Compilation, semantics, assembly, etc.

Richard Bonichon

20190117

Outline

Programming is hard

Overview of a compiler

Compilers are complex

Language surprises zoo

Programming is hard

90%

of programmers

make some kind of errors when coding binary search.

The div of death

```c
#include <stdio.h>

int div(int a) {
     return a / a;
}

int main () {
     printf("d5 = %d  ", div(5));
     printf("d0 = %d\n", div(0));
     return 0;
}
```

The answer(s)

gcc 8.2.1	clang 7.0.1
$d5 = 1$ $d0 = 1$	core dumped
$d5 = 1$ $d0 = 1$	$d5 = 1$ $d0 = 1$
$d5 = 1$ $d0 = 1$	$d5 = 1$ $d0 = 1$
$d5 = 1$ $d0 = 1$	$d5 = 1$ $d0 = 1$

What is printed ?

```c
#include "stdio.h"

long foo(int *x, long *y) {
  *x = 0;
  *y = 1;
  return *x;
}

int main(void) {
  long l;
  printf("%ld\n", foo((int *) &l, &l));
  return 0;
}
```

Answer(s) on x86

-O	gcc 8.2.1	clang 7.0.1
0	1	1
1	1	0
2	0	0
3	0	0

Overview of a compiler

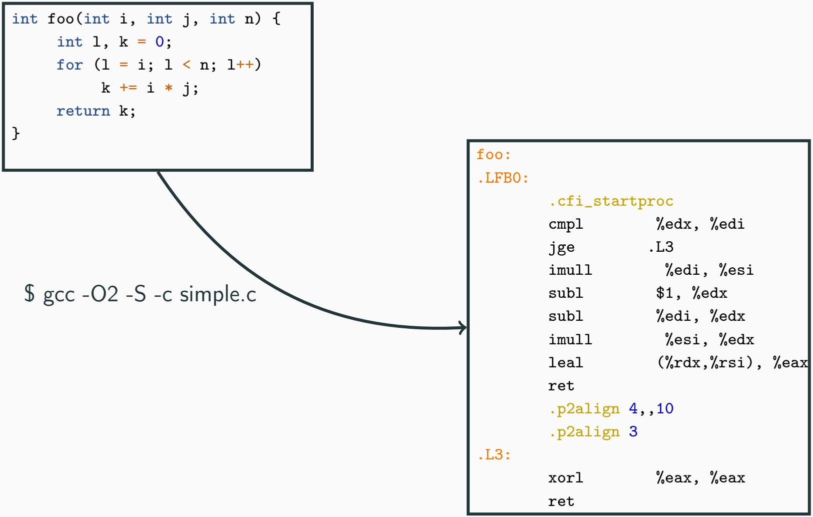
What is a compiler ?

Definition

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language).

The most common reason for wanting to transform source code is to create an executable program.

Compilation



```
int foo(int i, int j, int n) {
     int l, k = 0;
     for (l = i; l < n; l++)
          k += i * j;
     return k;
}
```

