Let‘s build a compiler

Why build a compiler?

```c
int main() {
  int n = 5;
  int res;
  if (n > 1) {
    int curr = 0;
    int prev1 = 1;
    int prev2 = 0;

    while (n > 0) {
        curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
        n--;
    }
    res = curr;
  } else { res = n; }

  return res;
}
```

C--

Python?

Go?

Rust?

ARM?

RISC-V?

HANS?

x86

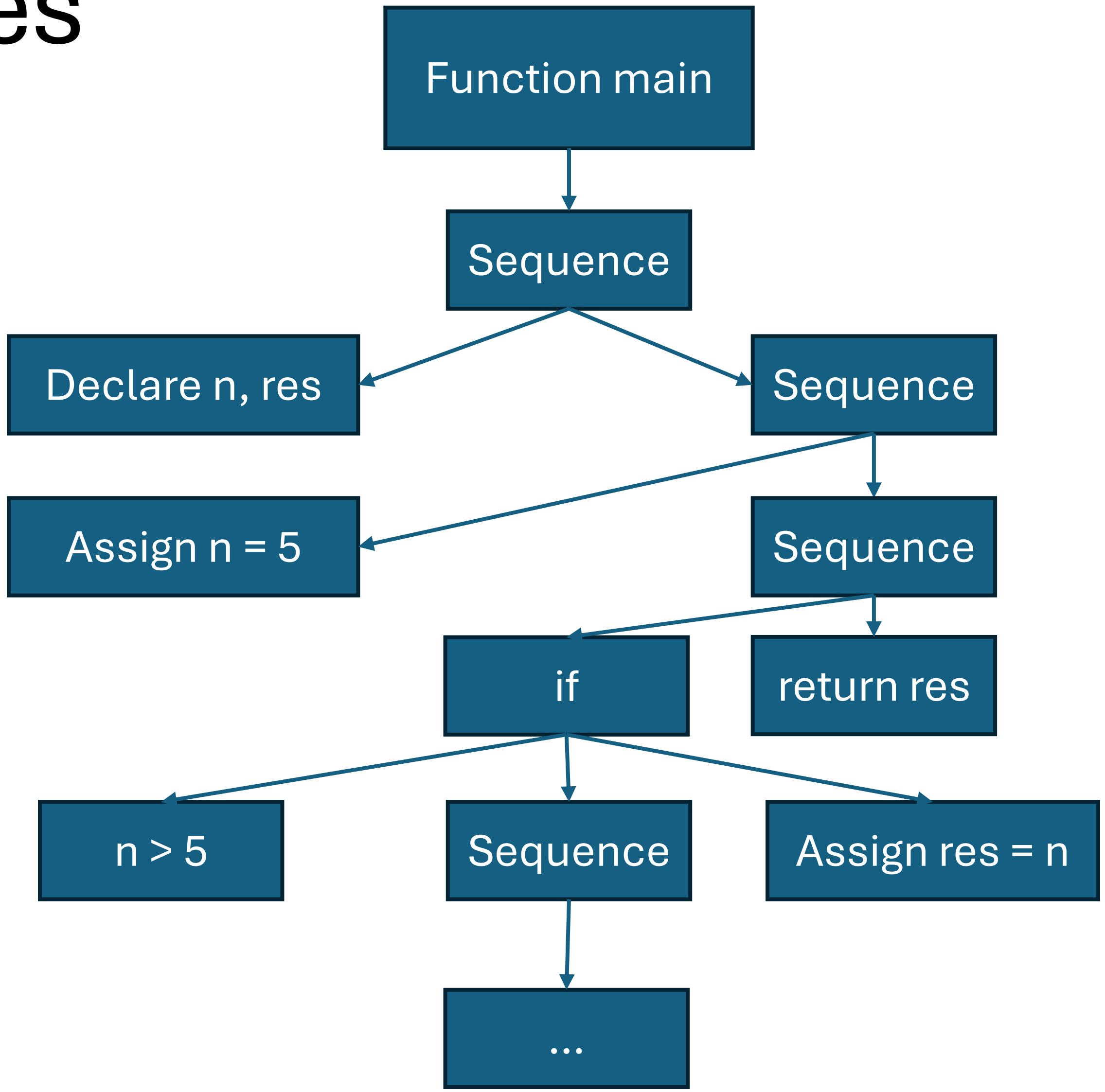
```asm
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-20], edi
        cmp     DWORD PTR [rbp-20], 1
        jle     .L2
        mov     DWORD PTR [rbp-8], 0
        mov     DWORD PTR [rbp-12], 1
        mov     DWORD PTR [rbp-16], 0
        jmp     .L3
.L4:
        mov     edx, DWORD PTR [rbp-12]
        mov     eax, DWORD PTR [rbp-16]
        add     eax, edx
        mov     DWORD PTR [rbp-8], eax
        mov     eax, DWORD PTR [rbp-12]
        mov     DWORD PTR [rbp-16], eax
        mov     eax, DWORD PTR [rbp-8]
        mov     DWORD PTR [rbp-12], eax
        sub     DWORD PTR [rbp-20], 1
.L3:
        cmp     DWORD PTR [rbp-20], 0
        jg      .L4
        mov     eax, DWORD PTR [rbp-8]
        mov     DWORD PTR [rbp-4], eax
        jmp     .L5
.L2:
        mov     eax, DWORD PTR [rbp-20]
        mov     DWORD PTR [rbp-4], eax
.L5:
        mov     eax, DWORD PTR [rbp-4]
        pop     rbp
        ret
```

