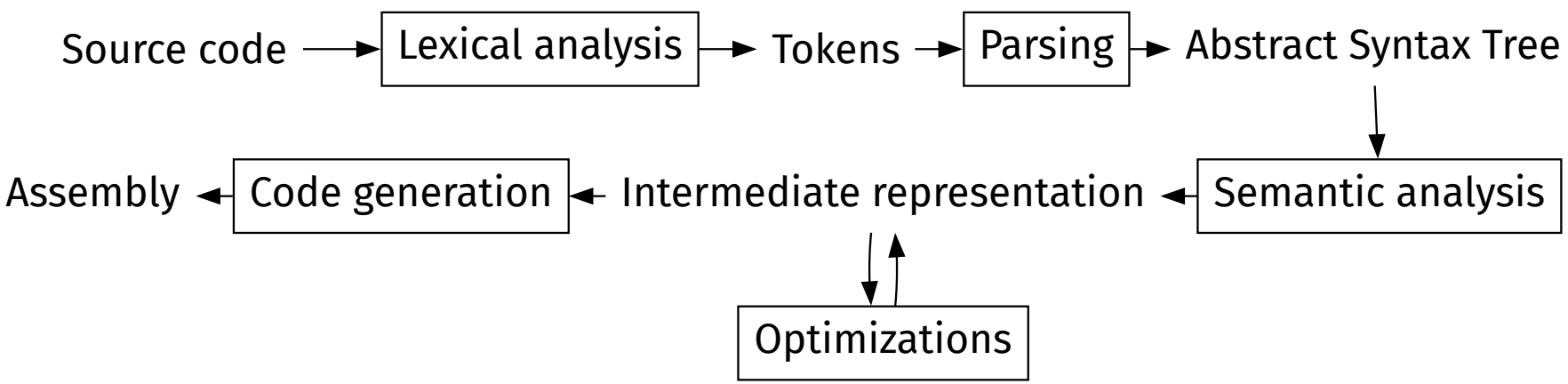
Hur har det gått?


Source code
Lexical analysis
Tokens
Parsing
Abstract Syntax Tree
Assembly
Code generation
Intermediate representation
Semantic analysis
Optimizations

Semantic analysis

(Vi skippar type checking)

Semantic analysis: Symbol tables

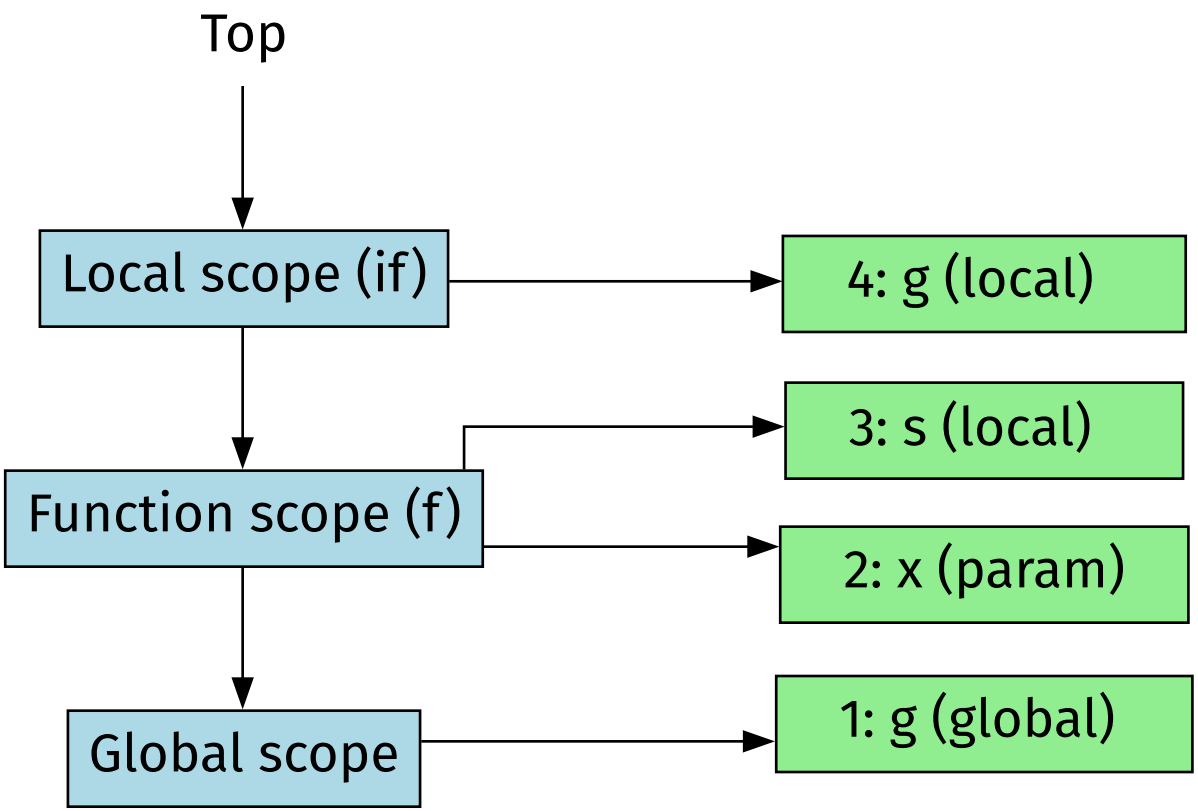

```
int g = 0;

void f(int x) {
  int s = x + 1;

  if (s < 0) {
    int g = 4;
    s += g;
  }

  g++;
}
```

