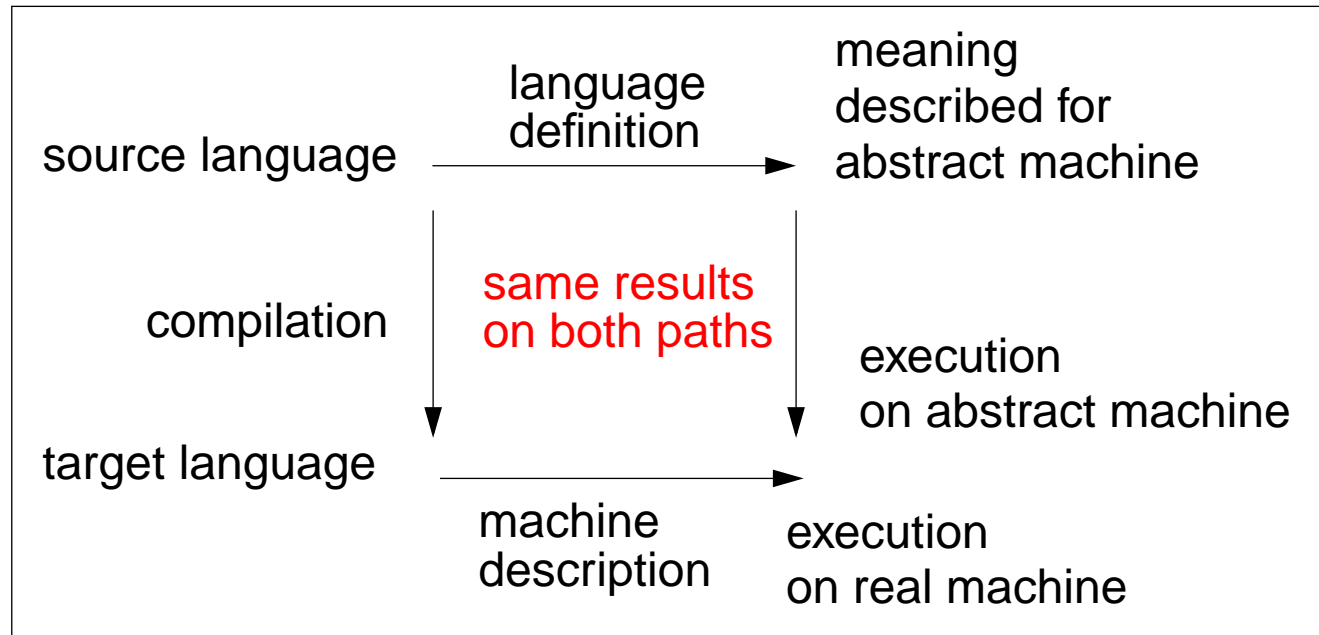# 1. Language properties - compiler tasks
Meaning preserving transformation

A **compiler** transforms **any correct sentence** of its **source language** into a sentence of its **target language** such that its **meaning is unchanged**.

A **meaning** is defined only for **all correct** programs => compiler task: error handling

Static language properties are analyzed at **compile time,** e. g. definitions of Variables, types of expressions; => determine the transformation, if the program **compilable**

Dynamic properties of the program are determined and checked at **runtime,** e. g. indexing of arrays => determine the effect, if the program **executable** (However, just-in-time compilation for Java: bytecode is compiled at runtime.)

Levels of language properties - compiler tasks

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

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

- **c. Static semantics**   **semantic analysis, transformation**
  binding names to program objects, typing rules
  usually defined by informal texts,
  formal definition: **attribute grammar**

- **d. Dynamic semantics**   **transformation, code generation**
  semantics, effect of the execution of constructs
  usually defined by informal texts
  in terms of an abstract machine,
  formal definition: **denotational semantics**

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

Example: Tokens and structure

Character sequence

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

Tokens

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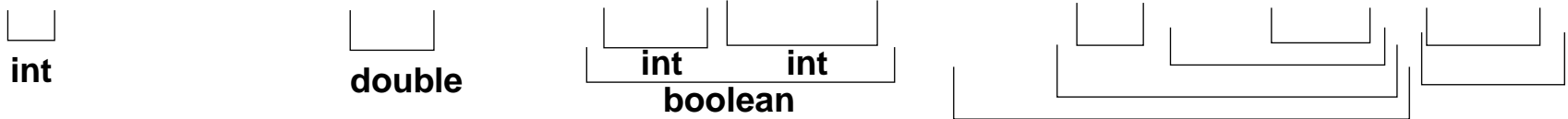
Expressions

Declarations

Statements

Structure

Example: Names, types, generated code

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

Structure

                int   double   int   int
                                                          boolean

