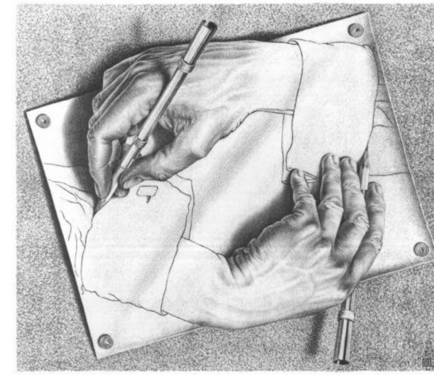


Drawing Hands
M.C. Escher, 1948

http://lara.epfl.ch/cc

Compiler Construction

Staff:

- Viktor Kuncak – Lectures
- Etienne Kneuss – Labs
- Eva Darulova – Exercises
- Yvette Gallay – secretary

Example: **javac**

- from Java to Bytecode

```
...
while (i < 10) {
  System.out.println(j);
  i = i + 1;
  j = j + 2*i+1;
}
...
```

javac Test.java
→ Test.class
javap –c Test

```
4: iload_1
5: bipush 10
7: if_icmpge 32
10: getstatic #2; //System.out
13: iload_2
14: invokevirtual #3; //println
17: iload_1
18: iconst_1
19: iadd
20: istore_1
21: iload_2
22: iconst_2
23: iload_1
24: imul
25: iadd
26: iconst_1
27: iadd
28: istore_2
29: goto 4
32: return
```

