CAP — Compilation and Program Analysis (#1) : Introduction

Laure Gonnord & Matthieu Moy & Gabriel Radanne & other

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

Master 1, ENS de Lyon et Dpt Info, Lyon1

2024-2025

Your teachers

Figure: Gabriel Radanne and Ludovic Henrio (CAP)

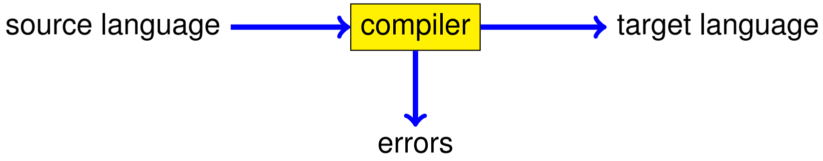
- Photo © Inria

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.

Most of the material is shared between the UCBL course “MIF08” (Compilation et Traduction de Programmes), the ENS course “CAP” (Compilation et Analyse de Programme) and ESISAR (Valence).

What's compilation?



Compilation toward the machine language

We immediately think of the translation of a high-level language (C,Java,OCaml) into the machine language of a processor (x86, 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);}
```

$\longrightarrow$

```
0010011110111101111111111110000010101111101111110000000000010100
1010111110100100000000000010000010101111101001010000000000100100
1010111110100000000000000001100010101111101000000000000000011100
10001111101011100000000000011100
```

Target Language

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 production.

Moreover, languages can be

- interpreted
- compiled into an intermediate language that will be interpreted
- compiled into another high level language
- compiled "on the fly"
- several of the above

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 :

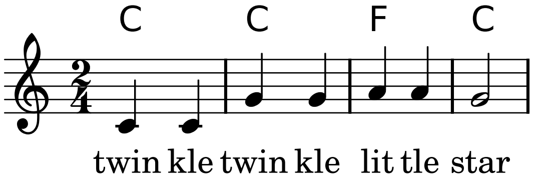
- The compiler makes a complex work once, to produce a code for whatever entry.
- An interpreter makes a simpler job, but on every entry.

▶ In general the code after compilation is more efficient.

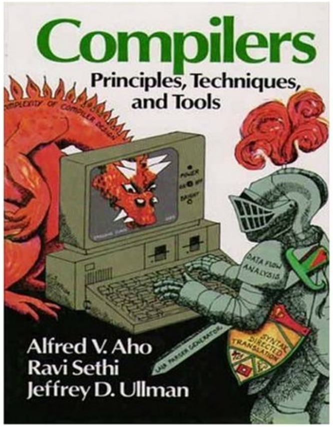
Example

$$\begin{tikzcd}\text{source} \arrow[r] & \boxed{\text{lilypond}} \arrow[r] & \text{PostScript file} \arrow[r] & \boxed{\text{gs}} \arrow[r] & \text{image}\end{tikzcd}$$

```
\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


Compilers
Principles, Techniques, and Tools
Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman
Complexity of Compiler Design
Data Flow Analysis
Syntax Directed Translation
LALR Parser Generator

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.”

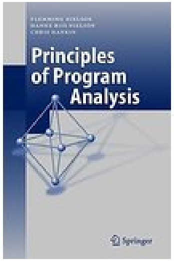
(Dragon Book, 2006)

Program Analysis

To prove:

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

. . . also in this course!



The course

- Syntax Analysis : lexing, parsing, AST, types.
- Evaluators.
- Code generation and optimisations.
- Formal Semantics (many versions!)
- Language extensions.

Goal:
Become familiar with the mechanisms inside a compiler
Understand how to reason about programming languages

The Lab

A complete compiler for the RISCV architecture!

⇒ Big guided programming project all along the semester

Support language: Python

Goal:

Become able to navigate a medium-sized programming project

Understand the link between theory and implementation

Grading

- Three graded labs
- One homework
- One final exam

$$Note = \frac{exam + average(Lab3, Lab4, Lab5, homework)}{2}$$

Grade, Plagiarism

- Part of this lab is graded (Individual work)
- You can collaborate but no code sharing is allowed for graded work
- Test files are considered as part of your work, no sharing either (obviously)
- Students sanctionned regularly for plagiarism
  $\Rightarrow$ Work shared between n students, grade divided by n.
- Graded work should be on time
  $\Rightarrow$ Minus 1 point per hour.

