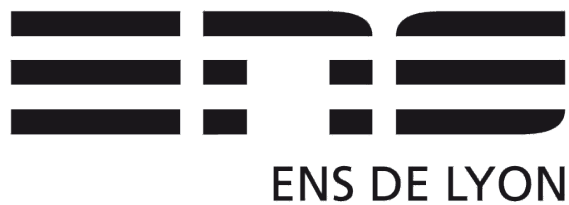
Introduction - Compilation Courses - 2020

Laure Gonnord & Matthieu Moy & other

https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2020-2021



Your teachers

Figure: Laure Gonnord (CAP) - Matthieu Moy (MIF08)

- Photo © Inria / G .Scagnelli

1 Intro, what's compilation

2 Compiler phases

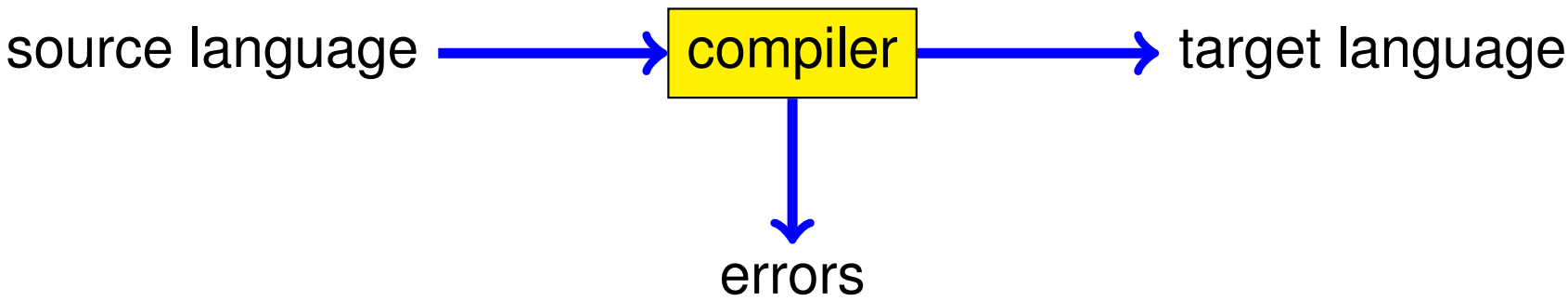
3 The RISCV architecture in a nutshell

4 One example

Credits

A large part of the compilation part of this intro course is inspired by the Compilation Course of JC Filliâtre at ENS Ulm who kindly offered the source code of his slides.

What's compilation?



Compilation toward the machine language

We immediatly think of the translation of a high-level language (C,Java,OCaml) into the machine language of a processor (Pentium, PowerPC...)

```
% gcc -o sum sum.c
```

```
int main(int argc, char **argv) {
  int i, s = 0;
  for (i = 0; i <= 100; i++) s += i*i;
  printf("0*0+...+100*100 = %d\n", s);}
```

$\longrightarrow$

```
00100111101111011111111111100000101011110111111000000000010100
10101111101001000000000001000001010111110100101000000000100100
10101111101000000000000000110001010111110100000000000000011100
10001111101011100000000000011100
```

Target Language

This aspect (compilation into assembly) will be presented in this course, but we will do more:

Compilation is not (only) code generation

A large number of compilation techniques are not linked to assembly code production.

Moreover, languages can be

- interpreted (Basic, COBOL, Ruby, Python, etc.)
- compiled into an intermediate language that will be interpreted (Java, OCaml, Scala, etc.)
- compiled into another high level language (or the same !)
- compiled "on the fly" (or just on time)

Compiler/ Interpreter

- A compiler translates a program P intro a program Q such that for all entry x, the output $Q(x)$ is the same as $P(x)$.

$$\forall P\ \exists Q\ \forall x...$$

- An interpreter is a program that, given a program P and an entry x, computes the output of $P(x)$:

$$\forall P\ \forall x\ \exists s...$$

Compiler vs Interpreter

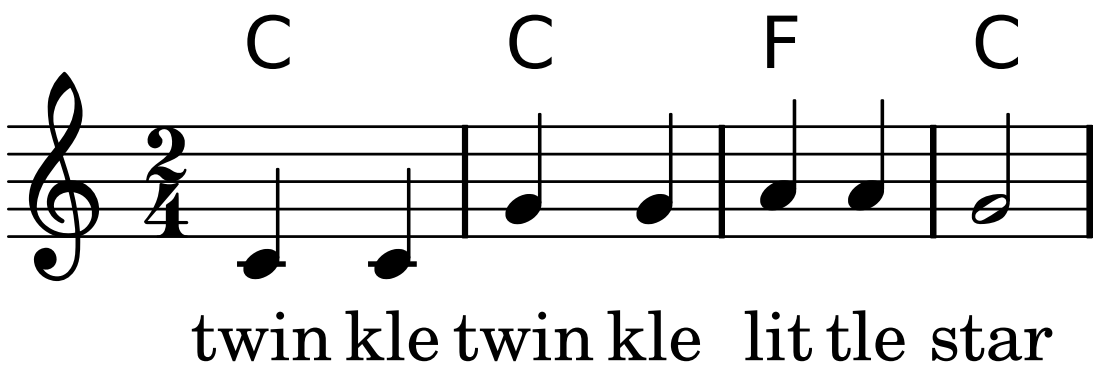
Or :

- The compiler makes a complex work once, to produce a code for whatever entry.
- An interpreter makes a simpler job, but on every entry.

- In general the code after compilation is more efficient.

Example

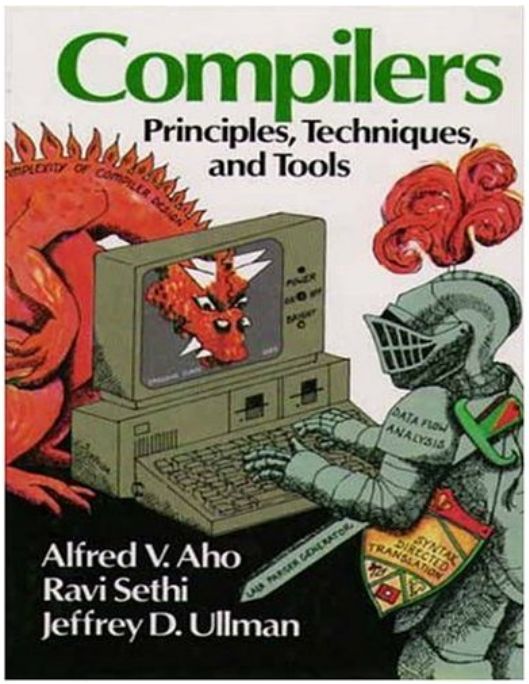

```
\chords { c2 c f2 c }
\new Staff \relative c' { \time 2/4 c4 c g'4 g a4 a g2 }
\new Lyrics \lyricmode { twin4 kle twin kle lit tle star2
```



Compiler Quality

Quality criteria ?

- correctness
- efficiency of the generated code
- its own efficiency



"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct."

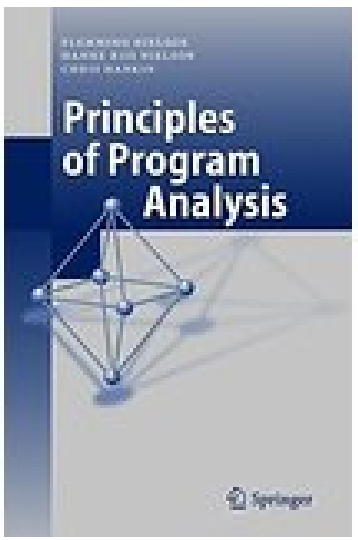
(Dragon Book, 2006)

Program Analysis ENSL only

To prove:

- Correctness of compilers/optimisations phases.
- Correctness of programs: invariants

. . . the second part of the course.



Course Objective

Be familiar with the mechanisms inside a (simple) compiler.

ENSL only Be familiar with basis of program analysis.

- And understand the links between them!

Course Content - Compilation Part

- Syntax Analysis : lexing, parsing, AST, types.
- Evaluators.
- Code generation.
- Code Optimisation.

Lab : a complete compiler for the RISCV architecture !

Support language: Python 3

Frontend infrastructure : ANTLR 4.

Course Content - Analysis Part ENSL Only!

- Concrete semantics (many versions!)
- Abstract Interpretation
- Language extensions

Labs : abstract interpretation, and language extension. Support language: Python 3.

Course Organization

Everything is on the webpage:

https://compil-lyon.gitlabpages.inria.fr/

Read your emails !

1 Intro, what's compilation

2 Compiler phases

3 The RISCV architecture in a nutshell

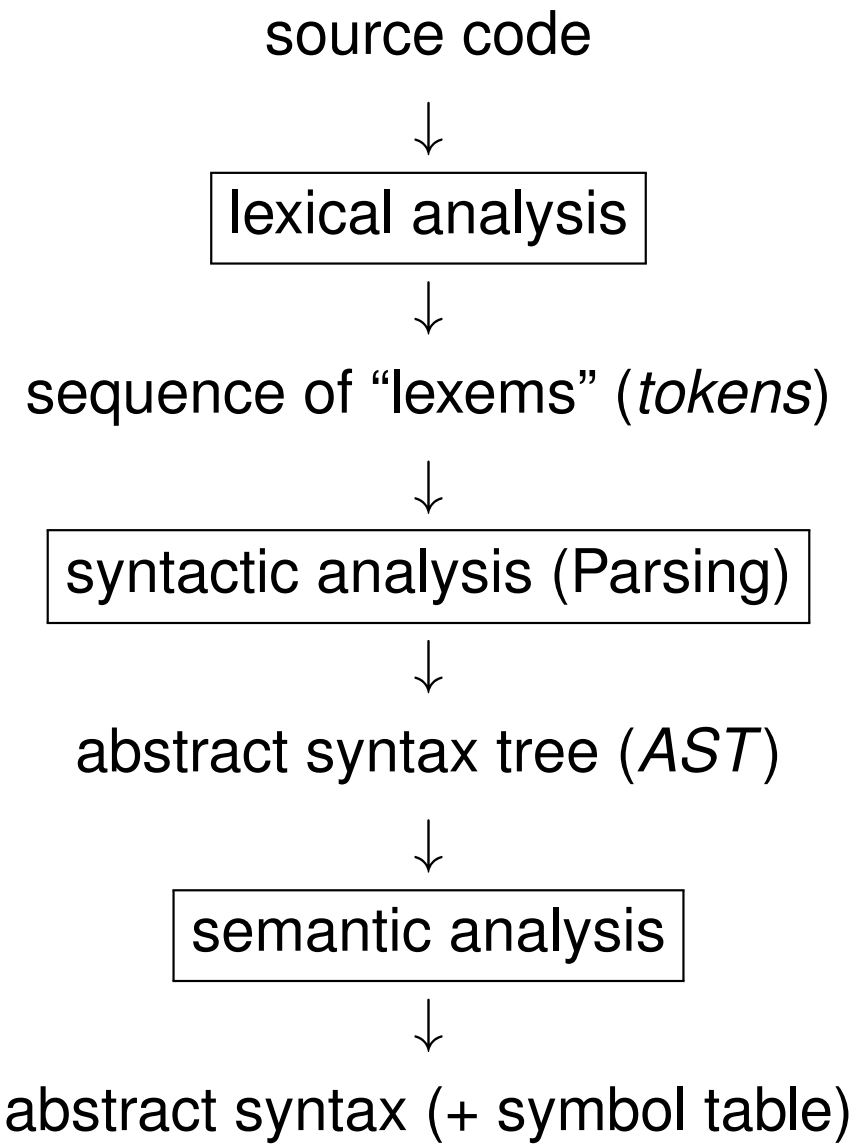
4 One example

Compiler phases

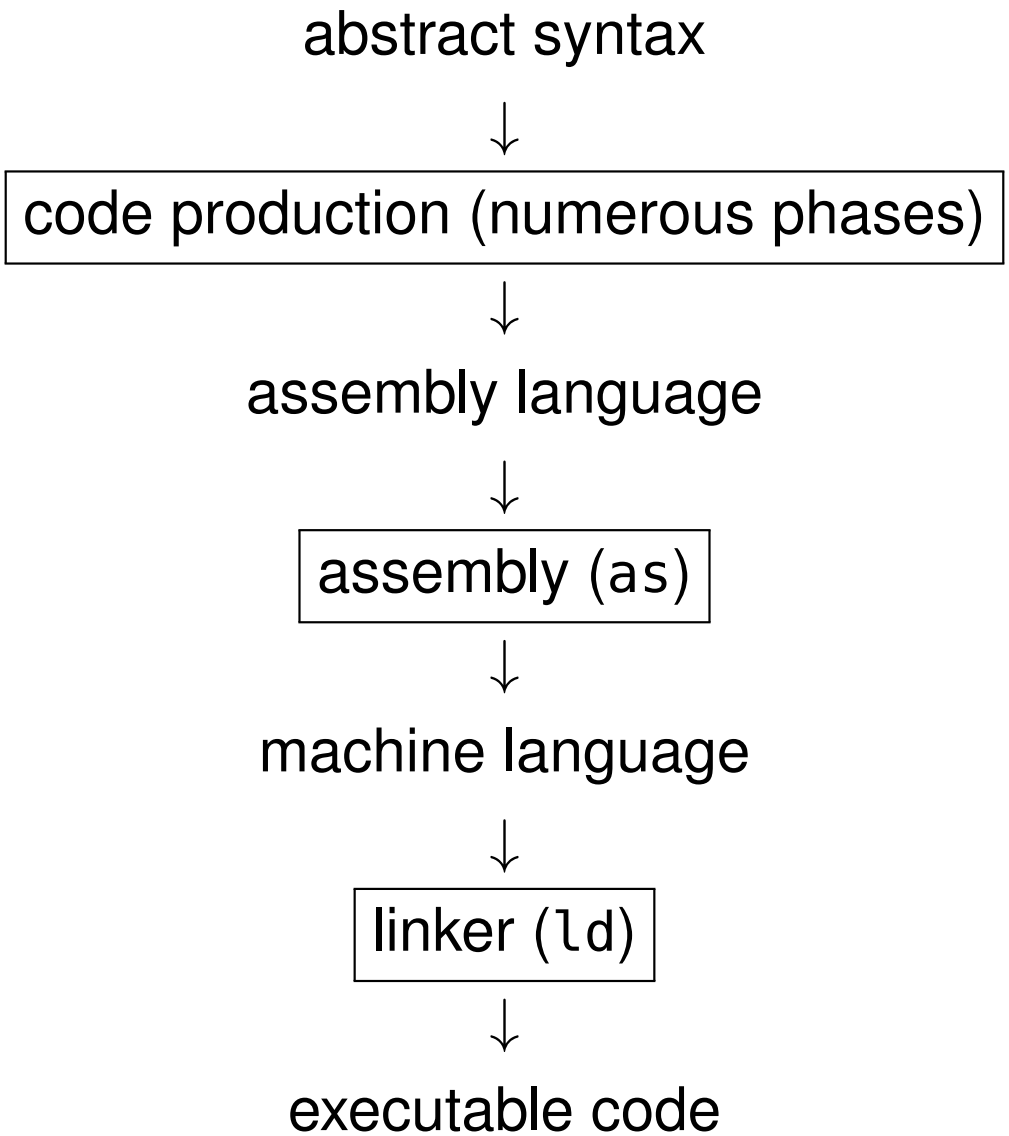
Usually, we distinguish two parts in the design of a compiler:

- an analysis phase:
  - recognizes the program to translate and its meaning.
  - raises errors (syntax, scope, types . . .)
- Then a synthesis phase:
  - produces a target file.
  - sometimes optimises.

Analysis Phase



Synthesis Phase



NEXT:

assembly

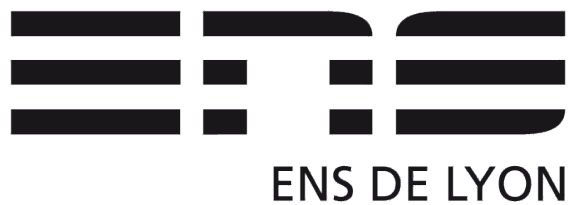
Introduction - Compilation Courses - 2020

Laure Gonnord & Matthieu Moy & other

https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2020-2021



1 Intro, what's compilation

2 Compiler phases

3 The RISCV architecture in a nutshell

4 One example

Our target machine : RISCV

Excerpts from https://en.wikipedia.org/wiki/RISC-V

> *RISC-V (pronounced "risk-five") is an open-source hardware instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. [...] RISC-V has a modular design, consisting of alternative base parts, with added optional extensions.*

- We will use a subset of the RV64I Base Integer Instruction Set, 64-bit + some shortcuts.

RISCV ecosystem

- Versions of state-of-the-art compilers are available: `riscv64-unknown-elf-gcc` for us.
- ISA simulators are available: `spike` for us.

RISCV registers

- Memory is addressed as 8-bit bytes
- 64-bit words can be accessed with the load (ld) and store (sd) instructions.
- In the RV64I version, instructions are encoded on 32 (32 as well in the RV32I)
- 32 (64-bit) registers, the first integer register is a zero register, and the rest is general purpose (but have symbolic names to implement standard conventions). Use only symbolic names when you write code
- In the base RV64I ISA, there are four core instruction formats (R/I/S/U).

RISCV ISA

We provide you an external document with a summary of the ISA.

Example : ADD instructions

- `add rd, rs1, rs2`, does `rd <- rs1 + rs2`.
  - ↪ All operands are registers.
  - ↪ Example : `add t1, t2, t3` executes `t1 <- t2 + t3`.

- `addi rd, rs1, imm`, does `rd <- rs + imm`.
  - ↪ The last operand is an immediate value (on xx bits) encoded in the instruction.
  - ↪ Example : `addi t1, t2, 5` executes `t1 <- t2 + 5`.

RISCV ADD/ADDi : encoding

R or I-typed instructions

class	action	encoding
add rd, ri, rj (R)	$r_d \leftarrow r_i + r_j$	`0000000|<rj(5bits)>|<ri(5bits)>|000|<rd(5bits)>|0110011`
addi rd, ri, cte (I)	$r_d \leftarrow r_i + cte$	`<cte(12bits)>|<ri(5bits)>|000|<rd(5bits)|0010011`

Example: assemble `addi t1, t2, 5`

RISCV: branching

Unconditional branching:

- `jal rd, c`, does `rd=PC+4; PC += c` (focus on PC for the moment)

Test and branch :

- `blt rs1, rs2, c`, does `PC += c` if $rs1 < rs2$

- Shortcuts : `j label` and `blt rs1, rs2, label`
- The label is assembled into the adequate offset of the jump.
- See the list of operators in the companion sheet.

RISCV Memory accesses instructions 1/2

- Load from memory (64-bit word) $r_d \leftarrow Mem[r_s + off]$:

```
1 ld rd, off(rs)
```

- Store to memory:

```
sd rs, off(rd)
```

- Load effective address (shortcut)

```
la rd, label
```

See the ISA for more info.

1 Intro, what's compilation

2 Compiler phases

3 The RISCV architecture in a nutshell

4 One example

Ex : Assembly code - demo

```
#simple RISCV assembly demo
#riscv64-unknown-elf-gcc  demo20.s ../../TP2020-21/TP01/code/libprint.s -o demo20
#spike pk demo20
        .text
        .globl main
main:
        addi    sp,sp,-16
        sd      ra,8(sp)
# your assembly code here
        addi t1, zero, 5              # first op : cte
        la t3, mydata                 # second, from memory
        ld t4, 0(t3)
        add a0, t1, t4                # add --> a0 = result
        call print_int
        call newline
## /end of user assembly code
        ld      ra,8(sp)
        addi    sp,sp,16
        ret
        .section .rodata
mydata:
        .dword 37
```

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
addi t1, s1, 0
```

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
addi t1, s1, 0
```

Format I

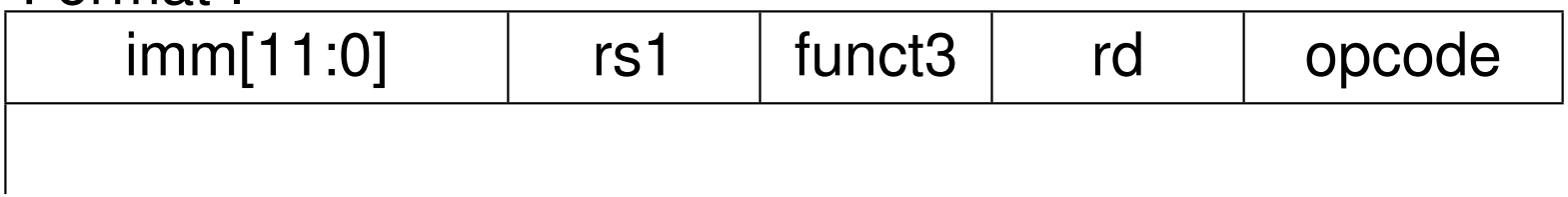
Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
addi t1, s1, 0
```

Format I



Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
addi t1, s1, 0
```

Format I

imm[11:0]	rs1	funct3	rd	opcode
0	9	0	6	0010011
0000 0000 0000	0100 1	000	0011 0	001 0011

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
addi t1, s1, 0
```

Format I

imm[11:0]	rs1	funct3	rd	opcode
0 0000 0000 0000	9 0100 1	0 000	6 0011 0	0010011 001 0011
0x00048313				

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Type I, lb, lw or lbu

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Type I, lb, lw or lbu

imm[11:0]	rs1	funct3	rd	opcode
0000 0000 1000	0001 0	000	0011 1	000 0011

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Type I, lb, lw or lbu

imm[11:0]	rs1	funct3	rd	opcode
0000 0000 1000	0001 0	000	0011 1	000 0011

Func3 = 0 $\Rightarrow$ lb; rs = x2 = sp; rd = x7 = t2

Exercise: RISCV Disassembly

Disassemble the following instruction: 0x00810383

Binary = 0000 0000 1000 0001 0000 0011 1000 0011

Opcode = 000011

Type I, lb, lw or lbu

imm[11:0]	rs1	funct3	rd	opcode
0000 0000 1000	0001 0	000	0011 1	000 0011

Func3 = 0 $\Rightarrow$ lb; rs = x2 = `sp`; rd = x7 = `t2`

```
lb t2, 8(sp)
```

Exercise: RISCV Program

Write a program that counts from 0 to infinity.

We provide `call print_int` (to display `a0` as an integer) and `call newline`.

Exercise: RISCV Program

Write a program that counts from 0 to infinity.

We provide `call print_int` (to display `a0` as an integer) and `call newline`.

