

# 1. Introduction

## Domain-Specific Knowledge

A **task**: „Implement a program to store collections of words, that describe animals“

**Categories of knowledge** required to carry out a task:

**General:** knowledge applicable to a wide variety of tasks  
e.g. English words; program in C

**Domain-specific:** knowledge applicable to all tasks of this type  
e.g. group word in sets;  
implement arbitrary numbers of sets of strings in C

**Task-specific:** knowledge about the particular task at hand  
e.g. sets of words to characterize animals

A domain-specific language is used to describe the particular task

A domain-specific generator creates a C program that stores the particular set of strings.

# Example for a Domain-Specific Generator

Input: collection of words:

```
colors{red blue green}
bugs{ant spider fly moth bee}
verbs{crawl walk run fly}
```

- simple domain-specific description
- errors easier to detect in the domain-specific description
- a number of tasks of the same kind
- constraints on representation using general knowledge require a more complex and detailed description (implementation)
- consistency conditions in the representation using general knowledge are difficult to check

Output: C header file:

```
int number_of_sets = 3;
char *name_of_set[] = {
"colors",
"bugs",
"verbs"};

int size_of_set[] = {
3,
5,
4};

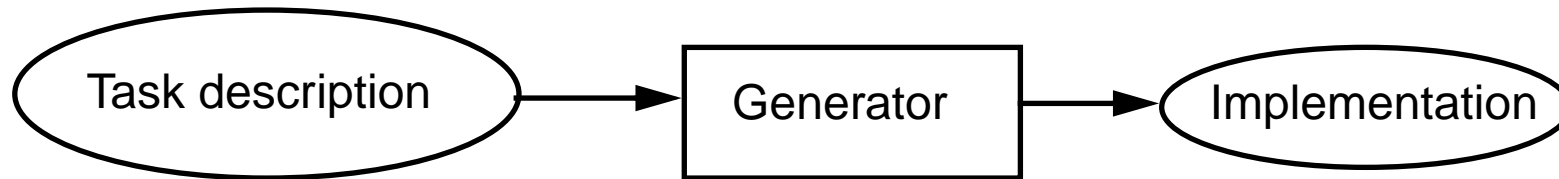
char *set_of_colors[] = {
"red",
"blue",
"green"};

char *set_of_bugs[] = {
"ant",
"spider",
"fly",
"moth",
"bee"};

char *set_of_verbs[] = {
"crawl",
"walk",
"run",
"fly"};

char **values_of_set[] = {
set_of_colors,
set_of_bugs,
set_of_verbs};
```

# The Generator Principle



**Application generator:** the most effective reuse method

[Ch. W. Kruger: Software Reuse]

**narrow, specific application domain**

completely understood

Implementation automatically generated

**Abstractions on a high level**

(using domain knowledge)

transformed into executable software

**User** understands

**abstractions** of the application domain

**Generator expert** understands

**implementation methods**

wide cognitive distance

**generator makes expert knowledge available**

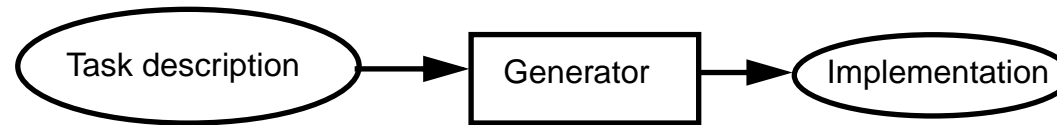
**Examples:**

Data base report generator

GUI generator

Parser generator

# Domain-Specific Languages for Generators



## Domain-specific languages (DSL)

### Domains outside of informatics

- Robot control
- Stock exchange
- Control of production lines
- Music scores

### Software engineering domains

- Data base reports
- User interfaces
- Test descriptions
- Representation of data structures (XML)

### Language implementation as domain

- Scanner specified by regular expressions
- Parser specified by a context-free grammar
- Language implementation specified for *Eli*

