# SBC

Slige Backup Compiler

1 uge

~3000 linjer kode

Human-readable

```
fn main() -> int {
    let a = 5;
    print("Hello\ world");
    return 0
}
```

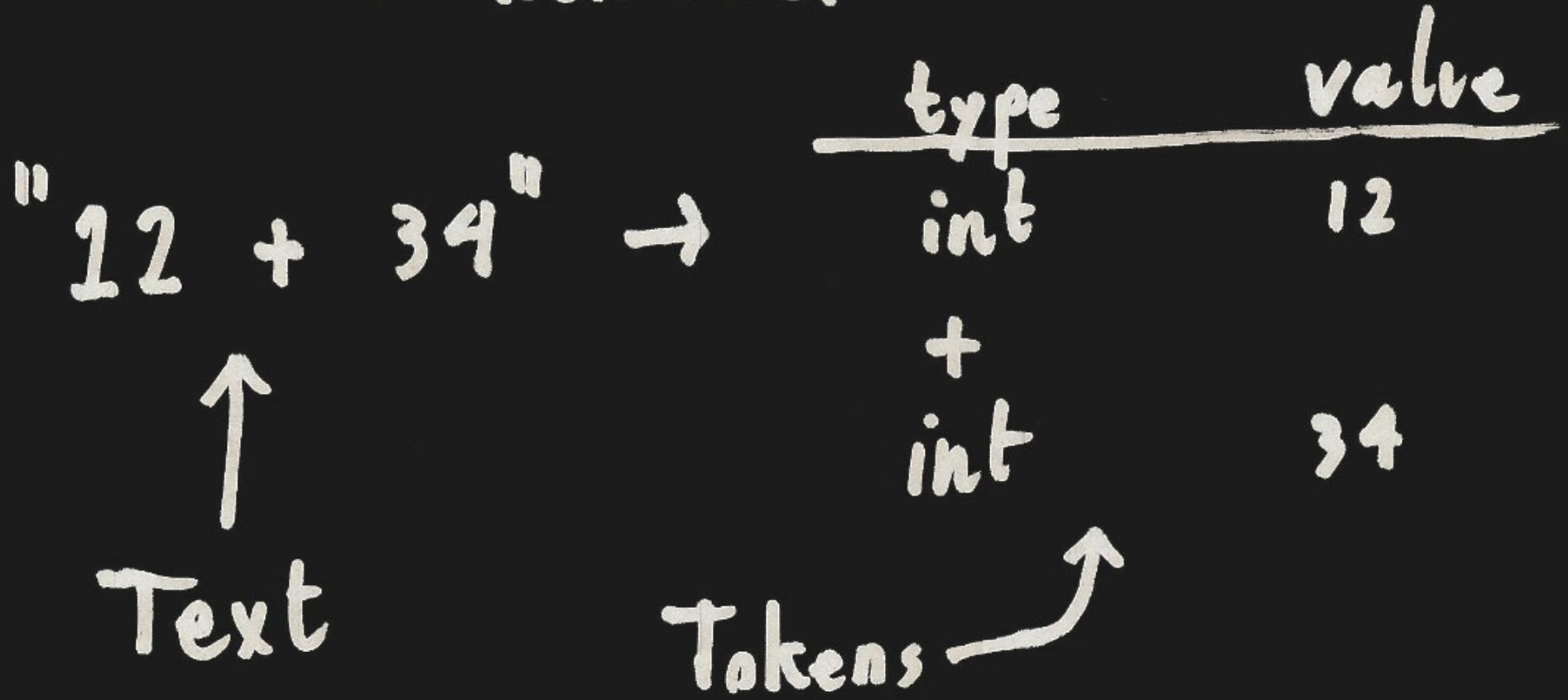
Machine code

```
main:
    sub rsp, 8
    mov [rsp], 5
    call print
    mov rax, 0
    ret
    .
    .
    .
```

```
2 fn main() -> int {
3
4     let i = 3 + 5;
5
6     println("Hello\ world");
7
8     return 0;
9 }
```

```
 8 sbc__main:
 9     push rbp
10     mov rbp, rsp
11     sub rsp, 16
12     jmp .L0
13 .L0:
14     mov rax, 3
15     mov rdi, 5
16     add rax, rdi
17     mov QWORD [rbp-16], rax
18     mov rax, sbc__string_0
19     push rax
20     call sbc__println
21     push rax
22     pop rax
23     pop rdi
24     mov QWORD [rbp-8], 0
25     jmp .exit
26 .exit:
27     mov rax, QWORD [rbp-8]
28     mov rsp, rbp
29     pop rbp
30     ret
```