- Innehåller information om symboler
- Variabler, parametrar och funktioner

Namn	Typ	…
g	int	…
f	void (int)	…
x	int	…
s	int	…
g	int	…

Symbol tables


```
int g = 0;

void f(int x) {
  int s = x + 1;

  if (s < 0) {
    int g = 4;
    s += g;
  }

  g++;
}
```



Intermediate representation

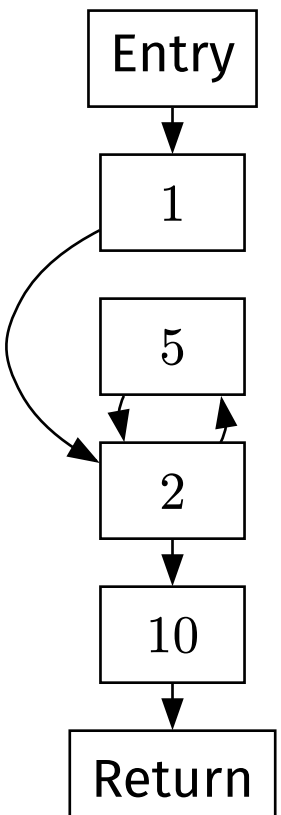
- En representation av programmet som lämpar sig för en viss uppgift.
- Kan vara ett slags pseudo-assembly, som lätt kan översättas till riktig assembly (t.ex. LLVM IR).

Intermediate representation: LLVM IR


```c
#include <stdio.h>

void count_to_ten(void)
{
  int i = 1;

  while (i <= 10)
  {
    printf("%d\n", i);
    i++;
  }

  printf("Done!\n");
}
```

```
; Function Attrs: noinline nounwind optnone sspstrong uwtable
define void @count_to_ten() #0 {
  %1 = alloca i32, align 4
  store i32 1, ptr %1, align 4
  br label %2

2:                                                ; preds = %5, %0
  %3 = load i32, ptr %1, align 4
  %4 = icmp sle i32 %3, 10
  br i1 %4, label %5, label %10

5:                                                ; preds = %2
  %6 = load i32, ptr %1, align 4
  %7 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %6)
  %8 = load i32, ptr %1, align 4
  %9 = add i32 %8, 1
  store i32 %9, ptr %1, align 4
  br label %2, !llvm.loop !5

10:                                               ; preds = %2
  %11 = call i32 (ptr, ...) @printf(ptr noundef @.str.1)
  ret void
}
```

Intermediate representation: LLVM IR

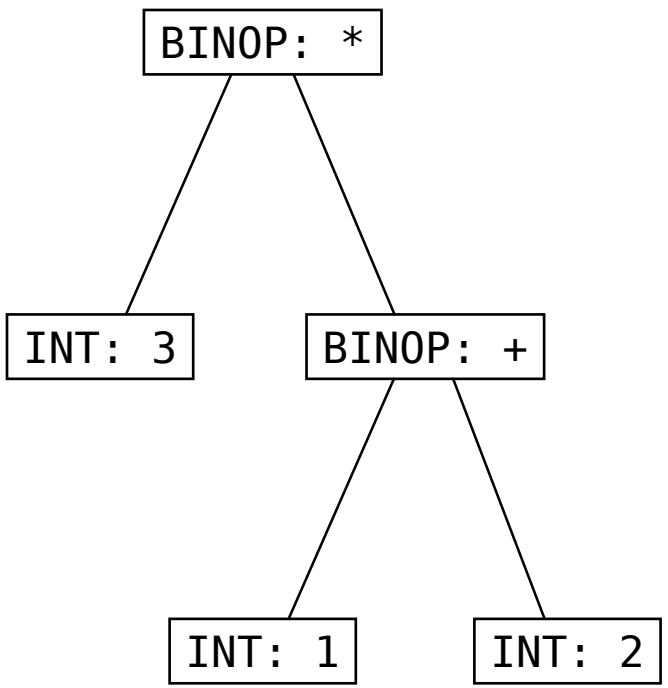



```
; Function Attrs: noinline nounwind optnone sspstrong uwtable
define void @count_to_ten() #0 {
  %1 = alloca i32, align 4
  store i32 1, ptr %1, align 4
  br label %2

2:                                                ; preds = %5, %0
  %3 = load i32, ptr %1, align 4
  %4 = icmp sle i32 %3, 10
  br i1 %4, label %5, label %10

5:                                                ; preds = %2
  %6 = load i32, ptr %1, align 4
  %7 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %6)
  %8 = load i32, ptr %1, align 4
  %9 = add i32 %8, 1
  store i32 %9, ptr %1, align 4
  br label %2, !llvm.loop !5

10:                                               ; preds = %2
  %11 = call i32 (ptr, ...) @printf(ptr noundef @.str.1)
  ret void
}
```