What the computer sees

```
         00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
00000000 69 6E 74 20 6D 61 69 6E 28 29 20 7B 0D 0A 20 20  int main() {..  
00000010 69 6E 74 20 6E 20 3D 20 35 3B 0D 0A 20 20 69 6E  int n = 5;..  in
00000020 74 20 72 65 73 3B 0D 0A 20 20 69 66 20 28 6E 20  t res;..  if (n 
00000030 3E 20 31 29 20 7B 0D 0A 20 20 20 20 69 6E 74 20  > 1) {..    int 
00000040 63 75 72 72 20 3D 20 30 3B 0D 0A 20 20 20 20 69  curr = 0;..    i
00000050 6E 74 20 70 72 65 76 31 20 3D 20 31 3B 0D 0A 20  nt prev1 = 1;.. 
00000060 20 20 20 69 6E 74 20 70 72 65 76 32 20 3D 20 30     int prev2 = 0
00000070 3B 0D 0A 0D 0A 20 20 20 20 77 68 69 6C 65 20 28  ;....    while (
00000080 6E 20 3E 20 30 29 20 7B 0D 0A 20 20 20 20 20 20  n > 0) {..      
00000090 20 20 63 75 72 72 20 3D 20 70 72 65 76 31 20 2B    curr = prev1 +
000000A0 20 70 72 65 76 32 3B 0D 0A 20 20 20 20 20 20 20   prev2;..       
000000B0 20 70 72 65 76 32 20 3D 20 70 72 65 76 31 3B 0D   prev2 = prev1;.
000000C0 0A 20 20 20 20 20 20 20 20 70 72 65 76 31 20 3D  .        prev1 =
000000D0 20 63 75 72 72 3B 0D 0A 20 20 20 20 20 20 20 20   curr;..        
000000E0 6E 2D 2D 3B 0D 0A 20 20 20 20 7D 0D 0A 20 20 20  n--;..    }..   
000000F0 20 72 65 73 20 3D 20 63 75 72 72 3B 0D 0A 20 20   res = curr;..  
00000100 7D 20 65 6C 73 65 20 7B 20 72 65 73 20 3D 20 6E  } else { res = n
00000110 3B 20 7D 0D 0A 0D 0A 20 20 72 65 74 75 72 6E 20  ; }....  return 
00000120 72 65 73 3B 0D 0A 7D +                           res;..}+
```

What the computer sees

```
int main() {
  int n = 5;
  int res;
  if (n > 1) {
    int curr = 0;
    int prev1 = 1;
    int prev2 = 0;

    while (n > 0) {
        curr = prev1 + prev2;
        prev2 = prev1;
        prev1 = curr;
        n--;
    }
    res = curr;
  } else { res = n; }

  return res;
}
```