You will build a compiler that generates such code

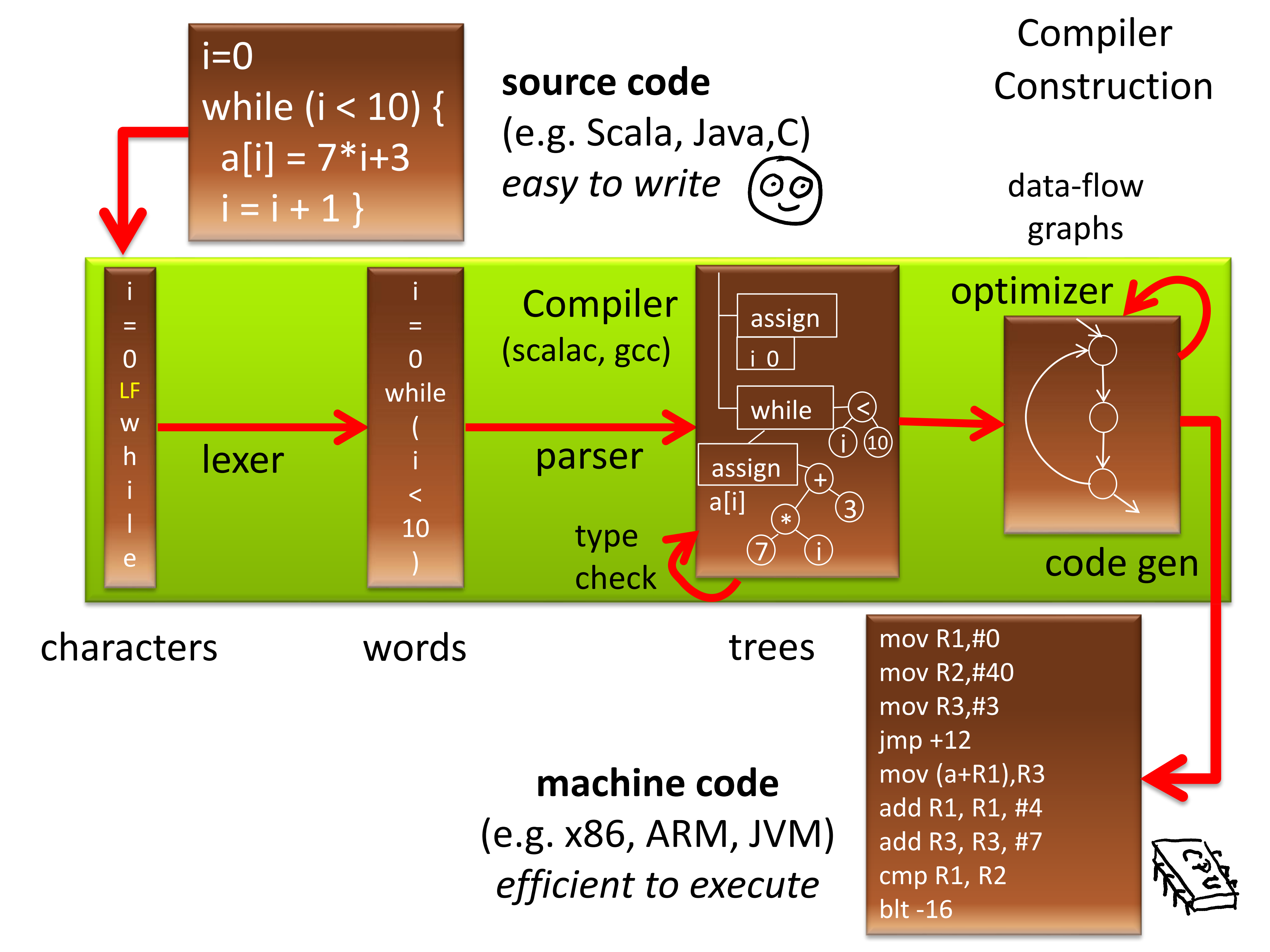
Example: **gcc**

- from C to Intel x86

```
#include <stdio.h>
int main(void) {
  int i = 0;
  int j = 0;
  while (i < 10) {
    printf("%d\n", j);
    i = i + 1;
    j = j + 2*i+1;
  }
}
```

gcc test.c –S
→ test.s

```
        jmp .L2
.L3:    movl -8(%ebp), %eax
        movl %eax, 4(%esp)
        movl $.LC0, (%esp)
        call printf
        addl $1, -12(%ebp)
        movl -12(%ebp), %eax
        addl %eax, %eax
        addl -8(%ebp), %eax
        addl $1, %eax
        movl %eax, -8(%ebp)
.L2:
        cmpl $9, -12(%ebp)
        jle .L3
```


Compiler
Construction
i=0
while (i < 10) {
a[i] = 7*i+3
i = i + 1 }
source code
(e.g. Scala, Java,C)
easy to write
data-flow
graphs
i = 0 LF w h i l e
lexer
i = 0 while (i < 10)
Compiler
(scalac, gcc)
parser
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
type
check
optimizer
code gen
characters
words
trees
mov R1,#0
mov R2,#40
mov R3,#3
jmp +12
mov (a+R1),R3
add R1, R1, #4
add R3, R3, #7
cmp R1, R2
blt -16
machine code
(e.g. x86, ARM, JVM)
efficient to execute

Compilers are Important

Source code (e.g. Scala, Java, C, C++, Python) – designed to be easy for programmers to use

- should correspond to way programmers think
- help them be productive: avoid errors, write at a higher level, use abstractions, interfaces

Target code (e.g. x86, arm, JVM, .NET) – designed to efficiently run on hardware / VM

- fast, low-power, compact, low-level

Compilers bridge these two worlds, they are essential for building complex software

NATIONAL PHYSICAL LABORATORY

TEDDINGTON, MIDDLESEX, ENGLAND

PAPER 2-3

AUTOMATIC PROGRAMMING

PROPERTIES AND PERFORMANCE OF

FORTRAN SYSTEMS I AND II

by

J. W. BACKUS

To be presented at a Symposium on
The Mechanization of Thought Processes,
which will be held at the National Physical Laboratory, Teddington, Middlesex, from 24th-27th November 1958. The papers and the discussions are to be published by H.M.S.O. in the Proceedings of the Symposium. This paper should not be reproduced without the permission of the author and of the Secretary, National Physical Laboratory.

A pioneering compiler: **FORTRAN** (FORmula TRANslator)

