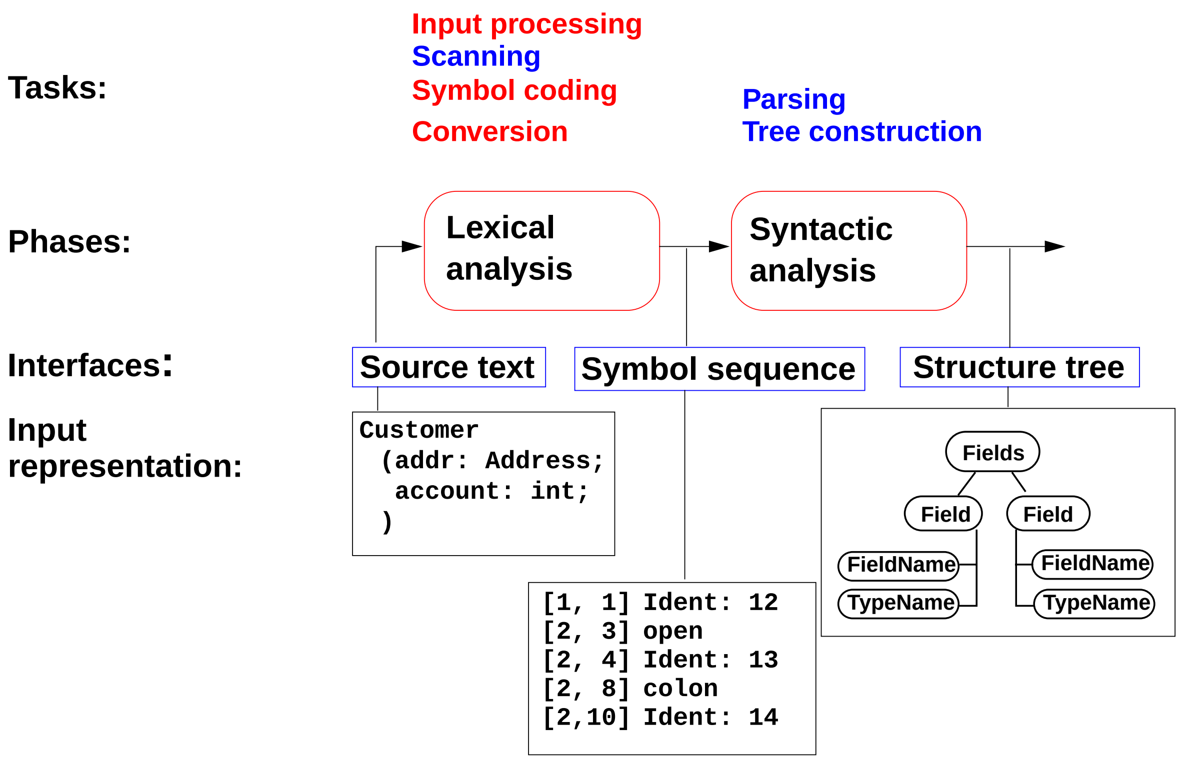
2. Constructing Trees - Overview

Check the notation and the structure of the input and represent it as a tree.