Course Organization

Everything is on the webpage:

https://github.com/Drup/cap-lab25

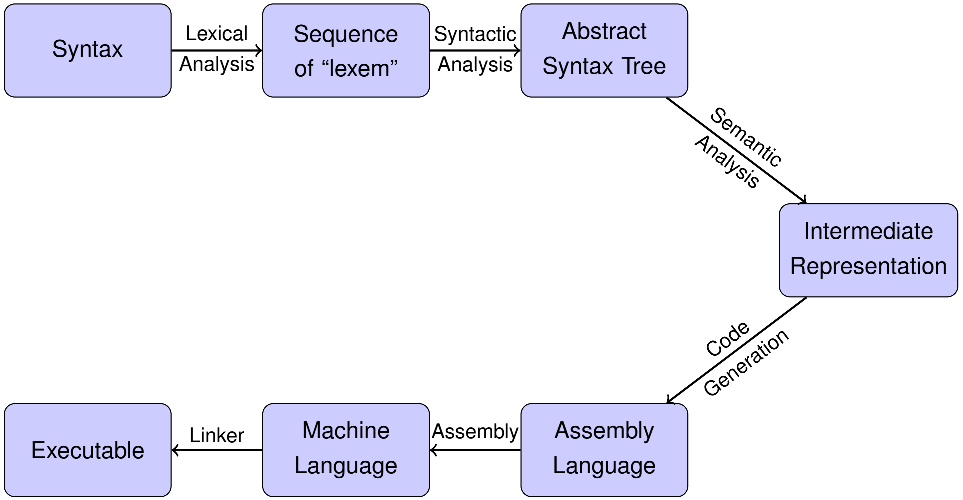
Read your emails !

Compiler phases

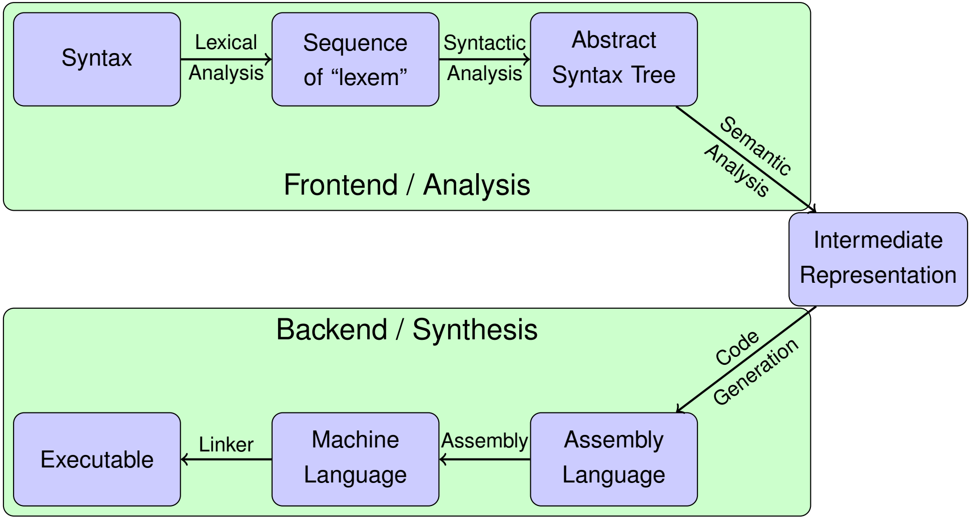
Usually, we distinguish two parts in the design of a compiler:

- 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.