                                                                              ...

k1: (count, local variable, int)   **k3: (maxVect, member variable, int)**
k2: (sum, local variable, double)   **k4: (vect, member variable, double array)**

Static properties: names and types

generated Bytecode

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

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

Source program

→ **Lexical analysis**

Token sequence

→ **Syntactic analysis**

Abstract program tree

→ **Semantic analysis**

Transformation

Analysis (frontend)

- -

Intermediate language

→ **Optimization**

Synthesis (backend)

Code generation

Peephole optimization

Abstract machine program

Assembly

Target program
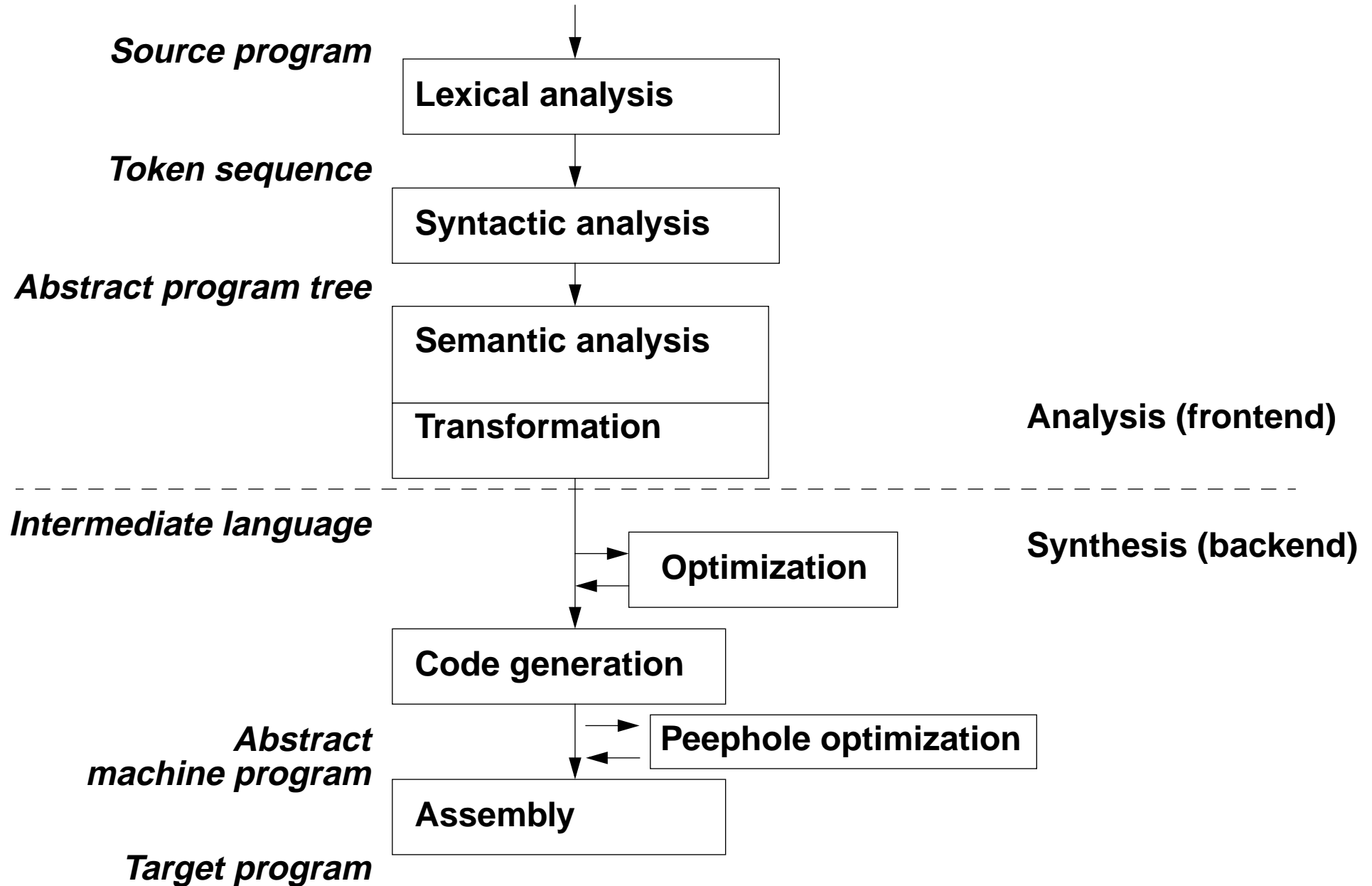
Software qualities of the compiler

- **Correctness**   Compiler translates correct programs correctly;
                          rejects wrong programs and gives 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**   Compiler gives a reasonable reaction on every input;
                          does not break on any program

