8. An Integrated Approach: Structure Generator Task Description

The structure generator takes **decriptions of structures with typed fields** as input, and generates an **implementation by a class in C++** for each structure. (see slides GSS 1.8 to 1.10)

1. An input file describes **several structures with its components**.
2. Each **generated class** has an **initializing constructor**, and a **data attribute**, a **set-** and a **get-method for each field.**
3. The **type** of a field may be **predefined**, a **structure** defined in the processed file, or an **imported** type.
4. The generator is intended to **support software development**.
5. **Generated classes have to be sufficiently readable**, s.th. they may be adapted manually.
6. The **generator is to be extensible**, e.g. reading and writing of objects.
7. The description language shall allow, that the **fields of a structure can be accumulated** from several descriptions of one structure.

Objectives:

Agree upon the task

In the lecture:

The items are explained.

Example for the Output of the Structure Generator

Import of externally defined strucures:

```
#include "util.h"
```

Forward references:

```
typedef class Customer_Cl *Customer;
typedef class Address_Cl *Address;
```

Class declaration, Fields, Initializing constructor, set- and get-methods for fields:

```
class Customer_Cl {
private:
   Address addr_fld;
   int account_fld;
public:
   Customer_Cl (Address addr, int account)
      {addr_fld=addr; account_fld=account; }
   void set_addr (Address addr)
      {addr_fld=addr;}
   Address get_addr ()
      {return addr_fld;}
   void set_account (int account)
      {account_fld=account;}
   int get_account ()
      {return account_fld;}
};
```

Further class declarations:

```
class Address_Cl {
...
```


Objectives:

Describe the generated results

In the lecture:

The items are explained.

Variants of Input Form

closed form:

sequence of struct descriptions,
each consists of a
sequence of field descriptions

```
Customer( addr:    Address;
          account: int;
        )
Address ( name:  String;
          zip:   int;
          city:  String;
        )
import String from "util.h"
```

several descriptions for the same struct accumulate the field descriptions

```
Address ( zip:   int;
          phone: int;
        )
```

open form:

sequence of qualified field descriptions

```
Customer.addr:  Address;
Address.name: String;
Address.zip: int;
import String from "util.h"
Customer.account: int;
```

several descriptions for the same struct accumulate the field descriptions

```
Address.zip: int;
Address.phone: int;
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 803

Objectives:

Discuss alternative input variants early

In the lecture:

The items are explained.

Task Decomposition for the Structure Generator

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

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

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

import String from "util.h"
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 804

Objectives:

Overview over subtasks

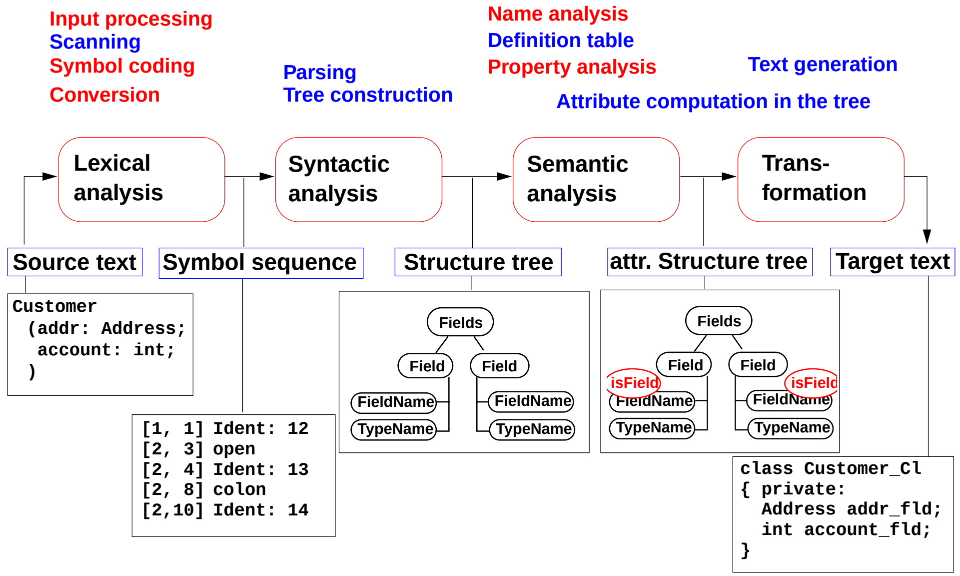
In the lecture:

The items are explained.

GSS-1.12 / 8.5

Task Decomposition Determines the Architecture of the Generator

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



Lecture Generating Software from Specifications WS 2013/14 / Slide 805

Objectives:

Structure of the generator

In the lecture:

The items are explained.

GSS-8.6

Concrete Syntax

Straight-forward natural description of language constructs:

```
Descriptions: (Import / Structure)*.
Import:       'import' ImportNames 'from' FileName.
ImportNames:  ImportName // ','.
Structure:    StructureName '(' Fields ')'.
Fields:       Field*.
Field:        FieldName ':' TypeName ';'.
```

Different nonterminals for identifiers in different roles:,

```
StructureName: Ident.
ImportName:    Ident.
FieldName:     Ident.
TypeName:      Ident.
```

Token specification:

```
Ident:     PASCAL_IDENTIFIER
FileName:  C_STRING_LIT
           C_COMMENT
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 806

Objectives:

Straight-forward specification

In the lecture:

The items are explained.

