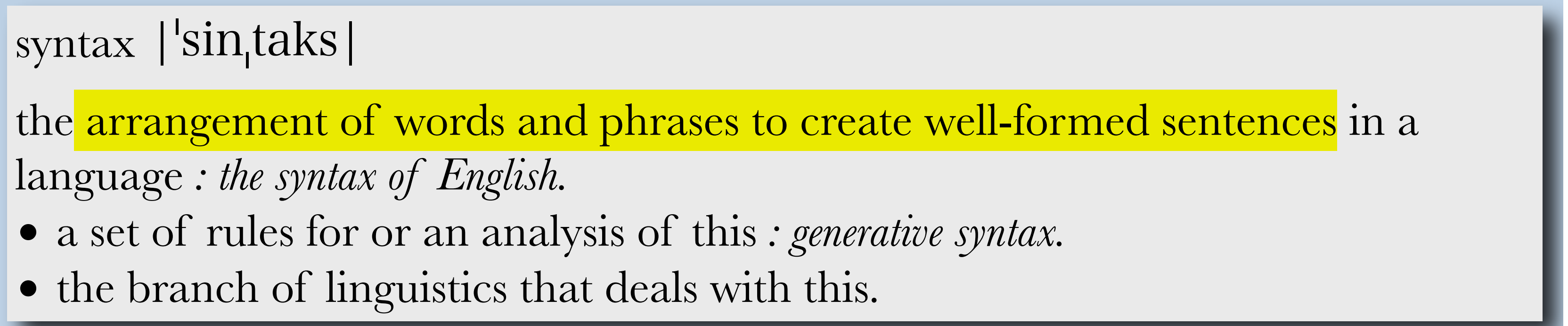
Interpretação e Compilação de Linguagens (de Programação)

21/22

Luís Caires (http://ctp.di.fct.unl.pt/~lcaires/)

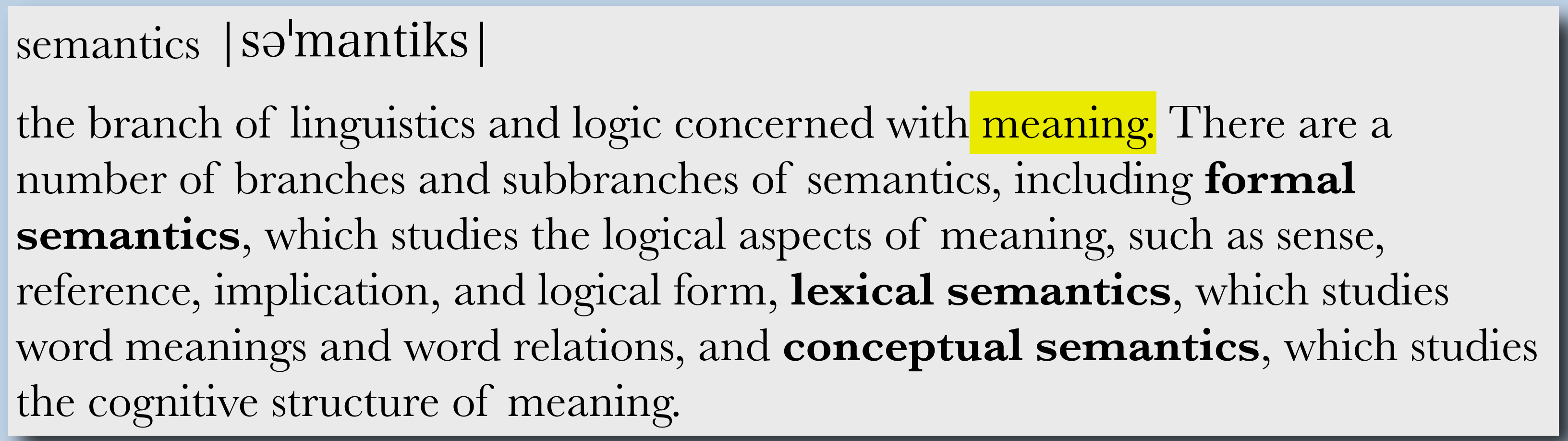
Programming Languages (PL)

