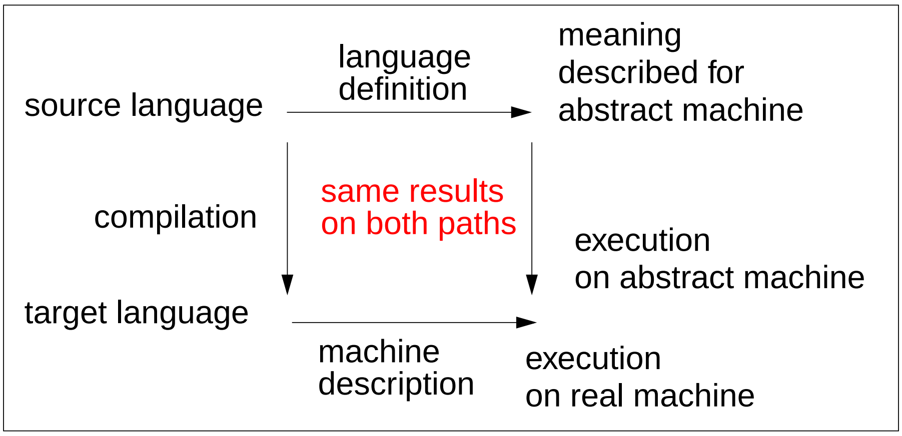
1. Language properties - compiler tasks Meaning preserving transformation

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.)

© 2010 bei Prof. Dr. Uwe Kastens

Levels of language properties - compiler tasks

• **a. Notation of tokens** keywords, identifiers, literals formal definition: **regular expressions**	**lexical analysis**
• **b. Syntactic structure** formal definition: **context-free grammar**	**syntactic analysis**
• **c. Static semantics** binding names to program objects, typing rules usually defined by informal texts, formal definition: **attribute grammar**	**semantic analysis, transformation**
• **d. Dynamic semantics** semantics, effect of the execution of constructs usually defined by informal texts in terms of an abstract machine**,** formal definition: **denotational semantics**	**transformation, code generation**
Definition of target language (target machine)	**transformation, code generation** **assembly**