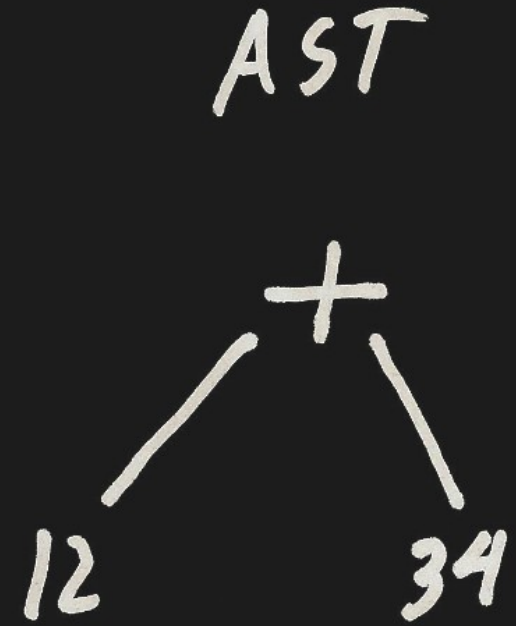
# Tokenizer



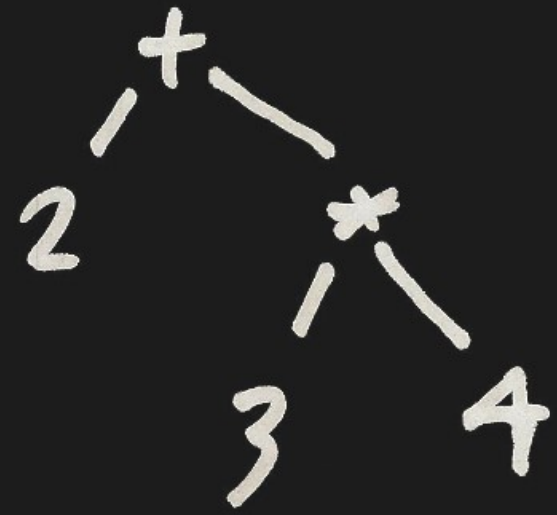
# Parser

Tokens

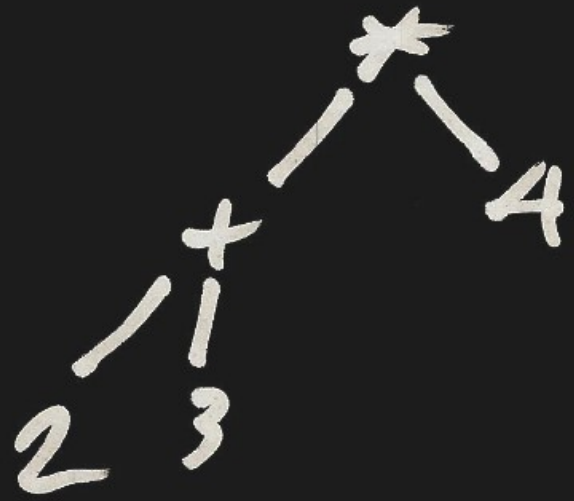
```
[
  int(12),
  +,
  int(34),
]
```



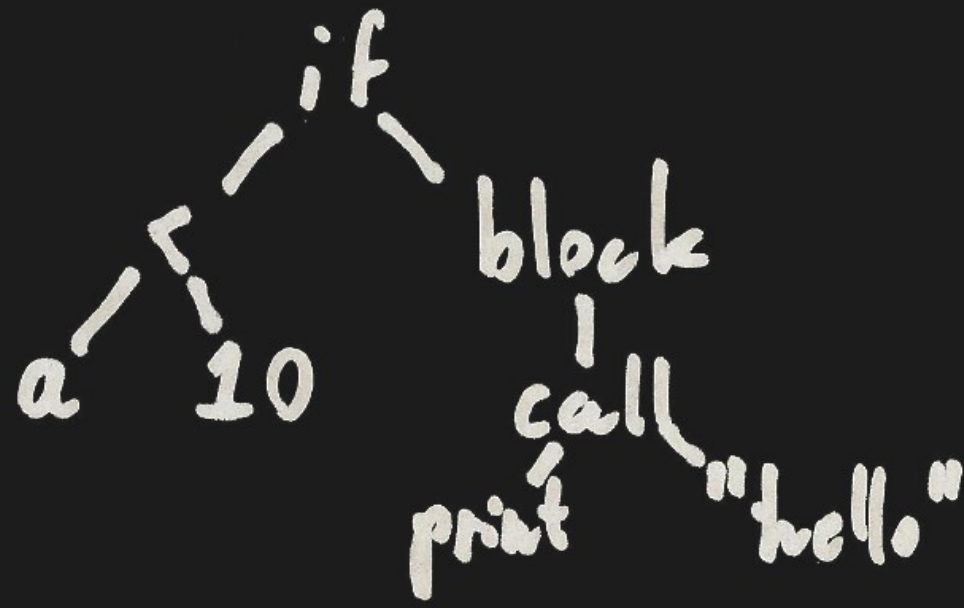
2 + 3 * 4




(2+3) * 4



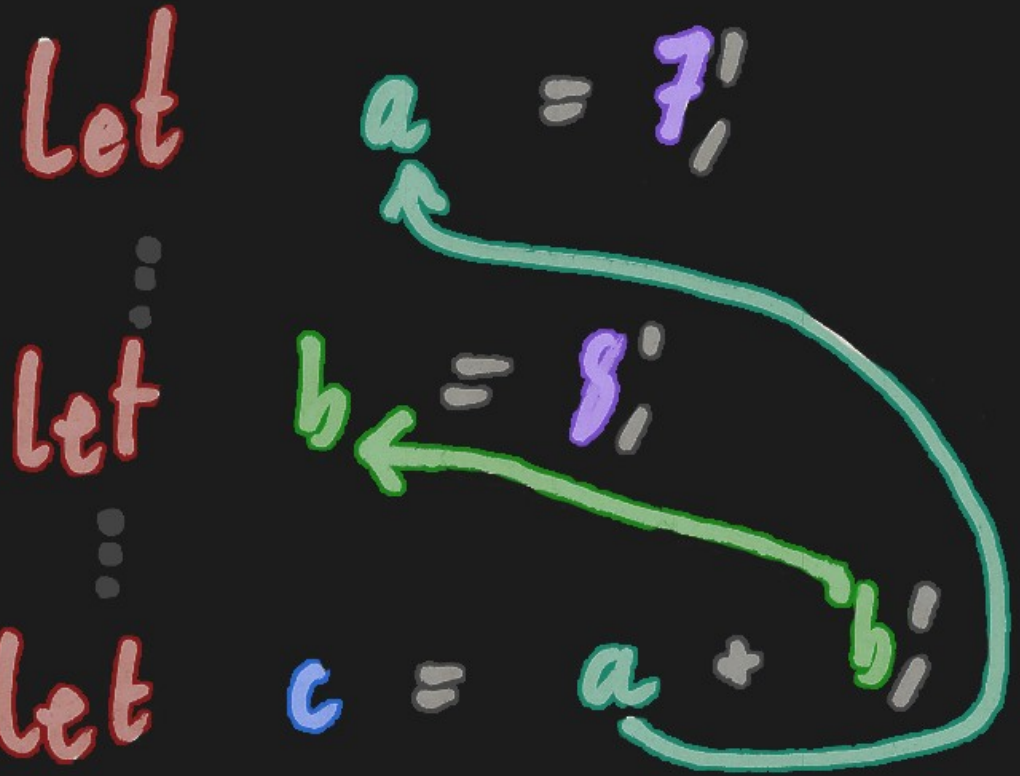
```
if a < 10 {
    print("hello.");
}
```



```
if a < 10 {
    println("Hello");
}
```

```
IfStmt {
    condition: LessThan {
        left: IdentExpr("a"),
        right: IntExpr(10),
    },
    body: Block([
        CallExpr {
            callee: IdentExpr("println"),
            arguments: [
                String("Hello"),
            ],
        }
    ]),
}
```