```
    .globl main
main:
    li a0, 0
lbl:
    call print_int
    call newline
    addi a0, a0, 1
    j lbl
```

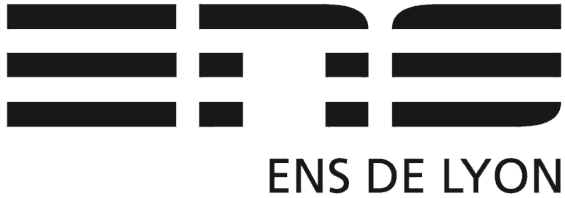
Lexing, Parsing (CAP+MIF08)

Laure Gonnord & Matthieu Moy & other

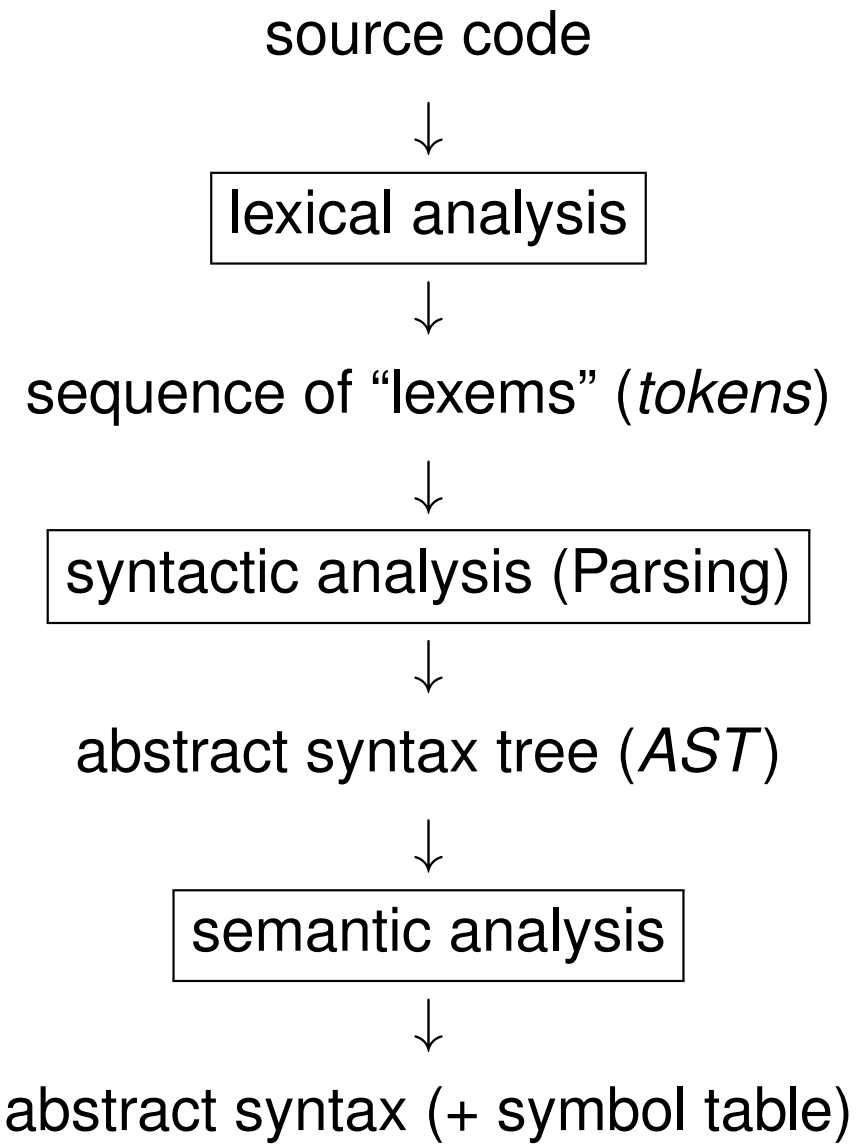
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2020-2021



Analysis Phase



Goal of this chapter

- Understand the syntaxic structure of a language;
- Separate the different steps of syntax analysis;
- Be able to write a syntax analysis tool for a simple language;
- Remember: syntax$\neq$semantics.

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens:
  - Words: groups of letters;
  - Punctuation;
  - Spaces.

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into:
  - Propositions;
  - Sentences.

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.
- Then proceed with word meanings:
  - Definition of each word.

    ex: a dog is a hairy mammal, that barks and...
  - Role in the phrase: verb, subject, ...

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.
- Then proceed with word meanings: **Semantics**
  - Definition of each word.

    ex: a dog is a hairy mammal, that barks and...
  - Role in the phrase: verb, subject, ...

Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.
- Then proceed with word meanings: **Semantics**
  - Definition of each word.

    ex: a dog is a hairy mammal, that barks and...
  - Role in the phrase: verb, subject, ...

Syntax analysis=Lexical analysis+Parsing

1 Lexical Analysis
- Principles
- Tools

2 Syntactic Analysis

What for ?

```
int y = 12 + 4*x ;
```

$\Longrightarrow$ [TINT, VAR("y"), EQ, INT(12), PLUS, INT(4), TIMES, VAR("x"), SCOL]

- Group characters into a list of **tokens**, e.g.:
  - The word “int” stands for type integer;
  - A sequence of letters stands for a variable;
  - A sequence of digits stands for an integer;
  - ...

1. Lexical Analysis
   - Principles
   - Tools
2. Syntactic Analysis
   - Principles
   - Tools

Principle

- Take a lexical description: $E = (\underbrace{E_1}_{\text{Tokens class}} | \ldots | E_n)^*$
- Construct an automaton.

Example - lexical description (“lex file”)

$E = ((0|1)^+|(0|1)^+.(0|1)^+|'+')^*$

What's behind

Regular languages, regular automata:

- Thompson construction ▶ non-det automaton
- Determinization, completion
- Minimisation

▶ And non trivial algorithmic issues (remove ambiguity, compact the transition table).

1. Lexical Analysis
   - Principles
   - Tools
2. Syntactic Analysis
   - Principles
   - Tools

Tools: lexical analyzer constructors

- Lexical analyzer constructor: builds an automaton from a regular language definition;
- Ex: Lex (C), JFlex (Java), OCamllex, **ANTLR** (multi), ...
- **input**: a set of regular expressions with actions (`Toto.g4`);
- **output**: a file(s) (`Toto.java`) that contains the corresponding automaton.

Analyzing text with the compiled lexer

- The **input of the lexer** is a text file;
- Execution:
  - Checks that the input is accepted by the compiled automaton;
  - Executes some actions during the “automaton traversal”.

Lexing tool for Java: ANTLR

- The official webpage : www.antlr.org (BSD license);
- ANTLR is both a lexer and a parser;
- ANTLR is multi-language (not only Java).

ANTLR lexer format and compilation

.g4

```
lexer grammar XX;
@header {
// Some init code...
}
@members {
// Some global variables
}
// More optional blocks are available
--->> lex rules
```

Compilation (using the java backend)

```
antlr4 Toto.g4          // produces several Java files
javac *.java            // compiles into xx.class files
java org.antlr.v4.gui.TestRig Toto tokens
```

Lexing with ANTLR: example

Lexing rules:

- Must start with an upper-case letter;
- Follow the usual extended regular-expressions syntax (same as `egrep`, `sed`, ...).

A simple example

```
lexer grammar Tokens;



HELLO : 'hello' ; // beware the single quotes
ID : [a-z]+ ; // match lower-case identifiers
INT : [0-9]+ ;
KEYWORD : 'begin' | 'end' | 'for' ; // perhaps this should be
    elsewhere
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Lexing - We can count!

Counting in ANTLR - CountLines2.g4

```
lexer grammar CountLines2;

// Members can be accessed in any rule
@members {int nbLines=0;}

NEWLINE : [\r\n] {
  nbLines++;
  System.out.println("Current lines:"+nbLines);} ;
WS : [ \t]+ -> skip ;
```

1 Lexical Analysis

2 Syntactic Analysis
- Principles
- Tools

1. Lexical Analysis
   - Principles
   - Tools
2. Syntactic Analysis
   - Principles
   - Tools

What's Parsing ?

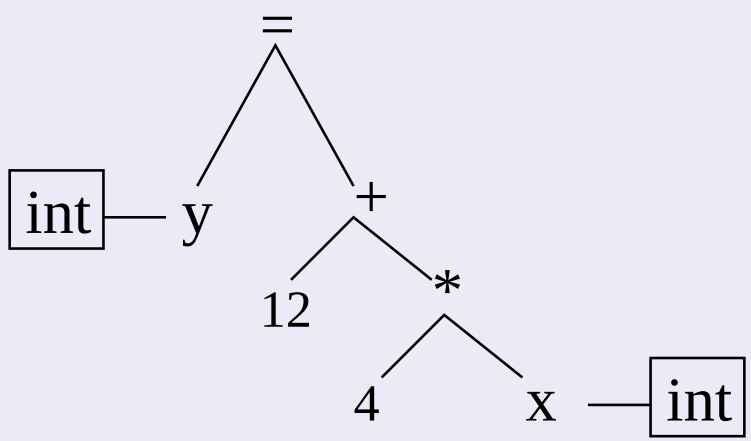
Relate tokens by structuring them.

Flat tokens

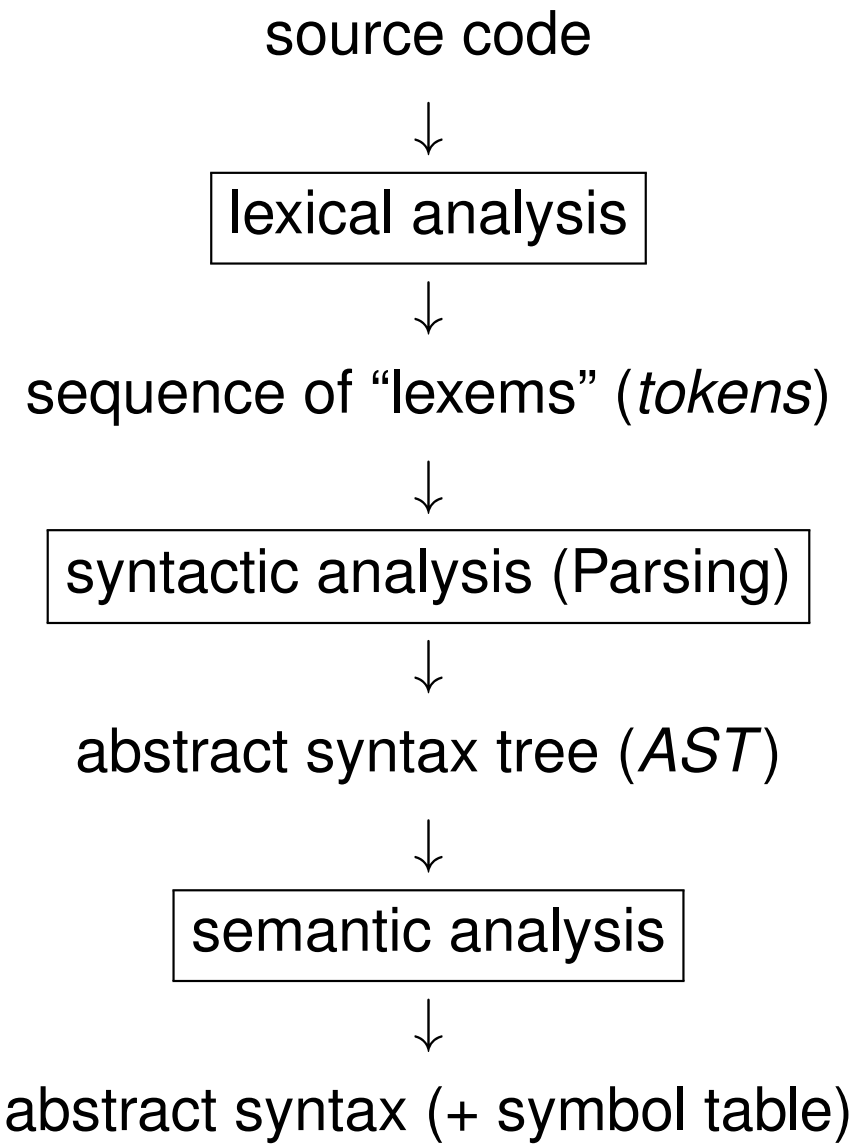
[TINT, VAR("y"), EQ, INT(12), PLUS, INT(4), TIMES, VAR("x"), SCOL]

$\Rightarrow$ **Parsing** $\Rightarrow$

Accept $\rightarrow$ Structured tokens



Analysis Phase



In this course

Only write acceptors “OK” or “Syntax Error”.

What's behind ?

From a Context-free Grammar, produce a Stack Automaton (already seen in L3 course?)

Recalling grammar definitions

Grammar

A **grammar** is composed of :

- A finite set N of non terminal symbols
- A finite set Σ of terminal symbols (disjoint from N)
- A finite set of production rules, each rule of the form $w \rightarrow w'$ where w is a word on $\Sigma \cup N$ with at least one letter of N. w' is a word on $\Sigma \cup N$.
- A start symbol $S \in N$.

Example

Example:

$$S \to aSb$$

$$S \to \varepsilon$$

is a grammar with $N =$ .... and ....

Associated Language

Derivation

G a grammar defines the relation :

$$x \Rightarrow_G y \text{ iff } \exists u, v, p, q x = upv \text{ and } y = uqv \text{ and } (p \to q) \in P$$

- A grammar describes a **language** (the set of words on Σ that can be derived from the start symbol).

Example - associated language

$$S \to aSb$$

$$S \to \varepsilon$$

The grammar defines the language $\{a^n b^n, n \in \mathbf{N}\}$

$$S \to aBSc$$

$$S \to abc$$

$$Ba \to aB$$

$$Bb \to bb$$

The grammar defines the language $\{a^n b^n c^n, n \in \mathbf{N}\}$

Context-free grammars

Context-free grammar

A **CF-grammar** is a grammar where all production rules are of the form $N \rightarrow (\Sigma \cup N)^*$.

Example:

$$S \rightarrow S + S | S * S | a$$

The grammar defines a language of arithmetical expressions.

- Notion of **derivation tree**.

Exercise: draw a derivation tree of a*a+a (with the previous grammar).

Parser construction

There exists algorithms to recognize class of grammars:

- Predictive (descending) analysis (LL)
- Ascending analysis (LR)

- See the Dragon book.

1. Lexical Analysis
   - Principles
   - Tools
2. Syntactic Analysis
   - Principles
   - Tools

Tools: parser generators

- Parser generator: builds a stack automaton from a grammar definition;
- Ex: yacc(C), javacup (Java), OCamlyacc, **ANTLR**, ...
- **input** : a set of grammar rules with actions (`Toto.g4`);
- **output** : a file(s) (`Toto.java`) that contains the corresponding stack automaton.

Lexing then Parsing

Concretely, we need a way:

- To declare terminal symbols (**tokens**);
- To write grammars.

- Use both Lexing rules and Parsing rules.

Parsing with ANTLR: example

$$S \rightarrow aSb$$

$$S \rightarrow \varepsilon$$

The grammar defines the language $\{a^n b^n, n \in \mathbf{N}\}$

Parsing with ANTLR: example (cont')

AnBnLexer.g4

```
lexer grammar AnBnLexer;

// Lexing rules: recognize tokens
A: 'a' ;
B: 'b' ;

WS : [\t\r\n ]+ -> skip ; // skip spaces, tabs, newlines
```

Parsing with ANTLR: example (cont')

AnBnParser.g4

```
parser grammar AnBnParser;
options {tokenVocab=AnBnLexer;} // extern tokens definition

// Parsing rules: structure tokens together
prog : s EOF ; // EOF: predefined end-of-file token
s : A s B {System.out.println("rule S");}
    | ; // nothing for empty alternative
```

ANTLR expressivity

LL(*)

At parse-time, decisions gracefully throttle up from conventional fixed $k \geqslant 1$ lookahead to arbitrary lookahead.

Further reading (PLDI'11 paper, T. Parr, K. Fisher)

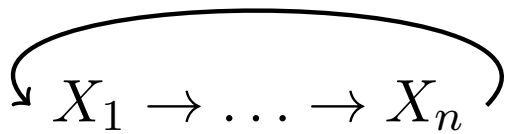
http://www.antlr.org/papers/LL-star-PLDI11.pdf

Left recursion

ANTLR permits left recursion:

```
a: a b;
```

But not indirect left recursion.



There exist algorithms to eliminate indirect recursions.

Lists

ANTLR permits lists:

```
prog: statement+ ;
```

Read the documentation!

https://github.com/antlr/antlr4/blob/master/doc/index.md

So Far ...

ANTLR has been used to:

- Produce acceptors for context-free languages;
- Do a bit of computation on-the-fly.

$\Rightarrow$ In a classic compiler, parsing produces an **Abstract Syntax Tree**.

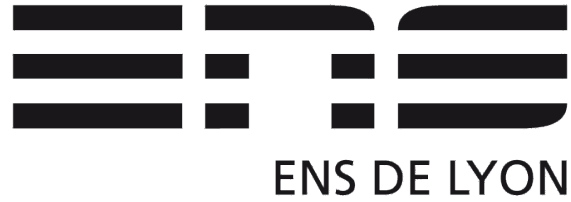
- Next course!

Compilation (#3): Semantics, Interpreters from theory to practice.

Laure Gonnord & Matthieu Moy & other
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2020-2021



1 Program Semantics

2 Grammars attributions and semantic actions

3 Useful notions: abstract syntax, AST

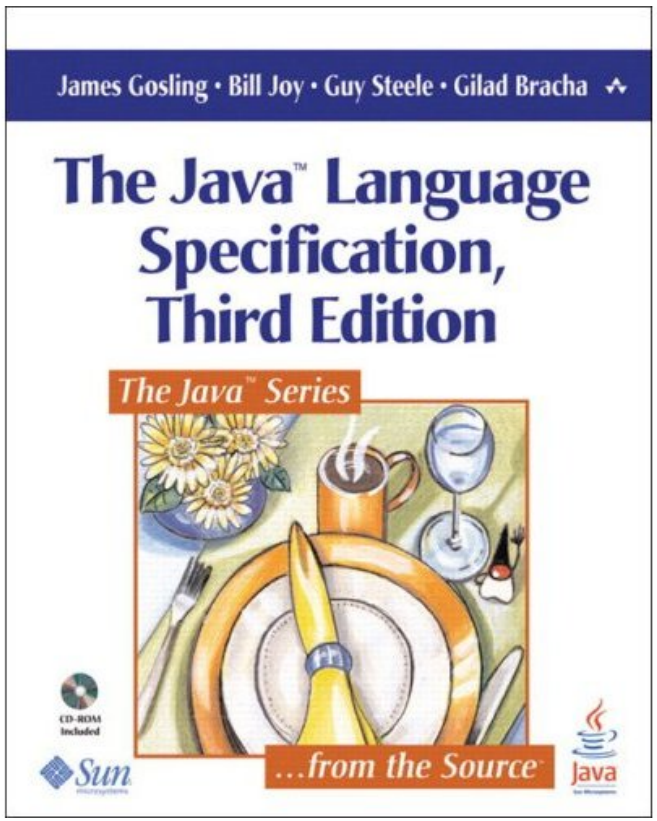
4 Interpreter

Meaning

How to define the meaning of programs in a given language ?

- Informal description most of the time (natural language, ISO, reference book. . .)
- Unprecise, ambiguous.

Informal Semantics



The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

It is recommended that code not rely crucially on this specification.

Formal semantics

The formal semantics mathematically characterises the computations done by a given program:

- useful to design tools (compilers, interpreters).
- mandatory to reason about programs and properties of the language.

Objective of this course

Implementation of program semantics with interpreters.

