Generating Software from Specifications

Prof. Dr. Uwe Kastens

WS 2013 / 14

Objectives

The participants will learn

- to **use generators** for specific software tasks,
- to **design domain specific languages (DSLs)**,
- to **implement domain specific languages (DSLs)**,
- to **use the Eli system** to create generators.

The participants will **define their own application project and implement it.**

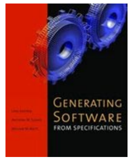
Contents

Phase 1: Lectures, practical tutorials, and individual work are tightly interleaved

Phase 2: Participants work in groups on their projects.
During lecture hours advice is given, problems are discussed, and experience are exchanged.

References

- U. Kastens: **Generating Software from Specifications Elektronic Script, SS 2012**
  http://ag-kastens.upb.de/lehre/material/gss
- Uwe Kastens, Anthony M. Sloane, William M. Waite:
  Generating Software from Specifications,
  Jones and Bartlett Publishers, 2007

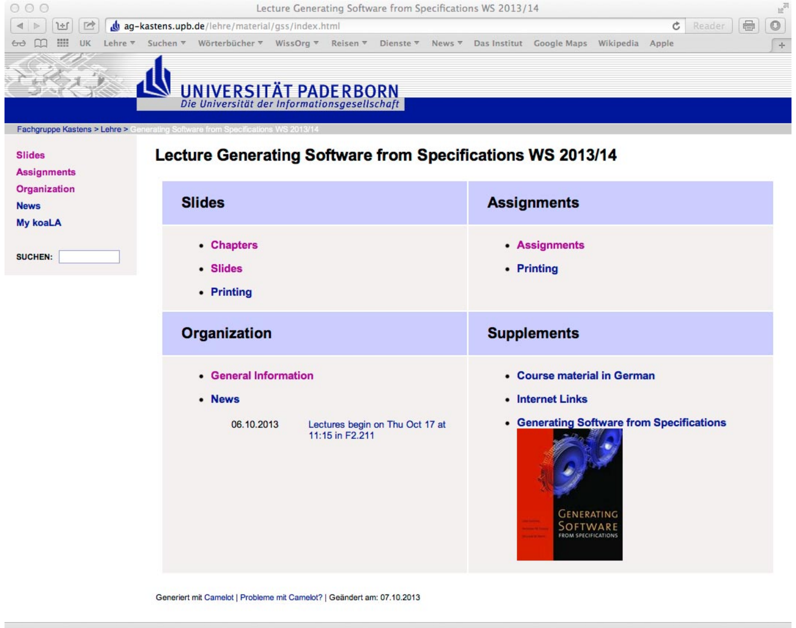


- **Eli Online Documentation and Download**
  http://eli-project.sourceforge.net (download)
- **DEViL - Development Environment for Visual Languages**
  http://devil.cs.upb.de

Papers on DSL and Reuse:

- Mernik, Heering, Sloane: When and How to Develop Domain-Specific Languages, ACM Computing Surveys, Vol. 37, No. 4, December 2005, pp. 316-344
- Ch. W. Kruger: Software Reuse, ACM Computing Surveys, 24(2), 1992
- R. Prieto-Diaz: Status Report: Software reusability, IEEE Software, 10(3), 1993

Home Page of GSS Lecture



Organization

Personen

Sprechstunde Uwe Kastens:

- `Mi  16:00 - 17:00 Uhr`
- `Die 11:00 - 12:00 Uhr`

Übungsbetreuer:

- Peter Pfahler

Termine

Vorlesung

- `Di, 9:15 - 10:45 Uhr F0.530`

Beginn: Di, 15. Oktober 2013 um 9:15 Uhr

Übungen

Die Übungen werden im 14-tägigen Abstand 2-stündig angeboten. Das Vorlesungsverzeichnis sieht 4 Übungsgruppen vor:

- **G1**: Dienstag 11:00 Uhr, *ungerade Wochen*, Beginn 22.10.2013, erst in F0.530, dann im Rechner-Pool F1 (hinterer Teil)
- **G2**: Dienstag 11:00 Uhr, *gerade Wochen*, Beginn 15.10.2013, erst in F0.530, dann im Rechner-Pool F1 (hinterer Teil)
- **G3**: Donnerstag 09:15 Uhr, *ungerade Wochen*, Beginn 24.10.2013, erst in F2.211, dann im Rechner-Pool F1 (hinterer Teil)
- **G4**: Freitag 09:15 Uhr, *gerade Wochen*, Beginn 18.10.2013, erst in F2.211, dann im Rechner-Pool F1 (hinterer Teil)

Prüfungstermine

Mündliche Prüfungen von ca 30 min Dauer im Rahmen von Modulprüfungen; für Studierende anderer Studiengänge als Informatik auch Einzelprüfungen.
Es werden zwei Prüfungszeiträume angeboten:

1. 12.-14. Februar 2014
2. 01.-03. April 2014

Zu Anmeldung in PAUL und Terminvergabe siehe http://www.cs.uni-paderborn.de/studierende/pruefungswesen/pruefungsanmeldung.html

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:

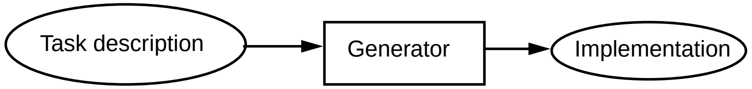
```
colors{red blue green}
bugs{ant spider fly moth bee}
verbs{crawl walk run fly}
```

- 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]

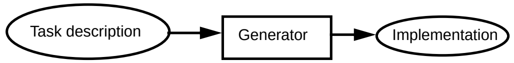
narrow, specific application domain	completely understood Implementation automatically generated
Abstractions on a high level (using domain knowledge)	transformed into executable software
User understands **abstractions** of the application domain	**Generator expert** understands **implementation methods**

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*

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

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

Reuse of Products

Product	**What is reused?**
Library of functions	Implementation
Module, component	Code
generic module	Planned variants of code
Software architecture	Design
Framework	Design and code
Design pattern	Strategy for design and construction
Generator	Knowledge, how to construct implementations from descriptions
Construction process	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**
ad hoc	• Code is copied and modified • adaptation of OO classes incrementally in sub-classes	• no a priori costs • very dangerous for maintanance
planned	• oo libraries, frameworks • Specialization of classes	• high a priori costs • effective reuse
automatic	• Generators, intelligent development environments	• high a priori costs • very effective reuse • wide cognitive distance

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:**

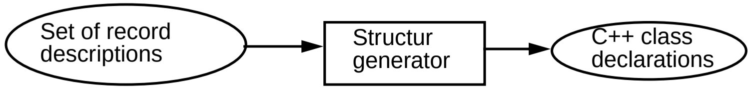
- The **same person**, group of persons, profession
- Provider assumes his own level of understanding for the reuser
- Examples: reuse of code, design patterns

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_Cl *Customer;
typedef class Address_Cl *Address;

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


Task Decomposition for the Implementation of Domain-Specific Languages

Structuring	**Lexical analysis**	**Scanning** **Conversion**
	Syntactic analysis	**Parsing** **Tree construction**
Translation	**Semantic analysis**	**Name analysis** **Property analysis**
	Transformation	**Data mapping** **Action mapping**

[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

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

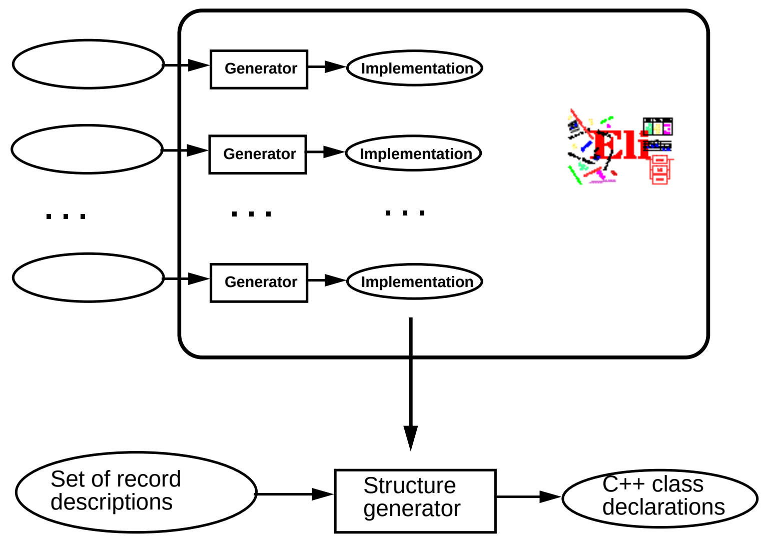
```
Customer ( addr:      Address;
           account:   int; )

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

import String from "util.h"
```

