Programming Languages and Compilers

Language Properties and Compiler Tasks

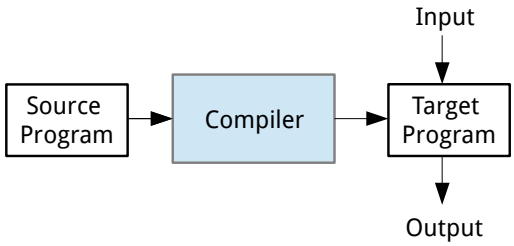
Dr. Peter Pfahler
Based on the lecture by Prof. Dr. Uwe Kastens

Universität Paderborn
Fakultät EIM
Institut für Informatik

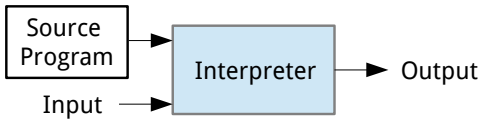
Winter 2016/2017

Compilation and Interpretation

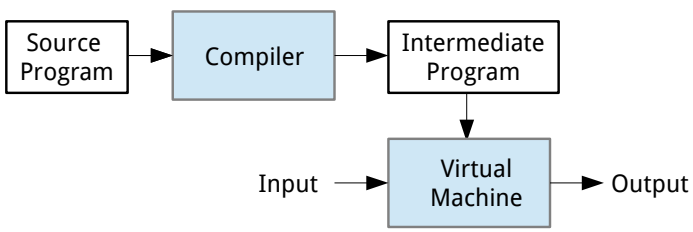
- **Compilation**
  A *compiler* translates a program in the source language into an equivalent program in the target language. Important task: report errors.



- **Interpretation**
  An *interpreter* appears to directly execute a source language program on a given input. Internally typical compilation tasks may have to be solved.



- **Hybrid Compilation**
  The compiler translates a program in the source language into an equivalent intermediate program which is interpreted by a *virtual machine* .



Levels of Language Properties related to Compiler Tasks

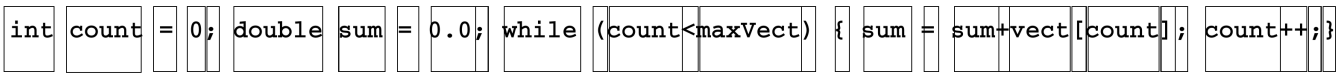
Level	Formal Specification	Compiler Task
Notation of Tokens	Regular Expressions	Lexical Analysis
Syntactic Structure	Context-free Grammars	Syntactic Analysis
Static Semantics • Name Binding • Typing Rules	• Often specified informally. • Attribute Grammars.	Semantic Analysis, Transformation
Dynamic Semantics	• Often specified informally in terms of an abstract machine. • Denotational Semantics.	Transformation, Code Generation