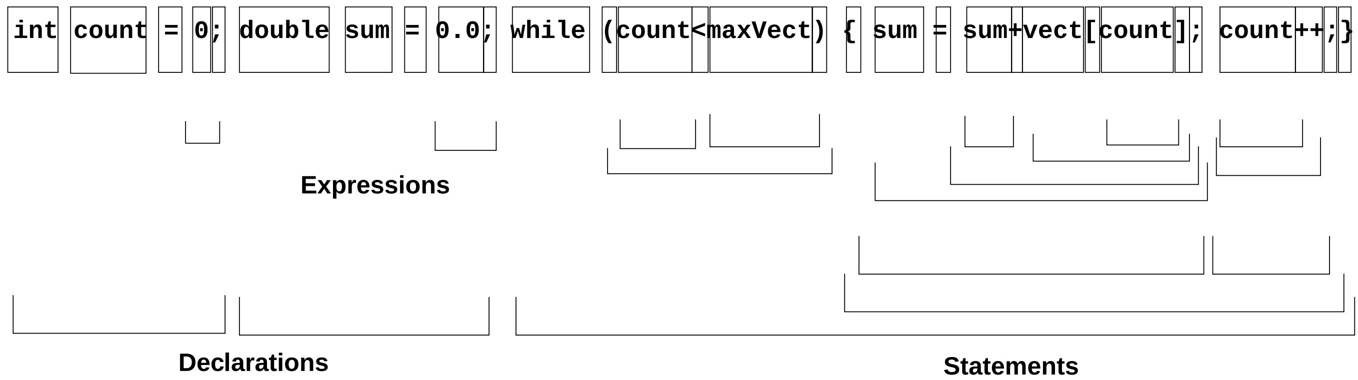
© 2003 bei Prof. Dr. Uwe Kastens

Example: Tokens and structure

Character sequence

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

Tokens

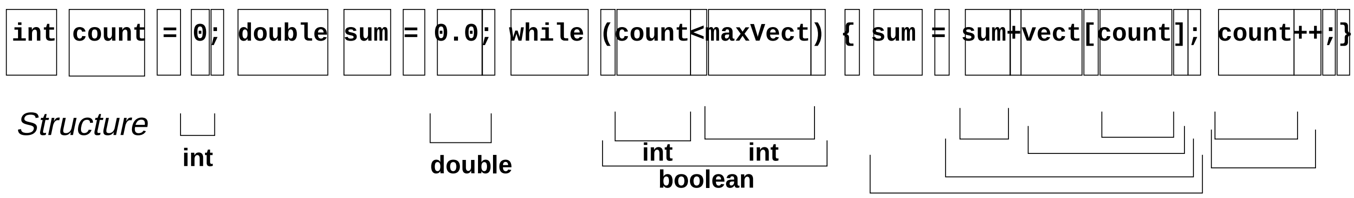


Structure

© 2003 bei Prof. Dr. Uwe Kastens

Example: Names, types, generated code

Structure



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

Static properties: 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
```

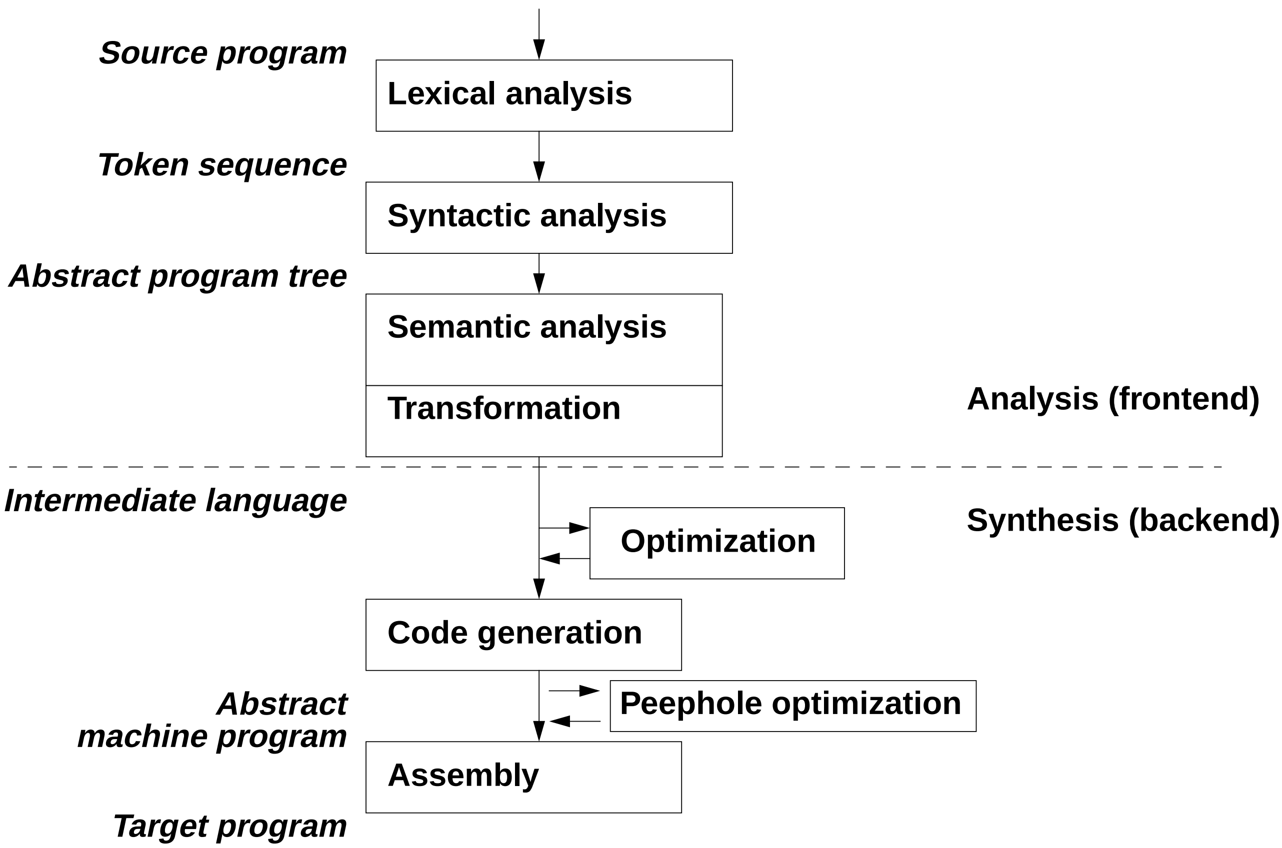
© 2003 bei Prof. Dr. Uwe Kastens

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

© 2003 bei Prof. Dr. Uwe Kastens

Compiler structure and interfaces



© 2003 bei Prof. Dr. Uwe Kastens

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

© 2003 bei Prof. Dr. Uwe Kastens

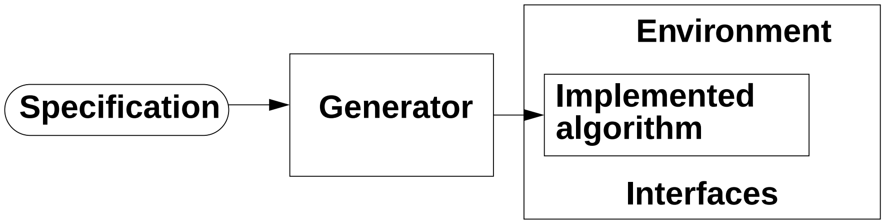
Strategies for compiler construction

