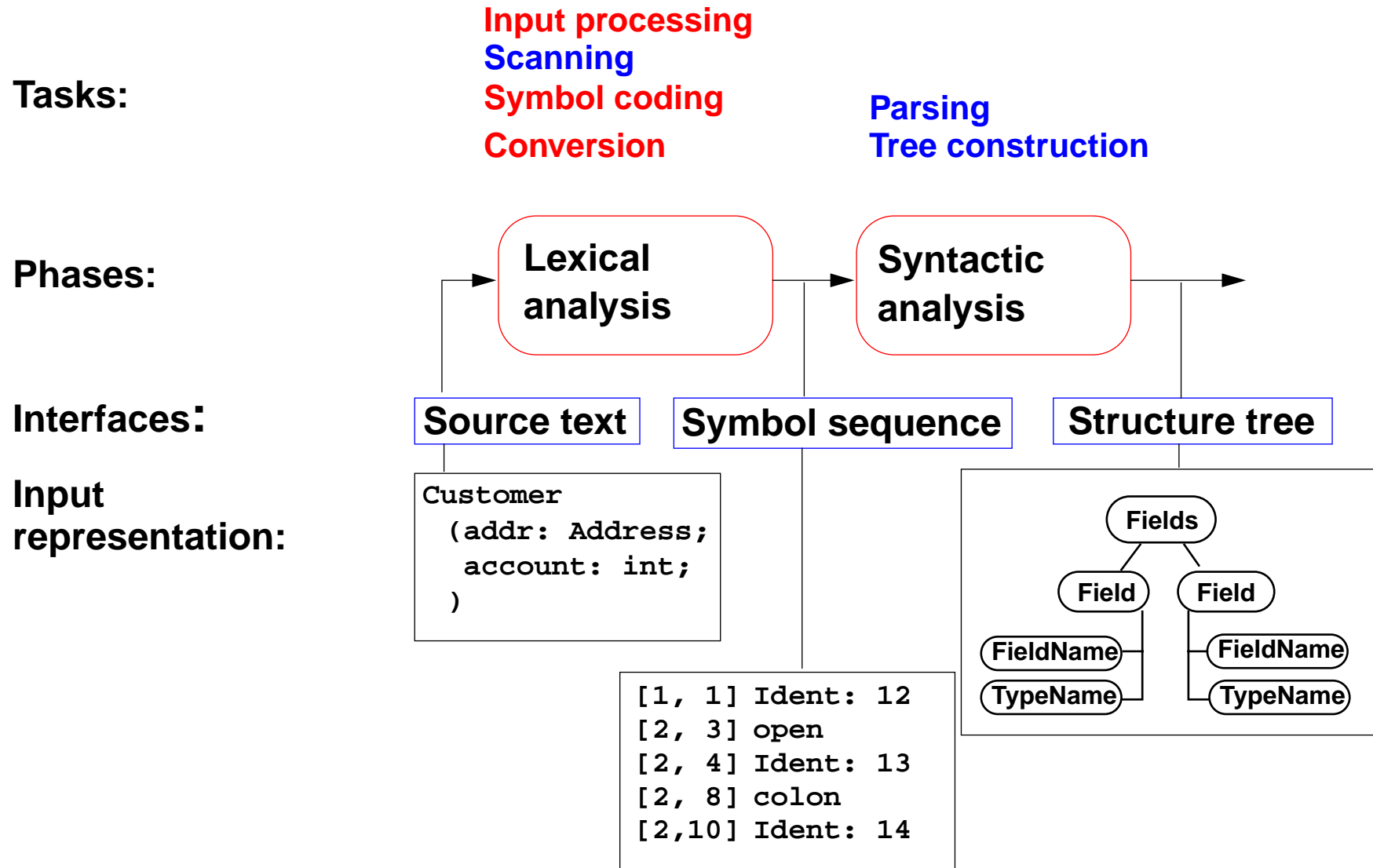
