

Compiler Construction

Staff:

Philipp Haller - Lectures

Mikael Blomstrand - Labs (TA)

Johan Bergdorf - Labs (TA)



About Myself

- 2006 Dipl.-Inform., Karlsruhe Institute of Technology (KIT), Germany
- 2010 Ph.D. Computer Science, Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland
- 2011–2012 Postdoc Stanford University, USA and EPFL, Switzerland
- 2012–2014 Consultant and software engineer, Typesafe Inc.
- Since Dec 2014 Assistant Professor at KTH

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

Example: javac

- from Java to JVM 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
```

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)

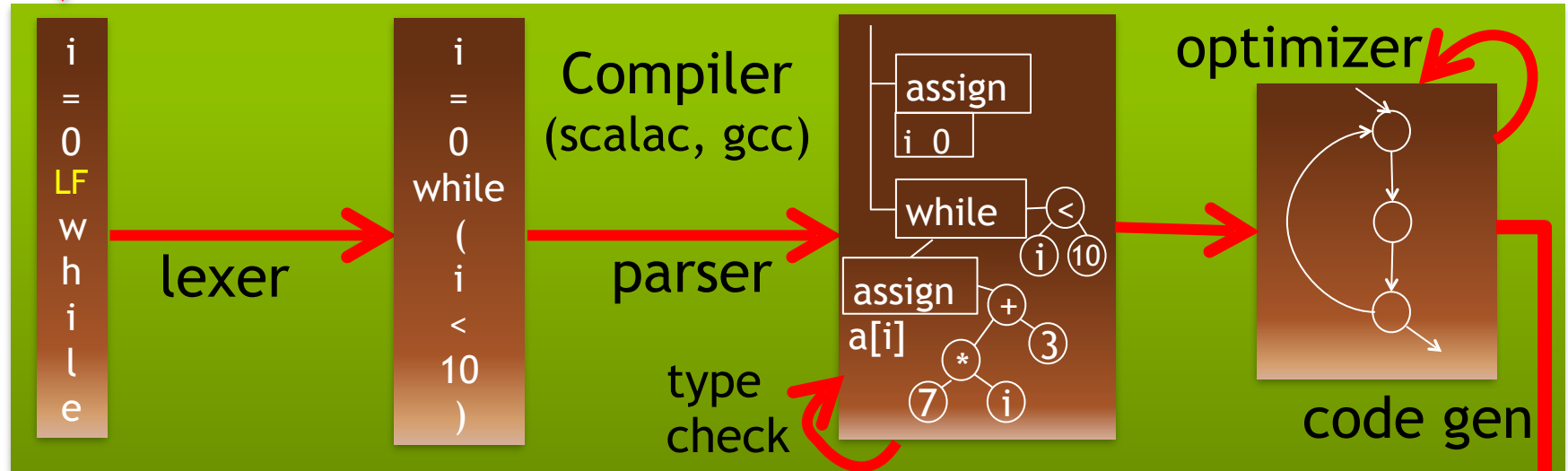
.L2:    cmpl $9, -12(%ebp)
        jle .L3
```

```
i=0
while (i < 10) {
  a[i] = 7*i+3
  i = i + 1 }
```