Kodgenerering

Exempel: QBE

Kodgenerering, exempel: QBE + aritmetik



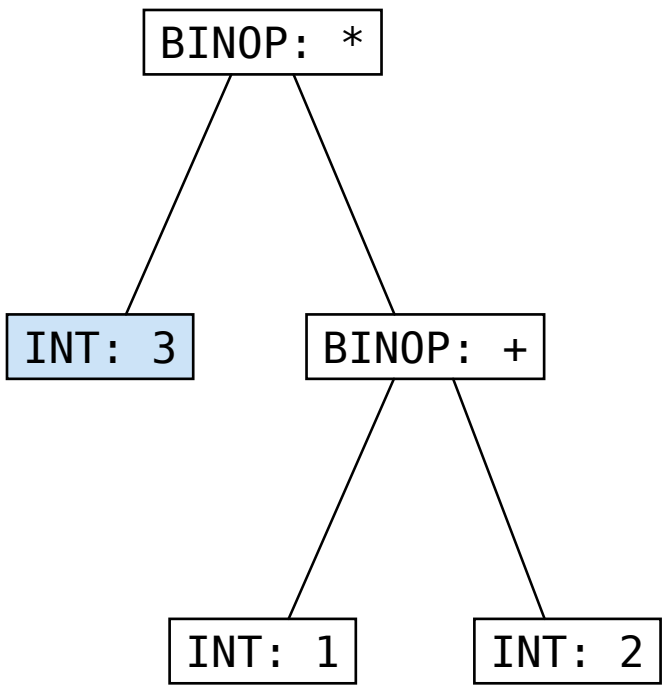
Vi kompilerar ett aritmetiskt uttryck till QBE IL:

$$3 * (2 + 1)$$

Vi kan allokera ett obegränsat antal variabler.

Kodgenerering, exempel: QBE + aritmetik


```
%x1 =w copy 3
```

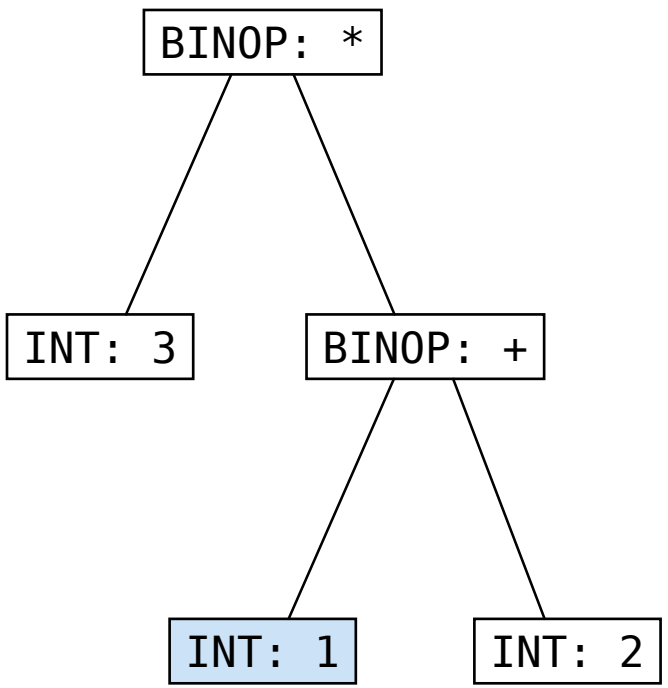


Vi kompilerar ett aritmetiskt uttryck till QBE IL:

$$3 * (2 + 1)$$

Vi kan allokera ett obegränsat antal variabler.

Kodgenerering, exempel: QBE + aritmetik



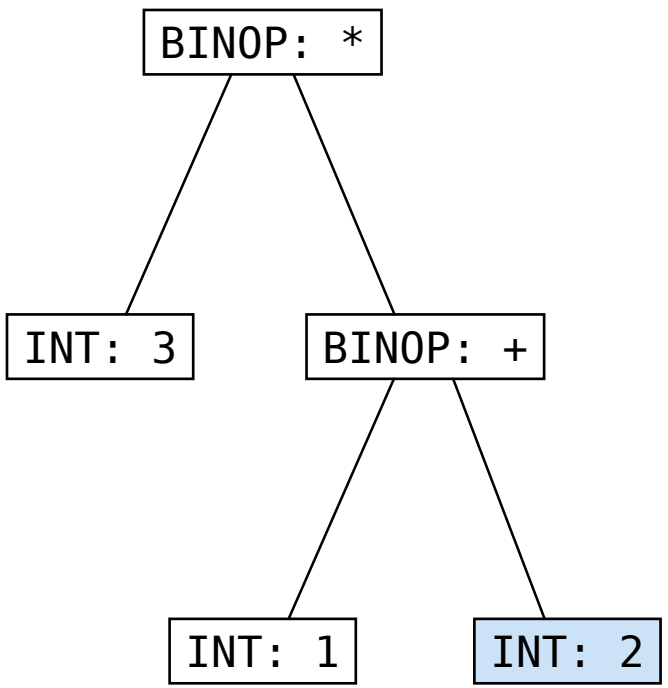

```
%x1 =w copy 3
%x2 =w copy 1
```

Vi kompilerar ett aritmetiskt uttryck till QBE IL:

$$3 * (2 + 1)$$

Vi kan allokera ett obegränsat antal variabler.