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

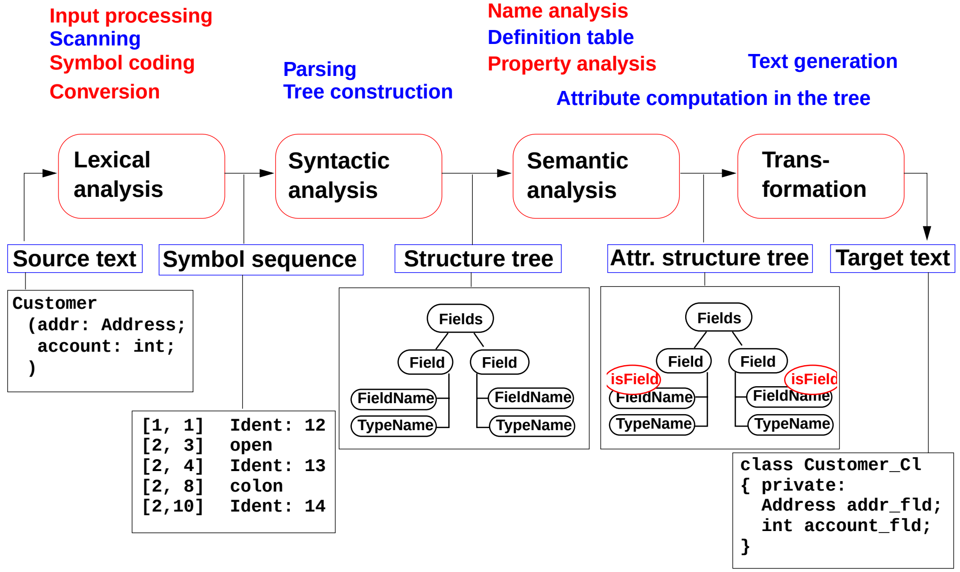
```
Customer ( addr:     Address;
           account:  int; )

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

import String from "util.h"
```


Eli Generates a Structure Generator

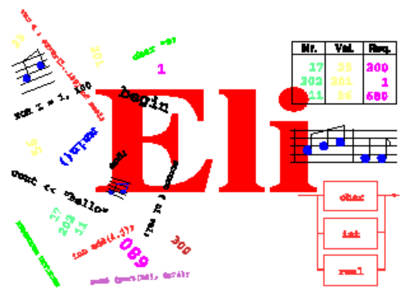


Task Decomposition Determines the Architecture of the Generator

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



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 since1989 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

2. Constructing Trees - Overview

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

Tasks:
Input processing
Scanning
Symbol coding
Conversion
Parsing
Tree construction

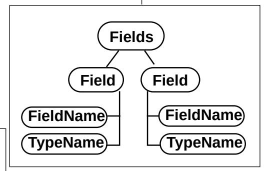
Phases: Lexical analysis → Syntactic analysis

Interfaces: Source text | Symbol sequence | Structure tree

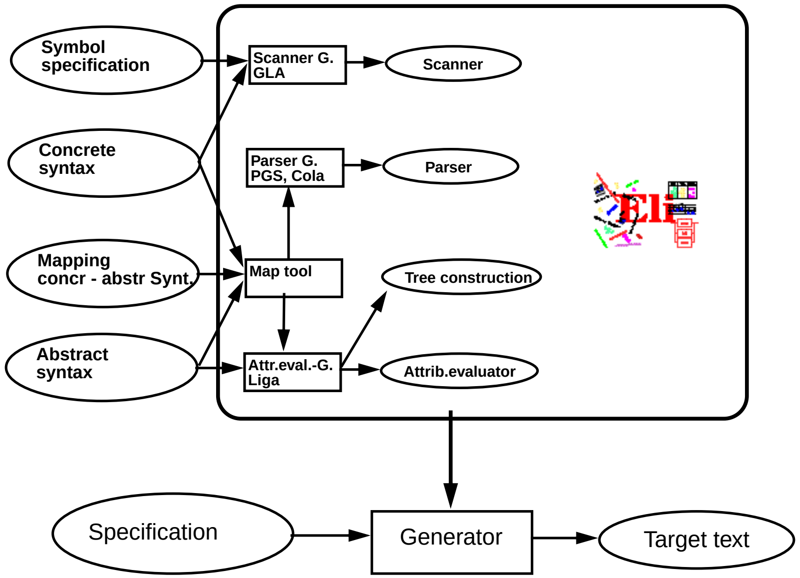
Input representation:

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

```
[1, 1] Ident: 12
[2, 3] open
[2, 4] Ident: 13
[2, 8] colon
[2,10] Ident: 14
```



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

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:

`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 }

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

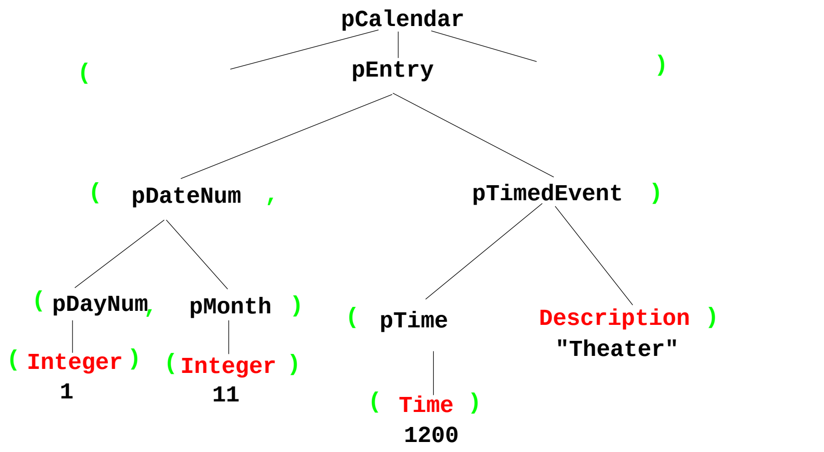
Tree output produced by Eli's unparser generator

```
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 .
```

Different productions of the concrete syntax

are **unified** in the abstract syntax

Mapping:

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

Abstract syntax:

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


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
```


3. Visiting Trees
Overview

Computations in structure trees may serve any suitable purpose, e.g.

- **compute or check properties of language constructs**, e. g. types, values
- **determine or check relations in larger contexts,** e.g. definition - use
- **construct data structure or target text**

Formal model for specification: attribute grammars (AGs)

Generator Liga transforms

a specification of computations in the structure tree
(an AG written in the specification language Lido)

into

a tree walking attribute evaluator that executes the specified computations for each given tree in a suitable order.

Computations in Tree Contexts Specified by AGs

Abstract syntax is augmented by:

Attributes associated to **nonterminals**:
e.g. Expr.Value   Expr.Type   Block.depth used to

store values at tree nodes, representing a property of the construct,
propagate values through the tree,
specify dependences between computations

Computations associated to **productions** (RULEs) or to nonterminals (SYMBOL):

Compute attribute values
using other attribute values of the particular context (RULE or SYMBOL), or

cause effects, e.g. store values in a definition table,
check a condition and issue a message, produce output

Each **attribute** of every node is **computed exactly once**.
Each **computation** is **executed exactly once** for every node of the RULE it is specified for.

The **order of the computation execution** is **determined by the generator**. It obeys the **specified dependences**.

Dependent Computations