Backus-Naur Form - BNF

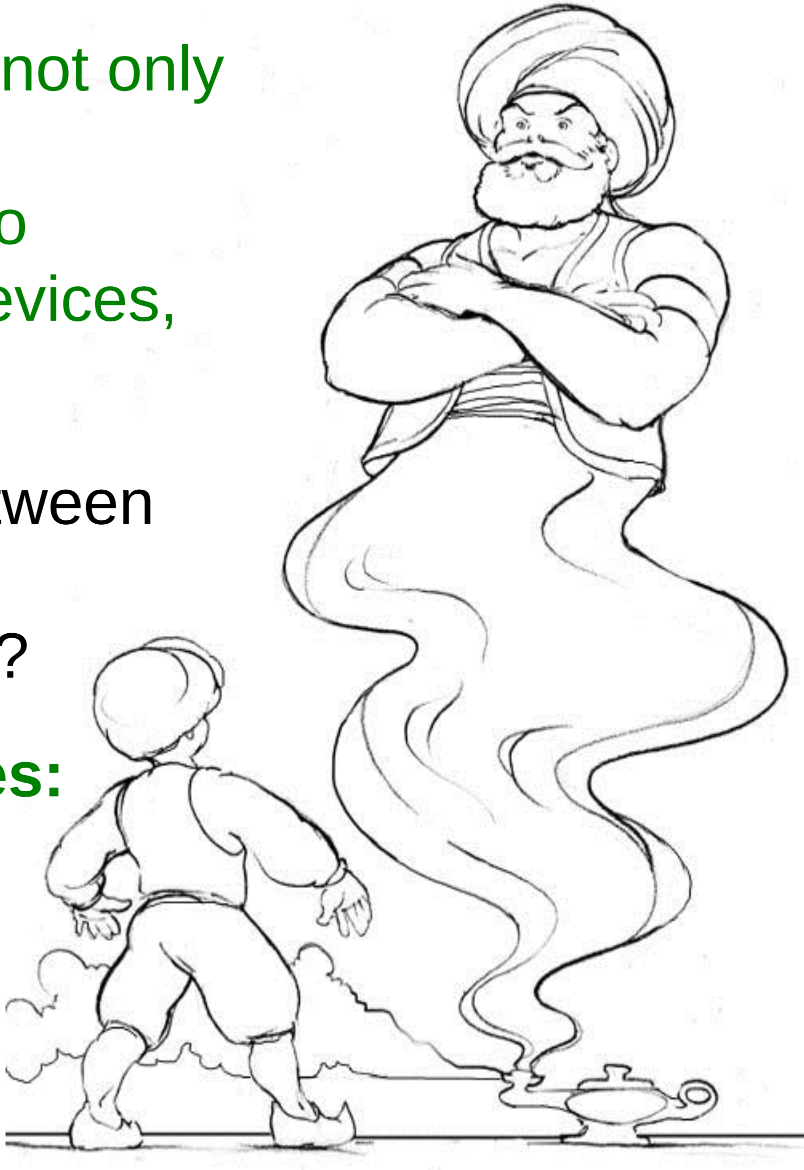
Turing Award 1977

Challenges for Future

Can target code **commands** include not only execution of commands on standard microprocessors processors, but also automatic design of new hardware devices, and control of physical devices?

Can compilers bridge the gap between wishes and commands, and help humans make the right decisions?

Can source code programs be **wishes:** specification languages, math, natural language phrases, diagrams, other forms of communication closer to engineers and users?



Some of Topics You Learn in Course

- Develop a compiler for a Java-like language
  - Write a compiler from start to end
  - Generates Java Virtual Machine (JVM) code

  (We provide you code stubs, libraries **in Scala)**
- Compiler generators – using and making them
- Analyze complex text
  - Automata, regular expressions, grammars, parsing
- Automatically detecting errors in code
  - name resolution, type checking, data-flow analysis
- Machine-like code generation

Potential Uses of Knowledge Gained

- understand how compilers work, use them better
- gain experience with building complex software
- build compiler for your next great language
- extend language with a new construct you need
- adapt existing compiler to new target platform (e.g. embedded CPU or graphics processor)
- regular expression handling in editors, grep
- build an XML parsing library
- process complex input box in an application (e.g. expression evaluator)
- parse simple natural language fragments

Schedule and Activities (6 credits)

- All activities take place in INM 202
  - Mondays **10:15**-12:00,
  - Wednesday **8:15**-10:00 and continuing to:
  - Wednesday **10:15**-12:00
- Lectures, Labs, Exercises
- At home
  - Continue with programming the compiler
  - Write the homework
- If you need more help, email us:
  - we will arrange additional meetings

