Compiler I

(dt. Übersetzer I)

Prof. Dr. Uwe Kastens

Winter 2001/2002

Objectives

The participants are taught to

- understand **fundamental techniques** of language implementation,
- use **generating tools and standard solutions**,
- understand compiler construction as a systematic combination of **algorithms, theories** and **software engineering** methods for the solution of a **precisely specified task**,
- apply compiler techniques for languages **other than programming languages**.

Forms of teaching:

Lectures

Tutorials

Exercises

Homeworks

Running project

Lectures in English

Some agreements about giving lectures in English:

- I’ll speak English unless someone asks me to explain something in German.
- Stop me or slow me down whenever you get lost.
- I don‘t speak as well as a native speaker; but I’ll do my best ...
- You may ask questions and give answers in English or in German.
- I’ll prepare the slides in English. A German version is available.
- You‘ll have to learn to speak about the material in at least one of the two languages.
- You may vote which language to be used in the tutorials.
- You may chose German or English for the oral exam.

Syllabus

Week	Chapter	Topic
1	**Introduction**	Compiler tasks
2		Compiler structure
3	**Lexical analysis**	Scanning, token representation
4	**Syntactic analysis**	Recursive decent parsing
5		LR Parsing
6		Parser generators
7		Grammar design
8	**Semantic analysis**	Attribute grammars
9		Attribute grammar specifications
10		Name analysis
11		Type analysis
12	**Transformation**	Intermediate language, target trees
13		Target texts
14	**Synthesis**	Overview
15	**Summary**	

Prerequisites

from Lecture	Topic	here needed for
Foundations of Programming Languages:		
	4 levels of language properties	Compiler tasks, compiler structure
	Context-free grammars	Syntactic analysis
	Scope rules	Name analysis
	Data types	Type analysis
	Lifetime, runtime stack	Storage model, code generation
Modeling:		
	Finite automata	Lexical analysis
	Context-free grammars	Syntactic analysis

References

Material for this course **Compiler I**: http://www.uni-paderborn.de/cs/ag-kastens/compi
in German **Übersetzer I** (1999/2000): http://www.uni-paderborn.de/cs/ag-kastens/uebi
in English **Compiler II**: http://www.uni-paderborn.de/cs/ag-kastens/uebii

Modellierung: http://www.uni-paderborn.de/cs/ag-kastens/model
Grundlagen der Programmiersprachen: http://www.uni-paderborn.de/cs/ag-kastens/gdp

U. Kastens: **Übersetzerbau**, Handbuch der Informatik 3.3, Oldenbourg, 1990
(not available on the market anymore, available in the library of the University)

W. M. Waite, L. R. Carter: **An Introduction to Compiler Construction,**
Harper Collins, New York, 1993

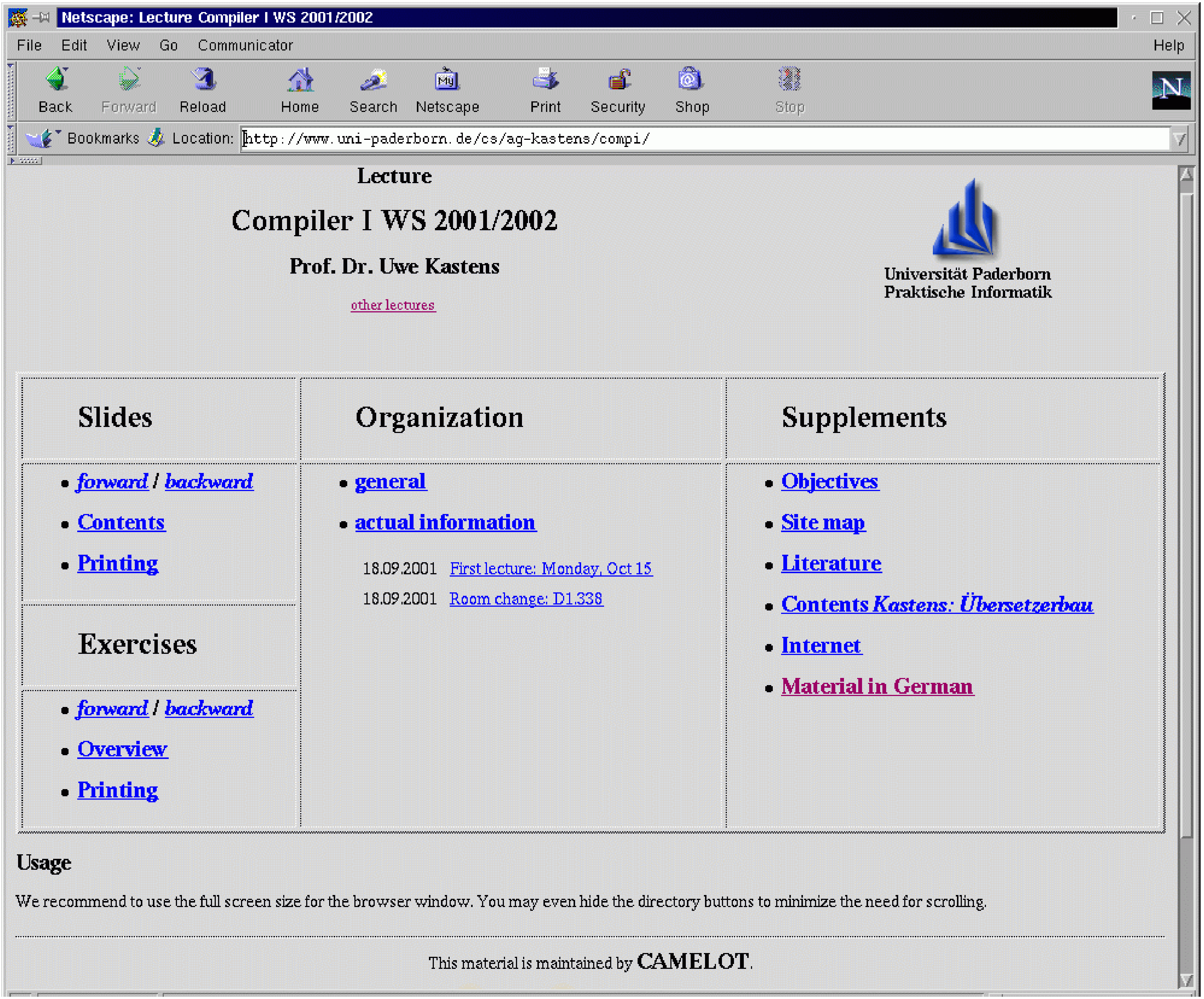
W. M. Waite, G. Goos: **Compiler Construction**, Springer-Verlag, 1983

