



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DIPARTIMENTO DI
INFORMATICA - SCIENZA E INGEGNERIA

CORSO 72671

COMPLEMENTI DI LINGUAGGI DI PROGRAMMAZIONE

COSIMO LANEVE

`cosimo.laneve@unibo.it`

welcome

COMPLEMENTI DI LINGUAGGI DI PROGRAMMAZIONE

code transformation and analysis

CODE TRANSFORMATION AND ANALYSIS

- * course informations
- * why study CODE TRANSFORMATION AND ANALYSIS?
- * a quick history of CODE TRANSFORMATION AND ANALYSIS
- * the structure of a translator/analyzer
- * the arguments in this course
- * **ANTLR**
- * some background

COURSE INFORMATIONS

instructor: me. If you want to discuss

- * send email to `cosimo.laneve@unibo.it`
- * come in my studio during office hours
- * fix ad-hoc meetings

e-learning page: download material

- * go to the course page at

`https://www.unibo.it/sitoweb/cosimo.laneve/teachings`

this year there will be a TUTOR:

Marco Bertoni

- exercises
- assistance in the team project
`marco.bertoni5@unibo.it`

grading policies

* **written examination** max score: **26** (min **15**)

* **team project** max score: **6**

} both are necessary

EXAMINATION

- * there are 6 **written examination** sessions (2 in Jan/Feb; 3 in May/Jun/Jul; 1 in Sept.)
- * the **team** of the team project must be composed of 2 students (exceptionally 3)
- * you may release the project **within a deadline** (usually in June: we will find an agreement — miss the deadline = -1 point)
- * the score of the project is individual: **one may pass** and **another of the group may fail** (in this case a new project must be delivered)
- * the project must be uploaded on `virtuale` — **don't send the project by email!**

COURSE INFORMATIONS

course prerequisites

- * Programming Languages (code 04138)
- * **Fondamenti Logici dell'Informatica** (Laurea Magistrale, 1[^] sem), not necessary, but highly recommended

references: downloadable material

- * Torben Mogenssen: **Basics of Compiler Design**. 2010. Downloadable from <http://www.diku.dk/~torbenm/Basics/>
 - you may also download it from the e-learning website
- * Terence Parr: **Language Implementation Patterns**. 2010 (this is for ANTLR) [look for it by yourself...]
- * material on the e-learning website

CODE TRANSFORMATION AND ANALYSIS

MOTIVATIONS

modern software development requires fast and sophisticated code transformation and analysis tools

- * `Java` code is verified by the Virtual Machine before execution
- * `Facebook`, before releasing its mobile apps, always submits them to a tool that finds bugs without running the code
- * `Google Chrome` and `Mozilla Firefox` analyze and optimize `JavaScript` code to make browsers more responsive
- * performance-critical applications require compilers that derive correct and optimized machine code from high-level source code

CODE TRANSFORMATION AND ANALYSIS

ADDITIONAL MOTIVATIONS

- * it is hard because code transformations and analyses must be **semantically correct**
- * see the **application of the theory**
- * learn **how programming languages work**
- * learn **how a development tool works** and how to use it
- * focus on **concepts** that we use all the time in a **translation**: data structures, model-driven code generators, source-to-source translators, source analysers, interpreters
 - most of us will never build compilers for general-purpose programming languages (which requires a strong computer science background)

CODE TRANSFORMATION AND ANALYSIS

OBJECTIVE OF THE COURSE

discuss modern code transformation and analysis techniques and illustrate their implementation

- * we will refer to the **ANTLR** framework that is widely used in academia and industry to build all sorts of languages, tools, and frameworks
- * **Twitter** search uses **ANTLR** for query parsing, with over 2 billion queries a day
- * you will apply the theory by extending a small, yet expressive and powerful language, by means of the **ANTLR** framework

CLP IS HARD

the average score of the last year is **25.2**

the course combines theory and practice

- * therefore you need to know in detail the theory and the development tool to write the sw
- * e.g. the **grammars and the syntax trees** and the **visitor process**; the **semantic rules and the proof trees** and **their implementation**, the **code generation** and the **assembly language**