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**

Generate from specifications

Pattern:
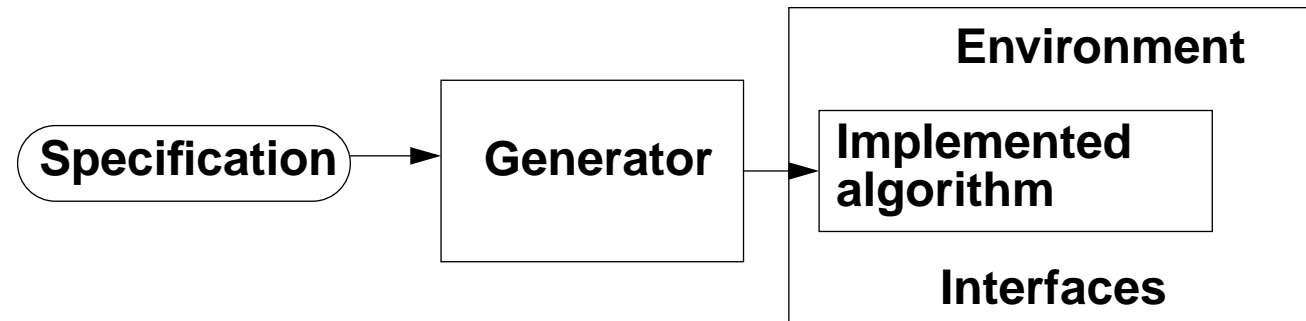
Typical compiler tasks solved by generators:

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:

Compiler Frameworks (Selection)

Amsterdam Compiler Kit: (Uni Amsterdam)
The Amsterdam Compiler Kit is fast, lightweight and retargetable compiler suite and toolchain written by Andrew Tanenbaum and Ceriel Jacobs.
Intermediate language EM, set of frontends and backends

ANTLR: (Terence Parr, Uni San Francisco)
ANother Tool for Language Recognition, (formerly PCCTS) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions

CoCo: (Uni Linz)
Coco/R is a compiler generator, which takes an attributed grammar of a source language and generates a scanner and a parser for this language. The scanner works as a deterministic finite automaton. The parser uses recursive descent.
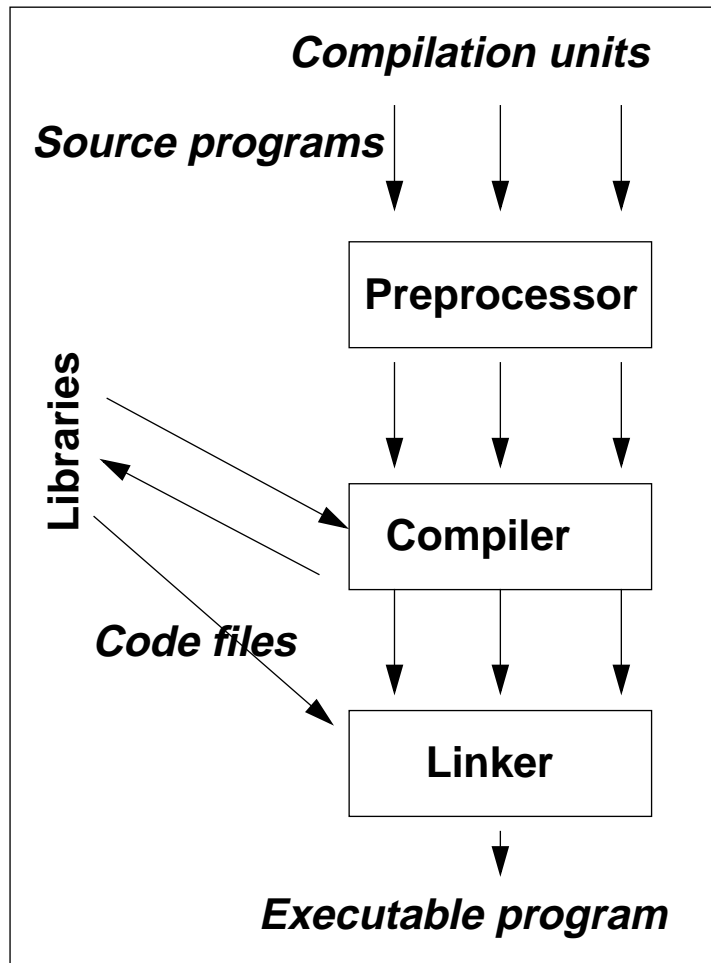
Eli: (Unis Boulder, Paderborn, Sydney)
Combines a variety of standard tools that implement powerful compiler construction strategies into a domain-specific programming environment called Eli. Using this environment, one can automatically generate complete language implementations from application-oriented specifications.