```
SYMBOL Expr, Opr: value: int SYNT;
SYMBOL Opr: left, right: int INH;
TERM Number: int;

RULE: Root ::= Expr COMPUTE
   printf ("value is %d\n", Expr.value);
END;

RULE: Expr ::= Number COMPUTE
   Expr.value = Number;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
   Expr[1].value = Opr.value;
   Opr.left = Expr[2].value;
   Opr.right = Expr[3].value;
END;

RULE: Opr ::= '+' COMPUTE
   Opr.value = ADD (Opr.left, Opr.right);
END;
RULE: Opr ::= '-' COMPUTE
   Opr.value = SUB (Opr.left, Opr.right);
END;
```

typed attributes of symbols

terminal symbol has int value

SYNThesized attributes are computed in lower contexts, INHerited attributes in upper c..

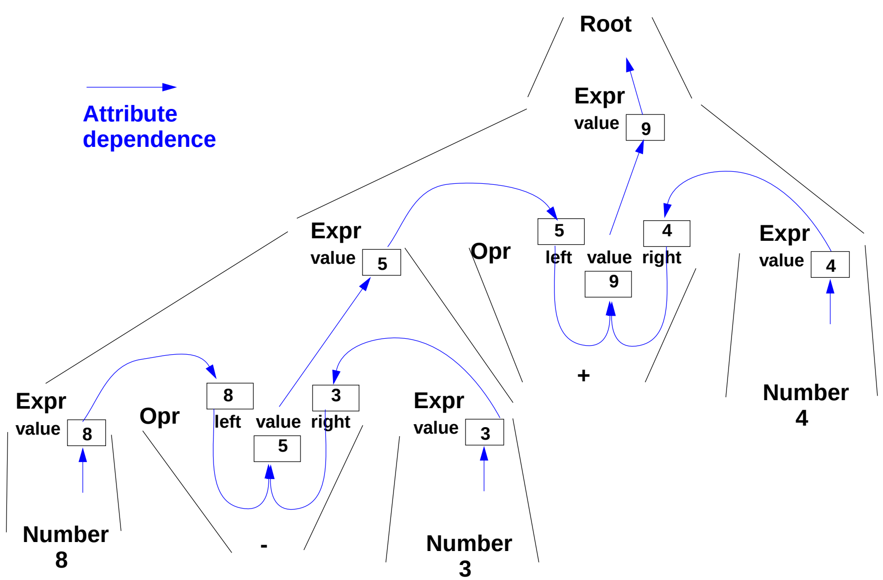
SYNT or INH usually need not be specified.

Generator determines the **order of computations** consistent with dependences.

Example:

Computation and output of an expression's value

An Attributed Structure Tree



Pre- and Postconditions of Computations

```
RULE: Root ::= Expr COMPUTE
   Expr.print = "yes";
   printf ("n") <- Expr.printed;
END;

RULE: Expr ::= Number COMPUTE
   Expr.printed =
      printf ("%d ", Number) <-Expr.print;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
   Expr[2].print = Expr[1].print;
   Expr[3].print = Expr[2].printed;
   Opr.print = Expr[3].printed;
   Expr[1].printed = Opr.printed;
END;

RULE: Opr ::= '+' COMPUTE
   Opr.printed =
      printf ("+ ") <- Opr.print;
END;
```

Attributes `print` and `printed` **don't have values** (type **`VOID`**)

They describe states being **pre- and postconditions** of computations

`Expr.print`:

Postfix output up to this node is completed.

`Expr.printed`:

Postfix output up to and including this node is completed.

Example:

Expression is printed in postfix form

Pattern: Dependences Left-to-Right Depth-First Through the Tree

```
CHAIN print: VOID;

RULE: Root ::= Expr COMPUTE
   CHAINSTART HEAD.print = "yes";
   printf ("n") <- TAIL.print;
END;

RULE: Expr ::= Number COMPUTE
   Expr.print =
      printf ("%d ", Number) <-Expr.print;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
   Expr[3].print = Expr[2].print;
   Opr.print = Expr[3].print;
   Expr[1].print = Opr.print;
END;

RULE: Opr ::= '+' COMPUTE
   Opr.print =
      printf ("+ ") <- Opr.print;
END;
```

CHAIN specifies **left-to-right depth-first** dependence.

CHAINSTART in the **root context** of the CHAIN (initialized with an irrelevant value)

Computations are inserted between **pre- and postconditions of the CHAIN**

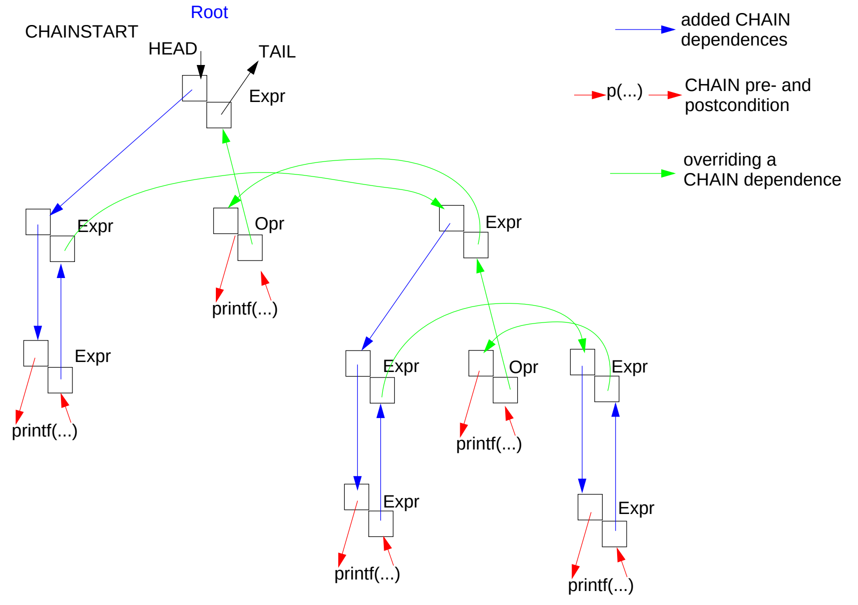
CHAIN order can be overridden.

Omitted CHAIN computations are added **automatically**

Example:

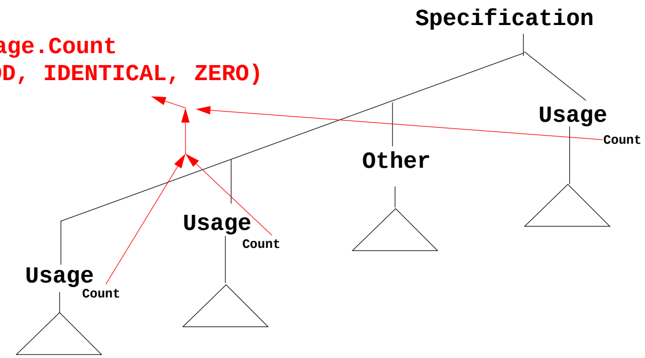
Output an expression in postfix form (cf. GSS-3.4)

Pattern: Dependences Left-to-Right Depth-First Through the Tree



Pattern: Combine Attribute Values of a Subtree

```
CONSTITUENTS Usage.Count
   WITH (int, ADD, IDENTICAL, ZERO)
```



CONSTITUENTS combines certain attributes of a subtree, here **Usage.Count**

	WITH (int,	ADD,	IDENTICAL,	ZERO)
Meaning:	type	binary function	unary function, applied to every attribute	constant function for optional subtrees

Pattern: Use an Attribute of a Remote Ancestor Node

```
SYMBOL Block: depth: int INH;

RULE: Root ::= Block COMPUTE
   Block.depth = 0;
END;

RULE: Block ::= '(' Sequence ')' END;
RULE: Sequence LISTOF
      Definition / Statement END;
...