**Generator:**      **transforms a specification language**  
 into an executable **program or/and into data**,  
 applies domain-specific methods and techniques

## Some GSS Projects

- Party organization
- Soccer teams
- Tutorial organization
- Shopping lists
- Train tracks layout
  
- LED descriptions to VHDL
- SimpleUML to XMI
- Rule-based XML transformation

# Reuse of Products

## Product

Library of functions

Module, component

generic module

Software architecture

Framework

Design pattern

Generator

Construction process

## What is reused?

Implementation

Code

Planned variants of code

Design

Design and code

Strategy for design and construction

Knowledge, how to construct  
implementations from descriptions

Knowledge, how to use and  
combine tools to build software

Ch. W. Kruger: Software Reuse, ACM Computing Surveys, 24(2), 1992

R. Prieto-Diaz: Status Report: Software reusability, IEEE Software, 10(3), 1993

# Organisation of Reuse

How	Products	Consequences
<b>ad hoc</b>	<ul style="list-style-type: none"><li>• Code is copied and modified</li><li>• adaptation of OO classes incrementally in sub-classes</li></ul>	<ul style="list-style-type: none"><li>• no a priori costs</li><li>• very dangerous for maintenance</li></ul>
<b>planned</b>	<ul style="list-style-type: none"><li>• oo libraries, frameworks</li><li>• Specialization of classes</li></ul>	<ul style="list-style-type: none"><li>• high a priori costs</li><li>• effective reuse</li></ul>
<b>automatic</b>	<ul style="list-style-type: none"><li>• Generators, intelligent development environments</li></ul>	<ul style="list-style-type: none"><li>• high a priori costs</li><li>• very effective reuse</li><li>• wide cognitive distance</li></ul>

# Roles of Provider and Reuser

## Reusable products are

- Constructed and prepared for being reused.      Role: provider
- Reused for a particular application.      Role: reuser

## Provider and reuser are on the same level of experience:

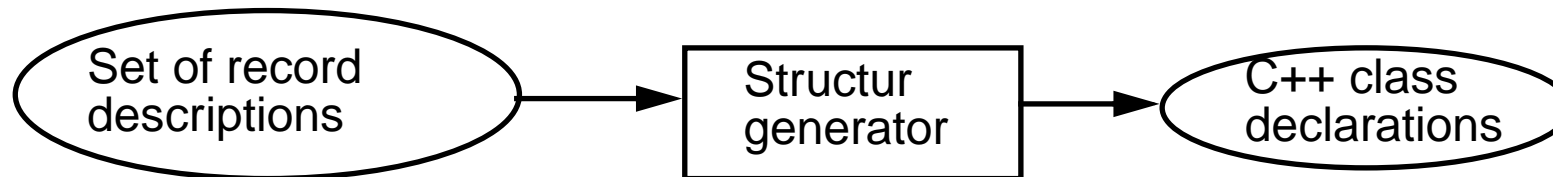
- The **same person**, group of persons, profession
- Provider assumes his own level of understanding for the reuser
- Examples: reuse of code, design patterns

## Provider is an expert, reusers are amateurs:

- Reuse bridges a **wide cognitive distance**
- **Expert knowledge** is made available for **non-experts**
- Application domain has to be **completely understood** by the expert; **that knowledge is then encapsulated**
- Requires domain-specific **notions on a high level**
- Examples: **Generators, frameworks, intelligent development environments**

# Project: Structure Generator (Lect. Ch. 8, Book Ch. 7)

Generator implements described record structures  
useful tool in software construction



```

Customer ( addr:      Address;
           account:  int; )

Address ( name:  String;
          zip:   int;
          city:  String; )

import String from "util.h"

```

```

#include "util.h"

typedef class Customer_C1 *Customer;
typedef class Address_C1 *Address;

class Customer_C1 {
    private:
        Address addr_fld;
        int account_fld;
    public:
        Customer_C1
            (Address addr, int account)
        {
            addr_fld=addr;
            account_fld=account;
        }
    ...
};

