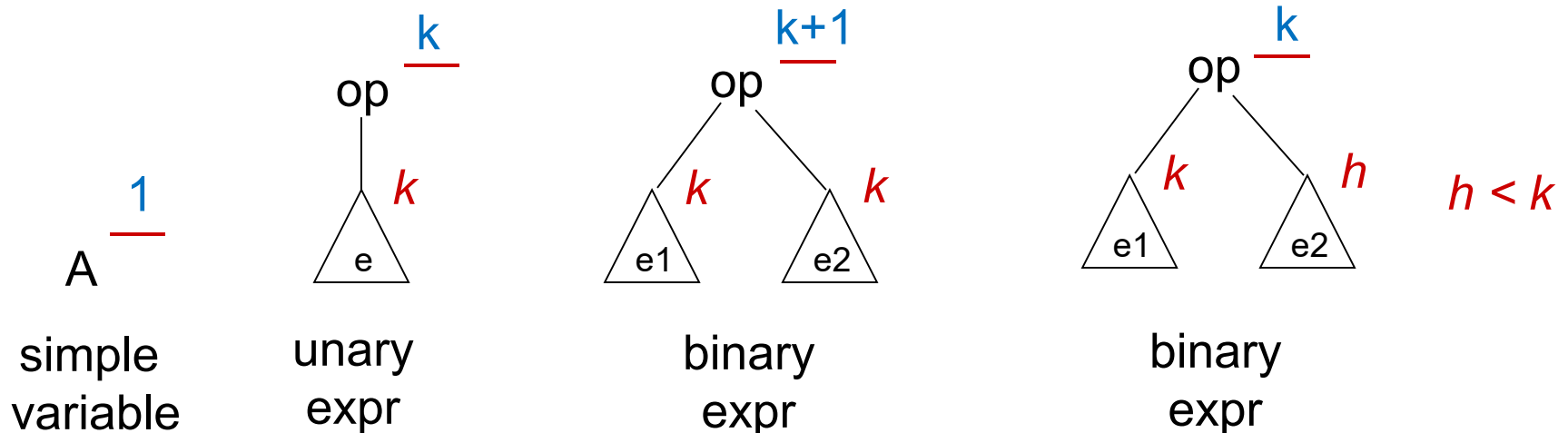
Tuple code:  
each  $t_i$  is a  
*temporary  
value*

We may easily run out of registers using this strategy !



# Expression evaluation

- Minimizing the number of registers needed
  - Sethi-Ullman labeling
    - each subexpression is labeled with the minimum number of registers needed to evaluate it
    - defined inductively by the rules below
  - rearrange expression evaluation order to minimize register use
    - evaluate the subexpression requiring more registers first

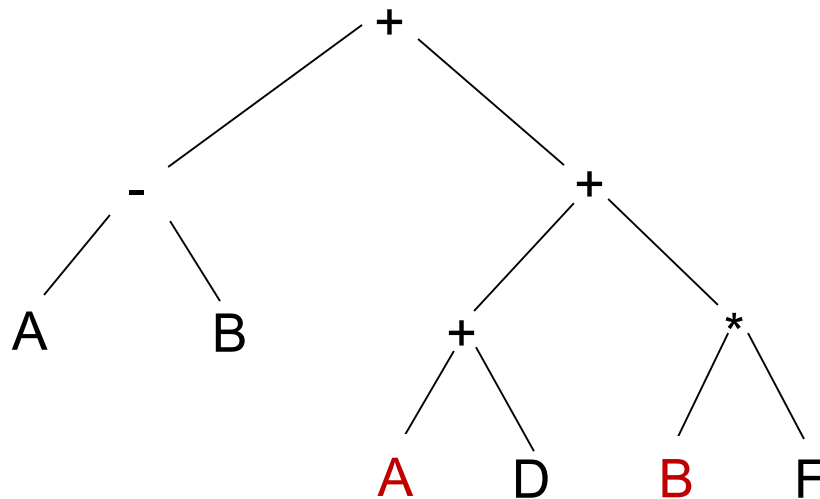


# Sethi-Ullman labeling and expression reordering

- **Questions**

- is evaluation order rearrangement always possible?
- is minimal register usage always best?