CLP IS HARD

CLP requires a strong background (and motivation)

comments from students:

- * Tutto il corso è un problema. In particolar modo non si può alla magistrale di informatica essere obbligati a fare questo esame in quanto dovrei poter scegliere cosa vorrei fare. Mi pento di non essermene andato a Milano a studiare dove forse sarei stato valorizzato di più e avrei fatto qualcosa di più interessante. Il docente almeno qualche materiale di supporto in più potrebbe fornirlo dimentica sempre tutto e non ci dà sicurezza sulla struttura dell'esame poco chiara
- * Tutto è piuttosto difficile da capire
- * Le prime lezioni sono state difficili da capire se non si avevano già conoscenze di questa materia. Le slide non sono molto chiare. Fornire più conoscenze di base e dare meno cose per scontate. Migliorare le slide. Fare più esercizi completi.
- * Gli esercizi svolti in classe sono subito esercizi d'esame o simili, sarebbe meglio una difficoltà più graduale nella presentazione degli esercizi in modo da consolidare meglio i concetti.
- * Troppo difficili gli argomenti del corso per studenti che alla triennale non hanno avuto corsi, i quali li permettono di avere una base già solida su questi argomenti

CLP IS HARD

if you are not strongly motivated, consider to ask for

variazione piano di studi

you may replace **CLP** with one of these

- * Sistemi Context-Aware
- * Didattica dell'Informatica
- * Internet of Things
- * Artificial Intelligence, Blockchain e Criptovalute nello sviluppo software
- * Human Data Science

as a motivation you may write one of these

- * La scelta del corso X deriva dal desiderio di approfondire le competenze dell'area Y
- * La scelta del corso X deriva dalla volontà di aumentare i possibili sbocchi professionali del mio percorso formativo anche con esso.

A SHORT HISTORY OF CODE TRANSFORMATION & ANALYSIS

- * **first**, there was nothing
- * **then**, there was machine code
- * **afterwards**, there were assembly languages (see Computer Architecture, code 11925)
- * **programming was expensive**: 50% of costs for machines went into programming
- * **high-level languages were/are conceived**: people needs translators to low-level codes (compilers and interpreters)
- * **commercial and critical codes** require powerful translators removing bugs and delivering optimized code

CODE TRANSFORMATIONS: COMPILATION AND INTERPRETATION

COMPILERS

- * transform code written in a high-level programming language into the machine code, at once, before program runs,
- * compiled code runs faster
- * **display all errors after compilation**
- * take an entire program

INTERPRETERS

- * covert each high-level program statement, one by one, into the machine code, during program run
- * interpreted code runs slower
- * **display errors of each line one by one**
- * take a single line of code

* **compiled languages:** C, C++, C#, Scala, Java

- Java is compiled into bytecode which is then interpreted by the Java Virtual Machine

* **interpreted languages:** PHP, Perl, Ruby, JavaScript, Python

- Python is compiled into bytecode which is then saved and executed afterwards instead of the source code

COMPILERS AND INTERPRETERS: JAVA

```
2  
3  
4  
5- int fact(int n) {  
6-     if (n==0) return 1;  
7-     else {  
8-         int m = n-1;  
9-         return n*fact(m);  
10-     }  
11- }  
12-  
13-  
14- public static void main(String[] args) {  
15-     int a = fact(10);  
16- }
```



*lexical
analysis*

identify logical pieces of the source code



*syntactic
analysis*

identify how these pieces relate to each other



*semantic
analysis*

identify whether pieces relate to each other in a correct way



*bytecode
generation*

generate code that is adequate for the Java Virtual Machine

output



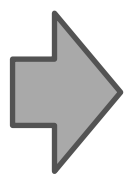
JVM **bytecode** is interpreted by the machine
– c.f. JVM plugin in the browser