RULE: Statement ::= Block COMPUTE
   Block.depth =
      ADD (INCLUDING Block.depth, 1);
END;

TERM Ident: int;

RULE: Definition ::= 'define' Ident
COMPUTE
   printf("%s defined on depth %d\n",
      StringTable (Ident),
      INCLUDING Block.depth);
END;
```

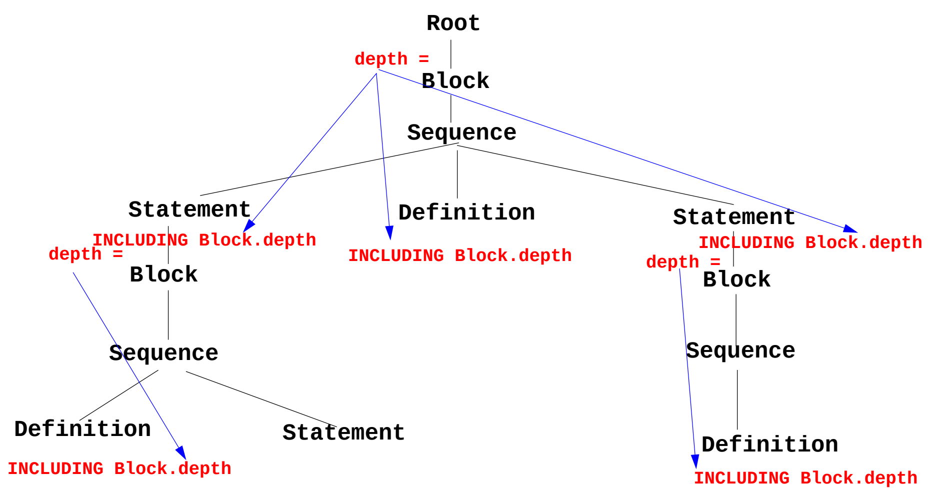
Example:

Compute nesting depth of blocks

INCLUDING Block.depth refers to the **depth** attribute of the next ancestor node (towards the root) that has type **Block**

The **INCLUDING** attribute is **automatically propagated** through the contexts between its definition in an ancestor node and its use in an **INCLUDING** construct.

Example for INCLUDING in a Tree



Pattern: Combine Preconditions of Subtree Nodes

```
SYMBOL Block: DefDone: VOID;

RULE: Root ::= Block END;

RULE: Block ::= '(' Sequence ')'
COMPUTE
   Block.DefDone =
      CONSTITUENTS Definition.DefDone;
END;

...

RULE: Definition ::= 'define' Ident
COMPUTE
   Definition.DefDone =
   printf("%s defined in line %d\n",
      StringTable (Ident), LINE);
END;

RULE: Statement ::= 'use' Ident
COMPUTE
   printf("%s used in line %d\n",
      StringTable (Ident), LINE)
      <- INCLUDING Block.DefDone;
END;
```

Example:

Output all definitions before all uses

The attributes `DefDone` do not have values - they specify **preconditions** for some computations

This `CONSTITUENTS` construct does not need a **`WITH` clause**, because it does not propagate values

Typical combination of a `CONSTITUENTS` construct and an `INCLUDING` construct:

Specify the order side-effects are to occur in.

Computations Associated to Symbols

Computations may be associated to **symbols**; then they are executed for **every occurrence** of the symbol in a production.

```
SYMBOL Expr COMPUTE
   printf ("expression value %d in line %d\n", THIS.value, LINE);
END;
```

Symbol computations may contain **`INCLUDING`**, **`CONSTITUENTS`**, and **`CHAIN`** constructs:

```
SYMBOL Block COMPUTE
   printf ("%d uses occurred\n",
      CONSTITUENTS Usage.Count WITH (int, ADD, IDENTICAL, ZERO);
END;
```

`SYNT.a` resp. **`INH.a`** indicates that the computation belongs to the **lower** resp. **upper context** of the symbol:

```
SYMBOL Block COMPUTE
   INH.depth = ADD (INCLUDING Block.depth);
END;
```

Computations in **RULE contexts override computations** for the same attribute **in SYMBOL context**, e.g. for begin of recursions, defaults, or exceptions:

```
RULE: Root ::= Block COMPUTE
   Block.depth = 0;
END;
```


Reuse of Computations

```
CLASS SYMBOL IdOcc: Sym: int;
CLASS SYMBOL IdOcc COMPUTE
   SYNT.Sym = TERM;
END;

SYMBOL DefVarIdent INHERITS IdOcc END;
SYMBOL DefTypeIdent INHERITS IdOcc END;
SYMBOL UseVarIdent INHERITS IdOcc END;
SYMBOL UseTypeIdent INHERITS IdOcc END;


CLASS SYMBOL CheckDefined COMPUTE
   IF (EQ (THIS.Key, NoKey),
   message ( ERROR,
             "identifier is not defined",
             0, COORDREF);
END;

SYMBOL UseVarIdent
   INHERITS IdOcc, CheckDefined END;
SYMBOL UseTypeIdent
   INHERITS IdOcc, CheckDefinedEND;
```

Computations are associated to **`CLASS`** symbols, which do not occur in the abstract syntax.

`INHERITS` binds **`CLASS`** symbols to tree symbols of the abstract syntax.

Reuse of Pairs of SYMBOL Roles

```
CLASS SYMBOL OccRoot COMPUTE
   CHAINSTART HEAD.Occurs = 0;
   SYNT.TotalOccs = TAIL.Occurs;
END;
CLASS SYMBOL OccElem COMPUTE
   SYNT.OccNo = THIS.Occurs;
   THIS.Occurs = ADD (SYNT.OccNo, 1);
END;
```

`CLASS` **symbols in cooperating roles**, e.g. count occurrences of a language construct (`OccElem`) in a subtree (`OccRoot`)

Restriction:
Every `OccElem`-node must be in an `OccRoot`-subtree.

```
SYMBOL Block      INHERITS OccRoot END;
SYMBOL Definition INHERITS OccElem END;

SYMBOL Statement INHERITS OccRoot END;
SYMBOL Usage     INHERITS OccElem END;
```

Reused in pairs:

`Block - Definition` and

`Statement - Usage`

must obey the restriction.

Library modules are used in this way (see Ch. 6)

Design Rules for Computations in Trees

1. Decompose the task into **subtasks**, that are small enough to be solved each by only a few of the specification patterns explained below.d
   Develop a `.lido` fragment for each subtask and explain it in the surrounding `.fw` text.
2. Elaborate the **central aspect of the subtask** and map it onto one of the following cases:
   A. The aspect is described in a natural way by **properties of some related program constructs**,
   e.g. types of expressions, nesting depth of blocks, translation of the statements of a block.
   B. The aspect is described in a natural way by **properties of some program entities,**
   e.g. relative addresses of variabes, use of variables before their definition.

   Develop the computations as described for A or B.
3. Step 2 may exhibit that further aspects of the subtask need to be solved (attributes may be used, for which the computations are not yet designed). Repeat step 2 for these aspects.

A: Compute Properties of Program Constructs

Determine the **type of values**, which describe the property. Introduce **attributes of that type for all symbols**, which represent the **program constructs**. Check which of the following cases fits best for the computation of that property:

A1: Each **lower context** determines the property in a different way:
Then develop **RULE computations for all lower contexts**.

A2: As A1; but **upper context**.

A3: The property can be determined **independently of RULE contexts**, by using only attributes of the symbol or attributes that are accessed via INCLUDING, CONSTITUENT(S), CHAIN:
Then develop a **lower (SYNT) SYMBOL computation**.

A4: As A3; but there are a **few exceptions**, where either lower of upper (not both) RULE contexts determine the property in a different way:
Then develop a upper (INH) or a lower (SYNT) **SYMBOL computation** and **override it in the deviating RULE contexts**.

A5: As A4; but for **recursive symbols**: The begin of the recursion is considered to be the exception of A4, e.g. nesting depth of Blocks.

If none of the cases fits, the design of the property is to be reconsiderd; it may be too complex, and may need further refinement.

4. Names, Entities, and Properties

Program constructs in the tree (e.g. definitions) may

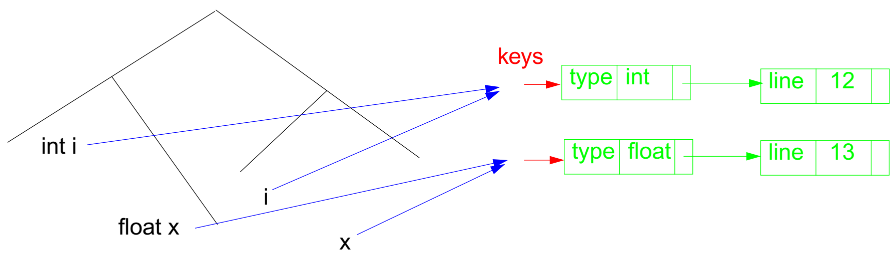
- introduce an **entity** (e.g. a variable, a class, or a function)
- **bind the entity to a name**
- associate **properties** **to the entity** (e.g. type, kind, address, line)

The **definition module** stores **program entities with their properties**, e.g. a variable with its type and the line number where it is defined.

Entities are identified by keys of the definition module.

Name analysis binds names to entities.

The **properties** of an entity are represented by a list of **(kind, value)-pairs**



Basic name analysis provided by symbol roles

Symbol roles:

Grammar root:

`SYMBOL Program INHERITS RootScope END;`

Ranges containing definitions:

`SYMBOL Block INHERITS RangeScope END;`

Defining identifier occurrence:

`SYMBOL DefIdent INHERITS IdDefScope END;`

Applied identifier occurrence:

`SYMBOL UseIdent INHERITS IdUseEnv, ChkIdUse END;`

Required attributes:

```
CLASS SYMBOL IdentOcc: Sym: int;
CLASS SYMBOL IdentOcc COMPUTE SYNT.Sym = TERM; END;

SYMBOL DefIdent INHERITS IdentOcc END;
SYMBOL UseIdent INHERITS IdentOcc END;
```

Provided attributes:

```
SYMBOL DefIdent, UseIdent: Key: DefTableKey, Bind: Binding;
SYMBOL Program, Block: Env: Environment;
```

Instantiation in a `.specs` file for Algol-like scope rules:

`$/Name/AlgScope.gnrc:inst`

for C-like scope rules:

`$/Name/CScope.gnrc: inst`

PDL: A Generator for Definition Modules

central data structure associates **properties to entities**,
e.g. *type of a variable, element type of an array type.*

Entities are identified by a **key** (type **`DefTableKey`**).

Operations:

`NewKey ( )`	yields a new key
`ResetP (k, v)`	for key **k** the property **P** is set to the value **v**
`SetP (k, v, d)`	for key **k** the property **P** is set to the value **v**, if it was not set, otherwise to the value **d**
`GetP (k, d)`	for key **k** it yields the value of the property **P** if it is set, otherwise it yields **d**

Functions are called in **computations in tree contexts**.

PDL generates functions **`ResetP, SetP, GetP`** from specifications of the form

PropertyName: ValueType;

e.g.

```
Line: int;
Type: DefTableKey;
```


Example: Set and Get a Property

The line number is associated as a property in a **`.pdl`** file:

`Line: int;`

It is set in definition contexts and got in use contexts.

All set computations in **definition** contexts have to precede any get in **use** contexts.

```
SYMBOL Program INHERITS RootScope END;
RULE: Program LISTOF Definition | Use COMPUTE
   Program.GotLine = CONSTITUENTS Definition.GotLine;
END;

RULE: Definition ::= 'def' NameDef END;
RULE: Use ::= 'use' NameUse END;

SYMBOL NameDef INHERITS IdentOcc, IdDefScope COMPUTE
   SYNT.GotLine = ResetLine (THIS.Key, LINE);
   printf ("%s defined in line %d\n", StringTable(THIS.Sym), LINE);
END;

SYMBOL NameUse INHERITS IdentOcc, IdUseEnv, ChkIdUse COMPUTE
   printf ("%s defined in line %d used in line %d\n",
           StringTable(THIS.Sym), GetLine (THIS.Key, 0), LINE)
   <- INCLUDING Program.GotLine;
END;
```


Design Rules for Property Access (B)

Preparation:

- Usually identifiers in the tree refer to entities represented by **`DefTableKeys`**; an identifier is bound to a key using the **name analysis module** (see Ch.5).
- Symbol nodes for identifiers have a **`Key`** attribute; it identifies the entity

Design steps for the computation of properties:

1. Specify **name and type of the property** in the notation of PDL.
2. Identify the **contexts where the property is set**.
3. Identify the **contexts where the property is used**.
4. Determine the **dependences between (2) and (3)**.
   In simple cases it is: "all set operations before any get operation".
5. Specify (2), (3), and the pattern of (4).

Try to locate the computations that **set or get properties** of an entity **in the context of the identifier**, if possible; avoid to propagate the **Key** values through the tree.

Use **SYMBOL computations** as far as possible (see design rules A).

Technique: Do it once

Task:

- Many occurrences of an identifier are bound to the same entity (key)
- For each entity a computation is executed at exactly one (arbitrary) occurrence of its identifier (e.g. output some target code)

Solution:

Compute an **attribute of type bool**:
True at exactly one occurrence of the key, false elsewhere.

Design steps:

1. Property specification: **`Done: int;`**
2. Set in name context, if not yet set.
3. Get in name context.
4. No dependences!
5. see on the right:

```
CLASS SYMBOL DoItOnce:
                  DoIt: int;

CLASS SYMBOL DoItOnce
      INHERITS IdentOcc COMPUTE
   SYNT.DoIt =
      IF (GetDone (THIS.Key, 0),
         0,
         ORDER
          (ResetDone (THIS.Key, 1),
          1));
END;
```

Anwendung:

```
SYMBOL StructName INHERITS DoITOnce
COMPUTE
   SYNT.Text =
      IF (THIS.DoIt,
         PTGTransform (...),
         PTGNULL);
END;
```


5. Binding Names to Entities

Names in the source code represent **entities** to describe the meaning of the text.

Occurrences of names are bound to **entities**.

Scope rules of the language specify how names are to be bound. E.g.:

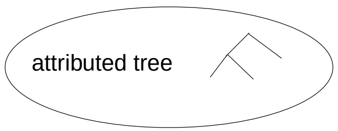
- Every name **a**, used as a structure name or as a type name is bound to the same entity.
- A type name **a** is an applied occurrence of a name. There must be a defining occurrences of **a** somewhere in the text.
- Field names are bound separately for every structure.

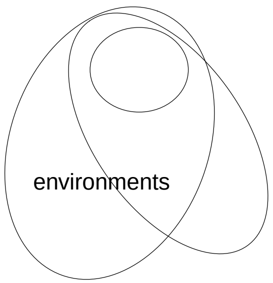
some occurrences of names: **some bindings:** **some entities:**