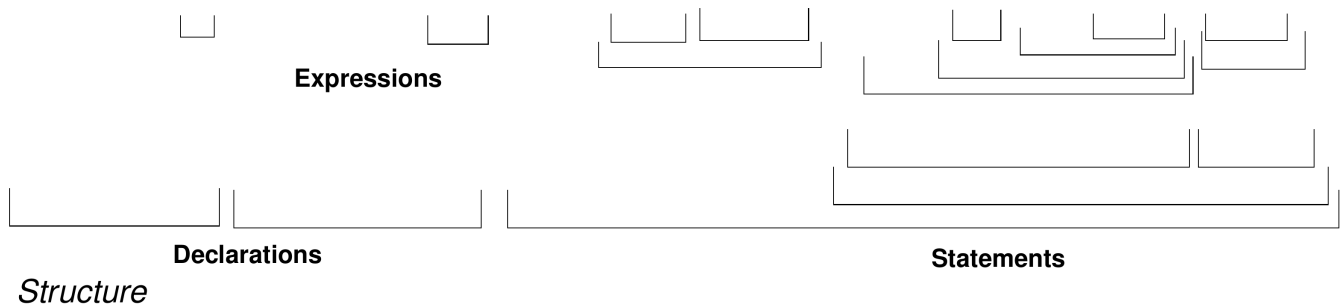
Example: Tokens and Structure

Character sequence

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

Tokens





Structure

Example: Names, Types, Generated code

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

Static properties: names and types

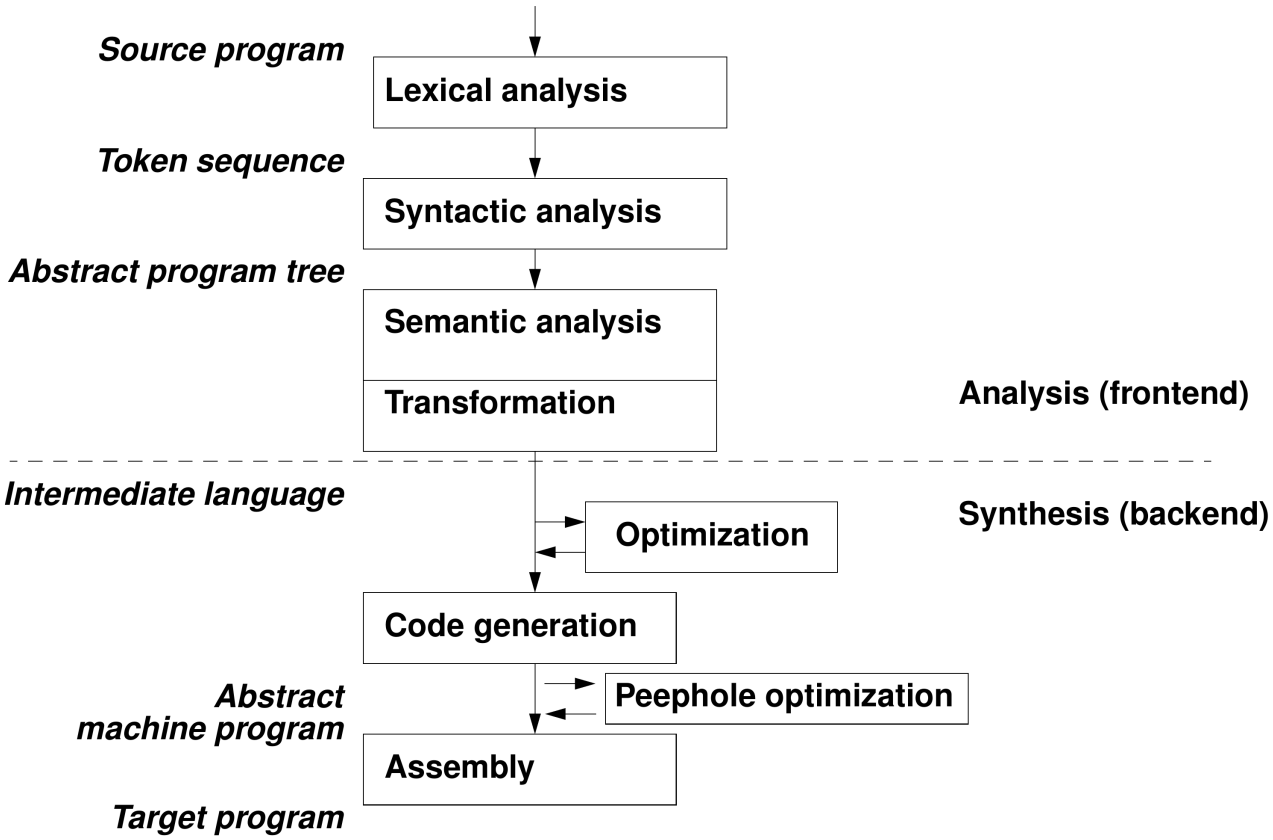
Generated Bytecode:

```
 0: iconst_0
 1: istore_1
 2: dconst_0
 3: dstore_2
 4: iload_1
 5: getstatic     #2                  // Field maxVect:I
 8: if_icmpge     25
11: dload_2
12: getstatic     #3                  // Field vect:[D
15: iload_1
16: daload
17: dadd
18: dstore_2
19: iinc          1, 1
22: goto          4
25: return
```