Lecture Generating Software from Specifications WS 2013/14 / Slide 201

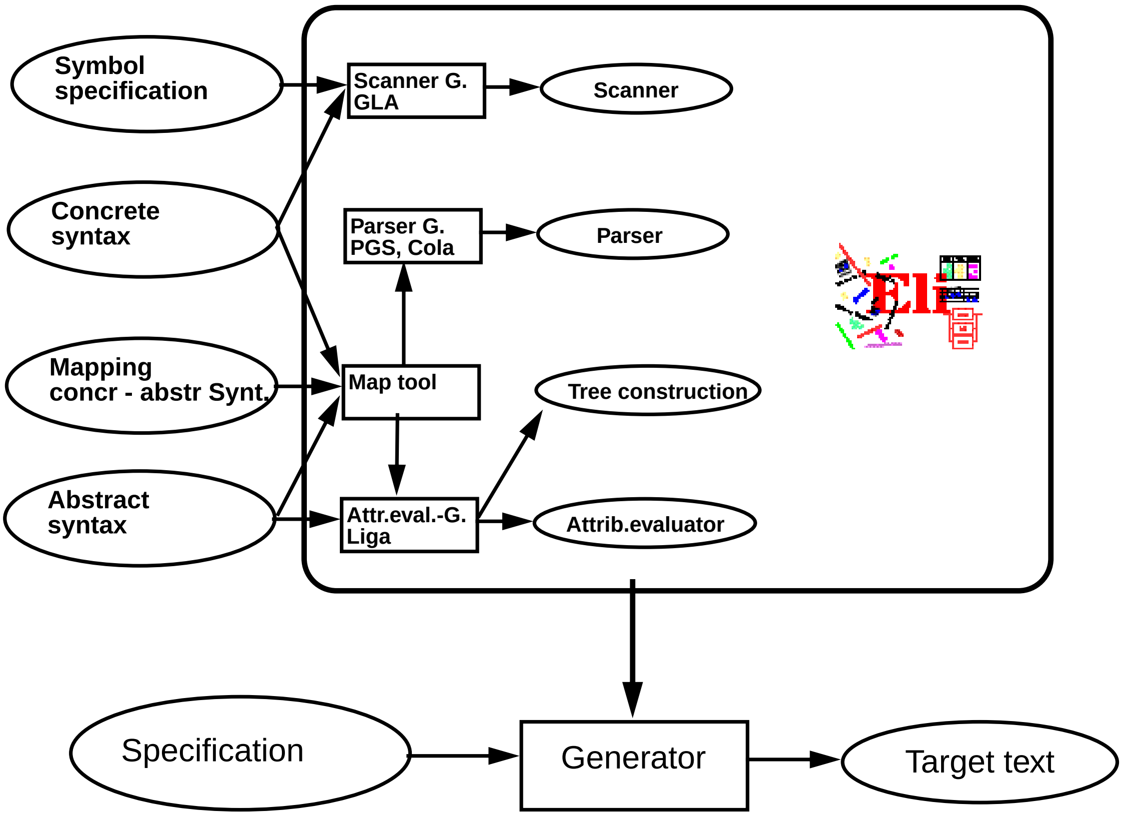
Objectives:

Understand the structuring phase

In the lecture:

- Remember the tasks of GSS-1.15.
- Explain the tasks and representations.

Eli: Specification of the Tree Construction



Lecture Generating Software from Specifications WS 2013/14 / Slide 202

Objectives:

Understand how the structuring phase is generated

In the lecture:

Explain

- Roles of the specifications,
- tasks of the generators,
- cooperation between the generators.

Specifications for the Structure Generator

Symbol specifications

Notations of non-literal tokens
.gla

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

Concrete syntax

Structure of input,
literal tokens
.con

```
Descriptions:(Import / Structure)*.
Structure:   StructureName '(' Fields ')'.
Fields:      Field*.
Field:       FieldName ':' TypeName.
...
```

Mapping concr - abstr Synt

.map

is empty if concret and abstract syntax coincide

Abstract syntax

Structure of trees
.lido

```
RULE: Descriptions LISTOF Import|Structure
COMPUTE ...

SYMBOL FieldName COMPUTE ...
SYMBOL TypeName COMPUTE ...
```

Only those symbols and productions, which need computations

Lecture Generating Software from Specifications WS 2013/14 / Slide 203

Objectives:

A simple example

In the lecture:

Get an idea of the specifications

Calendar Example: Structuring Task

A new example for the specification of the structuring task up to tree construction:

Input language: Sequence of calendar entries:

```
1.11.       20:00     "Theater"
Thu         14:15     "GSS lecture"
Weekday     12:05     "Dinner in Palmengarten"
Mon, Thu    8:00      "Dean's office"
31.12.      23:59     "Jahresende"
12/31       23:59     "End of year"
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 204

Objectives:

Introduce a new example

In the lecture:

Explain the task using the examples

Design of a Concrete Syntax

1. Develop a **set of examples**, such that all aspects of the intended language are covered.
2. Develop a **context-free grammar using a top-down strategy** (see PLaC-3.4aa), and
update the set of examples correspondingly.
3. Apply the **design rules** of PLaC-3.4c - 3.4f:
- Syntactic structure should **reflect semantic structure**
- **Syntactic restrictions** versus semantic conditions
- Eliminate **ambiguities**
- Avoid **unbounded lookahead**
4. Design notations of **non-literal tokens**.

Lecture Generating Software from Specifications WS 2013/14 / Slide 204a

Objectives:

Issues of grammar design

In the lecture:

- The strategy is explained.
- Repeat the methods learned in PLaC Sect. 3.2

Concrete Syntax

specifies the **structure of the input** by a context-free grammar:

```
Calendar:         Entry+ .
Entry:            Date Event.