- The purpose of a PL is to allow computational processes to be described specified by linguistic means.
- In early times, programs were mostly written in (physical) machine code.
- The definition of a PL involves to aspects, that must be characterised by a precise, **non-ambiguous way**:
  - Syntax
  - Semantics



syntax |ˈsinˌtaks|

the arrangement of words and phrases to create well-formed sentences in a language : *the syntax of English.*
- a set of rules for or an analysis of this : *generative syntax.*
- the branch of linguistics that deals with this.



semantics |səˈmantiks|

the branch of linguistics and logic concerned with meaning. There are a number of branches and subbranches of semantics, including **formal semantics**, which studies the logical aspects of meaning, such as sense, reference, implication, and logical form, **lexical semantics**, which studies word meanings and word relations, and **conceptual semantics**, which studies the cognitive structure of meaning.

Programming Languages

Example of syntactic ambiguity:

What is the value of `f(10)` ?

```
int f(int x) {
  if ( x > 0 )
    if ( x < 10 ) return x;
  else return 10;
  return 0;
}
```

```
f(10) = ??
```

Programming Languages

Example of semantic ambiguity:
What is the value of `f(2)+g(3)` ?

```
f(2) + g(3) = ??
```

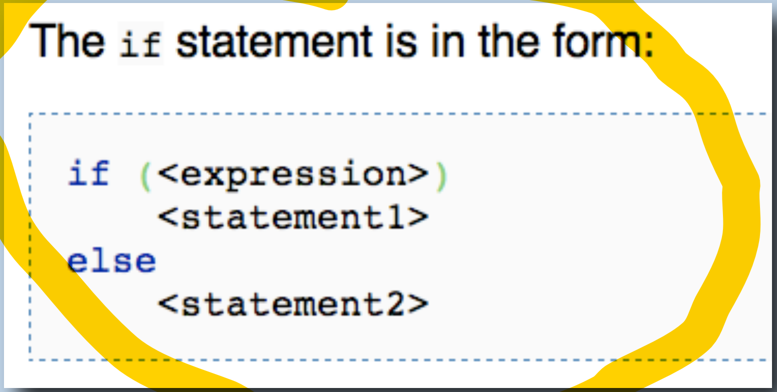
```
public class A {
   static int a = 0;

   static int f(int x) {
      a = a + 1;
      return x;
   }
   static int g(int y) { return y + a; }

   static int sum(int x, int y) { return x + y; }

   public static void main(String[] args) {
      System.out.println(sum(f(2), g(3)));

   }
}
```

Programming Languages (Syntax)

Syntax specifies the **form** of programs in the language, how they should be written, without attending to their meaning. Syntax is just about the structure of program phrases.

A PL syntax is conveniently defined by a **lexicon** (set of words or tokens) and a **grammar** (set of formation rules).

```
integer literal:  (“0” | [“1”-”9”][“0”-”9”]*)
real literal : ([“0”-”9”]) “.” ([“0”-”9”])* (“E” ...)?
identifier: [a-z,A-Z,_][a-z,A-Z,_,0-9]*
reserved symbol: int, float, void, while, class, ...
```



The `if` statement is in the form:

```
if (<expression>)
    <statement1>
else
    <statement2>
```

Programming Languages (Syntax)

The concrete syntax of a PL may be formally specified using regular languages for tokens and context free grammars for program phrases (see course “Theory of Computation”).

Given a description of tokens (eg. using regular expressions) and given a description of a grammar (eg., using grammar rules) one may construct lexical analysers and parsers, which are programs that check the syntax programs for syntactical correctness and construct **abstract syntax trees (AST).**

ASTs are data structures that represent syntactically correct programs in a structured form (not just as text - sequence of characters, in the source files).

Good news: there are **tools** that will do that for you automatically.