R. Wilhelm, D. Maurer: **Übersetzerbau - Theorie, Konstruktion, Generierung**,
Springer-Verlag, 1992

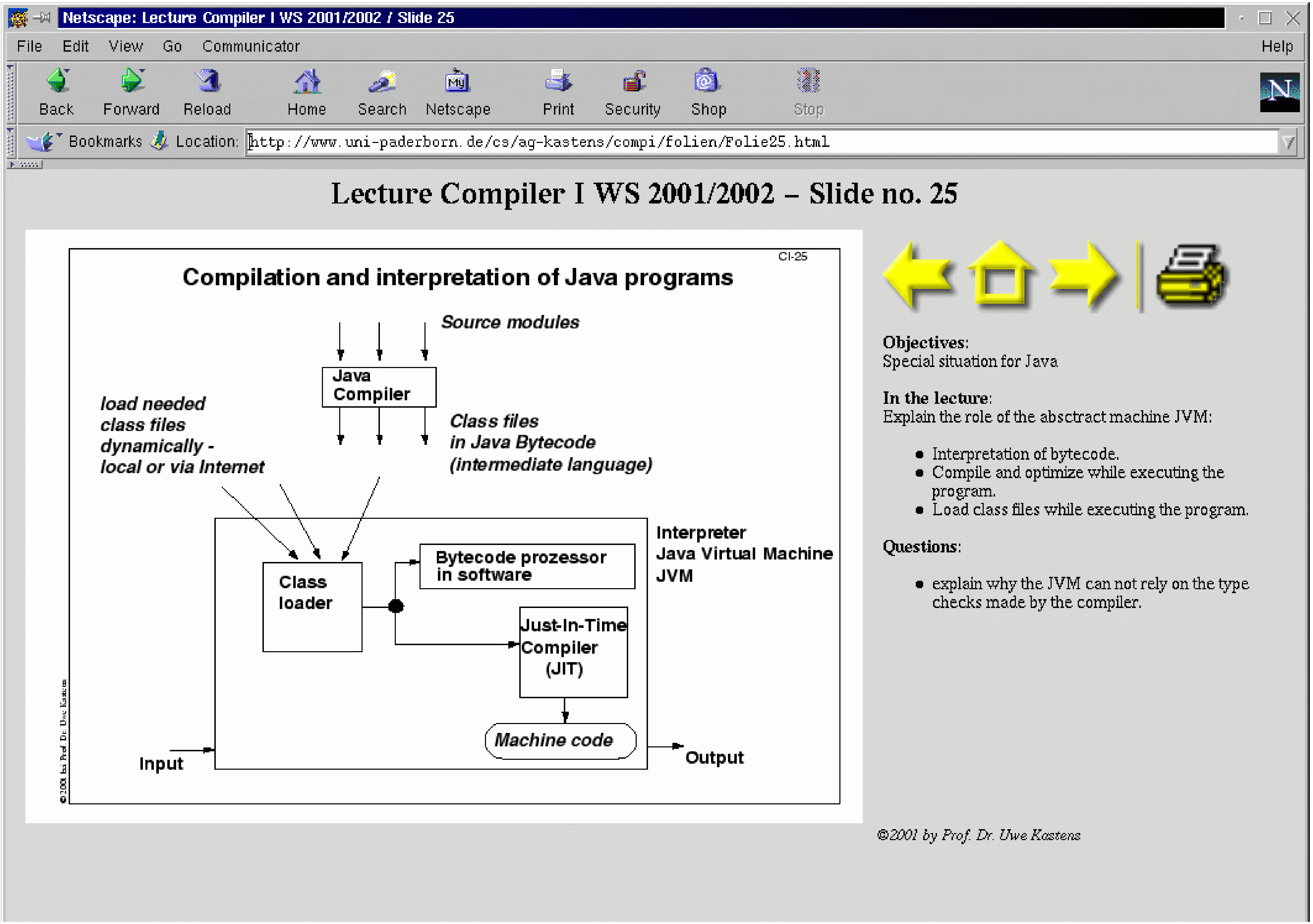
A. Aho, R. Sethi, J. D. Ullman: **Compilers - Principles, Techniques and Tools**,
Addison-Wesley, 1986

