

# Einführung in Gentle

von  
Ulrich Grude

Technische Fachhochschule Berlin  
Sommersemester 2007

Verbesserungsvorschläge und Hinweise auf Fehler sind jederzeit willkommen,  
am liebsten per email an grude [@tfh-berlin.de](mailto:grude@tfh-berlin.de).

**Inhaltsverzeichnis:**

|       |  |     |
|-------|--|-----|
| 1.    | Einleitung.....  | 4   |
| 2.    | hello_v1.g: Das unvermeidliche Hallo-Programm.....                       | 5   |
| 3.    | ahnen_v1.g: Ein einfacher Parser.....                                    | 7   |
| 3.1.  | Kommentare in den Eingabedateien eines Gentle-Programms:.....            | 10  |
| 3.2.  | Welche Zeichen sind zwischen Lexemen erlaubt?.....                       | 10  |
| 4.    | ahnen_v2.g: Ein Compiler, der Vorfahren in Zahlen übersetzt.....         | 11  |
| 5.    | ahnen_v3.g: Vorteile einer Zwischendarstellung.....                      | 13  |
| 6.    | ahnen_v4.g: Direkte Übersetzung.....                                     | 15  |
| 7.    | ahnen_v5.g: Übersetzung mit Zwischendarstellung.....                     | 16  |
| 8.    | ahnen_v6.g: Noch ein Ahnen-Compiler.....                                 | 19  |
| 9.    | Typen.....   | 22  |
| 10.   | Konkrete Syntax und abstrakte Syntax.....                                | 25  |
| 11.   | digits_1.g: Binärzahlen in Dezimalzahlen umwandeln.....                  | 28  |
| 12.   | digits_2.g: Hexadezimalzahlen in Binärzahlen umwandeln.....              | 29  |
| 13.   | Ein Scanner übersetzt Lexeme in Token.....                               | 31  |
| 13.1. | ides.g: Bezeichner mit einem STRING-Parameter.....                       | 32  |
| 13.2. | idei.g: Bezeichner mit einem IDENT-Parameter.....                        | 37  |
| 13.3. | int.g: Ganzzahl-Literale, natürlich mit einem INT-Parameter.....         | 41  |
| 14.   | calcul_1.g: Ein Interpreter für Ausdrücke.....                           | 44  |
| 15.   | Eine einfache Stapelmaschine und ihre Maschinensprache.....              | 46  |
| 16.   | calcul_2.g: Ein Compiler für Ausdrücke.....                              | 47  |
| 17.   | calcul_3.g: Ein Interpreter für erweiterte Ausdrücke.....                | 49  |
| 18.   | calcul_4.g: Polynome ableiten.....                                       | 51  |
| 19.   | Grundterme, Terme und Musterabgleich (pattern matching).....             | 54  |
| 20.   | Grundlegendes über Prädikate.....  | 56  |
| 21.   | Und/Oder-Bäume, tiefes und flaches Wiederaufsetzen.....                  | 65  |
| 22.   | altcon_1.g: Die Alternativen-Anweisung und die bedingte Anweisung.....   | 69  |
| 23.   | condit_1.g: Bedinungs-Prädikate (conditional predicates) und Listen..... | 70  |
| 24.   | condit_2.g: Bedingungs-Prädikate (conditional predicates) und Bäume..... | 73  |
| 25.   | sweep_v1.g: Feger-Prädikate (sweep predicates) und Bäume.....            | 77  |
| 26.   | Spezielle Muster und Terme.....  | 82  |
| 27.   | Unveränderbare Variablen und veränderbare Variablen.....                 | 85  |
| 28.   | Vordefinierte Prädikate.....   | 90  |
| 28.1. | Gleich und ungleich.....   | 90  |
| 28.2. | Größer, größergleich, kleiner, kleinergleich.....                        | 90  |
| 28.3. | Die print-Prädikate können alles mögliche ausgeben.....                  | 91  |
| 28.4. | Die where-Prädikate können unveränderbare Variablen erzeugen.....        | 91  |
| 28.5. | Das An-der-Position-Prädikat @.....                                      | 92  |
| 29.   | Verwaltung einer Symboltabelle.....                                      | 93  |
| 30.   | Verwaltung einer "blockorientierten Symboltabelle".....                  | 98  |
| 31.   | Wahl-Prädikate (choice predicates).....                                  | 103 |
| 32.   | Erzeugung eines Compilers mit Gentle.....                                | 110 |
| 33.   | Sachwortverzeichnis.....   | 120 |



## 1. Einleitung

Gentle ist eine problemorientierte Programmiersprache zum Schreiben von Compilern.

Compiler sind einerseits häufig sehr umfangreiche und komplexe Programme. Andererseits ähneln sich die Probleme, die verschiedene Compiler lösen müssen (kontextfreie Syntaxanalyse, Erzeugung einer internen Darstellung des zu compilierenden Programms, kontextsensitive Syntaxanalyse, Codeerzeugung etc.). Für diese Probleme wurden allgemeine, abstrakte Lösungsmethoden (z. B. die Methode des LR-Parsens) und konkrete Werkzeuge (z. B. der Parser-Generator `yacc`) entwickelt. Gentle faßt die wichtigsten dieser Methoden und Werkzeuge im Rahmen einer speziellen Programmiersprache zusammen.

Gentle ist eine relativ kleine Sprache. Ihre kontextfreie Grammatik paßt ohne große Mühe auf eine Seite. Trotzdem kann man alle Teile eines Compilers vollständig in Gentle schreiben. Für den Fall, dass man bestimmte Routinen "maschinennah und von Hand programmieren" möchte, bietet Gentle eine einfache Schnittstelle zur Sprache C. Außerdem kann man einen in Gentle geschriebenen Compiler durch sogenannte `lex`-Spezifikationen (für den Scanner-Generator `lex` bzw. `flex`) und durch sogenannte `yacc`-Spezifikationen (für den Parser-Generator `yacc` bzw. `bison`) ergänzen. In vielen Fällen kommt man aber völlig ohne solche "Ergänzungen auf niedriger Ebene" aus. Gentle macht es also möglich, einen Compiler auf einem sehr hohen und noch überschaubaren Abstraktionsniveau zu schreiben. Gleichzeitig hat man die Möglichkeit, beliebige Teilprogramme maschinennah "von Hand" zu codieren. Mit dem Gentle-System wurden bereits einige Compiler für die industrielle Anwendung geschrieben (z. B. ein C-Compiler der Firma Siemens und ein TTCN-3-Compiler der Fraunhofer-Gesellschaft).

Die Sprache Gentle hat kaum Ähnlichkeit mit üblichen imperativen Sprachen wie Fortran, C, Java oder Ada etc. Gentle ist vor allem eine *deklarative* Sprache wie Prolog oder SQL. Allerdings wurde in Gentle alles weggelassen, was die Ausführung von Prolog-Programmen manchmal so aufwendig macht (vor allem das sogenannte *deep backtracking*). Trotzdem wurden wichtige Vorteile deklarativer Programmiersprachen bewahrt: Das hohe Abstraktionsniveau, die Übersichtlichkeit, die Nähe zu mathematischen Notationen und zu formalen Beweisen. Auch wenn man nicht vorhat, einen Compiler oder ein Compiler-ähnliches Programm zu schreiben, kann es interessant und nützlich sein, sich mit der Sprache Gentle und dem deklarativen Programmierstil vertraut zu machen. Dieser Programmierstil ist anfangs möglicherweise sehr ungewohnt und befremdend. Aber wenn man ihn einmal beherrscht, dann kann man damit sehr elegant und in verschiedener Hinsicht effizient programmieren.

Im folgenden soll die Sprache Gentle anhand von Beispielprogrammen erläutert werden. Dabei wird folgendes vorausgesetzt:

Dem Leser steht ein PC unter Windows (98/2000/ME/NT/XP) zur Verfügung, auf dem ein Gentle-System zusammen mit dem GNU-C-Compiler `gcc`, den Werkzeugen `flex` und `bison` sowie den hier besprochenen Beispielprogrammen installiert ist.

**Anmerkung:** Gentle kann auch zusammen mit jedem anderen C-Compiler und/oder unter einem Unix-Betriebssystem (z. B. Linux) eingesetzt werden. Dazu müssen nur die hier verwendeten, Windows-spezifischen Skript-Dateien namens `build.bat` (die zum Übersetzen und Binden der einzelnen Beispielprogramme dienen), entsprechend angepasst bzw. durch andere Skriptdateien ersetzt werden.

Der Leser sollte zumindest ansatzweise mit *kontextfreien Grammatiken* vertraut sein und folgende Begriffe kennen: formale Sprache, kontextfreie Grammatik, Regel einer Grammatik (rule), Zwischensymbol (nonterminal symbol), Endsymbol (terminal symbol), Wort (sequence of terminal symbols), Satz-

form (sequence of nonterminal and/or terminal symbols), Ableitung eines Wortes (derivation) und Syntaxbaum eines Wortes (syntax tree).

Die Sprache Gentle wurde von Herrn F. W. Schröer entwickelt. Herr Schröer ist unter der email-Adresse [f.w.schroerer@first.fraunhofer.de](mailto:f.w.schroerer@first.fraunhofer.de) erreichbar. Der Autor dieser Einführung ist unter der email-Adresse [grude@tfh-berlin.de](mailto:grude@tfh-berlin.de) erreichbar.

## 2. hello\_v1.g: Das unvermeidliche Hallo-Programm

Im Verzeichnis `hello_v1` sollten sich mindestens die folgenden drei Dateien befinden:

`hello_v1.g`, `build.bat` und `unbuild.bat`. Hier ein Ausdruck des Gentle-Programms

`hello_v1.g`:

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- hello_v1.g: Einfaches Programm zum Pruefen, ob der Gentle-Compiler
3  -- "funktionsfaehig" ist.
4  -----
5  'root'
6    print("+-----+")
7    print(" | Hello World! | ")
8    print(" | The Gentle-Compiler has been | ")
9    print(" | installed correctly! | ")
10   print("+-----+")
11 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 1 bis 4:** Kommentare können mit zwei Bindestrichen `--` begonnen werden und reichen dann bis zum Ende der aktuellen Zeile.

**Zeile 5 bis 10:** Ein Gentle-Programm besteht vor allem aus *Prädikaten*. Jedes Gentle-Programm muss ein Wurzel-Prädikat (root predicate) enthalten. Dieses Wurzel-Prädikat entspricht dem Hauptprogramm bei anderen Programmiersprachen (z. B. der Funktion namens `main` in einem C-Programm). Nach dem Schlüsselwort `'root'` muss eine Folge von Prädikataufrufen (predicate invocations) stehen. Die Ausführung eines Gentle-Programms besteht darin, dass die Prädikataufrufe in seinem Wurzel-Prädikat (in der Reihenfolge, in der sie dort stehen) ausgeführt werden.

Hier im Beispiel besteht das Wurzel-Prädikat (Zeile 5 bis 10) aus 5 Aufrufen (invocations) des Prädikates `print`. Mit dem `print`-Prädikat kann man Daten beliebiger Typen ausgeben (z. B. Zeichenketten vom Typ `STRING`, Ganzzahlen vom Typ `INT` oder Daten eines beliebigen vom Programmierer vereinbarten Typs, Beispiele dazu später). Das `print`-Prädikat gibt aber immer zur aktuellen Ausgabe (d.h. zum Bildschirm) aus und man kann "die Form der Ausgabe" nicht beeinflussen. Z. B. bewirkt jeder Aufruf von `print`, dass eine neue Zeile begonnen wird. Später werden weitere und flexiblere Ein- und Ausgabe-Prädikate vorgestellt. Das `print`-Prädikat ist eigentlich nur für "Testausgaben für den Programmierer" gedacht und nicht für "ernsthafte Ausgaben für den Benutzer".

Aus dem Gentle-Programm `hello_v1.g` kann man z. B. mit der folgenden Stapeldatei `build.bat` ein lauffähiges Maschinenprogramm namens `hello_v1.exe` erzeugen:

```

1 rem -----
2 rem build.bat fuer hello_v1.g
3 rem -----
4 gentle    hello_v1.g
5 gcc      -o hello_v1.exe hello_v1.c ..\genlib\main0.o ..\genlib\grts.o
6 hello_v1
7 rem -----

```

**Zeile 1 bis 3: Kommentar**

**Zeile 4:** Mit dem `gentle`-Compiler wird das Gentle-Quellprogramm `hello_v1.g` in ein entsprechendes C-Quellprogramm namens `hello_v1.c` umgewandelt. Das gelingt natürlich nur, wenn der `gentle`-Compiler im Quellprogramm `hello_v1.g` keine formalen Fehler findet.

**Zeile 5:** `gcc` (wie `gnu compiler collection`) ist die Vorderseite des GNU-Compiler-Systems. Zu diesem System gehören unter anderem ein C-Compiler (und ein C++-Compiler und ein Ada-Compiler etc.) und ein Binder-Programm (linkage editor). Wenn man das Programm `gcc` aufruft, kann man (im Prinzip beliebig viele) Dateien angeben. Welche dieser Dateien mit welchem Programm des GNU-Compiler-Systems bearbeitet wird, folgt aus der Erweiterung ihres Namens:

- Jede `.c`-Datei wird mit dem C-Compiler in eine entsprechende `.o`-Datei (Objekt-Datei) umgewandelt.
- Alle `.o`-Dateien werden mit dem Binder-Programm zu einem Maschinenprogramm (unter Windows: in eine `.exe`-Datei) zusammengebunden.

Hier in Zeile 5 bezeichnet `genlib` ein Verzeichnis, in dem sich u.a. Dateien namens `main0.o` und `grts.o` befinden. Diese Dateien sind notwendig, um die ausführbare Datei (`hello_v1.exe`) zu erzeugen und werden mit dem Gentle-System mitgeliefert. Im Beispiel wird angenommen, dass das aktuelle Arbeitsverzeichnis (in dem die Gentle-Quelldatei `hello_v1.g` liegt) und das Verzeichnis `genlib` (in dem die Dateien `main.o` und `grts.o` liegen) zusammen (oder "nebeneinander") in einem Mutterverzeichnis `..` liegen (andernfalls muss man die Zeile 5 entsprechend ändern).

**Zeile 6:** Das Programm `hello_v1.exe` wird aufgerufen und gibt (hoffentlich) die erwartete Meldung zum Bildschirm aus.

Wenn man die Stapeldatei `build.bat` im Verzeichnis `hello_v1` ausführen läßt, dann entstehen dadurch zusätzliche Dateien. Mit der folgenden Stapeldatei `unbuild.bat` kann man diese Dateien wieder löschen und das Verzeichnis damit in seinen ursprünglichen Zustand zurückversetzen:

```
1 rem -----
2 rem unbuild.bat fuer hello_v1
3 rem -----
4 del hello_v1.c
5 del hello_v1.exe
6 rem -----
```

In diesem einfachen Hallo-Beispiel müssen nur 2 Dateien gelöscht werden. In anderen Beispielen (z. B. in den Verzeichnissen `ahnen_v1`, `ahnen_v2` etc.) werden durch die Stapeldatei `build.bat` sehr viel mehr Dateien erzeugt. Entsprechend viel größer und nützlicher ist in diesen Verzeichnissen die Datei `unbuild.bat`.

### 3. ahnen\_v1.g: Ein einfacher Parser

Ein *Parser* ist ein Programm, welches eine Zeichenkette einliest und prüft, ob sie zu einer bestimmten formalen Sprache gehört. Diese Sprache wird dabei in aller Regel durch eine kontextfreie Grammatik beschrieben. Falls der Parser feststellt, dass die Zeichenkette nicht zu der Sprache gehört, gibt er Fehlermeldungen aus, die einem beim Korrigieren der Zeichenkette (manchmal mehr und manchmal weniger) helfen können.

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- ahnen_v1.g: Ein Parser fuer die folgende Sprache:
3  -- {mutter, vater, grossmutter, grossvater, urgrossmutter, urgrossvater,
4  --   ... urururgrossmutter, ..., ururururgrossvater, ... }
5  -----
6  'nonterm' ahne          -- ahne ist ein Zwischensymbol
7    'rule' ahne  : ahne1  -- Alle kontextfreien Regeln mit
8    'rule' ahne  : ahne2  -- dem Zwischensymbol ahne auf der
9    'rule' ahne  : ahne3  -- linken Seite.
10
11 'nonterm' ahne1        -- ahne1 ist auch ein Zwischensymbol
12   'rule' ahne1 : "mutter" -- Alle Regeln mit ahne1 auf
13   'rule' ahne1 : "vater"  -- der linken Seite
14
15 'nonterm' ahne2        -- ahne2 ist auch ein Zwischensymbol
16   'rule' ahne2 : "gross" ahne1 -- "grossmutter" und "grossvater"
17
18 'nonterm' ahne3        -- ahne3 ist auch ein Zwischensymbol
19   'rule' ahne3 : "ur" ahne2  -- "urgrossmutter" und "urgrossvater"
20   'rule' ahne3 : "ur" ahne3  -- Ahnen mit 2 oder mehr "ur"s
21 -----
22 'root'
23   ahne
24
25   -- Hauptprogramm des Parsers
26   -- Lies eine Zeichenkette ein und pruefe,
27   -- ob sie sich aus dem Zwischensymbol ahne
28   -- ableiten laesst.
29   print("No parse error!") -- Falls die Zeichenkette ableitbar war.
30  --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 6:** Mit dieser Zeile wird vereinbart, dass *ahne* ein Zwischensymbol (nonterminal symbol) einer kontextfreien Grammatik sein soll. Alle Regeln der Grammatik, die mit dem Zwischensymbol *ahne* beginnen ("mit *ahne* auf der linken Seite") müssen unmittelbar hinter dieser Vereinbarung des Zwischensymbols stehen (siehe Zeile 7 bis 9).

**Zeile 7:** Diese Regel der kontextfreien Grammatik besagt: Aus dem Zwischensymbol *ahne* kann man das Zwischensymbol *ahne1* ableiten. Der Doppelpunkt `:` trennt die linke und rechte Seite der Regel voneinander.

**Zeile 11 bis 13:** Aus dem Zwischensymbol *ahne1* kann man entweder *mutter* oder *vater* ableiten.

**Zeile 15 bis 16:** Aus dem Zwischensymbol *ahne2* kann man entweder *grossmutter* oder *grossvater* ableiten.

**Zeile 18 bis 20:** Aus dem Zwischensymbol *ahne3* kann man unendlich viele Worte ableiten, unter anderem: *urgrossmutter*, *urgrossvater*, *ururgrossmutter*, *ururgrossvater*, *urururgrossmutter* ... etc.

**Anmerkung:** Durch die bisherigen Vereinbarungen (Zeile 6 bis 20) wurden 4 Zwischensymbol-Prädikate (nonterm predicates) namens *ahne*, *ahne1*, *ahne2* und *ahne3* vereinbart. Diese Prädikate ru-

fen sich gegenseitig auf. Z. B. wird in der zweiten Regel für `ahne` (Zeile 8) das Prädikat `ahne2` aufgerufen. Es fehlt bisher aber noch "ein Aufruf des obersten Prädikates, der alles in Gang setzt".

**Zeile 23:** Hinter diesem unscheinbaren Aufruf des Prädikates `ahne` stecken viele Jahre theoretischer Forschung und praktischer Entwicklung auf dem Gebiet der kontextfreien Syntaxanalyse. Der Parser `ahne` liest eine Zeichenkette ein und versucht, sie aus der Grammatik abzuleiten, die in den Zeilen 6 bis 20 beschrieben ist. Wenn ihm das nicht gelingt, gibt er eine Fehlermeldung aus (z. B. `Parse error in line 1 col 5`) und bricht seine Ausführung ab.

**Zeile 26:** Nur wenn die eingelesene Zeichenkette sich aus der Grammatik ableiten läßt, kommt das Programm `ahnen_v1` zu dieser Zeile und gibt die beruhigende Meldung `No parse error!` aus.