Abstract Syntax

Concrete syntax rewritten 1:1, EBNF sequences substituted by LIDO LISTOF:

```
RULE: Descriptions  LISTOF Import | Structure           END;
RULE: Import ::= 'import' ImportNames 'from' FileName    END;
RULE: ImportNames   LISTOF ImportName                    END;
RULE: Structure ::= StructureName '(' Fields ')'         END;
RULE: Fields        LISTOF Field                         END;
RULE: Field ::=     FieldName ':' TypeName ';'           END;
RULE: StructureName ::= Ident                            END;
RULE: ImportName ::=    Ident                            END;
RULE: FieldName ::=     Ident                            END;
RULE: TypeName ::=      Ident                            END;
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 807

Objectives:

Concrete syntax rewitten

In the lecture:

The items are explained.

Name Analysis

Described in GSS 5.8 to 5.11

Lecture Generating Software from Specifications WS 2013/14 / Slide 808

Objectives:

Already explained in Ch. 5

In the lecture:

The items are explained.

Property Analysis (1)

It is an error if the **name of a field**, say **addr**, of a structure occurs **as the type of a field** of that structure.

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

Introduce a PDL property

```
   IsField: int;
```

and check it:

```
SYMBOL Descriptions COMPUTE
   SYNT.GotIsField = CONSTITUENTS FieldName.GotIsField;
END;

SYMBOL FieldName COMPUTE
   SYNT.GotIsField = ResetIsField (THIS.Key, 1);
END;

SYMBOL TypeName COMPUTE
   IF (GetIsField (THIS.Key, 0),
      message (ERROR,
              CatStrInd ("Field identifier not allowed here: ",
                      THIS.Sym),
              0, COORDREF))
   <- INCLUDING Descriptions.GotIsField;
END;
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 809

Objectives:

A property introduced for checking

In the lecture:

The items are explained.

Property Analysis (2)

It is an error if the **same field** of a structure occurs **with different types specified**.

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

We introduce **predefined types int** and **float** as **keywords**. For that purpose we have to change both, concrete and abstract syntax correspondingly:

```
   RULE: Field ::=  FieldName ':' TypeName ';' END;
```

is replaced by

```
   RULE: Field ::= FieldName ':' Type ';' END;
   RULE: Type ::=  TypeName              END;
   RULE: Type ::=  'int'                 END;
   RULE: Type ::=  'float'                 END;
```

```
SYMBOL Type, FieldName: Type: DefTableKey;
RULE: Field ::= FieldName ':' Type ';' COMPUTE
   FieldName.Type = Type.Type;
END;
RULE: Type ::= TypeName COMPUTE
   Type.Type = TypeName.Key;
END;
RULE: Type ::= 'int' COMPUTE
   Type.Type = intType;
END;
... correspondingly for floatType
```

Type information is propagated to the **FieldName**

intType and **floatType** and **errType** are introduced as PDL known keys.

Lecture Generating Software from Specifications WS 2013/14 / Slide 810

Objectives:

A simple type analysis

In the lecture:

The items are explained:

- Predefined types: keywords are easier than identifiers!
- Late syntax modifications may occur.
- Use of known keys.

Property Analysis (3)

It is an error if the **same field** of a structure occurs **with different types specified**.

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

Request from PDL a property **Type** that has an operation `IsType (k, v, e)`.

```
Type: DefTableKey [Is]
```

It sets the **Type** property of key k to **v** if it is unset; it sets it to **e** if the property has a value different from **v**.

```
SYMBOL FieldName COMPUTE
   SYNT.GotType =
      IsType (THIS.Key, THIS.Type, ErrorType);

   IF (EQ (ErrorType, GetType (THIS.Key, NoKey)),
      message
      (ERROR, "different types specified for this field",
      0, COORDREF))
   <- INCLUDING Descriptions.GotType;
END;

SYMBOL Descriptions COMPUTE
   SYNT.GotType = CONSTITUENTS FieldName.GotType;
END;
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 811

Objectives:

PDL property functions are used

In the lecture:

The items are explained:

- There are more useful PDL property functions.
- Apply typical PDL usage pattern!

Structured Target Text

Methods and techniques are applied as described in Chapter 6.

For one structure there may be **several occurrences of structure descriptions** in the tree. At only one of them the complete class declaration for that structure is to be output. that is achived by using the `DoItOnce` technique (see GSS-4.5):

```
ATTR TypeDefCode: PTGNode;

SYMBOL Descriptions COMPUTE
   SYNT.TypeDefCode =
      CONSTITUENTS StructureName.TypeDefCode
      WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
END;

SYMBOL StructureName INHERITS DoItOnce COMPUTE
   SYNT.TypeDefCode =
      IF ( THIS.DoIt,
            PTGTypeDef (StringTable (THIS.Sym)), PTGNULL);
END;
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 812

Objectives:

Apply PTG techniques

In the lecture:

The items are explained:

- Recall the DoItOnce technique.
- Recall Chapter 6.