A. W. Appel: **Modern Compiler Implementation in C**, Cambridge University Press, 1997
(available for Java and for ML, too)

Course material in the Web



Commented slide in the course material



What does a compiler compile?

A **compiler** transforms correct sentences of its **source language** into sentences of its **target language** such that their **meaning is unchanged.**

Examples:

Source language:	**Target language:**
Programming language C++	**Machine language** Sparc code
Programming language Java	**Abstract machine** Java Bytecode
Programming language C++	**Programming language (source-to-source)** C
Application language LaTeX Data base language (SQL)	**Application language** HTML Data base system calls

What is compiled here?

```
class Average
    { private:
        int sum, count;
      public:
        Average (void)
          { sum = 0; count = 0; }
        void Enter (int val)
         { sum = sum + val; count++; }
        float GetAverage (void)
          { return sum / count; }
    };
--------------
_Enter__7Averagei:
            pushl %ebp
            movl %esp,%ebp
            movl 8(%ebp),%edx
            movl 12(%ebp),%eax
            addl %eax,(%edx)
            incl 4(%edx)
    L6:
            movl %ebp,%esp
            popl %ebp
            ret
```

```
class Average
{ private
    int sum, count;
  public
    Average ()
      { sum = 0; count = 0; }
    void Enter (int val)
      { sum = sum + val; count++; }
    float GetAverage ()
      { return sum / count; }
};
---------------
1: Enter: (int) --> void
   Access: []
   Attribute ‚Code‘ (Length 49)
      Code: 21 Bytes Stackdepth: 3 Locals: 2
      0:    aload_0
      1:    aload_0
      2:    getfield cp4
      5:    iload_1
      6:    iadd
      7:    putfield cp4
      10:   aload_0
      11:   dup
      12:   getfield cp3
      15:   iconst_1
      16:   iadd
```

What is compiled here?

```
program Average;
      var sum, count: integer;
          aver: integer;
      procedure Enter (val: integer);
          begin sum := sum + val;
                count := count + 1;
          end;
    begin
      sum := 0; count := 0;
      Enter (5); Enter (7);
      aver := sum div count;
    end.
-----------
void ENTER_5 (char *slnk , int VAL_4)
     {
     {/* data definitions: */
        /* executable code: */
        {
           SUM_1 = (SUM_1)+(VAL_4);
           COUNT_2 = (COUNT_2)+(1);
           ;
        }
     }}/* ENTER_5 */
```

```
\documentstyle[12pt]{article}
\begin{document}
\section{Introduction}
This is a very short document.
It just shows
\begin{itemize}
\item an item, and
\item another item.
\end{itemize}
\end{document}

-------------

%%Page: 1 1
1 0 bop 164 315 a Fc(1)81
b(In)n(tro)r(duction)
164 425 y Fb(This)16
b(is)g(a)h(v)o(ery)e(short)
i(do)q(cumen)o(t.)j(It)c(just)g
(sho)o(ws)237 527 y Fa(\017)24 b
Fb(an)17 b(item,)
c(and)237 628 y Fa(\017)24 b
Fb(another)17 b(item.)
961 2607 y(1)p
eop
```