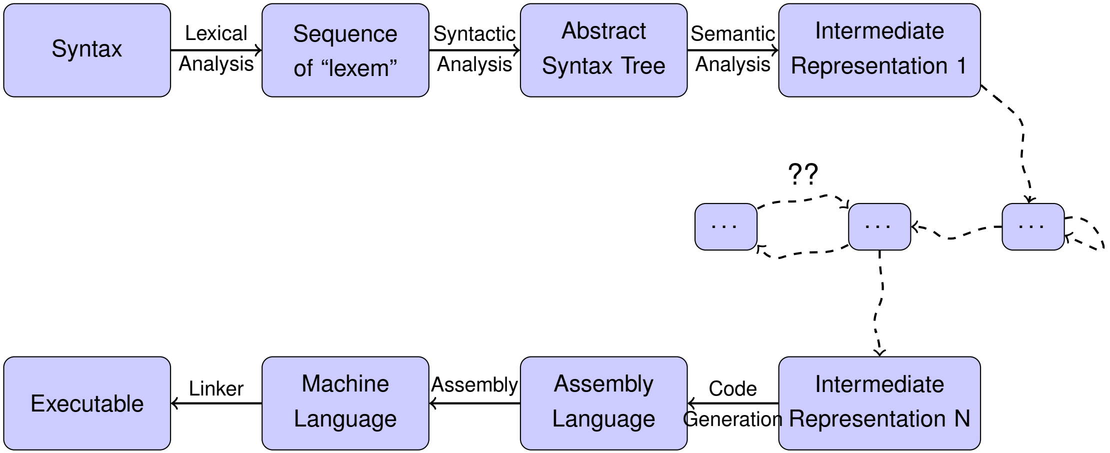
A Standard™ Compiler Pipeline



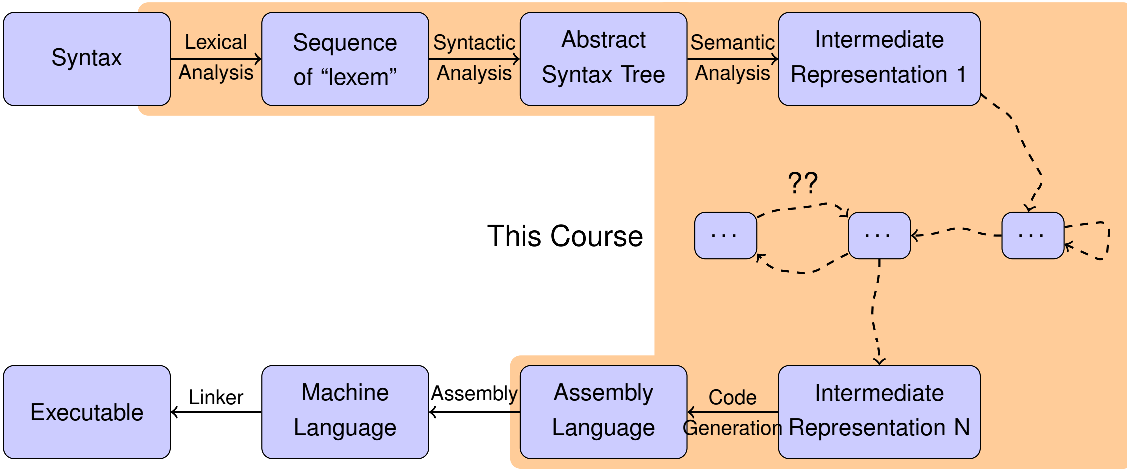
A Standard™ Compiler Pipeline



A Standard™ Compiler Pipeline



A Standard™ Compiler Pipeline

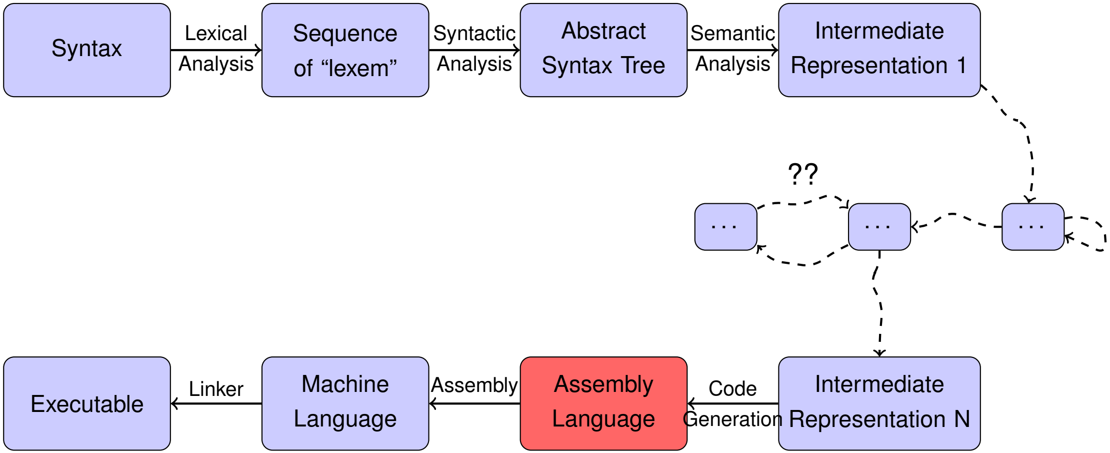


CAP — Compilation and Program Analysis (#1) : Introduction – **RiscV**

Master 1, ENS de Lyon et Dpt Info, Lyon1

2024-2025

A Standard™ Compiler Pipeline



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 (x0, x1, ...), the first integer register is a zero register, and the rest is general purpose. Registers have symbolic names (sp, fp, ...) 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 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`

cte=5, ri=t2=x7, 000, rd=t1=x6, 0010011

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`

cte=5, ri=t2=x7, 000, rd=t1=x6, 0010011

0000|0000|0101| 0011|1 000 |0011|0 001|0011

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`

cte=5, ri=t2=x7, 000, rd=t1=x6, 0010011

0000|0000|0101| 0011|1 000 |0011|0 001|0011

0x00538313

RISCV: branching

Unconditional branching (jump and link):

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

Test and branch (branch if lower than, etc.):

- `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 `riscv_isa.pdf`

RISCV Memory accesses instructions 1/2

- **L**oad from memory (64-bit **d**ouble word) $r_d \leftarrow Mem[r_s + off]$:

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

- **S**tore to memory:

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

- Load effective **a**ddress (shortcut)

```
la rd, label
```

See the ISA for more info.

3. The RiscV architecture in a nutshell
   - One example

Ex : Assembly code - demo

```
#simple RISCV assembly demo
#riscv64-unknown-elf-gcc  demo20.s ../../TP/TP03/riscv/libprint.s -o demo20
#spike pk demo20
        .text
        .globl main
main:
  addi    sp,sp,-16
  sd ra, 0(sp)
  sd fp, 8(sp)