COMPILERS: C (NO INTERPRETERS)

```
1 // fact.c
2
3 int fact(int n) {
4     if (n==0) return 1;
5     else {
6         int r = n * fact(n-1);
7         return r;
8     }
9 }
10
11 int main() {
12     int a = fact(100);
13     printf("%d", a);
14 }
```

```
00000000: push    %ebp
00000001: mov     %esp, %ebp
00000002: mov     %eax, [%ebp-arg 0]
00000003: pop     %ebp
00000004: movzx  %eax, %eax
00000005: test   %eax, [%eax+4]
00000006: jnz    %eax, [%eax+4]
00000007: call   fact(%eax)
00000008: mul    %eax, %eax
00000009: jmp    %eax, [%eax+4]
0000000a: ret    4
0000000b:
0000000c:
0000000d:
0000000e:
0000000f:
00000010:
00000011:
00000012:
00000013:
00000014:
00000015:
00000016:
00000017:
00000018:
00000019:
0000001a:
0000001b:
0000001c:
0000001d:
0000001e:
0000001f:
00000020:
00000021:
00000022:
00000023:
00000024:
00000025:
00000026:
00000027:
00000028:
00000029:
0000002a:
0000002b:
0000002c:
0000002d:
0000002e:
0000002f:
00000030:
00000031:
00000032:
00000033:
00000034:
00000035:
00000036:
00000037:
00000038:
00000039:
0000003a:
0000003b:
0000003c:
0000003d:
0000003e:
0000003f:
00000040:
00000041:
00000042:
00000043:
00000044:
00000045:
00000046:
00000047:
00000048:
00000049:
0000004a:
0000004b:
0000004c:
0000004d:
0000004e:
0000004f:
00000050:
00000051:
00000052:
00000053:
00000054:
00000055:
00000056:
00000057:
00000058:
00000059:
0000005a:
0000005b:
0000005c:
0000005d:
0000005e:
0000005f:
00000060:
00000061:
00000062:
00000063:
00000064:
00000065:
00000066:
00000067:
00000068:
00000069:
0000006a:
0000006b:
0000006c:
0000006d:
0000006e:
0000006f:
00000070:
00000071:
00000072:
00000073:
00000074:
00000075:
00000076:
00000077:
00000078:
00000079:
0000007a:
0000007b:
0000007c:
0000007d:
0000007e:
0000007f:
00000080:
00000081:
00000082:
00000083:
00000084:
00000085:
00000086:
00000087:
00000088:
00000089:
0000008a:
0000008b:
0000008c:
0000008d:
0000008e:
0000008f:
00000090:
00000091:
00000092:
00000093:
00000094:
00000095:
00000096:
00000097:
00000098:
00000099:
0000009a:
0000009b:
0000009c:
0000009d:
0000009e:
0000009f:
000000a0:
000000a1:
000000a2:
000000a3:
000000a4:
000000a5:
000000a6:
000000a7:
000000a8:
000000a9:
000000aa:
000000ab:
000000ac:
000000ad:
000000ae:
000000af:
000000b0:
000000b1:
000000b2:
000000b3:
000000b4:
000000b5:
000000b6:
000000b7:
000000b8:
000000b9:
000000ba:
000000bb:
000000bc:
000000bd:
000000be:
000000bf:
000000c0:
000000c1:
000000c2:
000000c3:
000000c4:
000000c5:
000000c6:
000000c7:
000000c8:
000000c9:
000000ca:
000000cb:
000000cc:
000000cd:
000000ce:
000000cf:
000000d0:
000000d1:
000000d2:
000000d3:
000000d4:
000000d5:
000000d6:
000000d7:
000000d8:
000000d9:
000000da:
000000db:
000000dc:
000000dd:
000000de:
000000df:
000000e0:
000000e1:
000000e2:
000000e3:
000000e4:
000000e5:
000000e6:
000000e7:
000000e8:
000000e9:
000000ea:
000000eb:
000000ec:
000000ed:
000000ee:
000000ef:
000000f0:
000000f1:
000000f2:
000000f3:
000000f4:
000000f5:
000000f6:
000000f7:
000000f8:
000000f9:
000000fa:
000000fb:
000000fc:
000000fd:
000000fe:
000000ff:
00000100:
```