source code
(e.g. Scala, Java, C)
easy to write 😊

Compiler Construction

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
```



NATIONAL PHYSICAL LABORATORY

TEDDINGTON, MIDDLESEX, ENGLAND

PAPER 2-3

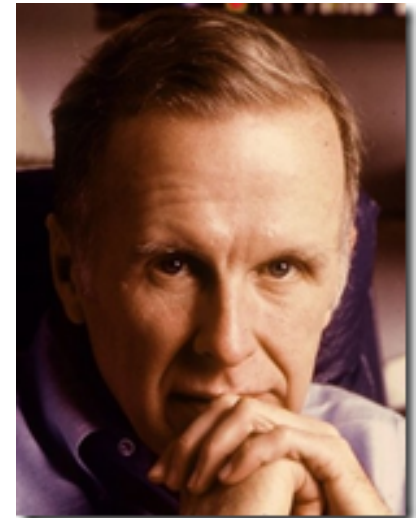
AUTOMATIC PROGRAMMING
PROPERTIES AND PERFORMANCE OF
FORTRAN SYSTEMS I AND II

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 repro-
duced without the permission of the author and of
the Secretary, National Physical Laboratory.

A pioneering
compiler:
FORTRAN
(FORmula
TRANslator)



Backus-Naur
Form - BNF

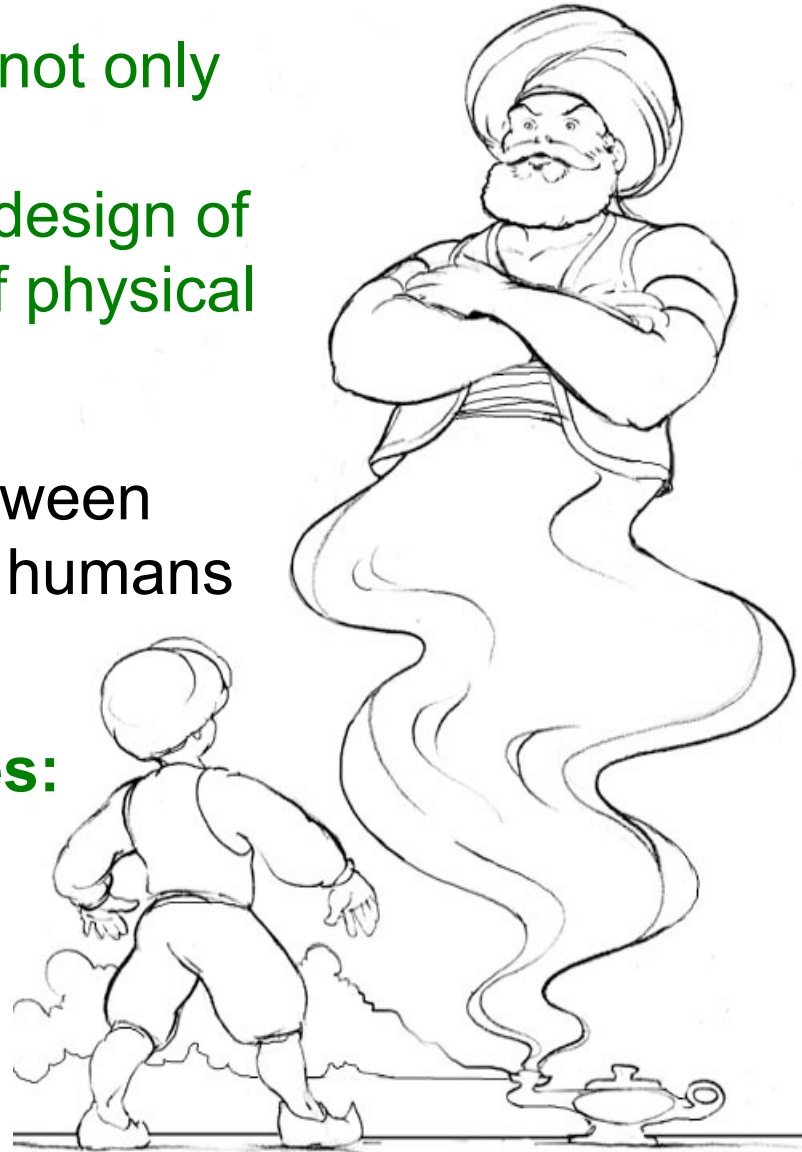
Turing Award
1977

Challenges for Future

Can target code **commands** include not only execution of commands on standard microprocessors, 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 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 a library for parsing JSON, XML, ...
- process complex input in an application (e.g. expression evaluator)
- parse simple natural language fragments

Schedule and Activities (9 credits)

- Time slots (typical, several exceptions)
 - Monday 13:00-15:00
 - Wednesday 10:00-12:00
 - Check schedule for each session, location changes often!
- Lectures, labs, exercises
 - Labs: best for questions, advice; continue working on code base outside lab sessions
- At home
 - Continue with programming the compiler
 - Practice solving problems to prepare for exam
- If you need more help, email us:
 - We will arrange additional meetings

How We Compute Your Grade

The grade is based on:

- Compiler project
 - Submit milestones which are tested and graded
 - Work in groups of 2, exceptionally 1 or 3
- Pen-and-paper exam
 - Duration: 4 hours

Computing the Course Grade

The course grade is determined using the following table:

Project →/ Exam ↓	A	B	C	D	E
A	A	B	B	C	D
B	A	B	C	C	D
C	A	B	C	D	D
D	B	B	C	D	E
E	B	C	C	D	E

You have to pass both the course project and the theory exam!

Learning Outcomes

learning outcome	E	D	C	B	A
Explain phases of a compilation pipeline	Show familiarity with the main terminology and concepts				
	assessed using theory exam				
Specify and construct automata-based lexical analyzers	Apply formal specifications to lexical analysis problems	Describe partial construction of lexical analyzers from formal specifications	Describe complete construction of lexical analyzers from formal specifications	Construct lexical analyzers from real-world formal specifications	
