Taking apart your compiler!

Jacques-Henri Jourdan

February 16th
Inria Paris Junior Seminar

Who am I?

- Me = Jacques-Henri Jourdan = PhD student in team Gallium.
- Gallium:
  - Formal verification (... of programming tools)
  - Functional programming (... in OCaml)
  - Programming computers with multiple processors

Why a compiler?

Your processor understands assembly (this is the “readable” version):

```
        pushq   %rbp
        movq    %rdi, %rbp
        pushq   %rbx
        subq    $32, %rsp
        movq    16(%rdi), %rcx
        testq   %rcx, %rcx
        je      .L112
        movq    8(%rsi), %r12
        movq    (%rsi), %r10
        movq    %r12, %rdi
        subq    %r10, %rdi
        sarq    $2, %rdi
        movq    %rdi, %rsi
.L121:
        movq    40(%rcx), %rbx
        movq    32(%rcx), %r9
        movq    %rbx, %r11
        subq    %r9, %r11
        sarq    $2, %r11
```

You don’t want to write assembly.

Why a compiler?

- You write your code in high-level languages (C, C++, OCaml, Java, Python, Mathlab, Javascript)...
- You need a tool that understands this language.
  - You may use an **interpreter**:
    - **Runs** your program directly one step after the other
    - Very slow
    - Python, Mathlab, (Javascript), ...
  - You may use a **compiler**:
    - **Translates** your program into assembly
    - C, C++, OCaml, Java, (Javascript), ...

Assembly: the language of the processor

How does a processor works?

- It has a few **registers**
  - Small pieces of memory (few bytes) available directly to the computing units
  - 16 registers in our case: `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, ...
- It executes one **simple** instruction after the other
  - Very **basic** instructions
- Examples of instructions:
  - `mov $1, %eax` → write (i.e. **mov**e) integer 1 to register `%eax`
  - `jmp .label` → **jump** to position labelled `.label`
  - `imul %edi, %eax` → **mul**tiply `%edi` by `%eax`, put result in `%eax`
  - `dec %edi` → **dec**rement `%edi`
- Also: read from/write to the **main memory** (RAM) of the computer

Example of compilation

- **Convention**: parameter taken in %edi, result computed in %eax

```
int fact(int n) {
  int res = 1;
  while(n > 1) {
    res = res * n;
    n = n - 1;
  }
  return res;
}
```

⟹

```
fact:
        mov     $1, %eax
.L5:
        cmp     $1, %edi
        jle     .L8
        imul    %edi, %eax
        dec     %edi
        jmp     .L5
.L8:
        ret
```

- How does the compiler do?
  - We give **one possible** compilation chain.

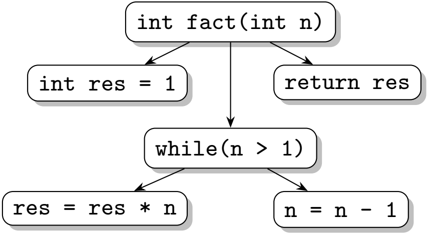
Step 1: lexing + parsing

Textual representation

```
int fact(int n) {
  int res = 1;
  while(n > 1) {
    res = res * n;
    n = n - 1;
  }
  return res;
}
```