1 Program Semantics

2 Grammars attributions and semantic actions

3 Useful notions: abstract syntax, AST

4 Interpreter

So far

From our grammars, we only generated **acceptors**.

- We want to execute some action/code each time a grammar rule is matched.

Semantic actions: example

Semantic actions: code executed each time a grammar rule is matched:

Printing as a semantic action in ANTLR

```
s : A s B { System.out.println("rule s"); }//java
```

Right rule : Python/Java/C++, depending on the back-end

Demo time!

Semantic actions in theory - attributes

An attribute is a set “of information” attached to non-terminals/terminals of the grammar

They are usually of two types:

- synthetized: sons $\rightarrow$ father.
- inherited: the converse.

Synthetized grammar attributes

We extend production rules $S \to S_1.S_2$ with arributes r_i, and we write:

$$S\{r\} \to S_1\{r_1\}.S_2\{r_2\}; \{r := f(r_1, r_2)\}$$

with the meaning:

- S recognizes a chain if the begining is recognized with S_1 and the rest by S_2.
- Recognizing a S(resp. S_1, S_2) produces a result r (resp. r_1, r_2)
- The result r is computed from the two results r_1, r_2 by the instruction $r := f(r_1, r_2)$
- All rules that produce a S should have attributes of the same type.

Example of a synthetized attribute

Value of an arithmetic expression, simple grammar:

$E \rightarrow E_1 + E_2 | c$

We define : $value(E) = v$ and $value(c) = v_c$ two attributes of type int for the propagation. Then:

$E\{v\} \rightarrow E_1\{v_1\} + E_2\{v_2\}\ ;\ \{v := v_1 + v_2\}$

$E\{v\} \rightarrow c\{v_c\};\ \{v := v_c\}$

which we can simply write:

```
E-> E1 + E2 {value(E):=value(E1)+value(E2)}
E-> c       {value(E):=value(c)}
```

In practice the value of c is given by the lexer.

Inherited grammar attributes

(left : inherited/right : synthetised) Now

$$\{r\}S\{r'\} \rightarrow \{r'_1 = h(r)\}; \{r_1\}S_1\{r'_1\} \quad ; \quad \{r_2 = g(r, r'_1)\}S_2\{r'_2\}$$
$$; \quad \{r' := f(r, r'_1, r'_2)\}$$

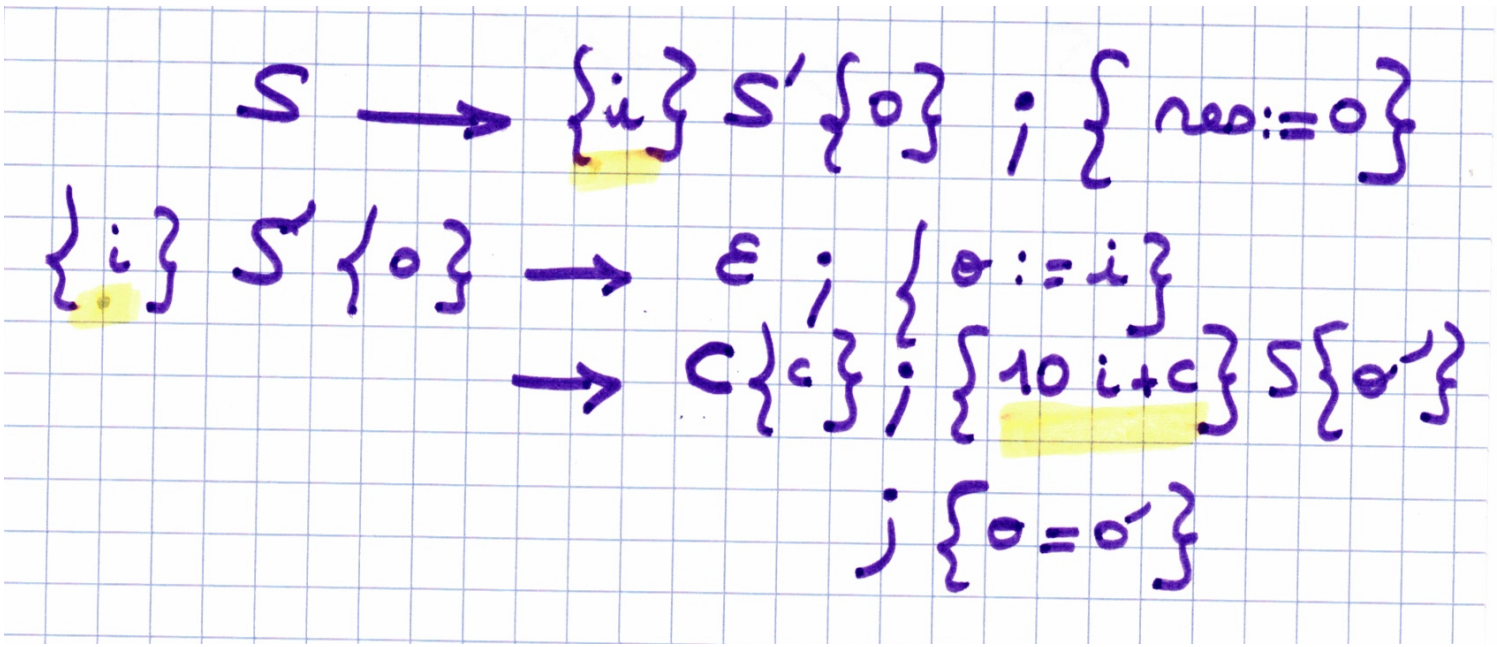
with the meaning:

- S recognizes a chain if the begining is recognized with S_1 and the rest by S_2.
- Recognizing a S_1 produces r'_1 from r_1 st $r'_1 = h(r)$.
- After recognizing $S_1.S_2$, the result r' is computed with $f(r, r'_1, r'2)$.

Example

Consider the grammar: $G = \begin{cases} S \to S' \\ S \to \varepsilon | SC \\ C \to' 0' | '1' | \ldots | '9' \end{cases}$

To compute $eval(\text{"}27\text{"}) = (int)27$ (attribution for C is left as exercice):



An important remark

- Synthetised attributes are easy to implement, thus they exist in most parser generators.
- Inherited attributes are often implemented as global/class variables (see later)

Semantic action in practice - ANTLR

ArithExprParser.g4 - Warning this is java

```
parser grammar ArithExprParser;
options {tokenVocab=ArithExprLexer;}

prog : expr EOF { System.out.println("Result: "+$expr.val); } ;

expr returns [ int val ] : // expr has an integer attribute
   LPAR e=expr RPAR { $val=$e.val; }
| INT { $val=$INT.int; } // implicit attribute for INT
| e1=expr PLUS e2=expr // name sub-parts
   { $val=$e1.val+$e2.val; } // access attributes
| e1=expr MINUS e2=expr { $val=$e1.val-$e2.val; }
;
```

1 Program Semantics

2 Grammars attributions and semantic actions

3 Useful notions: abstract syntax, AST

4 Interpreter

A bit about syntax

The texts:

```
2*(x+1)
```

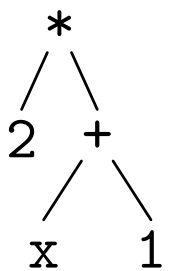
and

```
(2 * ((x) + 1))
```

and

```
2 * (* blablabla *) ( x + 1 )
```

have the same semantics ▶ they should have the **same internal representation**.



Example: syntax of expressions

The (abstract) grammar of arithmetic expressions is (avoiding parenthesis, syntactic sugar ...):

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{addition} \\
 & | & e \times e & \textit{multiplication} \\
 & | & \ldots &
\end{array}
$$

$\mathcal{A}$ ranges over arithmetic expressions: $e \in \mathcal{A}$

- The notion of AST.

Semantics

On the abstract syntax we define a semantics (its meaning):

- The example of numerical expressions
- And programs!

1 Program Semantics

2 Grammars attributions and semantic actions

3 Useful notions: abstract syntax, AST

4 Interpreter
- Interpreter with semantic actions
- Interpreter with explicit AST construction (ENSL Only)
- Interpreter with implicit AST

Definition

From Wikipedia:

> *In computer science, an interpreter is a computer program that* ***directly executes instructions*** *written in a programming or scripting language, without requiring them previously to have been compiled into a machine language program.*

- An **interpreter** executes the input program according to the programming language **semantics**.

Implementation strategies

From Wikipedia:

An interpreter generally uses one of the following strategies for program execution:

1. *Parse the source code and perform its behavior directly;* ▶ *Semantic actions !*
2. *Translate source code into some* ***efficient intermediate representation*** *and immediately execute this;* ▶ *Explicit or implicit* ***Abstract Syntax Tree****.*
3. *(Explicitly execute stored precompiled code made by a compiler which is part of the interpreter system.)*

1. Program Semantics
2. Grammars attributions and semantic actions
3. Useful notions: abstract syntax, AST
4. Interpreter
   - Interpreter with semantic actions
   - Interpreter with explicit AST construction (ENSL Only)
   - Interpreter with implicit AST

How

Use semantic attributes to “evaluate” your input program, by induction on the syntax.

$$(string)\text{”}37 + 5\text{”} \rightarrow \ldots \rightarrow (int)42$$

Recall the example

The evaluation of arithmetical expressions is defined by induction:

ArithExprParser.g4 - Warning this is java

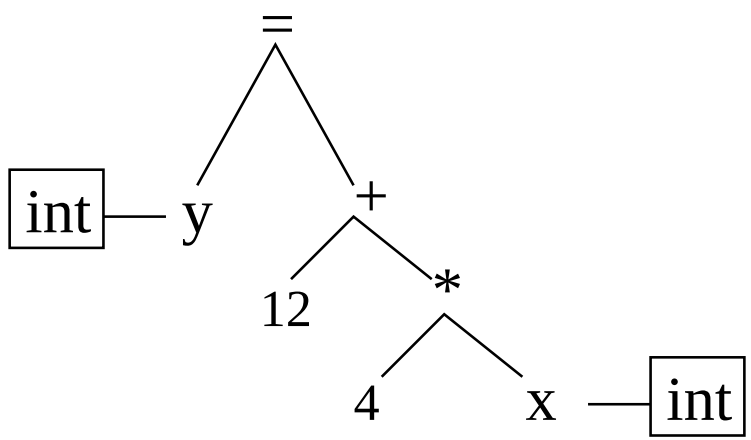
```
parser grammar ArithExprParser;
options {tokenVocab=ArithExprLexer;}

prog : expr EOF { System.out.println("Result: "+$expr.val); } ;

expr returns [ int val ] : // expr has an integer attribute
   LPAR e=expr RPAR { $val=$e.val; }
| INT { $val=$INT.int; } // implicit attribute for INT
| e1=expr PLUS e2=expr // name sub-parts
   { $val=$e1.val+$e2.val; } // access attributes
| e1=expr MINUS e2=expr { $val=$e1.val-$e2.val; }
;
```

Separation of concerns

- The meaning/semantics of the program could be defined in the semantic actions (of the grammar). Usually though:
  - Syntax analyzer only produces the Abstract Syntax Tree.
  - The rest of the compiler directly **works with this AST**.
- Why ?
  - Manipulating a tree (AST) is easy (recursive style);
  - Separate language syntax from language semantics;
  - During later compiler phases, we can assume that the AST is **syntactically correct** $\Rightarrow$ simplifies the rest of the compilation.

1. Program Semantics
2. Grammars attributions and semantic actions
3. Useful notions: abstract syntax, AST
4. Interpreter
   - Interpreter with semantic actions
   - Interpreter with explicit AST construction (ENSL Only)
   - Interpreter with implicit AST

Abstract Syntax Tree



- AST: memory representation of a program;
- Node: a language construct;
- Sub-nodes: parameters of the construct;
- Leaves: usually constants or variables.

Running example : semantics for numerical expressions

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{add} \\
 & | & e \times e & \textit{mult} \\
 & | & \ldots &
\end{array}
$$

Explicit construction of the AST

- Declare a type for the abstract syntax.
- Construct instances of these types during parsing (trees).
- Evaluate with tree traversal.

Example in OCaml 1/3

Types for the abstract syntax:

```
type binop = Add | Mul | ...

type expr_e =
  | Cte of int
  | Var of string
  | Bin of binop * expression * expression
  | ...
```

Example in OCaml 2/3

Pattern matching in parsing rules:

```
%type<Mysyntax.expr_e> expr

  expr:
  INT {Cte (Int64.of_string $1)}
  | LPAREN expr RPAREN { $2 }
  | expr PLUS expr { Bin(Add, $1, $3) }
  | var { Var ($1) }
```

Example in OCaml 3/3

Tree traversal with pattern matching (for expression eval):

```
let rec eval sigma = function
  | Cte(i) -> i
  | Bin(bop,e1,e2) -> let num1= eval sigma e1
                      and num2 = eval sigma e2 in ....
  | Var(s) -> Hashtbl.find sigma s
```

- we need σ, the environnement (map from variables to values).

See the interpreter order, we made a choice !

Example in Java 1/3

AST definition in Java: one class per language construct.

AExpr.java

```
public class APlus extends AExpr {
    AExpr e1,e2;

    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }

}
public class AMinus extends AExpr { ...
```

Example in Java 2/3

The parser builds an AST instance using AST classes defined previously.

ArithExprASTParser.g4

```
parser grammar ArithExprASTParser ;
options {tokenVocab=ArithExprASTLexer;}

prog returns [ AExpr e ] : expr EOF { $e=$expr.e; } ;

// We create an AExpr instead of computing a value
expr returns [ AExpr e ] :
  LPAR x=expr RPAR { $e=$x.e; }
| INT { $e=new AInt($INT.int); }
| e1=expr PLUS e2=expr { $e=new APlus($e1.e,$e2.e); }
| e1=expr MINUS e2=expr { $e=new AMinus($e1.e,$e2.e); }
;
```

Example in Java 3/3

Evaluation is an eval function per class:

AExpr.java

```
public abstract class AExpr {
    abstract int eval(); // need to provide semantics
}
```

APlus.java

```
public class APlus extends AExpr {
    AExpr e1,e2;
    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }
    // semantics below
    int eval() { return (e1.eval()+e2.eval()); }
}
```

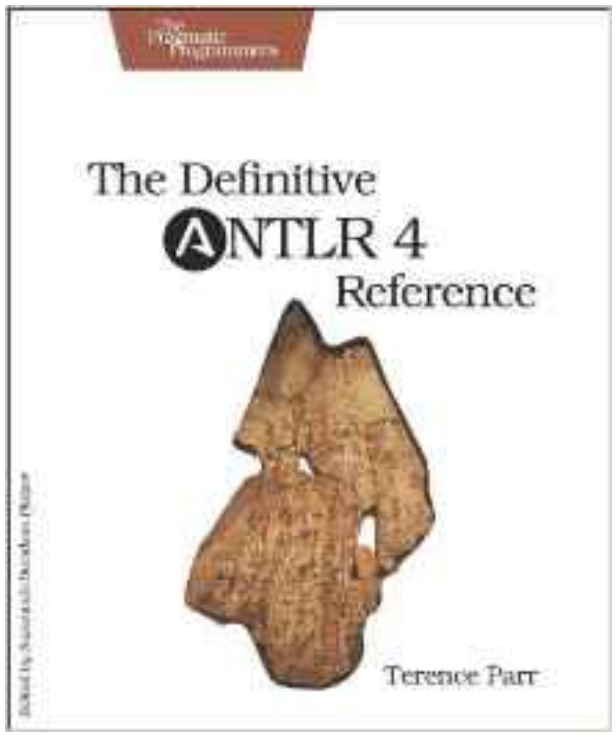
1. Program Semantics
2. Grammars attributions and semantic actions
3. Useful notions: abstract syntax, AST
4. Interpreter
   - Interpreter with semantic actions
   - Interpreter with explicit AST construction (ENSL Only)
   - Interpreter with implicit AST

Principle - OO programming

> *The visitor design pattern is a way of separating an algorithm from an object structure on which it operates.[...] In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function.*

https://en.wikipedia.org/wiki/Visitor_pattern

Application



Designing interpreters / tree traversal in ANTLR-Python

- The ANTLR compiler generates a Visitor class.
- We override this class to traverse the parsed instance.

Arit Example with ANTLR/Python 1/3

AritParser.g4

```
expr:
     expr mdop=(MULT | DIV) expr #multiplicationExpr
   | expr pmop=(PLUS | MINUS) expr #additiveExpr
   | atom #atomExpr
   ;

atom
   : INT #int
   | ID #id
   | '(' expr ')' #parens
```

- compilation with `-Dlanguage=Python3 -visitor`

Arit Example with ANTLR/Python 2/3 -generated file

AritVisitor.py (generated)

```
class AritVisitor(ParseTreeVisitor):
...
    # Visit a parse tree produced by AritParser#multiplicationExpr.
    def visitMultiplicationExpr(self, ctx):
        return self.visitChildren(ctx)


    # Visit a parse tree produced by AritParser#atomExpr.
    def visitAtomExpr(self, ctx):
        return self.visitChildren(ctx)
..
```

Arit Example with ANTLR/Python 3/3

Visitor class overriding to write the interpreter:

MyAritVisitor.py

```
class MyAritVisitor(AritVisitor):

    def visitInt(self, ctx):
        return int(ctx.getText())

    def visitMultiplicationExpr(self, ctx):
        leftval = self.visit(ctx.expr(0))
        rightval = self.visit(ctx.expr(1))
        if ctx.mdop.type == AritParser.MULT:
            return leftval * rightval
        else:
            return leftval / rightval

    def visitAdditiveExpr(self, ctx):
```

Arit Example with ANTLR/Python - Main

And now we have a full interpret for arithmetical expressions!

arit.py (Main)

```
lexer = AritLexer(InputStream(sys.stdin.read()))
stream = CommonTokenStream(lexer)
parser = AritParser(stream)
tree = parser.prog()
print("I'm here : nothing has been done")

visitor = MyAritVisitor()
visitor.visit(tree)
```

Bilan

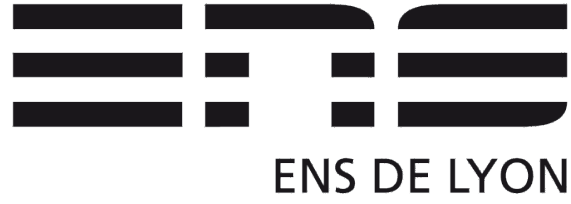
1. Program Semantics
2. Grammars attributions and semantic actions
3. Useful notions: abstract syntax, AST
4. Interpreter
   - Interpreter with semantic actions
   - Interpreter with explicit AST construction (ENSL Only)
   - Interpreter with implicit AST

Compilation and Program Analysis (#4) : Types, and Typing MiniWhile

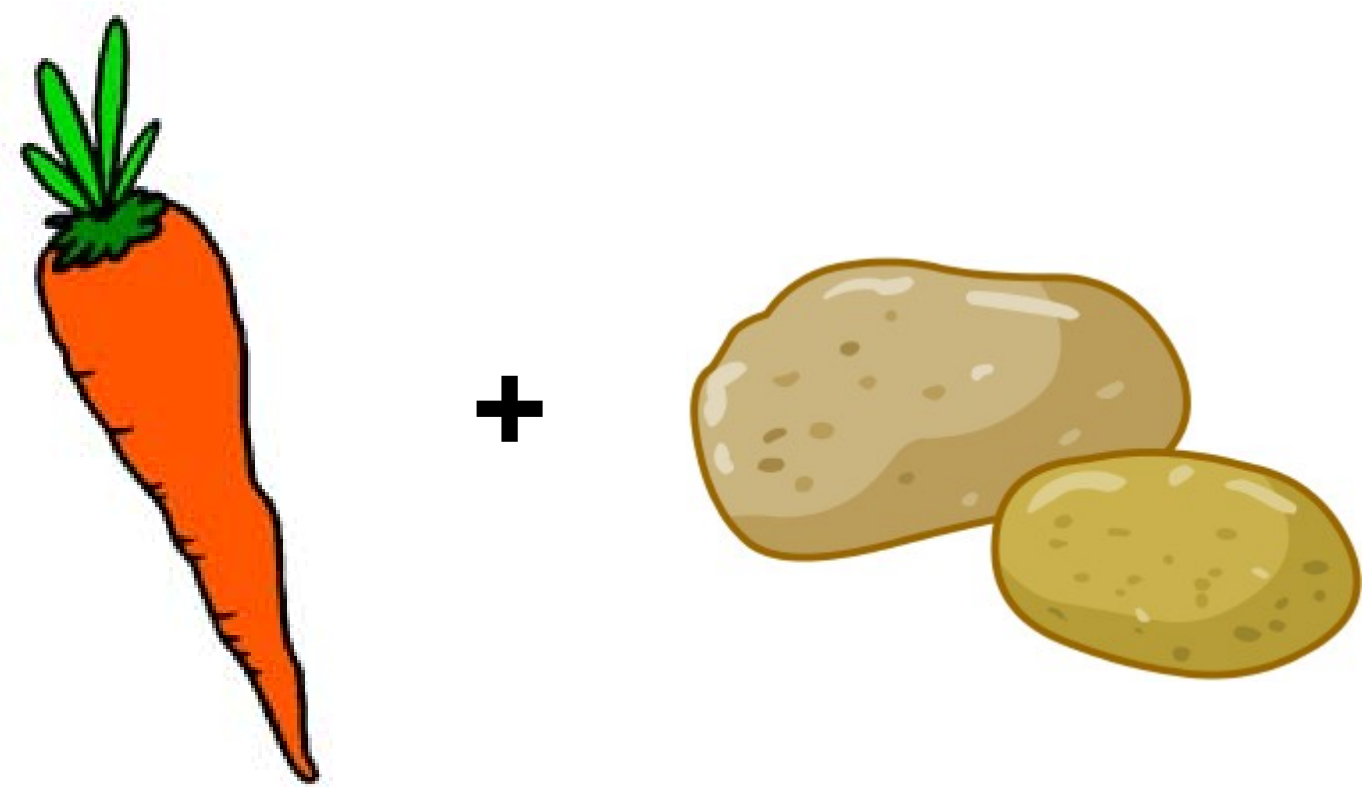
Laure Gonnord & Matthieu Moy & other
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

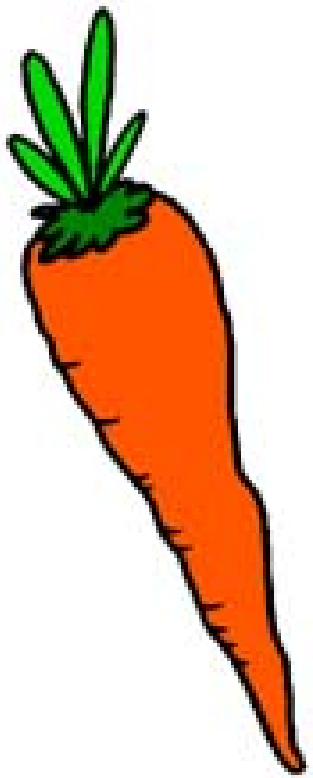
2020-2021



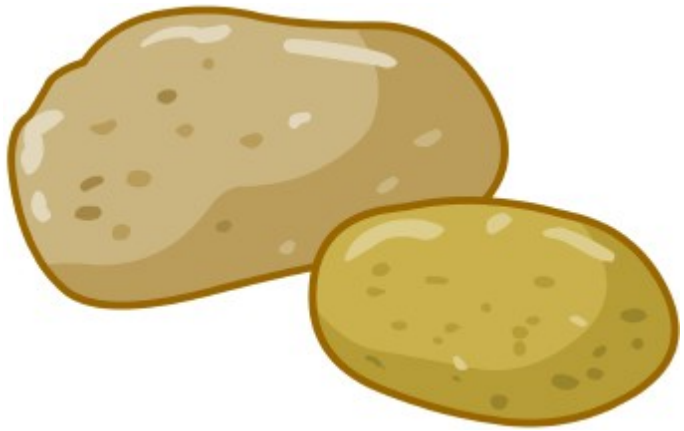
Typing



Typing



/



Typing

If you write: `"5" + 37`

what do you want to obtain

- a compilation error? (OCaml)
- an exec error? (Python)
- the int 42? (Visual Basic, PHP)
- the string `"537"`? (Java)
- anything else?

and what about `37 / "5"` ?