```
Customer ( addr: Address;
           account:int;
)
Address  ( name: String;
           zip:  int;
)
Article  ( name:  String;
           price: int;
)
```

- a structure (named **`Address`**)
- a field (named **`name`**)
- a Structur (named **`Article`**)
- a different field (named **`name`**)
- ...

Keys and Properties



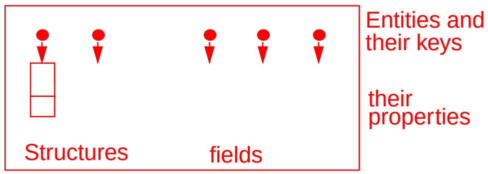


Eli tools implement properties of entities and of envivronments

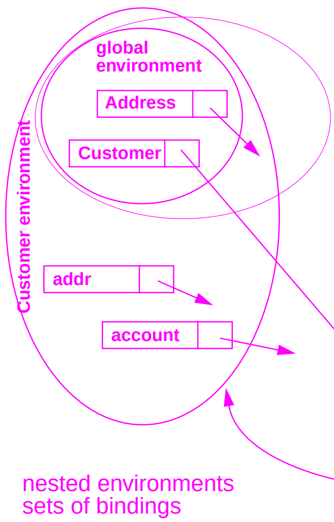
Entities are represented by keys.
Properties are associated to them.

Structures have a property called **`Environment`**

Definition module



Bindings and Environments



Environment: nested sets of bindings

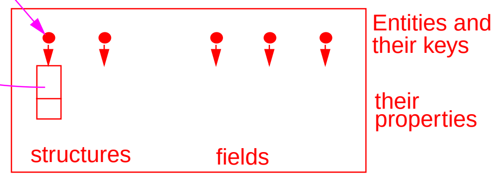
Binding: associates a name with a key

The global environment binds all structure and type names.

The environment of a structure binds its field names.

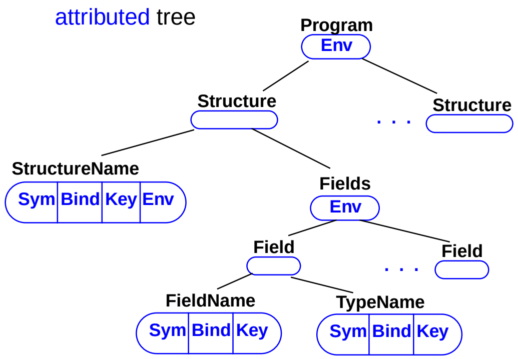
Eli tools implement properties of entities and of envivronments

Definition module



Attributed Tree for Name Analysis

Attributes of the tree nodes describe properties of the program construct



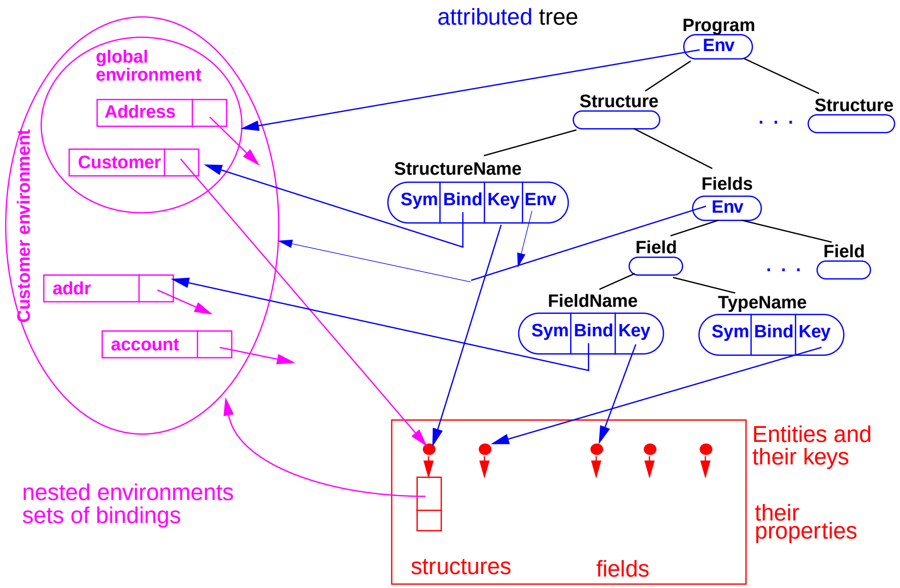
Program has the global environment

StructureName and Fields have the environment of the structure

Every node for a name occurrences has attributes for

- the code of the identifier,
- the binding of its name, and
- its key

Attributes, Environments, and Keys



Environment Module

Implements the abstract data type **Environment**:
hierarchally nested sets (tree) of **bindings (name, environment, key)**

Functions:

NewEnv () — creates a new environment e, that is the root of a new tree; used in **root context**

NewScope (e_1) — creates a new environment e_2 that is nested in e_1. Every binding of e_1 is a binding of e_2, too, if it is not hidden by a binding established for the same name in e_2; used in **range context**

BindIdn (e, id) — creates a new binding (id, e, k), if e does not yet have a binding for id; k is then a new key for a new entity; the result is in both cases the binding (id, e, k); used for **defining occurrences**.

BindingInEnv (e, id) — yields a binding (id, e_1, k) of e oder of a surrounding environment of e; if there is no such binding it yields NoBinding; used for **applied occurrences**

BindingInScope (e, id) — yields a binding (id, e, k) of e, if e directly contains such a binding; NoBinding otherwise; e.g. used for **qualified names**

Example: Names and Entities for the Structure Generator

Abstract syntax

```
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;
```

Different nonterminals for identifiers in different roles, because different computations are expected, e.g. for defining and applied occurrences.

Computation of Environment Attributes

Root of the environment hierarchy

`Fields` play the **role of a Range**.

The inherited computation of **Env** is overridden.

```
SYMBOL Descriptions INHERITS RootScope END;

SYMBOL Fields  INHERITS RangeScope END;

RULE: Structure ::= StructureName '(' Fields ')'
COMPUTE
   Fields.Env = StructureName.Env;
END;
```

Each structure entity has an **environment as its property**.

It is **created only once** for every occurrence of a structure entity.

That environment is **embedded in the global environment.**

In that environment the field names are bound.

```
SYMBOL StructureName COMPUTE
   SYNT.GotEnvir =
      IF (EQ (GetEnvir (THIS.Key, NoEnv), NoEnv),
         ResetEnvir
            (THIS.Key,
             NewScope (INCLUDING Range.Env)));

   SYNT.Env =
      GetEnvir (THIS.Key, NoEnv) <- SYNT.GotEnvir;
END;
```


Defining and Applied Occurrences of Identifiers

Computations **`IdentOcc`** for all identifier occurrences.

```
CLASS SYMBOL IdentOcc: Sym: int,
CLASS SYMBOL IdentOcc COMPUTE
   SYNT.Sym = TERM;
END;
```

All defining occurrences **bind** their names in the **next enclosing Range**

```
SYMBOL StructureName
   INHERITS IdentOcc, IdDefScope END;
SYMBOL ImportName
   INHERITS IdentOcc, IdDefScope END;
SYMBOL FieldName
   INHERITS IdentOcc, IdDefScope END;
```

Bind an applied occurrence of an identifier in the enclosing environment;
report an error if there is no valid binding.

```
SYMBOL TypeName
   INHERITS IdentOcc, IdUseEnv, ChkIdScope END;
```


6. Structured Output

Generator outputs structured text:

- programm in a suitable programming language
- data in suitable form (e.g. XML) to be processed by specific tools
- text in suitable form (e.g. HTML) to be presented by a text processor

Transformation phase of the generator defines the structure of the texts:

- parameterized text patterns
- instances of text patterns hierarchally nested

a text pattern with 2 parameters:

`#define ▭ Kind ▭`

2 instances:

`#define intKind 1`

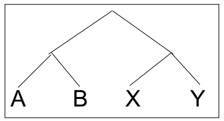
`#define PairPtrKind 2`

```
#ifndef WRAPPER_H
#define WRAPPER_H

#include "Pair.h"

#define noKind      0
#define intKind 1
#define PairPtrKind 2
#define floatKind 3

class intWrapper;
class PairPtrWrapper;
class floatWrapper;

class Object {
public:
 class WrapperExcept {};
 int getKind () { return kind; }

 int getintValue ();
 PairPtr getPairPtrValue ();
 float getfloatValue ();
protected:
 int kind;
};
```


„Structure Clash“ on Text Output

abstract program tree
drives creation of the target text by a tree walk

target text
is composed of fragments



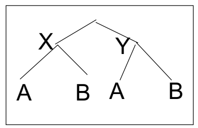
tree walk **order does not fit** to sequence of target text fragments

X A B Y A B

solution: text is composed into a buffer, and sequentially written from there

here:

the buffer is a tree or DAG representing pattern applications



PTG: Pattern-Based Text Generator

Generates **constructor functions** from
specifications of text patterns

A. PTG provides a
Specification language for text patterns
each is a sequence of text fragments and
insertion points

```
#define int Kind 1
```

B. PTG generates constructor functions
that build a data structure of pattern applications

one function per pattern
one parameter per insertion point

The functions are called on the tree walk.

C. PTG generates output functions
they walk recursively through the
data structure to output the target text

PTG's Specification Language: Introductory Example

Pattern: named sequence of C string literals and insertion points

