

# Programming Languages and Compilers

## Source-to-source translation

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

Universität Paderborn  
Fakultät EIM  
Institut für Informatik

Winter 2016/2017

## Source-to-Source Translation

### Source-to-source translation:

Translation of a **high-level source language** into a **high-level target language**.

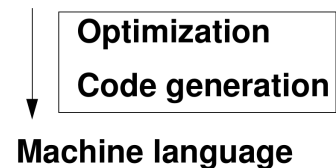
#### Source-to-source translator:

Specification language (SDL, UML, ...)  
Domain specific language (SQL, STK, ...)  
high-level programming language



#### Compiler:

Programming language



# The Transformation Phase in Source-to-Source Translation

**Input:** Attributed Abstract Syntax Tree

Tasks	Compiler Module
Target Tree Construction	Attribute Evaluator, Target Tree Constructors
Target Tree Linearization	Target Text Generator

**Output:** Target Text

The Transformation Phase is performed guided by the contexts of the AST:

**Computation Model:** Dependent Computations in Trees

**Specification:** *Attribute Grammar* and *Structured Target Text Description*

**Generator:** *Attribute Evaluator Generator* and *Generator for Target Trees/Texts*

In Eli: PTG ("Pattern Based Text Generator")

## A Generator for Producing Structured Target Texts

PTG: Pattern-based Text Generator

Generates target tree constructors from specifications of the target text structure. The target tree can be linearized to text files.

- 1 Specifying Target Text Templates using Insertion Points:

```
ProgramFrame: $1 "void main () {\n" $2 "}\n"
Exit:         "exit (" $1 int ");\n"
IOInclude:    "#include <stdio.h>"
```

- 2 PTG generates a constructor function for each pattern:

```
PTGNode a, b, c;
a = PTGIOInclude();
b = PTGExit(5);
c = PTGProgramFrame(a, b);
```

- 3 Output of the target structure:

```
PTGOutFile ("Output.c", c);
```

## Example: Generate Data Type Definitions in C

We specify a source-to-source translator from abstract type definitions like

```
bintree = bintree X int X bintree
```

to recursive type definitions in C:

```
typedef struct _bintree *bintree;
typedef struct _bintree {
    bintree field_1;
    int field_2;
    bintree field_3;
} bintree_elem;
```

### Proceeding

- Design target text
- Design source syntax
- Compute necessary properties of language entities (e.g. field number)
- Specify target tree construction, arrange for target text output

## Example: Design Target Text

PTG specification for recursive type definitions in C:

```
cstructs:
    "typedef struct _" $1 string " *" $1 string ";\n"
    "typedef struct _" $1 string " {\n" $2 "}" $1 string "_elem;\n"
field:
    "    " $1 string " field_" $2 int ";\n"
seq: $ $
```

## Example: Design Source Syntax

Lido specification of the abstract source language syntax:

```
RULE: struct ::= Ident '=' fields
END;

RULE: fields ::= field
END;

RULE: fields ::= fields 'X' field
END;

RULE: field ::= Ident
END;
```

## Example: Compute Field Numbers

Numbering fields from left to right:

```
ATTR fieldnumber : int;

RULE: fields ::= field COMPUTE
      fields.fieldnumber = 1;
      field.fieldnumber = 1;
END;

RULE: fields ::= fields 'X' field COMPUTE
      fields[1].fieldnumber = ADD(fields[2].fieldnumber,1);
      field.fieldnumber = fields[1].fieldnumber;
END;
```

## Example: Compute Target Tree

Tree construction and output of target text to file "out.c":

```
ATTR code: PTGNode;

RULE: struct ::= Ident '=' fields COMPUTE
      PTGOutFile("out.c",
                  PTGcstructs(StringTable(Ident),
                              fields.code));

END;
RULE: fields ::= field COMPUTE
      fields.code = field.code;

END;
RULE: fields ::= fields 'X' field COMPUTE
      fields[1].code = PTGseq(fields[2].code, field.code);

END;
RULE: field ::= Ident COMPUTE
      field.code = PTGfield(StringTable(Ident),
                           field.fieldnumber);

END;
```