	assessed using labs (for level B) and theory exam (for level E-C)				
Explain and apply basic parsing theories	Explain and apply basic concepts of parsing theory	Reason about parsing theory using simple concepts	Construct parsing automata for simple grammars	Reason about parsing theory using advanced concepts	Construct parsing automata for complex grammars
	assessed using labs (for level A) and theory exam (for level E-B)				
Explain formalizations of simple type systems	Reason about typed programs using simple formal type rules	Correlate type rules with formal properties of type systems		Design type rules with given formal properties	
	assessed using theory exam				
Explain the main tasks of code generators	Explain terms and tasks of a code generator	Apply heuristics for the solution of important problems			
	assessed using theory exam				
Practically implement components of a simple compilation pipeline	Implement lexical analysis, parser, basic semantic analysis, basic type checker Implement advanced semantic analysis Implement advanced type checking, implement basic code generator Implement advanced code generator Implement compiler extension				
	assessed using labs				

Grading of the Exam

Grading of exam: to reach a given level, say A, all learning targets have to be reached at level A if possible, or at the highest level otherwise.

- The theory exam is a written exam
- You are allowed to use notes *that you have written yourself*; copies of any kind are not allowed
- You may not give copies of your notes to other students

Collaboration and Its Boundaries

- For clarification questions, discuss them on the course website, which we monitor
- Work in groups of 2 for project
 - everyone should know every part of the code
 - we may ask you to explain specific parts of the code
- Do not copy lab solutions from other groups!
 - code plagiarism is not allowed and we will try to detect plagiarism using tools
 - we will check if you fully understand your code

Course Materials

Main 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)

Lecture slides self-contained

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:
 - Aarne Ranta: Implementing Programming Languages
 - H.Seidl, R.Wilhelm, S.Hack: 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
- Simpler than the language for which you implement your compiler
- ‘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

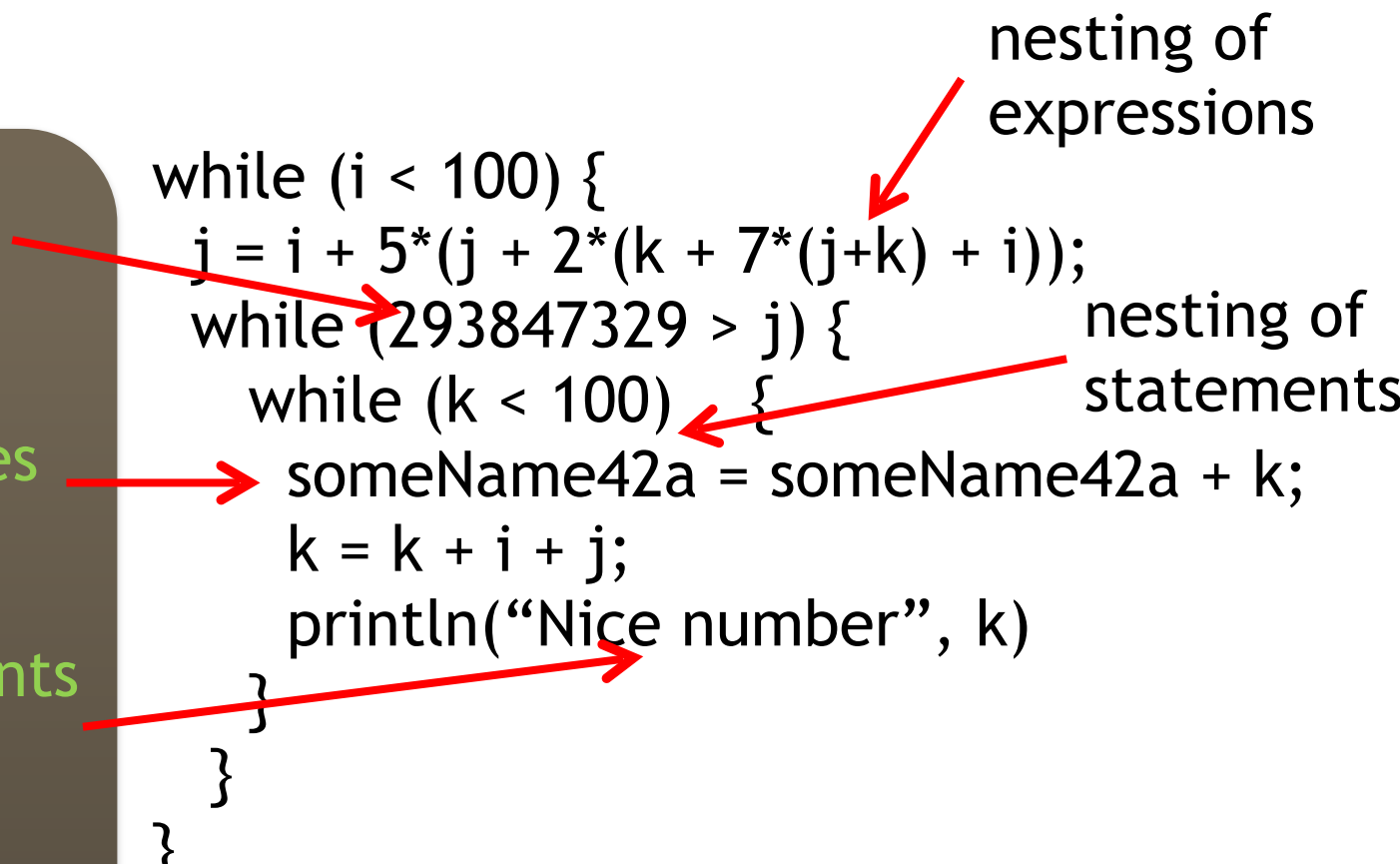
String constants
of any length

(words - tokens)