$(A - B) + ((C + D) + (E * F))$



$t1 := A$

$t2 := D$

$t1 := t1 + t2$

$t2 := B$

$t3 := F$

$t2 := t2 * t3$

$t1 := t1 + t2$

$t2 := A$

$t3 := B$

$t2 := t2 - t3$

$t1 := t2 + t1$





# Expression evaluation

- Reusing registers

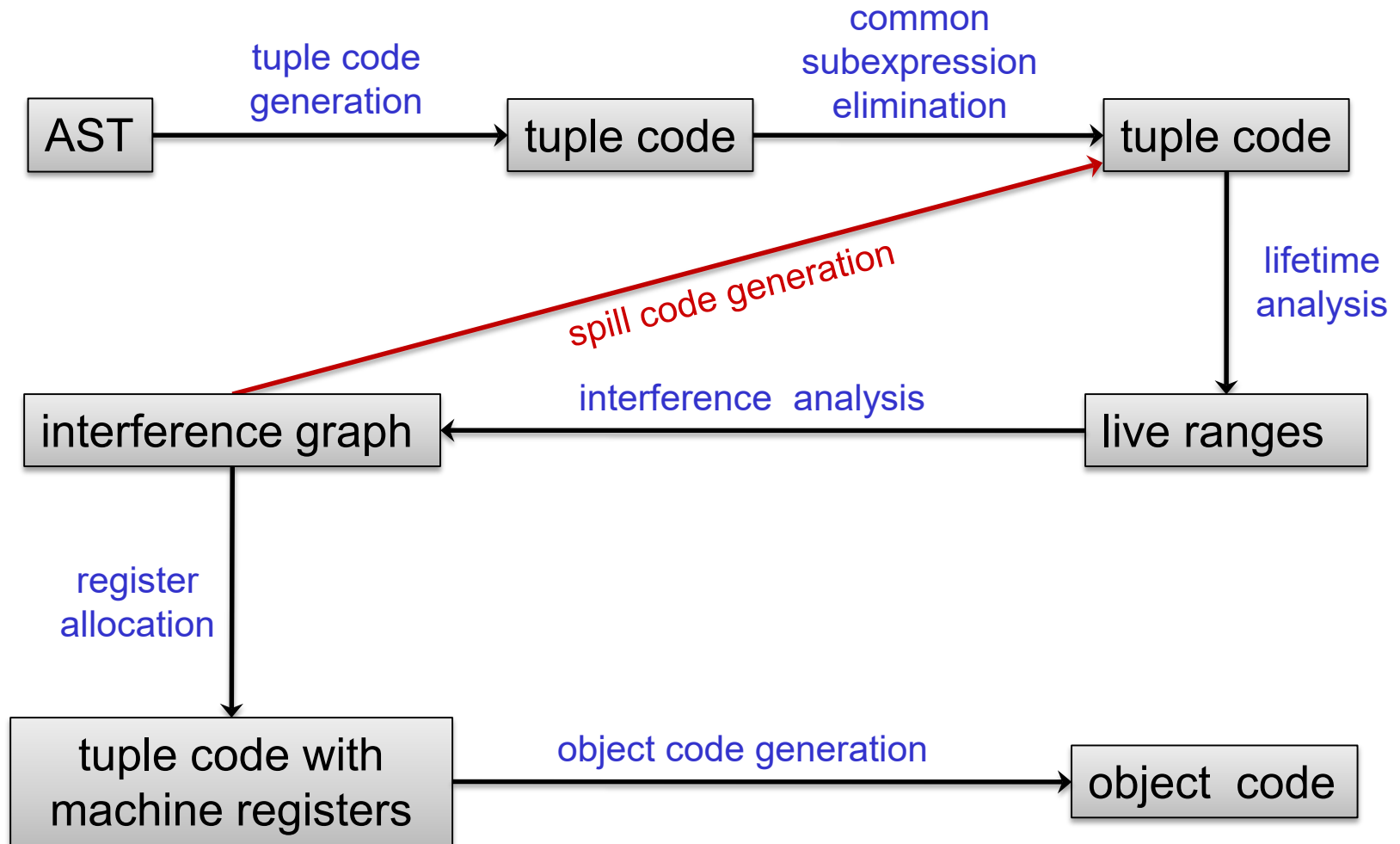
$$(A - B) + ((A + D) + (B * F))$$

- Consider tuple code - how long is each temporary value needed?
  - Lifetime starts at definition, ends at last use
- values with disjoint lifetimes may use the same register

	t1	t2	t3	t4	t5	t6	t7	t8	t9
t1 := A									
t2 := B									
t3 := t1 - t2									
t4 := D									
t5 := t1 + t4									
t6 := F									
t7 := t2 * t6									
t8 := t5 + t7									
t9 := t3 + t8									

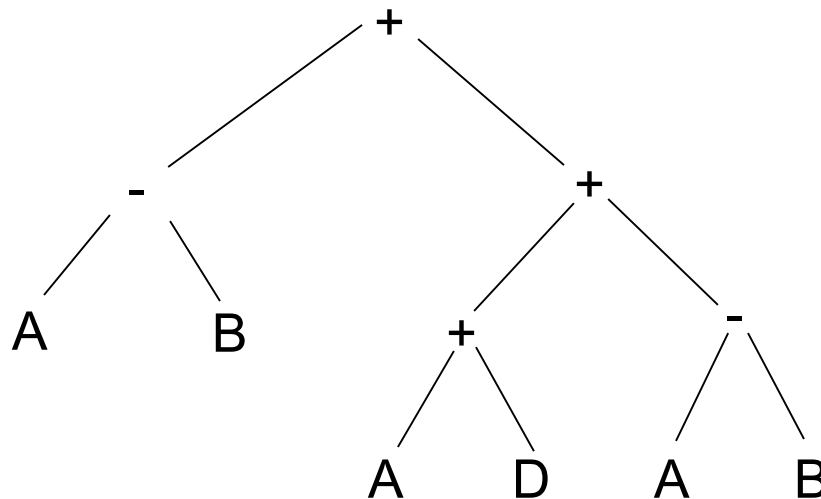