# Body of the function
        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 body of the function
  ld ra, 0(sp)
  ld fp, 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

Shortcut for: addi t1, s1, 0
```

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
Shortcut for: addi t1, s1, 0
```

Format I

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
Shortcut for: addi t1, s1, 0
```

Format I

imm[11:0]	rs1	funct3	rd	opcode

Exercise: RISCV Assembly

Assemble the following instruction:

```
mv t1, s1
```

```
Shortcut for: 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
```

```
Shortcut for: 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 ⇒ 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`.

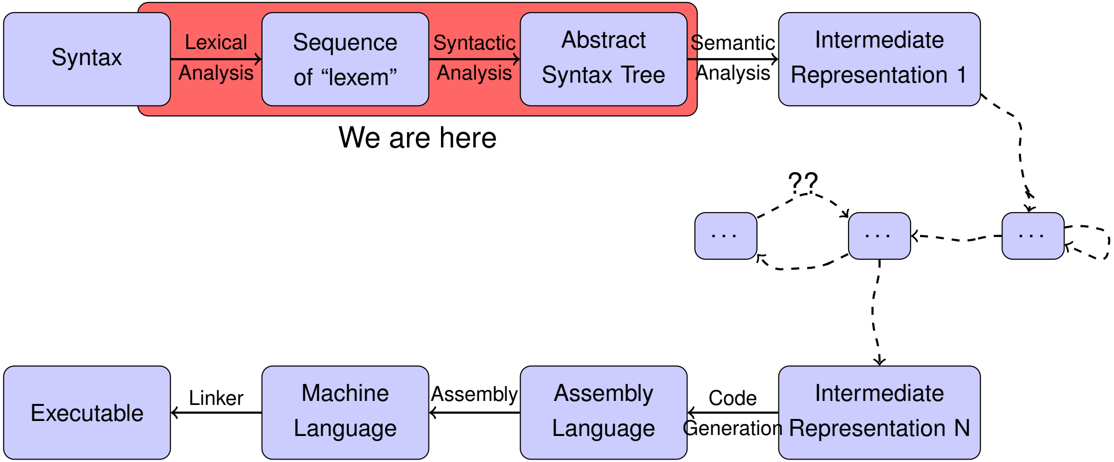
```
    .globl main
main:
    li a0, 0
lbl:
    call print_int
    call newline
    addi a0, a0, 1
    j lbl
```

CAP — Compilation and Program Analysis (#1) : Introduction – **Lexing and Parsing**

Master 1, ENS de Lyon et Dpt Info, Lyon1

2024-2025

A Standard™ Compiler Pipeline



Goal of this chapter

- Understand the syntactic 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

What for ?

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

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

- Group characters into a list of **tokens**, e.g.:
  - The word “int” stands for type integer (predefined identifier in most languages, keyword here);
  - A sequence of letters stands for a identifier (typically, a variable);
  - A sequence of digits stands for an integer literal;
  - ...

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).

Tools: lexical analyzer constructors

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

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 generator;
- ANTLR is multi-language (not only Java).

Lexing with ANTLR: example

Lexing rules:

- Must start with an upper-case letter;
- Follow 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
```