How We Compute Your Grade

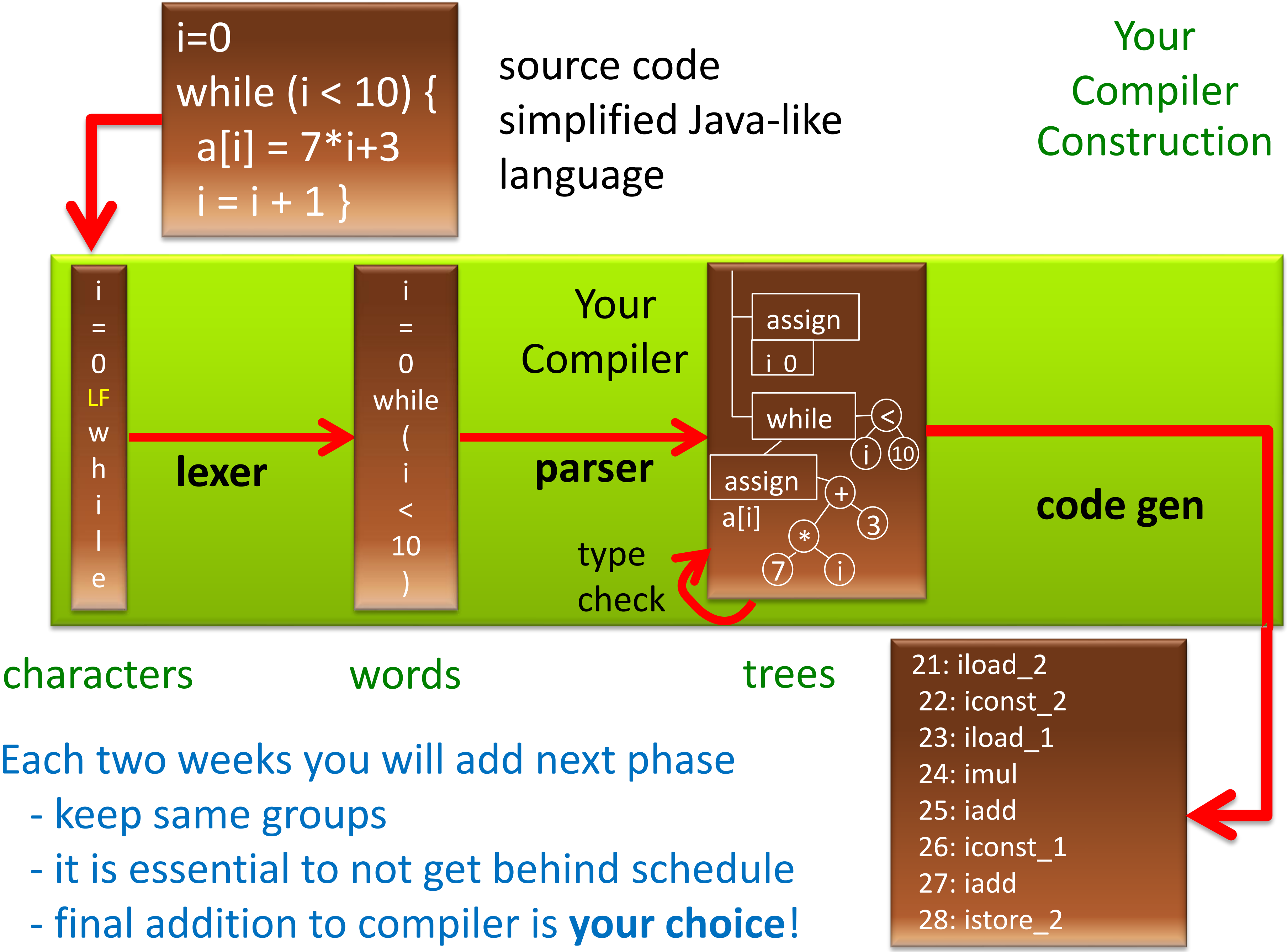
- 55% : project (submit, explain if requested)
  - submit through our wonderful online system
  - do them in groups of 2, exceptionally 1 or 3
- 20% : homework in the first part of the course
  - do them *individually*!
  - submit at the beginning of next exercise
  - participate in exercise sessions
- 25% : quiz in the last week of classes
  - will be on the last Wednesday of classes
  - do it *individually*

- Must get > 60% from *each* category to get 4.0

Collaboration and Its Boundaries

- For clarification questions, discuss them in the mailing list, which we monitor
- Work in groups of **2** for project
  - everyone should know every part of code
  - we may ask you to explain specific parts of code
- Do not copy lab solutions from other groups!
  - we use code plagiarism detection tools
  - we will check if you fully understand your code
- Do the homework and quiz ***individually***
  - You wouldn’t steal a handbag.
  - You wouldn’t steal a car.
  - You wouldn’t steal a compiler or homework!

Your Compiler Construction



Each two weeks you will add next phase

- keep same groups
- it is essential to not get behind schedule
- final addition to compiler is **your choice**!

EPFL Course Dependencies

- Theoretical Computer Science (CS-251)
  - If have not taken it, check the book "Introduction to the Theory of Computation" by Michael Sipser
- Knowledge of the Scala language (see web)
- Helpful general background
  - Discrete structures (CS-150), Algorithms (CS-250)
- This course provides background for MSc:
  - Advanced Compilers
  - Synthesis Analysis & Verification
  - Foundations of Software

Course Materials

Official Textbook:

Andrew W. Appel, Jens Palsberg:
Modern Compiler Implementation in Java
(2nd Edition). Cambridge University Press, 2002