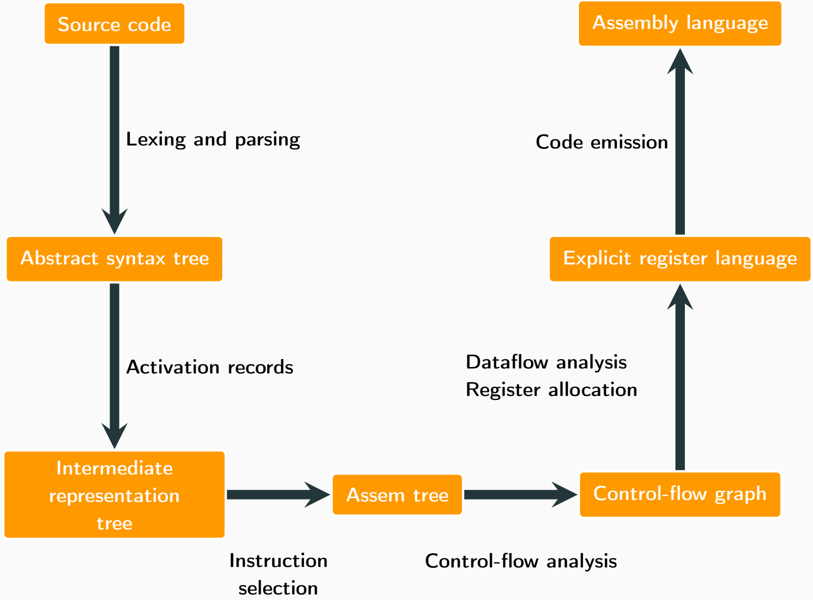
$ gcc -O2 -S -c simple.c

```
foo:
.LFB0:
        .cfi_startproc
        cmpl            %edx, %edi
        jge            .L3
        imull            %edi, %esi
        subl            $1, %edx
        subl            %edi, %edx
        imull            %esi, %edx
        leal            (%rdx,%rsi), %eax
        ret
        .p2align 4,,10
        .p2align 3
.L3:
        xorl            %eax, %eax
        ret
```

Fair warning

A sufficiently advanced compiler is indistinguishable from an adversary.
– John Regehr

Architecture of a modern compiler



A compiler must preserve the semantics of the original program through its many passes.

Semantics

Definition

- Semantics detail the meaning of the program (its statements, expressions, . . .)
- Formal semantics interpret programs using mathematics

Uses

Understanding a programming language

- what we can trust as regular programmers
- what we need to give as compiler programmers

Tool for designing languages

Fundamentals to show/prove properties of programs

Different types of semantics

Operational semantics

- What the program computes
- Concrete

Denotational semantics

- What the program computes
- Abstract

Axiomatic semantics

- Properties of programs

Example of operational semantics

$$\frac{}{\langle x := a, s\rangle \to s[x \mapsto \mathcal{A}[\![a]\!]s]}\ \text{Assigns}$$

$$\frac{\langle S_1, s\rangle \to s' \qquad [\![b]\!]s = \top}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \to s'}$$

$$\frac{}{\langle \text{skip}, s\rangle \to s}\ \text{Skip}$$

$$\frac{\langle S_2, s\rangle \to s' \qquad [\![b]\!]s = \bot}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s\rangle \to s'}$$

$$\frac{\begin{array}{c}\langle S_1, s\rangle \to s' \\ \langle S_2, s'\rangle \to s''\end{array}}{\langle S_1; S_2, s\rangle \to s''}\ \text{Seq}$$

These rules describe a simple imperative language without loops.

Lexing

Goal

Break the input into lexical unit (tokens)

"Does the teacher like compilation ?"

$\Rightarrow$

"Does", "the", "teacher", "like", "compilation", "?"

Parsing

Goal

Check the structure of sentences (i.e. the grammar)

A question of the form

Auxiliary/modal subject (main verb) (direct object) (question mark)

is grammatically valid.

Keywords

Lexing

- Regular expressions
- NFA
- DFA
- Minimization

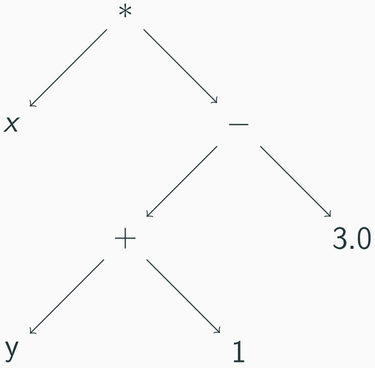
Parsing

- BNF
- LL(k)
- LR(k)

Lexing and parsing transform a concrete syntax tree into an abstract syntax tree.