Kodgenerering, exempel: QBE + aritmetik



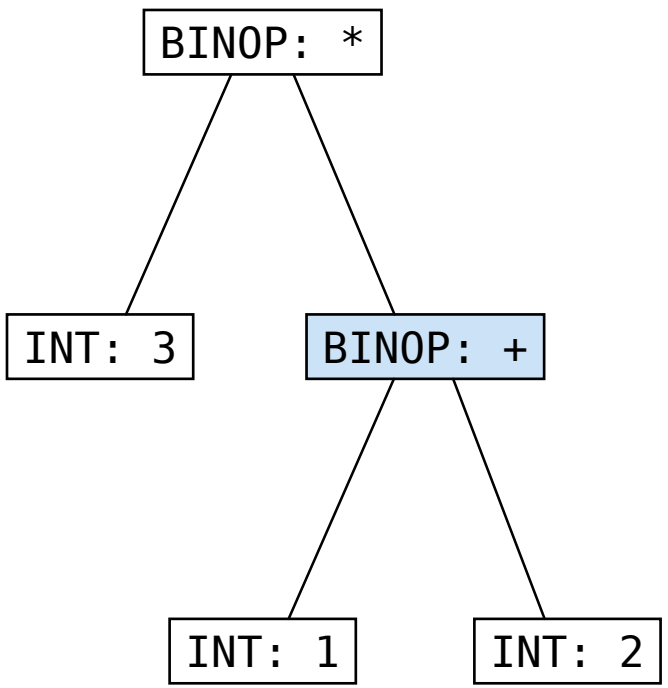

```
%x1 =w copy 3
%x2 =w copy 1
%x3 =w copy 2
```

Vi kompilerar ett aritmetiskt uttryck till QBE IL:

$$3 * (2 + 1)$$

Vi kan allokera ett obegränsat antal variabler.

Kodgenerering, exempel: QBE + aritmetik



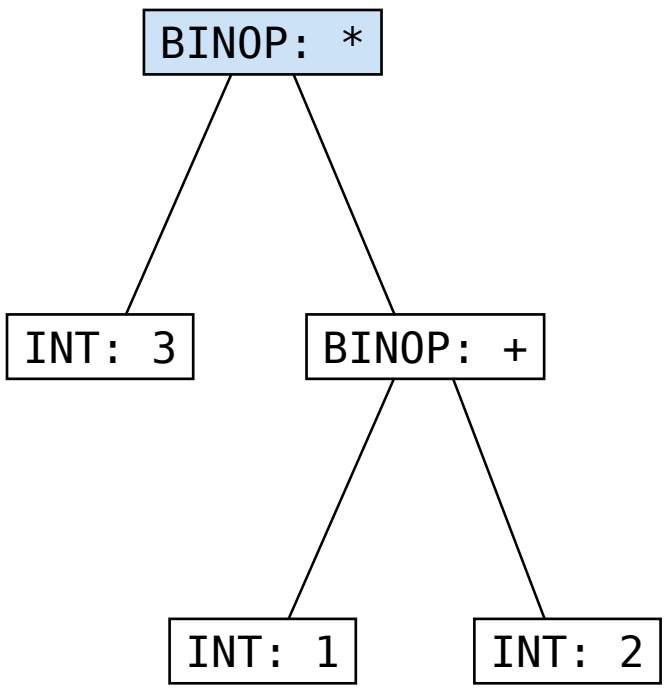

```
%x1 =w copy 3
%x2 =w copy 1
%x3 =w copy 2
%x4 =w add %x2, %x3
```

Vi kompilerar ett aritmetiskt uttryck till QBE IL:

$$3 * (2 + 1)$$

Vi kan allokera ett obegränsat antal variabler.

Kodgenerering, exempel: QBE + aritmetik




```
%x1 =w copy 3
%x2 =w copy 1
%x3 =w copy 2
%x4 =w add %x2, %x3
%x5 =w mul %x1, %x4
```

Vi kompilerar ett aritmetiskt uttryck till QBE IL:

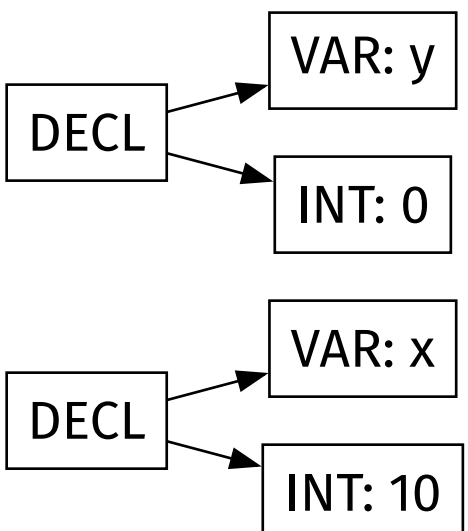
$$3 * (2 + 1)$$

Vi kan allokera ett obegränsat antal variabler.