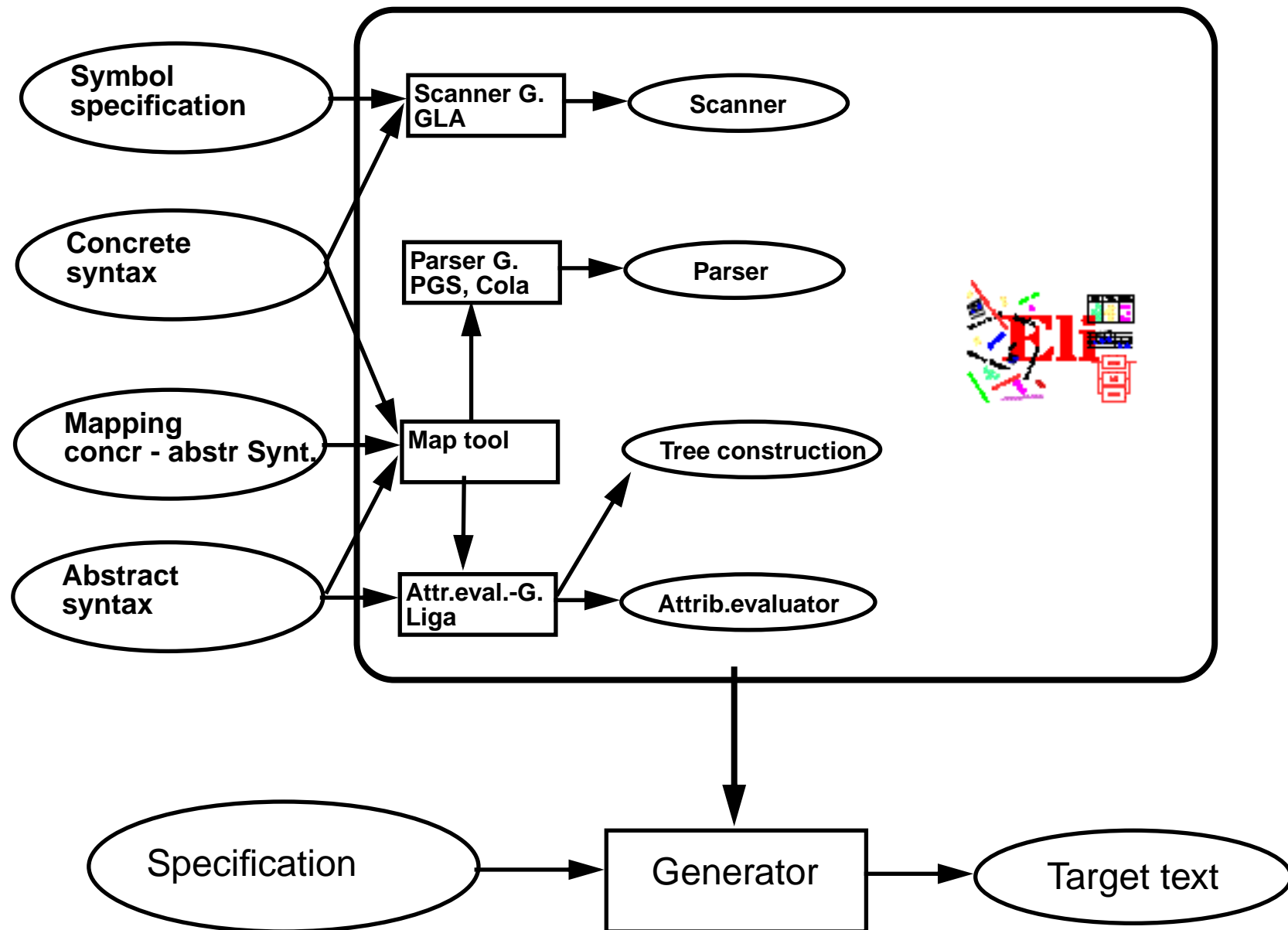