Languages for specification and modeling

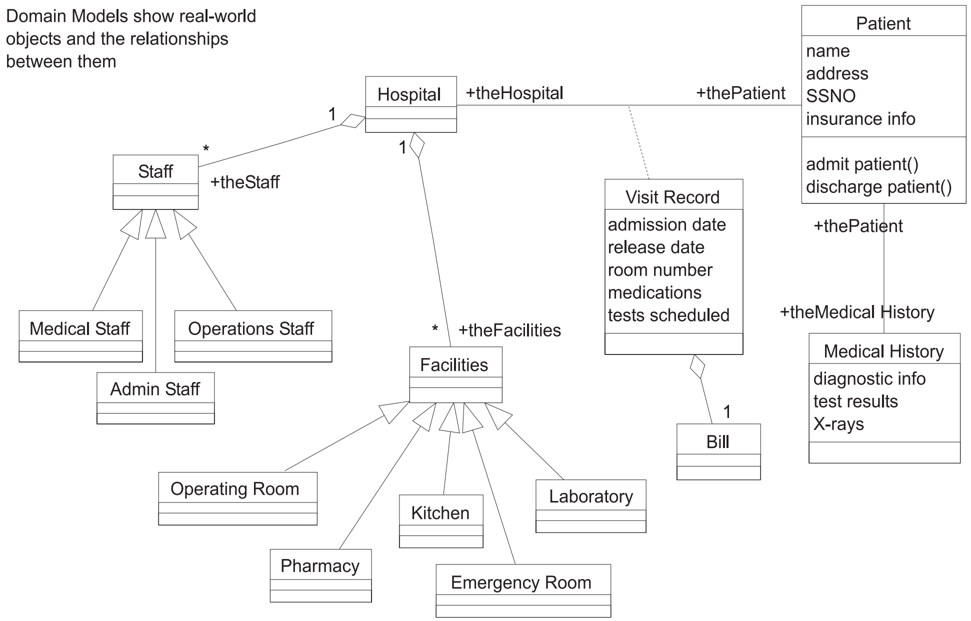
SDL (CCITT)
Specification and Description Language:

```
block Dialogue;
  signal
    Money, Release, Change, Accept, Avail, Unavail, Price,
    Showtxt, Choice, Done, Flushed, Close, Filled;
  process Coins referenced;
  process Control referenced;
  process Viewpoint referenced;
  signalroute Plop
    from env to Coins
      with Coin_10, Coin_50, Coin_100, Coin_x;
  signalroute Pong
    from Coins to env
      with Coin_10, Coin_50, Coin_100, Coin_x;
  signalroute Cash
    from Coins to Control
      with Money, Avail, Unavail, Flushed, Filled;
    from Control to Coins
      with Accept, Release, Change, Close;

  ...

  connect Pay and Plop;
  connect Flush and Pong;
endblock Dialogue;
```

UML
Unified Modeling Language:



Domain Specific Languages (DSL)

A language designed for a **specific application domain.**
Application Generator: Implementation of a DSL by a **program generator**

Examples:

- **Simulation of mechatronic feedback systems**
- **Robot control**
- **Collecting data from instruments**
- **Testing car instruments**
- **Report generator for bibliographies:**

```
string name =   InString "Which author?";
int since =     InInt "Since which year?";
int cnt = 0;

"\nPapers of ", name, " since ", since, ":\n";
[ SELECT name <= Author && since <= Year;
  cnt = cnt + 1;
  Year, "\t", Title, "\n";
]
"\n", name, " published ", cnt, "papers.\n";
```

U. Kastens: Construction of Application Generators Using Eli, Workshop on Compiler Techniques for Application Domain Languages ..., Linköping, April 1996

Programming languages as source or target languages

Programming languages as source languages:

- **Program analysis**
  call graphs, control-flow graph, data dependencies, e. g. for the year 2000 problem
- **Recognition of structures and patterns**
  e. g. for Reengineering

Program languages as target languages:

- **Specifications (SDL, OMT, UML)**
- **graphic modeling of structures**
- **DSL, Application generator**

=> Compiler task: Source-to-source compilation

Semester project as running example

A Structure Generator

We are going to develop a tool that implements **record structures**. In particular, the structure generator takes a set of **record descriptions**. Each specifies a **set of named and typed fields**. For each record a **Java class** declaration is to be generated. It contains a constructor method and access methods for the specified record fields.

The tool will be used in an environment where field description are created by other tools, which for example analyze texts for the occurrence of certain phrases. Hence, the descriptions of fields may occur in arbitrary order, and the same field may be described more than once. The structure generator **accumulates the field descriptions** such that for each record a single class declaration is generated which has all fields of that record.