Kodgenerering, exempel: QBE + loopar


```
int x = 10;
int y = 0;

while (x)

{
  y += x;
  x -= 1;
}

return y;
```

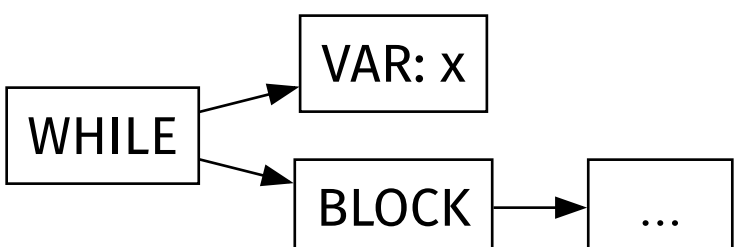
Kodgenerering, exempel: QBE + loopar




```
int x = 10;
int y = 0;

while (x)

{
  y += x;
  x -= 1;
}

return y;
```

```
@start
  %x =w copy 10
  %y =w copy 0
```

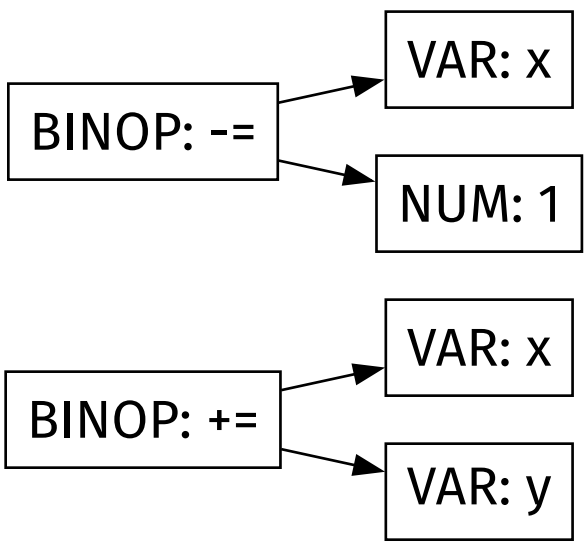
Kodgenerering, exempel: QBE + loopar




```
int x = 10;
int y = 0;

while (x)

{
  y += x;
  x -= 1;
}

return y;
```

```
@start
  %x =w copy 10
  %y =w copy 0

@check
  jnz %x, @loop, @done
```

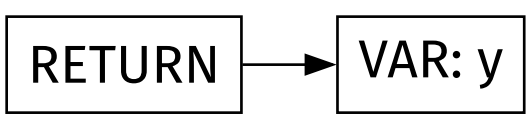
Kodgenerering, exempel: QBE + loopar




```
int x = 10;
int y = 0;

while (x)

{
  y += x;
  x -= 1;
}

return y;
```

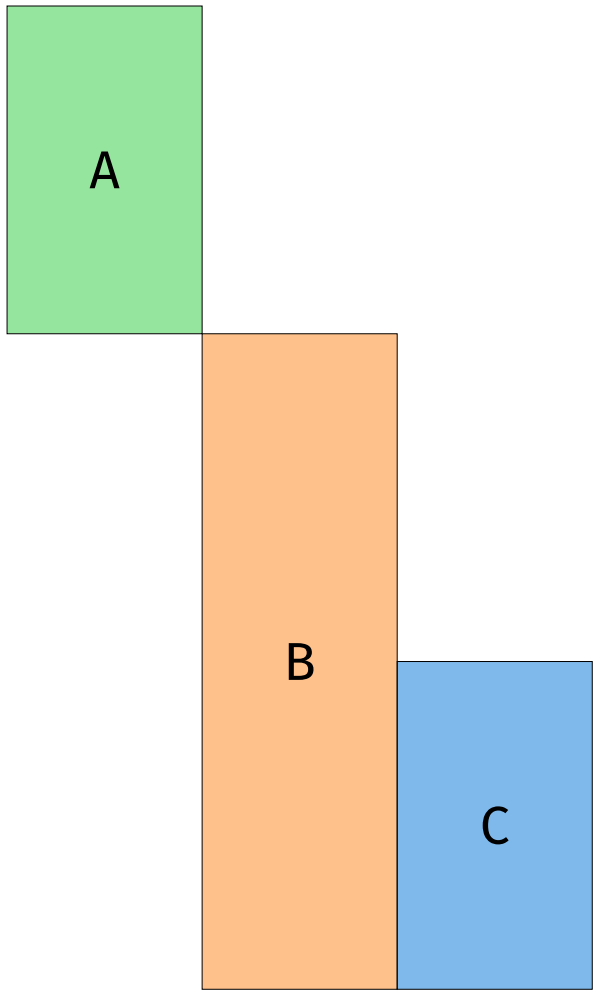
```
@start
  %x =w copy 10
  %y =w copy 0

@check
  jnz %x, @loop, @done

@loop
  %y =w add %y, %x
  %x =w sub %x, 1
  %jmp @check
```

Kodgenerering, exempel: QBE + loopar



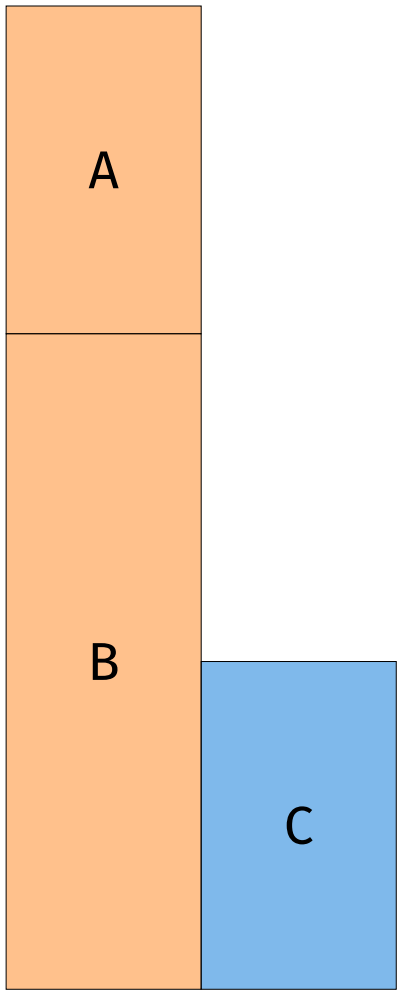

```
int x = 10;
int y = 0;

while (x)

{
  y += x;
  x -= 1;
}

return y;
```

```
@start
  %x =w copy 10
  %y =w copy 0

@check
  jnz %x, @loop, @done

@loop
  %y =w add %y, %x
  %x =w sub %x, 1
  %jmp @check

@done
  ret %y
```



