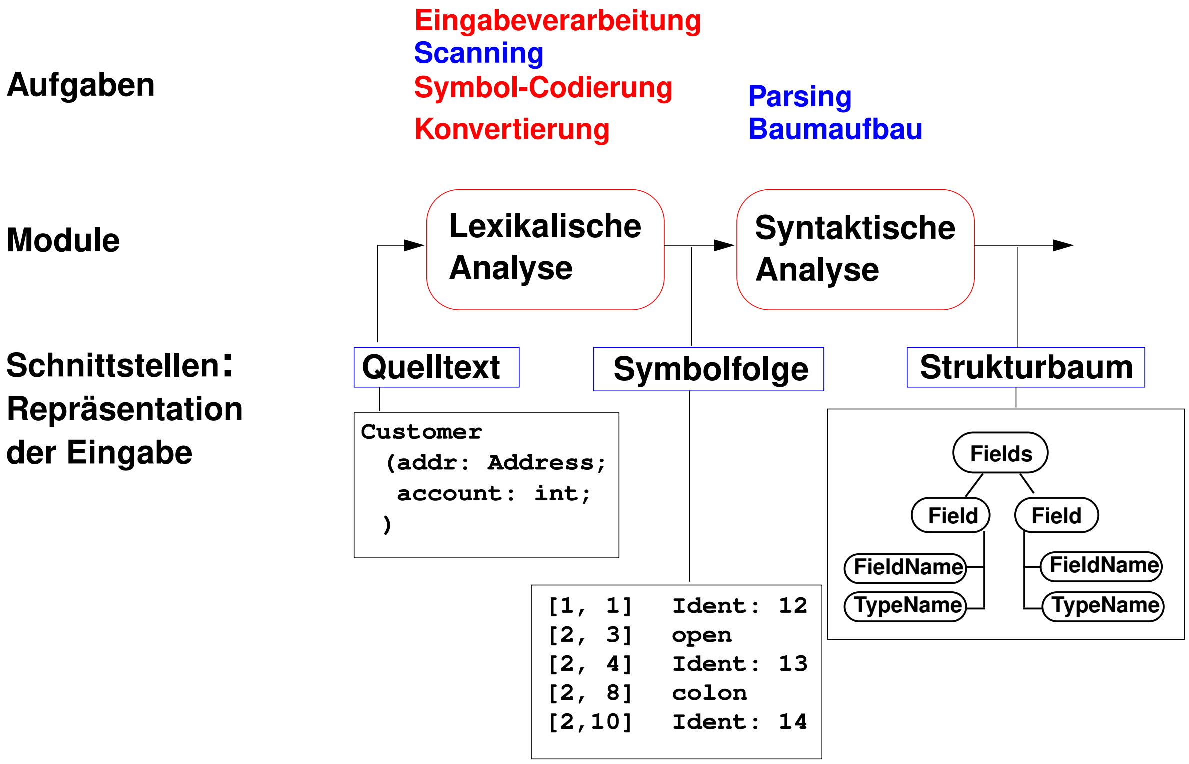
3. Bäume aufbauen - Überblick

Notation und Struktur der Eingabe prüfen und als Baum repräsentieren



Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 301

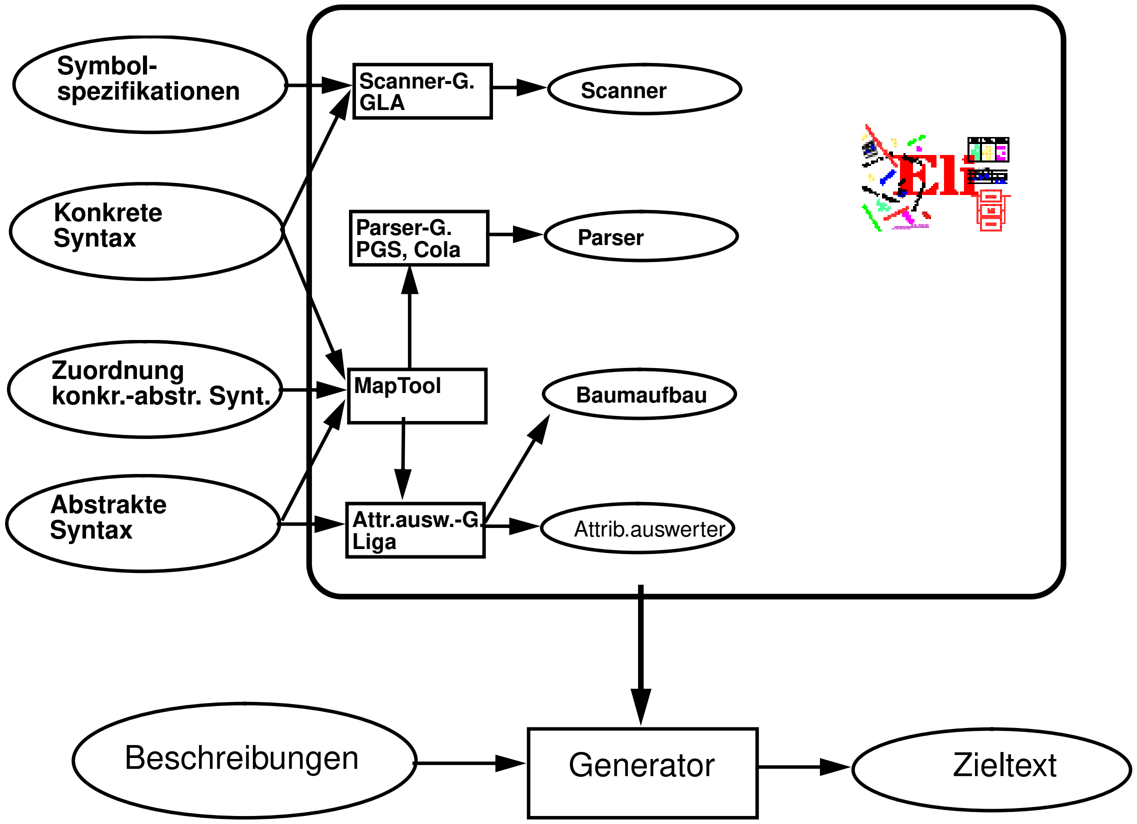
Ziele:

Strukturierungsphase verstehen

in der Vorlesung:

- An die Aufgaben aus Folie GSS-1.15 erinnern,
- Aufgaben und Repräsentationen erläutern.

Eli: Spezifikation und Generierung des Baumaufbaus



Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 302

Ziele:

Generierung der Strukturierungsphase verstehen

in der Vorlesung:

Themen der Folie erläutern:

- Rolle der Spezifikationen
- Aufgabe der Generatoren
- Zusammenwirken der Generatoren

Spezifikationen am Beispiel des Wrapper-Generators

Symbol-spezifikationen

Notation der Grundsymbole
.gla

```
Identifier:   C_IDENTIFIER
String:       C_STRING_LIT
              C_COMMENT
```

Konkrete Syntax

Struktur der Eingabe
.con

```
Specification:  Sequence.
Sequence:       Sequence Element / .
Element:        TypeName ';'.
Element:        FileName.
TypeName:       Identifier.
FileName:       String.
```

Zuordnung konkr.-abstr. Synt.

.map

leer -
konkrete und abstrakte Syntax sind gleich

Abstrakte Syntax

Struktur der Bäume
.lido

```
RULE: Specification LISTOF Element
COMPUTE ...

SYMBOL FileName COMPUTE ...