You will still need to know how to specify regular languages and (non-ambiguous) context free grammars.

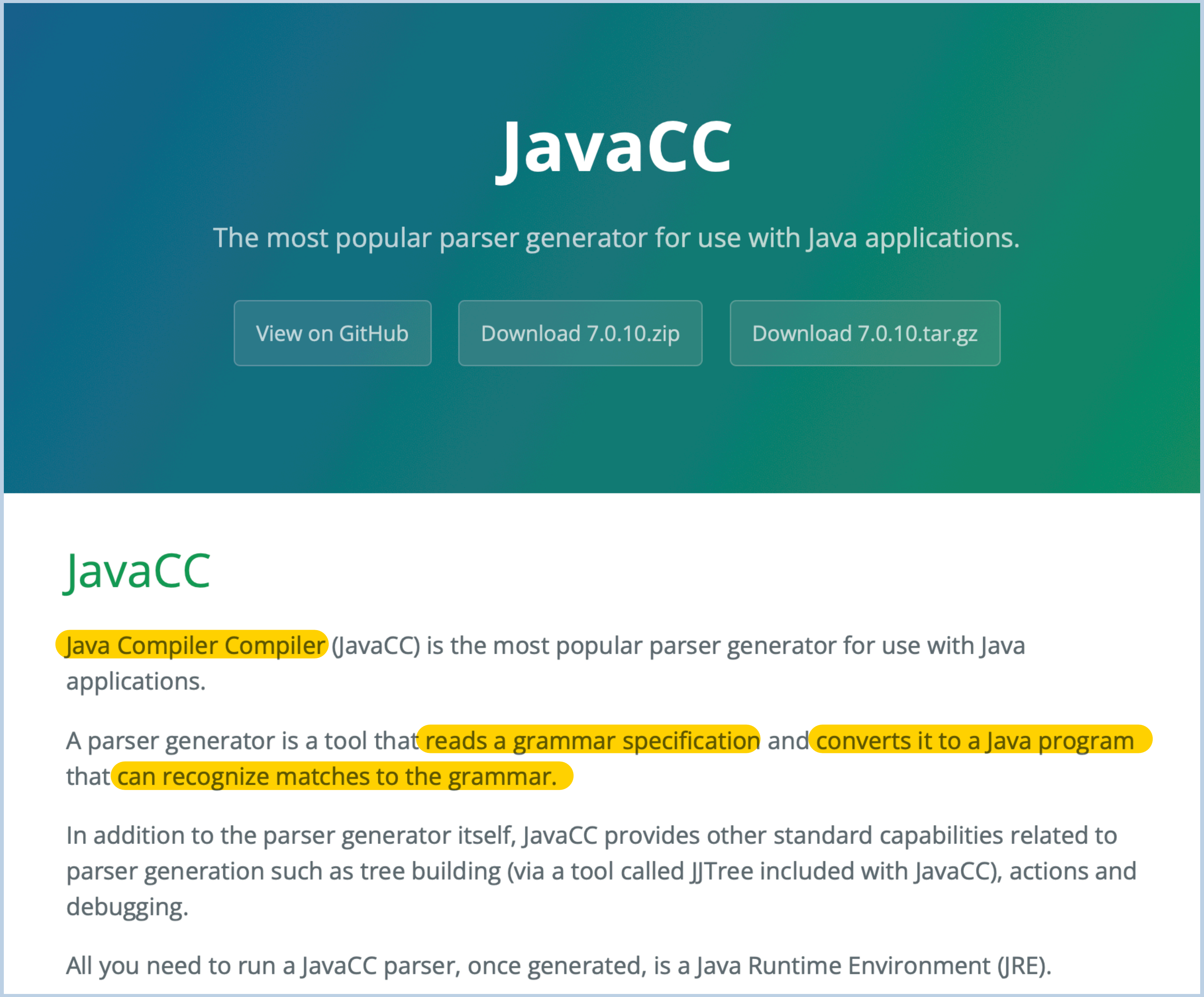
Grammar for arithmetic expressions (yacc)

```
%token  NAME
%token  NUMBER
%token  EQ
%token PLUS MINUS TIMES DIV
%left MINUS PLUS
%left TIMES DIV
%nonassoc UMINUS

%%

statement_list
        : statement
        | statement statement_list

statement
        : NAME EQ expression ';' {vbltable[$1] = $3; }

expression
        : expression PLUS expression {$$ = $1 + $3;}
        | expression MINUS expression {$$ = $1 - $3;}
        | expression TIMES expression {$$ = $1 * $3;}
        | expression DIV   expression {$$ = $1 / $3;}
        | MINUS expression %prec UMINUS {$$ = - $2;}
        | '(' expression ')' { $$ = $2; }
        | NUMBER
        | NAME   { $$ = vbltable[$1]; }
```