Abstract Syntax Tree : $x * ((y + 1) - 3)$

```
x * ((y + 1) - 3)
```



Typing

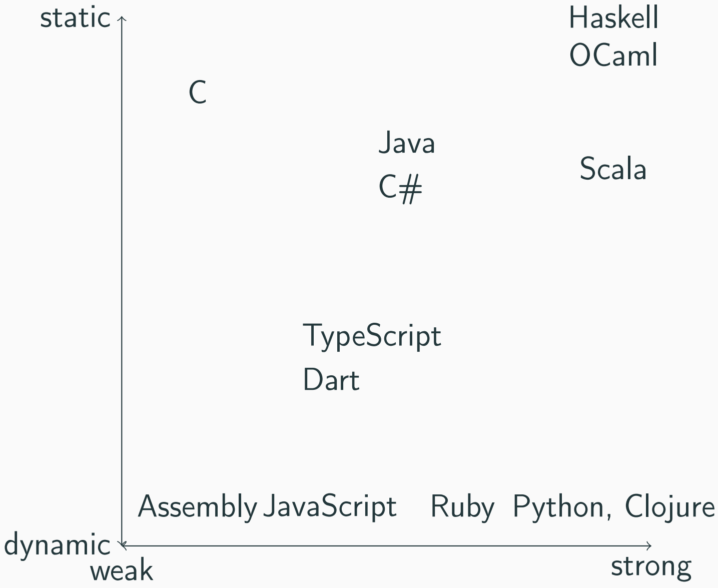
Definition (Typing)

Typing consists in attributing types to the data of the program

What for ?

Guarantee that programs make sense, i.e. are valid programs.

Typing systems landscape (Odersky)



Example

$$\frac{\Gamma \vdash b : \mathit{bool} \qquad \Gamma \vdash E_1 : T \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash \text{if } b \text{ then } E_1 \text{ else } E_2 : T}\ \text{if}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T}{\Gamma \vdash x := e : \mathit{unit}}\ \text{Assigns}$$

Intermediate representation

What is an IR ?

An Intermediate representation (IR) is the data structure or code used internally by a compiler to represent source code, usually for further processing (optimization, translation)

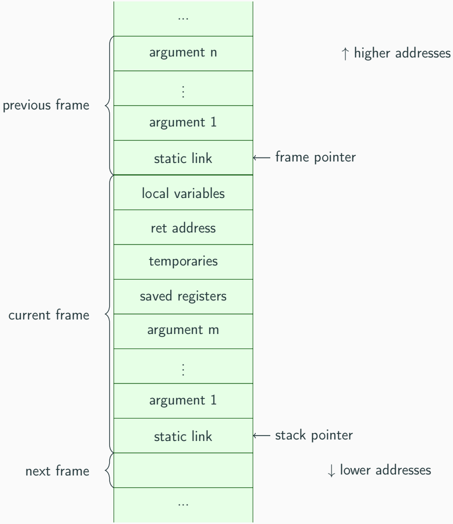
A good IR must be:

- accurate (no loss of information)
- independent of source/target languages.

Examples

- LLVM
- Gimple

Stack frame allocation



...

argument n

↑ higher addresses

⋮

previous frame

argument 1

static link ⟵ frame pointer

local variables

ret address

temporaries

saved registers

current frame

argument m

⋮

argument 1

static link ⟵ stack pointer

next frame

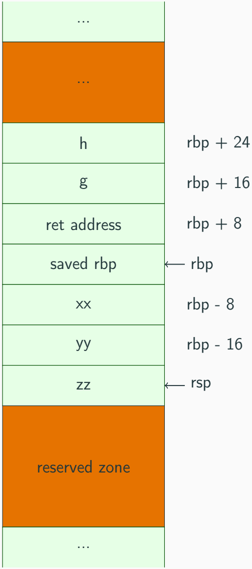
↓ lower addresses

...

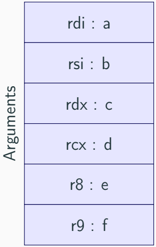
Example

```c
long myfunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    long xx = a * b * c * d * e * f * g * h;
    long yy = a + b + c + d + e + f + g + h;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}
```

Stack on calling `myfunc`



...	
...	
h	rbp + 24
g	rbp + 16
ret address	rbp + 8
saved rbp	⟵ rbp
xx	rbp - 8
yy	rbp - 16
zz	⟵ rsp
reserved zone	
...	