We do not strictly follow it

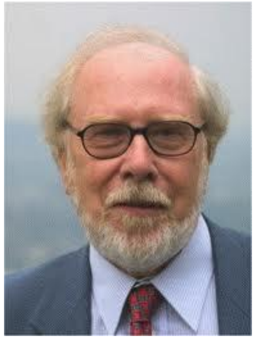
- program in Scala instead of Java
- use pattern matching instead of visitors
- hand-written parsers in the project
  (instead of using a parser generator)

Lectures in course wiki: http://lara.epfl.ch/w/cc

Additional Materials

- Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
  - comprehensive

- Compiler Construction by Niklaus Wirth
  - concise, has main ideas



“**Niklaus Emil Wirth** (born February 15, 1934) is a Swiss computer scientist, best known for designing several programming languages, including Pascal, and for pioneering several classic topics in software engineering. In 1984 he won the Turing Award for developing a sequence of innovative computer languages.”

- Additional recent books (2011-2012):
  - **Aarne Ranta**: Implementing Programming Languages
  - **H.Seidl, R.Wilhelm, S.Haack**: Compiler Design (3 vols, Springer)

Describing the **Syntax** of Languages

Syntax (from Wikipedia)

...In linguistics, **syntax** (from Ancient Greek σύνταξις "arrangement" from σύν - *syn*, "together", and τάξις - *táxis*, "an ordering") is the study of the principles and rules for constructing phrases and sentences in natural languages.

...In computer science, the **syntax** of a programming language is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language.

Describing Syntax: Why

- Goal: document precisely (a superset of) meaningful programs (for users, implementors)
  - Programs outside the superset: meaningless
  - We say programs inside make *syntactic* sense

    (They may still be ‘wrong’ in a deeper sense)
- Describing syntactically valid programs
  - There exist arbitrarily long valid programs, we cannot list all of them explicitly!
  - Informal English descriptions are imprecise, cannot use them as language reference

Describing Syntax: How

- Use theory of formal languages (from TCS)
  - regular expressions & finite automata
  - context-free grammars
- We can use such precise descriptions to
  - document what each compiler should support
  - manually derive compiler phases (lexer, parser)
  - automatically construct these phases using compiler generating tools
- We illustrate this through an example

While Language – Idea

- Small language used to illustrate key concepts
- Also used in your first lab – interpreter
  - later labs will use a more complex language
  - we continue to use *while* in lectures
- ‘while’ and ‘if’ are the control statements
  - no procedures, no exceptions
- the only variables are of ‘int’ type
  - no variable declarations, they are initially zero
  - no objects, pointers, arrays

While Language – Example Programs

```
while (i < 100) {
  j = i + 1;
  while (j < 100) {
    println(“ “,i);
    println(“,”,j);
    j = j + 1;
  }
  i = i + 1;
}
```

Nested loop

```
x = 13;
while (x > 1) {
  println("x=", x);
  if (x % 2 == 0) {
    x = x / 2;
  } else {
    x = 3 * x + 1;
  }
}
```

Does the program terminate for every initial value of x? (Collatz conjecture - open)

Even though it is simple, while is Turing-complete.

Reasons for Unbounded Program Length

constants of any length

variable names of any length

String constants of any length

(words - tokens)

nesting of expressions

nesting of statements

```
while (i < 100) {
 j = i + 5*(j + 2*(k + 7*(j+k) + i));
 while (293847329 > j) {
   while (k < 100) {
    someName42a = someName42a + k;
    k = k + i + j;
    println("Nice number", k)
   }
  }
}
```

Tokens (Words) of the *While* Language

Ident ::=

letter (letter | digit)*

or

repetition

regular expressions

integerConst ::=

digit digit*

stringConst ::=

" AnySymbolExceptQuote* "

keywords

if else while println

special symbols

( ) && < == + - * / % ! - { } ; ,

letter ::= a | b | c | … | z | A | B | C | … | Z

digit ::= 0 | 1 | … | 8 | 9

Double Floating Point Constants

Different rules in different languages

1) digit digit* [.] [digit digit*]

5.

( )?

2) digit digit* [. digit digit *]

5

5) digit* [.digit*]