# Symbol Resolution



# Type Checking

Checking

```
let v: int = 5;
```

type of 5 is 'int'

'int' assignable to 'int'? ✓

```
let k = 7;
```

k's type is 7; type = int

Inference

# Lowering (part 1)

AST

→

MIR

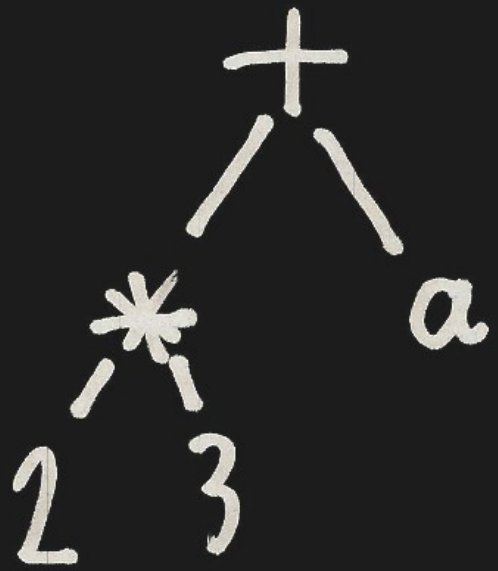
Mid-level

Intermediary

Representation

2*3+a

AST




MIR

```
push 2
push 3
mul
load %a
add
```

```
 3 fn main() -> int {
 4     let i = 0;
 5
 6     while i < 10 {
 7         println("Hello");
 8
 9         i = i + 1;
10     }
11     return 0;
12 }
```

```
main:
    %0 :: int
    %1 :: int
    .b0:
        push (int) 0
        store %1
        goto .b2
    .b2:
        load %1
        push (int) 10
        lt
        goto .b3, .b4
    .b3:
        push (*str) "Hello"
        push (fn(*str) -> int) println
        call 1
        pop
        load %1
        push (int) 1
        add
        store %1
        goto .b2
    .b4:
        push (int) 0
        store %0
        return
```

# Lowering (part 2)
## (elaboration)

MIR ⟶ LIR

**Mid**-level Intermediary Representation

**Low**-level Intermediary Representation

# 2 * 3 + a

## MIR

```
push 2
push 3
mul
load %.a
add
```

Stack based

Register based

## LIR

```
mov %.1, 2
push %.1
kill %.1
mov %.2, 3
push %.2
kill %.2
pop %.3
pop %.4
add %.3, %.4
⋮
```

```
main:
    %0 :: int
    %1 :: int
    .b0:
        push (int) 0
        store %1
        goto .b2
    .b2:
        load %1
        push (int) 10
        lt
        goto .b3, .b4
    .b3:
        push (*str) "Hello"
        push (fn(*str) -> int) println
        call 1
        pop
        load %1
        push (int) 1
        add
        store %1
        goto .b2
    .b4:
        push (int) 0
        store %0
        return
```

```
sbc__main:
    alloc_local %0, 8
    alloc_local %1, 8
.L0:
    store_val [%1], 0
    jmp .L1
.L1:
    load %4, [%1]
    mov_int %5, 10
    lt %4, %5
    kill %5
    jnz_reg %4, .L2
    kill %4
    jmp .L3
.L2:
    mov_string %9, string+0
    push %9
    kill %9
    call_fn sbc__println, 1
    pop %12
    pop %13
    kill %13
    kill %12
    load %15, [%1]
    mov_int %16, 1
    add %15, %16
    kill %16
    store_reg [%1], %15
    kill %15
    jmp .L1
.L3:
    store_val [%0], 0
    ret
```