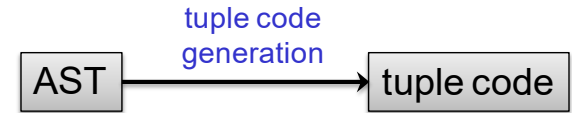


# Register allocation for expressions



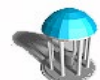
# Tuple code generation

- **Tuple code**
  - list of tuples
  - each tuple represents a machine instruction
    - (destination, operation, operands)
  - use temporaries rather than registers
  - operations at level of target instruction set
- **Tuple code generation**
  - postorder traversal of AST



```
t1 := A
t2 := B
t3 := t1 - t2
t4 := A
t5 := D
t6 := t4 + t5
t7 := A
t8 := B
t9 := t7 - t8
t10 := t6 + t9
t11 := t3 + t10
```

Example:  $(A - B) + ((A + D) + (A - B))$



# Common subexpression elimination



- Replace redundant loads and computations

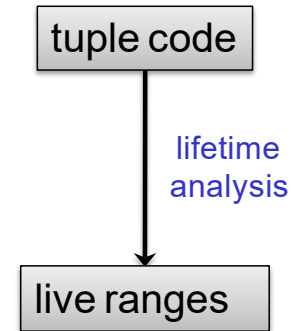
Example:  $(A - B) + ((A + D) + (A - B))$