3) digit* . digit digit*

.5

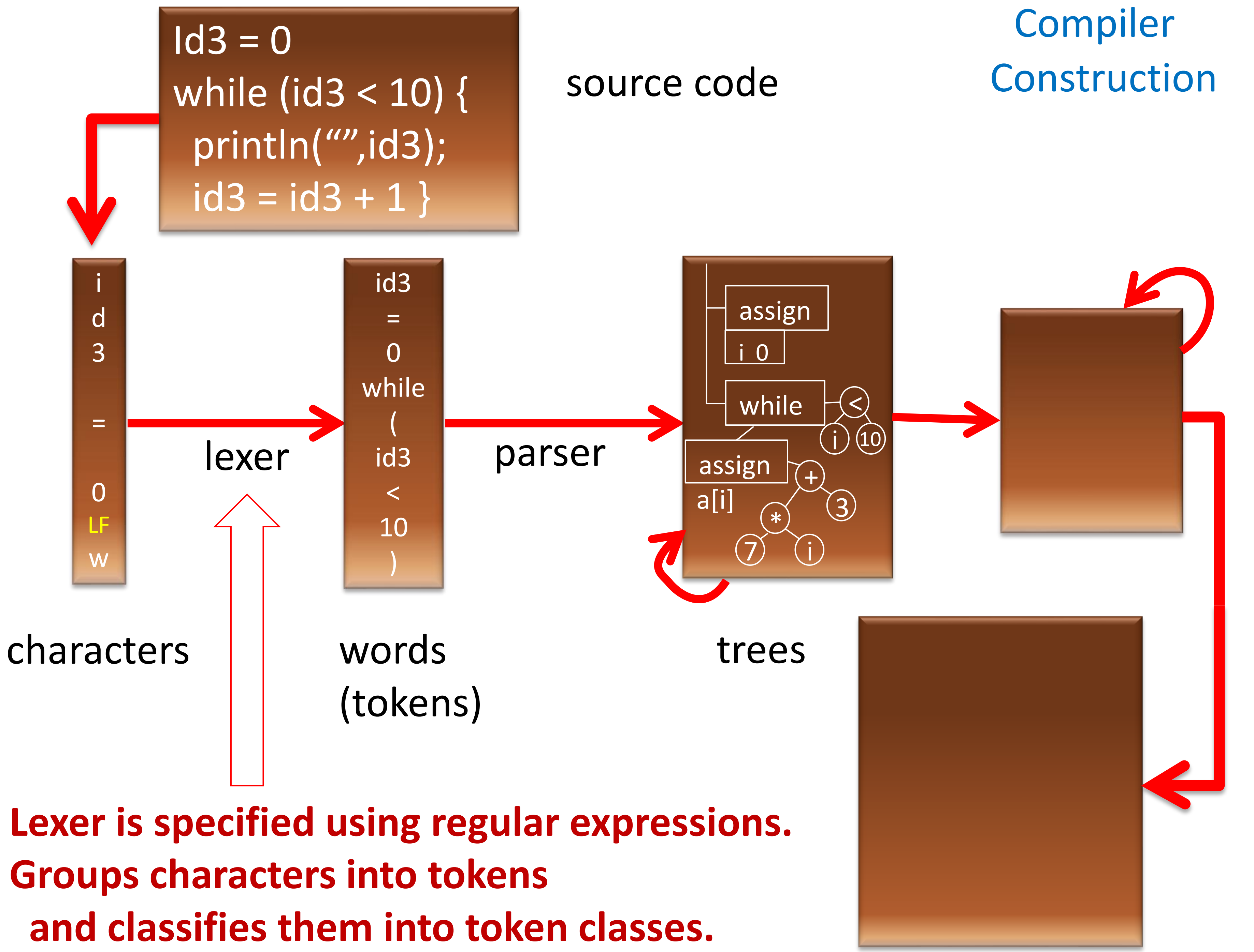
4) digit digit* . digit digit*

1.2

Identifiers

```
while (i < 100) {
  j = i + 5*(j + 2*(k + 7*(j+k) + i));
  while (293847329 > j) {
    while (k < 100)    {
      someName42a = someName42a + k;
      k = k + i + j;
      println("Nice number", k)
    }
  }
}
```

letter (letter | digit)*



Lexer is specified using regular expressions.
Groups characters into tokens
and classifies them into token classes.

More Reasons for Unbounded Length

nesting of expressions

constants of any length

```
while (i < 100) {
  j = i + 5*(j + 2*(k + 7*(j+k) + i));
  while (293847329847 > j) {
    while (k < 100) {
      someName42a = someName42a + k;
      k = k + i + j;
      println("Nice number", k)
    }
  }
}
```

nesting of statements

variable names of any length

String constants of any length

(words - tokens)

(sentences)

Sentences of the *While* Language

We describe sentences using **context-free grammar (Backus-Naur form)**. Terminal symbols are tokens (words)

program ::= statmt*

statmt ::= println(stringConst , ident)

| ident = expr

optional part

| if (expr) statmt (else statmt)?