# Lowering (part 3)

## (code generation)

LIR

Low-level

Intermediary

Representation

→

X86_64

assembly

ISA

Instruction set

Architecture

```
sbc__main:
    alloc_local %0, 8
    alloc_local %1, 8
.L0:
    mov_int %2, 0
    push %2
    kill %2
    pop %3
    store_reg [%1], %3
    kill %3
    jmp .L1
.L1:
    load %4, [%1]
    push %4
    kill %4
    mov_int %5, 10
    push %5
    kill %5
    pop %6
    pop %7
    lt %7, %6
    push %7
    kill %6
    kill %7
    pop %8
    jnz_reg %8, .L2
    kill %8
    jmp .L3
```

```
 8 sbc__main:
 9     push rbp
10     mov rbp, rsp
11     sub rsp, 16
12     jmp .L0
13 .L0:
14     mov QWORD [rbp-16], 0
15     jmp .L1
16 .L1:
17     mov rax, QWORD [rbp-16]
18     mov rdi, 10
19     cmp rax, rdi
20     setl al
21     cmp rax, 0
22     jne .L2
23     jmp .L3
```

# Register Allocation

LIR

```
mov %1, 3
mov %2, 4
add %1, %2
mov %3, 5
add %1, %3
```

```
mov rax, 3
mov rdi, 4
add rax, rdi
mov rdi, 5
add rax, rdi
```

rax: %1

rdi: %2 %3

```
mov_int %5, 3
mov_int %6, 4
add %5, %6
kill %6
mov_int %9, 5
add %5, %9
kill %9
store_reg [%2], %5
kill %5
```

```
11      mov rax, 3
12      mov rdi, 4
13      add rax, rdi
14      mov rdi, 5
15      add rax, rdi
16      mov QWORD [rbp-24], rax
```

# Optimization (part 1)

MIR →  → MIR

- Dead code elimination
- Jump elision

```
main:
    %0 :: int
    %1 :: int
    .b0:
        push (int) 0
        store %1
        goto .b2
    .b1:
        return
    .b2:
        load %1
        push (int) 10
        lt
        goto .b3, .b4
    .b3:
        push (*str) "Hello"
        push (fn(*str) -> int) println
        call 1
        pop
        load %1
        push (int) 1
        add
        store %1
        goto .b2
    .b4:
        push (int) 0
        store %0
        goto .b1
    .b5:
        goto .b1
```

```
main:
    %0 :: int
    %1 :: int
    .b0:
        push (int) 0
        store %1
        goto .b2
    .b2:
        load %1
        push (int) 10
        lt
        goto .b3, .b4
    .b3:
        push (*str) "Hello"
        push (fn(*str) -> int) println
        call 1
        pop
        load %1
        push (int) 1
        add
        store %1
        goto .b2
    .b4:
        push (int) 0
        store %0
        return
```