## 2. Constructing Trees - Overview

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



# Eli: Specification of the Tree Construction



# 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*

# 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"

# 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**.

# 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"

# 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.
  
```

# 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
  
```



# Scanner Specification: Regular Expressions

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

`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**

# 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:

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

# 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

# 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.

# 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)

## Example for a Structure Tree

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

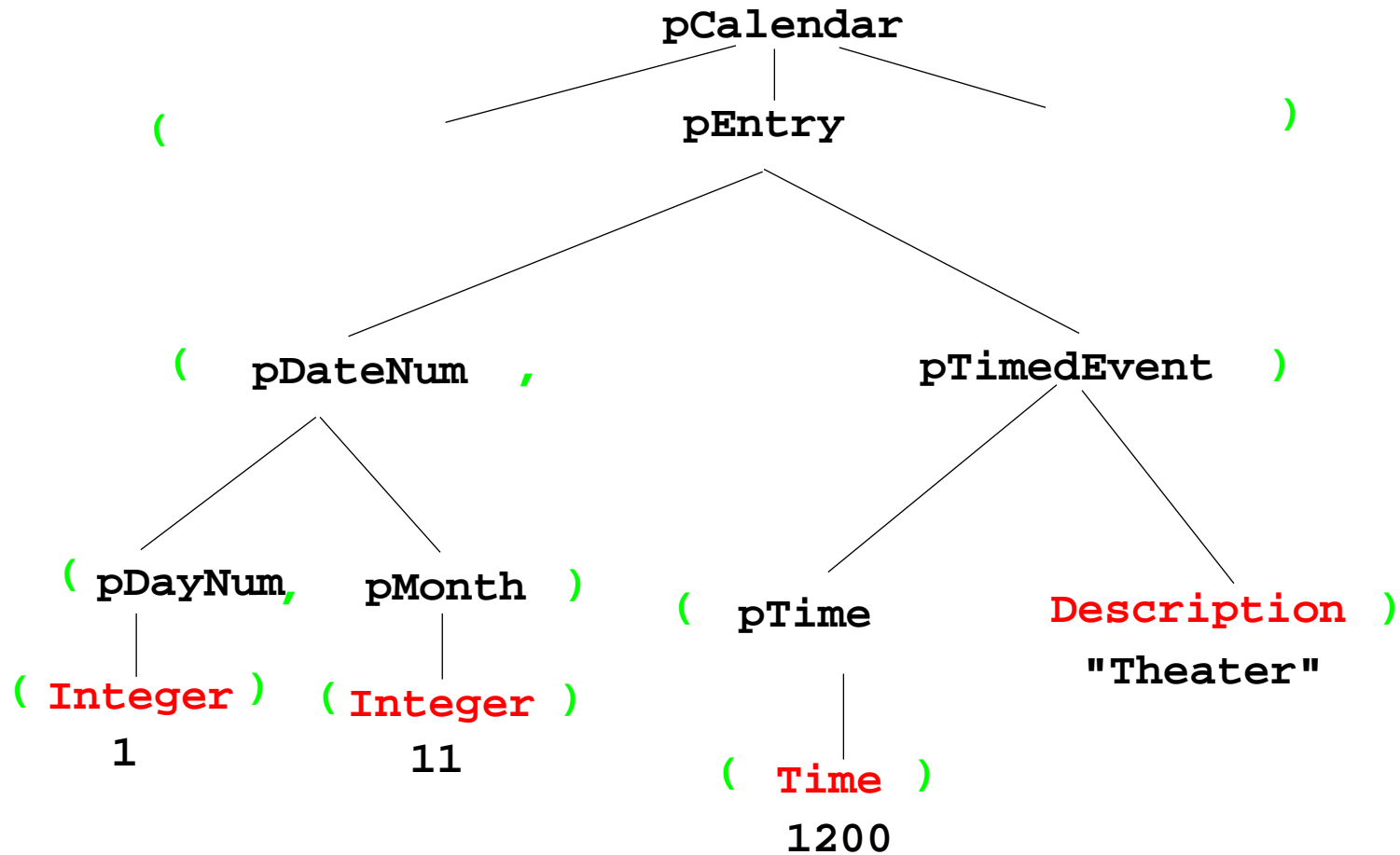
Tree output produced by Eli's  
unparser generator

```
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"))
```

# 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**



# 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;
```



# Rule Mapping

Concrete Syntax:

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

Mapping:

MAPRULE

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

Abstract syntax:

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

**Different productions** of the concrete syntax

are **unified** in the abstract syntax

# 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
```