machine code



lexical analysis

identify logical pieces of the source code



syntactic analysis

identify how these pieces relate to each other



semantic analysis

identify whether pieces relate to each other in a correct way



intermediate code generation

generate code that is adequate for an abstract machine



machine code generation



intermediate code optimisation

A COMPILER AT WORK

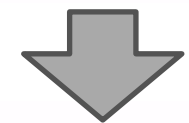
```
while (y < z) {  
    int x = a + b;  
    y = y + x;  
}
```



lexical
analysis



syntactic
analysis



semantic
analysis



intermediate
code
generation

A COMPILER AT WORK

```
while (y < z) {  
    int x = a + b;  
    y = y + x;  
}
```

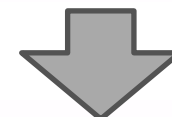
```
T_While  
T_LeftParen  
T_Identifier y  
T_Less  
T_Identifier z  
T_RightParen  
T_OpenBrace  
T_Int  
T_Identifier x  
T_Assign  
T_Identifier a  
T_Plus  
T_Identifier b  
T_Semicolon  
T_Identifier y  
T_Assign  
T_Identifier y  
T_Plus  
T_Identifier x  
T_Semicolon  
T_CloseBrace
```



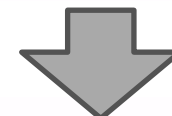
lexical
analysis



syntactic
analysis



semantic
analysis

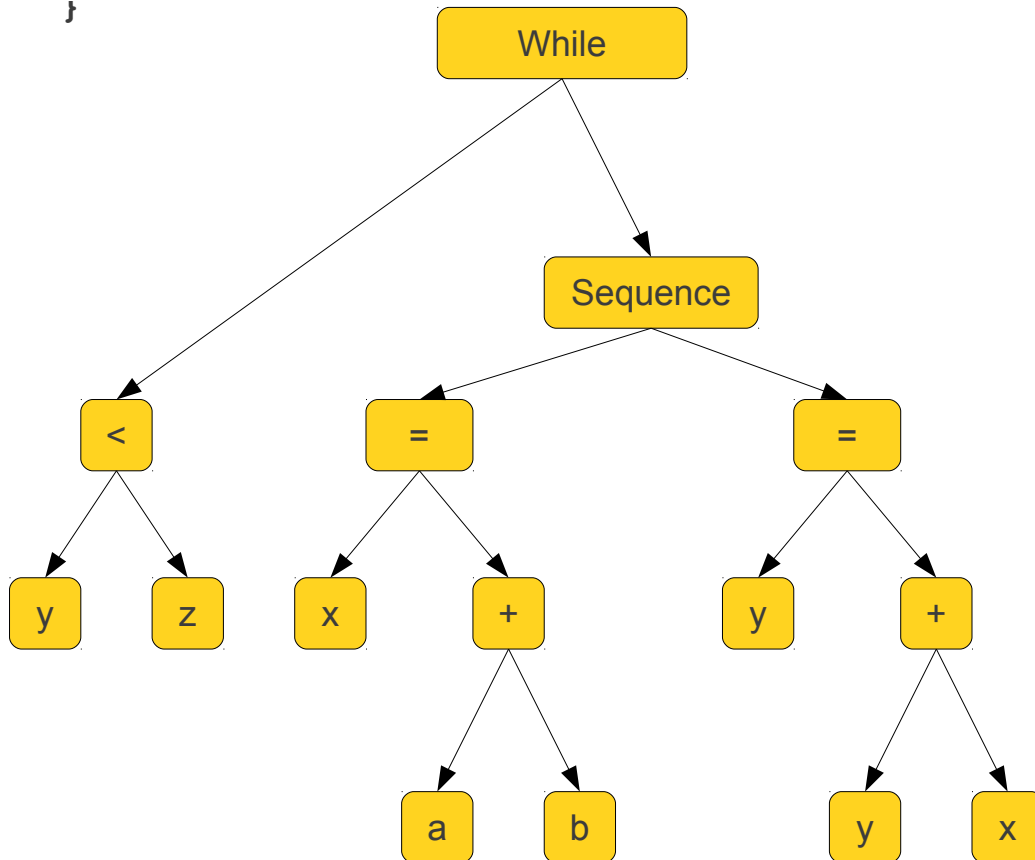


intermediate
code
generation

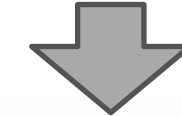
A COMPILER AT WORK

```
while (y < z) {  
    int x = a + b;  
    y = y + x;  
}
```

```
T_While  
T_LeftParen  
T_Identifier y  
T_Less
```