SUIF: (Uni Stanford)
The SUIF 2 compiler infrastructure project is co-funded by DARPA and NSF.
It is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers.

Environment of compilers

Compilation units

Source programs

Libraries

Preprocessor

Compiler

Code files

Linker

Executable program

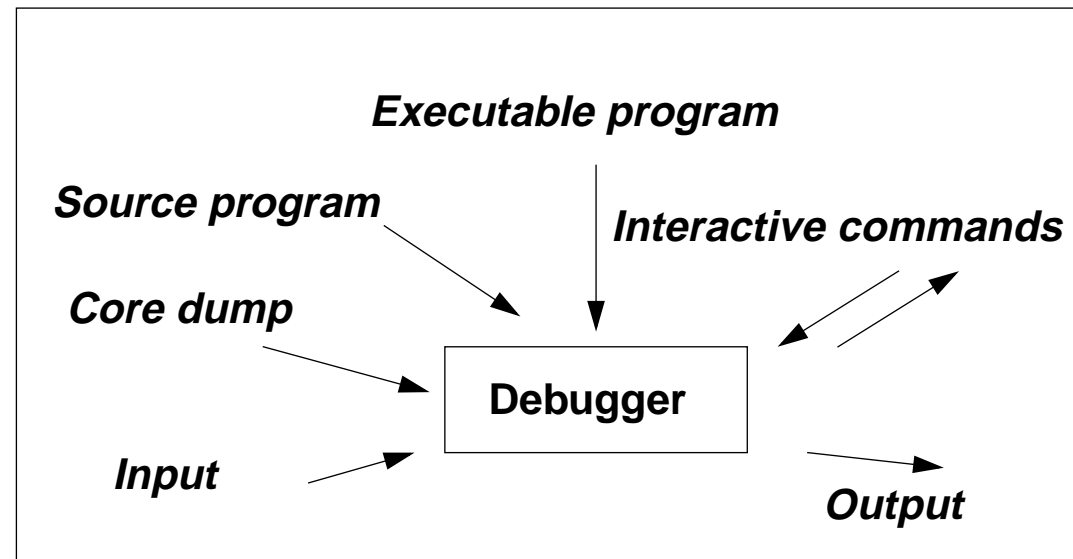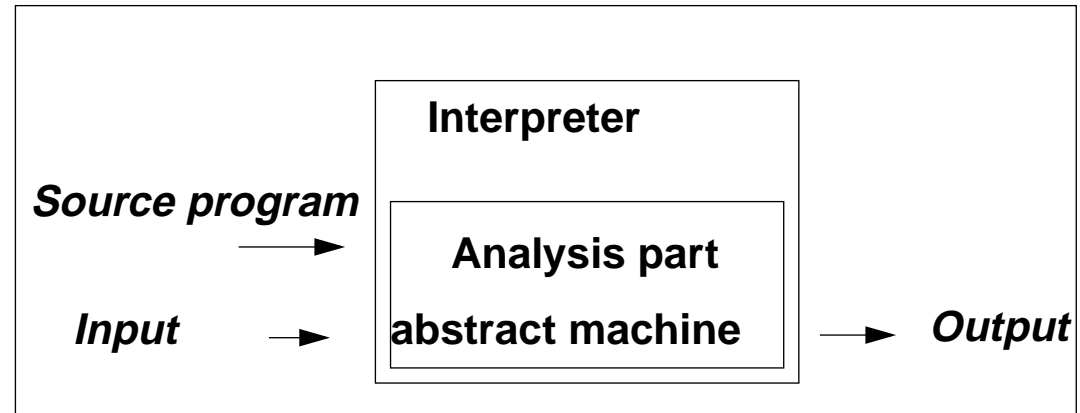**Preprocessor** cpp substitutes text macros in source programs, e.g.

```
#include <stdio.h>
#include "module.h"

#define SIZE 32
#define SEL(ptr,fld) ((ptr)->fld)
```

Separate compilation of compilation units

- with interface specification,
  consistency checks,
  and language specific linker:
  Modula, Ada, Java

- without ...;
  checks deferred to system linker:
  C, C++

Interpreter and Debugger

Interpreter

Source program

Analysis part

abstract machine

Input

Output

Executable program

Source program

Interactive commands

Core dump

Debugger

Input

Output

Compilation and interpretation of Java programs

Source modules

**Java
Compiler**

*needed class files
are loaded dynamically -
local or via Internet*

*Class files
in Java Bytecode
(intermediate language)*

**Interpreter
Java Virtual Machine
JVM**

**Class
loader**

**Bytecode prozessor
in software**

**Just-In-Time
Compiler
(JIT)**

Machine code

Input

Output