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

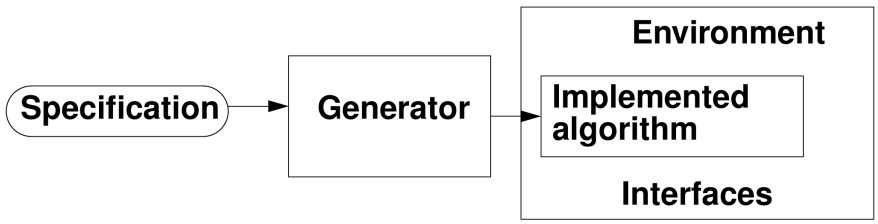
- Correctness
  Compiler translates correct programs correctly. Rejects wrong programs and gives error messages.
- Efficiency
  Storage and time used by the compiler.
- Code efficiency
  Storage and time used by the generated code.
  Compiler task: optimization
- User support
  Compiler task: Error handling (recognition, message, recovery).
- Robustness
  Compiler gives a reasonable reaction on every input. It does never break.

Strategies for Compiler Construction

- Adhere closely to the language specification
- Use generating tools
- Use standard components
- Apply standard methods
- Validate the compiler against a test suite
- Verify components of the compiler

Generating Compiler Components

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

Compiler Tools (Selection)

Compiler Generators

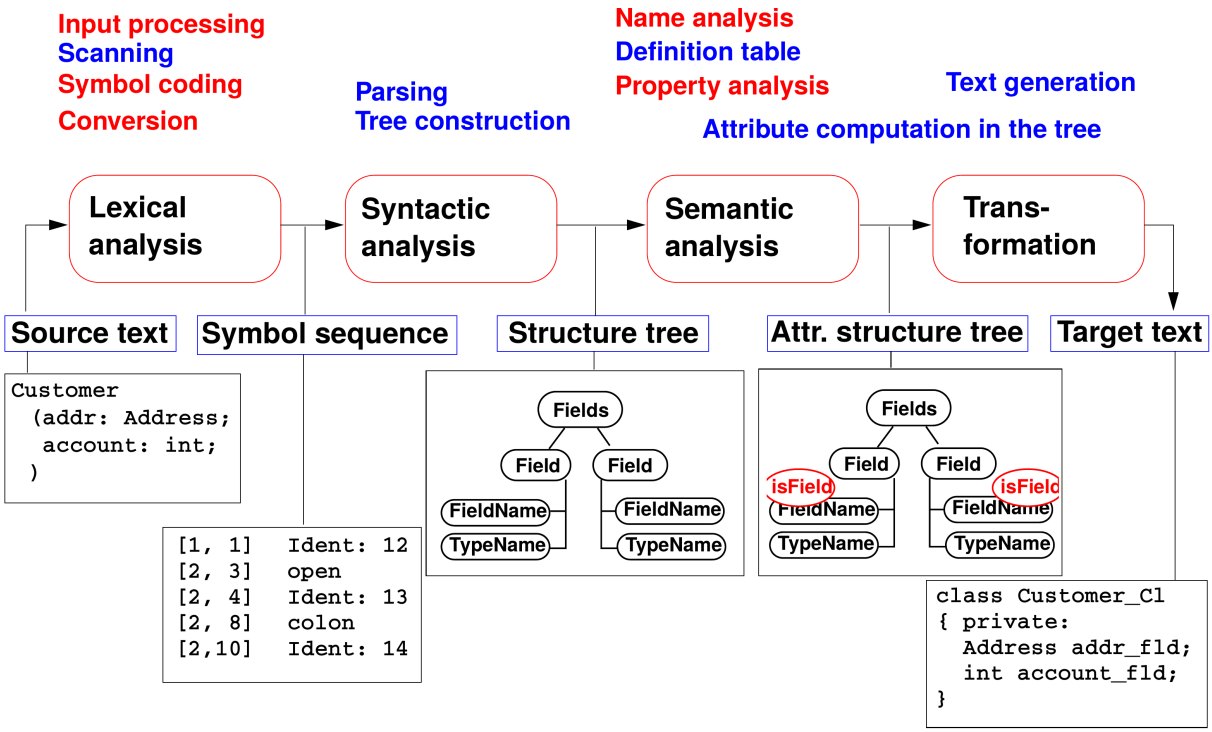
- ANTLR (University of San Francisco)
  generates lexers, parsers and tree traversal support. It uses adaptive LL(*) parsing and generates language tools in Java and C# .
- CoCo/R (Universität Linz)
  generates scanners and parsers from L-attributed grammars. It uses recusive-descent LL(1) parsing extended by multi-symbol lookahead (LL(k)).
- Eli (Universities Boulder, Sydney, Paderborn)
  Compiler development environment integrating various generators and support libraries. Uses LALR(1) parsing and visit-sequence-based attribute evaluation.

