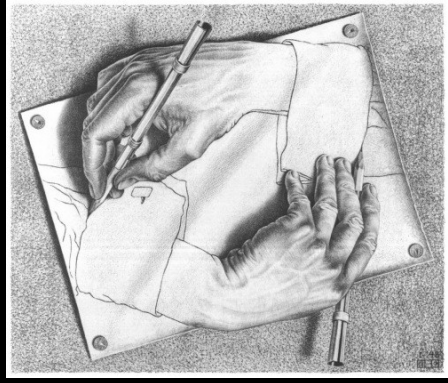
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE



Drawing Hands
M.C. Escher, 1948

http://lara.epfl.ch

Compiler Construction **2010** (6 credits)

Staff:

- Viktor Kuncak – Lectupes
- Hossein Hojjat – *Exercises*
- Philippe Suter – `{labs}`
- Étienne Kneuss, Ali Sinan Köksal – assistants
- Danielle Chamberlain – secretary

Today

- Compiler and its main phases
- Why we study compilers
- Course information

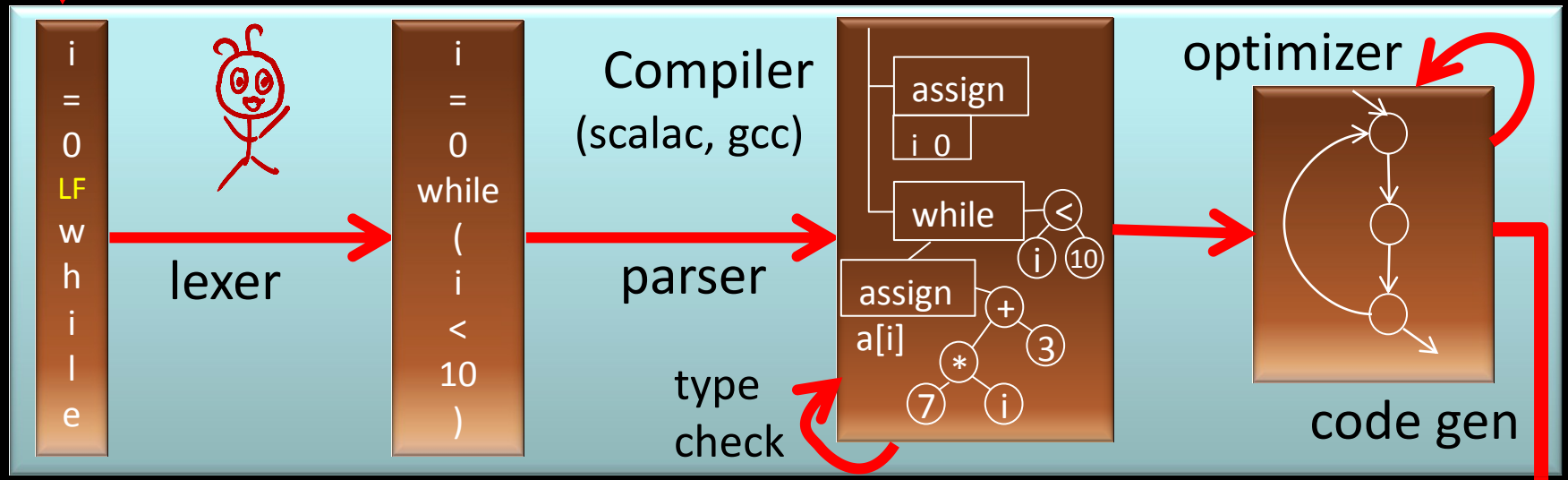
- Describing Syntax of Languages
- While language

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



characters words trees

machine code
(e.g. x86, arm, JVM)
efficient to execute

```
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
```

Example: javac

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

javac Test.java
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

```
#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