Typing

When is

```
e1 + e2
```

legal, and what are the semantic actions to perform ?

- Typing: an analysis that gives a type to each subexpression, and reject incoherent programs.

When

- Dynamic typing (during exec): Lisp, PHP, Python
- Static typing (at compile time): C, Java, OCaml

▶ Here: the second one.

Slogan

well typed programs do not go wrong

1 Generalities about typing

2 Typing ML - ENSL Only

3 Imperative languages (C, Mini-While)

Typing objectives

- Should be **decidable**.
- It should reject programs like `(1 2)` in OCaml, or `1+"toto"` in C before an actual error in the evaluation of the expression: this is **safety**.

  The type system is related to the kind of error to be detected: **operations on basic types** / method invocation (message not understood) / correct synchronisation (e.g. session types) in concurrent programs / ...
- The type system should be expressive enough and not reject too many programs. (**expressivity**)

Principle

All sub-expressions of the program must be given a type

$$\texttt{fun}\ (x : \texttt{int}) \rightarrow \texttt{let}\ (y : \texttt{int}) = (+ :)(((x : \texttt{int}), (1 : \texttt{int})) : \texttt{int} \times \texttt{int})\ \texttt{in}$$

What does the programmer write?

- The type of all sub-expressions (like above) easy to verify, but tedious for the programmer
- Annotate only variable declarations (Pascal, C, Java, . . .)

$$\texttt{fun}\ (x : \texttt{int}) \rightarrow \texttt{let}\ (y : \texttt{int}) = +(x, 1)\ \texttt{in}\ y$$

- Only annotate function parameters

$$\texttt{fun}\ (x : \texttt{int}) \rightarrow \texttt{let}\ y = +(x, 1)\ \texttt{in}\ y$$

- Annotate nothing: complete inference : Ocaml, Haskell, . . .

Properties

- correction: “yes” implies the program is well typed.
- completeness: the converse.

(optional)

- principality : The most general type is computed.

What is a good output for a type-checker?

- We do not want:

```
failwith "typing error"
```

  the origin of the problem should be clearly stated
- We keep the types for next phases.

In practice

- Input: Trees are decorated by source code lines.
- Output: Trees are decorated by types.

1 Generalities about typing

2 Typing ML - ENSL Only

3 Imperative languages (C, Mini-While)

1. Generalities about typing
2. Typing ML - ENSL Only
3. Imperative languages (C, Mini-While)
   - Simple Type Checking for mini-while
   - A bit of implementation (for expr)
   - Safety ENS Only

1. Generalities about typing
2. Typing ML - ENSL Only
3. Imperative languages (C, Mini-While)
   - Simple Type Checking for mini-while
   - A bit of implementation (for expr)
   - Safety ENS Only

Mini-While Syntax

Expressions:

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{addition} \\
 & | & e \times e & \textit{multiplication} \\
 & | & \ldots &
\end{array}
$$

Mini-while:

$$
\begin{array}{rcll}
S(Smt) & ::= & x := expr & \text{assign} \\
 & | & skip & \text{do nothing} \\
 & | & S_1 ; S_2 & \text{sequence} \\
 & | & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 & \text{test} \\
 & | & \texttt{while } b \texttt{ do } S \texttt{ done} & \text{loop}
\end{array}
$$

Typing rules for expr

Here types are basic types: Int|Bool

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{n \in \mathbb{Z}}{\Gamma \vdash n : \texttt{int}} \quad \text{(or } \mathrm{tt}\text{: bool, } \ldots\text{)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

Typing rules for statements: $\Gamma \vdash S$

A statement S is well-typed (there is no type for statements)

on board!

Typing While : recap

$$\frac{c \in \mathbb{Z}}{\Gamma \vdash c : \texttt{int}} \qquad \frac{\Gamma(x) = t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}}$$

$$\frac{\Gamma \vdash S_1 : \texttt{void} \;\; \Gamma \vdash S_2 : \texttt{void}}{\Gamma \vdash S_1; S_2 : \texttt{void}} \qquad \frac{\Gamma \vdash e : t \quad \Gamma \vdash x : t \quad t \in \{\texttt{int}, \texttt{bool}\}}{\Gamma \vdash x = e : \texttt{void}}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S : \texttt{void}}{\Gamma \vdash \texttt{while}\ b\ \texttt{do}\ S\ \texttt{done} : \texttt{void}}$$

$$\frac{\Gamma \vdash b : \texttt{bool} \quad \Gamma \vdash S_1 : \texttt{void} \quad \Gamma \vdash S_2 : \texttt{void}}{\Gamma \vdash \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 : \texttt{void}}$$

Typing: an example

Considering $\Gamma = \{x_1 \mapsto int\}$, prove that the given sequence of instructions is well typed:

```
x1 = 3 ;
x1 = x1+9 ;
```

on board!

Hybrid expressions

What if we have `1.2 + 42` ?

- reject?
- compute a float!

▶ This is **type coercion**. We will see how to implement it during a lab.

More complex expressions

What if we have types `pointer of bool` or `array of int`? We might want to check equivalence (for addition ...).

- This is called **structural equivalence** (see Dragon Book, "type equivalence"). This is solved by a basic graph traversal checking that each element are equivalent/compatible.

Sub-typing ENSL Only

- A type can be more precise than another one, e.g.

$$int <: num$$

- Need additional rule to use sub-typing:

$$\frac{e : \tau \qquad \tau <: \tau'}{e : \tau'}$$

- Sometimes, rule to compose sub-types, e.g. functions or parametric types

$$\frac{e : Array[\tau] \qquad \tau <: \tau'}{e : Array[\tau']}$$

How to define subtyping for functions?

Note: subtyping is heavily used in OOP

1. Generalities about typing
2. Typing ML - ENSL Only
3. Imperative languages (C, Mini-While)
   - Simple Type Checking for mini-while
   - A bit of implementation (for expr)
   - Safety ENS Only

Principle

- Gamma is constructed with lexing information or parsing (variable declaration with types).
- Rules are semantic actions. The semantic actions are responsible for the evaluation order, as well as typing errors.

Type Checking V1 : Visitor

MuTypingVisitor.py

```
# now visit expr

def visitAtomExpr(self, ctx):
    return self.visit(ctx.atom())

def visitOrExpr(self, ctx):
    lvaltype = self.visit(ctx.expr(0))
    rvaltype = self.visit(ctx.expr(1))
    if (BaseType.Boolean == lvaltype) and (BaseType.Boolean == rvaltype):
        return BaseType.Boolean
    else:
        self._raise(ctx, 'boolean operands', lvaltype, rvaltype)
```

In practice for mini-C (lab sessions)

No annotation is added to the AST (everything is int or bool, no ambiguity)

We can create associating type to variables, directly from parsing

1. Generalities about typing
2. Typing ML - ENSL Only
3. Imperative languages (C, Mini-While)
   - Simple Type Checking for mini-while
   - A bit of implementation (for expr)
   - Safety ENS Only

Summary

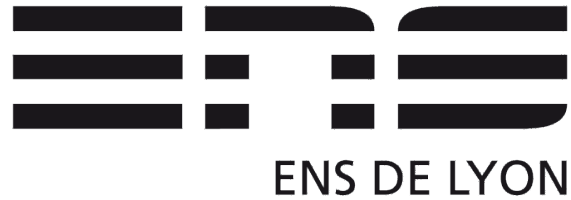
1. Generalities about typing
2. Typing ML - ENSL Only
3. Imperative languages (C, Mini-While)
   - Simple Type Checking for mini-while
   - A bit of implementation (for expr)
   - Safety ENS Only

Compilation (#5) : Syntax-Directed Code Generation

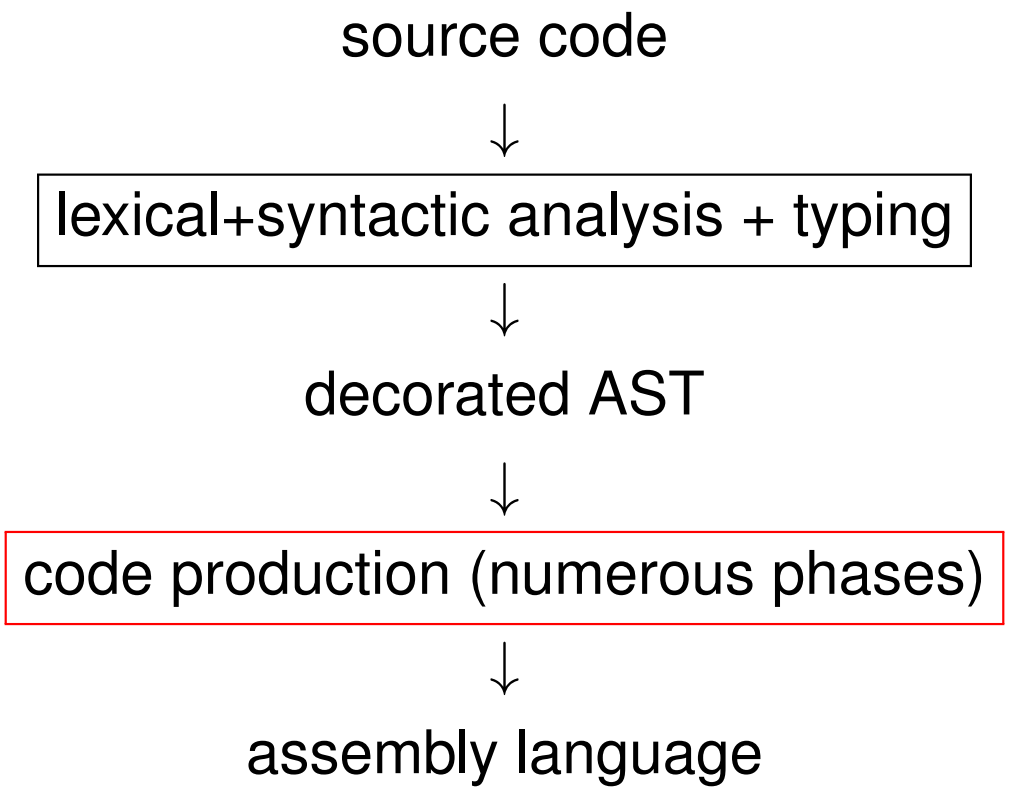
Laure Gonnord & Matthieu Moy & other
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2020-2021



Big picture


source code
lexical+syntactic analysis + typing
decorated AST
code production (numerous phases)
assembly language

Rules of the Game here

For this code generation:

- Still no functions and no non-basic types. (mini-while)
- Syntax-directed: one grammar rule $\rightarrow$ a set of instructions.
  - Code redundancy.
- No register reuse: everything will be stored on the stack.

The Target Machine : RISCV (course #1)

1 3-address syntax-directed Code Generation
- Rules

2 Memory allocation

3 LAB: Direct Code Generation

4 Conclusion

A first example (1/2)

How do we translate:

```
int x, y;
x=4;
y=12+x;
```

- Variable decl's visitor gives a place to each variable: $x \mapsto place0$, $y \mapsto place1$.
- Compute 4, store somewhere, then copy in x's place.
- Compute $12 + x$: 12 in `place2`, copy the value of x in `place3`, then add, store in `place4`, then copy into $y's$ place.

▶ the code generator will use a place generator called `new_tmp()`

A first example: 3@code (2/2)

"Compute 4 and store in x (temp0)":

```
li temp2, 4
mv temp0, temp2
```

Objective

3-address RISCV **Code Generation** for the Mini-While language:

- All variables are int/bool.
- All variables are global.
- No functions

with syntax-directed translation. Implementation in Lab (MiniC)

- This is called **three-adress code generation**

1. 3-address syntax-directed Code Generation
   - Rules
2. Memory allocation
3. LAB: Direct Code Generation
4. Conclusion

Code generation utility functions

We will use:

- A new (fresh) temporary can be created with a `newtemp()` function.
- A new fresh label can be created with a `new_label()` function.
- The generated instructions are close to the RISCV ones.

Abstract Syntax

Expressions:

e	$::=$	c	constant
	$\mid$	x	variable
	$\mid$	$e + e$	addition
	$\mid$	e or e	boolean or
	$\mid$	$e < e$	less than
	$\mid$	...	

and statements:

$S(Smt)$	$::=$	$x := expr$	assign
	$\mid$	$skip$	do nothing
	$\mid$	$S_1; S_2$	sequence
	$\mid$	`if` b `then` S_1 `else` S_2	test
	$\mid$	`while` b `do` S `done`	loop

Code generation for expressions, example

e ::= c (cte expr)	dr <-new_tmp() code.add(InstructionLI(dr, c)) return dr

- this rule gives a way to generate code for any constant.

Code generation for a boolean expression, example

e ::= $e_1 < e_2$	

```
dr <-new_tmp()
t1 <- GenCodeExpr(e1)
t2 <- GenCodeExpr(e2)
endrel <- new_label()
code.add(InstructionLI(dr, 0))
#if t1>=t2 jump to endrel
code.add(InstructionCondJUMP(endrel, t1, ">=" , t2)
code.add(InstructionLI(dr, 1))
code.addLabel(endrel)
return dr
```

- integer value 0 or 1.

Second example: a boolean test

Let us generate the code for $x < 4$:

```
li temp3, 4      // get 4
li  temp2, 0
geq temp0, temp3, lbl0  # >= comp + jump
li  temp2, 1
lbl0:
```

(temporary values on board)

Code generation for commands, example

`if` b `then` $S1$ `else` $S2$	`lelse,lendif <-new_labels()`
	`t1 <- GenCodeExpr(b)`
	`#if the condition is false, jump to else`
	`code.add(InstructionCondJUMP(lelse, t1, "=", 0))`
	`GenCodeSmt(S1) #then`
	`code.add(InstructionJUMP(lendif))`
	`code.addLabel(lelse)`
	`GenCodeSmt(S2) #else`
	`code.addLabel(lendif)`

Example for tests.

Let us generate the code for `if x<4 then y=7 else ...`

```
## preceding code
beq tmp2, zero, lelse1 # if false, jump
li temp4, 7
mv temp1, temp4    # y gets 7
jump lendif1
lelse1:
  # code for else branch
lendif1:
```

1 3-address syntax-directed Code Generation

2 Memory allocation

3 LAB: Direct Code Generation

4 Conclusion

From 3@ code to valid RISCV

3@code is not valid RISCV code !

3 “kinds of allocation”:

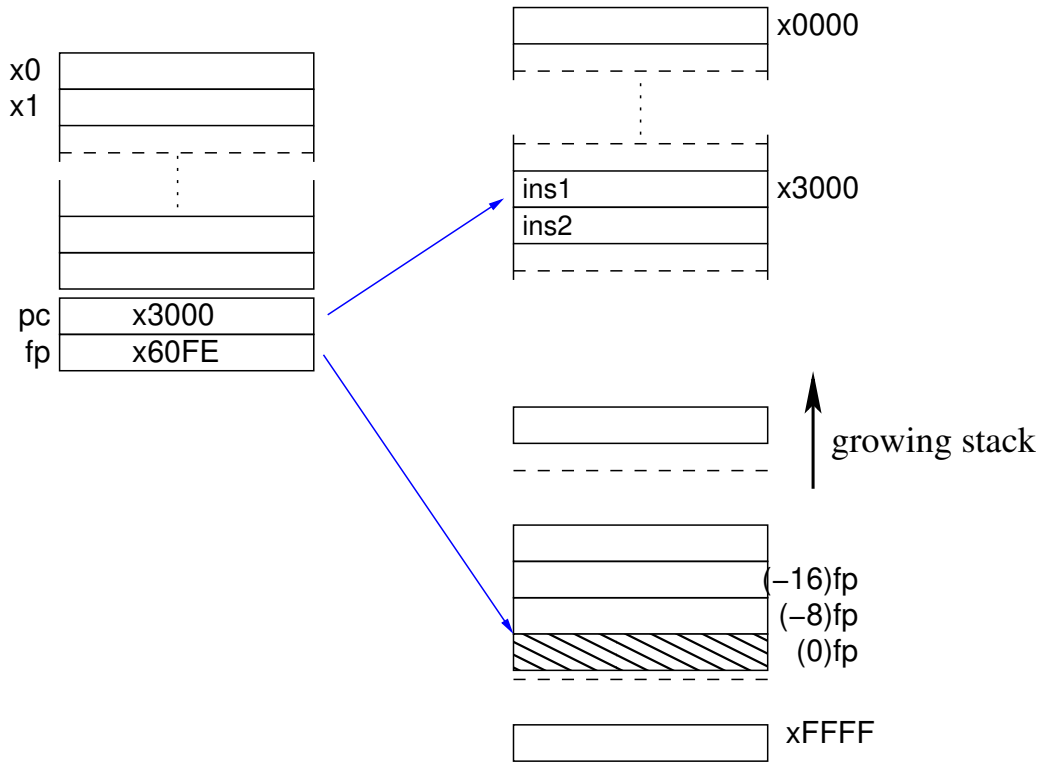
- All in registers (but ?) $place_i \rightarrow register$
- All in memory (here!) $place_i \rightarrow memory$
- Something in the middle (later!)

A stack, why ?

- Store constants, strings, . . .
- Provide an easy way to communicate arguments values (see later)
- Give place to store intermediate values (here)

Stack with RISCV

- There is a special register `fp`.
- Store and loads from `fp`



Nice picture by N. Louvet - adapted in 2019

How to store into the stack

Store (the content of) s_3 on the stack at offset $offset$!:

```
sd s3, -offset*8(fp)  # (Instru3A('sd', s3, Offset(
    FP, -offset*8)))
     # "write the value of s3 at address fp - offset
    *8"
```

1 3-address syntax-directed Code Generation

2 Memory allocation

3 LAB: Direct Code Generation

4 Conclusion

Code Generation

Input: a MiniC file:

```
int main(){
int n;
n=6;
return 0;}
```

Output: a RISCV file:

```
  [...]
        ;; (stat (assignment n = (expr (atom 6)) ;))
3       li t1, 6        ; t1 is a riscv register.
        mv t2, r1
  [...]
```

Steps

- 3-address code generation according to the code generation rules.
- Simple register/memory allocation + pretty print.