```
KindDef:
   "#define " $ string "Kind \t" $ int "\n"

WrapperHdr:
   "#ifndef WRAPPER_H\n"
   "#define WRAPPER_H\n\n"
   $1 /* Includes */

   "\n#define noKind      0\n"
   $2 /* KindDefs */
   "\n"

   $3 /* ClassFwds */
   "\n"

   "class Object {\n"
   "public:\n"
   "  class WrapperExcept {};\n"
   "  int getKind () { return kind; }\n"
   $4 /* ObjectGets */
   "protected:\n"
   "  int kind;\n"
   "};\n\n"
```

```
#define int Kind 1
```

```
#ifndef WRAPPER_H
#define WRAPPER_H

#include "Pair.h"

#define noKind      0
#define intKind 1
#define PairPtrKind 2
#define floatKind 3

class intWrapper;
class PairPtrWrapper;
class floatWrapper;

class Object {
public:
 class WrapperExcept {};
 int getKind () { return kind; }

 int getintValue ();
 PairPtr getPairPtrValue ();
 float getfloatValue ();
protected:
 int kind;
};
```


Constructor Functions

A **constructor function** for each pattern.

A parameter for each insertion point:

```
PTGNode PTGKindDef (char *a, int b) {...}

PTGNode PTGWrapperHdr (PTGNode a, PTGNode b, PTGNode c, PTGNode d)
   {...}
```

Call of a constructor function

- creates an instance of the pattern with the supplied arguments and
- yields a reference to that instance

```
ik = PTGKindDef ("int", 1);

hdr = PTGWrapperHdr (ik, xx, yy, zz);
```

The arguments of calls are such references (type **PTGNode**) or they are values of the type specified in the pattern (e. g. string or int)

Such calls are used to **build the data structure bottom-up.**
It is acyclic, a DAG.

Output Functions

Predefined output functions:

- Call:

  ```
  PTGOutFile ("example.h", hdr);
  ```

  initiates a recursive walk through the data structure
  starting from the given node (2nd argument)
- All text fragments of all pattern instances are output in the specified order.
- Shared substructures are walked through and are output on each visit from above.
- User defined functions may be called during the walk, in order to cause side-effects (e.g. set and unset indentation).

Important Techniques for Pattern Specification

Elements of pattern specifications:

- string literals in C notation `"Value ();\n"`
- value typed insertion points `$string` `$int`
- untyped insertion points (**PTGNode**) `$` `$1`
- comments in C notation `$ /* Includes */`
  e.g. to explain the purpose of insertion points

All charaters that **separate tokens** in the output and that **format the output** have to be **explicitly specified** using string literals `" "` `";\n"` `"\tpublic:"`

Identifiers can be augmented by prefixes or suffixes:

```
KindDef: "#define "$ string "Kind \t" $ int "\n"
```

may yield

```
#define PairPtrKind    2
```

There are advanced techniques to create „pretty printed“ output (see PTG documentation).

Important Techniques: Indexed Insertion Points

Indexed insertion points: $1 $2 ...

1. Application: **one argument is to be inserted at several positions**:

```
ObjectGet: "  " $1 string " get" $1 string "Value ();\n"
```

call: `PTGObjectGet ("PairPtr")` result: `PairPtr getPairPtrValue ();`

2. Application: **modify pattern - use calls unchanged**:

today: `Decl: $1 /*type*/ " " $2 /*names*/ ";\n"`

tomorrow: `Decl: $2 /*names*/ ": " $1 /*type*/ ";\n"`

unchanged call: `PTGDecl (tp, ids)`

Rules:

- If n is the greatest index of an insertion point the constructor function has n parameters.
- If an index does not occur, its parameter exists, but it is not used.
- The order of the parameters is determined by the indexes.
- Do not have both indexed and non-indexed insertion points in a pattern.

Important Techniques: Typed Insertion Points

Untyped insertion points: `$` `$1`

Instances of patterns are inserted, i.e. the results of calls of constructor functions
Parameter type: **PTGNode**

Typed insertion points: `$ string` `$1 int`

Values of the given type are passed as arguments and output at the required position
Parameter type as stated, e.g. **`char*`, `int`**, or other basic types of C

```
KindDef: "#define " $ string "Kind \t" $ int "\n"
```

call: `PTGKindDef ("PairPtr", 2)`

Example for an application: generate identifiers

```
KindId:      $ string "Kind"          PTGKindId("Flow")
CountedId:   "_" $ string "_" $ int   PTGCountedId("Flow", i++)
```

Example for an application: conversion into a pattern instance

```
AsIs:   $ string   PTGAsIs("Hello")
Numb:   $ int      PTGNumb(42)
```

Rule:

- **Same index** of two insertion points **implies the same types**.

Important Techniques: Sequences of Text Elements

Pairwise concatenation:

```
Seq: $ $          PTGSeq(PTGFoo(...),PTGBar(...))
                  res = PTGSeq(res, PTGFoo(...));
```

The application of an empty pattern yields PTGNULL

```
PTGNode res = PTGNULL;
```

Sequence with optional separator:

```
CommaSeq: $ {", "} $          res = PTGCommaSeq (res, x);
```

Elements that are marked optional by `{}` are not output,
if at least one insertion has the value **PTGNULL**

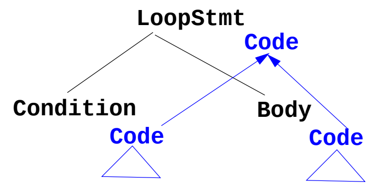
Optional parentheses:

`Paren: {"("} $ {")"}` no `( )` around empty text

The Eli specification **`$/Output/PtgCommon.fw`** makes some of these useful pattern definitions available: **`Seq, CommaSeq, AsIs, Numb`**

Compose Target Text in Adjacent Contexts

Attributes in adjacent tree contexts

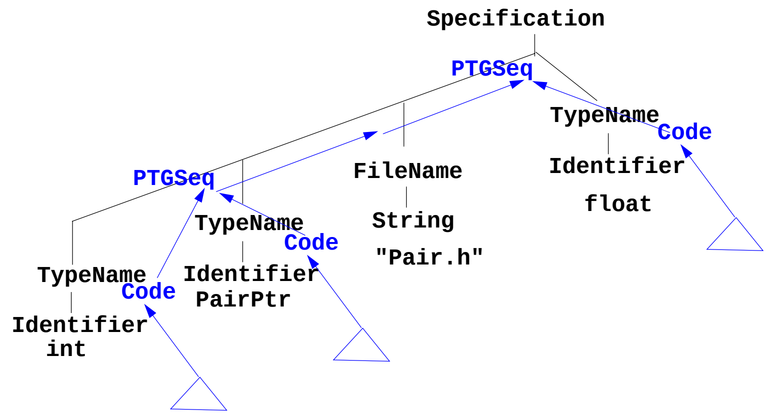



```
ATTR Code: PTGNode;
RULE: LoopStmt ::= Condition Body COMPUTE
   LoopStmt.Code =
      PTGWhile (Condition.Code, Body.Code);
END;
```

Application of the **While** pattern

Compose Subtree Elements

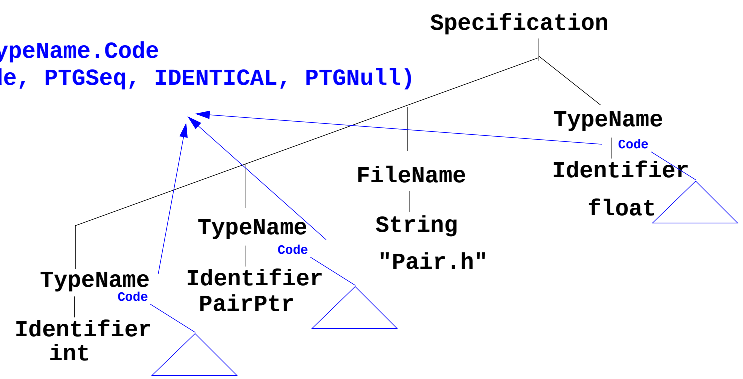
Example wrapper generator; consider abstract program tree for some input:
Specification is a sequence of tree nodes of type **TypeName** and **FileName**



Attributes **TypeName.Code** contain references to created pattern applications; they are composed by **PTGSeq** applications.

CONSTITUENTS Composes Attributes of a Subtree