Hier eine Stapel-Datei (a batch file), mit der man aus dem oben wiedergegebenen Gentle-Programm `ahnen_v1.g` ein lauffähiges Maschinenprogramm `ahnen_v1.exe` erzeugen kann:

```

1 rem -----
2 rem build.bat fuer ahnen_v1
3 rem -----
4 gentle ahnen_v1.g
5 reflex
6 flex  gen.l
7 bison gen.y
8
9 @copy %genlib%\errmsg.o > nul
10 @copy %genlib%\main1.o > nul
11 @copy %genlib%\grts.o > nul
12
13 gcc -o ahnen_v1.exe ahnen_v1.c lex.yy.c gen.tab.c errmsg.o main1.o grts.o
14
15 ahnen_v1  ururgrossmutter
16 ahnen_v1  in1.ahn
17 ahnen_v1  in2.ahn
18 rem -----

```

Die meisten der folgenden Einzelheiten kann man beim ersten Lesen überspringen. Sie werden später noch genauer behandelt.

**Zeile 4:** Mit dem `gentle`-Compiler wird das Gentle-Programm `ahnen_v1.g` in ein entsprechendes C-Programm namens `ahnen_v1.c` umgewandelt. Außerdem erzeugt der `gentle`-Compiler weitere Dateien (`gen.h`, `gen.lit`, `gen.tkn` und `gen.y`), die Informationen für den Scanner-Generator (andere sagen: Lexer-Generator) `flex` und den Parser-Generator `bison` enthalten.

**Zeile 5:** Das Programm `reflex` erzeugt aus verschiedenen Dateien (`gen.h`, `gen.lit`, `gen.tkn`, `layout.b` und `comments.b`) eine Spezifikation `gen.l` für den Scanner-Generator `flex`.

**Zeile 6:** Der Scanner-Generator `flex` liest die Spezifikation `gen.l` ein und erzeugt eine entsprechende C-Datei namens `lex.yy.c` (das ist der Scanner oder Lexer des Programms `ahnen_v1`).

**Zeile 7:** Der Parser-Generator `bison` liest die Spezifikation `gen.y` ein und erzeugt eine entsprechende C-Datei namens `gen.tab.c` (das ist der Parser des Programms `ahnen_v1`).

**Zeile 9 bis 11:** Die Objektdateien `errmsg.o`, `main1.o` und `grts.o` werden in das aktuelle Verzeichnis (`ahnen_v1`) kopiert. Dabei werden "geschwätzigte Meldungen" zum Gerät `nul` ausgegeben (d.h. weggeworfen). Hier wird angenommen, dass in der Umgebungsvariablen `genlib` der Pfadname des Verzeichnisses `genlib` (mit den Dateien `errmsg.o`, `main1.o` und `grts.o`) steht. Dieser Kopiervorgang ist ein Notbehelf. Eigentlich würde man diese Objektdateien gern als Argumente im `gcc`-

Aufruf (Zeile 13) angeben. Unter DOS ist die Länge einer Kommandozeile aber stark begrenzt (auf etwa 120 Zeichen), und es gibt keine Möglichkeit, ein Kommando über mehrere Zeilen zu verteilen. Pfadnamen wie `%genlib%\errmsg.o` bewirken leicht, dass eine Kommandozeile zu lang wird. Deshalb werden die Objektdateien kopiert und im `gcc`-Aufruf nur noch mit ihrem einfachen Namen `errmsg.o`, `main1.o` etc. angegeben.

**Zeile 13:** Das GNU-Compiler-System soll aus den Dateien `ahnen_v1.c`, `lex.yy.c`, `gen.tab.c`, `errmsg.o`, `main1.o` und `grts.o` ein Maschinenprogramm namens `ahnen_v1.exe` machen.

**Wichtige Zusammenhänge:** Wenn man im Gentle-Programm mindestens *ein* Zwischensymbol-Prädikat (nonterm predicate) vereinbart, wird mit dem Werkzeug `bison` ein Parser erzeugt. Dieser Parser verlangt unter anderem, dass man ihm eine C-Funktion namens `yyerror` zur Verfügung stellt. Er ruft diese Funktion auf, wenn er beim Parsen der Eingabe einen Fehler entdeckt. Eine solche Fehlermeldungs-Funktion wird in der Datei `errmsg.c` vereinbart. Deshalb muss im `gcc`-Aufruf die Objektdatei `errmsg.o` angegeben werden.

Im Parser (der mit dem Parser-Generator `bison` aus den Zwischensymbol-Prädikaten erzeugt wurde) wird unter anderem eine Datei-Variable (file variable) namens `yyin` vereinbart. Der Parser erwartet, dass man darin Informationen über eine geöffnete Datei hinterlegt, ehe man ihn aufruft. Aus dieser Datei liest er dann eine Zeichenkette und analysiert sie (d.h. er sucht nach einer Ableitung für sie). Die C-Datei `main1.c` enthält die Vereinbarung einer Funktion `main`, die geeignete Informationen in die Variable `yyin` des Parsers bringt (und dann das `root`-Prädikat des Gentle-Programms aufruft). Deshalb muss man im `gcc`-Aufruf auch die Objektdatei `main1.o` angeben.

Die Funktion `main` in der Datei `main1.c` ist relativ flexibel organisiert und erlaubt es, die zu parsende Zeichenkette wahlweise auf eine von drei Weisen einzugeben:

1. Indem man die Zeichenkette (z. B. `ururgrossmutter`) in eine Datei schreibt (die z. B. `in1.ahn` heißen könnte) und den Namen dieser Datei beim Aufruf des Programms `ahnen_v1.exe` angibt, z. B. so:

```
> ahnen_v1 in1.ahn
```

2. Indem man die Zeichenkette beim Aufruf des Programms `ahnen_v1.exe` direkt auf der Kommandozeile angibt, z. B. so:

```
> ahnen_v1 ururgrossmutter
```

3. Indem man das Programm `ahnen_v1.exe` ohne Parameter aufruft und dann die zu parsende Zeichenkette über die Standard-Eingabe (Tastatur) eingibt und mit `<Ctrl-Z>` (unter Windows) bzw. mit `<CTRL-D>` (unter Unix) abschließt, z. B. so:

```
> ahnen_v1<Ctrl>
> ururgrossmutter<Ctrl-Z>
```

Die C-Datei `grts.c` (gentle run time system) enthält Routinen, die jedes Gentle-Programm während der Ausführung benötigt. Deshalb muss man im `gcc`-Aufruf auch die Objektdatei `grts.o` angeben.

Zum Vergleich: Das Gentle-Programm `hello_v1.g` enthielt kein Zwischensymbol-Prädikat (nonterm predicate). Also wurde für `hello_v1` auch kein Parser erzeugt. Deshalb musste man im `gcc`-Aufruf für `hello_v1` die Datei `errmsg.o` nicht angeben und anstelle der Datei `main1.o` (die für den Parser eine Datei öffnet und Informationen über diese Datei in der Variablen `yyin` hinterlegt) genügte die einfachere Datei `main0.o`.

**Zeile 15 bis 17:** Das Programm `ahnen_v1.exe` wird dreimal aufgerufen. Beim ersten Mal wird die zu parsende Zeichenkette direkt auf der Kommandozeile angegeben, die anderen beiden Male wird der Name einer Datei angegeben (`IN1.AHN` bzw. `IN2.AHN`). Die Datei `IN2.AHN` enthält eine fehlerhafte Zeichenkette (`ururgrossmutter` mit `ß` anstelle von `ss`) und bewirkt (hoffentlich), dass der Parser die Fehlermeldung `line 2, col 5: illegal token` ausgibt.

### 3.1. Kommentare in den Eingabedateien eines Gentle-Programms:

Die Dateien `IN1.AHN`, `IN2.AHN` etc. sind als Eingabedateien für das Programm `ahnen_v1.exe` gedacht. Die Datei `comments.b` im Verzeichnis `ahnen_v1` bewirkt, dass diese Eingabedateien Kommentare enthalten dürfen, ohne dass der Parser diese als "falsche Urgroßmütter" ablehnt. In der Datei `comments.b` im Verzeichnis `ahnen_v1` wird festgelegt, dass ein Kommentar mit zwei Bindestrichen `--` beginnt und bis zum Ende der aktuellen Zeile reicht. Wenn Sie die Datei `comments.b` aus dem Verzeichnis `ahnen_v1` entfernen (oder umbenennen) und danach das Maschinenprogramm `ahnen_v1.exe` erneut erzeugen lassen (z. B. indem Sie die Stapeldatei `build.bat` aufrufen), werden alle Kommentare vom Parser als Fehler abgelehnt. Wenn Sie Kommentare in anderer Form erlauben wollen (z. B. geklammert durch `/*` und `*/` wie in C-Programmen), dann müssen Sie die Datei `comments.b` entsprechend ändern. Wie das geht, wird im Abschnitt 32 erläutert.

### 3.2. Welche Zeichen sind zwischen Lexemen erlaubt?

Aus der Grammatik im Gentle-Programm `ahnen_v1.g` folgt, dass alle erlaubten Zeichenketten aus den Lexemen `ur`, `gross`, `mutter` und `vater` bestehen müssen. Die Datei `layout.b` regelt, welche Zeichen *zwischen* den einzelnen Lexemen einer Eingabe erlaubt sind. Zeilenwechsel zwischen Lexemen (auch mehrere Zeilenwechsel zwischen zwei Lexemen) sind immer erlaubt. Die mitgelieferte Datei `layout.b` im Verzeichnis `ahnen_v1` erlaubt außerdem Unterstriche `_` zwischen den Lexemen, z. B. so:

```
ur_urur____grossmutter
```

Wenn Sie die Datei `layout.b` löschen (oder umbenennen) sind außer Zeilenwechsel auch Blanks und Tabulatorzeichen erlaubt. Wenn Sie eine leere Datei `layout.b` nennen, sind nur Zeilenwechsel zwischen Lexemen erlaubt.

**Aufgabe 3.1.:** Erzeugen Sie das Programm `ahnen_v1.exe` (z. B. indem Sie die Datei `build.bat` im Verzeichnis `ahnen_v1` aufrufen). Rufen Sie das Programm `ahnen_v1.exe` mehrmals auf, und geben Sie dabei sowohl richtige Zeichenketten (wie z. B. `mutter`, `vater`, `grossmutter`, `ur-grossvater` etc.) als auch falsche Zeichenketten (wie z. B. `abc`, `Mutter`, `VATER` etc.) ein. Probieren Sie alle drei Möglichkeiten aus, die zu parsende Zeichenkette anzugeben (in einer Datei, auf der Kommandozeile oder über die Standard-Eingabe). Versuchen Sie auch, zwischen den Lexemen der Eingabe erlaubte und nicht-erlaubte Zeichen anzugeben.

**Aufgabe 3.2.:** Bringen Sie das Verzeichnis `ahnen_v1` in seinen ursprünglichen Zustand (z. B. indem Sie die Stapeldatei `unbuild.bat` aufrufen). Erzeugen Sie dann erneut das Programm `ahnen_v1.exe`, aber ohne die Stapeldatei `build.bat` aufzurufen. Geben Sie jeden einzelnen Befehl, der in der Stapeldatei steht, von Hand ein, und beobachten Sie genau, welche Datei(en) durch welchen Schritt entstehen.

**Aufgabe 3.3.:** Schreiben Sie in Ihrer Lieblings-Programmiersprache ein Programm, welches genau das gleiche leistet, wie `ahnen_v1.exe`. Beobachten Sie dabei, wie lange Sie für die Entwicklung brauchen. Schätzen Sie: Wie lange braucht ein Gentle-Programmierer, um das Programm `ahnen_v1.g` zu entwickeln?

**Aufgabe 3.4.:** Besorgen Sie sich die kontextfreie Grammatik einer gängigen Programmiersprache (z. B. C, Pascal oder Ada etc.). Schätzen Sie ab: Wie lange würden Sie brauchen, um in Gentle einen Parser für diese Sprache zu schreiben?

#### 4. ahnen\_v2.g: Ein Compiler, der Vorfahren in Zahlen übersetzt

Der folgende Compiler `ahnen_v2.g` ist eine Erweiterung des Parsers `ahnen_v1.g` aus dem vorigen Abschnitt. Der Compiler übersetzt Bezeichnungen für Vorfahren in entsprechende Zahlen, die "ihre Entfernung in Generationen" angibt.

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- ahnen_v2.g: Ein Compiler, der folgende Worte in folgende Zahlen ueber-
3  -- setzt: mutter -> 1, vater -> 1, grossmutter -> 2, grossvater -> 2,
4  -- urgrossmutter -> 3, ..., urururgrossvater -> 5, ...
5  -----
6  'nonterm' ahne(->INT)          -- Das Zwischensymbol ahne hat einen
7                                -- Ausgabeparameter vom Typ INT
8  'rule' ahne(->N)              : ahne1(->N)
9  'rule' ahne(->N)              : ahne2(->N)
10 'rule' ahne(->N)              : ahne3(->N)
11
12 'nonterm' ahne1(->INT)         -- ahne1 hat auch einen INT-Ausgabeparam.
13 'rule' ahne1(->1)             : "mutter"
14 'rule' ahne1(->1)             : "vater"
15
16 'nonterm' ahne2(->INT)         -- ahne2 hat auch einen INT-Ausgabeparam.
17 'rule' ahne2(->N+1)           : "gross" ahne1(->N)
18
19 'nonterm' ahne3(->INT)         -- ahne3 hat auch einen INT-Ausgabeparam.
20 'rule' ahne3(->N+1)           : "ur" ahne2(->N)
21 'rule' ahne3(->N+1)           : "ur" ahne3(->N)
22 -----
23 'root'                          -- Hauptprogramm des Compilers
24 ahne ( ->ERG)                   -- Lies ein Wort ein und berechne ERG
25 print(ERG-> )                  -- Gib ERG aus
26 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 6:** Jedes Prädikat kann mit Parametern versehen werden. In der Vereinbarung des Prädikats wird die Anzahl der Parameter festgelegt und für jeden Parameter ein Typ. Man unterscheidet zwischen Eingabeparametern und Ausgabeparametern. Der Pfeil `->` dient als Trennzeichen zwischen (den Typen von) Eingabeparametern und Ausgabeparametern. Das Zwischensymbol-Prädikat (`nonterm predicate`) `ahne` hat also genau einen Ausgabeparameter vom Typ `INT`. Dieser Typ `INT` ist in Gentle vorvereinbart und entspricht genau dem Typ `int` des verwendeten C-Compilers.

**Anmerkung 1:** In diesem einfachen Beispiel haben zufällig alle Zwischensymbole (`ahne`, `ahne1`, `ahne2` und `ahne3`) je einen Ausgabeparameter vom selben Typ `INT`. In komplizierteren Beispielen haben verschiedene Prädikate verschieden viele Parameter von unterschiedlichen Typen.

**Anmerkung 2:** Zwischensymbol-Prädikate (nonterm predicates) dürfen nur Ausgabeparameter haben. Die folgenden Vereinbarungen sind also nicht erlaubt:

```
1 'nonterm' ahne4(INT -> )      -- ein Eingabeparameter vom Typ INT
2 'nonterm' ahne5(INT -> INT) -- ein Ein- und ein Ausgabeparameter
```

In späteren Beispielen werden weitere Arten von Prädikaten vorgestellt (z. B. Aktions-Prädikate und Bedingungs-Prädikate), die sowohl Eingabe als auch Ausgabeparameter haben dürfen. Das Prädikat `print` ist ein Aktions-Prädikat mit *einem* Eingabeparameter.

**Zeile 8:** In jedem Aufruf eines Prädikates (in every predicate invocation) muss für jeden Parameter ein entsprechendes Muster bzw. ein entsprechender Ausdruck angegeben werden. Auch dabei trennt der Pfeil `->` Eingabeparameter von Ausgabeparametern. `N` ist hier der Name einer Variablen. Diese Variable ist Regel-lokal, d.h. sie hat nichts mit Variablen in anderen Regeln zu tun, die zufällig (oder absichtlich) auch `N` genannt wurden (siehe z. B. die Regel in Zeile 9 oder die Regel in Zeile 10). Die Variable `N` wird dadurch vereinbart, dass man sie im Aufruf eines Prädikats hinschreibt. Ihr Typ folgt eindeutig aus der Vereinbarung des Prädikates (siehe Zeile 6). Der Name einer Variablen muss mit einem großen Buchstaben beginnen. Ansonsten sind in einem Variablennamen (große und kleine) Buchstaben und dezimale Ziffern erlaubt. Hier ein Vorschlag, wie man die Regel in Zeile 8 ins Deutsche übersetzen kann: "Wenn man aus dem Zwischensymbol `ahne` das Zwischensymbol `ahne1` ableitet und wenn dieses Zwischensymbol `ahne1` das Attribut `N` hat, dann hat auch das Zwischensymbol `ahne` das Attribut `N`". Oder etwas einfacher: "Das Zwischensymbol `ahne1` gibt sein Attribut `N` unverändert an das Zwischensymbol `ahne`".

**Zeile 13:** "Wenn man aus dem Zwischensymbol `ahne1` das Wort `mutter` ableitet, dann bekommt das Zwischensymbol `ahne1` eine `1` als Wert seines (einzigen Ausgabe-) Attributs".

**Zeile 17:** "Wenn man aus dem Zwischensymbol `ahne2` das Wort "gross" und das Zwischensymbol `ahne1` ableitet und wenn dabei das Zwischensymbol `ahne1` das Attribut `N` hat, dann soll das Zwischensymbol `ahne2` das Attribut `N+1` bekommen".

**Zeile 20:** "Wenn man aus dem Zwischensymbol `ahne3` das Wort `ur` und das Zwischensymbol `ahne2` mit Attribut `N` ableitet, dann soll `ahne3` das Attribut `N+1` bekommen".

**Zeile 24:** Hier wird das Prädikat `ahne` aufgerufen und sein Attribut berechnet. Der Attributwert bekommt den Namen `ERG` und in nachfolgenden Prädikataufrufen kann er mit diesem Namen bezeichnet werden.

**Zeile 25:** Der Wert namens `ERG` wird mit dem `print`-Prädikat ausgegeben. Da `print` nur (einen) Eingabeparameter hat, darf man den Pfeil `->` hinter `ERG` auch weglassen. Er wurde hier notiert, um die Entsprechungen und Unterschiede zwischen dem Ausgabeparameter von `ahne` und dem Eingabeparameter von `print` zu betonen und augenfällig zu machen.

**Anmerkung:** Normalerweise gehören zu einem Prädikat (predicate) wie `ahne`, `ahne1` etc. *mehrere* Regeln (rules). Z. B. gehören zu `ahne` drei Regeln (siehe Zeile 8 bis 10) und zu `ahne1` gehören zwei Regeln (Zeile 13 bis 14). Das Wurzel-Prädikat (root predicate) bildet eine Ausnahme, weil zu ihm immer genau eine Regel gehört (siehe Zeile 24 bis 25). Die folgende Schreibweise für das Wurzelprädikat ist in Gentle zwar nicht erlaubt, weil sie zu "umständlich und schwerfällig" ist. Sie soll aber die Entsprechungen zwischen "normalen Prädikaten" `ahne` und `ahne1` etc. und dem Wurzelprädikat deutlich machen: Das sog. Wurzel-Prädikat in einem Gentle-Programm entspricht eigentlich einer einzelnen *Regel* eines Prädikats. Das zugehörige „ganze Prädikat“ würde etwa so aussehen:

```

1 'action' root
2   'rule' root: ahne (    ->ERG)
3                   print(ERG->    )

```

Hier ist `ERG` eine Regel-lokale Variable. Sie wird in Zeile 2 vereinbart und mit einem Wert versehen. In Zeile 3 wird sie angewendet (d.h. ihr Wert wird mit dem `print`-Prädikat ausgegeben). Ganz entsprechende ist in der `ahne`-Regel in Zeile 8 die Variable `N` eine Regel-lokale Variable, die auf der rechten Seite der Regel vereinbart und mit einem Wert versehen wird (`ahne1(->N)`). Auf der linken Seite der Regel wird die Variable `N` angewendet (`ahne(->N)`).

**Aufgabe 4.1.:** Erzeugen Sie das Programm `ahnen_v2.exe` (z. B. indem Sie die Stapeldatei `build.bat` im Verzeichnis `ahnen_v2` aufrufen) und testen Sie es mit möglichst vielen (richtigen und falschen) Eingaben.