Details in the dedicated video/slides.

1 3-address syntax-directed Code Generation

2 Memory allocation

3 LAB: Direct Code Generation

4 Conclusion

Drawbacks of the former translation

Drawbacks:

- redundancies (constants recomputations, . . .)
- memory intensive loads and stores.

▶ we need a more efficient data structure to reason on: **the control flow graph (CFG)**. (see next course)

Summary : 3adress code generation

1. 3-address syntax-directed Code Generation
   - Rules
2. Memory allocation
3. LAB: Direct Code Generation
4. Conclusion

Exercice: 3 address code generation for

```
i = 0;
if (i == 10) {
    i = i + 1;
} else {
    i = i - 1;
}
```

Exercice: naive allocation (all in registers)

```
li temp_0, 42
li temp_1, 1
add temp_2, temp_1, temp_0
```

Exercice: “all in mem” allocation

```
li temp_0, 42
li temp_1, 1
add temp_2, temp_1, temp_0
```

Empty slide for drawing (1)

Empty slide for drawing (2)

Empty slide for drawing (3)

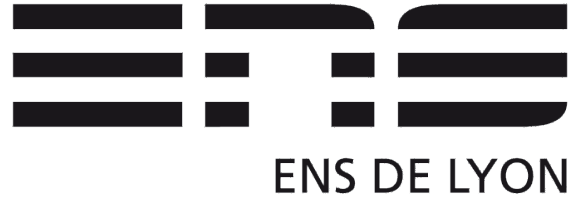
Empty slide for drawing (4)

Compilation (#6) : Intermediate Representations: CFG, DAGs (Instruction Selection and Scheduling), SSA

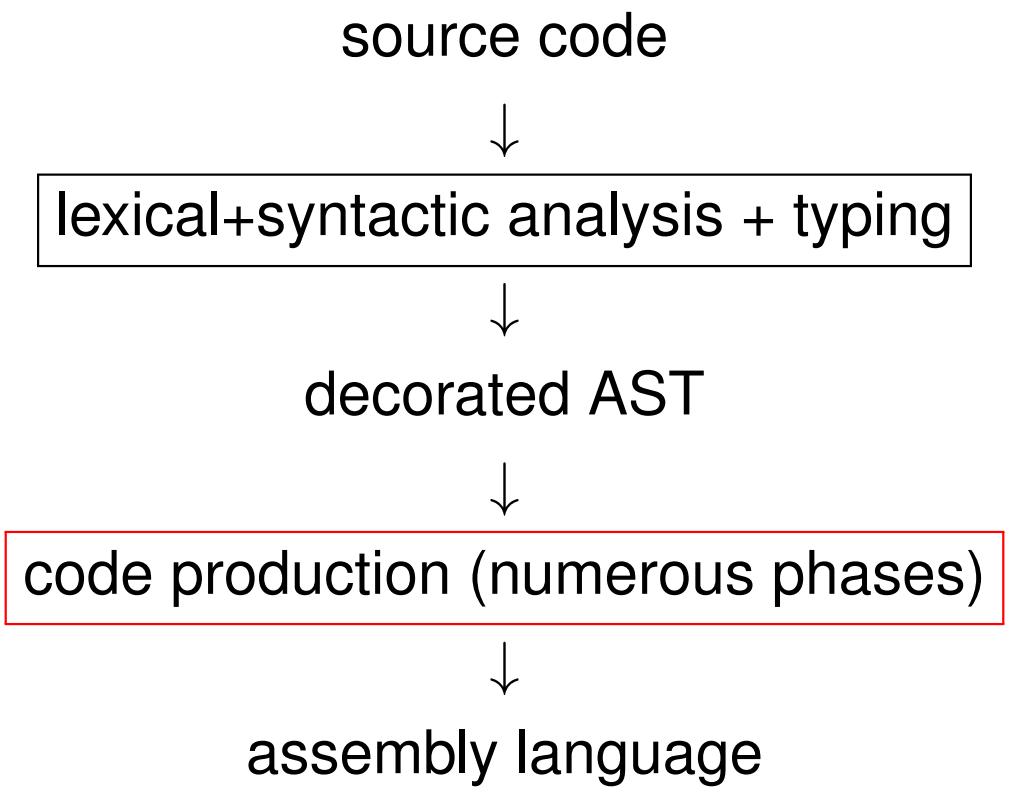
Laure Gonnord & Matthieu Moy & other
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2020-2021



Big picture


source code
lexical+syntactic analysis + typing
decorated AST
code production (numerous phases)
assembly language

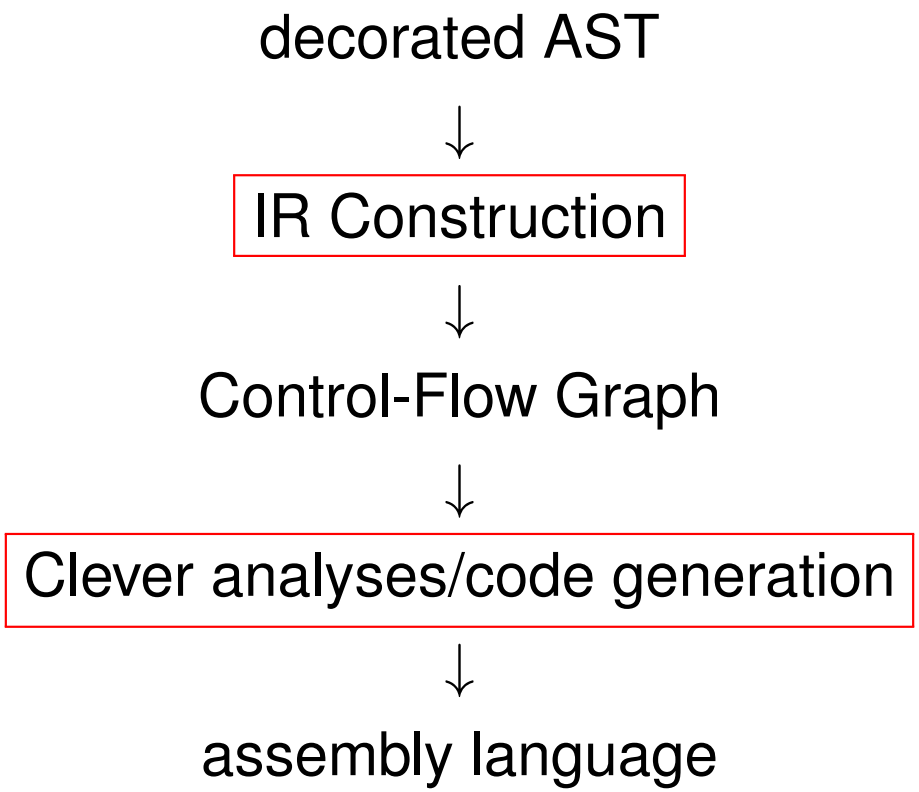
In context 1/2

In the last course we saw the need for a better data structure to propagate and infer information. We need:

- A data structure that helps us to reason about the flow of the program.
- Which embeds our three address code.

- Control-Flow Graph.

In context 2/2



1 Control flow Graph

2 Basic Bloc DAGs, instruction selection/scheduling

Definitions

Definition (Basic Block)

Basic block: largest (3-address RISCV*) instruction sequence without label. (except at the first instruction) and without jumps and calls.*

Definition (CFG)

It is a directed graph whose vertices are basic blocks, and edge $B_1 \to B_2$ *exists if* B_2 *can follow immediately* B_1 *in an execution.*

- two optimisation levels: local (BB) and global (CFG)

An example 1/2

Let us consider the program:

```
int x,y;
if (x<4) y=7; else y=42;
x=10;
```

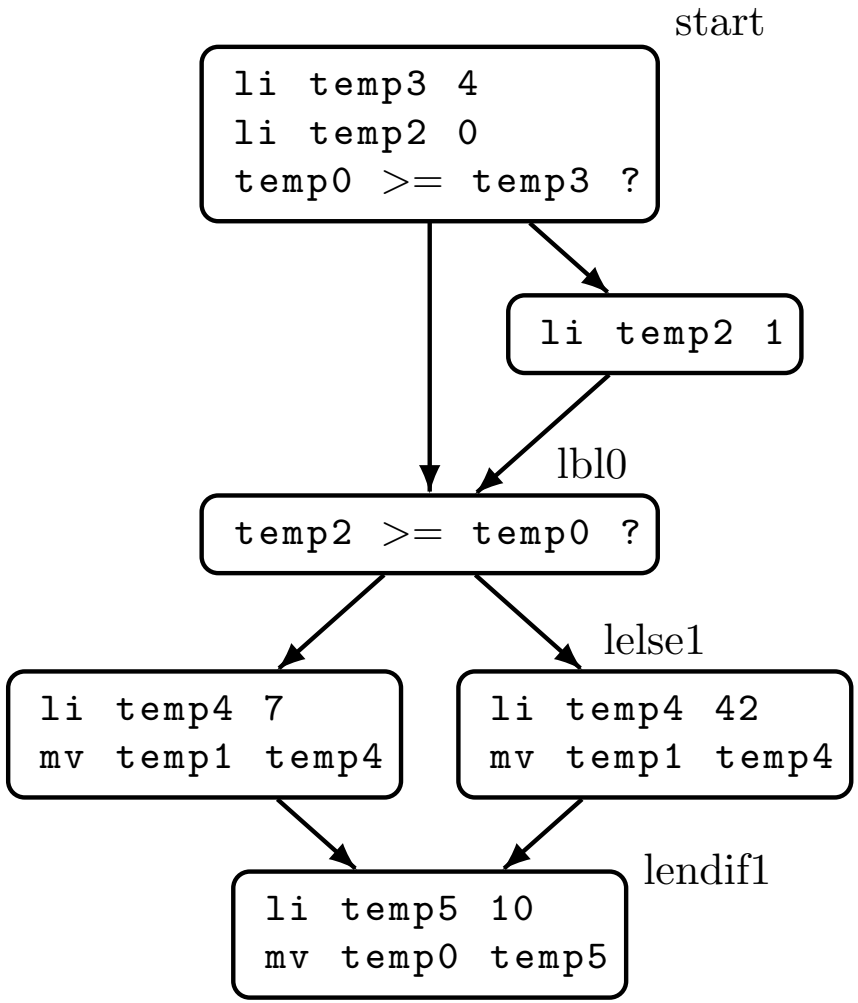
We already generated the (linear code) for a large part of it.

An example 2/2

```
  li temp3, 4
  li temp2, 0
  bge temp0, temp3, lbl0
  li temp2, 1
lbl0:  # if false, jump (skip the 'then')
  bge temp2, temp0, lelse1
  li temp4, 7
  mv temp1, temp4  # y gets 7
  jump lendif1
lelse1:
  li temp4 42
  mv temp1, temp4  # y gets 42
lendif1:
  li temp5, 10
  mv temp0, temp5  # end
```

An example 2/2

```
  li temp3, 4
  li temp2, 0
  bge temp0, temp3, lbl0
  li temp2, 1
lbl0:  # if false, jump (skip the 'then')
  bge temp2, temp0, lelse1
  li temp4, 7
  mv temp1, temp4  # y gets 7
  jump lendif1
lelse1:
  li temp4 42
  mv temp1, temp4  # y gets 42
lendif1:
  li temp5, 10
  mv temp0, temp5  # end
```



Identifying Basic Blocks (from 3 address code)

- The first instruction of a basic block is called a **leader**.
- We can identify leaders via these three properties:
  1. The first instruction in the intermediate code is a leader.
  2. Any instruction that is the target of a conditional or unconditional jump is a leader.
  3. Any instruction that immediately follows a conditional or unconditional jump is a leader.
- Once we have found the leaders, it is straighforward to find the basic blocks: for each leader, its basic block consists of the leader itself, plus all the instructions until the next leader.

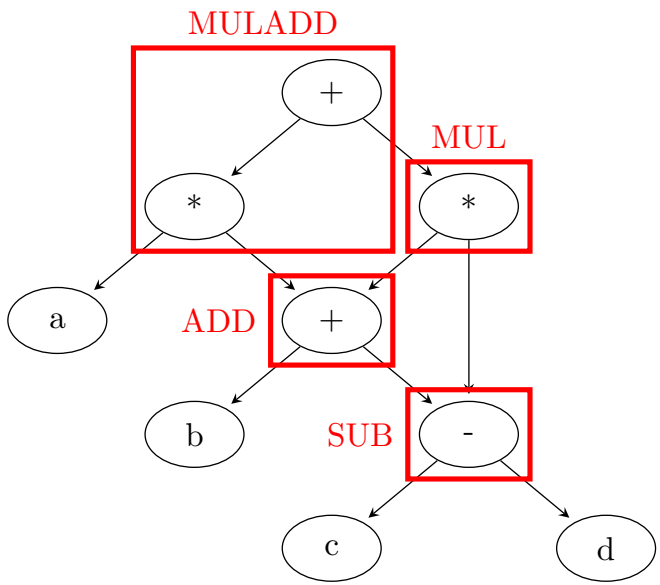
1 Control flow Graph

2 Basic Bloc DAGs, instruction selection/scheduling
- Basic Blocks DAG Construction
- Instruction Selection
- Instruction Scheduling

Big picture (Basic Block Optimisation)

- Front-end $\rightarrow$ a CFG where nodes are basic blocks.
- Basic blocks $\rightarrow$ DAGs that explicit common computations

```
u1 := c - d
u2 := b + u1
u3 := a * u2
u4 := u2 * u1
u5 := u3 + u4
```



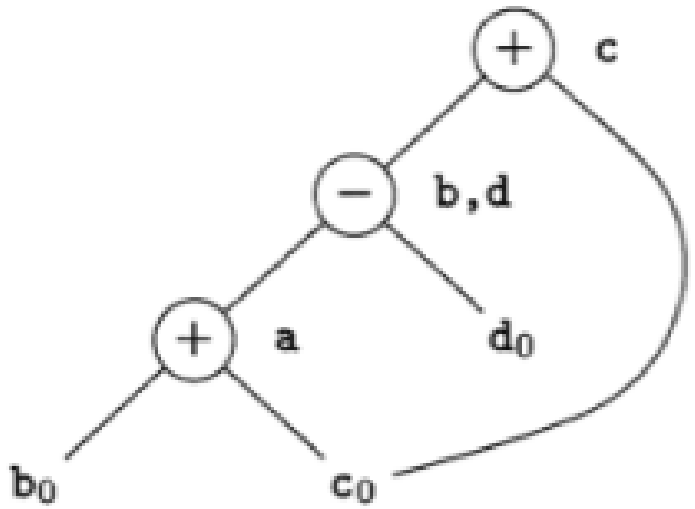
- choose instructions(**selection**) and order them (**scheduling**).

1 Control flow Graph

2 Basic Bloc DAGs, instruction selection/scheduling
- Basic Blocks DAG Construction
- Instruction Selection
- Instruction Scheduling

An Example of BB DAG construction

```
a = b + c
b = a - d
c = b + c
d = a - d
```



Useful links :

https://www.youtube.com/watch?v=PXTKWvyQUwE and https://www.cse.iitm.ac.in/~krishna/cs3300/pm-lecture3.pdf for other BB optimisations.

1 Control flow Graph

2 Basic Bloc DAGs, instruction selection/scheduling
- Basic Blocks DAG Construction
- Instruction Selection
- Instruction Scheduling

Instruction Selection, in general

The problem:

- a list of instructions/operations that compute one or more expressions.
- map these operations in “real machine instructions”.
- at minimum cost.

Instruction Selection

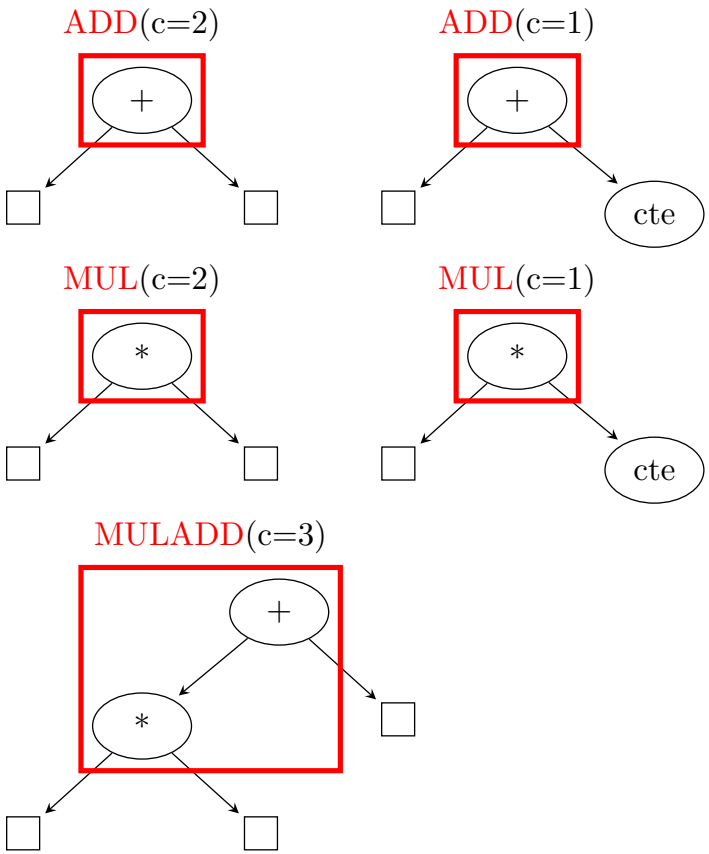
The problem of selecting instructions is a DAG-partitioning problem. But what is the objective ?

The best instructions:

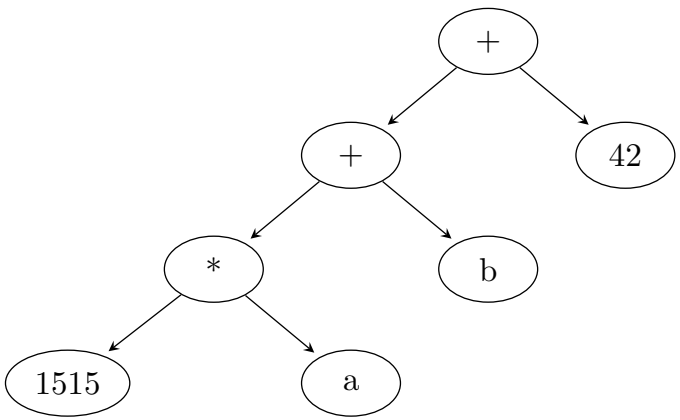
- cover bigger parts of computation.
- cause few memory accesses.

▶ Assign a cost to each instruction, depending on their addressing mode.

Instruction Selection: an example



What is the optimal instruction selection for:



- Finding a tiling of minimal cost: it is **NP-complete** (SAT reduction).

Tiling trees / DAGs, in practice

For tiling:

- There is an optimal algorithm for **trees** based on dynamic programming.
- For DAGs we use heuristics (decomposition into a forest of trees, . . .)

- The literature is plethoric on the subject.

Instruction Selection, in our compiler

Mapping one to one. No real choice.

1 Control flow Graph

2 Basic Bloc DAGs, instruction selection/scheduling
- Basic Blocks DAG Construction
- Instruction Selection
- Instruction Scheduling

Instruction Scheduling, in general

The problem:

- change the order of instructions.
- to “optimise’.
- without “cutting dependencies”.

Instruction Scheduling, what for?

We want an evaluation order for the instructions that we choose with **Instruction Scheduling**.

A scheduling is a function θ that associates a **logical date** to each instruction. To be correct, it must respect data dependancies:

```
(S1) u1 := c - d
(S2) u2 := b + u1
```

implies $\theta(S_1) < \theta(S_2)$.

- How to choose among many correct schedulings? depends on the target architecture.