Compiler Infrastructure Projects

- SUIF (Standford University)
  Modular compiler system with various frontends, an extensible intermediate language, and a backend infrastructure for analysis, optimization and code generation.
- LLVM (University of Illinois at Urbana Champaign)
  Compiler infrastructure based upon reusable libraries with well-defined interfaces. Provides various frontends, backends and compiler optimizations.

Architecture of a Compiler generated by Eli

Task decomposition determines the system architecture. Specialized tools generate solutions for the sub-tasks:



Using Eli

- **Invoking an interactive Eli session**
  `eli [-c cache-location] [-r]`
  - Eli stores intermediate results in a directory called "cache" and reuses them from there. The default value is `~/.ODIN` . Local cache directory is recommended.
  - `-r` resets the cache, `-R` recreates the cache
- **Eli documentation**
  to be found on `http://eli-project.sourceforge.net/elionline/`
- **Eli command examples**
  - Generate an executable language processor and display errors and warnings:
    `myspecs.fw : exe : warning>`
  - Generate an executable language processor and place it in the current directory:
    `myspecs.fw : exe > .`
  - Generate the processor's source code and place it in a directory named `src` :
    `myspecs.fw : source > src`
  - Generate documentation for the language project and place it in the file `mydoc.pdf` :
    `myspecs.fw : pdf > mydoc.pdf`
- **Literate programming**
  The language "FunnelWeb" integrates specifications (written in Eli's specification languages) and documentation (written in LaTeX, e.g.). Eli uses FunnelWeb texts (.fw) to generate both executable language processors and their documentation.

A FunnelWeb Example: Input file

```
@p maximum_input_line_length = infinity
@p typesetter = latex
\documentclass[a4paper,12pt]{scrartcl}
\usepackage{alltt}
\title{Hello World -- A First FunnelWeb Example}

\begin{document}
\maketitle
The file ‘‘helloworld.lido’’ contains a minimal Eli specification which
accepts only the empty input file and prints ‘‘Hello, World’’ to the
standard output.

@O@<helloworld.lido@>@{@-
RULE:   root ::= COMPUTE
    printf("Hello, World\n");
END;
@}
\end{document}
```

A FunnelWeb Example: Generating processor and documentation

Eli commands

```
hello.fw : pdf > hellodoc.pdf

hello.fw : exe > hello.exe
```

Hello World – A First FunnelWeb Example

May 7, 2015

The file "helloworld.lido" contains a minimal Eli specification which accepts only the empty input file and prints "Hello, World" to the standard output.

helloworld.lido[1]:

```
RULE:   root ::= COMPUTE
    printf("Hello, World\n");
END;
```

This macro is attached to a product file.