SYMBOL TypeName COMPUTE ...
```

nur die Symbole und Produktionen, die Berechnungen haben

Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 303

Ziele:

Einfaches Beispiel

in der Vorlesung:

Eindruck von den Spezifikationen vermitteln

Strukturierung am Beispiel Terminliste

Neues Beispiel für die Spezifikationen der Strukturierung bis zum Baumaufbau:

Eingabesprache: Listen von Terminangaben:

```
1.11.       20:00      "Theaterbesuch"
Do          14:15      "GSS Vorlesung"
werktags    12:05      "Essen im Palmengarten"
Mo, Do      8:00       "Dekanat"
31.12.      23:59      "Jahresende"
12/31       23:59      "End of year"
```


Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 304

Ziele:

Neues Beispielthema einführen

in der Vorlesung:

Aufgabe anhand der Beispieleingabe erläutern

Konkrete Syntax

spezifiziert die **Struktur der Eingabe** durch eine kontext-freie Grammatik:

```
Termine:       Eintrag+ .
Eintrag:       Tag Zeit Beschreibung .
Tag:           Datum / Muster .
Datum:         TagNum '.' MonNum '.' /
               MonNum '/' TagNum .
TagNum:        Integer .
MonNum:        Integer .
Muster:        Muster ',' WochenTag /
               WochenTag /
               TagesGruppe .
WochenTag:     TagesName .
TagesGruppe:   'werktags' .
Zeit:          Zeitpunkt .
Zeitpunkt:     Uhrzeit .
```