Architecture-dependant choices

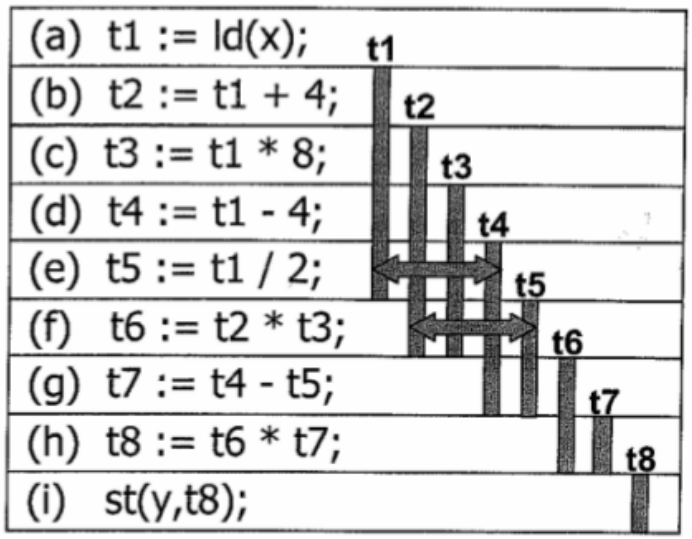
The idea is to exploit the different ressources of the machine at their best:

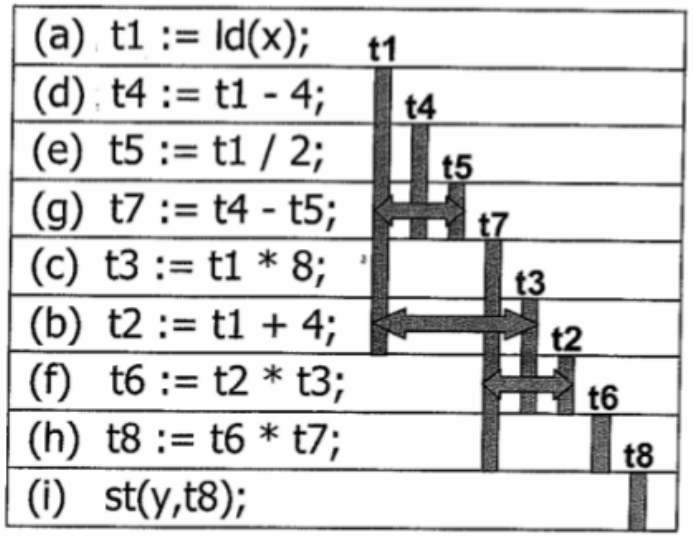
- instruction parallelism: some machine have parallel units (subinstructions of a given instruction).
- prefetch: some machines have non-blocking load/stores, we can run some instructions between a load and its use (hide latency!)
- pipeline.
- registers: see next slide.

(sometimes these criteria are incompatible)

Register use

Some schedules induce less **register pressure**:





- How to find a schedule with less register pressure?

Scheduling wrt register pressure ENSL Only

Result: this is a linear problem on trees, but NP-complete on DAGs (Sethi, 1975).

- Sethi-Ullman algorithm on trees, heuristics on DAGs

Sethi-Ullman algorithm on trees ENSL Only

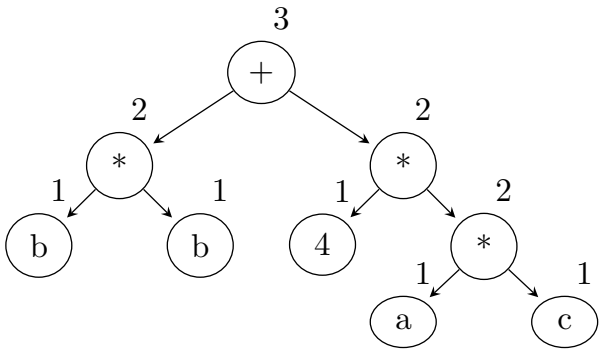
$\rho(node)$ denoting the number of (pseudo)-registers necessary to compute a node:

- $\rho(leaf) = 1$
- $\rho(nodeop(e_1, e_2)) = \begin{cases} max\{\rho(e_1), \rho(e_2)\} & \text{if } \rho(e_1) \neq \rho(e_2) \\ \rho(e_1) + 1 & \text{else} \end{cases}$

(the idea for non "balanced" subtrees is to execute the one with the biggest ρ first, then the other branch, then the op. If the tree is balanced, then we need an extra register)

▶ then the code is produced with postfix tree traversal, the biggest register consumers first.

Sethi-Ullman algorithm on trees - an example



	tmp_1	tmp_2	tmp_3	tmp_4
`mul tmp1, b, b`				
`mul tmp2, a, c`	■			
`leti tmp3, 4`	■	■		
`mul tmp4, tmp2, tmp3`	■	■	■	
`add tmp5, tmp1, temp4`	■			■

Instruction Selection, in our compiler

Same order as the 3-address code.

Conclusion (instruction selection/scheduling)

Plenty of other algorithms in the literature:

- Scheduling DAGs with heuristics, ...
- Scheduling loops (M2IF course on advanced compilation)

Practical session:

- we have (nearly) no choice for the instructions in the RISCV ISA.
- evaluating the impact of scheduling is a bit hard.

We won't implement any of the previous algorithms.

Summary

1. Control flow Graph

2. Basic Bloc DAGs, instruction selection/scheduling
   - Basic Blocks DAG Construction
   - Instruction Selection
   - Instruction Scheduling

Compilation (#7):

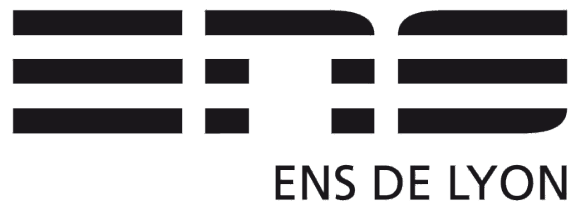
Register Allocation + Data Flow Analyses

Laure Gonnord & Matthieu Moy & other

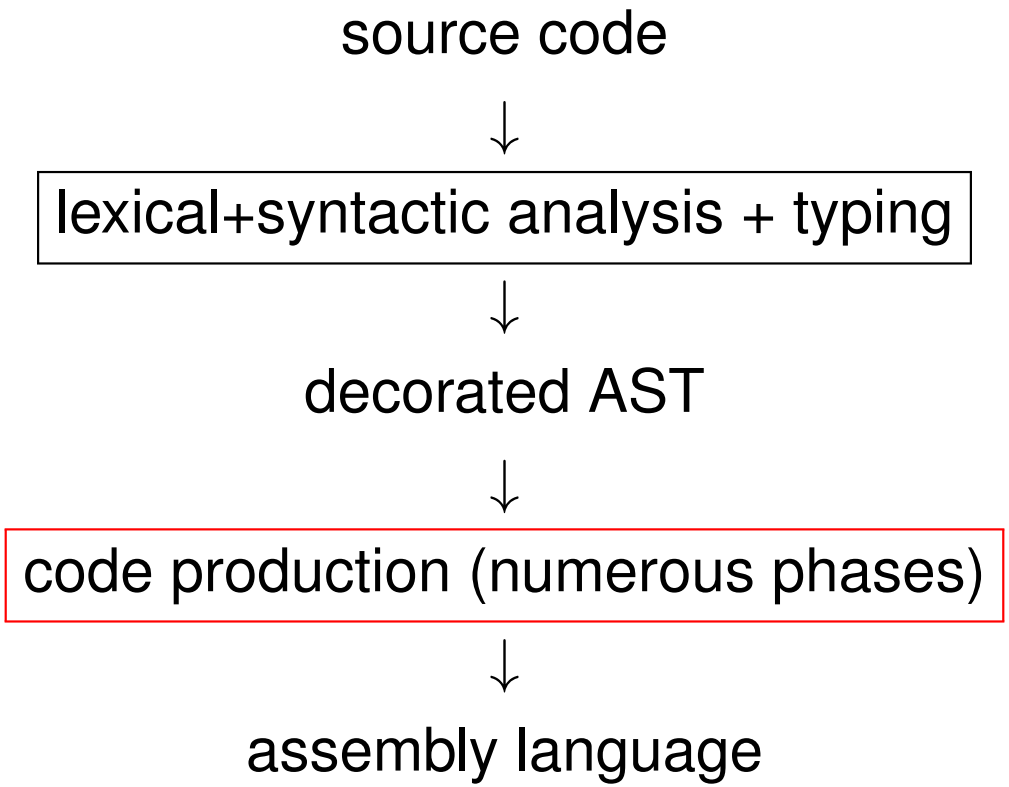
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2020-2021



Where are we ?



- We work on IRs (Middle-end).

1 Register allocation - Intro

2 A tour on data-flow Analyses

3 Register Allocation with graph coloring

4 LAB : smart code Generation

Credits

Fernando Pereira's course on register allocation:

http://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/RegisterAllocation.pdf

What for ?

- Finding storage locations to the values manipulated by the program ▶ registers or memory.
- registers are fast but in small quantity.
- memory is plenty, but slower access time.

▶ A good register allocator should strive to keep in registers the variables used more often.

"Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important - if not the most important - of the optimizations."

Hennessy and Patterson (2006) - [Appendix B; p. 26]

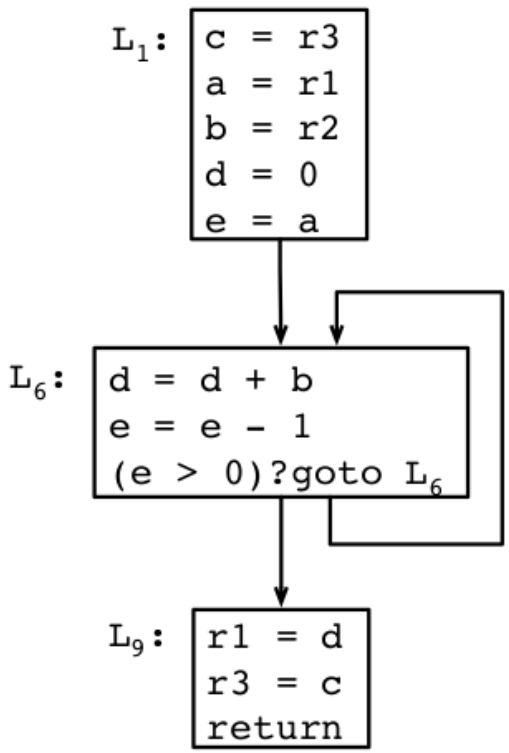
What for?

Expected behavior of **register allocation**:

- Input: a CFG with basic blocks with 3-address code (and pseudo-registers, aka temporaries)
- Output: same CFG but without pseudo-registers:
  - replace with physical registers as much as possible.
  - if not **splill**, ie allocate a place in memory.
  - all copies assigned to the same physical registers (“moves”) can be removed: **coalescing** (**optional**).

Register constraints

Some variable are assigned to some specific registers (compiler, architecture constraints)



- r1,r2,r3 are used to pass function arguments here.

The key notion: liveness

Observation

Two variables that are simultaneously **alive** must be assigned different registers.

(formal definition of alive follows)

Register assignment is NP-complete

Theorem

Given P and K general purpose registers, is there an assignment of the variables P in registers, such that (i) every variable gets at least one register along its entire live range, and (ii) simultaneously live variables are given different registers ?

Gregory Chaitin has shown, in the early 80's, that the register assignment problem is NP-Complete (register allocation via coloring, 1981)

3-phase algorithm

- **Liveness analysis**
  - When is a given value necessary for the rest of the computation?
- **Interference graph**
  - A graph that encodes which pseudo-registers cannot be mapped to the same location.
- **Graph coloring** then register allocation.
  - The effective allocation: physical registers and stack allocation for pseudo-registers.

1 Register allocation - Intro

2 A tour on data-flow Analyses
- A first example: Liveness Analysis
- Other data-flow analyses ENSL Only

3 Register Allocation with graph coloring

4 LAB : smart code Generation

1. Register allocation - Intro
2. A tour on data-flow Analyses
   - A first example: Liveness Analysis
   - Other data-flow analyses ENSL Only
3. Register Allocation with graph coloring
4. LAB : smart code Generation

Liveness analysis

In the sequel we call **variable** a pseudo-register or a physical register.

Definition (Alive Variable)

In a given program point, a variable is said to be alive if the value she contains may be used in the rest of the execution.

May: non decidable property ▶ overapproximation.

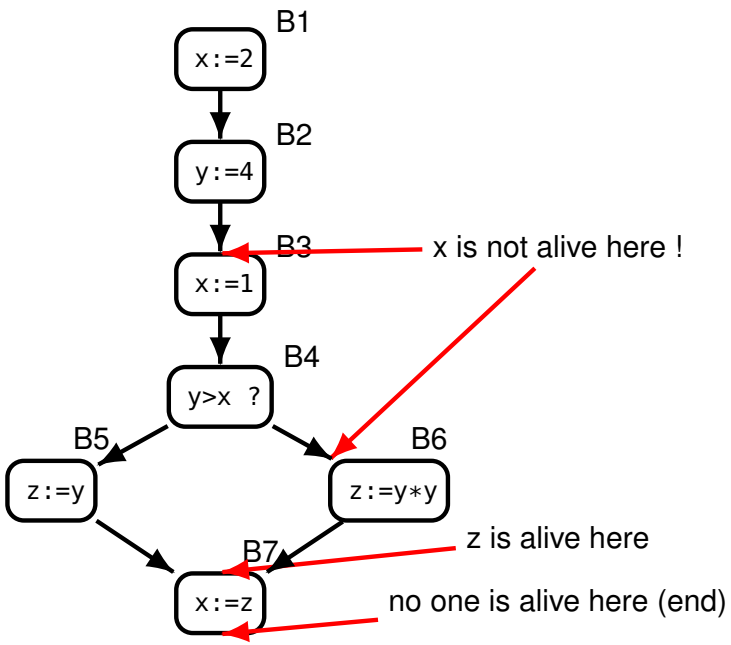
Important remark: here a block = a statement/program point.
We have the same kind of analyses with block=basic block.

An example for live ranges

Definition

A variable is **live** at the exit of a block if there exists a path from the block to a use of the variable that does not redefine the variable.

```
x:=2;
y:=4;
x:=1;
if (y>x)
   then z:=y
   else z=y*y ;
x:=z;
```



- The information flow is **backward**: from uses to definitions.

Data flow expressions

Definition

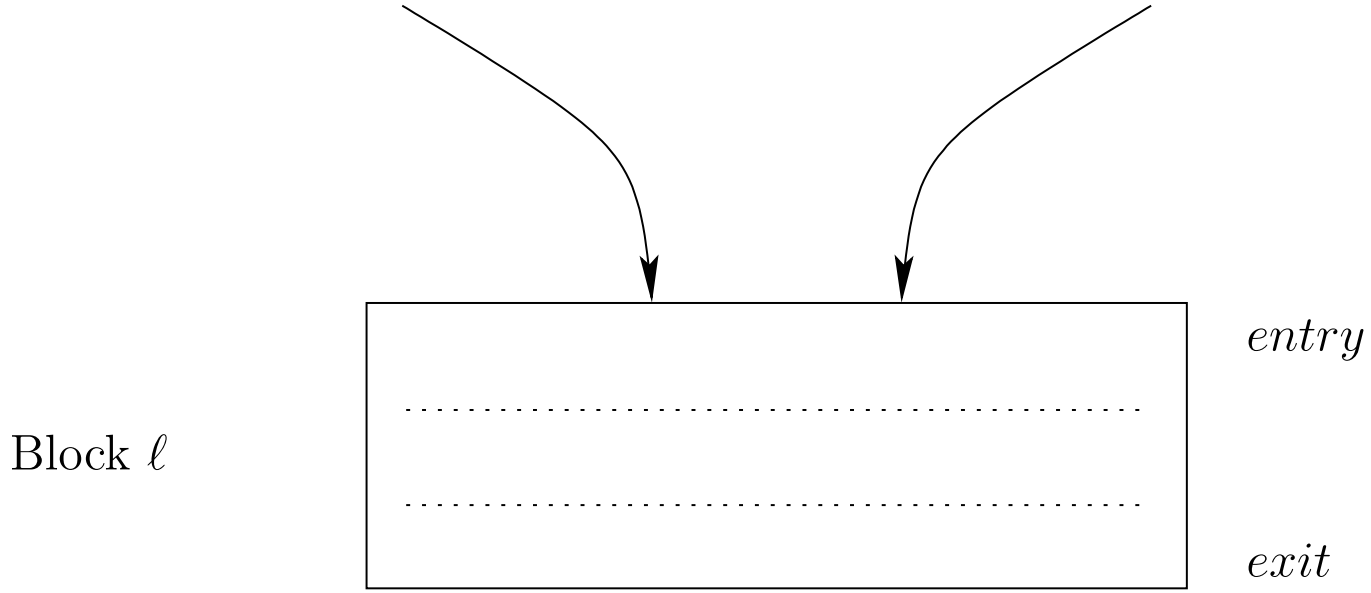
A variable that appears on the left hand side of an assignment is **killed** by the block. Tests do no kill variables.

Definition

A **generated** variable is a variable that appears in the block.

- Sets : $kill_{LV}(block)$ and $gen_{LV}(block)$

Data flow expressions



$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = final \\ \bigcup\{LV_{entry}(\ell') | (\ell, \ell') \in flow(G)\} \end{cases}$$

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

Data flow equation: solving

Here:

- Initialise LV sets to $\emptyset$.
- Compute LV_{entry} sets, then LV_{exit}, and continue.
- Stop when a fix point is reached.

▶ (vector of) Sets are strictly growing, and the live range set is at most the set of all variables, thus **this algorithm terminates**.

Steps

$LV_{entry}(\ell)$ denoted by $In(\ell)$, $LV_{entry}(\ell)$ by $Out(\ell)$ initilisation to emptysets is not depicted.

			Step 1		Step 2		Step 3 (stable)
ℓ	$kill(\ell)$	$gen(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$	$Out(\ell)$	$In(\ell)$
1	$\{x\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{y\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{y\}$	$\emptyset$
3	$\{x\}$	$\emptyset$	$\emptyset$	$\{x,y\}$	$\{y\}$	$\{x,y\}$	$\{y\}$
4	$\emptyset$	$\{x,y\}$	$\{x,y\}$	$\{y\}$	$\{x,y\}$	$\{y\}$	$\{x,y\}$
5	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
6	$\{z\}$	$\{y\}$	$\{y\}$	$\{z\}$	$\{y\}$	$\{z\}$	$\{y\}$
7	$\{x\}$	$\{z\}$	$\{z\}$	$\emptyset$	$\{z\}$	$\emptyset$	$\{z\}$

Final result and use

Backward analysis and we want the smallest sets, here is the final result : (we assume all vars are dead at the end).

ℓ	$LV_{entry}(\ell)$	$LV_{exit}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	$\emptyset$

- Use : Dead code elimination ! Note : can be improved by computing the use-defs paths. (see Nielson/Nielson/Hankin)

1. Register allocation - Intro
2. A tour on data-flow Analyses
   - A first example: Liveness Analysis
   - Other data-flow analyses ENSL Only
3. Register Allocation with graph coloring
4. LAB : smart code Generation

Common subexpressions / Available expressions

Avoiding the computation of an (arithmetic) expression :

```
x:=a+b;
y:=a*b;
while(y>a+b) do
   a:=a+a;
   x:=a+b;
done
```

AE : genesis of the analysis

An expression is available at a control point if its current value has already been computed earlier in the execution:

- Sets of expressions.
- How does information originate? from a use that do not redefine any operand!
- How does information propagate? from top to bottom.
- How do we join info after tests?

Some defs

Definition

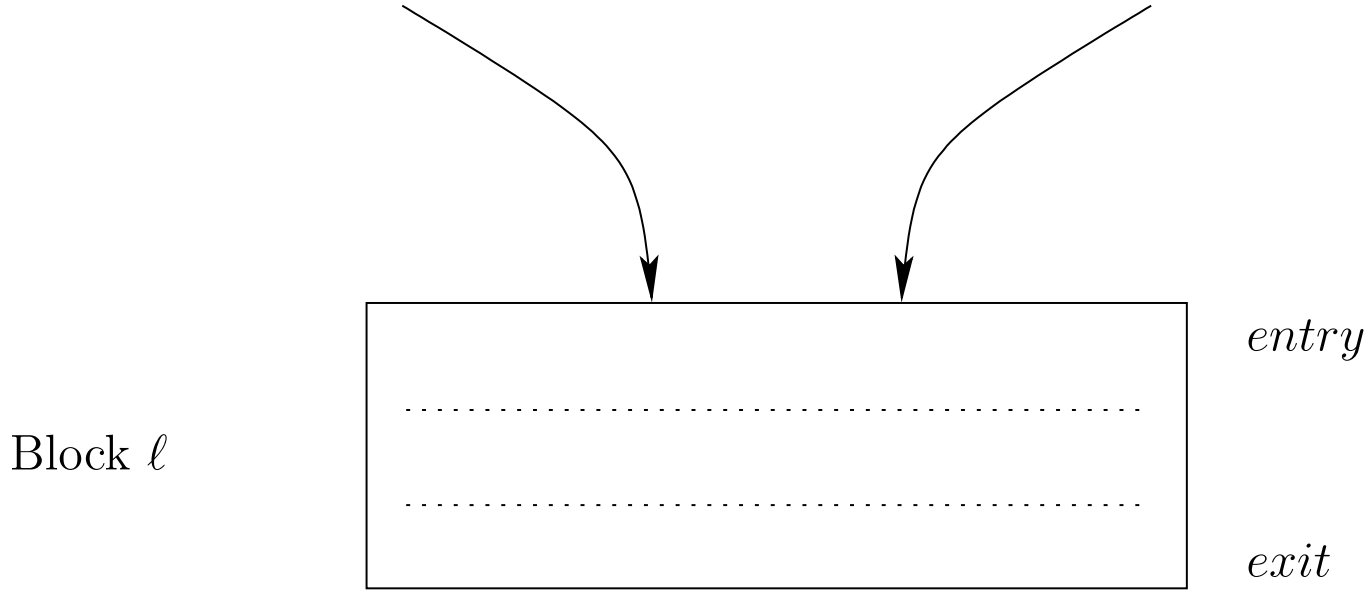
An expression is **killed** in a block if any of its variables is defined in the block.

Definition

A **generated** expression is an expression evaluated in the block and none of its variables is killed in the block.