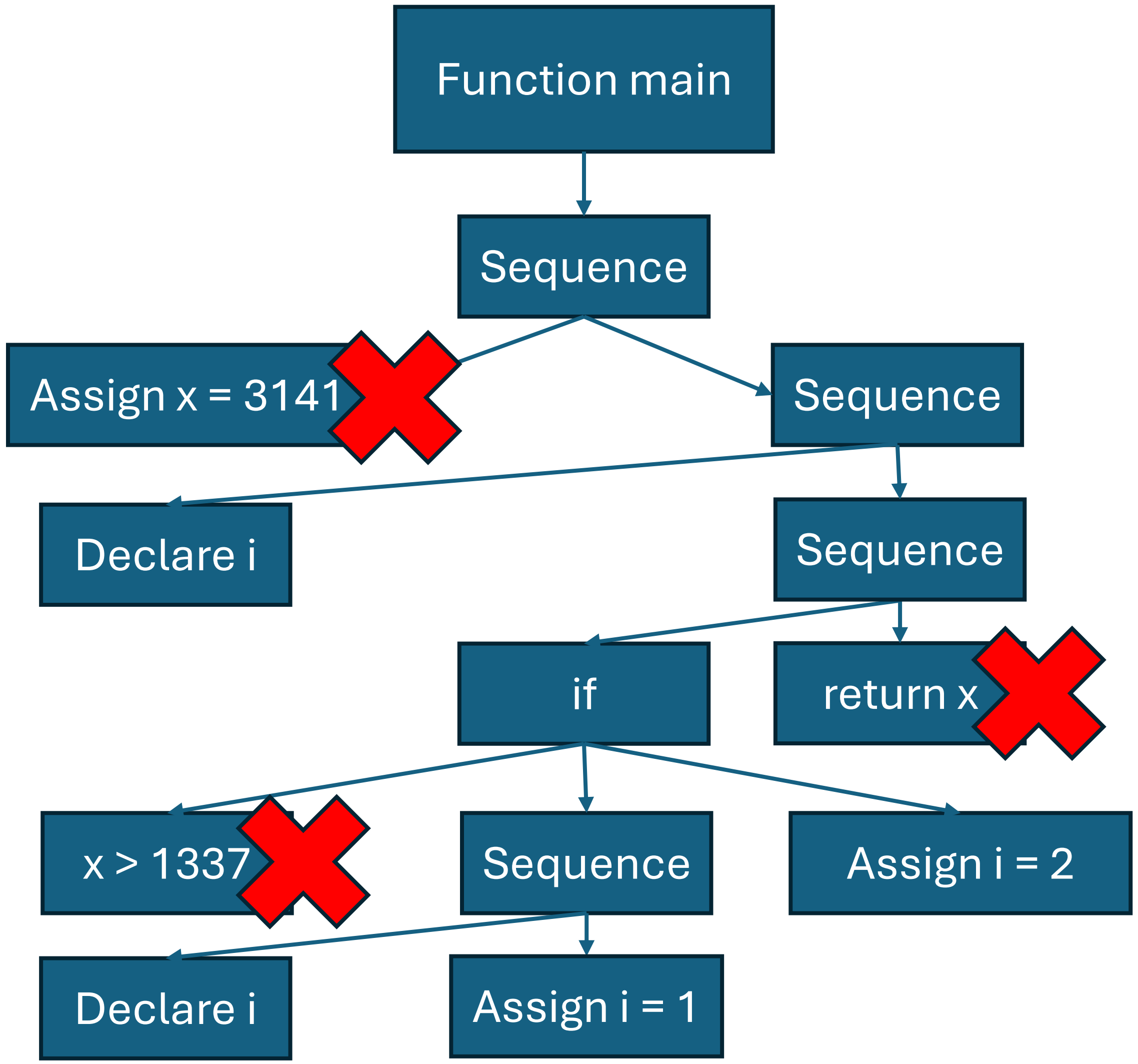


Building the Abstract Syntax Tree

```
fn parse_stmt(input: &str) -> IResult<&str, ast::Stmt> {
    alt((parse_control, parse_expr, parse_blk,))(input)
}

fn parse_expr(input: &str) -> IResult<&str, ast::Stmt> {
    alt((parse_assignment, parse_arithm, map(tuple((tag("("), parse_expr, tag(")"))), |(_, expr, _)|
expr), parse_var,))(input)
}

fn parse_blk(input: &str) -> IResult<&str, ast::Stmt> {
    let (input, (_, stmts, _)) = tuple((tag("{"), many0(parse_stmt), tag("}")))(input)?;
    Ok((input, ast::Stmt::Block(stmts)))
}

fn parse_control(input: &str) -> IResult<&str, ast::Stmt> {
    let (input, control) = alt((parse_if, parse_while, parse_return))(input)?;
    Ok((input, ast::Stmt::Control(control)))
}

fn parse_if(input: &str) -> IResult<&str, ast::Control> {
    let (input, (_, cond, then_stmt, else_stmt)) = tuple((tag("if"), parse_expr, parse_stmt,
opt(preceded(tag("else"), parse_stmt)), ))(input)?;
    Ok((input, ast::Control::If(cond_expr, Box::new(then_stmt), else_stmt.map(Box::new))))