### 5. ahnen\_v3.g: Vorteile einer Zwischendarstellung

Die Bezeichnungen `grossmutter`, `grossvater`, `urgrossmutter`, `ururgrossvater` etc. haben einige Eigenschaften, die einem Compilerbauer unangenehm sind:

1. Die Lexeme `ur`, `gross`, `mutter` und `vater`, aus denen diese Bezeichnungen bestehen, sind Zeichenketten unterschiedlicher Länge. Das Bearbeiten von Zeichenketten mit einem Computer ist relativ aufwendig, insbesondere wenn die Zeichenketten unterschiedliche Längen haben.
2. Die zweite Generation (`grossmutter` und `grossvater`) wird durch das Wort `gross` gekennzeichnet, alle weiteren Generationen aber durch entsprechend viele `ur`'s. Einfacher und regelmäßiger wären die Bezeichnungen, wenn man statt `grossmutter` (bzw. `grossvater`) einfach `urmutter` (bzw. `urvater`) sagen würde. An der deutschen Sprache kann ein Compilerbauer aber kaum etwas ändern.

Auch bei "richtigen Programmiersprachen" geht man häufig so vor, dass man zuerst das Quellprogramm (die konkrete Syntax) in eine interne Zwischendarstellung (abstrakte Syntax) übersetzt und dann aus dieser Zwischendarstellung das Zielprogramm erzeugt.

Der nächste Compiler `ahnen_v3.g` übersetzt die "unregelmäßigen Bezeichnungen" `mutter`, `grossvater`, `urgrossmutter` etc. in eine regelmäßiger und effizienter zu handhabende interne Darstellung. Diese interne Darstellung wird häufig auch als *Zwischendarstellung* bezeichnet, weil aus ihr später die gewünschte Ausgabe erzeugt wird (siehe dazu das übernächste Beispiel `ahnen_v5.g`).

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- ahnen_v3.g: Ein Compiler, der folgende Worte in folgende Terme
3  -- (d.h. Baeume) uebersetzt:
4  -- mutter           -> mu,
5  -- vater           -> va,
6  -- grossmutter     -> ur(mu),
7  -- grossvater      -> ur(va),
8  -- urgrossmutter   -> ur(ur(mu)),
9  -- ururgrossvater  -> ur(ur(ur(ur(va))))
10 -- ... etc.
11 -----
12 'type' ZAHNE           -- Zwischendarstellungen fuer Ahnen
13   mu                   -- Zwischendarstellung fuer mutter
14   va                   -- fuer vater
15   ur(ZAHNE)           -- fuer eine(n) andere(n) Ahne(n)
16 -----

```

```

17 'nonterm' ahne(->ZAHNE)    -- Das Zwischensymbol ahne hat einen
18                               -- Ausgabeparameter vom Typ ZAHNE
19 'rule' ahne(->Z)           : ahne1(->Z) -- Z ist eine Regel-lokale
20 'rule' ahne(->Z)           : ahne2(->Z) -- Variablen vom
21 'rule' ahne(->Z)           : ahne3(->Z) -- Typ ZAHNE.
22
23 'nonterm' ahne1(->ZAHNE)   -- ahne1 hat auch einen ZAHNE-Ausgabeparam.
24 'rule' ahne1(->mu)         : "mutter"  -- mu ist eine ZAHNE-Konstante.
25 'rule' ahne1(->va)         : "vater"   -- va ist eine ZAHNE-Konstante.
26
27 'nonterm' ahne2(->ZAHNE)   -- ahne2 hat auch einen ZAHNE-Ausgabeparam.
28 'rule' ahne2(->ur(Z))      : "gross"  ahne1(->Z)
29
30 'nonterm' ahne3(->ZAHNE)   -- ahne3 hat auch einen ZAHNE-Ausgabeparam.
31 'rule' ahne3(->ur(Z))      : "ur"     ahne2(->Z)
32 'rule' ahne3(->ur(Z))      : "ur"     ahne3(->Z)
33 -----
34 'root'                       -- Hauptprogramm des Compilers
35 ahne (    ->ERG)             -- Lies ein Wort ein und berechne ERG
36 print(ERG->    )            -- Gib ERG aus
37 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 12:** So kann der Gentle-Programmierer einen neuen Typ vereinbaren. Alle vom Programmierer vereinbarten Typen haben Terme (d.h. Bäume) als Werte. Zum Typ ZAHNE gehören folgende Terme:  $\mu$ ,  $\nu$ ,  $ur(\mu)$ ,  $ur(\nu)$ ,  $ur(ur(\mu))$ ,  $ur(ur(\nu))$ ,  $ur(ur(ur(\mu)))$  ... etc.. Solche Terme können in einem Computer durch sehr effiziente Datenstrukturen ("Bäume mit verzweigten Knoten") realisiert werden.

Typen sind, neben Prädikaten, die wichtigsten Konstrukte, aus denen ein Gentle-Programm besteht. Typen werden immer mithilfe von sogenannten *Funktoren* vereinbart. Im Beispiel sind  $\mu$ ,  $\nu$  und  $ur$  Funktoren, die ein Ergebnis vom Typ ZAHNE liefern. Die Funktoren  $\mu$  und  $\nu$  sind nullstellig (d.h. sie haben keine Parameter) und  $ur$  hat *einen* Parameter vom Typ ZAHNE. Im Prinzip kann ein Funktor beliebig viele Parameter von beliebigen Typen haben.

**Zeile 24:** "Wenn man aus dem Zwischensymbol  $ahne1$  das Wort *mutter* ableitet, dann soll  $ahne1$  den Term  $\mu$  als Wert seines Attributs bekommen"

**Zeile 32:** "Wenn man aus dem Zwischensymbol  $ahne3$  das Wort  $ur$  und das Zwischensymbol  $ahne3$  mit dem Attributwert  $Z$  ableitet, dann soll das Zwischensymbol  $ahne3$  (das erste, aus dem man abgeleitet hat) den Attributwert  $ur(Z)$  bekommen".

**Zeile 36:** Mit dem `print`-Prädikat kann man Werte beliebiger Typen ausgeben, also auch solche vom Typ ZAHNE. Das spricht für das Prädikat `print`. Der Wert wird in einem Standardformat ausgegeben, an dem man nichts ändern kann und an das man sich zuerst ein bißchen gewöhnen muss. Das spricht gegen `print`. Für "ernsthafte" Ein- und Ausgaben werden später weitere Prädikate vorgestellt.

**Aufgabe 5.1.:** Erzeugen Sie das Programm `ahnen_v3.exe` (z. B. indem Sie die Stapeldatei `build.bat` im Verzeichnis `ahnen_v3` aufrufen) und testen Sie es mit möglichst vielen (richtigen und falschen) Eingaben.

## 6. ahnen\_v4.g: Direkte Übersetzung

Der folgende Compiler `ahnen_v4.g` übersetzt bestimmte deutsche Bezeichnungen für Ahnen in entsprechende englische Bezeichnungen, z. B. `urgrossmutter` in `greatgrandmother`. Dieser Compiler ist in verschiedener Hinsicht kein gutes Vorbild: Er arbeitet zu viel mit Zeichenketten vom Typ `STRING` (was in aller Regel uneffizient ist) und trennt die Teilaufgaben eines Compilers (z. B. das Parsen des Quellprogramms und das Erzeugen des Zielprogramms) nicht sauber voneinander. Andererseits ist er relativ kurz und (hoffentlich) leicht verständlich. Außerdem soll er zeigen, wie man Prädikate, die man in C programmiert hat, in einem Gentle-Programm anwenden kann (hier das Prädikat `conc2`).

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- ahnen_v4.g: Ein Compiler, der folgende deutsche Worte in folgende
3  -- englische Worte uebersetzt und die Uebersetzung ausgibt:
4  -- mutter -> mother, vater -> father, grossmutter -> grandmother,
5  -- grossvater -> grandfather, urgrossmutter -> greatgrandmother,
6  -- urgrossvater -> greatgrandfather, ururgrossmutter -> greatgreatgrand-
7  -- mother, ..., urururgrossvater -> greatgreatgreatgrandfather, ...
8  -----
9  'nonterm' ahne(->STRING)  -- Das Zwischensymbol ahne hat einen
10                                     -- Ausgabeparameter vom Typ STRING
11  'rule' ahne(->S)         : ahne1(->S) -- S ist eine Regel-lokale
12  'rule' ahne(->S)         : ahne2(->S) -- Variablen vom
13  'rule' ahne(->S)         : ahne3(->S) -- Typ STRING.
14
15  'nonterm' ahne1(->STRING) -- ahne1 hat auch einen STRING-Ausgabeparam.
16  'rule' ahne1(->"mother"): "mutter"
17  'rule' ahne1(->"father"): "vater"
18
19  'nonterm' ahne2(->STRING) -- ahne2 hat auch einen STRING-Ausgabeparam.
20  'rule' ahne2(->Z0)       : "gross" ahne1(->Z1)
21                                     conc2("grand", Z1 -> Z0)
22
23  'nonterm' ahne3(->STRING) -- ahne3 hat auch einen STRING-Ausgabeparam.
24  'rule' ahne3(->Z0)       : "ur" ahne2(->Z1)
25                                     conc2("great", Z1 -> Z0)
26  'rule' ahne3(->Z0)       : "ur" ahne3(->Z1)
27                                     conc2("great", Z1 -> Z0)
28  -----
29  'action' conc2(Arg1: STRING, Arg2: STRING -> Erg: STRING)
30  -- Erg ist die Konkatenation von Arg1 und Arg2. Das Aktions-Praedikat
31  -- conc2 wurde als C-Funktion in der Datei str_hand.c realisiert.
32  -----
33  'root'                               -- Hauptprogramm des Compilers
34  ahne ( ->ERG)                         -- Lies ein Wort ein und berechne ERG
35  print(ERG-> )                         -- Gib ERG (vom Typ STRING) aus
36  --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 9:** Diesmal hat das Zwischensymbol `ahne` einen Ausgabeparameter vom Typ `STRING`. Entsprechendes gilt auch für die anderen Zwischensymbole `ahne1`, `ahne2` und `ahne3` (siehe Zeile 15, 19 und 23). Der Typ `STRING` ist in Gentle (ähnlich wie der Typ `INT`) vorvereinbart.

**Zeile 29:** Das Aktions-Prädikat `conc2` hat zwei Eingabeparameter vom Typ `STRING` und einen Ausgabeparameter vom selben Typ. Das Prädikat `conc2` trifft auf drei Zeichenketten `S1`, `S2` und `S3` zu (d.h. die Aussage `conc2(S1, S2 -> S3)` ist wahr), wenn `S3` die Konkatenation von `S1` und `S2` ist. Wenn man dem Prädikat `conc2` zwei aktuelle Eingabeparameter vorgibt, dann berechnet es daraus

den Wert seines Ausgabeparameters. In der Vereinbarung des Aktions-Prädikates `conc2` wurde von der (allgemeinen) Möglichkeit Gebrauch gemacht, den Parametern Namen zu geben (`Arg1`, `Arg2` und `Erg`), um damit das Lesen der Vereinbarung zu erleichtern.

**Zeile 20 bis 21:** "Wenn man aus dem Zwischensymbol `ahne2` das Wort `gross` und das Zwischensymbol `ahne1` ableitet und `ahne1` das `STRING`-Attribut `Z1` hat, und wenn `Z0` die Konkatenation von `grand` und `Z1` ist, dann soll das Zwischensymbol `ahne2` das `STRING`-Attribut `Z0` bekommen".

**Zeile 24 bis 25:** Entsprechend.

**Zeile 26 bis 27:** Entsprechend.

Das Aktions-Prädikat `conc2` wurde im Gentle-Programm nur bekannt gemacht (in Zeile 29), um dem Gentle-Compiler eine genaue Typüberprüfung aller Prädikataufrufe zu ermöglichen. Programmiert wurde das Prädikat in der Sprache C in der Datei `str_hand.c`. Diese Datei (bzw. die entsprechende Objektdatei `str_hand.o`) muss im `gcc`-Kommando für `ahnen_v4` angegeben werden (siehe Datei `build.bat` im Verzeichnis `ahnen_v4`).

**Anmerkung:** Die Datei `str_hand.c` enthält außer `conc2` noch weitere Funktionen namens `conc3`, `conc4` ... etc., mit denen man 3, 4, ... und sogar 9 Zeichenketten auf einmal konkatenieren kann. Will man eine dieser C-Funktionen als Prädikat verwenden, muss man im gentle-Programm eine der folgenden Vereinbarungen angeben:

```

1 'action' conc2(Arg1: STRING, Arg2: STRING           -> Erg: STRING)
2 'action' conc3(Arg1: STRING, Arg2: STRING, Arg3: STRING -> Erg: STRING)
3 'action' conc4(Arg1: STRING, Arg2: STRING,
4             Arg3: STRING, Arg4: STRING           -> Erg: STRING)
5 ...
6 'action' conc9(Arg1: STRING, Arg2: STRING, Arg3: STRING,
7             Arg4: STRING, Arg5: STRING, Arg6: STRING,
8             Arg7: STRING, Arg8: STRING, Arg9: STRING -> Erg: STRING)

```

**Aufgabe 6.1.:** Erzeugen Sie das Programm `ahnen_v3.exe` (z. B. indem Sie die Stapeldatei `build.bat` im Verzeichnis `ahnen_v3` aufrufen) und testen Sie es mit möglichst vielen (richtigen und falschen) Eingaben.

## 7. ahnen\_v5.g: Übersetzung mit Zwischendarstellung

Der Compiler `ahnen_v5.g` übersetzt bestimmte deutsche Bezeichnungen für Ahnen in eine effizient handhabbare Zwischendarstellung und diese Zwischendarstellung dann in entsprechende englische Bezeichnungen. Nach diesem Schema funktionieren auch viele wesentlich größere und komplexere Compiler.

```

1 --*****1*****2*****3*****4*****5*****6*****7*****
2 -- ahnen_v5.g: Ein Compiler, der folgende deutsche Worte in folgende
3 -- englische Worte uebersetzt und die Uebersetzung ausgibt:
4 -- mutter -> mother, vater -> father, grossmutter -> grandmother,
5 -- grossvater -> grandfather, urgrossmutter -> greatgrandmother,
6 -- urgrossvater -> greatgrandfather,
7 -- ururgrossmutter -> greatgreatgrandmother, ...
8 -----
9 'type' ZAHNE           -- Zwischendarstellungen fuer Ahnen
10 mu                   -- Zwischendarstellung fuer mutter
11 va                   -- fuer vater
12 ur(ZAHNE)           -- fuer eine(n) andere(n) Ahne(n)

```

```

13 -----
14 'nonterm' ahne(->ZAHNE) -- Das Zwischensymbol ahne hat einen
15 -- Ausgabeparameter vom Typ ZAHNE
16 'rule' ahne(->Z) : ahnel(->Z) -- Z ist eine Regel-lokale
17 'rule' ahne(->Z) : ahne2(->Z) -- Variablen vom
18 'rule' ahne(->Z) : ahne3(->Z) -- Typ ZAHNE.
19
20 'nonterm' ahnel(->ZAHNE) -- ahnel hat auch einen ZAHNE-Ausgabeparam.
21 'rule' ahnel(->mu) : "mutter" -- mu ist eine ZAHNE-Konstante.
22 'rule' ahnel(->va) : "vater" -- va ist eine ZAHNE-Konstante.
23
24 'nonterm' ahne2(->ZAHNE) -- ahne2 hat auch einen ZAHNE-Ausgabeparam.
25 'rule' ahne2(->ur(Z)) : "gross" ahnel(->Z)
26
27 'nonterm' ahne3(->ZAHNE) -- ahne3 hat auch einen ZAHNE-Ausgabeparam.
28 'rule' ahne3(->ur(Z)) : "ur" ahne2(->Z)
29 'rule' ahne3(->ur(Z)) : "ur" ahne3(->Z)
30 -----
31 'action' aus(ZAHNE->) -- Das Praedikat aus hat einen Eingabeparameter
32 -- vom Typ ZAHNE
33 'rule' aus(ur(ur(ZA))) : PutS("great") aus(ur(ZA))
34 'rule' aus(ur(ZA)) : PutS("grand") aus(ZA)
35 'rule' aus(mu) : PutS("mother") Nl -- mit newline abschliessen
36 'rule' aus(va) : PutS("father") Nl -- mit newline abschliessen
37 -----
38 'action' PutS(String->) -- Eine C-Funktion in der Datei text_io.c
39 'action' Nl -- Gib ein newline-Zeichen aus (C-Funktion)
40 -----
41 'root' -- Hauptprogramm des Compilers
42 ahne(->ERG) -- Lies ein Wort ein und berechne ERG
43 aus(ERG->) -- Gib ERG aus
44 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 14 bis 29:** Die Übersetzung der deutschen "Quellbezeichnung" in einen Term vom Typ ZAHNE erfolgt ganz genauso wie bei dem Compiler ahnen\_v3.g.

**Zeile 31:** Das Aktions-Prädikat `aus` bekommt einen Term vom Typ ZAHNE als Eingabeparameter, übersetzt ihn in eine entsprechende englische Bezeichnung und gibt diese zur aktuellen Ausgabe (zum Bildschirm) aus. Da das Prädikat `aus` nur (einen) Eingabeparameter hat, hätte man den Pfeil `->` auch weglassen können.

**Zeile 33:** Wenn der zu übersetzende ZAHNE-Term noch mindestens zwei `urs` hat, wird (für das erste `ur`) das Wort `great` ausgegeben und dann der Restterm (ohne das erste `ur`) mit `aus` bearbeitet.

**Zeile 34:** Diese Regel wird nur dann "probiert", wenn die vorige Regel nicht anwendbar ist. Das letzte `ur` wird mit `grand` übersetzt (und nicht mit `great` wie die anderen `urs`). Die Reihenfolge, in der man die Regeln für ein Prädikat angibt, kann also von ausschlaggebender Bedeutung sein.

**Zeile 38 bis 39:** Die Aktions-Prädikate `PutS` (wie `put string`) und `Nl` (wie `new line`, das Zeichen nach dem großen `N` ist also ein kleines `Ell`, keine `Eins`!) wurden in C programmiert. Die Datei `text_io.c` (bzw. die entsprechende Objektdatei `text_io.o`) muss im `gcc`-Kommando für `ahnen_v5` angegeben werden (siehe die Datei `build.bat` im Verzeichnis `ahnen_v5`).

**Aufgabe 7.1.:** Erzeugen Sie das Programm `ahnen_v5.exe` (z. B. indem Sie die Stapeldatei `build.bat` im Verzeichnis `ahnen_v5` aufrufen) und testen Sie es mit möglichst vielen (richtigen und falschen) Eingaben.

**Aufgabe 7.2.:** Schreiben Sie einen Compiler namens `ahnen_v7.g`, der die englischen Bezeichnungen `mother`, `father`, `grandmother`, `grandfather`, `greatgrandmother`, `greatgrandfather`, `greatgreatgrandmother`, ... etc. in die entsprechenden deutschen Bezeichnungen `mutter`, `vater`, `grossmutter`, `grossvater`, `urgrossmutter`, `urgrossvater`, `ururgrossmutter` ... etc. übersetzt. Der Compiler `ahnen_v7.g` soll also die Umkehrung des Compilers `ahnen_v5.g` sein.

## 8. ahnen\_v6.g: Noch ein Ahnen-Compiler

Angenommen, es gibt nur zwei Menschen, Maria und Johann (bzw. Mary and John). Dann kann man jeden Vorfahren dieser beiden Menschen durch einen Ausdruck der folgenden Art bezeichnen:

Die Mutter von Maria  
 Der Vater von Johann  
 Der Vater der Mutter des Vaters des Vaters von Maria  
 Die Mutter der Mutter der Mutter des Vaters des Vaters von Johann  
 ... etc.

Diese Ausdrücke sollen ins Englische übersetzt und dabei (durch ein bißchen Abstraktion) etwas vereinfacht werden, etwa so:

### Deutsch

Der Vater von Maria  
 Der Vater des Vaters von Johann  
 Der Vater der Mutter von Johann  
 Die Mutter des Vaters des Vaters von Maria  
 Die Mutter der Mutter der Mutter von Maria

### Englisch

The father of Mary -- The !  
 A grandfather of John -- A !  
 A grandfather of John  
 A greatgrandmother of Mary  
 A greatgrandmother of Mary

**Aufgabe 8.1.:** Schreiben Sie einen Compiler namens `ahnen_v6.g`, der die oben beschriebenen deutschen Ausdrücke in die entsprechenden (aber etwas allgemeineren) englischen Ausdrücke übersetzt. Eine wichtige Teilaufgabe besteht darin, eine "bequem zu handhabende" Zwischendarstellung zu entwerfen. Welche Informationen brauchen Sie über einen Vorfahren von Maria bzw. Johann, um daraus leicht "die richtige englische Bezeichnung" zu erzeugen? Wie kann man diese Informationen als Terme eines Typs darstellen? Hier eine Lösung für Aufgabe 8.1.:

```

1 /* -----
2 Datei ahnen_v6.g: Ein Compiler, der bestimmte deutsche Bezeichnungen
3 fuer Ahnen ("Quellbezeichnungen") in etwas allgemeinere englische
4 Bezeichnungen ("Zielbezeichnungen") uebersetzt und die Ziel-Bezeichnungen
5 ausgibt, z.B. so:
6 +-----+-----+-----+
7 | Quell-Bezeichnung | Zwischen-Darstellung | Ziel-Bezeichnung |
8 +-----+-----+-----+
9 | Die Mutter von Maria | za(w, 1, ma) | The mother of Mary |
10 | Der Vater von Maria | za(m, 1, ma) | The father of Mary |
11 | Die Mutter des Vaters | za(w, 2, jo) | A grandmother |
12 | von Johann | | of John |
13 | Die Mutter des Vaters | za(w, 3, jo) | A greatgrandmother of |
14 | des Vaters von Johann | | John |
15 +-----+-----+-----+
16 ----- */
17 'type' GESCHLECHT -- Geschlecht einer/eines Ahnen
18 w -- weiblich
19 m -- maennlich
20 -----
21 'type' PERSON
22 ma -- Maria/Mary
23 jo -- Johann/John
24 -----
25 'type' Z_AHNE -- Zwischendarstellung einer/eines Ahnen
26 za(Sex:GESCHLECHT, Entfernung:INT, Von: PERSON)
27 -----

```

```

28 -- Grundidee der Grammatik: Jedes Wort der Quellsprache besteht aus drei
29 -- Teilen:
30 -- vorn ("Die Mutter" oder "Der Vater")
31 -- mitte (z.B. "des Vaters der Mutter" oder leer)
32 -- hinten ("von Maria" oder "von Johann")
33 --
34 -- Variablennamen (siehe auch die Definition des Typs Z_AHNE):
35 -- S (wie Sex, Typ GESCHLECHT)
36 -- E (wie Entfernung, Typ INT)
37 -- V (wie Von, Typ PERSON)
38
39 'nonterm' q_ahne(-> Z_AHNE) -- Startsymbol
40 'rule' q_ahne(-> za(S, E, V)): vorn(-> S) mitte(-> E) hinten(-> V)
41
42 'nonterm' vorn(-> GESCHLECHT)
43 'rule' vorn(-> w): "Die" "Mutter" -- besser so als "Die Mutter"!
44 'rule' vorn(-> m): "Der" "Vater" -- besser so als "Der Vater" !
45
46 'nonterm' mitte(-> INT)
47 'rule' mitte(-> 1): -- mitte geht nach leer
48 'rule' mitte(-> E+1): "der" "Mutter" mitte(-> E)
49 'rule' mitte(-> E+1): "des" "Vaters" mitte(-> E)
50
51 'nonterm' hinten(-> PERSON)
52 'rule' hinten(-> ma): "von" "Maria"
53 'rule' hinten(-> jo): "von" "Johann"
54 -----
55 'action' aus(Z_AHNE ->)
56 'rule' aus(za(S, E, V)): aus1(E) aus2(E) aus3(S) aus4(V)
57
58 'action' aus1(INT ->)
59 'rule' aus1(E ->) : eq(E, 1) PutS("The ")
60 'rule' aus1(E ->) : PutS("A " )
61
62 'action' aus2(INT ->)
63 'rule' aus2(E ->) : gt(E, 2) PutS("great") aus2(E-1)
64 'rule' aus2(E ->) : eq(E, 2) PutS("grand") aus2(E-1)
65 'rule' aus2(E ->) :
66
67 'action' aus3(GESCHLECHT ->)
68 'rule' aus3(w ->) : PutS("mother ")
69 'rule' aus3(m ->) : PutS("father ")
70
71 'action' aus4(PERSON ->)
72 'rule' aus4(ma ->) : PutS("of Mary") Nl
73 'rule' aus4(jo ->) : PutS("of John") Nl
74 -----
75 -- Zwei C-Funktionen aus der Datei Text_IO.c:
76 'action' PutS(STRING ->) -- Gibt einen String zur Standardausgabe aus
77 'action' Nl -- Gibt ein newline-Zeichen zur Std-Ausgabe aus
78 -----
79 'root' -- Das Wurzelpraedikat des Compilers ("das Hauptprogramm")
80 q_ahne( -> ZA) -- q_ahne ist das Startsymbol der Grammatik
81 -- print (ZA -> ) -- Zum Pruefen der Zwischendarstellung ZA
82 aus (ZA -> ) -- Ausgabe der Uebersetzung ins Englische
83 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 17 bis 19:** Zum Typ `GESCHLECHT` gehören nur die zwei nullstelligen Funktoren `w` und `m`. Damit entspricht dieser Typ einem Aufzählungstyp (enumeration type) in Pascal oder Ada. Dieser Aufzählungstyp hat zwei Werte.

**Zeile 21 bis 23:** Auch der Typ `PERSON` entspricht einem Aufzählungstyp (mit zwei Werten).

**Zeile 25 bis 26:** Zum Typ `Z_AHNE` gehören sehr viele Werte (4 mal soviel Werte wie Ganzzahlen zum Typ `INT` gehören). Der Typ `Z_AHNE` hat Ähnlichkeit mit einem Verbundtyp (record type) in Pascal oder Ada oder einem Struktur-Typ (struct type) in C. Jeder Wert des Typs `Z_AHNE` beginnt mit dem Funktor `za` und dieser Funktor hat drei Argumente. Diesen Argumenten wurden Namen gegeben (`Von`, `Sex` und `Entfernung`), um die Lesbarkeit der Vereinbarung zu erhöhen. Man hätte diese Namen auch weglassen können. Die Typen der drei Argumente (`PERSON`, `GESCHLECHT` und `INT`) darf man dagegen nicht weglassen.

**Zeile 40:** Wenn man sich alles, was auf dieser Zeile in runden Klammern steht, erstmal wegdenkt, bleibt eine Grammatikregel übrig die besagt, dass man aus (dem Startsymbol) `q_ahne` eine Folge `vorn mitte hinten` ableiten darf. Mit anderen Worten: Jeder Satz der durch die Grammatik beschriebenen Sprache besteht aus drei Teilen, die hier mit den Zwischensymbolen `vorn`, `mitte` und `hinten` bezeichnet wurden. Dies ist die wichtigste Regel der Grammatik denn sie zerlegt das Gesamtproblem (die Sätze der Quellsprache zu beschreiben) in drei viel einfachere Probleme (die `vorn`-Teile, die `mitte`-Teile und die `hinten`-Teile eines Satzes zu beschreiben).

Jedes der Zwischensymbole `q_ahne`, `vorn`, `mitte` und `hinten` hat genau ein Ausgabeattribut (wie in den Zeilen 39, 42, 46 und 51 festgelegt wird). Diese Ausgabeattribute gehören allerdings zu unterschiedlichen Typen (`Z_AHNE` bzw. `GESCHLECHT` bzw. `INT` bzw. `PERSON`). Die in runde Klammern eingefassten Angaben in Zeile 40 legen fest, wie das Attribut des Zwischensymbols `q_ahne` (nämlich `za(S, E, V)`) aus den Attributen der Zwischensymbole `vorn`, `mitte` und `hinten` berechnet werden soll.

**Zeile 43:** Wenn man in dieser Zeile "Die Mutter" schreibt, dann muss später beim Einlesen und Parsen eines Satzes zwischen `Die` und `Mutter` genau ein Blank stehen. Wenn man statt dessen "Die" "Mutter" schreibt, dürfen zwischen `Die` und `Mutter` beliebig viele Trennzeichen (Blanks, Tab-Zeichen und Zeilenwechsel) stehen (es muss aber mindestens ein Trennzeichen dort stehen).

**Zeile 47:** Diese Grammatikregel drückt aus, dass man aus dem Zwischensymbol `mitte` auch nichts ableiten darf (die rechte Seite dieser Regel ist einfach leer gelassen). Beispiele für Sätze "ohne `mitte`" sind `Die Mutter von Maria` und `Der Vater von Johann` etc. Diese Sätze bezeichnen Ahnen, die 1 Generation von Maria bzw. Johann entfernt sind.

**Zeile 48:** Aus dem Zwischensymbol `mitte` auf der rechten Seite dieser Regel kann man entweder ein weiteres Exemplar von "des Vaters" oder "der Mutter" ableiten, oder nichts (siehe die Erläuterung zur Zeile 47).

**Zeile 55:** Das Aktions-Prädikat `aus` übersetzt einen Wert des Gentle-Typs `Z_AHNE` in eine entsprechende englische Wortfolge und gibt sie `aus`. Jedes der übrigen Aktions-Prädikate (`aus1` bis `aus4`) führt einen Teil dieser Übersetzung und Ausgabe durch und wird von `aus` aufgerufen.

**Zeile 58:** Das Aktionsprädikat `aus1` gibt entweder `The` oder `A` aus.

**Zeile 59:** Der Prädikataufruf `eq(E, 1)` glückt nur dann (und gibt `The` aus), wenn `E` gleich `1` ist. Nur wenn er nicht glückt, wird die nächste Regel probiert. `eq` (equals) ist ein in Gentle vordefiniertes Prädikat. weitere ähnliche Prädikate: `lt` (less than), `le` (less or equals), `gt` (greater than), `ge` (greater or equals) und `ne` (not equals).

**Zeile 60:** Diese Regel glückt immer und gibt A aus. Die Reihenfolge der beiden Regeln in den Zeilen 59 und 60 ist also wichtig.

**Zeile 63:** Der Prädikataufruf `gt(E, 2)` glückt genau dann, wenn E größer oder gleich 2 ist. Auch `gt` (greater than) ist ein in Gentle vordefiniertes Prädikat.

## 9. Typen

In Gentle gibt es drei vordefinierte Typen:

**INT**       Ganzzahlen, z. B. 17 oder -123. Dieser Typ entspricht genau dem Typ `int` des verwendeten C-Compilers.

**STRING**   Zeichenketten, z. B. "Hallo!".

**POS**       Positionen in einem Quellprogramm, z. B. `file 2, line 17, col 8`.

Zur Bearbeitung von INT-Werten stehen auch in Gentle die üblichen zweistelligen Funktionen `+`, `-`, `*` und `/` und die einstelligen Funktionen ("Vorzeichen") `+` und `-` zur Verfügung. Für Werte der Typen **STRING** und **POS** gibt es keine vordefinierten Funktionen. Der C-Modul `text_io.c` enthält Prädikate, mit denen man Werte der Typen **INT**, **STRING** bzw. **POS** ein- und ausgeben kann (`PutI`, `PutS`, `PutP`, `GetI`, `GetS`, `GetP` etc.).

Die Gentle-Syntax zum Vereinbaren neuer Typen ist sehr schlicht, aber ausdrucksstark. Man kann damit unter anderem Typen vereinbaren, die große Ähnlichkeit mit *Aufzählungstypen* (enumeration types), mit *Verbundtypen* (record types, struct types) bzw. mit *varianten Verbundtypen* (variant record types, union types) in Sprachen wie Pascal, Ada und C haben. Darüber hinaus kann man in Gentle sehr einfach *rekursive Typen* vereinbaren, die sich in vielen anderen Sprachen nur mithilfe von (fehlerträchtigen) Zeigern realisieren lassen. Hier ein paar Beispiele für Typvereinbarungen in Gentle. Die Bezeichnungen *Aufzählungstyp*, *Verbundtyp* etc. sind dabei nur als Kommentare zu verstehen und haben in Gentle keine "offizielle Bedeutung".

**Beispiel 9.1.:** Ein Aufzählungstyp:

```
'type' TAG mo di mi do fr sa so
```

Zu diesem Typ **TAG** gehören genau 7 Werte, die man mit den Funktoren `mo`, `di`, ..., `so` bezeichnen kann. Funktoren müssen mit einem kleinen Buchstaben beginnen (damit sie sich leicht von Variablen-Namen unterscheiden lassen, die mit einem großen Buchstaben beginnen müssen). Typennamen wie **TAG** dürfen beliebig mit einem großen oder kleinen Buchstaben beginnen. In Gentle werden große und kleine Buchstaben unterschieden. Z. B. sind **TAG**, **Tag** und **tag** drei verschiedene Bezeichner.

Bezeichner, die als Funktoren zulässig sind:       `mo`, `montag`, `mONTAG`, `mOnTaG`, `m17`, `mx_5`

Bezeichner, die als Funktoren nicht zulässig sind: `Mo`, `MONTAG`, `Montag`, `M17`, `Mx_5`

Bezeichner, die als Typnamen zulässig sind:       `TAG`, `Tag`, `tag`, `tAG`, `WochenTag`, `T17`, `T_35`

**Beispiel 9.2.:** Ein Verbundtyp (record type, struct type):

```
1 'type' TERMIN
2   ter(WochenTag: TAG, Stunde: INT)
```

Jeder Wert des Typs **TERMIN** ist ein Term, der mit dem Funktor `ter` beginnt und ansonsten zwei Komponenten umfaßt, eine vom Typ **TAG** und eine vom Typ **INT**. Die Bezeichner `WochenTag` und `Stunde` für diese Komponenten haben den Charakter von Kommentaren: In Anwendungen des Typs

TERMIN tauchen sie nicht wieder auf und in der Vereinbarung des Typs kann man sie auch weglassen, z. B. so:

```
1 'type' TERMIN
2   ter(TAG, INT)
```

Ist `Anzahl(INT)` die Anzahl der Werte vom Typ `INT`, dann gibt es genau  $7 * \text{Anzahl}(INT)$  viele Werte vom Typ `TERMIN`. Bei vielen C-Compilern gehören etwa 4 Milliarden Werte zum Typ `int`, und somit etwa ( $7 * 4$  gleich) 28 Milliarden Werte zum Typ `TERMIN`.

**Beispiel 9.3.:** Ein varianter Verbundtyp (variant record type, union type):

```
1 'type' KFZ_INFO
2   personen_wagen(Kennzeichen: STRING, Sitze: INT)
3   last_wagen    (Kennzeichen: STRING, Gewicht: INT, Achsen: INT)
```

Ein Wert des Typs `KFZ_INFO` ist ein Term, der entweder mit dem Funktor `personen_wagen` beginnt und zwei Komponenten umfaßt (Kennzeichen und Sitze) oder mit dem Funktor `last_wagen` beginnt und drei Komponenten umfaßt (Kennzeichen, Gewicht und Achsen). In Pascal oder Ada würde man sagen: `KFZ_INFO` ist ein Verbundtyp mit zwei Varianten. In C hieße es entsprechend: Der Typ `KFZ_INFO` ist die Vereinigung (union) von zwei Verbundtypen (struct types).

Die folgende Vereinbarung des Typs `KFZ_INFO` (ohne die Komponenten-Bezeichner `Kennzeichen` und `Sitze`) ist für den Gentle-Compiler völlig äquivalent zur obigen Vereinbarung, für einen menschlichen Leser möglicherweise aber schwerer zu verstehen:

```
1 'type' KFZ_INFO
2   personen_wagen(STRING, INT)
3   last_wagen    (STRING, INT, INT)
```

Ein Verbundtyp `VT` ist *rekursiv*, wenn er Verbunde umfaßt, die eine Komponente vom Typ `VT` (oder mehrere Komponenten vom Typ `VT`) besitzen. Rekursive (Verbund-) Typen kann man in Pascal, Ada und C nur mithilfe von *Zeigern* realisieren. Hier ein paar Vereinbarungen von rekursiven Typen in Gentle:

**Beispiel 9.4.:** Listen, die möglicherweise leer sind

```
1 'type' LISTE0
2   leer
3   liste(K1: STRING, K2: INT, Rest: LISTE0)
```

Ein Wert vom Typ `LISTE0` besteht entweder nur aus dem Funktor `leer`, oder er beginnt mit dem Funktor `liste` und umfaßt drei Komponenten. Die Komponenten `K1` und `K2` sind die "Nutzdaten eines Listenelementes", die dritte Komponente `Rest` ist (rekursiv!) ein Wert vom Typ `LISTE0`. Hier ein paar Werte (d.h. Terme oder Listen) vom Typ `LISTE0`:

```
leer
liste("Hallo", 123, leer)
liste("Wie geht es?", 2, liste("Danke, es geht so!", 1, leer))
liste("Hallo", 3, liste("Wie geht es?", 2, liste("Danke!", 1, leer)))
```

Die Funktoren `leer` und `liste` haben in Gentle keine "vordefinierte Bedeutung". Sie erhalten ihre Bedeutungen nur dadurch, dass sie in der Vereinbarung des Typs `LISTE0` eingeführt werden. Anstelle von `leer` und `liste` hätte man auch `otto` und `emil` als Funktoren verwenden können. Möglicherweise würde dadurch einem menschlichen Leser das Verstehen der Typvereinbarung eher erschwert.

**Beispiel 9.5:** Listen, die mindestens ein Element enthalten

```

1 'type' LISTE1
2   letzt(K1: STRING, K2: INT)
3   liste(K1: STRING, K2: INT, Rest: LISTE1)

```

Das letzte (und möglicherweise einzige) Element einer Liste vom Typ `LISTE1` beginnt mit dem Funktor `letz`t und umfaßt nur zwei (Nutzdaten-) Komponenten `K1` und `K2`. Eine Liste mit mehr als einem Element beginnt auch hier (wie bei einer Liste vom Typ `LISTE0`) mit dem Funktor `liste` und umfaßt drei Komponenten (die Nutzdaten `K1` und `K2` und "den Rest" der Liste vom Typ `LISTE1`).

Typen wie `LISTE0` und `LISTE1` können im selben Gentle-Programm vereinbart werden, obwohl der Funktor `liste` in beiden Typvereinbarungen vorkommt. Aus dem Zusammenhang folgt jeweils, ob mit dem Funktor `liste` ein Wert vom Typ `LISTE0` oder ein Wert vom Typ `LISTE1` beginnt. Entsprechendes gilt für alle Funktoren (auch für nullstellige Funktoren wie `mo`, `di`, ... `so` und `leer`).

**Beispiel 9.6:** Binäre Bäume, die möglicherweise leer sind

```

1 'type' BIN_BAUM0
2   leer
3   binb(K1: STRING, K2: INT, Links: BIN_BAUM0, Rechts: BIN_BAUM0)

```

**Beispiel 9.7:** Binäre Bäume, die mindestens ein Element enthalten

```

1 'type' BIN_BAUM1
2   blatt(K1: STRING, K2: INT)
3   binb (K1: STRING, K2: INT, Links: BIN_BAUM1, Rechts: BIN_BAUM1)

```

Die Bäume des Typs `BIN_BAUM0` sind in einem bestimmten Sinne sehr "gleichförmig": Jeder Unterbaum eines solchen Baumes ist wieder vom Typ `BIN_BAUM0`. Ganz entsprechendes gilt auch für die Bäume des Typs `BIN_BAUM1`. Hier noch ein Beispiel für "abwechslungsreichere Bäume", deren Knoten farbig sind und unterschiedlich viele Unterbäume besitzen können. Um solche Bäume zu realisieren, sind mehrere zusammenhängende Typvereinbarungen notwendig:

**Beispiel 9.8:** Farbige Bäume, die nicht notwendig binär sind:

```

1 'type' R_BAUM -- rote Bäume
2   r_leer
3   r3(UB1: G_BAUM, UB2: G_BAUM, UB3: R_BAUM)
4   r2(UB1: B_BAUM, G_BAUM)
5 'type' G_BAUM -- grüne Bäume
6   g_leer
7   g2(UB1: R_BAUM, UB2: B_BAUM)
8   g1(UB1: B_BAUM)
9 'type' B_BAUM -- blaue Bäume
10  b1(UB1: B_BAUM)
11  b2(UB1: R_BAUM, UB2: G_BAUM)

```

Ein roter Baum ist entweder leer oder er hat drei Unterbäume (zwei grüne und einen roten) oder er hat zwei Unterbäume (einen blauen und einen grünen).