Date:             DayNum '.' MonNum '.' /
                  MonNum '/' DayNum /
                  DayNames / GeneralPattern.

DayNum:           Integer.
MonNum:           Integer.

DayNames:         DayName /
                  DayNames ',' DayName.
DayName:          Day.

GeneralPattern:   SimplePattern /
                  SimplePattern Modifier.
SimplePattern:    'Weekday' / 'Weekend'.
Modifier:         '+' DayNames / '-' DayNames.

Event:            When Description / Description.

When:             Time / Time '-' Time.
```

Notation:

- Sequence of productions
- literal terminals between '
- EBNF constructs:
  / alternative
  ( ) parentheses
  [ ] option
  +, * repetition
  // repetition with separator

(for meaning see GPS)

Example:

```
1.11.        20:00        "Theater"
Thu          14:15        "GSS lecture"
Weekday      12:05        "Dinner in Palmengarten"
Mon, Thu     8:00         "Dean's office"
31.12.       23:59        "Jahresende"
12/31        23:59        "End of year"
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 205

Objectives:

Learn the CFG notation

In the lecture:

- Design of productions,
- notation of productions,
- relate to example input.

Literal and Non-Literal Terminals

Definition of notations of

- literal terminals (unnamed): **in the concrete syntax**
- non-literal terminals (named): in an additional **specification for the scanner generator**

```
Calendar:        Entry+ .
Entry:           Date Event.

Date:            DayNum '.' MonNum '.' /
                 MonNum '/' DayNum /
                 DayNames / GeneralPattern.

DayNum:          Integer.
MonNum:          Integer.

DayNames:        DayName /
                 DayNames ',' DayName.
DayName:         Day.

GeneralPattern:  SimplePattern /
                 SimplePattern Modifier.
SimplePattern:   'Weekday' / 'Weekend'.
Modifier:        '+' DayNames / '-' DayNames.

Event:           When Description / Description.

When:            Time / Time '-' Time.
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 206

Objectives:

Classification of terminals

In the lecture:

Notation of terminals specified in different ways

Specification of Non-Literal Terminals

The generator GLA generates a scanner from

- notations of literal terminals, extracted from the concrete syntax by Eli
- specifications of non-literal terminals in files of type **.gla**

Form of specifications:

```
Name:        $  regular expression                  [Coding function]
Day:         $  Mon|Tue|Wed|Thu|Fri|Sat|Son         [mkDay]
Time:        $(([0-9]|1[0-9]|2[0-3]):[0-5][0-9])    [mkTime]
```

Canned specifications:

```
Description:  C_STRING_LIT
Integer:      PASCAL_INTEGER
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 207

Objectives:

Understand scanner specifications

In the lecture:

Explain

- Notation of regular expressions,
- Task and interface of coding function,
- canned specifications.