```
CONSTITUENTS TypeName.Code
   WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull)
```



CONSTITUENTS composes **TypeName.Code** attributes of the subtree

	WITH (PTGNode,	PTGSeq,	IDENTICAL,	PTGNull)
Meaning:	type	dyadic composition function	monadic composition function	constant function for optional subtrees

7. Library of Specification Modules

A reusable specification modul

- solves a frequently occurring task, e.g. name analysis according Algol-like scope rules,
- provides abstract symbol roles (**CLASS**) with computations that contribute to the solution of the task, z. B. **IdUseEnv** for applied occurrences,
- contains all specifications, functions, etc. that are necessary to implement the task's solution (FunnelWeb file)
- is a member of a library of modules that support related topics, e.g. name analysis according to different scope rules
- has a descriptive documentation

Users

- select a suitable module,
- instantiate it,
- let symbols of their abstract syntax inherit some of the symbol roles,
- use the computed attributes for their own computations.

Basic Module for Name Analysis

Symbol roles:
Grammar root:

```
SYMBOL Program INHERITS RootScope END;
```

Ranges containing definitions:

```
SYMBOL Block INHERITS RangeScope END;
```

Defining identifier occurrence:

```
SYMBOL DefIdent INHERITS IdDefScope END;
```

Applied identifier occurrence:

```
SYMBOL UseIdent
   INHERITS IdUseEnv,ChkIdUse END;
```

Provided attributes:

```
   DefIdent, UseIdent: Key, Bind
   Program, Block: Env
```

Instantiation
in a .specs file
for Algol-like scope rules:

```
$/Name/AlgScope.gnrc:inst
```

for C-like scope rules:

```
$/Name/CScope.gnrc: inst
```

for a new name space

```
$/Name/AlgScope.gnrc
      +instance=Label
      :inst
```

Symbol roles:
`LabelRootScope`,
`LabelRangeScope`, ...

Specification Libraries in Eli

Contetnts of the Eli Documentation
Specification Module Library:

- Introduction of a running example
- How to use Specification Modules

- Name analysis according to scope rules
- Association of properties to definitions
- Type analysis tasks
- Tasks related to input processing
- Tasks related to generating output
- Abstract data types to be used in specifications
- Solutions of common problems

- Migration of Old Library Module Usage

Name Analysis, Type Analysis

Name analysis according to scope rules

- Tree Grammar Preconditions
- Basic Scope Rules, 3 variants: Algol-like, C-like, Bottom-Up
- Predefined Identifiers
- Joined Ranges (3 variants)
- Scopes being Properties of Objects (4 variants)
- Inheritance of Scopes (3 variants)
- Name Analysis Test
- Environment Module

Type analysis tasks

- Types, operators, and indications
- Typed entities
- Expressions
- User-defined types
- Structural type equivalence
- Error reporting in type analysis
- Dependence in type analysis

Association of Properties to Entities

Association of properties to definitions

- Common Aspects of Property Modules
- Count Occurrences of Objects
- Set a Property at the First Object Occurrence
- Check for Unique Object Occurrences
- Determine First Object Occurrence
- Map Objects to Integers
- Associate Kinds to Objects
- Associate Sets of Kinds to Objects
- Reflexive Relations Between Objects
- Some Useful PDL Specifications

Input and Output

Tasks related to input processing

- Insert a File into the Input Stream
- Accessing the Current Token
- Command Line Arguments for Included Files

Tasks related to generating output

- PTG Output for Leaf Nodes
- Commonly used Output patterns for PTG
- Indentation
- Output String Conversion
- Pretty Printing
- Typesetting for Block Structured Output
- Processing Ptg-Output into String Buffers
- Introduce Separators in PTG Output

Other Useful Modules

Abstract data types to be used in specifications

- Lists in LIDO Specifications
- Linear Lists of Any Type
- Bit Sets of Arbitrary Length
- Bit Sets of Integer Size
- Stacks of Any Type
- Mapping Integral Values To Other Types
- Dynamic Storage Allocation

Solutions of common problems

- String Concatenation
- Counting Symbol Occurrences
- Generating Optional Identifiers
- Computing a hash value
- Sorting Elements of an Array
- Character string arithmetic

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.

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:

```
class Customer_Cl {
private:
```

Fields:

```
   Address addr_fld;
   int account_fld;
public:
```

Initializing constructor:

```
   Customer_Cl (Address addr, int account)
      {addr_fld=addr; account_fld=account; }
```

set- and get-methods for fields:

```
   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 {
...
```

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;
```


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"
```


Task Decomposition Determines the Architecture of the Generator

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

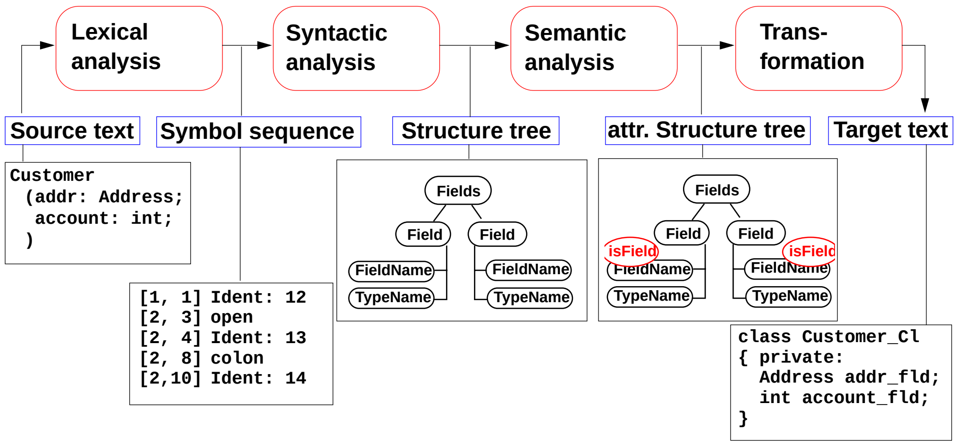
Input processing
Scanning
Symbol coding
Conversion

Parsing
Tree construction

Name analysis
Definition table
Property analysis

Text generation

Attribute computation in the tree



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
```

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;
```


Name Analysis

Described in GSS 5.8 to 5.11

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;
```


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.

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;
```


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;
```


9. Individual Projects
Steps for the Development of a Generator

1. Task Definition
   a. Task description
   b. Examples for input (DSL)
   c. Examples for generated output
   d. Description of analysis and transformation tasks
2. Structuring Phase
   a. Develop concrete syntax
   b. Specify notation of tokens
   c. Develop abstract syntax
   d. Comprehensive tests
3. Semantic Analysis
   a. Characterize erroneous inputs by test cases
   b. Specify binding of names
   c. Specify computation and checks of properties
   d. Comprehensive tests
4. Transformation
   a. Develop output patterns
   b. Develop computations to create output
   c. Comprehensive tests
5. Documentation and Presentation of the Generator

Individual Projects in Current Lecture

	Topic	Student team
A		
B		
C		
D		
E		
F		
G		
H		

10. Visual Languages Developed using DEViL

Two conference presentations are available in the lecture material:

Domain-Specific Visual Languages: Design and Implemenation

Uwe Kastens, July 2007 CoRTA

Outline:

1. What are visual languages?

2. Domain-specific visual languages

3. Ingredients for Language design

4. A Development Environment for Visual Languages

5. Pattern-Based Specifications in DEViL

Specifying Generic Depictions of Language Constructs for 3D Visual Languages

Jan Wolter, September 2013, VL / HCC

Outline:

1. 3D Visual Languages

2. DEViL3D - Generator Framework for 3D Visual Languages

3. Generic Depictions