Running an ANTLR lexer (for debug)

Compilation (using the java backend)

```
$ antlr4 Toto.g4       # produces several Java files
$ javac *.java         # compiles into xx.class files
$ echo 'foo bar hello 42' | \
    java org.antlr.v4.gui.TestRig Tokens tokens -tokens
[@0,0:2='foo',<ID>,1:0]
[@1,4:6='bar',<ID>,1:4]
[@2,8:12='hello',<'hello'>,1:8]
[@3,14:15='42',<INT>,1:14]
[@4,17:16='<EOF>',<EOF>,2:0]
```

Lexer rules: quick reference

`NAME : ... ;`	rule definition (upper case for lexer)
`(...)`	grouping
`|`	alternative, e.g. `(a|b)`
`'s'`	char or string literal. `'\n'` for newline.
`.`	any character
`a .. b`	range, e.g. `('0'..'9')`
`{...}`	action
`+`	1 or more, e.g. `('0' .. '9')+`
`*`	0 or more
`?`	optional (or semantic predicate)
`[...]`	choice between characters, e.g. `[abc]`
`~[...]`	match not, e.g. `~[abc]`
`// ...`	single-line comment
`/* ... */`	multi-line comment

Lexer rules : dealing with ambiguities

A grammar with ambiguities

```
FLOAT: [0-9]+ '.' [0-9]+;
INT: [0-9]+;
DOT: '.';

IF: 'if';
ID: [a-zA-Z]+;
THEN: 'then';
```

Ambiguities:

4.2	`INT`, `DOT`, `INT` or `FLOAT` ?
if	`ID` or `IF` ?
then	`ID` or `THEN` ?

Two rules to resolve ambiguities (with ANTLR, and most lexer generators) :

1. Longest match;
2. In case of tie, first rule in the grammar file.

Lexer rules : dealing with ambiguities

A grammar with ambiguities

```
FLOAT: [0-9]+ '.' [0-9]+;
INT: [0-9]+;
DOT: '.';

IF: 'if';
ID: [a-zA-Z]+;
THEN: 'then';
```

Ambiguities:

4.2 INT, DOT, INT or FLOAT ?

if ID or IF ?

then ID or THEN ?

Two rules to resolve ambiguities (with ANTLR, and most lexer generators) :

1. Longest match;
2. In case of tie, first rule in the grammar file.

Actions in an ANTLR lexer

- Basic flow: g4 → Java/Python (“host code”) → Execution
- Alternative: embed host code into g4 file

Foo.g4

```
lexer grammar XX;
@header { // Some init (host) code...
}
@members { // Some global (host) variables
}
// More optional blocks are available
// rules with actions: code within {...} is
// inserted in the generated host code and
// executed when matching.
FOO : 'foo' {System.out.println("foo found");
     };
```