```
        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
```

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

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 code generation, garbage collection

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)

- Mondays 10:15-12:00 - Lectures in INM 202
  - Presentation of the material (ask questions!)
    Viktor Kuncak
- Wednesday 08:15-10:00 – Labs in INF3
  - Write code of your compiler, ask questions
    Philippe Suter, with help of Étienne Kneuss
- Wednesday 10:15-11:45 – Exercises in CO123
  - Do problems similar to homework and quizzes
    Hossein Hojjat , with help of Ali Sinan Köksal
- Home: homework, coding/debugging, review

For additional office hours, email us

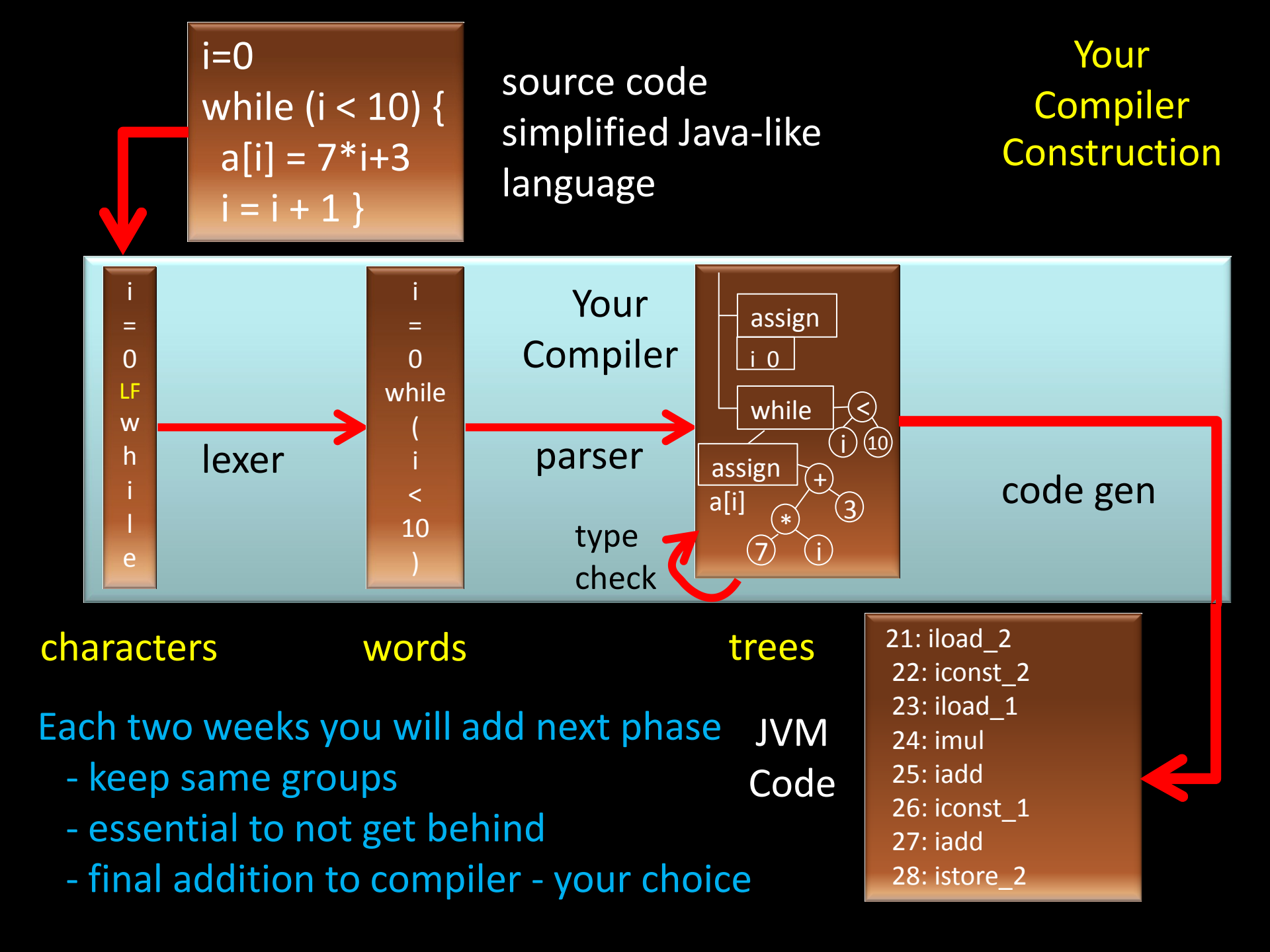
How We Compute Your Grade

- 55% : project (submit, explain if requested)
  - submit through Moodle
  - 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 moodle online forum, or ask us
- Encouraged: 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 plagarism detection tools
  - we will check if you fully understand your code
- Do the homework and quiz ***individually***
- You wouldn't steal a car. You wouldn't steal a compiler or a homework!



Each two weeks you will add next phase
- keep same groups
- essential to not get behind
- final addition to compiler - 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