| while (expr) statmt

nesting of statements

| { statmt* }

expr ::= intLiteral | ident

nesting of expressions

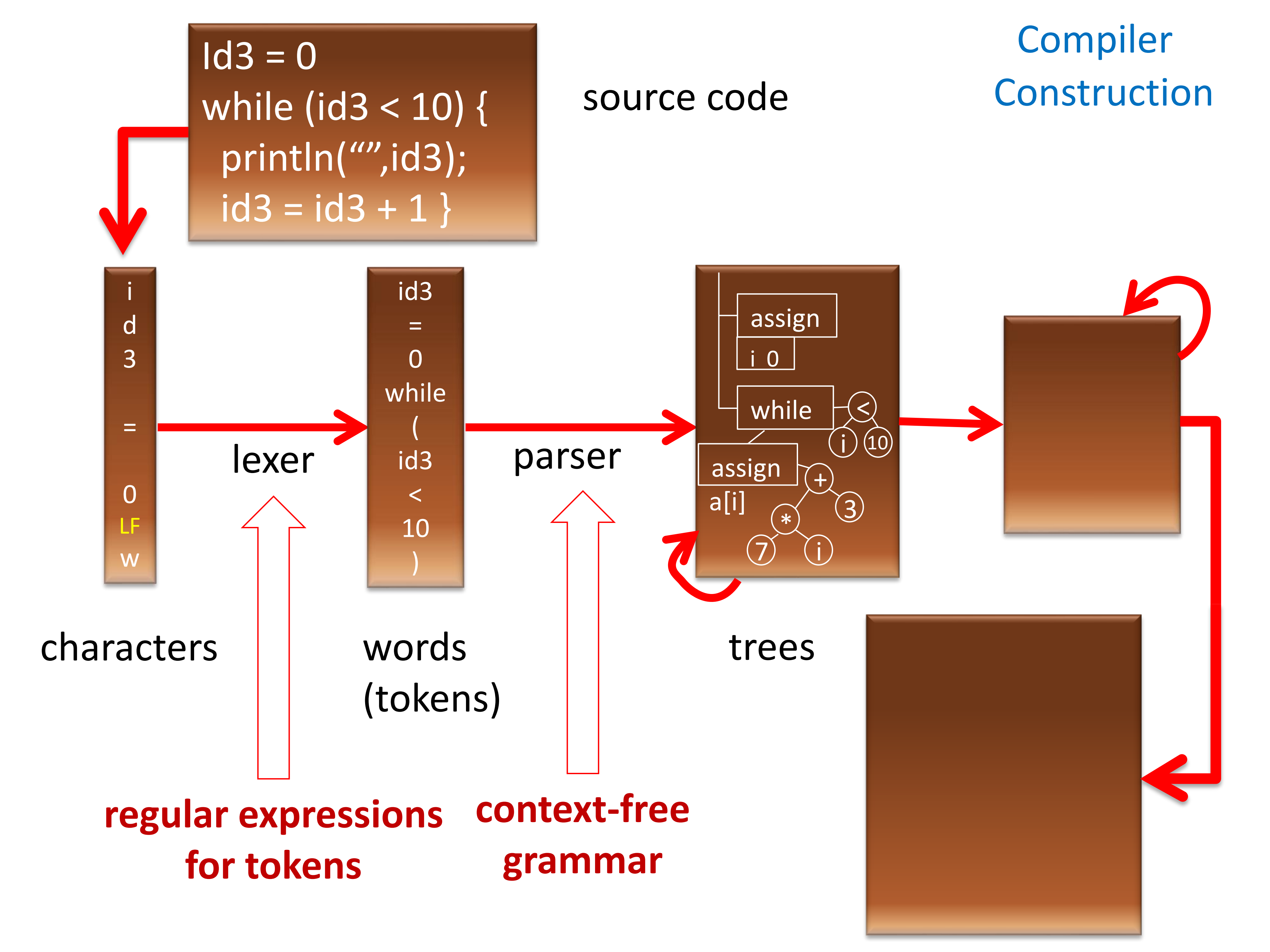
| expr (&& | < | == | + | - | * | / | %) expr

| ! expr | - expr

While Language without Nested Loops

```
statmt ::= println( stringConst , ident )
         | ident = expr
         | if ( expr ) statmt (else statmt)?
         | while ( expr ) statmtww
         | { statmt* }
statmtww ::= println( stringConst , ident )
         | ident = expr
         | if ( expr ) statmtww (else statmtww)?
         | { statmtww* }
```

statement without while


Compiler
Construction
Id3 = 0
while (id3 < 10) {
println(“”,id3);
id3 = id3 + 1 }
source code
i
d
3
=
0
LF
w
lexer
id3
=
0
while
(
id3
<
10
)
parser
assign
i 0
while
<
i
10
assign
a[i]
+
3
*
7
i
characters
words
(tokens)
trees
regular expressions
for tokens
context-free
grammar