<u>Tuple code</u>	<u>repl tuples by copy</u>	<u>subst, del copy</u>	<u>renumb temps</u>
t1 := A	t1 := A	t1 := A	t1 := A
t2 := B	t2 := B	t2 := B	t2 := B
t3 := t1 - t2	t3 := t1 - t2	t3 := t1 - t2	t3 := t1 - t2
t4 := A	t4 := t1		
t5 := D	t5 := D	t5 := D	t4 := D
t6 := t4 + t5	t6 := t4 + t5	t6 := t1 + t5	t5 := t1 + t4
t7 := A	t7 := t1		
t8 := B	t8 := t2		
t9 := t7 - t8	t9 := t3		
t10 := t6 + t9	t10 := t6 + t9	t10 := t6 + t3	t6 := t5 + t3
t11 := t3 + t10	t11 := t3 + t10	t11 := t3 + t10	t7 := t3 + t6

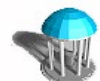
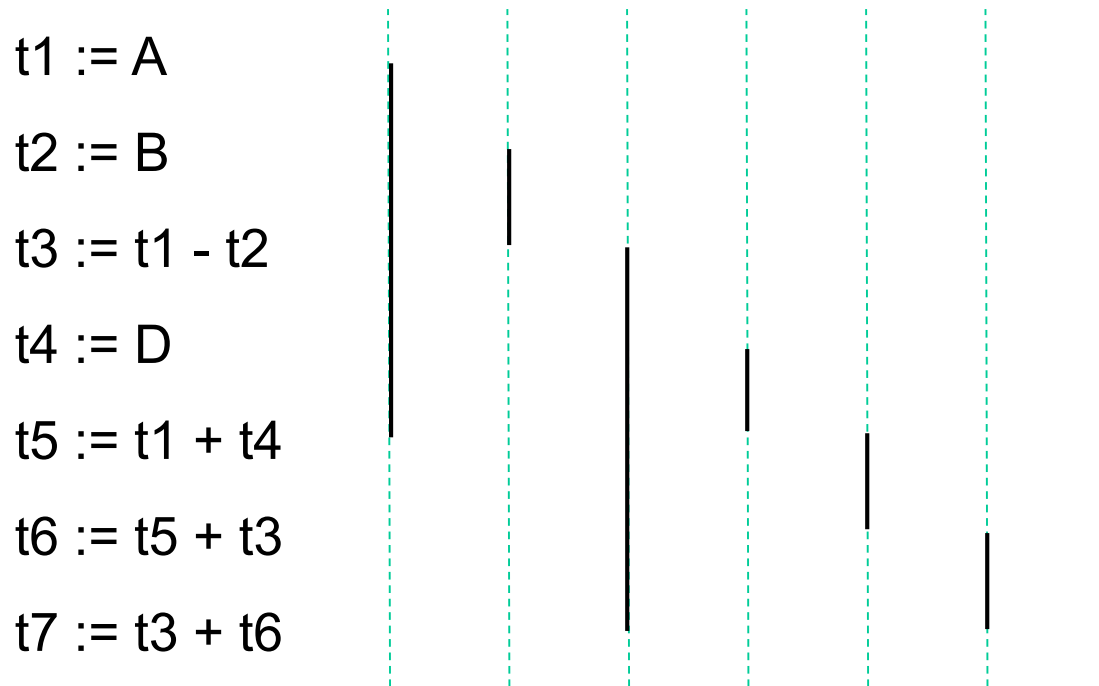