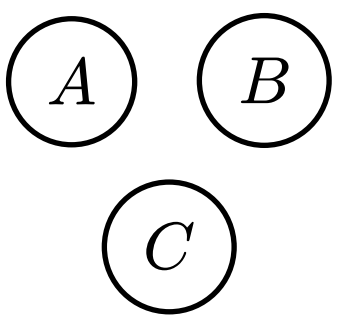
Kolla på livstiden för variabler

```
%A =w copy 1
%B =w copy 2
%C =w copy 3
%C =w add %C, %B
```

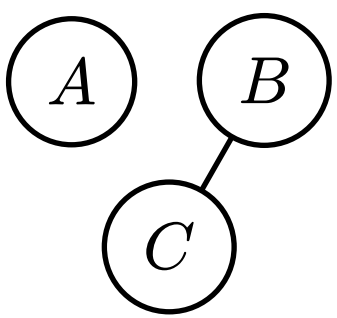


De som inte överlappar kan dela register

```
%A =w copy 1
%B =w copy 2
%C =w copy 3
%C =w add %C, %B
```

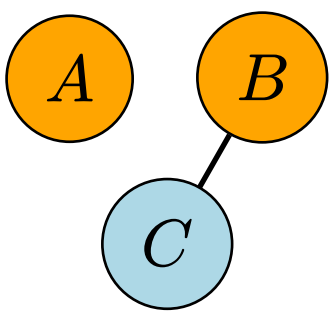


Kan lösas med graffärgning



Sätt en kant mellan variabler vars livstid överlappar

Registerallokering



= %rax

= %rbx

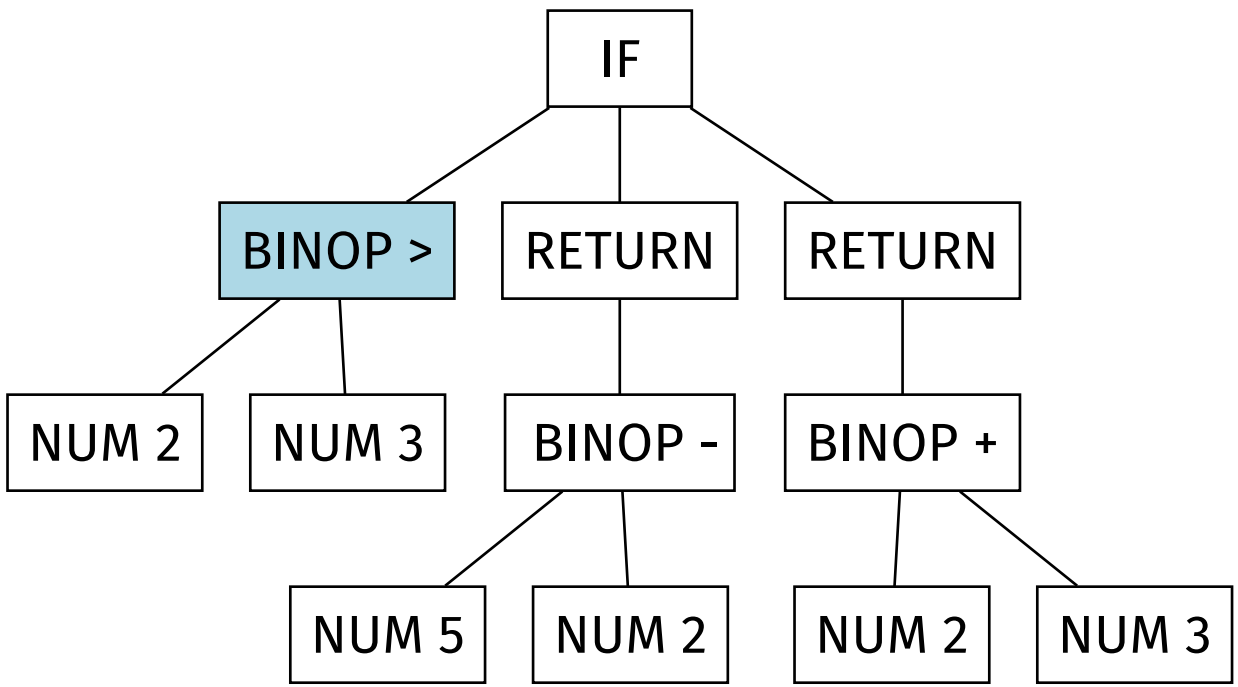
Funkar inte alltid. Man kan behöva ”spilla” (flytta en variabel till stacken)