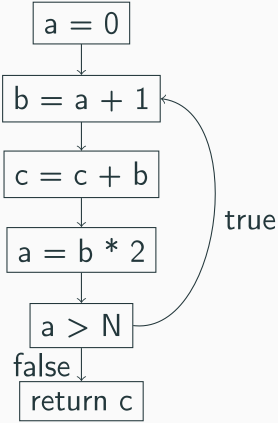
Arguments
rdi : a
rsi : b
rdx : c
rcx : d
r8 : e
r9 : f

CFG

```
#define N 10

int main () {
  int a,b,c;

  a = 0;
 l1:
  b = a + 1;
  c = c + b;
  a = b * 2;
  if (a < N) goto l1;
  return c;
}
```



CFG construction in a nutshell

Howto

- Every node has one statement;
- A directed edge connects two nodes N and M whenever M can be executed right after N in the program

Remarks

In order to know if one statement can follow another, one needs precise semantics!

CFGs can be constructed directly from the AST or after it: it is a basic data structure of compilation or static analysis.

CFGs provide a means to compute reachability of a given program part. An unreachable code in the CFG:

- will never ever be executed and
- can safely be removed from the program at compile time (this is dead code).

Optimizations are done on the CFG through data-flow analyses.

Common subexpression elimination

Definition

Given a statement

- $s : t \leftarrow x \odot y$,

where the expression $x \odot y$ is available at s,

the computation within s can be eliminated.

Example CSE

Before

```
a = b * c + g;
d = b * c * e;
```

After

```
tmp = b * c;
a = tmp + g;
d = tmp * e;
```

Constant/copy propagation

Definition

Suppose we have a statement s_1 : x $\leftarrow$ t,

where t is either a constant, or a simple variable.

And another : s_2 : y $\leftarrow$ x bop z.

x is constant in s_2 if:

- s_1 reaches s_2 and
- no other definition of x reaches s_2

In this case : s_2 : y $\leftarrow$ t bop z

Example

Before

```
t = 12;
x = 4;
y = t;
z = x * y - t;
```

After

```
t = 12;
x = 4;
y = t;
z = 36;
```

Dead code elimination

Definition

If there is a quadruple

- $s : a \leftarrow b \odot c$; or
- $s : a \leftarrow M[x]$,

such that a is not live-out of s,

then the quadruple can be deleted.

Example

Before

```
t = 12;
x = 4;
y = t;
z = 36;
```

After

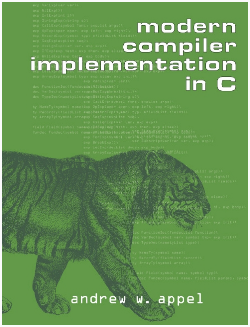
```
z = 36;
```

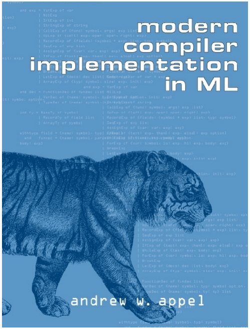
Register allocation

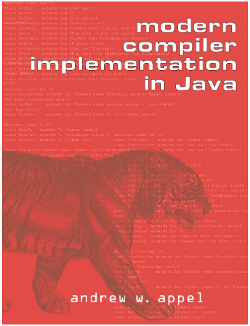
The job of the register allocator is to assign:

- the many temporaries to a small number of machine registers
- — where possible — the source and destination of a MOVE to the same register so that the MOVE can be deleted.