# Lifetime analysis

- For each temporary
  - construct interval from first definition to last use



t1    t2    t3    t4    t5    t6    t7



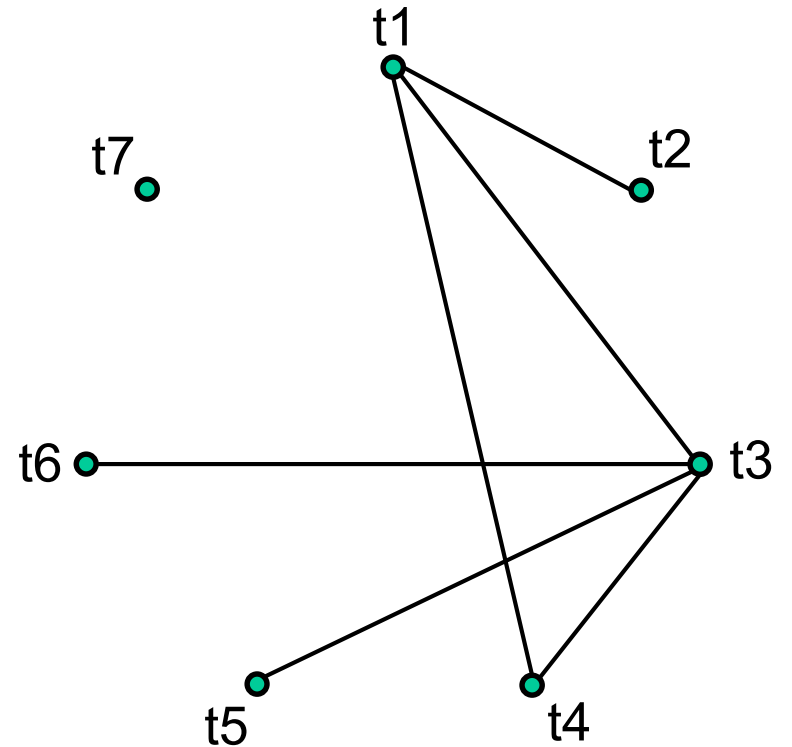
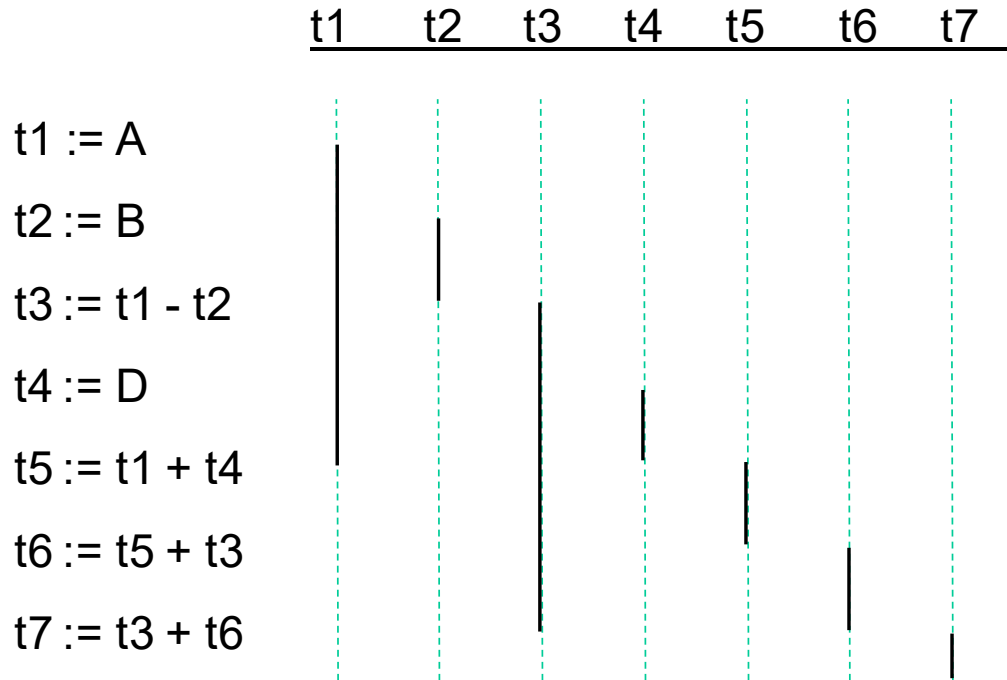
# Allocating temporaries to registers