Optimering

Ett (litet) exempel: Constant folding
Kan göras direkt i syntaxträdet

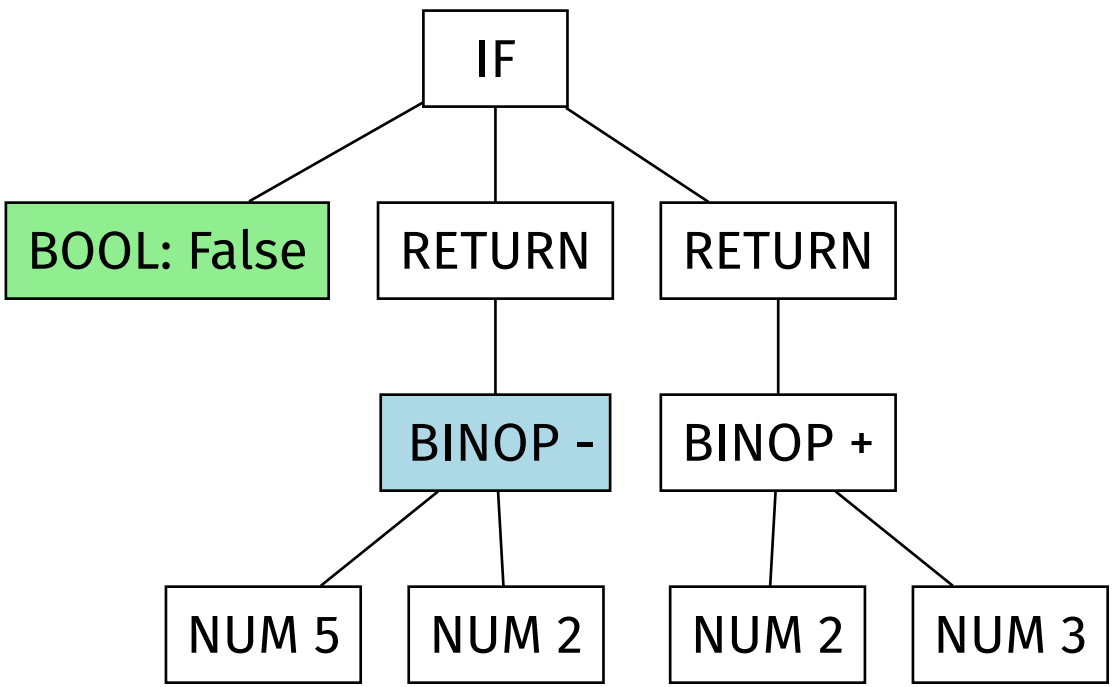
Optimering: Constant folding


```
if 2 > 3:
  return 5 - 2
else:
  return 2 + 3
```



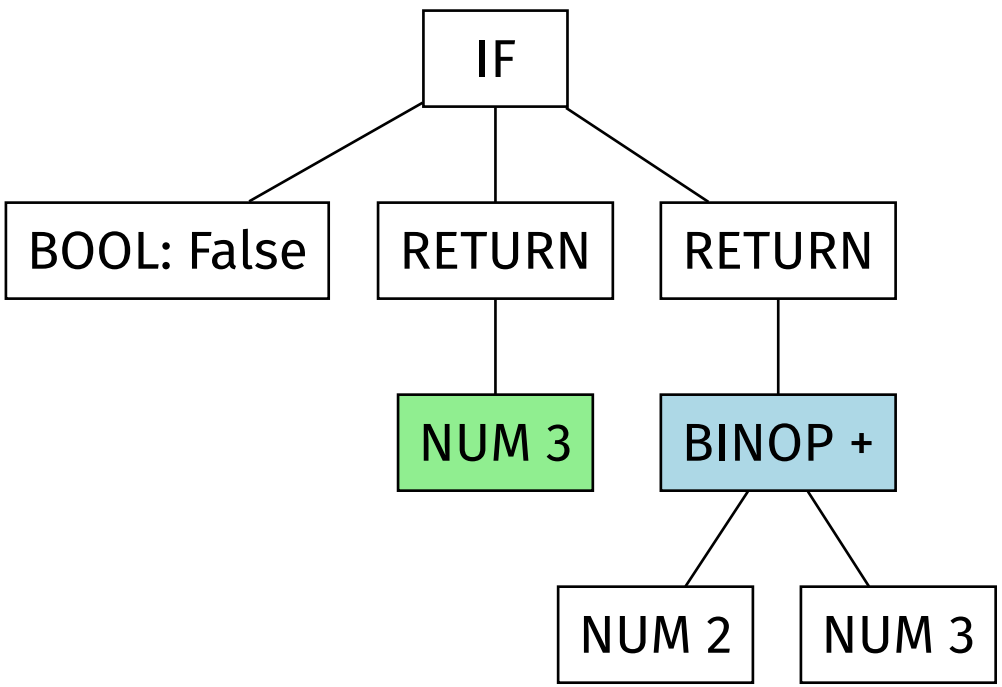
Optimering: Constant folding


```
if False:
   return 5 - 2
else:
   return 2 + 3
```