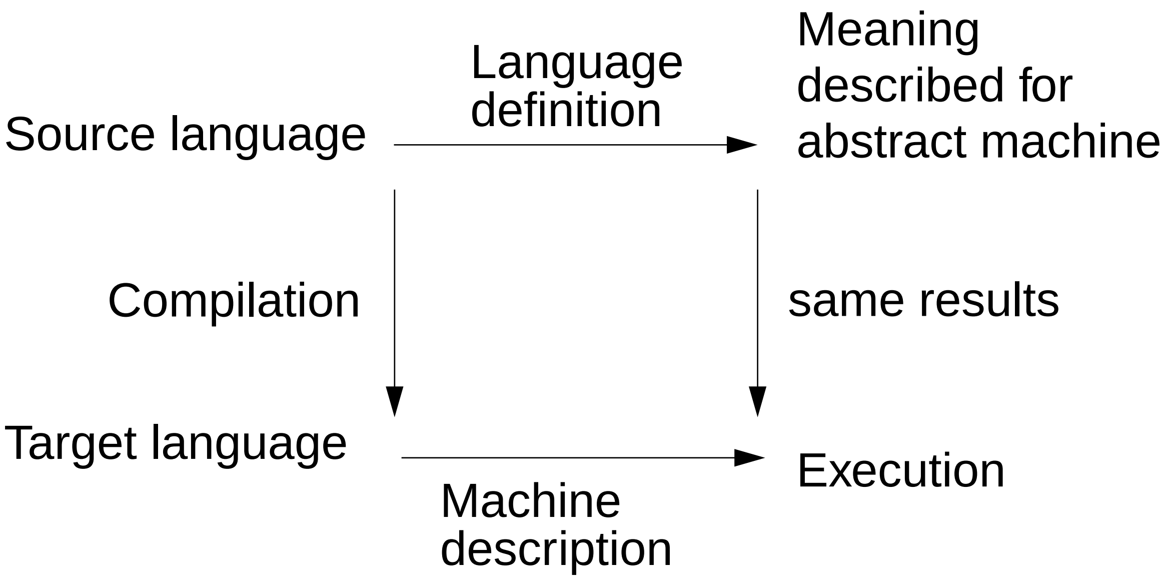
Design a **domain specific language**.

Implement an **application generator** for it.

Apply all **techniques of the course** that are useful for the task.

Meaning preserving transformation

A **compiler** transforms correct sentences of its **source language** into sentences of its **target language** such that their **meaning is unchanged**.



A **meaning** is defined only for **correct** programs. Compiler task: Error handling

The compiler analyses **static** properties of the program at **compile time,**
e. g. definitions of Variables, types of expressions. Decides: Is the program **compilable?**

Dynamic properties of the program are checked at **runtime,**
e. g. indexing of arrays. Decides: Is the program **executable?**

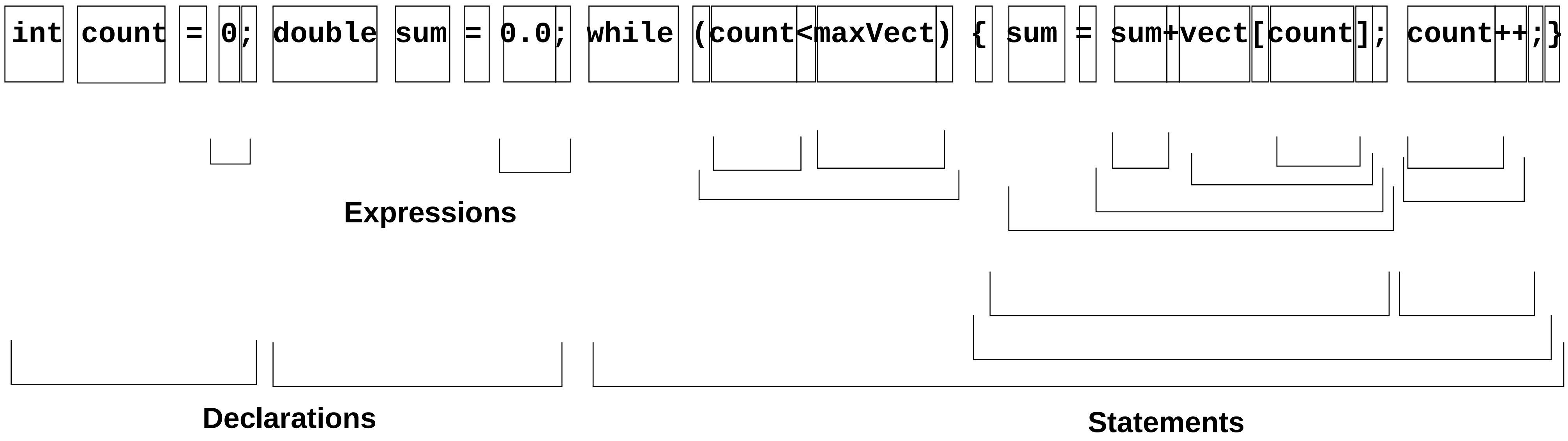
But in Java: Compilation of bytecode at runtime, just in time compilation (JIT)

Example: Tokens and structure

Character sequence

```
int count = 0; double sum = 0.0; while (count<maxVect) { sum = sum+vect[count]; count++;}
```