```



# Task Decomposition for the Implementation of Domain-Specific Languages

<b>Structuring</b>	<b>Lexical analysis</b>	<b>Scanning</b> <b>Conversion</b>
	<b>Syntactic analysis</b>	<b>Parsing</b> <b>Tree construction</b>
<b>Translation</b>	<b>Semantic analysis</b>	<b>Name analysis</b> <b>Property analysis</b>
	<b>Transformation</b>	<b>Data mapping</b> <b>Action mapping</b>

*[W. M. Waite, L. R. Carter: Compiler Construction, Harper Collins College Publisher, 1993]*

Corresponds to task decomposition for  
**frontends** of compilers for programming languages (no machine code generation)  
**source-to-source** transformation

# Design and Specification of a DSL

<b>Structuring</b>	<b>Lexical analysis</b>	<b>Design the notation of tokens Specify them by regular expressions</b>
	<b>Syntactic analysis</b>	<b>Design the structure of descriptions Specify it by a context-free grammar</b>
<b>Translation</b>	<b>Semantic analysis</b>	<b>Design binding rules for names and properties of entities. Specify them by an attribute grammar</b>
	<b>Transformation</b>	<b>Design the translation into target code. Specify it by text patterns and their instantiation</b>

```
Customer ( addr:    Address;
           account: int; )
```

```
Address ( name:  String;
          zip:    int;
          city:  String; )
```

```
import String from "util.h"
```

# Task Decomposition for the Structure Generator

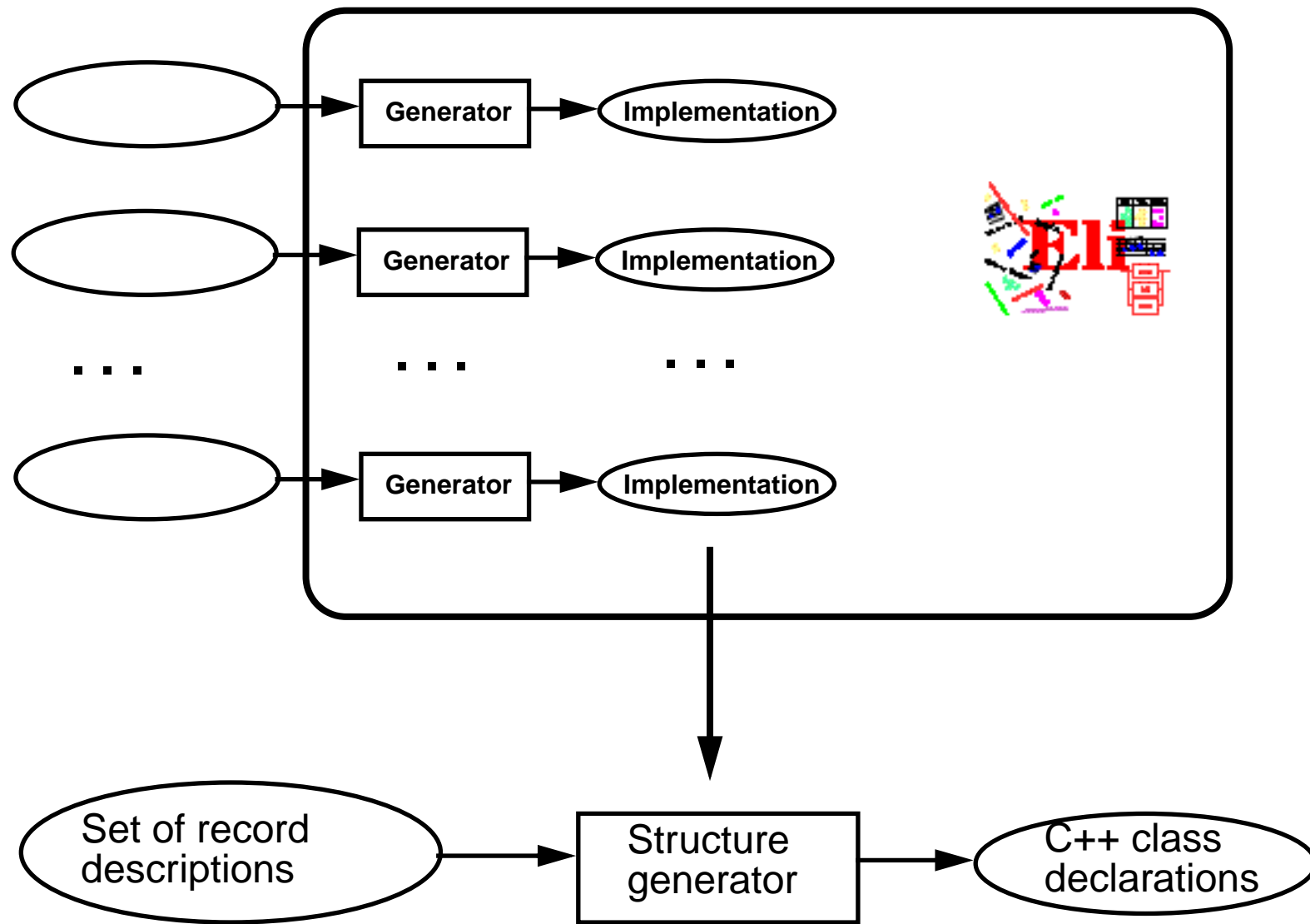
<b>Structuring</b>	<b>Lexical analysis</b>	<b>Recognize the symbols of the description Store and encode identifiers</b>
	<b>Syntactic analysis</b>	<b>Recognize the structure of the description Represent the structure by a tree</b>
<b>Translation</b>	<b>Semantic analysis</b>	<b>Bind names to structures and fields Store properties and check them</b>
	<b>Transformation</b>	<b>Generate class declarations with constructors and access methods</b>

```
Customer ( addr:    Address;  
           account: int; )
```

```
Address ( name:  String;  
          zip:    int;  
          city:  String; )
```

```
import String from "util.h"
```