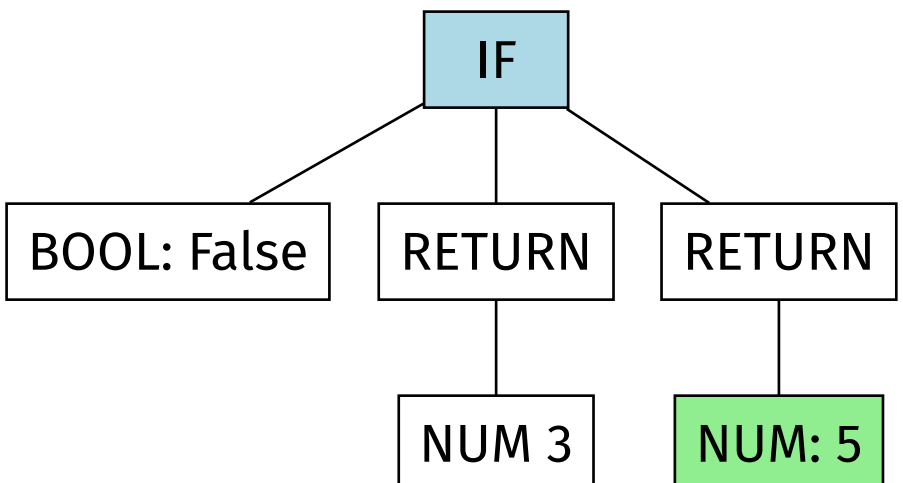
Optimering: Constant folding


```
if False:
  return 3
else:
  return 2 + 3
```



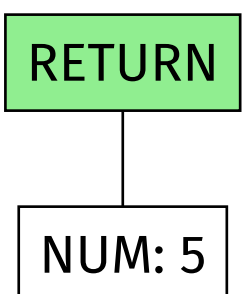
Optimering: Constant folding


```
if False:
   return 3
else:
   return 5
```



Optimering: Constant folding


```
return 5
```



Optimering: Constant folding


```
function constfold_if_statement(node)
    result = node

    node.condition = constfold(node.condition)
    node.then_branch = constfold(node.then_branch)
    node.else_branch = constfold(node.else_branch)

    if (is_constant_boolean(node.condition)) then
      if (node.condition.boolean_value) then
        result = node.then_branch
      else
        result = node.else_branch
      end
    end

    return result
end
```

Static Single Assignment form (SSA)

Varje tilldelning betraktas som en unik variabel m.h.a subscripts.

Användbart för exempelvis:

- Dead code elimination
- Constant folding
- Value numbering
- Register allocation

```
x = 1
y = 2
if condition then
  z = 8 + x
else
  z = 5 + y
end
y = z
```

→

```
x₁ = 1
y₁ = 2
if condition then
  z₁ = 8 + x₁
else
  z₂ = 5 + y₁
end
y₂ = z?
```

SSA: Fi-funktionen

Fi-funktionen (Φ) används för att ”välja” rätt värde av en variabel beroende på var vi kommer från.

$\Phi(x, y)$ betyder helt enkelt ”välj x eller y”

```
x1 = 1
y1 = 4
if condition then
  z1 = 8 + x1
else
  z2 = 5 + y1
end
z3 = Φ(z1, z2)
y2 = z3
```

Användbara algebraiska egenskaper:

$$\Phi(x) = x$$
$$\Phi(x, x, y) = \Phi(x, y)$$
$$\Phi(x, \Phi(y, z)) = \Phi(x, y, z)$$

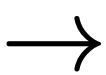
”Magisk” funktion som bara används för optimering, vi behöver inte kunna implementera den i praktiken. Försvinner senare vid översättning ut ur SSA-form.

Optimering: Constant Propagation

Flytta fram konstanter till uttrycken där de används.

```
x₁ = 1
y₁ = 4
if condition then
  z₁ = 8 + x₁
else
  z₂ = 5 + y₁
end
z₃ = Φ(z₁, z₂)
y₂ = z₃
```

```
x₁ = 1
y₁ = 4
if condition then
  z₁ = 8 + 1
else
  z₂ = 5 + 4
end
z₃ = Φ(z₁, z₂)
y₂ = z₃
```



```
x₁ = 1
y₁ = 4
if condition then
  z₁ = 8 + 1
else
  z₂ = 5 + 4
end
z₃ = Φ(8 + 1, 5 + 4)
y₂ = z₃
```

Optimering: Constant Folding

Beräkna konstanta uttryck.

```
x₁ = 1
y₁ = 4
if condition then
  z₁ = 8 + 1
else
  z₂ = 5 + 4
end
z₃ = Φ(8 + 1, 5 + 4)
y₂ = z₃
```

→

```
x₁ = 1
y₁ = 4
if condition then
  z₁ = 9
else
  z₂ = 9
end
z₃ = Φ(9, 9)
y₂ = z₃
```

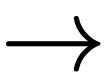
→

```
x₁ = 1
y₁ = 4
if condition then
  z₁ = 9
else
  z₂ = 9
end
z₃ = 9
y₂ = z₃
```

Optimering: Dead Code Elimination

Ta bort onödig kod eller kod som aldrig körs.

```
x1 = 1
y1 = 4
if condition then
  z1 = 9
else
  z2 = 9
end
z3 = 9
y2 = 9
```



```
x1 = 1
y1 = 4
z3 = 9
y2 = 9
```

Veckans läxa

- Skriv en backend till er kompilator
- För att genererar assembly kan ni använda t.ex. QBE (enklast) eller LLVM.
- Det går även bra att kompilera till ett annat språk, men...
- **...språkets syntax måste vara väsentligt annorlunda!**