- **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**

© 2003 bei Prof. Dr. Uwe Kastens

Generate from specifications

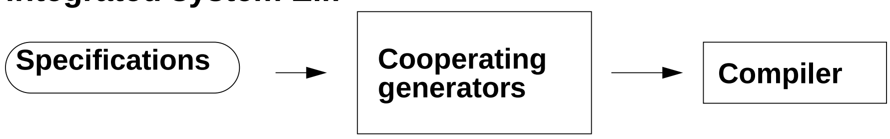
Pattern:



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:



© 2003 bei Prof. Dr. Uwe Kastens

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

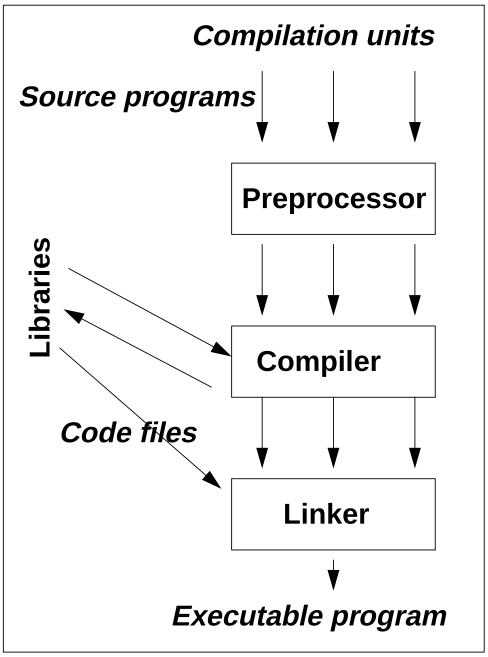
CoCo: (Uni Linz)
Coco/R is a compiler generator, which takes an attributed grammar of a source language and generates a scanner and a parser for this language. The scanner works as a deterministic finite automaton. The parser uses recursive descent.

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.

© 2007 bei Prof. Dr. Uwe Kastens

Environment of compilers



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

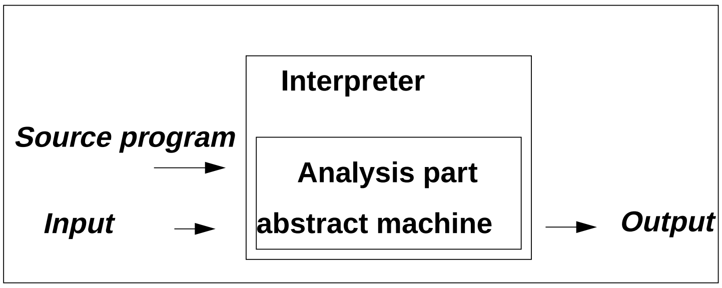
```
#include <stdio.h>
#include "module.h"