}
```

Semantic Analysis

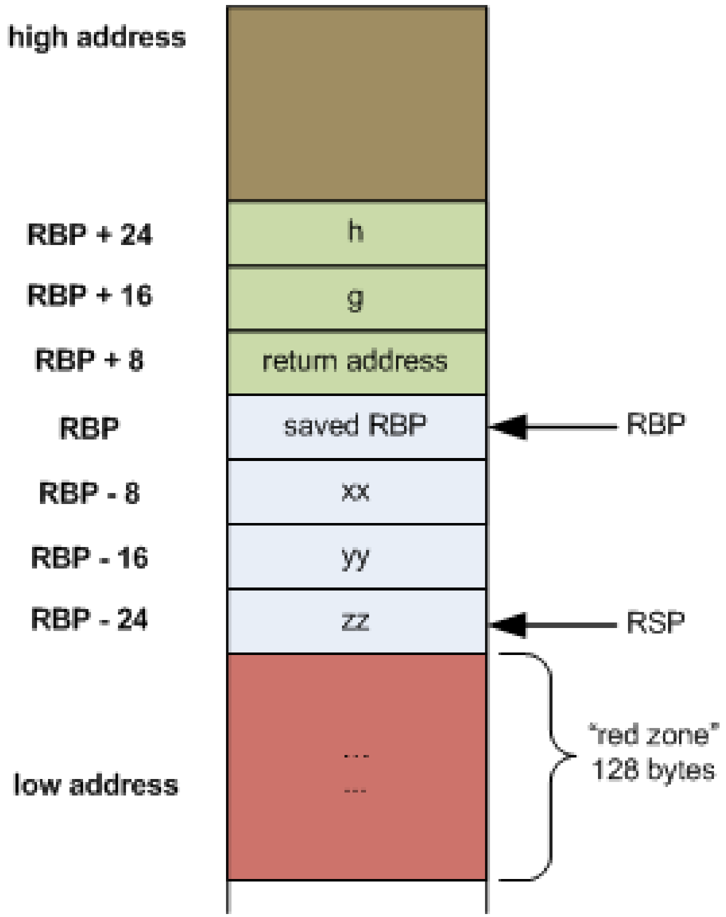
- We create a Symbol Table to uniquely identify variables (and constants, objects, functions...)
- The Symbol Table contains all useful information about the variables
  - Memory Location, Type...

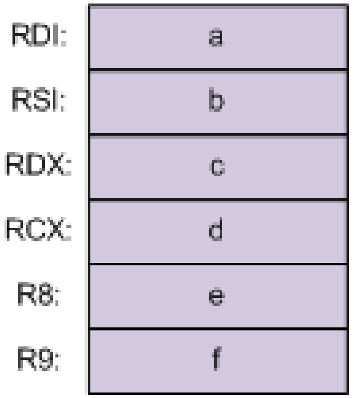


Symbol Table

```rust
impl SymbolTable {
    fn new() -> SymbolTable {
        SymbolTable {
            scopes: vec![HashMap::new()],
            next_reg: 1,
        }
    }

    fn add_var(&mut self, name: String) -> u32 {
        let reg = self.next_reg;
        let len = self.scopes.len();
        self.next_reg += 1;
        self.scopes[len-1].insert(name, reg);
        reg
    }

    fn get_var(&self, name: &str) -> Option<u32> {
        for scope in self.scopes.iter().rev() {
            if let Some(reg) = scope.get(name) { return Some(*reg); }
        } None
    }

    fn enter_scope(&mut self) -> () { self.scopes.push(HashMap::new()); }

    fn exit_scope(&mut self) -> () { self.scopes.pop(); }
}
```