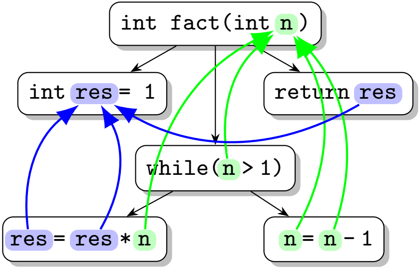
$\Longrightarrow$

Syntax tree



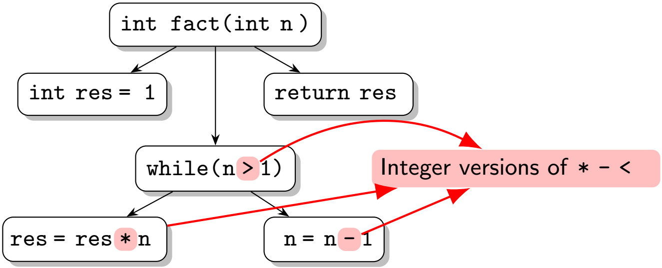
Step 2: front-end

- scope resolution
- resolution of overloading
- type checking



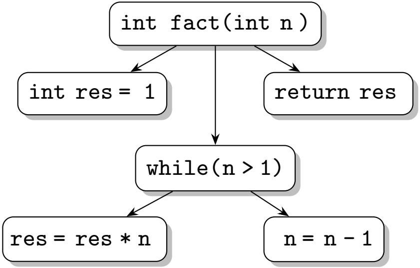
Step 2: front-end

- scope resolution
- resolution of overloading
- type checking



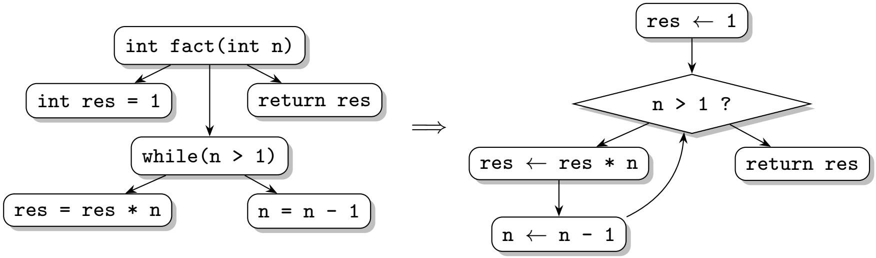
Step 2: front-end

- scope resolution
- resolution of overloading
- type checking



Step 3: CFG construction

- Structured syntax: for you, not for processors
- We identify **control points** in the source
- Control points: nodes in the Control Flow Graph:

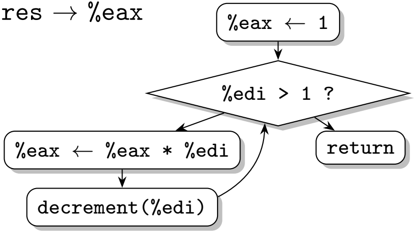


Step 4: Optimizing the CFG

- **Many** different passes ($>$ 100 in GCC)
  - Common subexpressions elimination
  - Constant propagation
  - Dead code elimination
  - Loop optimizations
  - ...
- Many depend on **static analyses**
  - SAs predict some properties of the program before execution
- In our case, only one optimization:
  - `n = n - 1` $\Rightarrow$ `decrement(n)`

Step 5: Register allocation

- We search for memory for storing variables
  - Registers are fast but limited
  - Main memory is huge but slow and more difficult to access
- In our case:
  - The initial value of `n` is given in `%edi`
  - The result is returned in `%eax`
  - Best solution: `n` → `%edi`   `res` → `%eax`



Step 6: Linearization

- We still have a control flow **graph**
- In assembly, instructions have a linear order
- We need to find an order for CFG nodes
  - When the successor of a node is not following it: insert a jump
  - Minimize the number of jumps

Finally:

```
        mov     $1, %eax
.L5:
        cmp     $1, %edi
        jle     .L8
        imul    %edi, %eax
        dec     %edi
        jmp     .L5
.L8:
        ret
```

Work at Gallium – Proving compilers and static analyzers

- What is a **correct** compiler?
  - "Any behavior of the generated code is allowed by the source code"
- How do we prove that?
  - **Formal semantics**: description of source and generated languages.
  - We prove that the source **simulates** the generated assembly.
- My work: verifying **static analyzers**
  - Predicting the behavior of the program before its execution
  - For better **optimizations**
  - For **avoiding bugs**

Conclusion

- We have omitted many, many details
- Compilers are truly interestings objects
  - Interesting problems
  - Many users (you all!)

Questions?