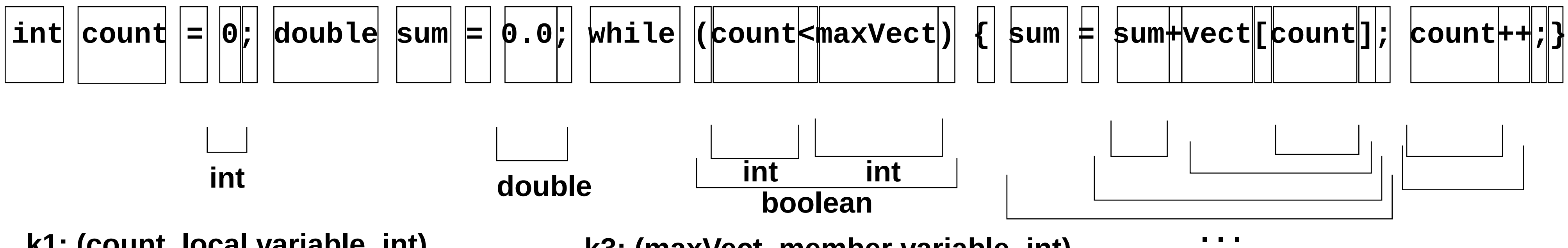
Tokens



Structure

Example: Names, types, generated code

Tokens



k1: (count, local variable, int)
k2: (sum, local variable, double)

k3: (maxVect, member variable, int)
k4: (vect, member variable, double array)

Names and types

generated Bytecode

```
0 iconst_0
1 istore_1
2 dconst_0
3 dstore_2
4 goto 19
7 dload_2
8 getstatic #5 <vect[]>
11 iload_1
12 faload
13 f2d
14 dadd
15 dstore_2
16 iinc 1 1
19 iload_1
20 getstatic #4 <maxVect>
23 if_icmplt 7
```

Language definition - Compiler task

- **Notation of tokens** — **lexical analysis**
  keywords, identifiers, literals
  formal definition: regular expressions

- **Syntactic structure** — **syntactic analysis**
  formal definition: context-free grammar

- **Static semantics** — **semantic analysis, transformation**
  binding names to program objects, typing rules
  usually defined by informal texts

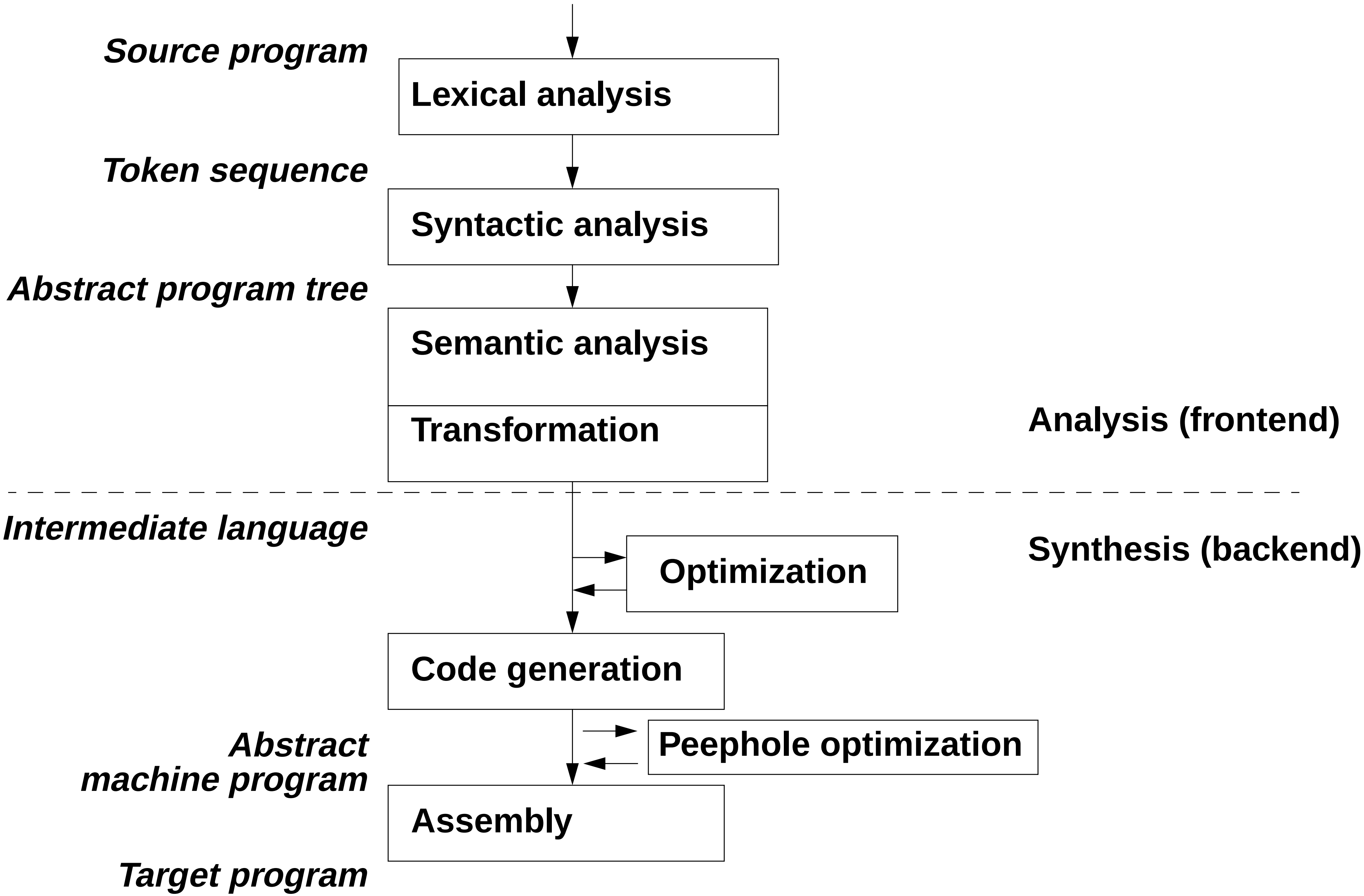
- **Dynamic semantics** — **transformation, code generation**
  semantics, effect of the execution of constructs
  usually defined by informal texts
  in terms of an abstract machine