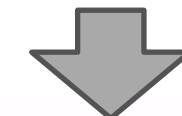
*lexical
analysis*



*syntactic
analysis*



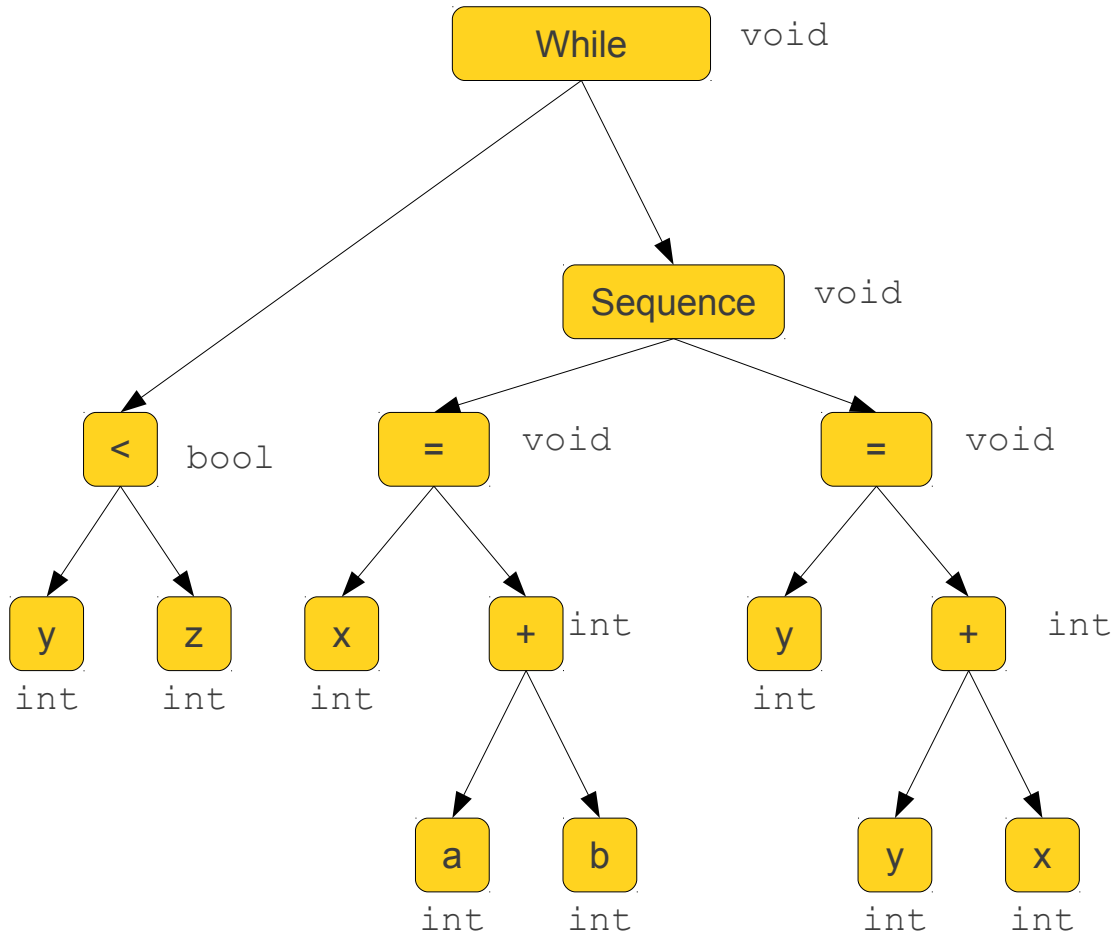
*semantic
analysis*



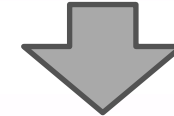
*intermediate
code
generation*

A COMPILER AT WORK

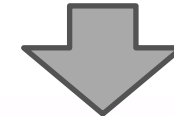
```
while (y < z) {  
    int x = a + b;  
    y = y + x;  
}
```



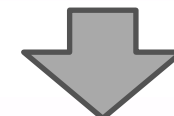
lexical
analysis



syntactic
analysis



semantic
analysis



intermediate
code
generation

A COMPILER AT WORK & THE JVM INTERPRETER

```
while (y < z) {  
    int x = a + b;  
    y = y + x;  
}
```

```
Loop:  _t1 = y < z  
       iffalse _t1 goto Exit  
       x = a + b  
       y = x+y  
       goto Loop
```

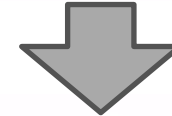
```
Exit:
```

this bytecode is given in input to the JVM which

- * interprets every instruction
- * executes it on the machine



lexical
analysis



syntactic
analysis



semantic
analysis



intermediate
code
generation

A COMPILER AT WORK (IN C LANGUAGES)

```
while (y < z) {  
    int x = a + b;  
    y = y + x;  
}
```

```
    x = a + b  
Loop:  _t1 = y < z  
       iffalse _t1 goto Exit  
       y = x+y  
       goto Loop  
Exit:
```