```
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)  
        }  
    }  
}
```

nesting of
expressions

nesting of
statements



The diagram illustrates the unbounded growth of program length through nesting. It features a code snippet with three nested loops. Red arrows point from descriptive text on the left to specific parts of the code: 'constants of any length' points to the number '293847329'; 'variable names of any length' points to 'someName42a'; 'String constants of any length' points to the closing brace of the innermost loop. On the right, 'nesting of expressions' points to the expression '7*(j+k) + i' inside the first loop, and 'nesting of statements' points to the opening brace of the third loop. The bottom text '(words - tokens)' indicates that each of these elements contributes to the total count of tokens in the program.

Tokens (Words) of the *While* Language

Ident ::=

letter (letter | digit)*

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) $\text{digit digit}^* [.] [\text{digit digit}^*]$

2) $\text{digit digit}^* [. \text{digit digit}^*]$

3) $\text{digit}^* . \text{digit digit}^*$

4) $\text{digit digit}^* . \text{digit digit}^*$

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)*

Compiler Construction

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

source code

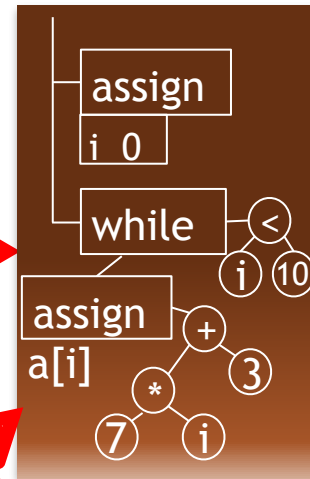


i
d3
=
0
LF
w

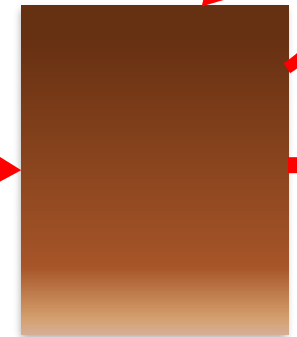
lexer

id3
=
0
while
(
id3
<
10
)

parser



trees



characters

words
(tokens)

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

More Reasons for Unbounded Length

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)
```

```
        }
```

```
    }
```

```
}
```