- Sets : $kill_{AE}(block)$ and $gen_{AE}(block)$

Data flow expressions

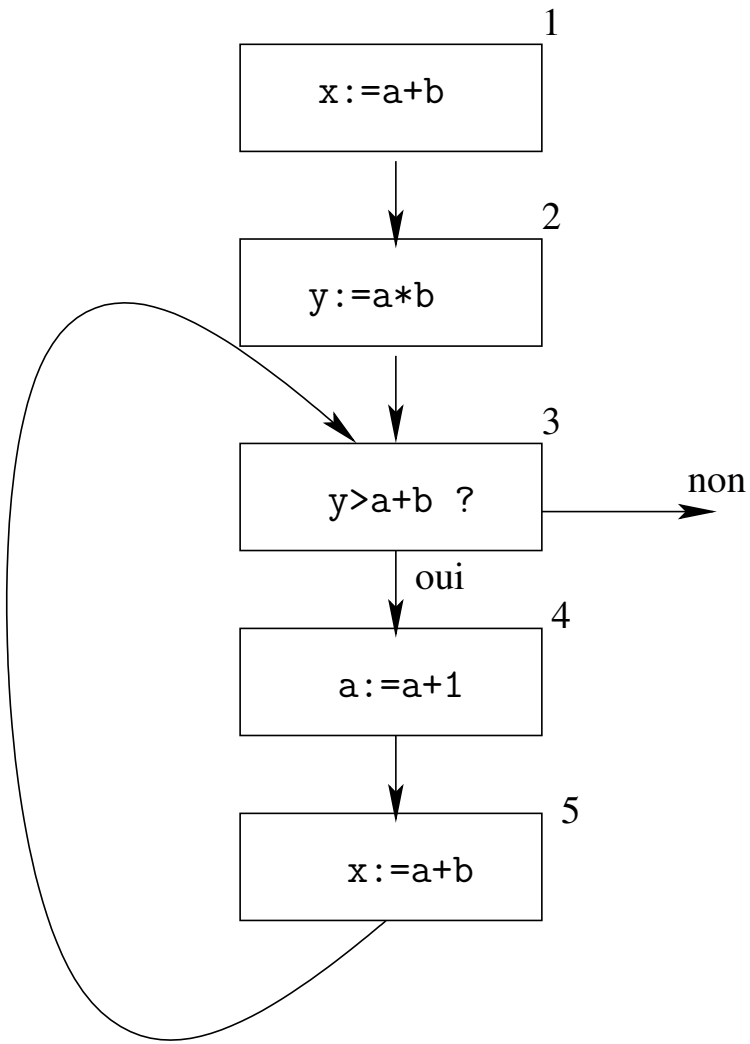


$$AE_{entry}(\ell) = \begin{cases} \emptyset & \text{if } \ell = init \\ \bigcap\{AE_{exit}(\ell') | (\ell', \ell) \in flow(G)\} & \end{cases}$$

$$AE_{exit}(\ell) = \left(AE_{entry}(\ell) \backslash kill_{AE}(\ell)\right) \cup gen_{AE}(\ell)$$

On the example - equations

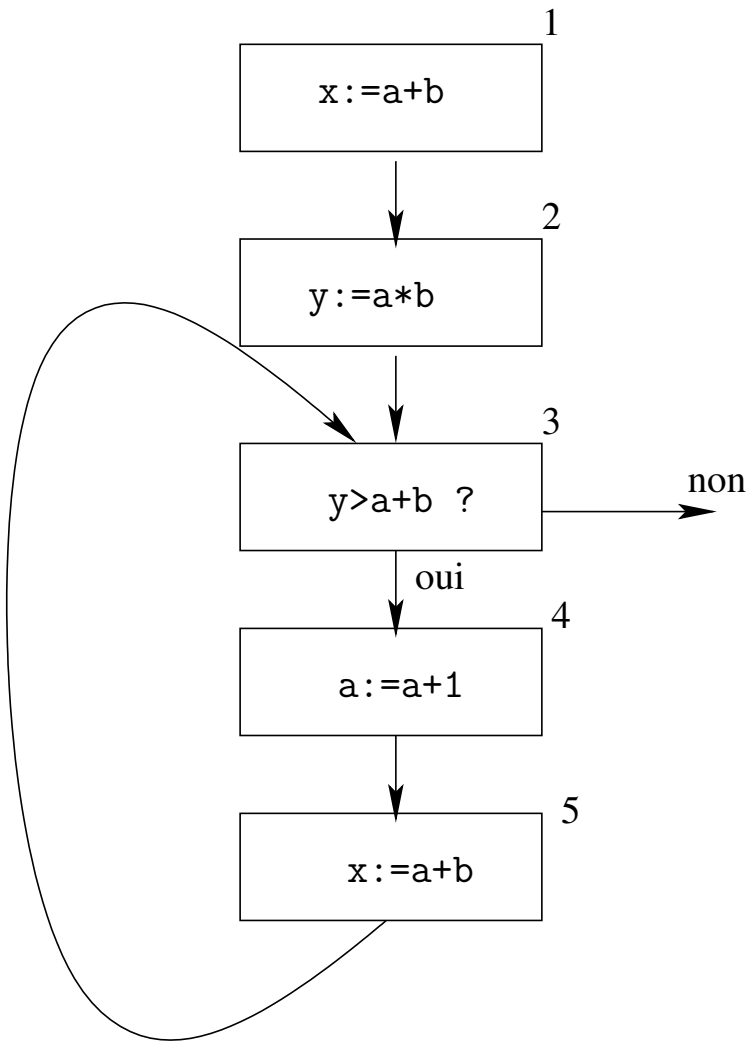
ℓ	$kill_{AE}(\ell)$	$gen_{AE}(\ell)$
1	$\emptyset$	{a+b}
2	$\emptyset$	{a∗b}
3	$\emptyset$	{a+b}
4	{a+b, a∗b, a+1}	$\emptyset$
5	$\emptyset$	{a+b}



On the example - final solution

ℓ	$AE_{entry}(\ell)$	$AE_{exit}(\ell)$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b, a∗b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}

- a+b is available on entry to the loop, not a∗b
- Improvement of code generation



Digression: common points ENSL Only

- Computing growing sets from $\emptyset$ via fixpoint iterations. (or the dual)
- Sets of equations of the form (collecting semantics) :

$$(\ell) = \bigcup_{(\ell',\ell)\in E} f((\ell'))$$

where f is computed w.r.t. the program statements.

- is an **abstract interpretation** of the program (see the course on Abstract Interpretation, later)

See also : Kam, J. B. and J. D. Ullman, "Monotone Data Flow Analysis Frameworks", Actal Informatica 7:3 (1977), pp. 305-318.

1 Register allocation - Intro

2 A tour on data-flow Analyses

3 Register Allocation with graph coloring

4 LAB : smart code Generation

Step 2: Interferences

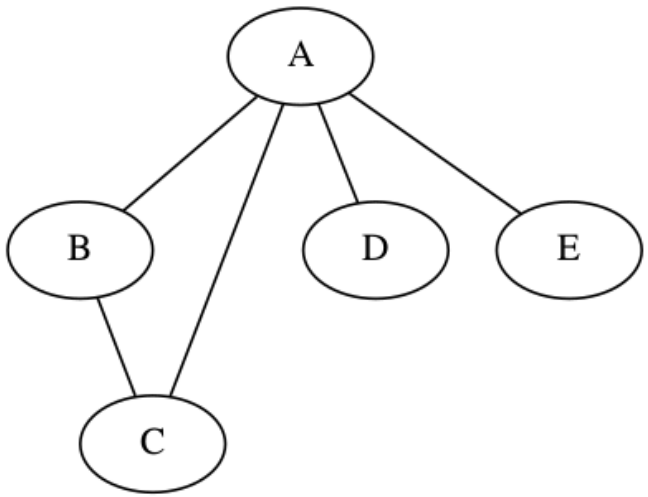
Here is the output of the liveness analysis for $a + (b + c)$:

	tmp_1	tmp_2	tmp_3	tmp_4	tmp_5	tmp_6
`load tmp1,la`						
`load tmp2,lb`	■					
`load tmp3,lc`	■	■				
`ADD tmp4, tmp2, tmp3`	■	■	■			
`LETI tmp5, tmp4`	■			■		
`ADD tmp6, tmp1, tmp5`	■				■	
⋮						■

- `tmp1` is in conflit with `tmp2` (because of instruction 3) denoted by $tmp_1 \bowtie tmp_2$.

Interference graph

A denotes `tmp1`, ... $\bowtie$ defines a graph:



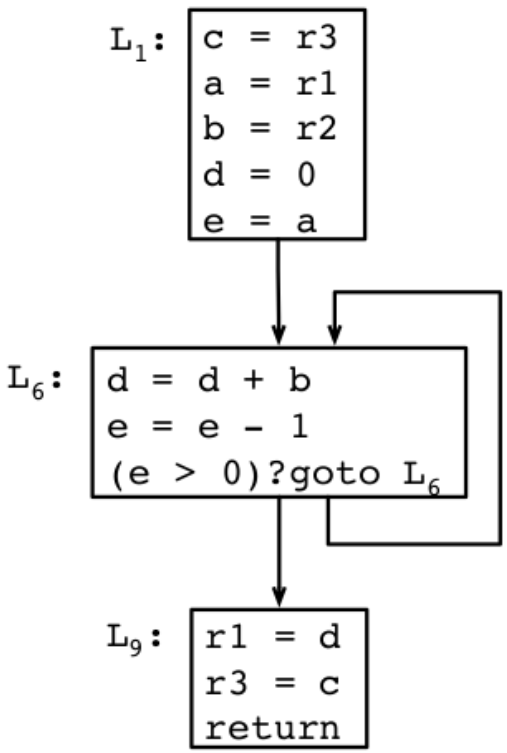
We want a **correct allocation** with respect to $\bowtie$:

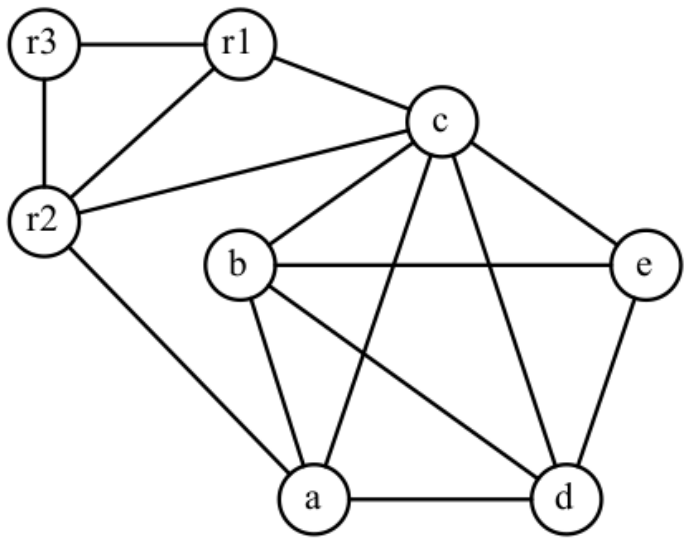
$tmp_1 \bowtie tmp_2 \implies Alloc(tmp_1) \neq Alloc(tmp_2)$.

- Graph coloring.

Running example

Important: in this example consider the r_i as temporary registers, like the others.





Kempe's simplification algorithm 1/2

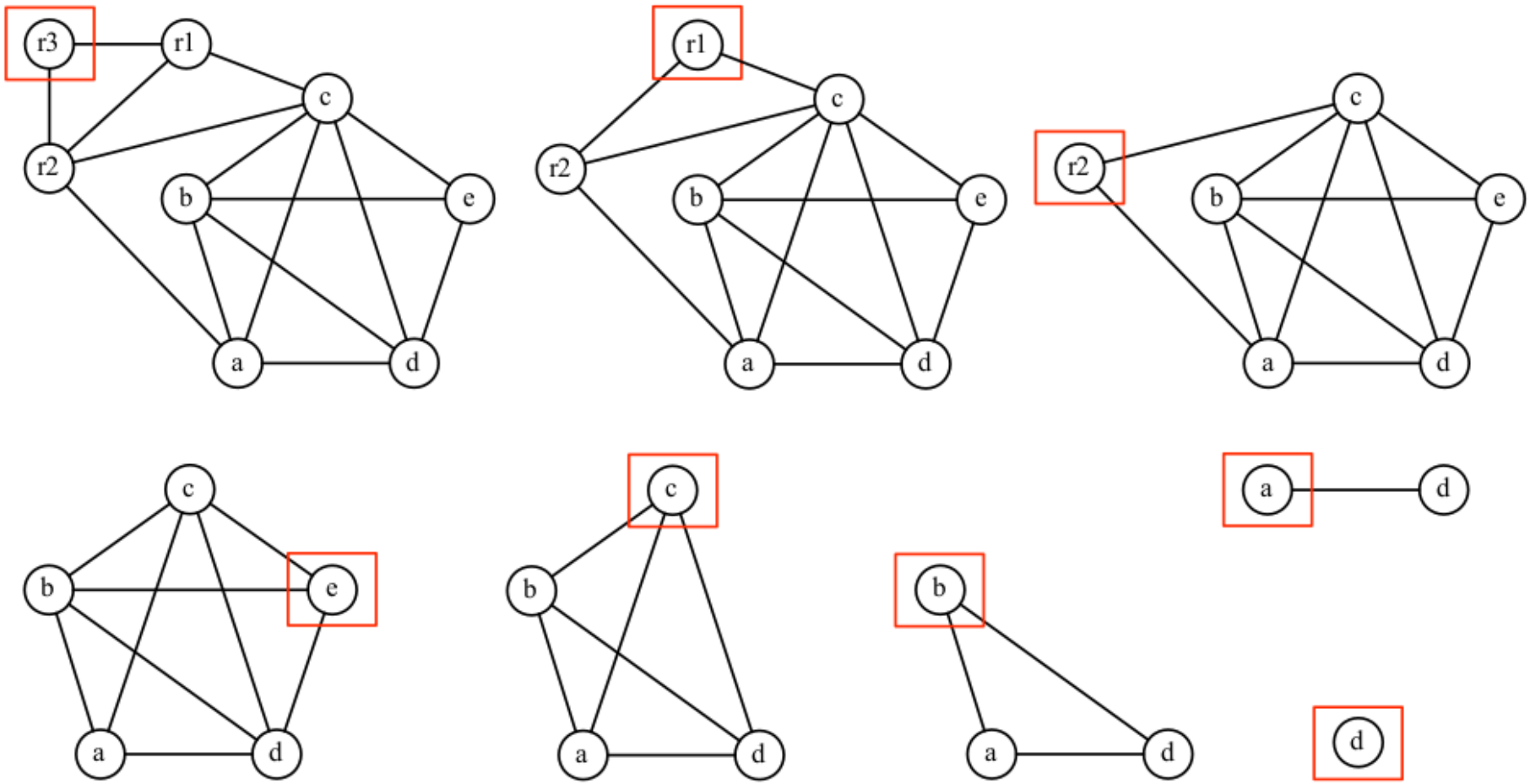
On the interference graph (without coalesce edges):

Proposition (Kempe 1879)

Suppose the graph contains a node m with fewer than K neighbours. Then if $G' = G \setminus \{m\}$ can be colored, then G can be colored as well.

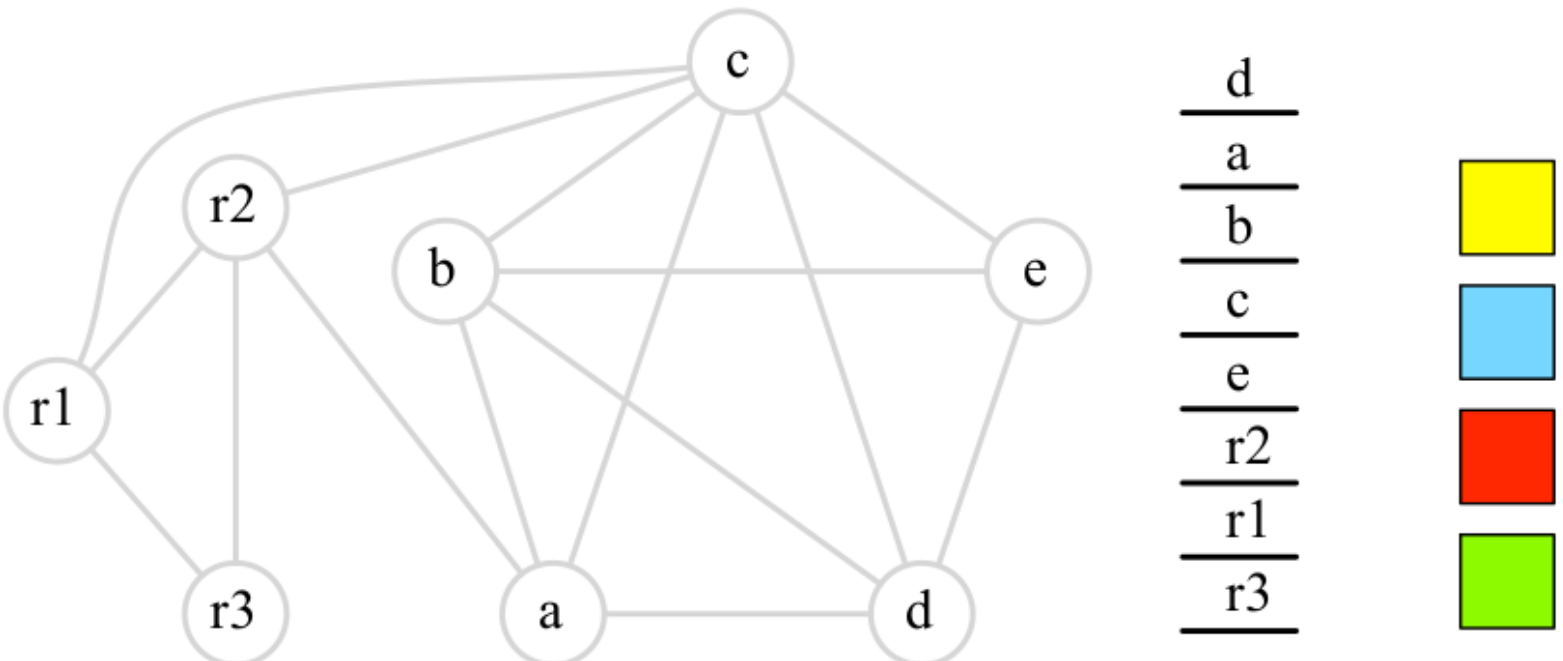
- Pick a low degree node, and remove it, and continue until remove all (the graph is K-colorable) or ...

Kempe's simplification algorithm 2/2

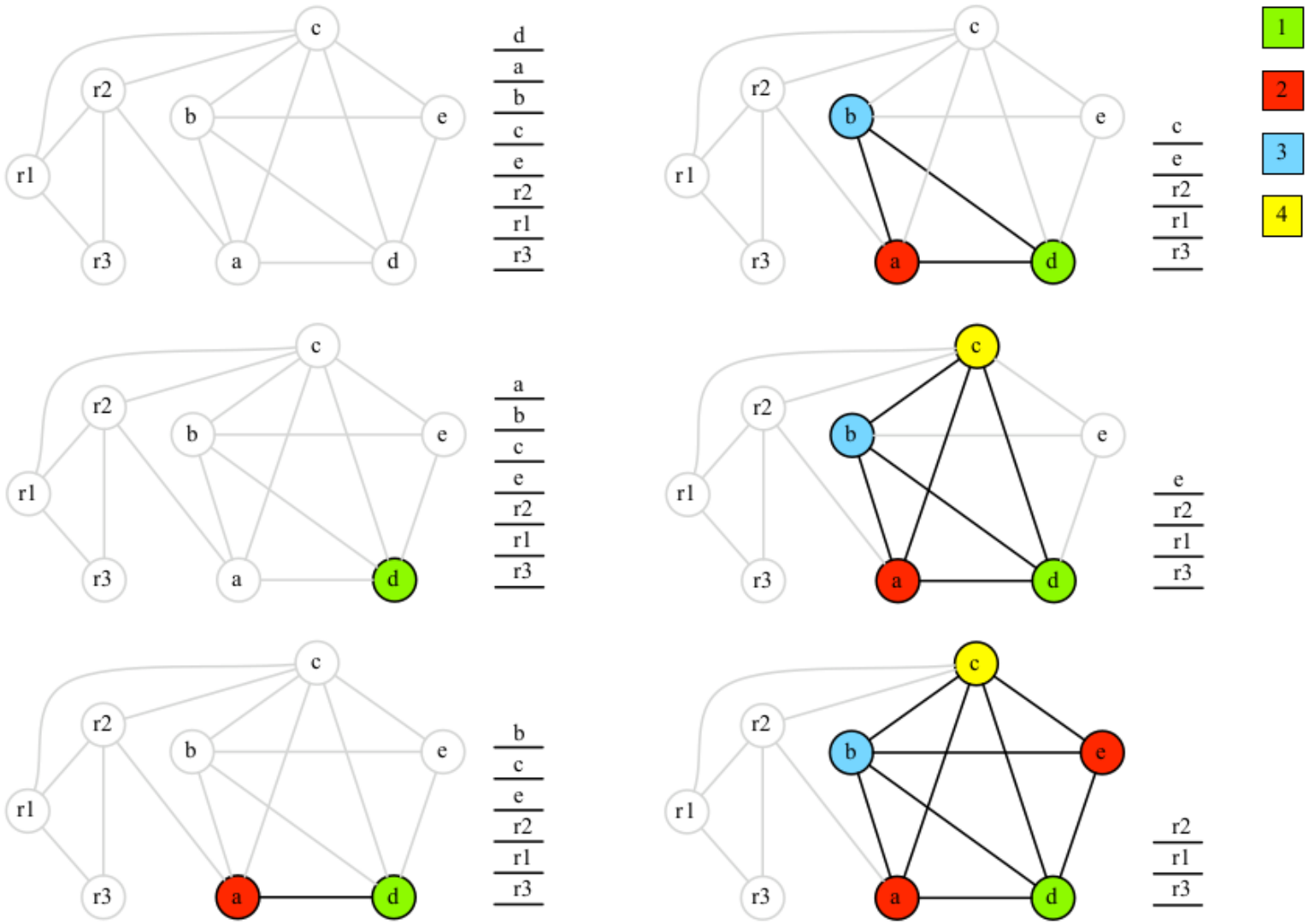


Let's color!