Ein grüner Baum ist entweder leer oder er hat zwei Unterbäume (einen roten und einen blauen) oder er hat nur einen Unterbaum (nämlich einen blauen).

Ein blauer Baum kann nicht leer sein. Er hat entweder nur einen Unterbaum (einen blauen) oder er hat zwei Unterbäume (einen roten und einen grünen).

Hier drei Beispiele für Werte (d.h. Terme oder Bäume) vom Typ `R_BAUM`:

```

1 r_leer
2 r2(b2(r_leer, g_leer), g_leer)

```

```
3 r3(g1(b2(r_leer, g_leer), g_leer), r2(b2(r_leer, g_leer), g_leer))
```

Ganz ähnliche Bäume eignen sich gut dazu, ein Quellprogramm (welches gerade von einem Compiler eingelesen und analysiert wurde) intern darzustellen. Die einzelnen Unterbäume eines solchen Baumes entsprechen den "größeren und kleineren Teilen des Programms" (z. B. den einzelnen Vereinbarungen, den einzelnen Anweisungen, den einzelnen Teilen einer Vereinbarung oder einer Anweisung etc.).

Der Gentle-Compiler stellt Werte von Typen wie LISTE0, LISTE1, BIN\_BAUM0, BIN\_BAUM1, R\_BAUM etc. intern durch verzeigerte Strukturen dar, mit denen heutige Rechner besonders effizient umgehen können. Als Gentle-Programmierer braucht man sich aber nicht um diese konkrete Darstellung der Werte zu kümmern. Insbesondere ist man nicht gezwungen, mit fehlerträchtigen Zeigern zu hantieren. Es genügt, sich die Werte der Typen abstrakt als Terme oder als Listen bzw. Bäume vorzustellen.

**Aufgabe 9.1.:** Programmieren Sie die folgenden Aktions-Prädikate in Gentle:

```
1 'action' AnzRotLinks(LISTE0 -> INT) -- Wie oft kommt in einer Liste rot
2                                     -- als Links-Farbe vor?
3 'action' AnzGleichLR(LSITE0 -> INT) -- Wie oft ist die Links-Farbe eines
4                                     -- Elements gleich der Rechts-Farbe?
5 'action' AnzGleichNL(LISTE0 -> INT) -- Wie oft ist die Links-Farbe eines
6                                     -- Listenelementes gleich der
7                                     -- Farbe seines Nachfolgers?
8 'action' VertauschLR(LISTE0 -> LISTE0) -- Die Links- und Rechts-Farben sollen
9                                     -- alle miteinander vertauscht werden.
```

**Aufgabe 9.2.:** Ersetzen Sie in der vorigen Aufgabe überall den Typ LISTE0 durch den Typ LISTE1 und lösen Sie die Aufgabe dann erneut.

## 10. Konkrete Syntax und abstrakte Syntax

Der erste Arbeitsschritt vieler Compiler besteht darin, dass sie ein Quellprogramm einlesen, auf formale Fehler hin prüfen und in eine interne Darstellung übersetzen. Häufig bezeichnet man das eingelesene Quellprogramm auch als "die konkrete Syntax" und die interne Darstellung als "die abstrakte Syntax" des Programms. Mit "Syntax" werden dabei mehrere Dinge bezeichnet: 1. Die (konkrete bzw. abstrakte) Darstellung eines bestimmten Programms und 2. Die allgemeinen Regeln, denen alle diese (konkreten bzw. abstrakten) Darstellungen genügen müssen.

Hier ein Beispiel, anhand dessen man die Unterschiede und Zusammenhänge zwischen konkreter und abstrakter Syntax konkret (oder abstrakt?) diskutieren kann.

1. Eine konkrete Syntax für arithmetische Ausdrücke, in der Sprache Gentle notiert. Die Bindungsstärke der einzelnen (Teil-) Ausdrücke bzw. der einzelnen Operatoren geht von 1 ("schwächste Bindung") bis 5 ("stärkste Bindung"):

```
1 'nonterm' AAusd1 -- Arithmetische Ausdruecke der Bindungsstaerke 1
2   'rule' AAusd1 : AAusd1 "+" AAusd2 -- "+" ist linksassoziativ
3   'rule' AAusd1 : AAusd1 "-" AAusd2 -- "-" ist linksassoziativ
4   'rule' AAusd1 : AAusd2
5
6 'nonterm' AAusd2 -- Arithmetische Ausdruecke der Bindungsstaerke 2
7   'rule' AAusd2 : AAusd2 "*" AAusd3 -- "*" ist linksassoziativ
8   'rule' AAusd2 : AAusd2 "/" AAusd3 -- "/" ist linksassoziativ
9   'rule' AAusd2 : AAusd3
10
```

```

11 'nonterm' AAusd3 -- Arithmetische Ausdruecke der Bindungsstaerke 3
12   'rule' AAusd3 : AAusd4 "*" AAusd3  -- "*" ist rechtsassoziativ
13   'rule' AAusd3 : AAusd4
14
15 'nonterm' AAusd4 -- Arithmetische Ausdruecke der Bindungsstaerke 4
16   'rule' AAusd4 : "+" AAusd5        -- Vorzeichen "+"
17   'rule' AAusd4 : "-" AAusd5        -- Vorzeichen "-"
18   'rule' AAusd4 : AAusd5
19
20 'nonterm' AAusd5 -- Arithmetische Ausdruecke der Bindungsstaerke 5
21   'rule' AAusd5 : Ident              -- Ein Bezeichner
22   'rule' AAusd5 : GanzLit            -- Ein Ganzzahl-Literal
23   'rule' AAusd5 : "(" AAusd1 ")"     -- Ein geklammerter Ausdruck

```

Aus dem Symbol `Ident` soll man Bezeichner wie z. B. `otto`, `emil` und `carl` ableiten können, und aus dem Symbol `GanzLit` Ganzzahlliterale wie `0`, `537` oder `123456789`. Die entsprechenden Regeln werden hier nicht angegeben und sind vorerst auch unwesentlich. Diese kontextfreie Grammatik drückt unter anderem aus:

1. Punktrechnung geht vor Strichrechnung (d.h. `*` und `/` binden stärker als `+` und `-`).
2. Die Operatoren `+`, `-`, `*` und `/` sind linksassoziativ.
3. Der Potenzierungs-Operator `**` ist rechtsassoziativ.
4. Zwei aufeinanderfolgende Vorzeichen sind nicht zulässig.

Die konkrete Syntax für Ausdrücke ist vor allem deshalb so kompliziert, weil sich viele Menschen leider an die *Infixnotation* von Operatoren wie `+`, `-`, `*`, `/` und `**` gewöhnt haben. Hätte man früher auf bestimmte polnische Mathematiker oder später auf die Firma Hewlett Packard gehört, dann würde man Ausdrücke grundsätzlich in polnischer Notation (Präfixnotation) oder in umgekehrter polnischer Notation (Postfixnotation, UPN) schreiben. Damit wären Klammern im Prinzip entbehrlich und man bräuhete keine komplizierten Regeln, um die Bindungsstärke und Assoziativität der einzelnen Operatoren festzulegen.

2. Eine abstrakte Syntax für arithmetische Ausdrücke, in der Sprache Gentle notiert (natürlich in polnischer Notation). Die Klammern sollen nur das Parsen und Lesen vereinfachen und könnten im Prinzip auch weggelassen werden (was in Gentle allerdings nicht erlaubt ist):

```

1 'type' AAUSD -- arithmetische Ausdruecke
2   add(AAUSD, AAUSD)
3   sub(AAUSD, AAUSD)
4   mul(AAUSD, AAUSD)
5   div(AAUSD, AAUSD)
6   pot(AAUSD, AAUSD) -- Potenzierung
7   min(AAUSD)        -- Vorzeichen Minus
8   ide(STRING)       -- Bezeichner, z. B. ide("otto")
9   lit(INT)          -- Ganzzahlliteral, z. B. lit(123)

```

Terme vom Typ `AAUSD` werden in einem Gentle-Programm zwar in einer bestimmten Notation geschrieben (präfix), aber das ist unwesentlich. Die Terme stellen Werte (oder: Bäume) dar, die von allen konkreten Notationen (infix, präfix, postfix oder sonstwie) abstrahieren und nur "die wesentlichen Informationen über den Ausdruck" enthalten, z. B.:

1. Zu einer Addition, Subtraktion, Multiplikation etc. gehören immer *zwei* Ausdrücke, deren Werte addiert, subtrahiert, multipliziert etc. werden sollen

2. Zu einer einstelligen Minusoperation ("Vorzeichen Minus") gehört immer *ein* Ausdruck, dessen Wert "umgedreht" werden soll.

3. Die einfachsten (d.h. "nicht-rekursiven") Ausdrücke sind *Bezeichner* (identifier) und *Literale*.

Man beachte, dass es in dieser abstrakten Darstellung zwar Ausdrücke mit einem Vorzeichen Minus gibt (z. B. `min(ide("otto"))` oder `min(lit(123))` etc.), aber keine Ausdrücke mit einem Vorzeichen Plus, da ein Ausdruck der Form `plus(A)` theoretisch und praktisch immer den gleichen Wert hat, wie der Ausdruck `A`. Mit anderen Worten: Beim Übergang von einer konkreten Darstellung eines Ausdrucks zu einer abstrakten Darstellung kann man Plus-Vorzeichen ersatzlos weglassen.

Im Zusammenhang mit der abstrakten Syntax ist es nicht mehr nötig (und wäre es unsinnig) zu fragen, welche "Rechnungsarten stärker binden" als andere und ob ein bestimmter Operator links- oder rechts-assoziativ ist. In diesem Sinne ist die obige abstrakte Syntax viel einfacher (und "maschinengerechter") als die konkrete Syntax.

**Aufgabe 10.1.:** Übersetzen Sie ("von Hand") die folgenden Ausdrücke in entsprechende Terme des Typs AAUSD:

```

1  otto +  emil
2  otto *  123
3  otto -  123 - carl      -- Achtung: - ist linksassoziativ
4  otto ** 123 ** carl    -- Achtung: "*" ist rechtsassoziativ
5  otto +  emil * carl ** 2
6  otto ** emil * carl + 2
7  + otto * -17          -- Vorzeichen "+" und -
8  (otto + emil) * 17    -- Klammern

```

**Aufgabe 10.2.:** Ergänzen Sie in der oben angegebene konkrete Syntax für Ausdrücke die Zwischensymbol-Prädikate (nonterm predicates) AAUSD1, AAUSD2, ..., AAUSD5 so mit Ausgabeparametern, dass beim Einlesen eines konkreten Ausdrucks automatisch die entsprechende abstrakte Darstellung des Ausdruck erzeugt wird. Setzen Sie dabei folgendes voraus:

1. Das Prädikat `Ident` hat einen Ausgabeparameter vom Typ `STRING`. Wenn aus dem Symbol `Ident` z. B. der Bezeichner `otto` abgeleitet wurde, dann liefert der Parser in diesem `STRING`-Parameter die Zeichenkette `otto`.

2. Das Prädikat `GanzLit` hat einen Ausgabeparameter vom Typ `INT`. Wenn aus dem Symbol `GanzLit` z. B. das Literal `123` abgeleitet wurde, dann liefert der Parser in diesem `INT`-Parameter die Ganzzahl `123`.

```
'token' Ident (->STRING)
'token' GanzLit(->INT )
```

Wieviele Ausgabeparameter haben die Zwischensymbol-Prädikate (AAUSD1, AAUSD2, ..., AAUSD5) und von welchem Typ sind die einzelnen Parameter? Wie müssen die Werte dieser Ausgabeparameter berechnet werden?

**Merke:** Zwischensymbol-Prädikate (nonterm predicates) dürfen (aus naheliegenden aber nicht ganz einfach zu erklärenden Gründen) nur Ausgabeparameter haben, aber keine Eingabeparameter.

## 11. digits\_1.g: Binärzahlen in Dezimalzahlen umwandeln

Das folgende Programm `digits_1.g` geht auf ein "genial einfaches" Beispiel von D. Knuth zurück. Es handelt sich um einen "Compiler", der jeweils eine Binärzahl einliest und eine entsprechende Dezimalzahl ausgibt. Obwohl dieses Programm sehr kurz und simple ist, hat es doch wichtige Ähnlichkeiten mit einem "richtigen Compiler".

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- digits_1.g: Ein Compiler, der Binaerzahlen in Dezimalzahlen umwandelt.
3  -----
4  'root'
5    number( -> N)
6    print (N -> )
7
8  'nonterm' number(-> INT)
9    number(-> D)      : digit (-> D) .
10   number(-> N*2 + D) : number(-> N) digit(-> D) .
11
12 'nonterm' digit(-> INT)
13   digit(-> 0)      : "0" .
14   digit(-> 1)      : "1" .
15 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 4:** Ein Gentle-Programm besteht im wesentlichen aus einer Folge von Vereinbarungen (von Prädikaten, Typen etc.). Was die Reihenfolge dieser Vereinbarungen betrifft, läßt die Sprache dem Programmierer einen großen Spielraum. So kann man z. B. das Wurzel-Prädikat (root predicate) am Anfang, irgendwo in der Mitte oder ganz am Ende eines Programms vereinbaren. Entsprechend kann man auch die anderen Prädikate und die Typen eines Programms in beliebiger Reihenfolge vereinbaren. Der Programmierer sollte diese Möglichkeiten dazu benützen, seine Gentle-Programme möglichst übersichtlich und leicht lesbar zu gestalten.

**Zeile 8:** Das Zwischensymbol `number` hat einen Ausgabeparameter vom Typ `INT`, nämlich den Wert der Binärzahl, die man aus diesem Zwischensymbol `number` ableitet.

**Man beachte:** Die Werte des Typs `INT` sind Ganzzahlen (0, +1, -1, +2, -2, +3 ...). Man sollte möglichst keine Annahmen darüber machen, ob diese Werte intern als Binärzahlen oder als Dezimalzahlen oder sonstwie dargestellt werden. Wichtig ist nur, dass die Prädikate `print` und `putI` `INT`-Werte als Dezimalzahlen ausgeben. Man könnte aber jederzeit weitere Prädikate vereinbaren (in Gentle oder in C), die `INT`-Werte als Binärzahlen oder als Hexadezimalzahlen oder "in Worten" (z. B. "einhundert-unddreiundzwanzig") ausgeben (siehe dazu auch das nächste Beispiel 10.). Ganzzahlen haben also nur bei der Ein- und Ausgabe "eine konkrete Notation", intern sind sie abstrakte Zahlen.

**Zeile 9 bis 10:** Die Regeln eines Prädikates kann man wahlweise mit dem Schlüsselwort `'rule'` beginnen (wie in den vorigen Beispielen `ahnen_v1.g` bis `ahnen_v6.g`) oder mit einem Punkt `.` abschließen (wie hier). Wer gerne Hosenträger *und* einen Gürtel trägt kann die beiden Notationen auch kombinieren. Die Notation mit dem Punkt am Ende macht etwas weniger Schreibarbeit und sieht eleganter aus, führt aber erfahrungsgemäß beim Ändern und Ergänzen der Regeln häufiger zu charakteristischen "Schreibfehlern". Wählen Sie eine der beiden Noationen und vertrauen Sie dabei auf Ihren guten Geschmack.

**Zeile 9:** "Wenn eine Binärzahl (`number`) nur aus einer Ziffer (`digit`) besteht und die Ziffer den Wert `D` hat, dann hat auch die Binärzahl den Wert `D`".

**Zeile 10:** "Wenn eine Binärzahl (`number` auf der linken Seite der Regel) aus einer Binärzahl (`number` auf der rechten Seite der Regel) mit dem Wert `N` und einer Ziffer (`digit`) mit dem Wert `D` besteht, dann hat die Binärzahl (`number` auf der rechten Seite der Regel) den Wert  $N*2 + D$ ". In Gentle kann man das deutlich kürzer und prägnanter formulieren als auf Deutsch.

**Zeile 12:** Auch das Zwischensymbol `digit` hat einen Ausgabeparameter vom Typ `INT`, nämlich den Wert der Binärziffer, die man aus diesem Zwischensymbol `digit` ableitet.

**Zeile 13:** "Wenn man aus dem Zwischensymbol `digit` eine `0` ableitet, dann bekommt das Zwischensymbol eine `0` als Parameterwert".

**Zeile 14:** Entsprechend wie Zeile 13.

**Aufgabe 11.1.:** Erzeugen Sie das Programm `digits_1.exe` (z. B. indem Sie die Stapeldatei `build.bat` im Verzeichnis `digits_1` aufrufen) und testen Sie es mit möglichst vielen (richtigen und falschen) Eingaben.

**Aufgabe 11.2.:** Binärzahlen kann man auch als Folgen der Buchstaben `N` (für Null) und `E` (für Eins) darstellen. Statt `110` schreibt man dann `EEN`. Ändern Sie das Programm `digits_1.g` so, dass es eine solche N-E-Folge einliest und eine entsprechende Dezimalzahl ausgibt. Hinweis: Sie müssen im Programm `digits_1.g` nur zwei Zeichen ändern. Welche?

**Aufgabe 11.3.:** Ändern Sie das Programm `digits_1.g` so, dass es eine Oktalzahl einliest und eine entsprechende Dezimalzahl ausgibt.

Im Programm `digits_1.g` werden die Binärzahlen, die eingelesen und übersetzt werden sollen, vollständig durch Zwischensymbol-Prädikate (`nonterm predicates`) beschrieben, nämlich durch die Prädikate `number` und `digit`. Jede Ziffer `0` bzw. `1` wird als ein Lexem beschrieben und getrennt von den anderen Ziffern eingelesen und bearbeitet. Das ist zwar möglich, aus Gründen der Effizienz aber unüblich. Im Abschnitt 13. wird die übliche und effizientere Methode zum Einlesen und Erkennen von Literalen, Bezeichnern und ähnlichen "elementaren Bestandteilen eines Quellprogramms" vorgeführt.

## 12. digits\_2.g: Hexadezimalzahlen in Binärzahlen umwandeln

Das Problem, Hexadezimalzahlen in Binärzahlen umzuwandeln, hat auf den ersten Blick große Ähnlichkeit mit dem Problem, Binärzahlen in Dezimalzahlen umzuwandeln (siehe voriges Abschnitt). Wenn man dann aber beide Probleme durch Gentle-Programme löst, ergeben sich ein paar wichtige und möglicherweise interessante Unterschiede. Vielleicht möchten Sie zuerst einmal selbst ein Gentle-Programm schreiben, welches eine Hexadezimalzahl einliest und eine entsprechende Binärzahl ausgibt, ehe Sie sich das folgende Programm ansehen?

Hier müssen wir unter anderem folgendes Problem lösen: Mit dem Prädikat `print` kann man (unter anderem) auch Ganzzahlen ausgeben, aber nur als Dezimalzahlen. Da wir Ganzzahlen hier aber als Binärzahlen ausgeben wollen, müssen wir dafür ein spezielles Prädikat vereinbaren. Dieses Prädikat muss aus einer Ganzzahl die einzelnen Ziffern ihrer binären Darstellung berechnen und (mithilfe des in C programmierten Prädikates `PutI`) ausgeben.

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- digits_2.g: Ein Compiler, der Hexadezimalzahlen in Binaerzahlen
3  -- uebersetzt.
4  -----
5  'root'
6  number( -> N)
7  binaus(N -> )

```

```

8 -----
9 'nonterm' number(-> INT)
10   number(-> D)      : digit (-> D) .
11   number(-> N*16 + D) : number(-> N) digit(-> D) .
12
13 'nonterm' digit(-> INT)
14   digit(-> 0) : "0" .
15   digit(-> 1) : "1" .
16   digit(-> 2) : "2" .
17   digit(-> 3) : "3" .
18   digit(-> 4) : "4" .
19   digit(-> 5) : "5" .
20   digit(-> 6) : "6" .
21   digit(-> 7) : "7" .
22   digit(-> 8) : "8" .
23   digit(-> 9) : "9" .
24   digit(-> 10) : "a" .
25   digit(-> 10) : "A" .
26   digit(-> 11) : "b" .
27   digit(-> 11) : "B" .
28   digit(-> 12) : "c" .
29   digit(-> 12) : "C" .
30   digit(-> 13) : "d" .
31   digit(-> 13) : "D" .
32   digit(-> 14) : "e" .
33   digit(-> 14) : "E" .
34   digit(-> 15) : "f" .
35   digit(-> 15) : "F" .
36 -----
37 'action' binaus(Ganzzahl: INT ->)
38   -- Gibt eine nicht-negative Ganzzahl als Binaerzahl aus:
39   binaus(N): eq( N, 0) PutI(0).
40   binaus(N): eq((N/2)*2, N) binaus(N/2) PutI(0).
41   binaus(N): lt((N/2)*2, N) binaus(N/2) PutI(1).
42
43 'action' PutI(INT) -- Wurde in C realisiert (siehe Datei text_io.c)
44 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 24 bis 35:** Um dem Benutzungskomfort dieses Compilers zu erhöhen, werden grosse und kleine Buchstaben (a bis f und A bis F) als Hexadezimalziffern erlaubt. Hier ist der Unterschied zwischen einer Ziffer (z. B. a) und ihrem Wert (10) augenfälliger als im vorigen Beispiel `digits_1.g`. Außerdem können hier verschiedene Ziffern (z. B. a und A) den gleichen Wert (10) haben.