Compilation (using the java backend):

```
$ antlr4 Foo.g4
$ javac *.java
$ java org.antlr.v4.gui.TestRig \
       Foo tokens
```

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 ;
```

What's Parsing ?

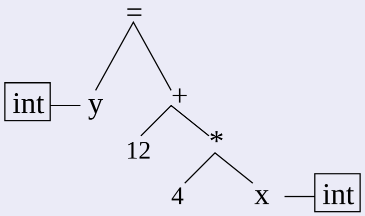
Relate tokens by structuring them.

Flat tokens

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

⇒ **Parsing** ⇒

Accept → Structured tokens



For now

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

What's behind ?

From a Context-free Grammar, produce a Pushdown Automaton[1] (already seen in L3 course?)

[1] Automate à Pile

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 $w \to w'$ 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:

$$S \rightarrow aSb$$

$$S \rightarrow \varepsilon$$

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 $N \to (\Sigma \cup N)^*$.

Example:

$$S \to S + S | S * S | a$$

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.

Tools: parser generators

- Parser generator: builds a Pushdown Automaton from a grammar definition;
- Ex: yacc (C), javacup (Java), OCamlyacc, **ANTLR**, ...
- **input** of ANTLR: a set of grammar rules with actions (`Toto.g4`);
- **output** of ANTLR: a file (`Toto.java`) that contains the corresponding Pushdown 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 applied");}
  | // nothing for empty alternative
  ;
```

Parser rules: quick reference

`name : ... ;`	: rule definition (lower-case for parsing rules)
same as lexer	: same meaning, in particular
`(...)`	: grouping
`|`	: alternative, e.g. `(a|b)`
new in parser	: rules can call each other recursively (`a: B a | ;`)

- Compile/execute with:

```
antlr4 explLexer.g4 explParser.g4
javac *.java
echo 'aabb' | java org.antlr.v4.gui.TestRig expl prog -gui
```

Parser rules: recommended format

```
// Do
rule: alternative A
    | alternative B
    | empty alternative
    ; // aligned with the |

// Don't (empty alternative hardly visible)
rule:
   | alternative A
   | alterantive B
   ;
```

Parser rules : dealing with ambiguities

A grammar with ambiguities

```
parse: expr EOF;

expr: expr '*' expr
    | expr '+' expr
    | INT
    ;

INT: [0-9]+;
WS: [ \t\n\r]+ -> skip;
```

Ambiguities:

1+2*3 (1+2)*3 or 1+(2*3) ? (precedence)

1+2+3 (1+2)+3 or 1+(2+3) ? (associativity)

ANTLR rules:

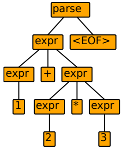
1. First alternative (`'*'` here) has highest precedence
2. Left-associative by default (customizeable, e.g. `'^'<assoc=right>`)

Parser rules : dealing with ambiguities

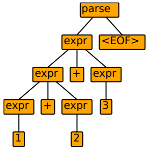
A grammar with ambiguities

```
parse: expr EOF;

expr: expr '*' expr
    | expr '+' expr
    | INT
    ;

INT: [0-9]+;
WS: [ \t\n\r]+ -> skip;
```

1+2*3 (1+2)*3 or 1+(2*3) ? (precedence)



1+2+3 (1+2)+3 or 1+(2+3) ? (associativity)



ANTLR rules:

1. First alternative ('∗' here) has highest precedence
2. Left-associative by default (customizeable, e.g. `'^'<assoc=right>`)

ANTLR4 expressivity

ALL(*) = Adaptive LL(*)

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

Further reading (SIGPLAN Notices'14 paper, T. Parr, K. Fisher)

https://www.antlr.org/papers/allstar-techreport.pdf

Left recursion

ANTLR allows left recursion (but right recursion usually more efficient):

```
a: a b;
```

But not indirect left recursion.

$$\begin{tikzcd} X_1 \arrow[r] & \ldots \arrow[r] & X_n \arrow[ll, bend right] \end{tikzcd}$$

There exist algorithms to eliminate indirect recursions.

Lists

ANTLR allows lists:

```
prog: statement+ ; // one or more statements
block: statement∗ ; // zero or more statements
```

Read the documentation!

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