- **Create interference graph**
  - node for each temporary
  - edge  $(u,v)$  if live range of  $u$  overlaps live range of  $v$
- **Allocate tuple code temporaries to registers**
  - temporaries connected by an edge in the interference graph must be in distinct registers
    - because the lifetimes of the two temporaries overlap
  - can all temporaries be allocated to  $k$  registers?
    - possible iff interference graph can be  $k$ -colored



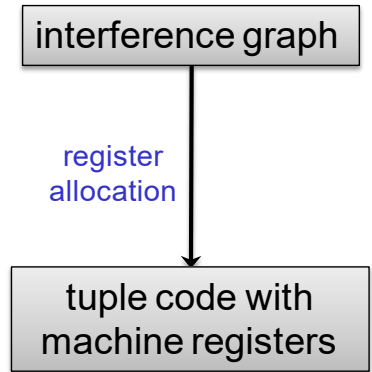
# Interference graph



# Register Allocation

---

- **k-coloring interference graph**
  - k is number of available registers
  - k-coloring is an NP-complete problem
  - linear time greedy heuristic algorithm (may fail)
    - repeatedly remove a node and all associated edges from a graph and add to a list
      - choose a node with indegree  $< k$  whenever possible
    - when graph is empty, color nodes in reverse of order removed
  - if heuristic algorithm fails
    - pick a temporary t with a long lifetime and indegree  $\geq k$
    - **Generate spill code**
      - insert tuple operation to store t when generated and fetch t where needed
      - repeat lifetime analysis, interference analysis
- **Additional considerations**
  - constrain temporaries for function arguments to appropriate machine registers
  - constrain temporaries live across a call to callee-preserved registers (i.e. s0-s7)





# Greedy k-coloring algorithm [Kempe 1879]

- Given undirected graph  $G = (V, E)$ , try to color the nodes in  $V$  using  $k$  colors so that for any  $(u, v) \in E$ ,  $\text{color}(u) \neq \text{color}(v)$