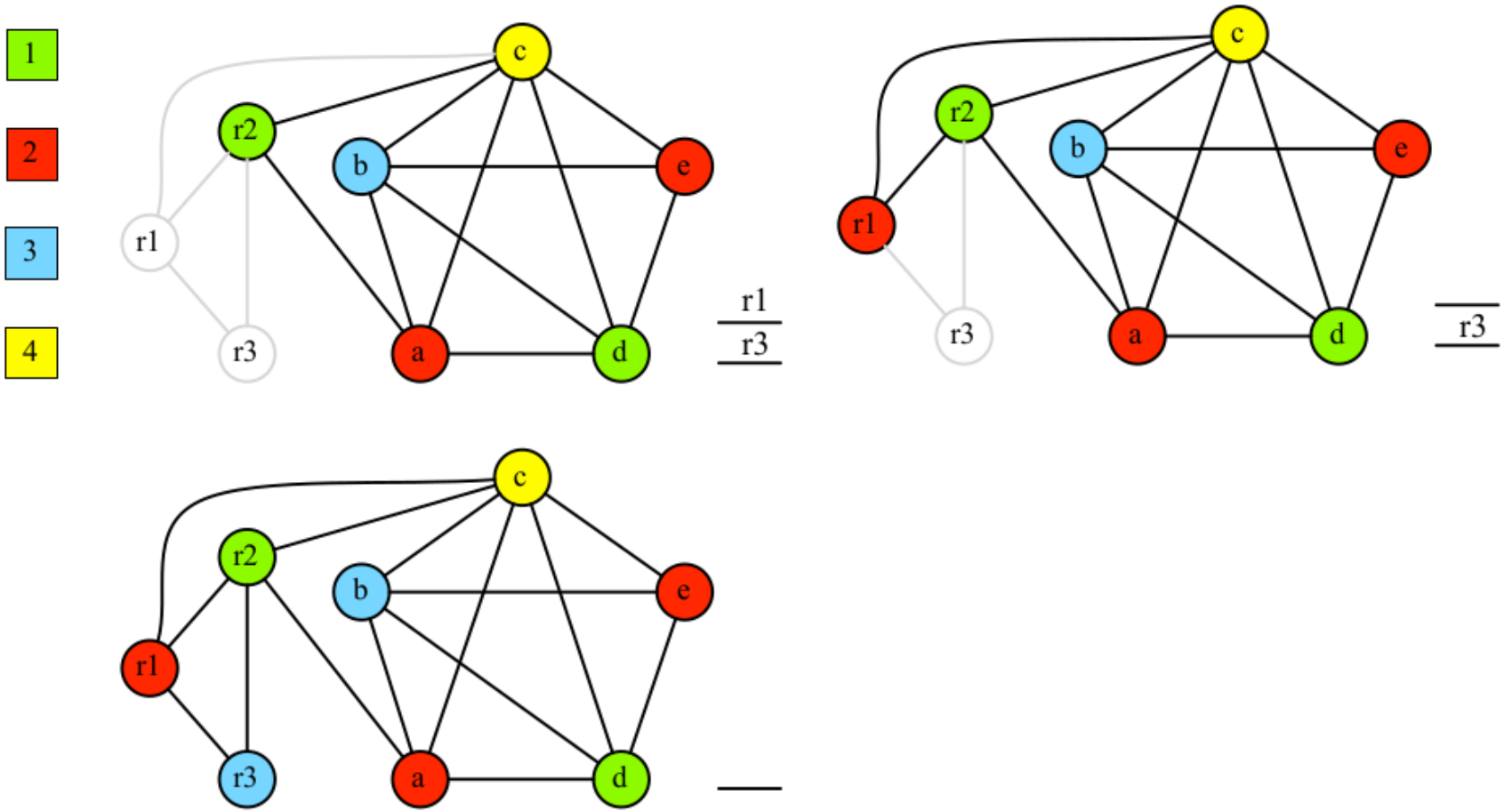
- We assign colors to the nodes greedily, in the reverse order in which nodes are removed from the graph.
- The color of the next node is the first color that is available, i.e. not used by any neighbour.



Greedy coloring example 1/2



Greedy coloring example 2/2



If the graph is not colorable

Non-colored variables are named **spilled pseudo-registers**.

Idea: Modify the code to lower the number of simultaneously alive registers.
Plenty of solutions, the simplier is to reserve a dedicated location for a given spilled variable, and store and load from memory.

Solution 1: for non-colored variables

Invent 2 versions of the same variable (**live-range splitting**), and modify the code into:

```
ADD temp51, temp4, temp3
STORE temp51, [locationinmemory] # replace with
                                 # actual location
..
LOAD temp52, [locationinmemory]  #same
ADD temp6, temp52, #5
```

- But now we have to allocate these two new variables!

We relaunch the coloring algorithm. This is called iterative register coloring.

Second solution

- Launch the coloration algorithm with an infinite number of colors:
  - first colors are mapped to registers (used in priority by the coloring algorithm)
  - other colors are mapped to offsets in the stack, i.e. spilled to memory
- Drawback: we need a few registers to implement the spilling $\Rightarrow$ less efficient than iterative register coloring.

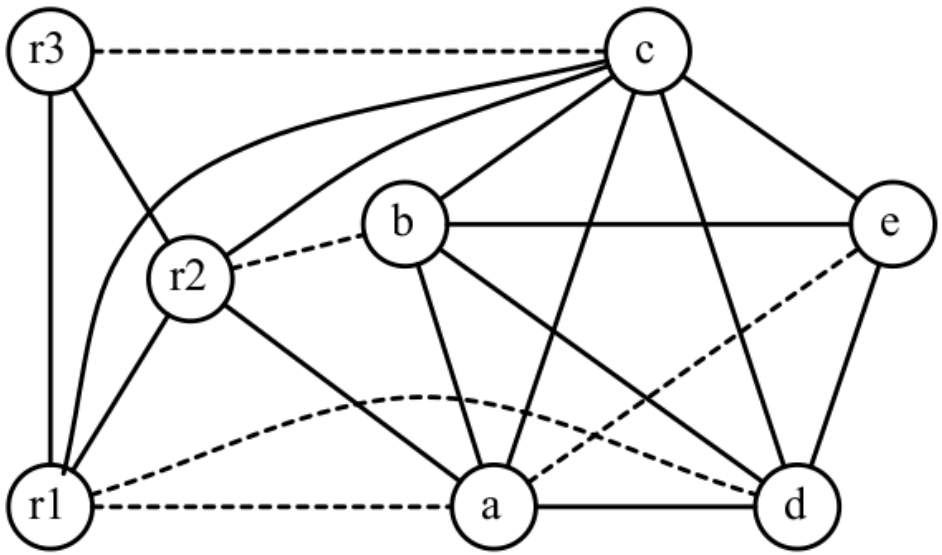
▶ We implement this one.

Physical Memory Allocation

We will invent physical memory locations from the stack pointer (like in Lab 4).

Other Algorithms

- **Linear scan**: greedy coloring of interval graphs. (see Fernando Pereira's slides on register allocation: 18 to 35)
- **Iterative Register Coalescing** (George/Appel, TOPLAS, 1996) (same, from slides 44), which uses "coalesce edges" (variables are related by move instructions).
- Plenty of other heuristics for splilling.



A nice result

Chordal graphs are P-colorable

For certain classes of graphs, graph coloring is P. This is the case for **cordal graphs** where every cycle with 4 or more edges has a chord (connects 2 vertices in the cycle but not part of the cycle).

Important result (Sebastian Hack): Programs in strict SSA form have this property.

- Pereira Palsberg Register allocation (APLAS 2005).

1 Register allocation - Intro

2 A tour on data-flow Analyses

3 Register Allocation with graph coloring

4 LAB : smart code Generation

Code Generation

Input: a MiniC file:

```
int n;
n=6;
```

Output: a RISCV file:

;; (stat (assignment n = (expr (atom 6)) ;))

```
li t6, 6
mv t7, t6
```

- but with a smart allocation of registers and memory.

Summary

1. Register allocation - Intro
2. A tour on data-flow Analyses
   - A first example: Liveness Analysis
   - Other data-flow analyses ENSL Only
3. Register Allocation with graph coloring
4. LAB : smart code Generation

Follow up: functions, program analyses, language extensions: ENSL Only

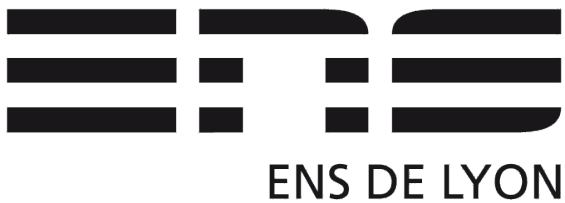
Compilation Labs (2020)

Laure Gonnord & Matthieu Moy & other

https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2020-2021



1 LAB: RISCV

2 LAB : ANTLR Startup

3 LAB : Interpreter for MiniC

4 LAB: Direct Code Generation

5 LAB: Code Generation with smart IRs

Content

RiscV startup

MIF08

and a Python tutorial

1 LAB: RISCV

2 LAB : ANTLR Startup

3 LAB : Interpreter for MiniC

4 LAB: Direct Code Generation

5 LAB: Code Generation with smart IRs

Semantic actions in practice: ANTLR/Lab 2

(ariteval) Input:

```
20 + 22;
a = 4;
a + 2;
a * 5;
```

Output :

```
20+22 = 42
a now equals 4
a+2 = 6
a*5 = 20
```

Code Infrastructure (Python)

```
../TP02/ariteval$ ls
Arit2.g4        test_ariteval.py
testfiles/      arit2.py
Makefile        test_expect_pragma.py
```

- The grammar is written in `Arit2.g4`.
- `arit2.py` the main file (command line handling, ...).
- `test_ariteval`: unit test script.
- `testfiles/` test files.

Code: Development

- Code and test the grammar (g4 file)
- Use semantic actions (in `Arit2.g4`) :

```
expr returns [int val]:
      e=expr '+' t=term
        {$val=$e.val+$t.val;}
| ... ;
```

- While developing, test **single files** with command line:

```
make ; python3 arit2.py testfiles/blablabla.txt
```

Code: Unit tests.

To test for a (quite) large number of testcases, we will use the `pytest` infrastructure. Test files have the form (expected results in comments):

```
1;
-12;
// EXPECTED
//1 = 1
//-12 = -12
```

and the rest is automatic, for one single file type:

```
python3 arit1.py testfiles/montest.txt
```

and for all tests:

```
make tests
```

- You should write (and deliver) your own test cases !

Grade, Plagiarism

- Part of this lab is graded (Individual work)
- No code sharing allowed for any graded work
- Students sanctionned regularly for plagiarism

1 LAB: RISCV

2 LAB : ANTLR Startup

3 LAB : Interpreter for MiniC

4 LAB: Direct Code Generation

5 LAB: Code Generation with smart IRs

Example 2 ANTLR/Python : MiniC Interpreter (Lab3)

Input: a .c file:

```
int main(){
  float s;
  s=3.14;
  print_float(s);
  return 0;
}
```

Output: on std output:

```
3.14
```

Code Infrastructure

```
>cap-labs20.git/MiniC$ ls
Errors.py  MiniC.g4             README-interpreter.md  test_interpreter.py
Makefile   MiniCInterpreter.py  test_expect_pragma.py  TP03
>cap-labs20.git/MiniC$ls TP03/
MiniCInterpretVisitor.py  MiniCTypingVisitor.py  __pycache__  tests
```

- The grammar of the MiniC language:`MiniC.g4`.
- The main file (command line, driver for the lexer/parser/visitor): `MiniCInterpreter.py`
- Two visitors: one for typing, the other one to evaluate.
- A Makefile, a README.
- Testfiles, and a test script `test_interpreter.py`.

MiniC typing, MiniC visit

- A `MiniCTypingVisitor` to type MiniC programs given, it rejects programs like:

  ```
  int x;
  x="blablabla";
  ```

  $\Rightarrow$ You only have to read the code and play with it to understand how it works.

- A `MiniCInterpretVisitor`, that executes the program. We provide you as an example the arithmetic expression evaluation (and the corresponding test `test00.c`).
  $\Rightarrow$ You have to complete the evaluation for assignments, tests, while.

Visitors

See course 3 or the pdf of the lab. Implement according to grammar rules names:

```
    | expr myop=(PLUS|MINUS) expr                #additiveExpr
```

(used to accept expressions like $43 - 1$ or $40 + 2$). While parsing, this rule will launch the function (in `MiniCInterpretVisitor`):

```
    def visitAdditiveExpr(self, ctx):
[...]
```

which eventually compute the addition/substraction of the two subexpressions.

How to store the interpreter state or the typing environnement?

```
x = 42;        // store the value during assigment (sigma)
print_int(x); // get back this value
```

The store should be global, thus a class variable, here we chose a dictionary: $name \rightarrow value$.

```
class MiniCInterpretVisitor(MiniCVisitor):
    def __init__(self):
        self._memory = dict()
# and somewhere:
    self._memory[name] = value
# and somewhere else:
   val = self._memory[name]
```

Test infrastructure (same as in Lab 2)

You write your testcases and expected results:

```
int main(){
  print_int(3^2+45*(-2/-1));
  print_int(23+19);
  print_int(false || 3 != 7)
  print_string("coucou");
  return 0;}
// EXPECTED
// 99
// 42
// 1
// coucou
```

```
int main(){
  int u; bool b;
  u=3;  b=true;
  if (b) { u=u+1; }
  else { u=u-1; }
  print_int(u);
  return 0;}
// EXPECTED
// 4
```

$\Rightarrow$ a helper script (using `pytest`) compares the actual and the expected outputs.

Test infrastructure 2/2

```
========================================= test session starts ==========================
platform linux -- Python 3.7.3, pytest-3.10.1, py-1.7.0, pluggy-0.8.0 -- /usr/bin/py
cachedir: .pytest_cache
rootdir: /home/laure/Documents/VCS/Teaching/compil-lyon/TP2019-20/TP03/MiniC-type-in
, inifile:
plugins: cov-2.7.1
collected 7 items

test_evaluator.py::TestEval::test_eval[./ex/test00.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/double_decl00.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/bad_type01.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/bad_type_bool_bool.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/bad_type00.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/bad_type03.c] PASSED
test_evaluator.py::TestEval::test_eval[./ex-types/bad_type02.c] PASSED

================================== 7 passed in 0.49 seconds ============================
```

⇒ Using this test framework is mandatory.

1 LAB: RISCV

2 LAB : ANTLR Startup

3 LAB : Interpreter for MiniC

4 LAB: Direct Code Generation

5 LAB: Code Generation with smart IRs

Code Generation

Input: a MiniC file:

```
int main(){
int n;
n=6;
return 0;}
```

Output: a RISCV file:

```
  [...]
        ;; (stat (assignment n = (expr (atom 6)) ;))
3       LI t1, 6        ; t1 is a riscv register.
        MV t2, t1
  [...]
```

Code Generation, first step

- 3-address code generation according to the code generation rules of the course:

```
// e1+e2 code generation rule
  temp_1 <- GenCodeExpr(e_1)
  temp_2 <- GenCodeExpr(e_2)
  dest_tmp <- new_tmp()
  code.add(InstructionADD(dest_tmp, temp_1, temp_2))
  return dr
```

- **TODO: implement them**:

```
tmpl = self.visit(ctx.expr(0))
tmpr = self.visit(ctx.expr(1))
dest_tmp = self._current_function.new_tmp()
if ctx.myop.type == MuParser.PLUS:
    self._current_function.addInstructionADD(dest_tmp, tmpl,
        tmpr)
```

Result after first step

The previous step uses instructions of an API like:

> self._current_function.addInstructionADD(dr, tmpl, tmpr)

whose side effect is to construct a RISCV prog as a list of 3 adresses instructions with temporaries (virtual registers, from the class `Temporary`.

This list can be dumped (with `printCode` in the API) into a .s file:

;; (stat (assignment n = (expr (atom 6)) ;))

```
li temp_1, 6
mv temp_2, temp_1
```

We cannot test: it is not executable!

Code Generation, second step

The allocation process:

- takes as input the preceding result
- modifies the list of instructions with temporaries into list of instructions with physical registers or accesses to memory.
- a trivial allocator is given.

TODO : all in memory allocation (see course)

Code Infrastructure (only files for THIS LAB))

```
MiniC$ ls
Makefile MiniC.g4 test_codegen.py MiniCC.py [...]

MiniC$ ls TP04/
APIRiscV.py       libprint.s                 Operands.py  tests
Instruction3A.py  MiniCCodeGen3AVisitor.py   printlib.h
```

- The MiniC grammar in `MiniC.g4`, a Makefile, as usual.
- Unit tests in `test_codegen.py`.
- API for generating RISCV code : `APIRiscV`, ...
- `Allocations.py`: allocators for RISCV code.
- **TODO : edit and fill `MiniCCodeGen3AVisitor.py` mainly**. You may have other changes to make in other files (Allocation).

RISCV API

In this API (APICodeRISCV, Instruction3A, Operands) :

- A class for a program RISCV `RiscVFunction`. The program contains a list of instructions, methods to add instructions, to increment temporary numbers, ...
- Classes for instructions: `Instruction`, `Instru3A`, `Label`
- A 3 address instruction contains arguments that can be `Immediate`, `Temporary`, `Register`, or a `Condition` in the special case of the condjump. ...
- the `CondJump` instruction (label,dr1,cond,dr2) has the meaning: `if (dr1 cond dr2) jump to label`
- Ignore code concerning graphs (print dot, add edges) and dataflow (in and out sets), this is for next lab.

Allocations/replace

In `TP05/Allocations.py`:

- `replace_*` functions replace temporary operands of a given instruction with the help of the current allocation (see the example for naive allocation).
- The allocation itself is done before in Allocator classes : `*Allocator` (ignore the smart allocation in Lab 4).

tests

While developing, write appropriate (mini) tests and use :

```
python3 MiniCC.py --reg-alloc=xxx /path/to/example.c
```

and have a look at the generated file.

Next step is to verify everything:

```
make tests
```

to launch all tests in `tests*/*` files (this setting is in `test_codegen.py`.

1 LAB: RISCV

2 LAB : ANTLR Startup

3 LAB : Interpreter for MiniC

4 LAB: Direct Code Generation

5 LAB: Code Generation with smart IRs

Code Generation

Input: a MiniC file:

```
int main(){
int n;
n=6;
return 0;
}
```

Output: a RISCV file:

```
[...]
2           ;; (stat (assignment n = (expr (atom 6)) ;))
            li t6, 6
            mv t7, t6
[...]
```

Big picture

- Construct the CFG (already done)
- Compute liveness information:
  - **TODO** initialize GEN and KILL)
  - **TODO** ENSL Only fixpoint computation.
- Compute the interference graph (**TODO**: interfere function)
- Color it (**TODO** call an API method)
- Allocation: temps in registers, splilled temps in memory (**TODO**)
- Rewrite instructions wrt the allocation. (**TODO**)
- Pretty-print code (automatic)
- Test.

Liveness and interference graph

TODO for liveness, in `TP05/Allocations.py`

- Initialize dataflow sets (function `set_gen_kill`)
- ENSL Only implement the fixpoint computation. (function `run_dataflow_analysis`)
- Implement a `interfere` function, and complete `build_interference_graph`.

Coloring / Smart Allocator

TODO for coloring, in `TP05/Allocations.py`

Now for the coloring, in `smart_alloc`

```
(coloringreg, _, _)
      = self._igraph.color();
```

(calls the `Libgraph` coloring function).

then **TODO implement smart allocator**: (in smartalloc):

- if a temporary has a "register color" `(color < len(GP_REGS))`, allocate in a physical register.
- else in stack with an offset computed from the color.

Do not forget to implement `replace_smart`

Tests

Write **appropriate tests**; then run:

```
make tests
```

It launches tests for the dataflow, and for smart alloc and smart codegen.