# Optimization (part 2)

LIR →  → LIR

- Stack spill reduction
- Instruction reduction

```
sbc__main:
    alloc_local %0, 8
    alloc_local %1, 8
.L0:
    mov_int %2, 0
    push %2
    kill %2
    pop %3
    store_reg [%1], %3
    kill %3
    jmp .L1
.L1:
    load %4, [%1]
    push %4
    kill %4
    mov_int %5, 10
    push %5
    kill %5
    pop %6
    pop %7
    lt %7, %6
    push %7
    kill %6
    kill %7
    pop %8
    jnz_reg %8, .L2
    kill %8
    jmp .L3
.L2:
    mov string %9, string+0
```

```
sbc__main:
    alloc_local %0, 8
    alloc_local %1, 8
.L0:
    store_val [%1], 0
    jmp .L1
.L1:
    load %4, [%1]
    mov_int %5, 10
    lt %4, %5
    kill %5
    jnz_reg %4, .L2
    kill %4
    jmp .L3
.L2:
    mov string %9, string+0
```

```
fn test(iterations: int) -> int {
    let acc = 0;
    let i = 0;
    while i < iterations {
        acc = acc + i + acc % 3;
        i = i + 1;
    }
    return acc;
}
```

```
int64_t test(int64_t iterations)
{
    int64_t acc = 0;
    int64_t i = 0;
    while (i < iterations) {
        acc += i + acc % 3;
        i += 1;
    }
    return acc;
}
```

```
sbc__test:
    push rbp
    mov rbp, rsp
    sub rsp, 24
    jmp .L0
.L0:
    mov QWORD [rbp-16], 0
    mov QWORD [rbp-24], 0
    jmp .L1
.L1:
    mov rax, QWORD [rbp-24]
    mov rdi, QWORD [rbp+16]
    cmp rax, rdi
    setl al
    cmp rax, 0
    jne .L2
    jmp .L3
.L2:
    mov rax, QWORD [rbp-16]
    mov rdi, QWORD [rbp-24]
    add rax, rdi
    push rax
    mov rax, QWORD [rbp-16]
    mov rdi, 3
    push rax
    push rdi
    call sbc__mod
    mov rax, rax
    pop rdi
    add rdi, rax
    mov QWORD [rbp-16], rdi
    mov rax, QWORD [rbp-24]
    mov rdi, 1
    add rax, rdi
    mov QWORD [rbp-24], rax
    jmp .L1
.L3:
    mov rax, QWORD [rbp-16]
    mov QWORD [rbp-8], rax
    jmp .exit
.exit:
    mov rax, QWORD [rbp-8]
    mov rsp, rbp
    pop rbp
    ret
```

```
test:
        push    rbp
        mov     rbp, rsp
        mov     qword ptr [rbp - 8], rdi
        mov     qword ptr [rbp - 16], 0
        mov     qword ptr [rbp - 24], 0
.LBB0_1:
        mov     rax, qword ptr [rbp - 24]
        cmp     rax, qword ptr [rbp - 8]
        jge     .LBB0_3
        mov     rax, qword ptr [rbp - 24]
        mov     qword ptr [rbp - 32], rax
        mov     rax, qword ptr [rbp - 16]
        mov     ecx, 3
        cqo
        idiv    rcx
        mov     rax, qword ptr [rbp - 32]
        add     rax, rdx
        add     rax, qword ptr [rbp - 16]
        mov     qword ptr [rbp - 16], rax
        mov     rax, qword ptr [rbp - 24]
        add     rax, 1
        mov     qword ptr [rbp - 24], rax
        jmp     .LBB0_1
.LBB0_3:
        mov     rax, qword ptr [rbp - 16]
        pop     rbp
        ret
```

```c
void route_post_carts_purchase(HttpCtx* ctx)
{
    // ...
    CartsPurchaseReq req;
    // ...
    ProductPriceVec prices;
    // ...
    int64_t total_dkk_cent
        = sbc_calculate_total_price((int64_t)item_amount, &req.items, &prices);

```

```rust
#[c_export("sbc_calculate_total_price")]
fn calculate_total_price(item_amount: int, items: *(), prices: *()) -> int {
```

```
#[c_export("sbc_calculate_total_price")]
fn calculate_total_price(item_amount: int, items: *(), prices: *()) -> int {
    let total_dkk_cent = 0;

    let i = 0;
    while i < item_amount {
        let price = prices_get_price(prices, i);
        let amount = items_get_amount(items, i);

        total_dkk_cent = total_dkk_cent + price * amount;

        i = i + 1;
    }

    return total_dkk_cent;
}
```

```
19 #[c_function("sbcs_prices_get_price")]
20 fn prices_get_price(prices: *(), i: int) -> int {}
21
22 #[c_function("sbcs_items_get_amount")]
23 fn items_get_amount(items: *(), i: int) -> int {}
```

```
113 int64_t sbcs_prices_get_price(const ProductPriceVec* prices, int64_t i)
114 {
115     return prices->data[i].price_dkk_cent;
116 }
117 int64_t sbcs_items_get_amount(const CartsItemVec* items, int64_t i)
118 {
119     return items->data[i].amount;
120 }
```