**Zeile 37:** Das Aktions-Prädikat `binaus` hat einen Eingabeparameter vom Typ `INT` und gibt den entsprechenden Wert als Binärzahl zur aktuellen Ausgabe (zum Bildschirm) aus.

**Zeile 39 bis 41:** Die einzelnen Ziffern der Binärzahl werden mit dem Prädikat `PutI` ausgegeben. Das Prädikat `print` wäre hier nicht geeignet, weil es nach jeder Ausgabe eine neue Zeile beginnt.

**Zeile 39:** "Wenn `N` gleich 0 ist, wird eine 0 ausgegeben und damit ist die Ausgabe der betreffenden Zahl beendet".

**Zeile 40:** Für die Ganzzahldivision `/` gilt z. B.  $7/3 = 2$  (und nicht gleich 2,3 oder 2,33 oder 2,33333333, all diese Ergebnisse sind ja mathematisch nicht ganz korrekt, das Ergebnis 2 ist dagegen das mathematisch korrekte Ergebnis der Ganzzahldivision  $7/3$ ). Wenn man eine gerade Ganzzahl `N` durch 2 teilt und das Ergebnis mit 2 multipliziert, erhält man wieder `N`. Macht man das gleiche mit einer ungeraden Ganzzahl `N`, so erhält man als Ergebnis `N-1` (z. B.  $(6/2)*2=6$ , aber  $(7/2)$

\*2=6). Die Regel in Zeile 40 besagt: "Wenn  $N$  gerade ist, dann wird zuerst der Wert  $N/2$  als Binärzahl ausgegeben und dahinter eine 0".

**Zeile 41:** "Wenn  $N$  ungerade ist, dann wird zuerst der Wert  $N/2$  ausgegeben und dahinter eine 1".

**Aufgabe 12.1.:** Erzeugen Sie das Programm `digits_2.exe` (z. B. indem Sie die Stapeldatei `build.bat` im Verzeichnis `digits_2` aufrufen) und testen Sie es mit möglichst vielen (richtigen und falschen) Eingaben.

**Aufgabe 12.2.:** Die Ausgabe des Programms `digits_2.g` beginnt immer mit einer 0. Ändern Sie `digits_2.g` so, dass diese führende 0 nicht ausgegeben wird. Nur der Wert Null soll natürlich als 0 ausgegeben werden. Möglicherweise müssen Sie für diese kleine Änderung ein weiteres Aktions-Prädikat vereinbaren.

### 13. Ein Scanner übersetzt Lexeme in Token

Die Aufgabe eines Scanners besteht darin, ein Quellprogramm Zeichen für Zeichen einzulesen, in Lexeme zu zerlegen und für jedes Lexem ein entsprechendes Token (an den Parser) zu liefern. Ein Lexem ist dabei eine Zeichenkette im Quellprogramm, z. B. ein Schlüsselwort wie `begin`, `if` oder `end`, oder ein Bezeichner wie `summand_16` oder `bisherigesMaximum` etc., oder ein Operator (eine "Operationsbezeichnung") wie `:=`, `<=`, `=`, `+`, `-`, `*`, `/` etc. etc. Ein Lexem kann also aus einem oder mehreren Zeichen bestehen. Ein Token ist eine interne Datenstruktur des Compilers, die so strukturiert und organisiert ist, dass der Compiler möglichst effizient damit umgehen kann. Aufgabe eines Scanners ist es, eine Folge von Zeichen einzulesen, die Zeichenfolge in eine Folge von Lexemen zu zerlegen und für jedes Lexem ein entsprechendes Token zu liefern.

Einige Lexeme stehen in einer "Eins-zu-Eins-Beziehung" zu ihren Token. Z. B. wird ein Schlüsselwort-Lexem wie `begin` vom Scanner in ein entsprechendes `begin`-Token und ein Lexem `:=` in ein entsprechendes `:=`-Token übersetzt. Alle `begin`-Token sind gleich und alle `:=`-Token sind gleich, aber ein `begin`-Token und ein `:=`-Token unterscheiden sich voneinander. Solche Lexeme (bzw. Token) werden im folgenden als Eins-zu-Eins-Lexeme (bzw. Eins-zu-Eins-Token) bezeichnet (gemeint ist jeweils: Ein Lexem, ein Token).

Für andere Lexeme gilt, dass der Scanner mehrere verschiedene Lexeme in gleiche Token übersetzt. Z. B. wird jedes Bezeichner-Lexem in ein Bezeichner-Token übersetzt. Die Bezeichner-Lexeme im Quellprogramm unterscheiden sich, trotzdem sind alle Bezeichner-Token gleich. Nur durch geeignete Attribute (oder: Parameter) wird z. B. das Token für den Bezeichner `summand_16` vom Token für den Bezeichner `bisherigesMaximum` unterschieden. Wenn ein Scanner für mehrere verschiedene Lexeme gleiche Token liefert, dann bezeichnen wir die Lexeme bzw. Token als Viele-zu-Eins-Lexeme bzw. als Viele-zu-Eins-Token (gemeint ist jeweils: Viele Lexeme, ein Token).

Es ist meistens möglich und üblich, dass ein Scanner alle Eins-zu-Eins-Lexeme als Konstanten enthält und die Lexeme im Quellprogramm durch Vergleiche mit diesen Konstanten erkennt. In einem Gentle-Programm kann man solche Lexeme als Zeichenketten-Literale angeben (z. B. `mutter`, `vater`, `gross`, `ur`, `Die`, `Mutter`, `von`, `Maria` etc.). Bei typischen Viele-zu-Eins-Lexemen ist das nicht möglich: Ein Scanner kann z. B. nicht eine Liste aller in Pascal erlaubten Bezeichner enthalten, weil diese Liste viel zu groß wäre. Stattdessen enthält der Scanner ein kleines Unterprogramm, welches von einer Zeichenkette feststellen kann, ob sie als Bezeichner zulässig ist oder nicht. Solche Unterprogram-

me sollte man aus Gründen der Effizienz nicht in der Sprache Gentle schreiben. Man muss sie aber in einem Gentle-Programm durch sogenannte Token-Prädikate (token predicates) repräsentieren.

Die in vielen Programmiersprachen wichtigsten Viele-zu-Eins-Lexeme sind Bezeichner (identifier), Ganzzahl-Literale (integer literals), Bruchzahl-Literale (real literals) und Zeichenketten-Literale (string literals). Zum Gentle-System gehören auch einige Unterprogramme, die verschiedene Arten von Bezeichnern (Bezeichner wie in Pascal, Bezeichner wie in C, Bezeichner wie in Ada etc.) und verschiedene Arten von Ganzzahl-Literalen (Ganzzahl-Literale wie in Pascal oder wie in Ada) erkennen und ein entsprechendes Token liefern können. Diese Unterprogramme kann man in einem Gentle-Programm durch entsprechende Token-Prädikate beschreiben und einbinden lassen. Die folgenden Beispiel-Programme sollen zeigen, wie das im Einzelnen funktioniert.

### 13.1. ides.g: Bezeichner mit einem STRING-Parameter

Das Programm `ides.g` liest eine Folge von Bezeichnern ein, zählt sie und gibt die Bezeichner und ihre Anzahl aus. Für jeden Bezeichner, den der Scanner erkennt, liefert er ein bestimmtes Token (hier: ein Bezeich-Token) an den Parser. Damit man die einzelnen Bezeichner unterscheiden kann, hat jedes Bezeich-Token einen Parameter vom Typ `STRING`, nämlich den erkannten Bezeichner. Alle Bezeich-Token sind gleich. Sie unterscheiden sich nur durch ihre `STRING`-Parameter.

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- ides.g: Dient zum Testen der Token-Beschreibungsdateien
3  -- id_s_pas.t, id_s_eif.t, id_s_ceh.t, id_s_ada.t,
4  -- Der hier spezifizierte Compiler liest eine Folge von Bezeichnern
5  -- (identifier) ein und gibt die Bezeichner und ihre Anzahl aus.
6  -----
7  'nonterm' Bez_Folge(->INT)    -- Eine nichtleere Folge von Bezeichnern
8    'rule' Bez_Folge(->N+1): Bezeich(->ID) Bez_Folge(->N) PutS(ID) PutS(" ")
9    'rule' Bez_Folge(-> 1): Bezeich(->ID)                PutS(ID) PutS(" ")
10 -----
11 'token'   Bezeich(->STRING)  -- Ein Bezeichner, beschrieben in .t-Datei
12 -----
13 'root'
14   Bez_Folge(->ANZAHL)
15   N1 PutS("Anzahl der Bezeichner: ") PutI(ANZAHL) N1
16 -----
17 'action' PutS(STRING)        -- Ist in der Datei text_io.c definiert
18 'action' PutI(INT)           -- Ist in der Datei text_io.c definiert
19 'action' N1                  -- Ist in der Datei text_io.c definiert
20 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 7:** Aus dem Zwischensymbol `Bez_Folge` kann man eine Folge von Bezeichnern ableiten. Diese Folge muss mindestens einen Bezeichner enthalten. Das Prädikat `Bez_Folge` hat einen Ausgabe-Parameter vom Typ `INT`. Der gibt an, aus wievielen Bezeichnern die Bezeichnerfolge besteht.

**Zeile 8:** Eine `Bez_Folge` besteht entweder aus einem `Bezeich` gefolgt von einer `Bez_Folge` oder **Zeile 9:** nur aus einem `Bezeich`.

**Zeile 11:** Das Unterprogramm zum Erkennen von Bezeichnern wurde (aus Gründen der Effizienz) nicht in der Sprache Gentle geschrieben sondern "auf einer niedrigeren, maschinennäheren Ebene" (siehe unten). Dieses Unterprogramm wird im Gentle-Programm durch ein Token-Prädikat (token predicate) repräsentiert. Aus der Vereinbarung dieses Prädikates `Bezeich` folgt: Immer wenn der Scanner mithilfe dieses Unterprogramms einen Bezeichner erkennt, liefert er ein `Bezeich`-Token und einen (Ausga-

be-) Parameter vom Typ `STRING`. Weiter unten werden wir sehen: Dieser `STRING`-Parameter ist der Bezeichner selbst (d.h. wenn der Scanner z. B. den Bezeichner `summand_16` erkennt, liefert er ein Bezeich-Token mit dem `STRING`-Parameter "`summand_16`", wenn er den Bezeichner `bisherigesMaximum` erkennt, liefert er ein Bezeich-Token mit dem `STRING`-Parameter "`bisherigesMaximum`" etc.). In den Zeilen 8 und 9 werden die `STRING`-Parameter der Bezeich-Token ausgegeben (durch die Prädikat-Aufrufe `PutS(ID)`).

**Zeile 14:** Der Parser versucht, eine Zeichenkette einzulesen, die man aus dem Zwischensymbol `Bez_Folge` ableiten kann. Wenn ihm das gelingt, berechnet er auch den Parameter `ANZAHL` dieser Bezeichner-Folge, d.h. die Anzahl der Bezeichner, aus der die Folge besteht.

**Zeile 15:** Nach einem `newline`-Zeichen wird die Anzahl der eingelesenen Bezeichner ausgegeben. Die Bezeichner selbst wurden schon durch die Aufrufe `PutS(ID)` in den Zeilen 8 und 9 ausgegeben.

Die folgende DOS-Stapeldatei `build_s.bat` erzeugt aus dem einen Gentle-Programm `ides.g` vier ausführbare Programme (`ides_pas.exe`, `ides_eif.exe`, `ides_ceh.exe` und `ides_ada.exe`) und läßt sie ausführen. Die vier ausführbaren Programme machen im Prinzip alle das Gleiche: Sie lesen eine Folge von Bezeichnern ein und geben die Bezeichner und ihre Anzahl wieder aus (wie im Gentle-Programm `ides.g` beschrieben). Sie unterscheiden sich aber darin, was für Lexeme sie als Bezeichner akzeptieren: `ides_pas.exe` akzeptiert Bezeichner im Stil von Pascal und Modula, `ides_eif.exe` akzeptiert Eiffel-Bezeichner und `ides_ceh.exe` bzw. `ides_ada.exe` erlauben Bezeichner wie in C bzw. Ada. Die Unterschiede zwischen den ausführbaren Programmen kommen dadurch zustande, dass für das Token-Prädikat `Bezeich` (siehe Gentle-Programm `ides.g`) jedesmal ein anderes Unterprogramm eingebunden wird, welches die entsprechenden Bezeichner erkennt.

```

1 @echo off
2 rem -----
3 rem  build_s.bat fuer ides.g im Verzeichnis test_ide
4 rem -----
5 copy    %genlib%\text_io.o > nul
6 copy    %genlib%\errmsg.o > nul
7 copy    %genlib%\main1.o  > nul
8 copy    %genlib%\grts.o   > nul
9
10 gentle ides.g
11
12 echo    Bezeichner wie in Pascal oder Modula,      mit STRING-Attribut:
13 reflex Bezeich=id_s_pas.t
14 flex   gen.l
15 bison  gen.y
16 gcc    ides.c gen.tab.c lex.yy.c text_io.o errmsg.o main1.o grts.o
17 ren    a.exe ides_pas.exe
18 ides_pas inpas.ide
19
20 echo    Bezeichner wie in Eiffel,                  mit STRING-Attribut:
21 reflex Bezeich=id_s_eif.t
22 flex   gen.l
23 bison  gen.y
24 gcc    ides.c gen.tab.c lex.yy.c text_io.o errmsg.o main1.o grts.o
25 ren    a.exe ides_eif.exe
26 ides_eif ineif.ide
27
28 echo    Bezeichner wie in C,                       mit STRING-Attribut:
29 reflex Bezeich=id_s_ceh.t
30 flex   gen.l

```

```

31 bison      gen.y
32 gcc        ides.c gen.tab.c lex.yy.c text_io.o errmsg.o main1.o grts.o
33 ren        a.exe ides_ceh.exe
34 ides_ceh   inceg.ide
35
36 echo       Bezeichner wie in Ada,                mit STRING-Attribut:
37 reflex     Bezeich=id_s_ada.t
38 flex       gen.l
39 bison      gen.y
40 gcc        ides.c gen.tab.c lex.yy.c text_io.o errmsg.o main1.o grts.o
41 ren        a.exe ides_ada.exe
42 ides_ada   inada.ide
43 rem -----

```

**Zeile 5 bis 8:** Die vier Objektdateien `text_io.o`, `errmsg.o`, `main1.o` und `grts.o` werden hier nur als "Notbehelf" aus dem Gentle-Bibliotheksverzeichnis in das Verzeichnis `test_ide` kopiert, damit später die Kommandozeilen mit den `gcc`-Aufrufen (siehe Zeile 17, 25, 33 und 41) nicht länger werden, als unter DOS erlaubt.

**Zeile 10:** Das Gentle-Programm `ides.g` wird mit dem Gentle-Compiler `gentle` in ein entsprechendes C-Programm namens `ides.c` umgewandelt. Gleichzeitig erzeugt der Gentle-Compiler ein paar Hilfsdateien (`gen.h`, `gen.lit`, `gen.tkn` und `gen.y`) mit Informationen für den Hilfs-Generator `reflex` und den Parser-Generator `bison`.

**Zeile 13:** Dieser Aufruf des Hilfs-Generators `reflex` faßt die Dateien `gen.h`, `gen.lit`, `gen.tkn` und `id_s_pas.t` zu einer Scanner-Spezifikation namens `gen.l` zusammen, aus der später mit dem Scanner-Generator `flex` ein Scanner generiert wird (siehe Zeile 15). Die Angabe `Bezeich=id_s_pas.t` besagt, dass das Unterprogramm zum Erkennen von `Bezeich`-Lexemen und zum Berechnen des dazugehörigen `STRING`-Parameters in der Datei `id_s_pas.t` steht. Wenn die Angabe `Bezeich=id_s_pas.t` fehlt, dann erwartet `reflex`, dass dieses Unterprogramm in einer Datei namens `Bezeich.t` beschrieben wird. Dabei ist `Bezeich` der Name des Token-Prädikates, das im Gentle-Programm vereinbart wurde (siehe Zeile 11 des Programm `ides.g`).

**Zeile 14:** Mit dem Scanner-Generator `flex` wird aus der Scanner-Spezifikation `gen.l` ein Scanner (d.h. eine C-Datei namens `lex.yy.c`) erzeugt.

**Zeile 15:** Mit dem Parser-Generator `bison` wird aus der Parser-Spezifikation `gen.y` ein Parser (d.h. eine C-Datei namens `gen.tab.c`) erzeugt.

**Zeile 16:** Mit dem GNU-Compiler-und-Binder-System wird aus den angegebenen `.c`- und `.o`-Dateien ein Maschinenprogramm namens `a.exe` erzeugt.

**Zeile 17:** Die Datei `a.exe` wird umbenannt in `ides_pas.exe`

**Zeile 18:** Das Programm `ides_pas.exe` wird (mit dem Dateinamen `inpas.ide` als Parameter) aufgerufen. Die Datei `inpas.ide` enthält eine Folge von 5 Bezeichnern, wie sie in den Programmiersprachen Pascal und Modula erlaubt sind.

**Zeile 20 bis 26:** Diese Zeilen entsprechen weitgehend den Zeilen 12 bis 18. Den wichtigsten Unterschied erkennt man, wenn man die Zeilen 13 und 21 vergleicht. Die Angabe `Bezeich=id_s_eif.t` in Zeile 21 besagt, dass das Unterprogramm zum Erkennen von `Bezeich`-Lexemen und zum Berechnen des dazugehörigen `STRING`-Parameters in der Datei `id_s_eif.t` steht. Die Datei `ineif.ide` (siehe Zeile 26) enthält eine Folge von 5 Bezeichnern, wie sie in der Programmiersprache Eiffel erlaubt sind.

**Zeile 28 bis 34:** Diese Zeilen entsprechen weitgehend den Zeilen 12 bis 18. Den wichtigsten Unterschied erkennt man, wenn man die Zeilen 13 und 29 vergleicht. Die Angabe `Bezeich=id_s_ceh.t` in Zeile 29 besagt, dass das Unterprogramm zum Erkennen von `Bezeich`-Lexemen und zum Berechnen des dazugehörigen `STRING`-Parameters in der Datei `id_s_ceh.t` steht. Die Datei `inceh.ide` (siehe Zeile 34) enthält eine Folge von 5 Bezeichnern, wie sie in der Programmiersprache C erlaubt sind.

**Zeile 36 bis 42:** No more comment.

Hier die Datei `id_s_pas.t`, in der beschrieben wird, wie der Scanner `Bezeich`-Lexeme erkennen und ihren `STRING`-Parameter berechnen soll:

```

1 /* ----- */
2 /* id_s_pas.t: Bezeichner wie in Pascal oder Modula */
3 /* Liefert den Bezeichner als Zeichenkette vom Typ STRING */
4 /* ----- */
5 [A-Za-z][A-Za-z0-9]* {
6     yylval.attr[1] = strdup(yytext);
7     yysetpos();
8     return Bezeich;
9 }
10 /* ----- */

```

**Zeile 1 bis 4:** Diese Kommentarzeilen müssen mit einem Blank (oder Tab-Zeichen) beginnen, damit der Scanner-Generator `flex` sie als Kommentare erkennt. Außerdem müssen sie wie C-Kommentare aussehen, damit der C-Compiler sie als Kommentare erkennt.