Notation:

- Folge von Produktionen
- literale Terminale in '
- EBNF-Konstrukte:
  / Alternative
  ( ) Klammern
  [ ] Option
  +, * Wiederholung
  // Wiederholung mit Trenner
  Bedeutung siehe GdP

Beispiel:

```
Do          14:15     "GSS Vorlesung"
werktags    12:05     "Essen im Palmengarten"
Mo, Do      8:00      "Dekanat"
31.12.      23:59     "Jahresende"
12/31       23:59     "End of year"
```


Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 305

Ziele:

Syntax-Notation kennenlernen

in der Vorlesung:

- Entwurf der Produktionen und
- Schreibweise der Produktionen erläutern.
- Bezug zum Eingabe-Beispiel erläutern

Abstrakte Syntax

spezifiziert die **Strukturbäume** durch eine kontext-freie Grammatik:

```
RULE pTermine:      Termine   LISTOF Eintrag              END;
RULE pZeitEintrag:  Eintrag   ::= Tag Zeit Beschreibung   END;

RULE pDatum:        Tag       ::= Datum                   END;
RULE pMuster:       Tag       ::= Muster                  END;

RULE pDatumZahlen:  Datum     ::= TagNum MonNum           END;
RULE pTag:          TagNum    ::= Integer                 END;
RULE pMonat:        MonNum    ::= Integer                 END;

RULE pMusterListe:  Muster    LISTOF Element              END;
RULE pTagesName:    Element   ::= TagesName               END;
RULE pWerktage:     Element   ::= 'werktags'              END;

RULE pZeitpunkt:    Zeit      ::= Zeitpunkt               END;
RULE pUhrzeit:      Zeitpunkt ::= Uhrzeit                 END;
```

Notation:

- Sprache Lido für Berechnungen in Strukturbäumen
- optional benannte Produktionen,
- kein EBNF außer LISTOF (evtl. leere Folge)

Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 306

Ziele:

Notation der abstrakten Syntax kennenlernen

in der Vorlesung:

- Entwurf der Produktionen und
- Schreibweise der Produktionen erläutern.

Beispiel eines Strukturbaumes

- Produktionsnamen als Knotentypen
- Werte der Terminale an den Blättern

Ausgabe automatisch erzeugt durch Eli's Unparser-Generator

```
pZeitEintrag(
   pDatum(pDatumZahlen(pTag(1),pMonat(11))),
   pZeitpunkt(pUhrzeit(1200)),"Theaterbesuch"),
pZeitEintrag(
   pMuster(pTagesName(4)),
   pZeitpunkt(pUhrzeit(855)),"GSS Vorlesung"),
pZeitEintrag(
   pMuster(pWerktage()),
   pZeitpunkt(pUhrzeit(725)),"Essen im Palmengarten"),
pZeitEintrag(
   pMuster(pTagesName(1),pTagesName(4)),
   pZeitpunkt(pUhrzeit(480)),"Dekanat"),
pZeitEintrag(
   pDatum(pDatumZahlen(pTag(31),pMonat(12))),
   pZeitpunkt(pUhrzeit(1439)),"Jahresende"),
pZeitEintrag(
   pDatum(pDatumZahlen(pTag(31),pMonat(12))),
   pZeitpunkt(pUhrzeit(1439)),"End of year")
```


Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 307

Ziele:

Ausgabe des Strukturbaum in Klammer-Notation

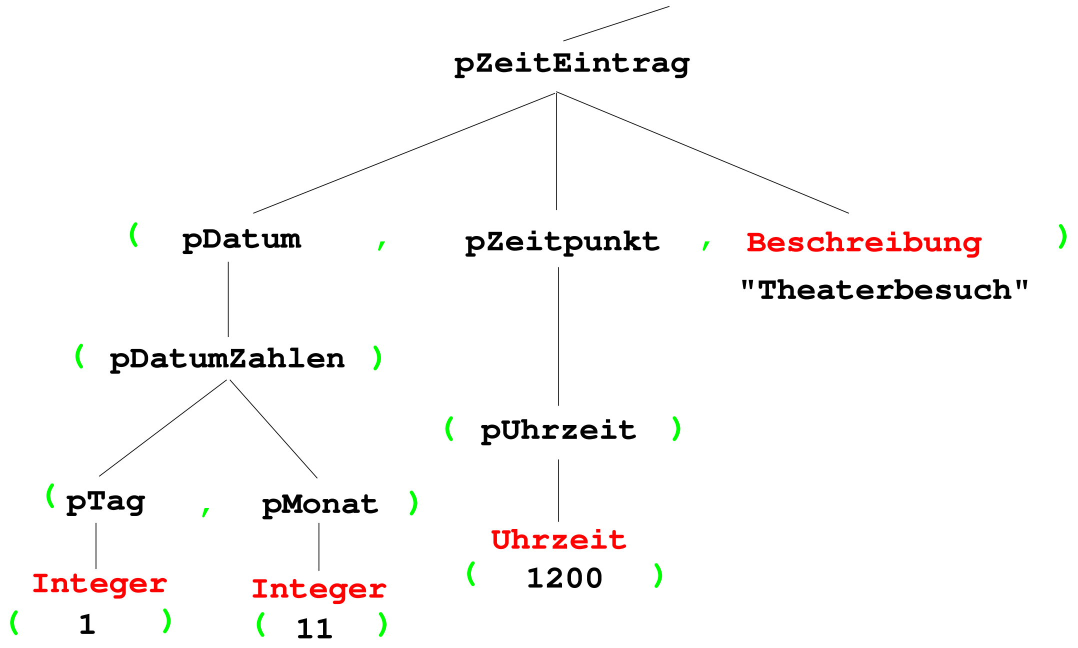
in der Vorlesung:

- Bezug zum Eingabe-Beispiel erläutern
- Bezug zur abstrakten Syntax erläutern.

Strukturbaumausschnitt graphisch

- Produktionsnamen als Knotentypen
- Werte der Terminale an den Blättern

Ausgabe automatisch erzeugt durch Eli‘s Unparser-Generator, Baumstruktur durch Klammerung



Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 307a

Ziele:

Baumdarstellung verstehen

in der Vorlesung:

Zusammenhang zwischen abstrakter Syntax (= Baumgrammatik) und textueller Darstellung zeigen

Symbolabbildung: konkrete - abstrakte Syntax

konkrete Syntax:

```
Muster:       Muster ',' WochenTag /
              WochenTag /
              TagesGruppe .
WochenTag:    TagesName .
TagesGruppe: 'werktags' .
```

Vereinfachung der abstrakten Syntax

Menge von Nichtterminalen der konkreten Syntax ->

ein Nichtterminal der abstrakten Syntax

Abbildung:

```
MAPSYM
Element ::= TagesGruppe WochenTag.
```

Element steht in der abstrakten Syntax für die angegebenen Nichtterminale

abstrakte Syntax:

```
RULE pMusterListe:  Muster    LISTOF Element          END;
RULE pTagesName:    Element   ::= TagesName           END;
RULE pWerktage:     Element   ::= 'werktags'          END;
```


Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 308

Ziele:

Vereinfachung des Strukturbaumes

in der Vorlesung:

- Symbolabbildung am Beispiel erläutern.
- Siehe auch Symbolabbildung zur Vereinfachung von Ausdrucksgrammatiken (GdP-2-9)

Regelabbildung

konkrete Syntax:

```
Datum:        TagNum '.' MonNum '.' /
              MonNum '/' TagNum .
```

Abbildung:

```
MAPRULE
Datum: TagNum '.' MonNum '.' < $1 $2 >.
Datum: MonNum '/' TagNum     < $2 $1 >.
```

unterschiedliche Produktionen der konkreten Syntax

werden in der abstrakten Syntax **vereinheitlicht**

abstrakte Syntax:

```
RULE pDatumZahlen:   Datum ::= TagNum MonNum END;
```


Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 309

Ziele:

Vereinfachung des Strukturbaumes

in der Vorlesung:

- Regelabbildung am Beispiel erläutern.
- Siehe auch Vereinfachung von Ausdrucksgrammatiken (GdP-2-9)
- Abstrakte Syntax kann auch ohne explizite Spezifikation aus der konkreten generiert werden.
- Konkrete Syntax kann auch ohne explizite Spezifikation aus der abstrakten generiert werden.
- Dies kann auch stückweise in beiden Richtungen geschehen.