Grammar for arithmetic expressions (javacc)

```
void Start() :
{ }
{
      exp() <EOL>
}

void exp() :
{ }
{
      term() [ <PLUS> exp() ]
}

void term() :
{ }
{
      factor() [ <MULTIPLY> term() ]
}

void factor() :
{  }
{
      <CONSTANT>
|     <LPAR> exp() <RPAR>
}
```

Grammar for arithmetic expressions (javacc)


JavaCC
The most popular parser generator for use with Java applications.
View on GitHub
Download 7.0.10.zip
Download 7.0.10.tar.gz

JavaCC

Java Compiler Compiler (JavaCC) is the most popular parser generator for use with Java applications.

A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar.

In addition to the parser generator itself, JavaCC provides other standard capabilities related to parser generation such as tree building (via a tool called JJTree included with JavaCC), actions and debugging.

All you need to run a JavaCC parser, once generated, is a Java Runtime Environment (JRE).

Programming Languages (Semantics)

A PL **semantics** describes in a **precise way** the **meaning** of program **elements and** phrases. Typically you find verbose descriptions in reference manuals. There is no place for ambiguity or arbitrariness in meaning!

10.4 Array Access

A component of an array is accessed by an array access expression (§15.13) that consists of an expression whose value is an array reference followed by an indexing expression enclosed by `[` and `]`, as in `A[i]`. All arrays are `0`-origin. An array with length *n* can be indexed by the integers `0` to *n*-1.

Arrays must be indexed by `int` values; `short`, `byte`, or `char` values may also be used as index values because they are subjected to unary numeric promotion (§) and become `int` values. An attempt to access an array component with a `long` index value results in a compile-time error.

All array accesses are checked at run time; an attempt to use an index that is less than zero or greater than or equal to the length of the array causes an `ArrayIndexOutOfBoundsException` to be thrown.

in *The Java Language Specification*

Programming Languages (Semantics)

The semantics of a PL may be defined by giving a computable function I which assigns a definite meaning to each program (fragment)

$$I : \text{PROG} \rightarrow \text{DENOT}$$

PROG = set of all programs (as syntactical structures)

DENOT = set of all meanings (denotations)

The semantic mapping I may be mathematically defined, typically using set theoretic constructions.

This is referred to as an **denotational semantics**.

More pragmatically the semantic mapping may also be given by an interpreter algorithm.

Such **interpreter algorithm** takes as input any program (as a syntactical data structure / AST) and yields its **value** and/or **effect**.

This is referred to as an **operational semantics** (approach in this course).

Code as Data

An interpreter for a language L is a program I that accepts as input the representation of a program in the source language (as an data structure) and realizes its execution according to the semantics of L.

The interpreter may be written in some language X, in general the language that is accepts is a different one. Eg., we may write a Python interpreter in C.

This process is called "bootstrapping" and is used in general to define interpreters for arbitrary languages. Languages that allows one to write interpreter for any language are called Turing complete, and this is the main content of the so-called Turing universality.

An expressive programming language should allow one to write an interpreter / compiler for itself (f"ull bootstrapping").