Abstract Syntax - Trees

To get abstract syntax (trees, cases classes),
start from context-free grammar for tokens, then

- remove punctuation characters
- interpret rules as **tree descriptions**, not string descriptions

concrete syntax	abstract syntax
statmt ::= println(stringConst , ident)	PRINT(String,ident)
| ident := expr	ASSIGN(ident,expr)
| if (expr) statmt (else statmt)?	IF(expr,stmt,Option[statmt])
| while (expr) statmt	WHILE(expr,statmt)
| { statmt* }	BLOCK(List[statmt])

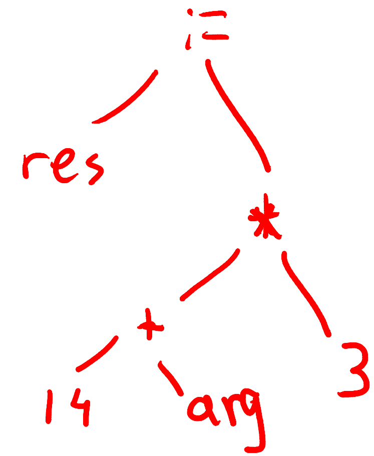
Scala trees for this abstract syntax

abstract class statmt
case class PRINT(id:ident) **extends** statmt
case class ASSIGN(id:ident, e:expr) **extends** statmt
case class IF(e:expr, s1:statmt, s2:statmt)
extends statmt ...

option[statmt]

Example of Parsing

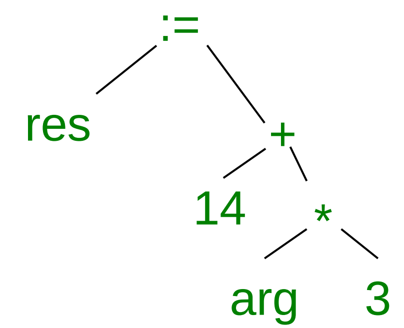
res = 14 + arg * 3



Lexer:

$\boxed{\text{res}}\ \boxed{=}\ \boxed{14}\ \boxed{+}\ \boxed{\text{arg}}\ \boxed{*}\ \boxed{3}$

Parser:



ASSIGN(res,
    PLUS(CONST(14),
        TIMES(VAR(arg),CONST(3))))

Code generator then “prints” this tree into instructions.

Reminder about Formal Languages

Languages Formally

$\emptyset \cdot L = \emptyset$ $\emptyset \cup L = L$

- A *word* is a finite, possibly empty, sequence of elements from some set Σ

  Σ – *alphabet,* Σ^* - set of all words over Σ

- For lexer: characters; for parser: token classes
- uv denotes concatenation of words u and v

  $u(vw)=(uv)w$

  $u \cdot \varepsilon = u = \varepsilon \cdot u$

- By a *language* we mean a subset of Σ^*
  - union, intersection, complement wrt. Σ^*

$L_1 \cdot L_2 = \{ u_1 u_2 \mid u_1 \text{ in } L_1, u_2 \text{ in } L_2 \}$

$L^0 = \{\varepsilon\}$ $\varepsilon = ""$ $L^1 = L \cdot L^0 = L \cdot \{\varepsilon\} = \{ u_1 \varepsilon \mid u_1 \in L \} = L$

$L^{k+1} = L\, L^k$ $L^* = \cup_k L^k$ (Kleene star)

Are there finitely many tokens?

- There are finitely many **token classes**
  - identifier
  - string
  - {
  - }
  - (

  ... (many, but finitely many)

There is unbounded number of *instances* of token classes identifier and string

When we discuss grammars, we work with token classes.

Examples of Languages

$\Sigma = \{a,b\}$

$\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, \ldots\}$

Examples of two languages, subsets of Σ^* :

$L_1 = \{a, bb, ab\}$ (finite language, three words)

$L_2 = \{ab, abab, ababab, \ldots\}$

$= \underbrace{\{ (ab)^n \mid n \geq 0 \}}_{\text{ad-hoc}}$ (infinite language)

Examples of Operations

$\Sigma = \{a,b\}$

L = {a,ab} ← L^*

L L = { aa, aab, aba, abab }

L* = { a, ab, aa, aab, aba, abab, aaa, … } = $(a|ab)^*$

(is bb inside L* ?)

$\subseteq$ $\supseteq$ = { w | immediately before each b there is a }

| ab | a | a | ab | ab | a | a |

$\underbrace{\boxed{ab}\,a\boxed{ab}\boxed{ab}a\boxed{ab}a}_{\cup}$ $\in \Sigma^*$