Sequence 4.4 – Basic blocks

P. de Oliveira Castro S. Tardieu

Code generation

- In general, code generation works one function at a time.
- For every function in a compilation unit, code is generated then optimized.
- Functions are further split into basic blocks.

A *basic block* is a block of code that:

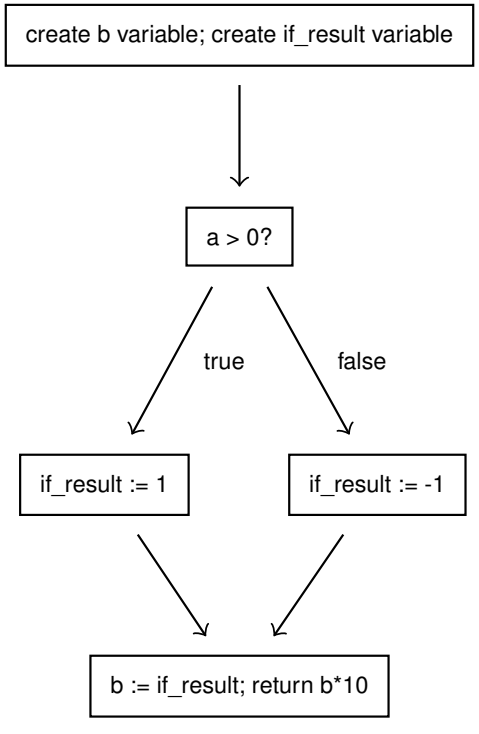
- runs sequentially;
- has only one entry point at the top;
- terminates with one of those three alternatives:
  - a branch to another block;
  - a return from the function;
  - a conditional branch to several blocks.

At the beginning of the function, a block (the *entry*) groups all local variable creations.

Example of `if/then/else`

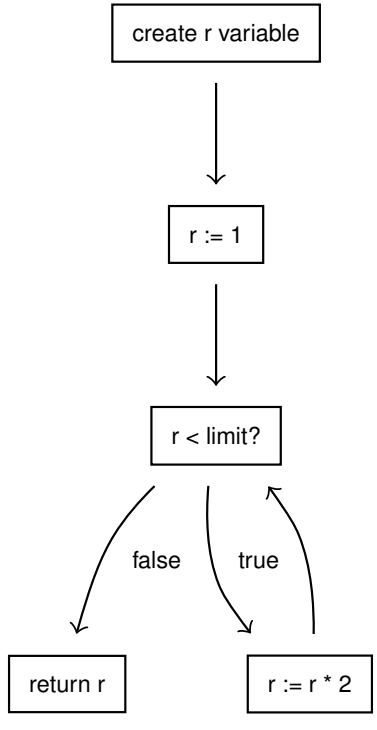
An `if_result` temporary variable is introduced by the compiler to hold the result of the `if/then/else` expression.

```
let function f(a: int): int =
  let var b := if a > 0 then 1 else -1 in b * 10 end
in ... end
```



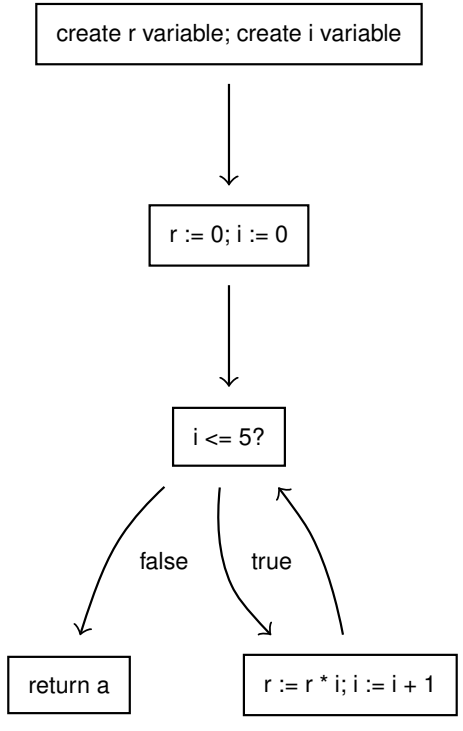
Example of while loop

```
// pow2 computes the smallest power of 2 >= limit
let function pow2(limit: int): int =
  let var r := 1 in while r < limit do r := r * 2; r end
in ... end
```



Example of for loop

```
let function fact(n: int): int =
  let var r := 1 in for i := 2 to n do r := r * i; r end
in ... end
```



Tools at our disposal

LLVM, which we use as a backend in our Tiger compiler, offers several tools to manipulate basic blocks:

- a function to create a new local variable (we will use this to create new variables in the entry block);
- a function to create a new basic block (with an optional label, useful for debugging);
- a function to set the insertion point of the generated instructions at the end of a given basic block;
- functions to generate branches to exit a basic block.

Control flow is lowered to branches and label

```
let var a := 0 in print_int(if a then 1 else 2) end
```

compiles to,

```
  %a = alloca i32              ; allocate variable a
  %if_result = alloca i32      ; allocate temporary

  store i32 0, i32* %a         ; var a := 0

  %0 = load i32, i32* %a
  %1 = icmp ne i32 %0, 0       ; if a (is a <> 0 ?)

  br i1 %1, label %if_then, label %if_else
```

Control flow (2/2)

```
let var a := 0 in print_int(if a then 1 else 2) end
```

compiles to,

```
[...]

if_then:
  store i32 1, i32* %if_result ; then, store if result 1
  br label %if_end

if_else:
  store i32 2, i32* %if_result ; else, store if result 2
  br label %if_end

if_end:
  %2 = load i32, i32* %if_result ; read if result
  call void @__print_int(i32 %2) ; print if result
```

What about loops ?

Question: How would you write the following program in LLVM IR?

```
let var a := 10 in while a do (a := a - 1; print_int(a)) end
```

Answer

```
  %a = alloca i32
  store i32 10, i32* %a       ; var a := 10
  br label %while_test        ; jump to %while_test

while_test:
  %0 = load i32, i32* %a      ; read a
  %1 = icmp ne i32 %0, 0      ; is *a zero ?
  br i1 %1, label %while_body, label %while_end

while_body:
  %2 = load i32, i32* %a          ; read a
  %3 = sub i32 %2, 1              ; *a - 1
  store i32 %3, i32* %a           ; write (*a - 1) to a
  %4 = load i32, i32* %a          ; read a
  call void @__print_int(i32 %4) ; print *a
  br label %while_test            ; loop back to test

while_end:
```

Conclusion

- Code is generated one function at a time.
- For every function, we generate basic blocks.
- Every basic block as a unique entry point, and a unique exit point (return from function, unconditional jump to another block, conditional jump towards several blocks).
- Local variables are declared using `alloca` in the first basic block (entry block).
- Local variables are accessed through `store` and `load` operations.
- The `mem2reg` optimization pass will remove all redundant `alloca`/`store`/`load` operations.