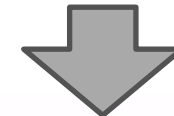
optimizations produce better code



*intermediate
code
generation*



*intermediate
code
optimisation*



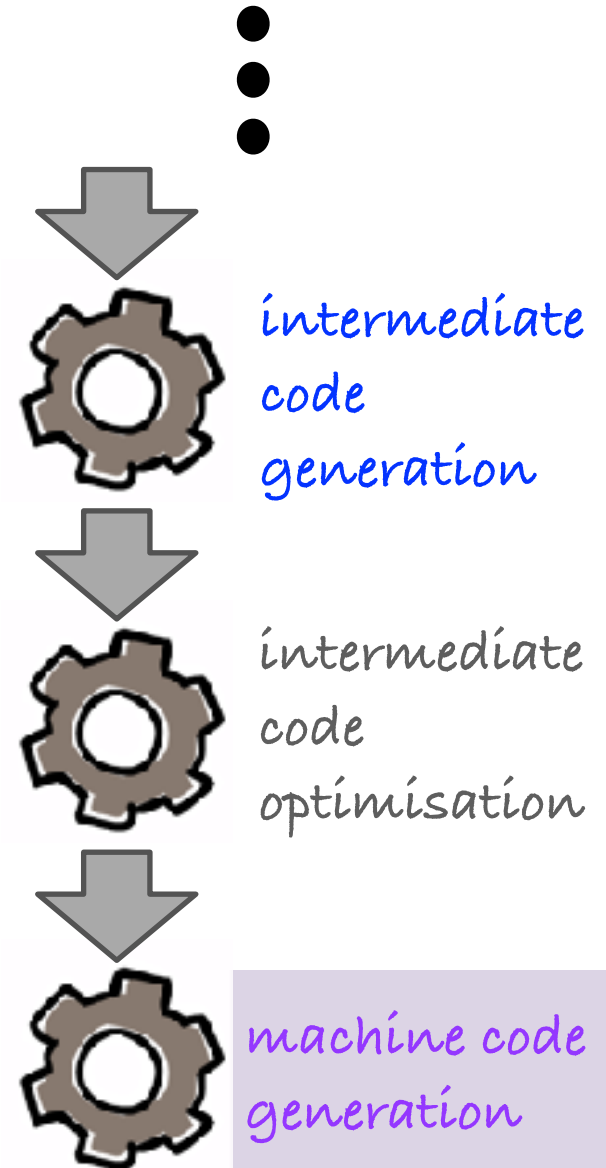
*machine code
generation*

A COMPILER AT WORK (IN C LANGUAGES)

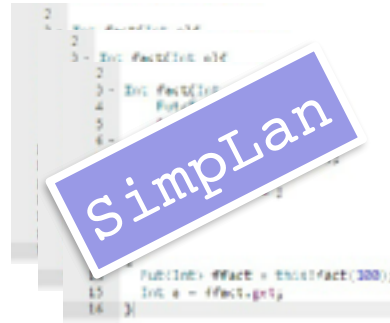
```
while (y < z) {  
    int x = a + b;  
    y = y + x;  
}
```

```
    add $1, $2, $3  
Loop: slt $6, $4, $5  
      beq $6, Exit  
      add $4, $1, $4  
      b Loop  
Exit:
```

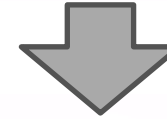
build the code for your machine



THIS COURSE



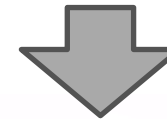
*lexical
analysis*



*syntactic
analysis*



*semantic
analysis*



*intermediate
code
generation*



the **SimpLan**
interpreter

we will **build a compiler** for a simple programming language — **SimpLan** — and we will build an interpreter for executing **SimpLan**

as **team project** you must

- * develop a compiler for an extension of **SimpLan**
- * use the ANTLR development tool

RECAPS ABOUT GRAMMARS

- * see Torben Mogensen: **Basics of Compiler Design**,
chapter 3, section 1, 2 and 3

RECAPS ABOUT GRAMMARS

Definition: context-free grammar

A **context-free grammar** is a tuple $(\mathbf{N}, \mathbf{T}, \rightarrow, S)$ where

- \mathbf{N} is a finite set of **non-terminal symbols**