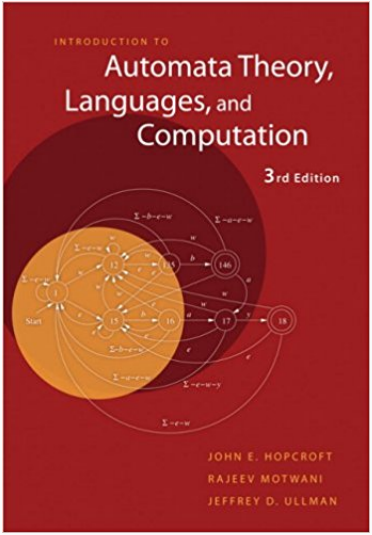
More on compilation

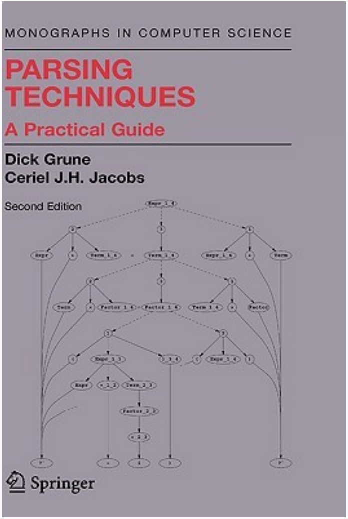






Lexing / parsing





Compilers are complex

Source of errors

Coding in C (for example) exposes the programmer to several difficulties

1. Tricky semantics
2. Unforeseen optimizations
3. Undefined behaviors (might be seen as tricky semantics), due to the fact that it is an unsafe language

Only 3 is specific to C . . .

One more

```c
#include <stdio.h>

int main(void)
{
     unsigned char a = 0xff;
     char b = 0xff;
     int c = a == b; // true, or false?
     printf("c = %d\n",c);
}
```

Division by zero

```c
/* Linux kernel : lib/mpi/mpi-pow.c */

if (!msize)
    msize = 1 / msize; /* provoke a signal */
```

Oversized shift (Fix for CVE-2009-4307)

```c
/* Linux kernel: fs/ext4/super.c */

groups_per_flex = 1 << sbi->s_log_groups_per_flex;
/* There are some situations, after shift the value of 'groups_per_flex' can
   become zero  and divsion with 0 result in fixpoint divide exception.
*/
if (groups_per_flex == 0) return 1;

flex_group_count = ... / groups_per_flex;
```

Silent breakage (from Regehr)

```c
#include <limits.h>
#include <stdio.h>

int foo(int x) {
    return (x + 1) > x;
}

int main(void) {
    printf("%d\n", (INT_MAX + 1) > INT_MAX);
    printf("%d\n", foo(INT_MAX));
    return 0;
}
```

`INT_MAX + 1` is both larger and not larger than `INT_MAX`.

Mixing signed/unsigned

```c
#include <stdio.h>

int main (void)
{
     long a = -1;
     unsigned b = 1;
     printf ("%d\n", a > b);
     return 0;
}
```

Why, oh why ?

CPUs are typically fastest on integers at their native size.

On x86, 32-bit arithmetic can be twice as fast as 16-bit one.

C is a language focused on performance, so it will do the integer promotion to make the program as fast as possible.

In a nutshell

Keep integer promotion rules in mind to avoid integer overflow vulnerability issues.

Undefined behaviors

Some C operations are left implementation-defined but other are undefined in the Standard.

C compilers trust the programmer not to submit code with undefined behaviors.

They optimize code under such assumptions.

> *Permissible undefined behavior ranges from ignoring the situation completely, with unpredictable results, to having demons fly out of your nose.*

But it works on my computer !

Somebody once told me that in basketball you can't hold the ball and run.
I got a basketball and tried it and it worked just fine.
He obviously didn't understand basketball.
– Roger Miller

Why is it good and bad ?

Good

- Makes compiler's job easier
  For example, loop optimizations do not have to worry about signed integers overflows — it is undefined behavior.