Stack of nodes  $S$  initially empty

Undirected graph  $G = (V, E)$

**while**  $G \neq \{\}$  **do**

    choose some minimum degree node  $t$  in  $V$

$S.\text{push}(t)$

$G.\text{remove}(t)$  // remove  $t$  and edges incident on  $t$  from  $G$

**end**

**while**  $S \neq \{\}$  **do**

$t = S.\text{pop}()$

$G.\text{insert}(t)$  // add  $t$  and edges in  $G$  incident on  $t$

    color  $t$ , if possible

**end**

} simplify

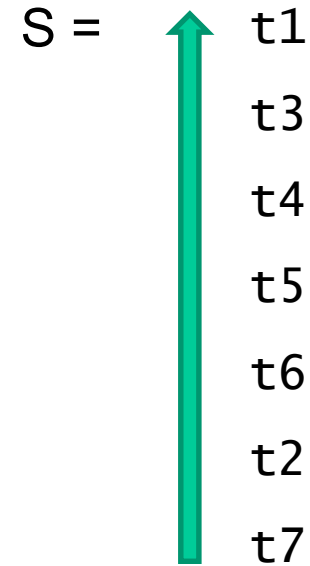
} color



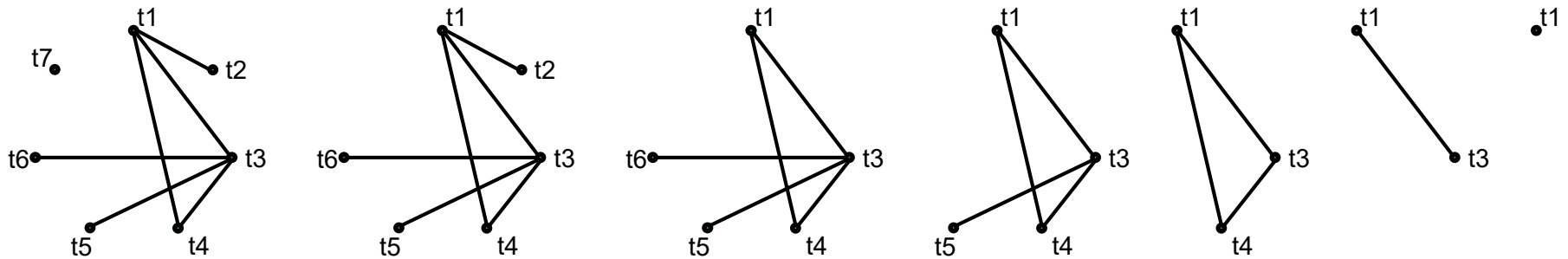
# Kempe algorithm – simplify step

Stack of nodes  $S$  initially empty  
Undirected graph  $G = (V, E)$

```
while  $G \neq \{\}$  do  
  choose some minimum degree  
  node  $t$  in  $V$   
   $S.push(t)$   
   $G.remove(t)$ 
```



$G =$



# Kempe algorithm – color step


Stack of nodes  $S$  to color in order

Undirected graph  $G = (V, E)$  initially empty

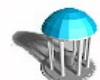
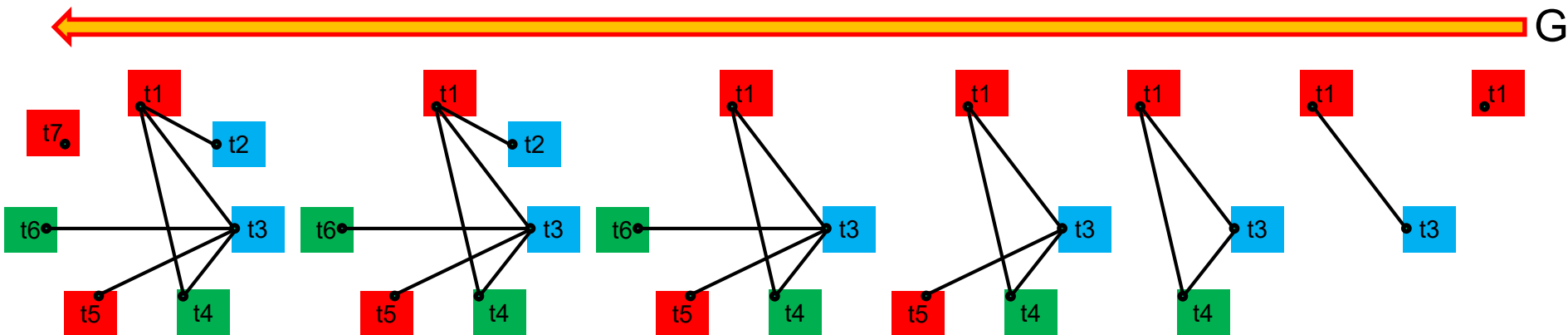
```
while  $S \neq \{\}$  do
   $t = S.pop()$ 
   $G.insert(t)$  // add  $t$  and edges in  $G$  incident on  $t$ 
  color  $t$ , if possible
end
```

$k = 3$  colors: ■ ■ ■

$S =$



t1  
t3  
t4  
t5  
t6  
t2  
t7



# Register Allocation

- rewrite tuple code with register allocation

t1 := A

t2 := B

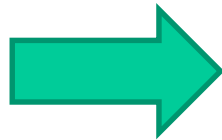
t3 := t1 - t2

t4 := D

t5 := t1 + t4

t6 := t5 + t3

t7 := t3 + t6



r1 := A

r2 := B

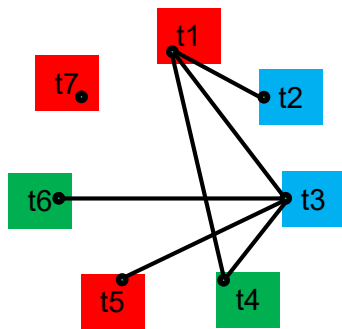
r2 := r1 - r2

r3 := D

r1 := r1 + r3

r3 := r1 + r2

r1 := r2 + r3



r1

r2

r3

interference graph

register  
allocation

tuple code with  
machine registers



# Putting it together : RISC code generation