Scanner Specification: Regular Expressions

<table>
<tr><th>Notation</th><th>accepted character sequences</th></tr>
<tr><td><code>c</code></td><td>the character c; except characters that have special meaning, see \c</td></tr>
<tr><td><code>\c</code></td><td>space, tab, newline, <code>\".[]^()|?+*{}/$<</code></td></tr>
<tr><td><code>"s"</code></td><td>the character sequence s</td></tr>
<tr><td><code>.</code></td><td>any single character except newline</td></tr>
<tr><td><code>[xyz]</code></td><td>exactly one character of the set {x, y, z}</td></tr>
<tr><td><code>[^xyz]</code></td><td>exactly one character that is not in the set {x, y, z}</td></tr>
<tr><td><code>[c-d]</code></td><td>exactly one character, the ASCII code of which lies between c and d (incl.)</td></tr>
<tr><td><code>(e)</code></td><td>character sequence as specified by e</td></tr>
<tr><td><code>ef</code></td><td>character sequences as specified by e followed by f</td></tr>
<tr><td><code>e | f</code></td><td>character sequence as specified by e or by f</td></tr>
<tr><td><code>e?</code></td><td>character sequence as specified by e or empty sequence</td></tr>
<tr><td><code>e+</code></td><td>one or more character sequences as specified by e</td></tr>
<tr><td><code>e*</code></td><td>character sequence as specified by e+ or empty</td></tr>
<tr><td><code>e {m,n}</code></td><td>at least m, and at most n character sequences as specified by e</td></tr>
</table>

e and f are regular expressions as defined here.

Each regular expression **accepts the longest character sequence**, that obeys its definition.

Solving ambiguities:

1. the **longer accepted sequence**
2. equal length: the **earlier stated rule**

Lecture Generating Software from Specifications WS 2013/14 / Slide 208

Objectives:

Notation of regular expressions

In the lecture:

Explain how to apply the definintions

Scanner Specification: Programmed Scanner

There are situations where the to be accepted character sequences are very difficult to define by a regular expression. A function may be implemented to accept such sequences.

The begin of the squence is specified by a regular expression, followed by the name of the function, that will accept the remainder. For example, line comments of Ada:

```
$-- (auxEOL)
```

Parameters of the function: a pointer to the first character of the so far accepted sequence, and its length.
Function result: a pointer to the charater immediately following the complete sequence:

```
char *Name(char *start, int length)
```

Some of the available programmed scanners:

`auxEOL`	all characters up to and including the next newline
`auxCString`	a C string literal after the opening "
`auxM3Comment`	a Modula 3 comment after the opening (*, up to and including the closing *); may contain nested comments paranthesized by (* and *)
`Ctext`	C compound statements after the opening {, up to the closing }; may contain nested statements parenthesized by { and }

Lecture Generating Software from Specifications WS 2013/14 / Slide 209

Objectives:

Recognize useful applications

In the lecture:

- Explain the principle and examples,
- refer to the list of available functions in the documentation.

Scanner Specification: Coding Functions

The **accepted character sequence** (**`start`**, **`length`**) is passed to a coding function.

It computes the code of the accepted token (**`intrinsic`**)
i.e. an **integral number, representing the identity of the token.**

For that purpose the function may **store and/or convert** the character sequence, if necessary.

All coding functions have the same **signature**:

```
void Name (char *start, int length, int *class, int *intrinsic)
```

The **token class** (terminal code, parameter **`class`**) may be changed by the function call, if necessary, e.g. to distinguish keywords from identifiers.

Available coding functions:

`mkidn`	enter character sequence into a hash table and encode it bijectively
`mkstr`	store character sequence, return a new code
`c_mkstr`	C string literal, converted into its value, stored, and given a new code
`mkint`	convert a sequences of digits into an integral value and return it value
`c_mkint`	convert a literal for an integral number in C and return its value

Lecture Generating Software from Specifications WS 2013/14 / Slide 210

Objectives:

Recognize the principle and useful applications

In the lecture:

- Explain the interface and examples
- refer to the list of available functions in the documentation

Scanner Specification: Canned Specifications

Complete canned specifications (regular expression, a programmed scanner, and a coding function) can be instantiated by their **names**:

```
Identifier:  C_IDENTIFIER
```

For many tokens of several programming languages canned specifications are available (complete list of descriptions in the documentation):

C_IDENTIFIER, C_INTEGER, C_INT_DENOTATION, C_FLOAT, C_STRING_LIT, C_CHAR_CONSTANT, C_COMMENT

PASCAL_IDENTIFIER, PASCAL_INTEGER, PASCAL_REAL, PASCAL_STRING, PASCAL_COMMENT

MODULA2_INTEGER, MODULA2_CHARINT, MODULA2_LITERALDQ, MODULA2_LITERALSQ, MODULA2_COMMENT

MODULA3_COMMENT, ADA_IDENTIFIER, ADA_COMMENT, AWK_COMMENT

SPACES, TAB, NEW_LINE
are only used, if some token begins with one of these characters, but, if these characters still separate tokens.