variable names
of any length

nesting of
expressions

nesting of
statements

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

| if (expr) statmt (else statmt)?

| while (expr) statmt

| { statmt* }

nesting of
statements



expr ::= intLiteral | ident

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

| ! expr | - expr

nesting of
expressions



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* }
```


Compiler Construction

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

source code

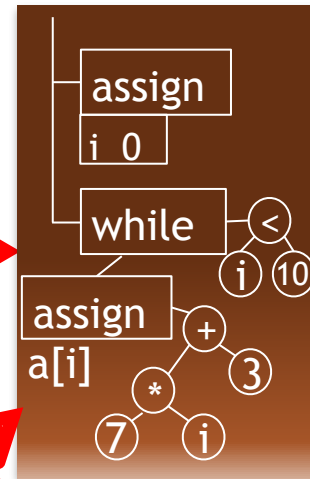


i
d3
=
0
LF
w

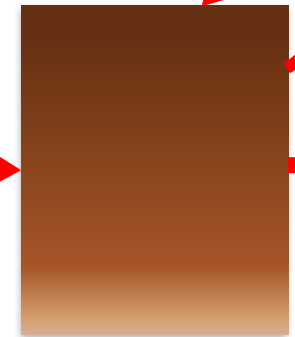
lexer

id3
=
0
while
(
id3
<
10
)

parser



trees



characters

words
(tokens)

regular expressions context-free
for tokens grammar

Abstract Syntax Trees

To get abstract syntax trees,
start from context-free grammar for tokens, then

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

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])

concrete syntax

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: Option[statmt])
 extends statmt ...

abstract syntax

ocaml vs Haskell vs Scala

ocaml:

```
type tree = Leaf | Node of tree * int * tree
```

Haskell:

```
data Tree = Leaf | Branch Tree Int Tree
```

Scala:

```
abstract class Tree  
case object Leaf extends Tree  
case class Node(left: Tree, x: Int, right: Tree)  
  extends Tree
```

Example of Parsing

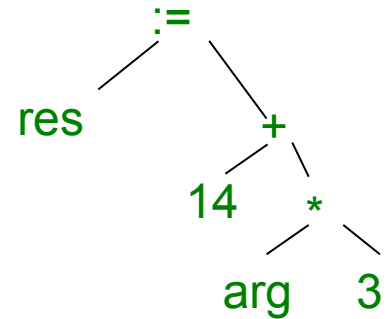
res = 14 + arg * 3

Lexer:

res = 14 + arg * 3

Parser:

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



Code generator then “prints” this tree into instructions.