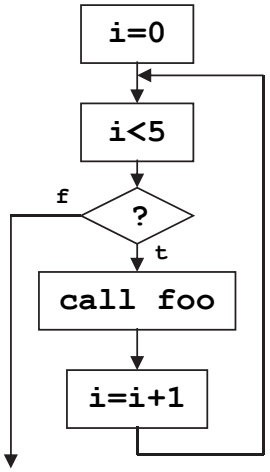
COMPILATION

- **Program –> machine code**
  - Phases
    - Converting into a sequence of abstract operations
    - Optimizations – constant & variable propagation, etc.
  - Converting into a sequence of operations
    - Statement (language construct) -> sequence of basic blocks
  - Basic block ~ expression

```
for (i=0;i<5;i++) { foo(); }
```




```
<for_cycle> ::= 'for' '(' (<expr>)* ';' (<expr>)* ';'
                          (<expr>)* ')' [<statement>] ';'
```

COMPILATION

- **Converting into a sequence of operations**
  - Expression –> sequence of operations
    - Polish notation, a.k.a. prefix notation (Jan Łukasiewicz)
      - Postfix notation can be used too
    - Priorities of operations
    - Primary expressions
      - Parentheses, variables, functions, …
    - Stack or (virtual) registers
  - Common sub-expressions
  - Subroutines (sub-programs)
    - Parameters and return address in the stack
  - Variables
    - .stack / .data / .heap

```
x = a + b + c ;
```

```
r1 <- + a b
r2 <- + r1 c
x <- r2
```

```
y = a + b * c ;
```

```
r1 <- * b c
r2 <- + a r1
y <- r2
```

```
z = (a + b) * c ;
```

```
r1 <- + a b
r2 <- * r1 c
z <- r2
```

COMPILATION - EXAMPLE

C program

```
#include <stdio.h>
int main (void) {
  printf ("Hello, world!\n");
  return 0;
}
```

Generating the assemby code

```
> gcc -S hello.c
```

Labels in the object file

```
> nm hello.o
00000000 T main
         U puts
>
```

Assembly code

```
    .file  "hello.c"
    .section          .rodata
.LC0:
    .string           "Hello, world!"
    .text
.globl main
    .type  main, @function
main:
    leal   4(%esp), %ecx
    andl   $-16, %esp
    pushl  -4(%ecx)
    pushl  %ebp
    movl   %esp, %ebp
    pushl  %ecx
    subl   $4, %esp
    movl   $.LC0, (%esp)
    call   puts
    movl   $0, %eax
    addl   $4, %esp
    popl   %ecx
    popl   %ebp
    leal   -4(%ecx), %esp
    ret
    .size  main, .-main
    .ident "GCC: (GNU) 4.1.2 20061115 (prerelease) (SUSE Linux)"
    .section          .note.GNU-stack,"",@progbits
```

LINKING

- Compiler's output is so called object file (.o)
  - Has a special table for labels (solved by linker)
- Compiler's type depends on the processor
  - Different OS-s may use the same compiler (e.g., gcc)

- **Linker** merges different object files, solves labels' locations in the memory and creates the program's memory image (executable file – .com/.exe/a.out/…)
- The memory map is defined by OS
  - Program (code) from address 0 (read-only)
  - Data in readable & writeable segment
- Static linking
  - All subroutines are in the memory image
- Dynamic linking
  - Standard (library) subroutines are loaded when needed