Literale und nicht-literale Terminale

Definition der Notation von

- literalen Terminalen (unbenannt): **in der konkreten Syntax**
- nicht-literalen Terminalen (benannt): in zusätzlicher **Spezifikation für den Scanner-Generator**

```
Termine:      Eintrag+ .
Eintrag:      Tag Zeit Beschreibung .
Tag:          Datum / Muster .
Datum:        TagNum '.' MonNum '.' /
              MonNum '/' TagNum .
TagNum:       Integer .
MonNum:       Integer .
Muster:       Muster ',' WochenTag /
              WochenTag /
              TagesGruppe .
WochenTag:    TagesName .
TagesGruppe:  'werktags' .
Zeit:         Zeitpunkt .
Zeitpunkt:    Uhrzeit .
```


Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 310

Ziele:

Klassifikation von Terminalen

in der Vorlesung:

Notation von Terminalen wirdunterschidlich festgelegt

Spezifikation nicht-literaler Terminale

Der Generator GLA generiert einen Scanner aus

- Notation der literalen Terminale
  von Eli aus der konkreten Syntax extrahiert
- Spezifikation der nicht-literalen Terminale
  in Dateien vom Typ **`.gla`**

Form der Spezifikationen:

```
Name:         $  regulärer Ausdruck                 [CodierFunktion]

TagesName:    $  Mo|Di|Mi|Do|Fr|Sa|So               [mktag]

Uhrzeit:      $(([0-9]|1[0-9]|2[0-3]):[0-5][0-9])  [mkuhrzeit]
```

Vorgefertigte Spezifikationen:

```
Beschreibung:   C_STRING_LIT
Integer:        PASCAL_INTEGER
```


Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 311

Ziele:

Scanner-Spezifikation verstehen

in der Vorlesung:

Themen der Folie erläutern:

- Notation der regulären Ausdrücke,
- Aufgabe und Schnittstelle der Codierfunktion,
- vergefertigte Spezifikationen

Scanner-Spezifikation: Reguläre Ausdrücke

Notation	**Zeichenfolgen, die akzeptiert werden**
`c`	das Zeichen c; außer Zeichen mit spezieller Bedeutung, siehe c
`\c`	Zwischenraum, Tab, Zeilenwechsel, `\".[]^()|?+*{}/$<`
`"s"`	die Zeichenfolge s
`.`	jedes einzelne Zeichen außer Zeilenwechsel
`[xyz]`	genau ein Zeichen aus der Menge {x, y, z}
`[^xyz]`	genau ein Zeichen, das nicht aus der Menge {x, y, z} ist
`[c-d]`	genau ein Zeichen, dessen ASCII-Code zwischen c und d einschließlich liegt
`(e)`	durch e spezifizierte Zeichenfolge
`ef`	durch e spezifizierte Zeichenfolge gefolgt von einer durch f spezifizierten
`e | f`	durch e oder durch f spezifizierte Zeichenfolge
`e?`	leere oder durch e spezifizierte Zeichenfolge
`e+`	ein oder mehrere Auftreten von Zeichenfolgen, die durch e spezifiziert sind
`e*`	leere oder durch e+ spezifizierte Zeichenfolge
`e {m,n}`	mindestens m und höchstens n Zeichenfolgen, die durch e spezifiziert sind

e und f sind reguläre Ausdrücke, wie sie hier definiert sind.

Jeder reguläre Ausdruck **akzeptiert die längste Zeichenfolge**, die seiner Definition genügt.

Auflösung von **Mehrdeutigkeiten**:
1. die **längere akzeptierte Zeichenfolge**
2. bei gleicher Länge: die **früher angegebene Regel**

Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 311a

Ziele:

Notation von regulären Ausdrücken

in der Vorlesung:

Anwendung der Konstrukte erläutern:

Scanner-Spezifikation: Programmierte Scanner

Das Akzeptieren bestimmter Zeichenfolgen kann durch eine Funktion implementiert werden. Damit kann man Zeichenfolgen operational spezifizieren, die durch reguläre Ausdrücke nicht oder schwierig beschreibbar sind.