Bad

- 191 kinds of undefined behaviors in C99

Security problems

```c
void process_something(int size)
{
    // Catch integer overflow.
    if (size > size + 1) abort();
    ... // Error checking from this code elided.

    char *string = malloc(size + 1);
    read(fd, string, size);
    string[size] = 0;
    do_something(string);
    free(string);
}
```

Optimization is hard

```
void contains_null_check(int *p)
{
     int dead = *p;
     if (p == 0)
          return;
     *p = 4;
}
```

Unwanted dead code elimination

```c
void check_password(char *pwd);

void get_password(void)
{
    char pwd[64];
    if (retrieve_password(pwd, sizeof(pwd))) {
        check_password(pwd);
    }
    memset(pwd, 0, sizeof(pwd));
}
```

What is printed ?

```c
#include "stdio.h"

long foo(int *x, long *y) {
  *x = 0;
  *y = 1;
  return *x;
}

int main(void) {
  long l;
  printf("%ld\n", foo((int *) &l, &l));
  return 0;
}
```

The answer(s)

gcc 8.2.1	clang 7.0.1
1	1
1	0
0	0
0	0

What is returned ?

```cpp
#include <iostream>
#include <complex>
using namespace std;

int main() {
    complex<int> delta;
    complex<int> mc[4] = {0};
    int di;

    for(di = 0; di < 4; di++, delta = mc[di]) {
        cout << "di:" << di << endl;
        cout << "delta: " << delta << endl;
    }
    cout << "mc[di]:" << mc[di] << endl;
    return 0;
}
```

Take away

At low-level, there is (almost) no undefined behavior.

aka

Low-level does not lie.

Language surprises zoo

Java

```java
public class Main {

 public static void main(String[] args) {
      int a1 = 1000, a2 = 1000;
      System.out.println(a1 == a2);
      Integer b1 = 1000, b2 = 1000;
      System.out.println(b1 == b2);
      Integer c1 = 100, c2 = 100;
      System.out.println(c1 == c2);
 }
}
```

OCaml (< 4.05.0)

```
open Nums

let x = Big_int.big_int_of_int 1 ;;

x = x ;;
```

OCaml

```ocaml
let s = string_of_bool true ;;

s.[0] <- 'f' ;;
s.[1] <- 'a' ;;
s.[3] <- 'x' ;;

1 = 1;;

Printf.printf "1 = 1 est %b\n" (1 = 1) ;;
```

OCaml

```ocaml
(*#warnings "-3";; (* :-*) *)

let f x =
  match x with
  | true  -> "T"
  | false -> "F"
;;

f true ;;
f false ;;

(f false).[0] <- 'T' ;;
(f true).[0] <- 'F' ;;

f true ;;
f false ;;
```

Python

```python
l = [ s for s in [1, 2, 3] ]
print(l)
print(s)
```

JavaScript: arithmetic

```
1 / 0

NaN == NaN

999999999999999

9999999999999999

"2" + 1

"2" - 1

"2" - - 1
```

JavaScript: ==

```
[1] == [1]

[] == ![]

[1] == true

2 == [2]

0 == '0'

0 == '0.0'

'0' == '0.0'

null == undefined
```

More fun

- `https://www.youtube.com/watch?v=et8xNAc2ic8`
- `https://github.com/denysdovhan/wtfjs226`

PHP

```
$x = "2d8" ;
$y = "2d8" ;

++$x == $y + 1;

print (++ $x . "\n") ;
print (++ $x . "\n") ;

print($y + 1 . "\n") ;
```

PHP

```
$h1 = md5 ('QNKCDZO') ;
$h2 = md5 ('240610708') ;
$h3 = md5 ('A169818202') ;
$h4 = md5 ('aaaaaaaaaaaumdozb') ;
$h5 = sha1('badthingsrealmlavznik') ;
```

Which ones are == to each other ?

a) none

b) h3 and h5

c) h1, h3 and h4

d) *La réponse D*

Scala

```scala
List("a", "b", "c").toSet() + "d"
```

Questions ?

https://rbonichon.github.io/teaching/2018/asi36/