- **Definition of the target language (machine)** — **transformation, code generation assembly**

Compiler tasks

Structuring	Lexical analysis	Scanning Conversion
	Syntactic analysis	Parsing Tree construction
Translation	Semantic analysis	Name analysis Type analysis
	Transformation	Data mapping Action mapping
Encoding	Code generation	Execution-order Register allocation Instruction selection
	Assembly	Instruction encoding Internal Addressing External Addressing

Compiler structure and interfaces



Software qualities of the compiler

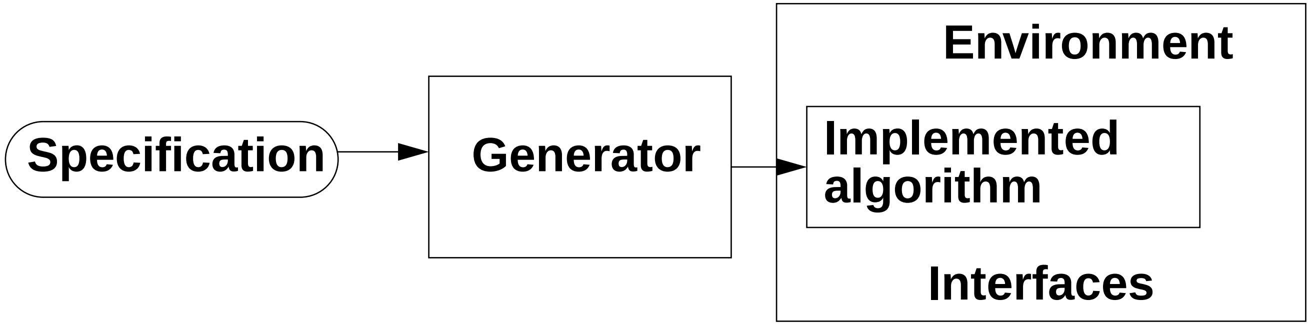
- **Correctness** Translate correct programs correctly.
  Reject wrong programs and give error messages
- **Efficiency** Storage and time used by the compiler
- **Code efficiency** Storage and time used by the generated code
  Compiler task: Optimization
- **User support** Compiler task: Error handling
  (recognition, message, recovery)
- **Robustness** Give a reasonable reaction on every input

Strategies for compiler construction

- Obey exactly to the language definition
- Use generating tools
- Use standard components
- Apply standard methods
- Validate the compiler against a test suite
- Verify components of the compiler

Generators

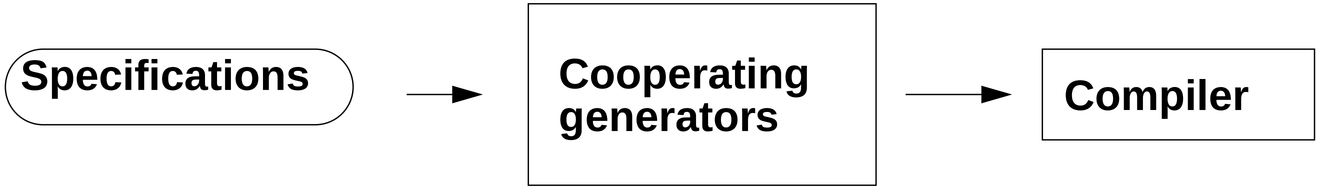
Pattern:



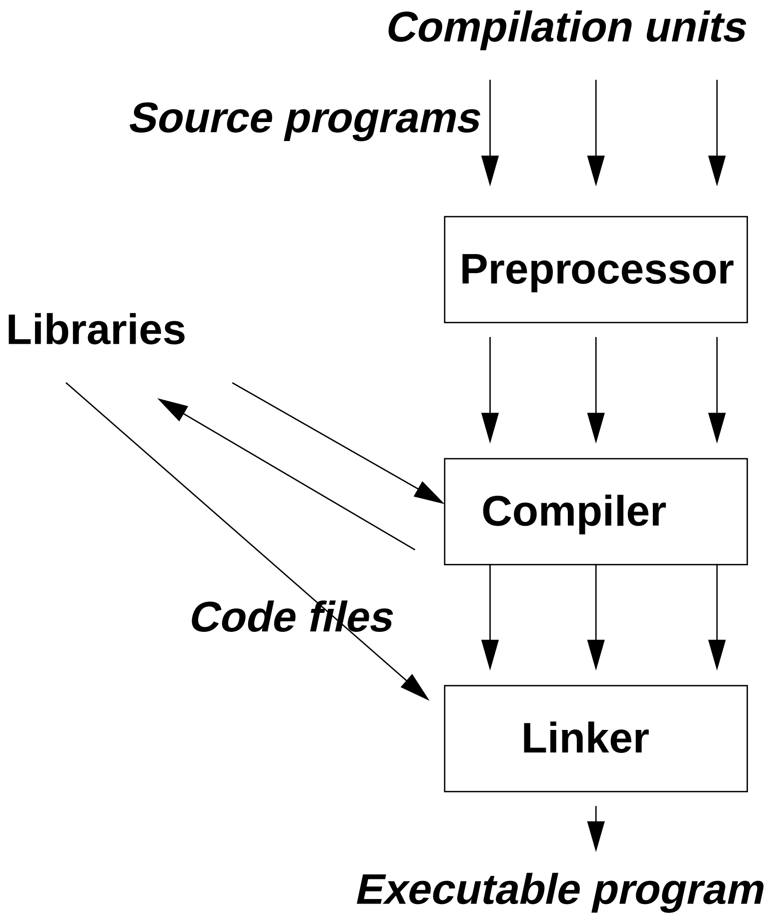
Typical compiler tasks solved by generators:

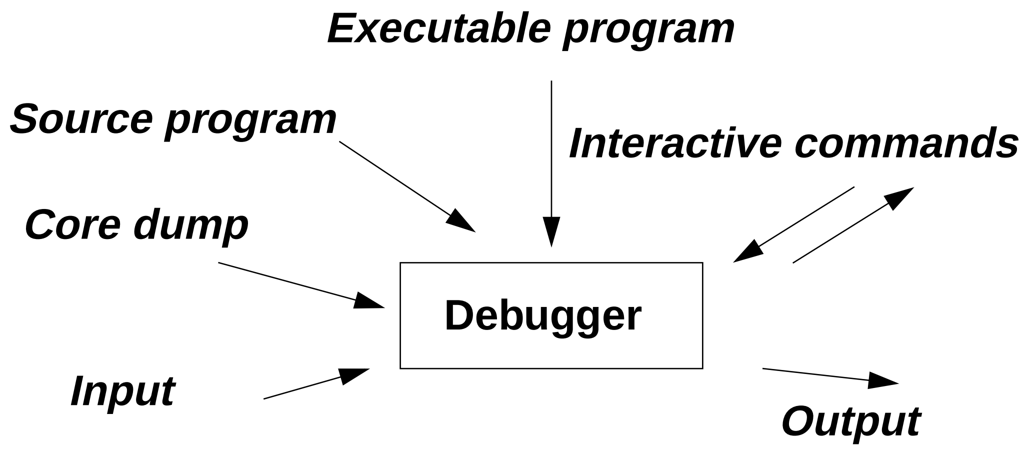
Regular expressions	**Scanner generator**	Finite automaton
Context-free grammar	**Parser generator**	Stack automaton
Attribute grammar	**Attribute evaluator generator**	Tree walking algorithm
Code patterns	**Code selection generator**	Pattern matching

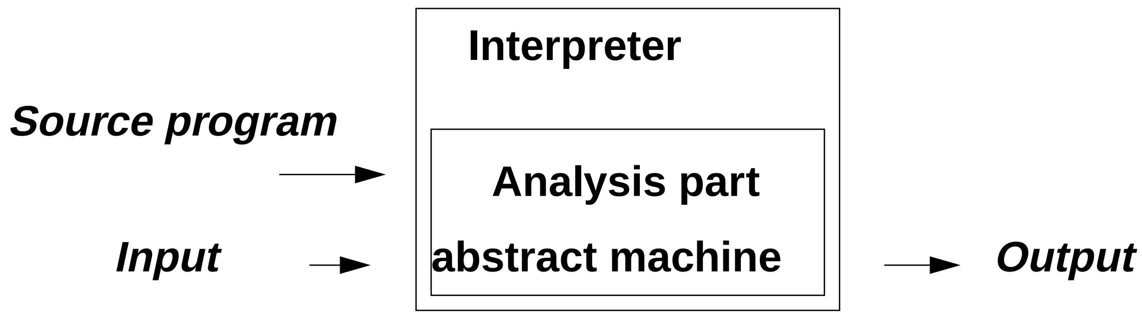
integrated system Eli:



Environment of compilers







Compilation and interpretation of Java programs