# Eli Generates a Structure Generator



# Task Decomposition Determines the Architecture of the Generator

Specialized tools solve specific sub-tasks for creating of the product:

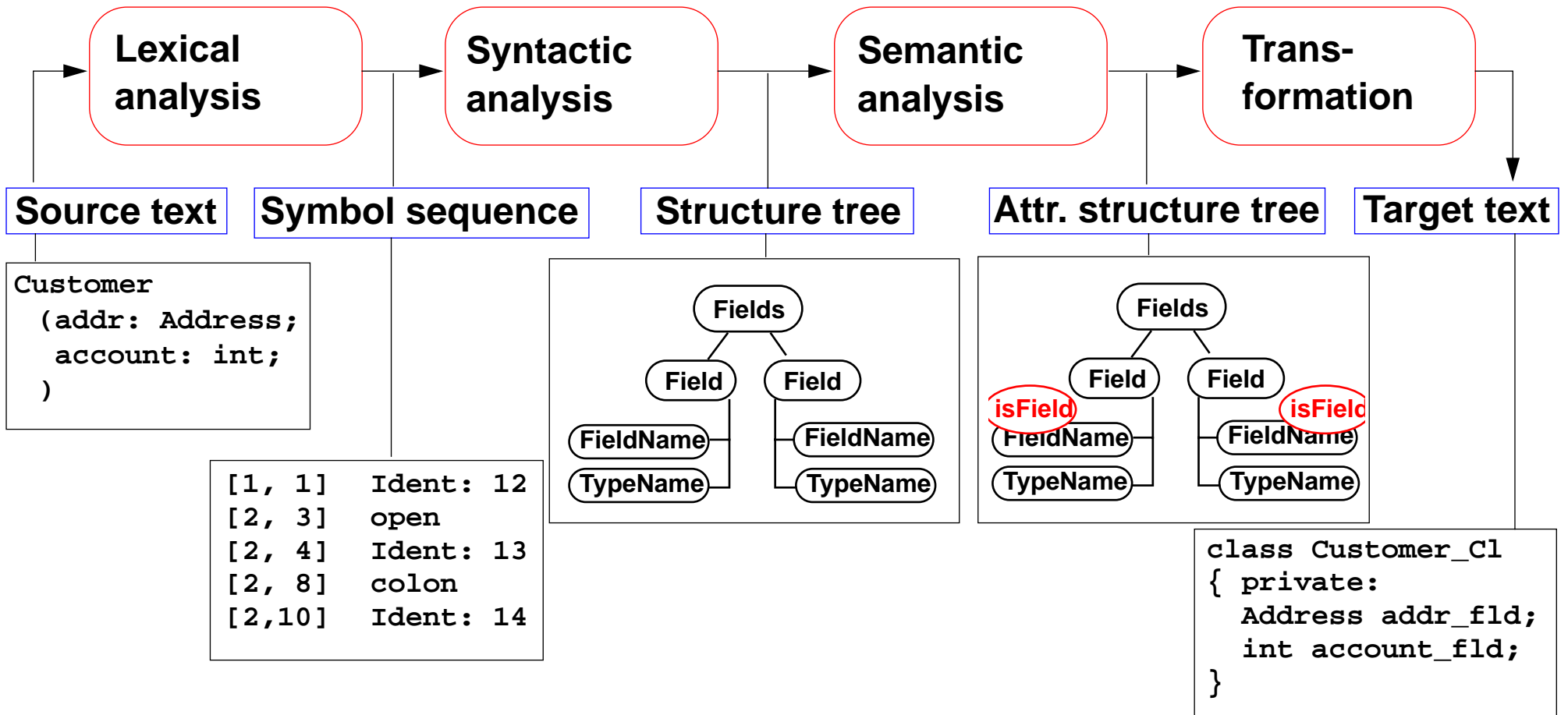
**Input processing**  
**Scanning**  
**Symbol coding**  
**Conversion**

**Parsing**  
**Tree construction**

**Name analysis**  
**Definition table**  
**Property analysis**

**Attribute computation in the tree**

**Text generation**



# The Eli System

- **Framework for language implementation**
- Suitable for any kind of textual language:  
**domain-specific languages**,  
programming languages
- **state-of-the-art compiler technique**
- Based on the (complete)  
**task decomposition** (cf. GSS-1.9)
- **Automatic construction process**
- Used for many **practical projects** world wide
- Developed, extended, and maintained since 1989 by  
William M. Waite (University of Colorado at Boulder),  
Uwe Kastens (University of Paderborn), and  
Antony M. Sloane (Macquarie University, Sydney)
- **Freely available** via Internet from  
<http://eli-project.sourceforge.net>



# Hints for Using Eli

## 1. Start Eli:

```
/comp/eli/current/bin/eli [-c cacheLocation][-r]
```

Without `-c` a cache is used/created in directory `~/ .ODIN`. `-r` resets the cache

## 2. Cache:

Eli stores all intermediate products in cache, a tree of directories and files.

Instead of recomputing a product, Eli reuses it from the cache.

The cache contains only derived data; can be recomputed at any time.

## 3. Eli Documentation:

*Guide for New Eli Users*: Introduction including a little tutorial

*Products and Parameters* and *Quick Reference Card*: Description of Eli commands

*Translation Tasks*: Conceptual description of central phases of language implementation.

*Reference Manuals, Tools and Libraries in Eli, Tutorials*

## 4. Eli Commands:

A common form: Specification : Product > Target e.g.

```
Wrapper.fw : exe > .
```

from the specification derive the executable and store it in the current directory

```
Wrapper.fw : exe : warning >
```

from ... derive the executable, derive the warnings produced and show them

## 5. Eli Specifications: A set of files of specific file types.

## 6. Literate Programming: FunnelWeb files comprise specifications and their documentation