**Zeile 5:** Dieser reguläre Ausdruck beschreibt, welche Zeichenfolgen als `Bezeich`-Lexeme gelten sollen: In Pascal und Modula muss das erste Zeichen ein großer oder kleiner Buchstabe sein, danach dürfen beliebig viele (d.h. 0 oder mehr) Buchstaben oder Ziffern kommen.

**Zeile 5 bis 9:** Auf den regulären Ausdruck folgt ein C-Block (d.h. eine Folge von Anweisungen, die in der Programmiersprache C geschrieben sind). Ein solcher Block muss in geschweifte Klammern eingeschlossen sein.

**Zeile 6:** `yylval` ist ein Verbund (struct). Die Verbund-Komponente `yylval.attr` ist eine Reihung (array). Die Reihungs-Komponente `yylval.attr[1]` entspricht dem ersten (und einzigen) Ausgabe-Parameter des `Bezeich`-Tokens (welches in Zeile 8 vom Scanner an den Parser zurückgeliefert wird). Jeder Parameter muss genau ein Maschinenwort (unter DOS/Windows: 32 Bit) lang sein. Längere Parameter müssen durch Zeiger repräsentiert werden. Die Variable `yytext` enthält die Zeichenkette, die der Scanner gerade als Lexem entsprechend dem regulären Ausdruck in Zeile 5 erkannt hat. Diese Zeichenkette wird mit `strdup` kopiert und die Kopie wird zum ersten (und einzigen) Parameter des `Bezeich`-Tokens gemacht.

**Zeile 7:** Der Scanner "weiß immer, an welcher Stelle der Eingabedatei er gerade ist". Mit dem Aufruf der Funktion `yysetpos` wird die Stelle hinter dem gerade erkannten `Bezeich`-Lexem zur aktuellen Stelle gemacht.

**Zeile 8:** Der Scanner gibt (an den Parser) ein `Bezeich`-Token zurück. Das `STRING`-Attribut dieses Tokens steht in der Variablen `yylval.attr[1]`.

Die folgenden Dateien `id_s_eif.t`, `id_s_ceh.t` und `id_s_ada.t` unterscheiden sich von `id_s_pas.t` jeweils nur durch den regulären Ausdruck in Zeile 5:

```

1  /* ----- */
2  /* id_s_eif.t: Bezeichner (identifizier) wie in Eiffel */
3  /* Liefert den Bezeichner als Zeichenkette vom Typ STRING */
4  /* ----- */
5  [A-Za-z][A-Za-z0-9_]* {
6      yylval.attr[1] = strdup(yytext);
7      yysetpos();
8      return Bezeich;
9  }
10 /* ----- */

```

**Zeile 5:** In Eiffel muss das erste Zeichen eines Bezeichners ein (großer oder kleiner) Buchstabe sein. Danach dürfen beliebig viele Buchstaben, Ziffern oder Unterstriche `_` folgen.

```

1  /* ----- */
2  /* id_s_ceh.t: Bezeichner wie in C */
3  /* Liefert den Bezeichner als Zeichenkette vom Typ STRING */
4  /* ----- */
5  [A-Za-z_][A-Za-z0-9_]* {
6      yylval.attr[1] = strdup(yytext);
7      yysetpos();
8      return Bezeich;
9  }
10 /* ----- */

```

**Zeile 5:** In C muss das erste Zeichen eines Bezeichners ein (großer oder kleiner) Buchstabe oder ein Unterstrich sein. Danach dürfen beliebig viele Buchstaben, Ziffern oder Unterstriche folgen.

```

1  /* ----- */
2  /* id_s_ada.t: Bezeichner (identifizier) wie in Ada */
3  /* Liefert den Bezeichner als Zeichenkette vom Typ STRING */
4  /* ----- */
5  [A-Za-z](_[A-Za-z0-9]|[A-Za-z0-9])* {
6      yylval.attr[1] = strdup(yytext);
7      yysetpos();
8      return Bezeich;
9  }
10 /* ----- */

```

**Zeile 5:** In Ada muss das erste Zeichen eines Bezeichners ein (großer oder kleiner) Buchstabe sein. Danach dürfen beliebig viele "Zeichengruppen" kommen, wobei jede Zeichengruppe entweder aus einem Unterstrich `_` gefolgt von einem Buchstaben oder einer Ziffer besteht oder nur aus einem Buchstaben oder einer Ziffer besteht. Aus dieser Beschreibung folgt: In einem Ada-Bezeichner darf ein Unterstrich nur zwischen zwei anderen ("nicht-Unterstrich-") Zeichen stehen.

Hier die Datei `inpas.ide`, die zum Testen des Programms `ides_pas.exe` gedacht ist:

```

1  -- inpas.ide: Eingabedatei fuer ides_pas.exe und idei_pas.exe
2  anna berta
3  a7 b5 x1234

```

Und hier die entsprechenden Ausgaben des Programms `ides_pas.exe`:

```

1  x1234 b5 a7 berta anna
2  Anzahl der Bezeichner: 5

```

Man beachte, dass die Bezeichner im Vergleich zur Eingabe in umgekehrter Reihenfolge ausgegeben werden. Das liegt daran, dass der Parser (der mit dem Parser-Generator `bison` aus den Zwischensymbol-Prädikaten (nonterm predicates) des Gentle-Programms erzeugt wurde), nach der LR-Methode arbeitet. Mit dieser Methode findet man (wenn möglich) eine rechtskanonische Ableitung in umgekehrter Reihenfolge (right canonical derivation in reverse order).

So wie hier das Programm `ides_pas.exe` mit der Datei `inpas.ide` getestet wurde, kann man die Programme `ides_eif.exe`, `ides_ceh.exe` und `ides_ada.exe` mit den Dateien `ineif.ide`, `inceh.ide` bzw. `inada.ide` testen.

**Aufgabe 13.1.1.:** Erzeugen Sie die mithilfe der Stapeldatei `build_s.bat` die Programme `ides_pas.exe`, `ides_eif.exe`, `ides_ceh.exe` und `ides_ada.exe`. Rufen Sie jedes dieser Programme mehrfach auf und geben Sie gleich auf der Kommandozeile eine Folge von Bezeichnern an, z. B. so:

```
ides_pas "otto b1234 xxx"
ides_pas "123a emil"
ides_eif "A_B_C A123_b"
ides_ceh "_abc_ _ A_B"
ides_ada "a_b_c a__b"
```

Geben Sie gezielt auch "falsche Bezeichner" an und prüfen Sie, ob Sie eine entsprechende Fehlermeldung erhalten. Machen Sie sich auf diese "experimentelle Weise" genau damit vertraut, welches der vier Programme welche Bezeichner akzeptiert bzw. ablehnt.

### 13.2. `idei.g`: Bezeichner mit einem IDENT-Parameter

Im Programm `ides.g` wurde ein Token-Prädikat namens `Bezeich` vereinbart, welches einen Ausgabe-Parameter vom Typ `STRING` hat. Auf diesen `STRING`-Parameter soll der Buchstabe `s` bzw. `S` in den Datei-Namen `ides.g`, `build_s.bat`, `id_s_pas.t`, `id_s_eif.t`, `id_s_ceh.t` und `id_s_ada.t` verweisen. Der `STRING`-Parameter des Token-Prädikates ist zwar (hoffentlich) leicht verständlich, aber es gibt noch eine bessere Lösung. Da praktisch jeder Compiler Bezeichner erkennen, in einer Symboltabelle speichern und verwalten muss, gibt es in Gentle dafür eine "vorgefertigte Standardlösung". Die Datei `idents.c` stellt einem unter anderem zwei Prädikate

```
1 'action'    string_to_id(STRING -> IDENT)
2 'action'    id_to_string(IDENT  -> STRING)
```

zur Verfügung mit der man Werte vom Typ `STRING` in Werte vom Typ `IDENT` und umgekehrt umwandeln lassen kann. Der Umgang mit Werten vom Typ `IDENT` ist im allgemeinen effizienter, als der Umgang mit `STRING`-Werten. Außerdem stellt die Datei `idents.c` einem noch weitere Prädikate

```
1 'action'    DefMeaning(IDENT, MEANING -> )
2 'condition' HasMeaning(IDENT          -> MEANING)
```

zur Verfügung, mit denen man bequem und sehr effizient eine Symboltabelle verwalten kann. Diese weiteren Prädikate werden in späteren Beispielen benützt und erläutert (siehe ???). Im folgenden Beispiel-Programm `idei.g` wird gezeigt, wie man den Typ `IDENT` und die Prädikate `string_to_id` und `id_to_string` anwenden kann. Der Buchstabe `i` bzw. `I` in den Datei-Namen `idei.g`, `build_i.bat`, `id_i_pas.t`, `id_i_eif.t`, `id_i_ceh.t` und `id_i_ada.t` soll auf den Typ `IDENT` verweisen.

Das Programm `idei.g` liest (ganz ähnlich wie das Programm `ides.g`) eine Folge von Bezeichnern ein, zählt sie und gibt die Bezeichner und ihre Anzahl aus. Das Token-Prädikat `Bezeich` hat hier aber keinen Parameter vom Typ `STRING`, sondern einen Parameter vom Typ `IDENT`:

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- idei.g: Dient zum Testen der Token-Beschreibungsdateien
3  -- id_i_pas.t, id_i_eif.t, id_i_ceh.t, id_i_ada.t,
4  -- Der hier spezifizierte Compiler liest eine Folge von Bezeichnern
5  -- (identifizier) ein und gibt die Bezeichner und ihre Anzahl aus.
6  -----
7  'nonterm' Bez_Folge(->INT)    -- Eine nichtleere Folge von Bezeichnern
8    'rule' Bez_Folge(->N+1): Bezeich(->ID) Bez_Folge(->N)
9      id_to_string(ID -> SID)
10     PutS(SID) PutS(" ")
11  'rule' Bez_Folge(-> 1): Bezeich(->ID)
12     id_to_string(ID -> SID)
13     PutS(SID) PutS(" ")
14  -----
15  'token'   Bezeich(->IDENT)    -- Ein Bezeichner, beschrieben in .t-Datei
16  'type'    IDENT
17  -----
18  'root'
19     Bez_Folge(->ANZAHL)
20     Nl PutS("Anzahl der Bezeichner: ") PutI(ANZAHL) Nl
21  -----
22  'action'  PutS          (STRING)    -- In Datei text_io.c definiert
23  'action'  PutI          (INT)       -- In Datei text_io.c definiert
24  'action'  Nl            -- In Datei text_io.c definiert
25  'action'  id_to_string(IDENT->STRING) -- In Datei Idents.c definiert
26  --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 8:** Das Prädikat `Bezeich` hat einen Parameter namens `ID` vom Typ `IDENT`.

**Zeile 9:** Dieser Parameter `ID` wird in einen entsprechenden Wert namens `SID` vom Typ `STRING` umgewandelt und

**Zeile 10:** ausgegeben.

**Zeile 11 bis 13:** Entsprechend wie 8 bis 10.

**Zeile 15:** Hier wird vereinbart, dass das Token-Prädikat `Bezeich` einen Ausgabe-Parameter vom Typ `IDENT` haben soll. Das Unterprogramm, mit dem der Scanner `Bezeich`-Lexeme erkennen und ihren `IDENT`-Parameter berechnen kann, wird nicht hier im Gentle-Programm beschrieben, sondern auf einer "niedrigeren und effizienteren" Ebene (in einer `.t`-Datei, siehe unten).

**Zeile 16:** Der Name `IDENT` bezeichnet einen Typ. Welche Werte zu diesem Typ gehören, wird aber auch nicht hier im Gentle-Programm beschrieben, sondern in der C-Datei `Idents.c`.

**Zeile 25:** Auch das Prädikat `id_to_string` wird von der Datei `Idents.c` zur Verfügung gestellt. Deshalb brauchen (und dürfen) hier keine Regeln für dieses Prädikat angegeben werden.

Die folgende DOS-Stapeldatei `build_i.bat` erzeugt aus dem einen Gentle-Programm `idei.g` vier ausführbare Programme (`idei_pas.exe`, `idei_eif.exe`, `idei_ceh.exe` und `idei_ada.exe`), die sich durch die Art der Bezeichner unterscheiden, die sie akzeptieren:

```

1  @echo off
2  rem -----
3  rem  build_i.bat fuer idei.g im Verzeichnis test_ide
4  rem -----
5  copy      %genlib%\text_io.o > nul

```

```

6 copy      %genlib%\errmsg.o > nul
7 copy      %genlib%\idents.o > nul
8 copy      %genlib%\main1.o > nul
9 copy      %genlib%\grts.o > nul
10
11 gentle   idei.g
12
13 echo      Bezeichner wie in Pascal oder Modula,      mit IDENT-Attribut:
14 reflex    Bezeich=id_i_pas.t
15 flex      gen.l
16 bison     gen.y
17 gcc       idei.c gen.tab.c lex.yy.c text_io.o errmsg.o idents.o main1.o grts.o
18 ren       a.exe idei_pas.exe
19 idei_pas  inpas.ide
20
21 echo      Bezeichner wie in Eiffel,                  mit IDENT-Attribut:
22 reflex    Bezeich=id_i_eif.t
23 flex      gen.l
24 bison     gen.y
25 gcc       idei.c gen.tab.c lex.yy.c text_io.o errmsg.o idents.o main1.o grts.o
26 ren       a.exe idei_eif.exe
27 idei_eif  ineif.ide
28
29 echo      Bezeichner wie in C,                        mit IDENT-Attribut:
30 reflex    Bezeich=id_i_ceh.t
31 flex      gen.l
32 bison     gen.y
33 gcc       idei.c gen.tab.c lex.yy.c text_io.o errmsg.o idents.o main1.o grts.o
34 ren       a.exe idei_ceh.exe
35 idei_ceh  inceh.ide
36
37 echo      Bezeichner wie in Ada,                      mit IDENT-Attribut:
38 reflex    Bezeich=id_i_ada.t
39 flex      gen.l
40 bison     gen.y
41 gcc       idei.c gen.tab.c lex.yy.c text_io.o errmsg.o idents.o main1.o grts.o
42 ren       a.exe idei_ada.exe
43 idei_ada  inada.ide
44 rem -----

```

**Zeile 14:** Bezeich-Lexeme sollen mithilfe des Unterprogramms erkannt werden, welches in der Datei `id_i_pas.t` beschrieben wird.

**Zeile 22:** Entsprechend.

**Zeile 30:** Entsprechend.

**Zeile 38:** Entsprechend.

**Anmerkung:** Den Umbenennungsbefehl `ren a.exe idei_pas.exe` in Zeile 18 kann man sich sparen, indem man "den richtigen Namen der ausführbaren Datei" schon in Zeile 17 festlegt, etwa so:

```
17 gcc -o idei_pas.exe idei.c gen.tab.c lex.yy.c ... main1.o grts.o
```

In diesem Kommando soll die Option `-o` an "name of output" erinnern.

Hier die Dateien `id_i_pas.t`, `id_i_eif.t`, `id_i_ceh.t` und `id_i_ada.t`:

```

1  /* ----- */
2  /* id_i_pas.t: Bezeichner (identifizier) wie in Pascal oder Modula. */
3  /* Liefert einen Wert vom Typ IDENT (d.h. einen Zeiger) */
4  /* ----- */
5  [A-Za-z][A-Za-z0-9]* {
6    long id;
7    string_to_id (yytext, &id);
8    yylval.attr[1] = id;          /* Wert vom Typ IDENT nach yyval.attr[1] */
9    yysetpos();
10   return Bezeich;              /* Dem Parser ein Token Bezeich melden */
11 }
12 /* ----- */

```

**Zeile 5:** Siehe oben Zeile 05 der Datei `id_s_pas.t`

**Zeile 6:** Ein Wert vom Typ `IDENT` wird intern als eine Ganzzahl vom Typ `long` dargestellt.

**Zeile 7:** Der vom Scanner erkannte Bezeichner in der Variablen `yytext` wird in einen Wert vom Typ `IDENT` umgewandelt und dieser Wert wird der Variablen `id` zugewiesen.

**Zeile 10:** Der Scanner liefert dem Parser ein `Bezeich`-Token. Der Ausgabe-Parameter dieses Tokens ist vom Typ `IDENT` und steht in `yylval.attr[1]`.

```

1  /* ----- */
2  /* id_i_eif.t: Bezeichner (identifizier) wie in Eiffel. */
3  /* Liefert einen Wert vom Typ IDENT (d.h. einen Zeiger) */
4  /* ----- */
5  [A-Za-z][A-Za-z0-9_]* {
6    long id;
7    string_to_id (yytext, &id);
8    yylval.attr[1] = id;          /* Wert vom Typ IDENT nach yyval.attr[1] */
9    yysetpos();
10   return Bezeich;              /* Dem Parser ein Token Bezeich melden */
11 }
12 /* ----- */

1  /* ----- */
2  /* id_i_ceh.t: Bezeichner (identifizier) wie in C. */
3  /* Liefert einen Wert vom Typ IDENT (d.h. einen Zeiger) */
4  /* ----- */
5  [A-Za-z_][A-Za-z0-9_]* {
6    long id;
7    string_to_id (yytext, &id);
8    yylval.attr[1] = id;          /* Wert vom Typ IDENT nach yyval.attr[1] */
9    yysetpos();
10   return Bezeich;              /* Dem Parser ein Token Bezeich melden */
11 }
12 /* ----- */

1  /* ----- */
2  /* id_i_ada.t: Bezeichner (identifizier) wie in Ada. */
3  /* Liefert einen Wert vom Typ IDENT (d.h. einen Zeiger) */
4  /* ----- */
5  [A-Za-z](_[A-Za-z0-9]|[A-Za-z0-9])* {
6    long id;
7    string_to_id (yytext, &id);
8    yylval.attr[1] = id;          /* Wert vom Typ IDENT nach yyval.attr[1] */
9    yysetpos();
10   return Bezeich;              /* Dem Parser ein Token Bezeich melden */
11 }

```

```
12 /* ----- */
```

Hier noch einmal die Datei `inpas.ide`, die auch zum Testen des Programms `idei_pas.exe` gedacht ist:

```
1 -- inpas.ide: Eingabedatei fuer ides_pas.exe und idei_pas.exe
2 anna berta
3 a7 b5 x1234
```

Und hier die entsprechende Ausgabe des Programms `idei_pas.exe`:

```
1 x1234 b5 a7 berta anna
2 Anzahl der Bezeichner: 5
```

So wie hier das Programm `idei_pas.exe` mit der Datei `inpas.ide` getestet wurde, kann man die Programme `idei_eif.exe`, `idei_ceh.exe` und `idei_ada.exe` mit den Dateien `ineif.ide`, `inceh.ide` bzw. `inada.ide` testen.

**Aufgabe 13.2.1.:** Erzeugen Sie die mithilfe der Stapeldatei `build_i.bat` die Programme `idei_pas.exe`, `idei_eif.exe`, `idei_ceh.exe` und `idei_ada.exe`. Rufen Sie jedes dieser Programme mehrfach auf und geben Sie gleich auf der Kommandozeile eine Folge von Bezeichnern an, z. B. so:

```
idei_pas "otto b1234 xxx"
idei_pas "123a emil"
idei_eif "A_B__C A123_b"
idei_ceh "_abc_ _ A_B"
idei_ada "a_b_c a__b"
```

Geben Sie bewußt auch "falsche Bezeichner" an und prüfen Sie, ob Sie eine entsprechende Fehlermeldung erhalten. Machen Sie sich auf diese "experimentelle Weise" genau damit vertraut, welches der vier Programme welche Bezeichner akzeptiert bzw. ablehnt.