  - you can learn it from www.scala-lang.org (if you need to learn it, start now)
- Helpful general background
  - Discrete structures (CS-150), Algorithms (CS-250)
- This course provides background for:
  - Advanced Compilers (Spring 2011)
  - Synthesis Analysis & Verification (Spring 2011)

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

More Course 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

- For the links to the books and more, see http://lara.epfl.ch (the Courses section)

Today

- Compiler and its main phases √
- Why we study compilers √
- Course information √

Questions so far?

- Describing Syntax of Languages
  - *While* language

Describing the Syntax of Languages

Describing Syntax: Why

- Goal: document precisely a superset of meaningful programs
  - 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 e.g. 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 0
  - no objects, pointers, arrays

While Language – Example Programs

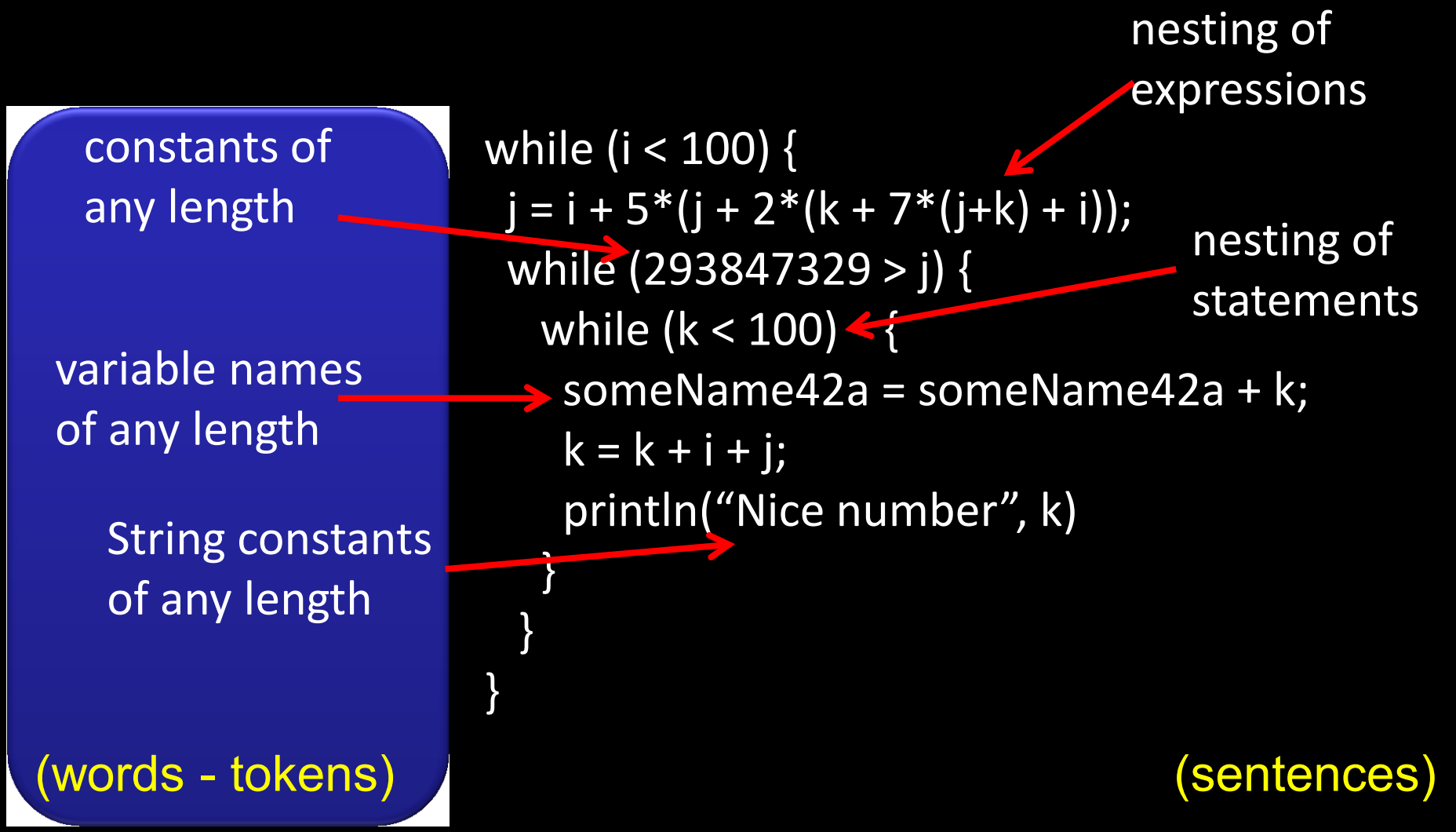
```
while (i < 100) {
  j = i + 1;
  while (j < 100) {
    println(“ “,i);
    println(“,”,j);
    j = j + 1;
  }
  i = i + 1;
}
```

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? (Collataz conjecture - open)

Reasons for Unbounded Program Length



Tokens (Words) of the *While* Language

Ident ::=
letter (letter | digit)*

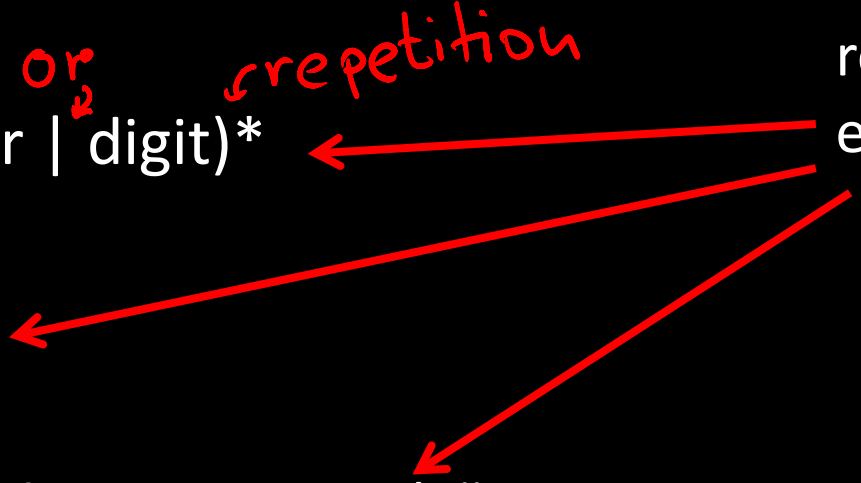
integerConst ::=
digit digit*

stringConst ::=
" AnySymbolExceptQuote* "