Der Anfang der Zeichenfolge wird durch einen regulären Ausdruck spezifiziert, es folgt der geklammerte Name der Funktion. z. B. für Zeilenkommentare wie in Ada

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

Parameter der Funktion: ein Pointer auf das erste Zeichen der bisher akzeptiertenFolge, und ihre Länge. Ergebnis: Pointer auf das erste Zeichen nach der gesamten Folge:

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

Einige verfügbare programmierte Scanner und was sie akzeptieren:

auxEOL	alle Zeichen bis zum nächsten Zeilenwechsel einschließlich
auxCString	ein C-String-Literal nach dem einleitenden "
auxM3Comment	Modula-3-Kommentar nach dem einleitenden (*, mit (* und *) geklammert, auch geschachtelt
Ctext	C-Verbundanweisungen nach dem einleitenden { mit { und } geklammert, auch geschachtelt

Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 311b

Ziele:

Einsatzmöglichkeiten erkennen

in der Vorlesung:

- Prinzip und Beispiele erläutern.
- Auf die Liste in der Dokumentation verweisen.

Scanner-Spezifikation: Codier-Funktionen

Die **akzeptierte Zeichenfolge** wird an eine Codier-Funktion übergeben (`start`, `length`)

Sie liefert die Codierung des akzeptierten Symbols (`intrinsic`)
d. h. eine ganze **Zahl, die seine Identität oder sein Wert repräsentiert.**

Dazu wird ggf. die Zeichenfolge **gespeichert und/oder konvertiert**.

Alle Codier-Funktionen haben dieselbe **Signatur**:

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

Die **Symbolklasse** (Terminalsymbol) kann ggf. geändert werden (`class`), z. B. um Wortsymbole von Bezeichnern zu unterscheiden.

Verfügbare Codierfunktionen:

`mkidn`	Zeichenfolge in eine Hash-Tabelle eintragen und bijektiv codieren
`mkstr`	Zeichenfolge speichern; jede erhält einen neuen Code
`c_mkstr`	C-String-Literal in seinen Wert konvertieren, speichern und codieren
`mkint`	Ziffernfolge in eine ganze Zahl konvertieren und diesen Wert als Code liefern
`c_mkint`	Literal für ganze Zahlen in C konvertieren und diesen Wert als Code liefern

Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 311c

Ziele:

Prinzip und Einsatzmöglichkeiten erkennen

in der Vorlesung:

- Prinzip und Beispiele erläutern.
- Auf die Liste in der Dokumentation verweisen.

Scanner-Spezifikation: Vorgefertigte Spezifikationen

Vorgefertigte vollständige Spezifikationen (regulärer Ausdruck, ggf. programmierter Scanner und Codier-Funktion) können **unter ihrem Namen** abgerufen werden:

```
Identifier:  C_IDENTIFIER
```

Für viele Symbole aus Programmiersprachen sind vorgefertigte Spezifikationen verfügbar (vollständige Spezifikation in der Dokumentation):

```
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
```

werden nur benutzt, wenn irgendein Symbol mit solch einem Zeichen beginnt, diese Zeichen aber trotzdem Symbole trennen sollen.

Die Codier-Funktionen können auch überschrieben werden.

Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 311d

Ziele:

Vorrat kennenlernen

in der Vorlesung:

- Einige der Symbole erläutern
- Auf die Beschreibung in der Dokumentation verweisen.

Generierung der Baumausgabe

Strukturbaum mit Knotentypen und Werten an den terminalen Blättern ausgeben:

```
pZeitEintrag(
   pDatum(pDatumZahlen(pTag(1),pMonat(11))),
   pZeitpunkt(pUhrzeit(1200)),"Theaterbesuch"),
pZeitEintrag(
```

Ausgabe durch Aufrufe von Pattern-Konstruktor-Funktionen in Baumkontexten

Spezifikationen dafür kann der Unparser-Generator automatisch **erzeugen**:

Unparser generiert aus abstrakter Syntax:

```
Abstract.lido:tree
```

Ausgabe an der Grammatik-Wurzel:

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

Ausgabe nicht-literaler Terminale:

```
TagesName:    $ int
Uhrzeit:      $ int
Integer:      $ int
```

Umbenennung einer PTG-Funktion:

```
#define PTGBeschreibung(x)PTGId(x)
```

Benutzung vordefinierter PTG-Pattern:

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


Vorlesung Generierung von Software aus Spezifikationen WS 2002 / Folie 312

Ziele:

Unparser-Generator benutzen

in der Vorlesung:

Rolle der Spezifikationen erläutern:

- Unparser-Generator generiert Spezifikationen (ptg und lido)!
- Nur Wurzel und Blätter der Baumgrammatik müssen manuell behandelt werden.
- Eine andere Variante des Unparser-Generators kann den Eingabetext reproduzieren (:idem statt :tree); nützlich für Spracherweiterungen.