Code as Data

A compiler for a language L is a program I that accepts as input the representation of a program in the source language (as an data structure) and outputs an equivalent program (with the same meaning) in another language.

A compiler is a **translator**, unlike an interpreter it does not execute the source program but translates the source program in a program in a simpler language, or in a language for which an interpreter or compiler already exists, or even a physical (an hardware processor) or an abstract machine (e.g., JVM) able to execute the target code directly.

The translation process must preserve the meaning of programs, that is, the code generated by the compiler (target code) must have the same meaning as the source code.

Goals of the Course: Knowledge

- What techniques are used in the design and implementation of programming languages ?
- What are the building blocks of a programming language ?
- How to describe, analize and justify the features and characteristics of a programming language using the concepts studied?
- How to design interpreters and compilers ?
- How can we define and predict the effect of programming constructs in a precise manner ?
- How can we express and ensure properties of programming languages such as error absence and type safety ?
- How can we define and express validation algorithms for programming languages, that will work for any program written on it ?
- How do modern runtime support environments for programming languages (JVM, .NET, LLVM) work ?

Goals of the Course: Hands-on

- How do we construct a syntactic analyser (parser) ?
- What tools are there to assist on that ?
- How do we represent programs as data ?
- How do we specify the semantics of a programming language ?
- How to do it for abstract high level concepts such as higher order functions, objects, classes, etc ?
- How do we use it to implement an interpreter or compiler ?
- How does a compiler generate code for a real or abstract machine?
- How do we define and implement basic static analysis algorithms (e.g., type-checking) ?

“Road Map”

We will progress in a “onion” layer incremental way, covering key construct present in all programming languages.

The course tightly couples principles design and implementation, in the hands-on part you will develop an interpreter and compiler for a realistic programming language.

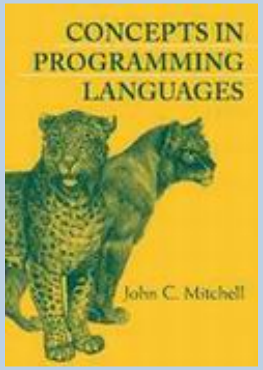
This is an “integrative” course, you combine lots of stuff: AED, TC, AC, SE,

- Values, Basic Operations and Expressions
- Naming and Binding
- State (mutable memory)
- Functional abstraction
- Types and Type Systems
- Data Abstraction
- Objects, Classes and Modules
- We will see how the various primitives may be interpreted and compiled to a target machine (JVM / LLVM)

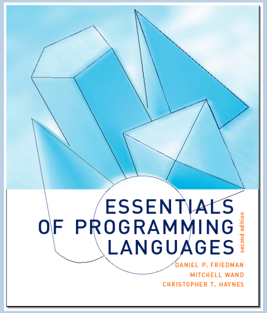
"If you don't understand interpreters, you can still write programs; you can even be a competent programmer. But you can't be a master."

(Hal Abelson, Essentials of Programming Languages de Friedman et al.)

Bibliography

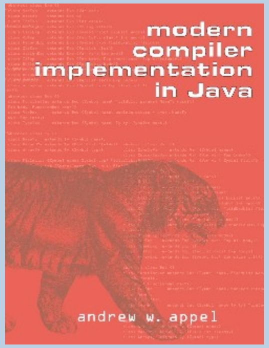


“*Concepts in Programming Languages*”,
John C. Mitchell,
Cambridge University Press.
ISBN 0 521 78098 5



“Essentials of Programming Languages”,
Daniel Friedman, Mitchell Wand, Christopher Haynes,
MIT Press.

“The Study of Programming Languages”,
Ryan Stansifer,
Prentice Hall International Edition.



“Modern Compiler Implementation in Java”
Andrew W. Appel
Cambridge University Press

Course Grading

Continuous Evaluation:

Midterm test (8)

Final test (8)

Handout (2 phases) 4

Must be realised in groups of 2

Exam (for those failing in the CE)

Admittance to exam requires submission of the handout