Assembly and the Stack






```
main:
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], 5
        cmp     DWORD PTR [rbp-4], 1
        jle     .L2
        mov     DWORD PTR [rbp-12], 0
        mov     DWORD PTR [rbp-16], 1
        mov     DWORD PTR [rbp-20], 0
        jmp     .L3
.L4:
        mov     edx, DWORD PTR [rbp-16]
        mov     eax, DWORD PTR [rbp-20]
        add     eax, edx
        mov     DWORD PTR [rbp-12], eax
        mov     eax, DWORD PTR [rbp-16]
        mov     DWORD PTR [rbp-20], eax
        mov     eax, DWORD PTR [rbp-12]
        mov     DWORD PTR [rbp-16], eax
        sub     DWORD PTR [rbp-4], 1
.L3:
        cmp     DWORD PTR [rbp-4], 0
        jg      .L4
        mov     eax, DWORD PTR [rbp-12]
        mov     DWORD PTR [rbp-8], eax
        jmp     .L5
.L2:
        mov     eax, DWORD PTR [rbp-4]
        mov     DWORD PTR [rbp-8], eax
.L5:
        mov     eax, DWORD PTR [rbp-8]
        pop     rbp
        ret
```

Runtime

- No function calls → all in one stack frame
- We can‘t allocate registers yet
  - Every variable gets saved to the stack
  - When we need it, we load it from the stack
  - Whenever an expression is calculated, ist result gets put in `rax`

Code Generation

```rust
fn generate_while(cond: &Expr, body: &Block, symtab: &mut SymbolTable, output: &mut String) -> () {
    let mut cond_output = String::new();
    generate_expr(cond, symtab, &mut cond_output);
    let mut body_output = String::new();
    generate_block(body, symtab, &mut body_output);
    let label_begin = symtab.get_label();
    let label_end = symtab.get_label();
    output.push_str(&format!(".Lbegin{}:\n", label_begin));
    output.push_str(&format!("{}cmp rax, 0\nje .Lend{}\n", cond_output, label_end));
    output.push_str(&body_output);
    output.push_str(&format!("jmp .Lbegin{}\n", label_begin));
    output.push_str(&format!(".Lend{}:\n", label_end));
}
```

Comparison to rustc

Our compiler

- Source
- AST
- .asm

rustc

- Source
- AST
- High-Level IR
- THIR
- Mid-Level IR
- LLVM IR
- .asm

LLVM Intermediate Representation

- „High-Level Assembly“
- Single Static Assignment (SSA)

  ⇒ Infinite Registers
- Typed
- Control Flow Graph:
  - Basic Blocks without branches
  - Basic Block starts with a label and ends with a branch
  - Arranged in a directed Graph

```
define i32 @factorial(i32) {
entry:
  %eq = icmp eq i32 %0, 0   // n == 0
  br i1 %eq, label %then, label %else

then:                                             ; preds = %entry
  br label %ifcont

else:                                             ; preds = %entry
  %sub = sub i32 %0, 1   // n - 1
  %2 = call i32 @factorial(i32 %sub) // factorial(n-1)
  %mult = mul i32 %0, %2  // n * factorial(n-1)
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  %iftmp = phi i32 [ 1, %then ], [ %mult, %else ]
  ret i32 %iftmp
}
```

Some Optimizations

- Technically only improvements
- Arithmetic simplification:

```
a := x * 8   ⇒   a := x ≪ 3
b := y * 0   ⇒   b := 0
```

- Constant folding:

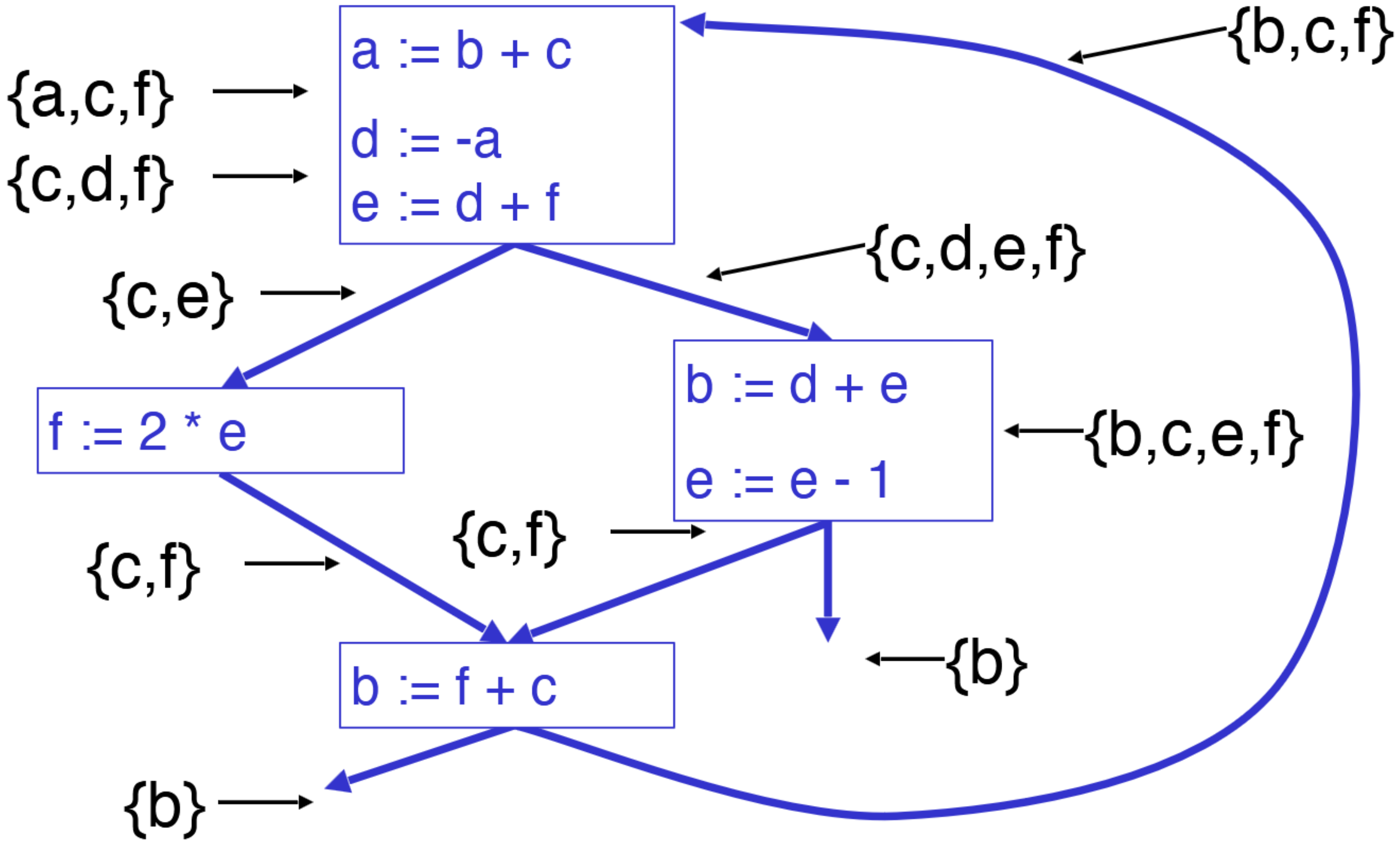
```
c := 2 + 2   ⇒   c := 4
```

- Ảvy̧lẰ ỷvy̧ẻɔẻävử8

```
x := 4           x := 4            x := 4
y := x       ⇒   y := 4       ⇒    y := 4
z := y + 2       z := x + 2        z := 6
```

Register Allocation

- What if we used 100% of the registers?
- Determine which variables conflict with each other
- Assuming we have four registers: Can we allocate them?



Register Interference Graph

- Construct the Register Interference Graph
  - Each variable is a node
  - Iff two variables conflict, they are connected
- Now it‘s a graph colouring problem!
  - Largest clique: {b, c, e, f} / {c, d, e, f}
- What if we only have three registers?
  - Put some variables in memory („spilling“)
  - Or recompute
  - Performance depends on which ones are spilt