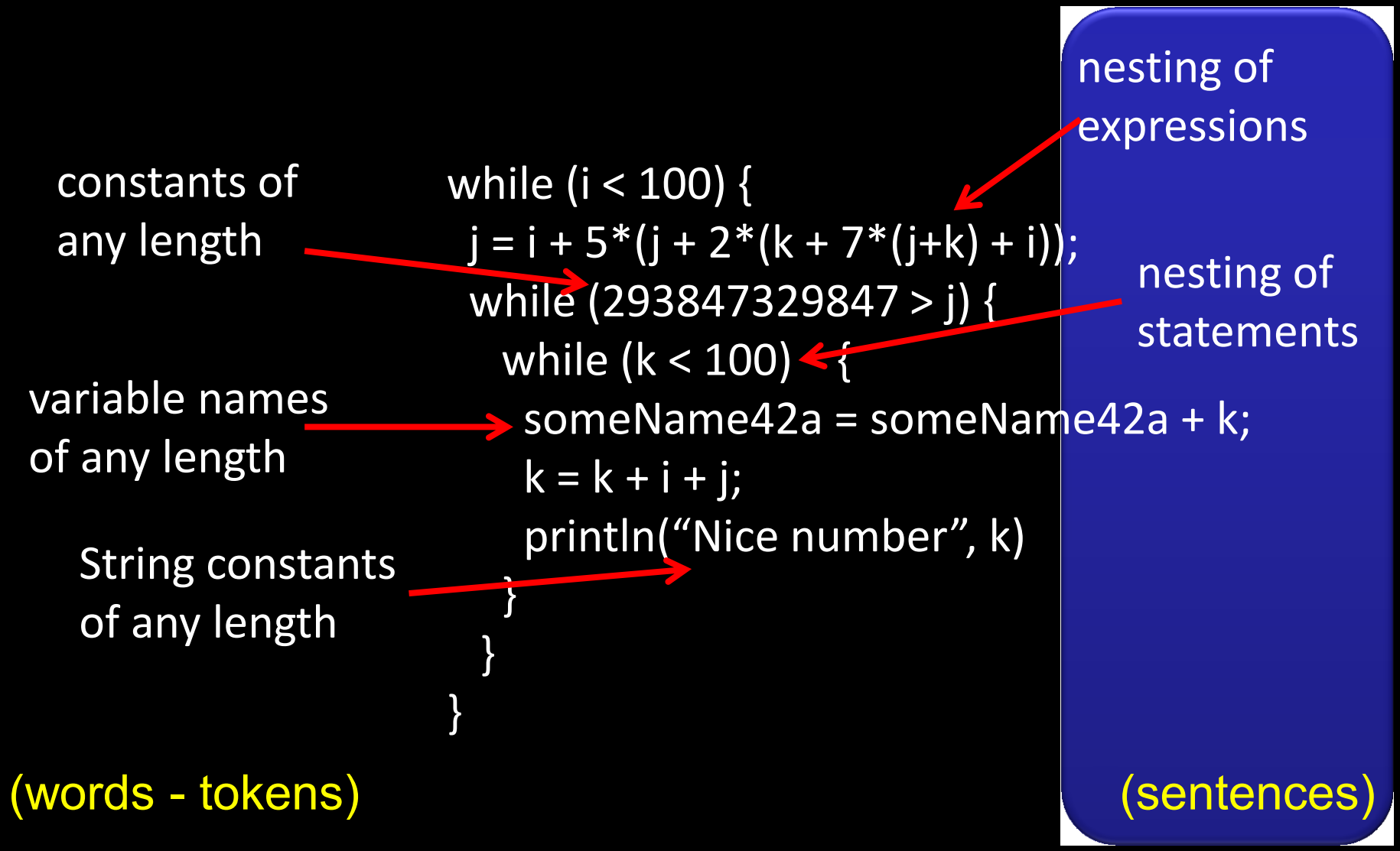
keywords
if else while println

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

letter ::= a | b | c | … | z | A | B | C | … | Z
digit ::= 0 | 1 | … | 8 | 9



Reasons for Unbounded Program Length



Sentences of the *While* Language

We give it as a context-free grammar where terminal symbols are tokens (words)

program ::= statmt*

statmt ::= println(stringConst , ident)

| ident = expr

| if (expr) statmt (else statmt)?

| while (expr) statmt

| { statmt* }

expr ::= intLiteral | ident

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

| ! expr | - expr

optional part

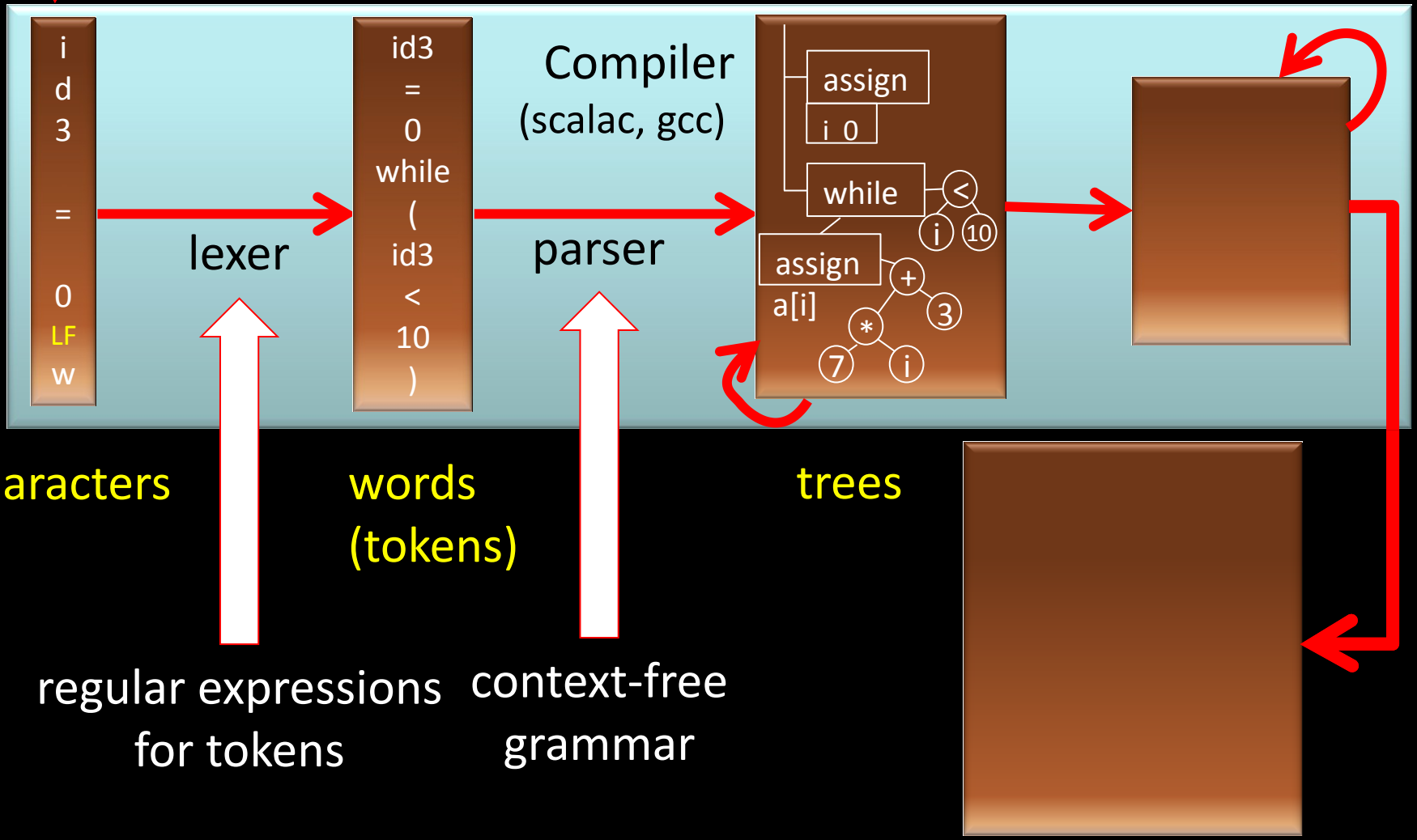
nesting of statements

nesting of expressions

Compiler Construction