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

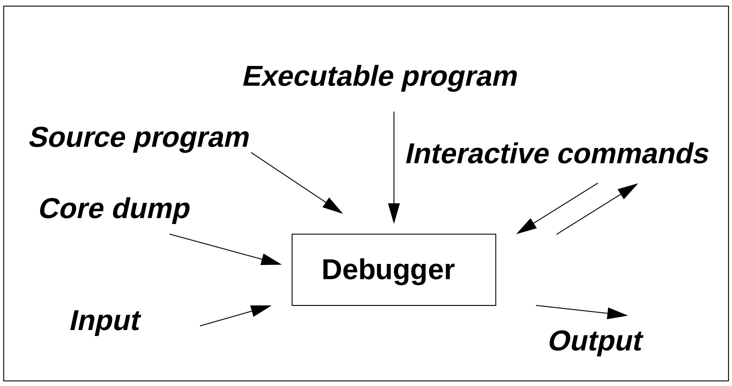
Separate compilation of compilation units

- with interface specification, consistency checks, and language specific linker: Modula, Ada, Java
- without ...; checks deferred to system linker: C, C++

© 2007 bei Prof. Dr. Uwe Kastens

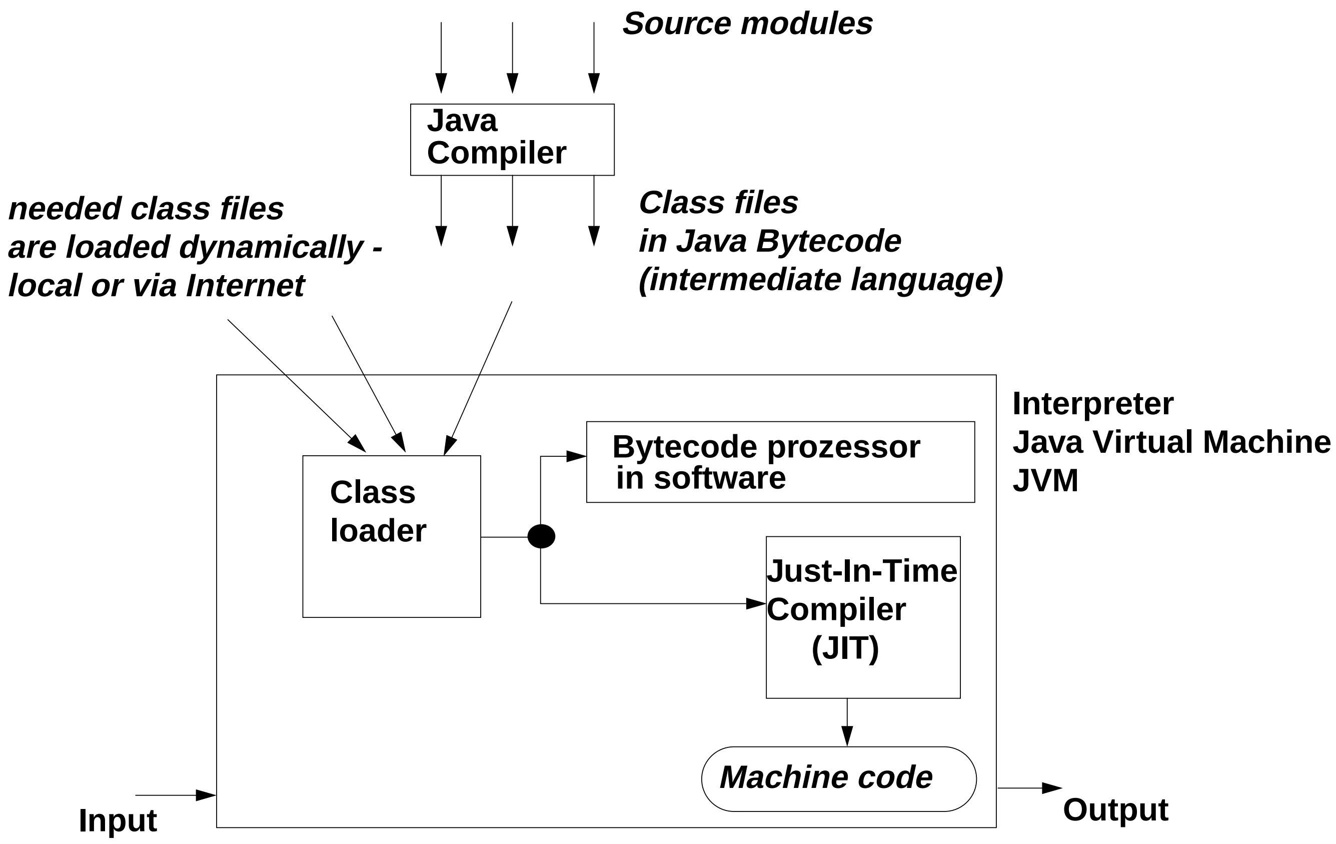
Interpreter and Debugger





© 2005 bei Prof. Dr. Uwe Kastens

Compilation and interpretation of Java programs



© 2003 bei Prof. Dr. Uwe Kastens