- \mathbf{T} is a finite set of **terminal symbols**
- \rightarrow is a finite set of **productions** of type

$$A \rightarrow \alpha_1 \dots \alpha_n \quad \text{with } A \in \mathbf{N} \text{ and } \alpha_1 \dots \alpha_n \in \mathbf{N} \cup \mathbf{T}$$

- $S \in \mathbf{N}$ is called **initial symbol**

example:

$$\text{BExp} \rightarrow (\text{BExp})$$
$$\text{BExp} \rightarrow \text{Digit}$$
$$\text{Digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$$

compact syntax that represents
10 productions

we will always keep the tuple **IMPLICIT!**

FORMAL NOTIONS

Definition: derivations

Let $G = (\mathbf{N}, \mathbf{T}, \rightarrow, S)$ be a context-free grammar and γ and δ be sequences of symbols in $\mathbf{N} \cup \mathbf{T}$. A **one-step derivation** of G is

$$\gamma A \delta \rightarrow \gamma \alpha_1 \dots \alpha_n \delta$$

where $A \rightarrow \alpha_1 \dots \alpha_n \in \rightarrow$

notation: \rightarrow^* (0 or more steps) \rightarrow^+ (1 or more steps)

$\gamma \rightarrow^* \delta$ is called **derivation**

Definition: language generated by a context-free grammar

The **language (generated)** by $G = (\mathbf{N}, \mathbf{T}, \rightarrow, S)$ is the set

$$\mathcal{L}(G) = \{ \gamma \mid \gamma \in \mathbf{T}^* \text{ e } S \rightarrow^+ \gamma \}$$