```
Id3 = 0
while (id3 < 10) {
  println(“”,id3);
  id3 = id3 + 1 }
```

source code



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

program ::= statmt*

statmt ::= println(stringConst , ident) Print (String, Ident)

| ident = expr Assign (Ident, Expr)

| if (expr) statmt (else statmt)? If (Expr, Statmt, Option [Statmt])

| while (expr) statmt While (Expr, Statmt)

| { statmt* } List [Statmt]

Languages

- 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: chars for parser: tokens
- uv denotes concatenation of words u and v
- A set of words L subset is Σ^* is called language
  - union, intersection, complement wrt. Σ^*

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

$L^0 = \varepsilon$

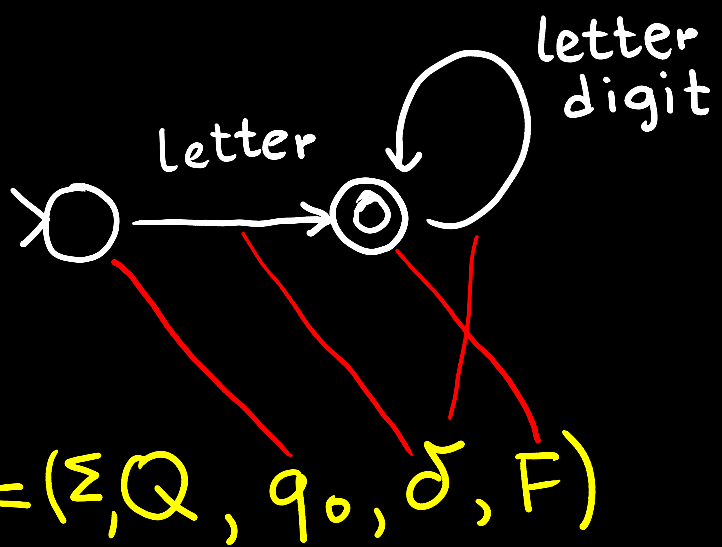
$L^{k+1} = L L^k$ $L^* = U_k L^k$ (Kleene star)

Regular Expressions

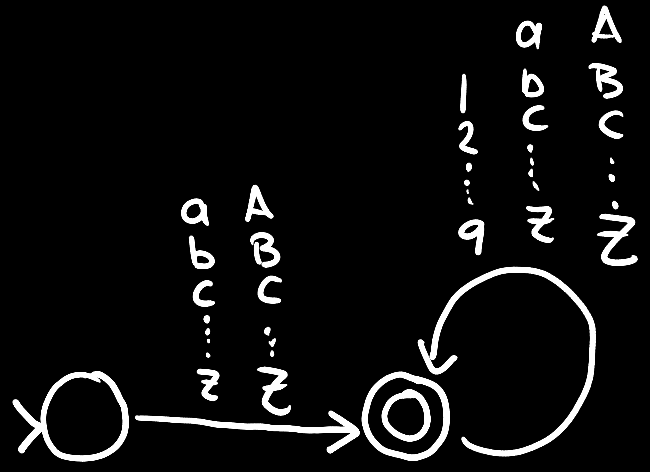
- One way to denote (often infinite) languages
- Any expression built from
  - empty language ∅
  - {ε} denoted just ε
  - {a} for a in Σ, denoted simply by a
  - union, denoted | or, sometimes +
  - concatenation, as multiplication or nothing
  - Kleene star *
- Identifiers: letter (letter | digit)*

  (letter,digit are shorthands from before)

Finite Automata



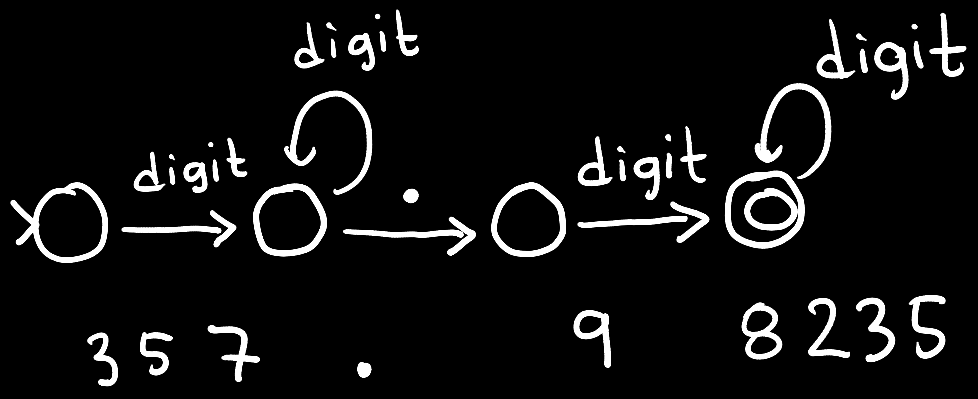
i.e.



$A = (\Sigma, Q, q_0, \delta, F)$ $\quad \delta \subseteq Q \times \Sigma \times Q$

- If L is a set of words, then it is a value of a regular expression if and only if it is the set accepted by some finite automaton
  - We say L is a regular language

Numbers with Decimal Point



digit digit* . digit digit*

What if the decimal part is optional?

Regular Expressions and Automata

- If L is a set of words, then it is a value of a regular expression if and only if it is the set accepted by some finite automaton

(review of construction)

More Examples

- Find automaton or regular expression for:
  - as many digits before as after decimal point?
  - Sequence of open and closed parantheses of even length?
  - Sequence of balanced parentheses
    ((()) ()) - balanced
    ( )) (( ) - not balanced
  - Comment as a sequence of space,LF,TAB, and comments from // until LF
  - Nested comments like /* … /* */ … */