The used coding functions may be overridden.

Lecture Generating Software from Specifications WS 2013/14 / Slide 211

Objectives:

Recognize the potential for reuse

In the lecture:

- Explain some of the specifications,
- refer to the documentation

Abstract Syntax

specifies the **structure trees** using a context-free grammar:

```
RULE pCalendar:    Calendar LISTOF Entry              END;
RULE pEntry:       Entry ::= Date Event               END;
RULE pDateNum:     Date ::= DayNum MonNum             END;
RULE pDatePattern: Date ::= Pattern                   END;
RULE pDateDays:    Date ::= DayNames                  END;
RULE pDayNum:      DayNum ::= Integer                 END;
RULE pMonth:       MonNum ::= Integer                 END;
RULE pDayNames:    DayNames LISTOF DayName            END;
RULE pDay:         DayName ::= Day                    END;
RULE pWeekday:     Pattern ::= 'Weekday'              END;
RULE pWeekend:     Pattern ::= 'Weekend'              END;
RULE pModifier:    Pattern ::= Pattern Modifier       END;
RULE pPlus:        Modifier ::= '+' DayNames          END;
RULE pMinus:       Modifier ::= '-' DayNames          END;
RULE pTimedEvent:  Event ::= When Description         END;
RULE pUntimedEvent:Event ::= Description              END;
RULE pTime:        When ::= Time                      END;
RULE pTimeRange:   When ::= Time '-' Time             END;
```

Notation:
- Language ***Lido*** for computations in structure trees
- optionally named productions,
- no EBNF, except **LISTOF** (possibly empty sequence)

Lecture Generating Software from Specifications WS 2013/14 / Slide 212

Objectives:

Learn the notation for abstract syntax

In the lecture:

- Design of productions,
- notation of productions

Example for a Structure Tree

- Production names are node types
- Values of terminals at leaves

Tree output produced by Eli's unparser generator

```
pEntry( pDateNum(pDayNum(1),pMonth(11)),
        pTimedEvent(pTime(1200),"Theater")),

pEntry( pDateDays(pDay(4)),pTimedEvent(pTime(855),"GSS lecture")),

pEntry( pDatePattern(pWeekday()),
        pTimedEvent(pTime(725),"Dinner in Palmengarten")),

pEntry( pDateDays(pDay(1),pDay(4)),pUntimedEvent("Dean's office")),

pEntry( pDateNum(pDayNum(31),pMonth(12)),
        pTimedEvent(pTime(1439),"Jahresende")),

pEntry( pDateNum(pDayNum(31),pMonth(12)),
        pTimedEvent(pTime(1439),"End of year"))
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 213

Objectives:

Read tree in notation of named parenthesis

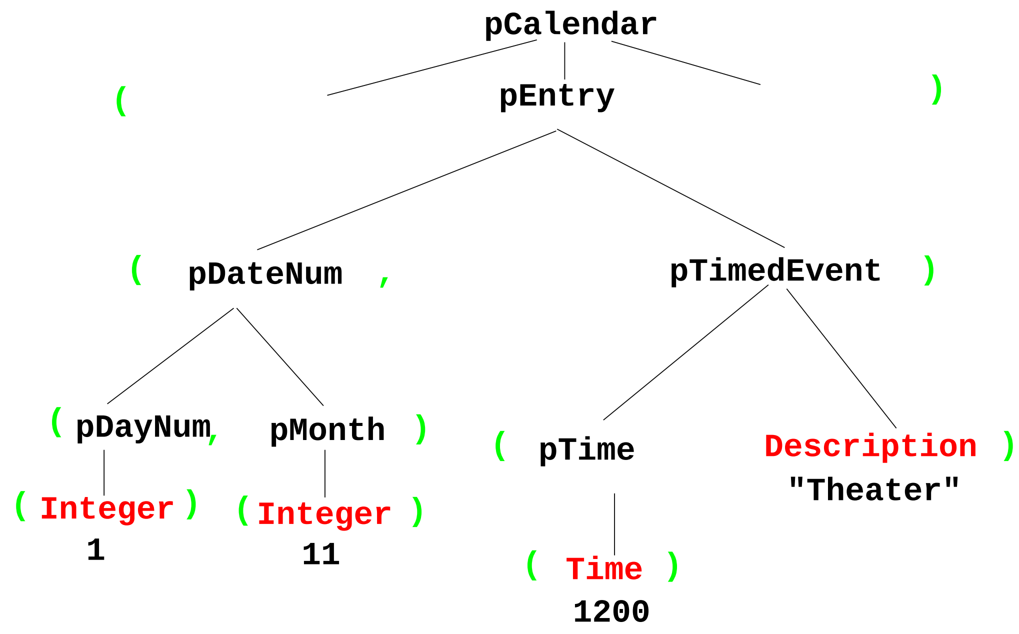
In the lecture:

- Relate to example input,
- relate to abstract syntax.

Graphic Structure Tree

- Names of productions as node types
- Values of terminals at leaves

Output produced by
Eli's unparser generator,
Tree structure given by parentheses



Lecture Generating Software from Specifications WS 2013/14 / Slide 214

Objectives:

Understand the tree representation

In the lecture:

Understand the relation between the abstract syntax (tree grammar) and the textual representation

Symbol Mapping: Concrete - Abstract Syntax

concrete syntax:

```
SimplePattern:  'Weekday' / 'Weekend'.

GeneralPattern: SimplePattern /
                SimplePattern Modifier.
```

simplify to create abstract syntax:

Set of nonterminals of the concrete syntax mapped to

one nonterminal of the abstract syntax

mapping:

```
MAPSYM
Pattern ::=  GeneralPattern
             SimplePattern.
```

abstract syntax:

```
RULE pWeekday:     Pattern ::= 'Weekday'           END;
RULE pWeekend:     Pattern ::= 'Weekend'           END;
RULE pModifier:    Pattern ::= Pattern Modifier    END;
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 215

Objectives:

Simplification of the structure tree

In the lecture:

- Explain symbol mapping,
- cf. symbol mapping for expression grammars in (GPS-2-9)

Rule Mapping

Concrete Syntax:

```
Date:        DayNum '.' MonNum '.' /
             MonNum '/' DayNum .
```

Mapping:

```
MAPRULE
Date: DayNum '.' MonNum '.'   < $1 $2 >.
Date: MonNum '/' DayNum       < $2 $1 >.
```

Different productions of the concrete syntax

are **unified** in the abstract syntax

Abstract syntax:

```
RULE pDateNum:       Date ::= DayNum MonNum END;
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 216

Objectives:

Tree simplification

In the lecture:

- Explain rule mapping,
- cf. simplification of expression grammars (GPS-2-9),
- abstract sytax can be genrated from concrete syntax and mapping specification,
- concrete syntax can be generated from abstract syntax and mapping specification,
- Abstract and concrete syntax can be matched, yielding the mapping specification.
- The grammars can be matched piecewise.

Generate Tree Output

Produce structure trees with node types and values at terminal leaves:

```
pEntry( pDateNum(pDayNum(1),pMonth(11)),
        pTimedEvent(pTime(1200),"Theater")),
```

Pattern constructor functions are called in tree contexts to produce output.

Specifications are **created automatically** by Eli's **unparser generator**:

Unparser is generated from the specification:

```
Calendar.fw
Calendar.fw:tree
```

Output at grammar root:

```
SYMBOL ROOTCLASS COMPUTE
   BP_Out(THIS.IdemPtg);
END;
```

Output of non-literal terminals:

```
Idem_Day:     $ int
Idem_Time:    $ int
Idem_Integer: $ int
```

Use predefined PTG patterns:

```
$/Output/PtgCommon.fw
```


Lecture Generating Software from Specifications WS 2013/14 / Slide 217

Objectives:

Learn to use the unparser generator

In the lecture:

Explain the roles of the specification

- Unparser generator generates Eli specifications (ptg and lido)!
- Individual specifications needed for the root and the leaves only.
- Another variant of the unparser generator can reproduce the input text: instead of ":tree" derive ":idem". It may be used for language extensions.