\mathbf{T}^* is the **Kleene closure**: every sequence of symbols in \mathbf{T}

example: if $\mathbf{T} = \{a, b\}$, $\mathbf{T}^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$

DERIVATIONS — EXAMPLES

take the grammar

$$\text{BExp} \rightarrow (\text{BExp})$$
$$\text{BExp} \rightarrow \text{Digit}$$
$$\text{Digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$$
$$\text{BExp} \rightarrow (\text{BExp}) \rightarrow ((\text{BExp})) \rightarrow ((\text{Digit})) \rightarrow ((1))$$

* is a **derivation**

* the **sequence of terminals** $((1))$ belongs to the language generated by the grammar

* the sequences $((9))$ e $((((1))))$ and 3 **belong** to the language, as well — what are the derivations?

* the sequences $((\text{BExp}))$ and $((((10))))$ and $((3))$ **do not belong to** the language — why?

DERIVATIONS — LEFTMOST AND RIGHTMOST

a **leftmost derivation** is a derivation where the non-terminal symbol that is replaced every time is the **leftmost** one

rightmost/the rightmost one

example

$\text{Exp} \rightarrow \text{Exp} - \text{Exp}$

$\text{Exp} \rightarrow \text{Digit}$

$\text{Digit} \rightarrow 0 \mid 1 \mid \dots \mid 9$

I am replacing this Exp

$\text{Exp} \rightarrow \text{Exp} - \text{Exp} \rightarrow \text{Exp} - \text{Exp} - \text{Exp}$

$\rightarrow \text{Digit} - \text{Exp} - \text{Exp} \rightarrow 3 - \text{Exp} - \text{Exp} \rightarrow 3 - \text{Digit} - \text{Exp}$

$\rightarrow 3 - 2 - \text{Exp} \rightarrow 3 - 2 - \text{Digit} \rightarrow 3 - 2 - 1$

is a **leftmost derivation**

note: $\text{Exp} \rightarrow \text{Exp} - \text{Exp} \rightarrow \text{Digit} - \text{Exp} \rightarrow 3 - \text{Exp}$

$\rightarrow 3 - \text{Exp} - \text{Exp} \rightarrow 3 - \text{Digit} - \text{Exp}$

$\rightarrow 3 - 2 - \text{Exp} \rightarrow 3_0 - 2 - \text{Digit} \rightarrow 3 - 2 - 1$

this is also a
leftmost derivation!

ANTLR

ANTLR = ANother Tool for Language Recognition

- * is a powerful parser generator for reading, processing, executing, or translating structured text or binary files
- * it's widely used to build languages, tools, and frameworks
- * from a grammar, **ANTLR** generates a parser that can build and walk parse trees

ANTLR

you need

- * Eclipse/IntelliJ
- * ANTLR plugin

in the e-learning website, there is a folder (see Argomento 2) where

- * you can find installation informations about **ANTLR**
- * a simple example of what you can do with **ANTLR**

to play, you can use the online tool:

<http://lab.antlr.org/>

ANTLR — AN INITIAL EXAMPLE

grammar file has suffix g4

Arrayofint.g4

the syntax

```
grammar ArrayofInt ;
```

this is the first line; the name is the same of the file without suffix

```
init : '{' value (',' value)* '}' ;
```

these are the parser productions

- nonterminal begins with lowercase letter
- the first production identifies the initial symbol

```
value : init | INT ;
```

```
INT : [0-9]+ ;
```

this is a lexer production

- nonterminal are in capital letter
- rhs are always regular expressions

ANTLR — AN INITIAL EXAMPLE

programs:

* { 12, 245, 3327 }

* { 1, { 1, 2, 3 }, 3 }

ANTLR — THE ANALYSIS

ANTLR returns the syntax tree

- * you may compute the sum of integers
- * the number of integers
- * the maximal nesting
- * ...

example: { 1, { 1, 2, 3 }, 3 }

- * the sum is 10
- * the number of integers is 5
- * the maximal nesting is 2

NEXT LECTURE