### 13.3. int.g: Ganzzahl-Literale, natürlich mit einem INT-Parameter

Gleich nach Bezeichnern sind Ganzzahl-Literale die wichtigsten Viele-zu-Eins-Lexeme bzw. Viele-zu-Eins-Token, die es in Programmiersprachen gibt. Das Programm `int.g` liest eine Folge von Ganzzahl-Literalen ein und gibt ihre Werte und die Summe aller Werte aus.

```
1 -----
2 -- int.g: Dient zum Testen der Tokenbeschreibungs-Dateien ilit_pas.t und
3 -- ilit_ada.t etc.. Der hier spezifizierte Compiler liest eine Folge von
4 -- Ganzzahl-Literalen ein und gibt sie und die Summer ihrer Werte aus.
5 -----
6 'nonterm' GanzLit_Folge(->INT) -- Nichtleere Folge von Ganzzahl-Literalen
7   'rule' GanzLit_Folge(->N+M): GanzLit(->N) GanzLit_Folge(->M)
8     PutI(N) PutS(" ")
9   'rule' GanzLit_Folge(-> N): GanzLit(->N)
10     PutI(N) PutS(" ")
11
12 'token' GanzLit(->INT) -- Ein einzelnes Ganzzahl-Literal
13
14 'root'
15   GanzLit_Folge(->SUMME)
16   N1 PutS("Summe der Ganzzahlen: ") PutI(SUMME) N1
17
18 'action' PutS(STRING) -- Ist in der Datei text_io.c definiert
```

```

19 'action' PutI(INT)      -- Ist in der Datei text_io.c definiert
20 'action' N1           -- Ist in der Datei text_io.c definiert
21 -----

```

**Zeile 12:** Das Token-Prädikat `GanzLit` hat einen Ausgabe-Parameter vom Typ `INT`. Wie der Scanner `GanzLit-Lexeme` erkennen und wie er ihren `INT`-Parameter berechnen soll, wird nicht hier im Gentle-Programm, sondern in einer `.t`-Datei beschrieben (siehe unten).

Die folgende DOS-Stapeldatei `build.bat` erzeugt aus dem einen Gentle-Programm `int.g` zwei ausführbare Programme namens `int_pas.exe` und `int_ada.exe`, die sich durch die Art der Ganzzahl-Literale unterscheiden, die sie akzeptieren:

```

1 @echo off
2 rem -----
3 rem build.bat fuer TEST_INT
4 rem -----
5 copy %genlib%\text_io.o > nul
6 copy %genlib%\errmsg.o > nul
7 copy %genlib%\main1.o > nul
8 copy %genlib%\grts.o > nul
9
10 gentle int.g
11
12 echo Ganzzahl-Literale im Pascal-Stil:
13 reflex GanzLit=ilit_pas.t
14 flex gen.l
15 bison gen.y
16 gcc int.c gen.tab.c lex.yy.c text_io.o errmsg.o main1.o grts.o
17 ren a.exe int_pas.exe
18 int_pas inpas.int
19
20 echo Ganzzahl-Literale im Ada-Stil:
21 reflex GanzLit=ilit_ada.t
22 flex gen.l
23 bison gen.y
24 gcc int.c gen.tab.c lex.yy.c text_io.o errmsg.o main1.o grts.o
25 ren a.exe int_ada.exe
26 int_ada inada.int
27 rem -----

```

**Zeile 13:** Das Token-Prädikat `GanzLit` (welches in Zeile 12 des Programms `int.g` vereinbart wurde) soll durch das Unterprogramm realisiert werden, welches in der Datei `ilit_pas.t` beschrieben wird.

**Zeile 21:** Das Token-Prädikat `GanzLit` soll durch das Unterprogramm in der Datei `ilit_ada.t` realisiert werden.

Hier die beiden Dateien `ilit_pas.t` und `ilit_ada.t`:

```

1 /* ----- */
2 /* ilit_pas.t: beschreibt Ganzzahl-Literale, die (nur) aus dezimalen */
3 /* Ziffern bestehen duerfen. Vorzeichen, Unterstriche etc. sind NICHT */
4 /* erlaubt. Liefert den Wert des Literals als Ganzzahl vom Typ int. */
5 /* ----- */
6 [0-9]+ {
7     yylval.attr[1] = atoi (yytext);
8     yysetpos();
9     return GanzLit;

```

```

10 }
11 /* ----- */

```

**Zeile 7:** Mit der C-Funktion `atoi` ("ascii to int") wird die Zeichenkette in `yytext` in den entsprechenden `int`-Wert umgewandelt.

**Zeile 9:** Der Scanner liefert dem Parser ein `GanzLit`-Token. Der Ausgabe-Parameter dieses Tokens steht in `yylval.attr[1]`.

```

1 /* ----- */
2 /* ilit_ada.t: beschreibt Ganzzahl-Literale, die aus dezimalen Ziffern*/
3 /* und Unterstrichen bestehen duerfen (aehnlich wie in Ada). */
4 /* Ein Unterstrich darf aber nur zwischen zwei Ziffern stehen. */
5 /* Liefert den Wert des Literals als Ganzzahl vom Typ INT. */
6 /* ----- */
7 [0-9](_[0-9]|[0-9])* {
8     char *p = yytext;
9     int d = 0; /* Distanz, um die Zeichen verschoben werden */
10    while (*p != '\0') { /* Unterstriche "_" aus yytext entfernen */
11        if (*p == '_')
12            d++;
13        else
14            *(p-d) = *p; /* Zeichen um d Stellen nach links kopieren */
15        p++;
16    }
17    *(p-d) = '\0'; /* abschliessende Null nach yytext */
18
19    yylval.attr[1] = atoi (yytext);
20    yysetpos();
21    return GanzLit;
22 }
23 /* ----- */

```

**Zeile 7:** Dieser reguläre Ausdruck beschreibt, welche Lexeme als Ganzzahl-Literale erkannt werden sollen: Das erste Zeichen muss eine Ziffer zwischen 0 und 9 sein, danach dürfen null oder mehr Zeichengruppen kommen, von denen jede aus einem Unterstrich `_` und einer Ziffer zwischen 0 und 9 oder nur aus einer Ziffer zwischen 0 und 9 bestehen muss. Der Stern `*` bedeutet "null oder mehr", der senkrechte Strich `|` bedeutet "oder".

**Zeile 7 bis 21:** Dieser C-Block gibt an, was passieren soll, wenn ein Ganzzahl-Literal in der Eingabe erkannt wurde. Dieser Block besagt: Die Unterstriche sollen aus dem Literal entfernt und die derart "ge-reinigte Ziffernfolge" soll mit dem C-Unterprogramm `atoi` (ascii to integer) in einen internen Wert vom Typ `int` umgewandelt werden. Dieser Wert soll dem Token `GanzLit` als Parameter mitgegeben werden.

Hier die Datei `inpas.int`, mit der man das Programm `int_pas.exe` testen kann:

```

1 -- inpas.int: Eingabe fuer int_pas.exe
2 7 15 18 10

```

Und hier die entsprechende Ausgabe des Programms `int_pas.exe`:

```

1 10 18 15 7
2 Summe der Ganzzahlen: 50

```

Hier die Datei `inada.int`, mit der man das Programm `int_ada.exe` testen kann:

```

1 -- inada.int: Eingabe fuer int_ada.exe
2 7 1_5 1_8 1_0

```

Und hier die entsprechende Ausgabe des Programms `int_ada.exe`:

```

1 10 18 15 7
2 Summe der Ganzzahlen: 50

```

Man beachte, dass hier nicht die eingelesenen Ganzzahl-Literale ausgegeben werden, sondern nur ihre Werte. Praktisch bedeutet das: In der Ausgabe haben die Zahlen 10, 18, etc. keine Unterstriche. Werte sind etwas abstrakteres als Literale. Viele verschiedene Ganzzahl-Literale können den gleichen Wert haben, z. B. die Literale 10, 010, 0010, 1\_0 etc.

#### 14. calcul\_1.g: Ein Interpreter für Ausdrücke

Der folgende Interpreter `calcul_1.g` liest einen Ausdruck ein (z. B.  $1+1$  oder  $13+(24*35)/46-57$ ) und gibt seinen Wert (als Dezimalzahl) aus. Dabei wird ein Token-Prädikat namens `GanzLit` benutzt. Es bezeichnet ein Unterprogramm, welches bestimmte Ganzzahl-Literale erkennen kann. Das Unterprogramm selbst ist nicht in Gentle geschrieben (sondern als Spezifikation für den Scanner-Generator `flex`), und wird beim Binden des Gentle-Programms mit eingebunden (siehe unten).

```

1  -----1*****2*****3*****4*****5*****6*****7*****
2  -- calcul_1.g: Ein Interpreter fuer Ausdruecke.
3  -- Die Ausdruecke duerfen die folgenden Bestandteilen enthalten:
4  -- Dezimale Ganzzahl-Literale mit oder ohne Vorzeichen, die zweistelligen
5  -- Operationen +, -, * und / und (runde) Klammern. Beispiele:
6  -- 0, -1, +1580, 1+1, +123 + -27, 100+200*10/20, (100+200)*(10/20).
7  -- Ausgegeben wird jeweils der Wert des Ausdrucks
8  -----
9  'root' expr1(-> X) print(X)
10 -----
11 'nonterm' expr1(-> INT)
12   'rule' expr1(-> X): expr2(-> X)
13   'rule' expr1(-> X+Y): expr1(-> X) "+" expr2(-> Y) -- linksassoziativ
14   'rule' expr1(-> X-Y): expr1(-> X) "-" expr2(-> Y) -- linksassoziativ
15
16 'nonterm' expr2(-> INT)
17   'rule' expr2(-> X): expr3(-> X)
18   'rule' expr2(-> X*Y): expr2(-> X) "*" expr3(-> Y) -- linksassoziativ
19   'rule' expr2(-> X/Y): expr2(-> X) "/" expr3(-> Y) -- linksassoziativ
20
21 'nonterm' expr3(-> INT)
22   'rule' expr3(-> X): GanzLit(-> X)
23   'rule' expr3(-> - X): "-" expr3(-> X) -- Vorzeichen Minus
24   'rule' expr3(-> + X): "+" expr3(-> X) -- Vorzeichen Plus
25   'rule' expr3(-> X): "(" expr1(-> X) ")" -- Ausd. in Klammern
26
27 'token' GanzLit(-> INT) -- Wird naeher beschrieben in einer .t-Datei
28 -----1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 9:** Natürlich kann man mehrere Prädikataufrufe (predicate invocations) in eine Zeile schreiben. Aus den Vereinbarungen aller Zwischensymbol-Prädikate (nonterm predicates) und aller Token-Prädikate (token predicates) wird ein Scanner und ein Parser erzeugt. Der Aufruf des Prädikates `expr1` hier im Wurzelprädikat (root predicate) bewirkt, dass der Scanner eine Zeichenkette einliest und in eine Tokenfolge umwandelt, und dass der Parser versucht, für diese Tokenfolge eine Ableitung aus dem Zwischensymbol `expr1` zu finden. Wenn das gelingt, dann wird dabei auch der Ausgabeparameter des Zwischensymbols `expr1` berechnet und `X` genannt. Anschließend wird der Wert von `X` mit dem `print`-Prädikat ausgegeben.

**Zeile 11 bis 25:** Wenn man die Parameter in den Klammern außer Acht läßt, dann bleibt eine typische kontextfreie Grammatik für Ausdrücke übrig. Die Zwischensymbole dieser Grammatik heißen `expr1`, `expr2`, `expr3` und `GanzLit`. Zu den Endsymbolen der Grammatik gehören die Lexeme `+`, `-`, `*`, `/`, `(`, `)` und alle Zeichen, aus denen Ganzzahl-Literale bestehen dürfen. Die Grammatik drückt aus, dass die Vorzeichen `+` und `-` am stärksten binden (siehe Zeile 23 und 24), dass die Operationen `*` und `/` (siehe Zeile 18 und 19) stärker binden als die zweistelligen Operationen `+` und `-` (siehe Zeile 13 und 14), d.h. "Punktrechnung geht vor Strichrechnung", und dass die zweistelligen Operationen `+`, `-`, `*` und `/` linksassoziativ sind.

**Zeile 27:** Wenn der Scanner in der Eingabe ein Ganzzahlliteral erkennt, dann liefert er ein `GanzLit`-Token. Dieses `GanzLit`-Token hat einen Ausgabe-Parameter vom Typ `INT`, nämlich den Wert des betreffenden Ganzzahl-Literals. Welche Lexeme der Scanner als Ganzzahlliteral erkennen und wie er ihren Wert berechnen soll, wird nicht hier im Gentle-Programm beschrieben sondern in einer sogenannten `.t`-Datei. Diese Datei muss entweder `GanzLit.t` heißen, oder aber ihr Name muss dem `reflex`-Programm als Parameter mitgeteilt werden (siehe unten).

Hier die Datei `build.bat`, mit der man den Interpreter `calcul_1.exe` erzeugen lassen kann:

```

1 rem -----
2 rem build.bat fuer calcul_1
3 rem -----
4 gentle calcul_1.g
5 reflex GanzLit=ilit_ada.t
6 flex gen.l
7 bison gen.y
8
9 @copy %genlib%\errmsg.o > nul
10 @copy %genlib%\main1.o > nul
11 @copy %genlib%\grts.o > nul
12
13 gcc -o calcul_1.exe calcul_1.c lex.yy.c gen.tab.c errmsg.o main1.o grts.o
14
15 calcul_1 2+3*4
16 calcul_1 in1.clc
17 calcul_1 in2.clc
18 rem -----

```

**Zeile 5:** Dem `reflex`-Programm wird mitgeteilt, dass das Unterprogramm für das Token-Prädikat `GanzLit` in der Datei namens `ILitAda.t` beschrieben wird. Das `reflex`-Programm erzeugt eine `flex`-Spezifikation und überträgt die Informationen aus der Datei `ILitAda.t` dorthinein. Die anderen Zeilen der Datei `build.bat` enthalten nichts Neues. Die Datei `ILitAda.t` wurde schon oben im Abschnitt 11.3 behandelt.

#### Zusammenhänge:

1. Im Gentle-Programm `calcul_1.g` wird ein Token-Prädikat namens `GanzLit` vereinbart.
2. In der Datei `build.bat` wird dem `reflex`-Programm mitgeteilt, dass es nähere Angaben zu diesem Token-Prädikat in der Datei `ilit_ada.t` findet.
3. In der Datei `ilit_ada.t` wird die (sehr große) Menge aller Lexeme beschrieben, die vom Scanner in ein `GanzLit`-Token umgewandelt werden sollen. Der letzte Befehl im entsprechenden C-Block muss `return GanzLit;` lauten (siehe Zeile 21 der Datei `ilit_ada.t`).

Wenn Ihnen die Notation von Ada-Ganzzahl-Literalen nicht zusagt und Sie mit dem Interpreter `calcul_1.exe` lieber Ganzzahl-Literale im Pascal-Stil verarbeiten möchten (d.h. Ganzzahl-Literale, bei

denen Unterstriche zwischen den Ziffern nicht erlaubt sind), dann können Sie folgendermaßen vorgehen:

1. Kopieren Sie die Datei namens `ilit_pas.t` (z. B. aus dem Verzeichnis `test_int`) in das Verzeichnis `caclul_1`.
2. Ändern Sie in der Datei `build.bat` den Aufruf des `reflex`-Programms zu  
`reflex GanzLit=ilit_pas.t`
3. Rufen Sie die Stapeldatei `build.bat` (erneut) auf. Der dadurch erzeugte Interpreter namens `calcul_1.exe` wird dann nur noch Ganzzahl-Literale im Pascal-Stil akzeptieren und bei Unterstrichen innerhalb eines Ganzzahl-Literals einen Syntax-Fehler melden.

## 15. Eine einfache Stapelmaschine und ihre Maschinensprache

Ein Compiler übersetzt Programme seiner Quellsprache in entsprechende Programme seiner Zielsprache. Als Vorbereitung für das nächste Beispiel soll in diesem Abschnitt eine besonders einfache Zielsprache beschrieben werden. Es handelt sich dabei um die Maschinensprache einer sehr simplen Stapel-Maschine (stack machine). Eine solche Maschine besitzt einen Stapel als Speicher und eine Anzeige (die man sich wie die Anzeige eines Taschenrechners vorstellen kann). Die Stapel-Maschine kann folgende Befehle ausführen:

```
push 137 -- Die Zahl 137 wird als oberstes Element auf den Stapel gelegt.
pop      -- Die oberste Zahl auf dem Stapel wird vom Stapel entfernt
         -- ("weggeworfen").
add      -- Die obersten beiden Zahlen auf dem Stapel werden durch
         -- ihre Summe ersetzt.
sub      -- Die obersten beiden Zahlen auf dem Stapel werden durch
         -- ihre Differenz ersetzt.
         -- Dabei wird die oberste Zahl von der zweitobersten Zahl abgezogen.
mult     -- Die obersten beiden Zahlen auf dem Stapel werden durch
         -- ihr Produkt ersetzt.
div      -- Die obersten beiden Zahlen auf dem Stapel werden durch
         -- ihren Quotienten
         -- ersetzt. Dabei wird die zweitoberste Zahl durch
         -- die oberste Zahl geteilt.
put      -- Die oberste Zahl wird vom Stapel entfernt und
         -- in der Anzeige angezeigt.
```

**Beispiel 15.1.:** Das folgende Programm ist in der Sprache der Stapel-Maschine geschrieben. Es bewirkt, dass der Wert des Ausdrucks  $10 + 20 * 30 / 40$  berechnet und in der Anzeige der Maschine angezeigt wird:

**Befehl Operand Danach sieht der Stapel so aus ("links ist oben"):**

```
push 10      -- 10
push 20      -- 20 10
push 30      -- 30 20 10
mult         -- 600 10
push 40      -- 40 600 10
div         -- 15 10
add         -- 25
put         -- leer
```

**Aufgabe 15.1.:** Geben Sie ein Stapel-Programm an, welches den Wert des folgenden Ausdrucks berechnet:  $100 - 200 / 5 * (17 - 15)$ . Geben Sie für jeden Befehl ihres Programms an, wie der Stapel nach seiner Ausführung aussieht (wie im Beispiel 15.1.).

## 16. calcul\_2.g: Ein Compiler für Ausdrücke

Im Abschnitt 14. wurde ein Interpreter (calcul\_1.g) für bestimmte Ausdrücke angegeben. In diesem Beispiel calcul\_2.g soll ein kleiner Compiler vorgestellt werden, der die gleichen Ausdrücke in die Befehle der Stapelmaschine aus dem vorigen Abschnitt 15. übersetzt.

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- calcul_2.g: Ein Compiler, der Ausdruecke in die Sprache einer Stapel-
3  -- Maschine uebersetzt. Die Stapel-Maschine kennt folgende Anweisungen:
4  -- push ZAHL, add, sub, mult div und put. Die Anweisung add ersetzt die
5  -- obersten beiden Elemente des Stapels durch ihre Summe; sub, mult und div
6  -- funktionieren entsprechend; put gibt das oberste Stapel-Element aus.
7  -----
8  'type' OP      -- Rechen-Operationen
9      plus        -- entspricht der add-Anweisung
10     minus       -- entspricht der sub-Anweisung
11     mal         -- entspricht der mult-Anweisung
12     durch       -- entspricht der div-Anweisung
13
14 'type' ZAUS    -- Zwischendarstellung eines Ausdrucks
15     zahl(INT)
16     ausd(ZAUS, ZAUS, OP)
17 -----
18 'nonterm' expr1(-> ZAUS)
19     'rule' expr1(-> Z)                : expr2(-> Z)
20     'rule' expr1(-> ausd(Z1, Z2, plus )) : expr1(-> Z1) "+" expr2(-> Z2)
21     'rule' expr1(-> ausd(Z1, Z2, minus)) : expr1(-> Z1) "-" expr2(-> Z2)
22
23 'nonterm' expr2(-> ZAUS)
24     'rule' expr2(-> Z)                : expr3(-> Z)
25     'rule' expr2(-> ausd(Z1, Z2, mal )) : expr2(-> Z1) "*" expr3(-> Z2)
26     'rule' expr2(-> ausd(Z1, Z2, durch)) : expr2(-> Z1) "/" expr3(-> Z2)
27
28 'nonterm' expr3(-> ZAUS)
29     'rule' expr3(-> zahl(N))           : GanzLit(-> N)
30     'rule' expr3(-> ausd(zahl(0), Z, minus)) : "-" expr3(-> Z)
31     'rule' expr3(-> Z)                 : "+" expr3(-> Z)
32     'rule' expr3(-> Z)                 : "(" expr1(-> Z) ")"
33
34 'token' GanzLit(-> INT) -- Wird naeher beschrieben in einer .t-Datei
35 -----
36 'action' aus(ZAUS->)
37     'rule' aus(zahl(N))                : PutS("push") Tab PutI(N)      Nl
38     'rule' aus(ausd(Z1, Z2, plus))     : aus(Z1) aus(Z2) PutS("add" ) Nl
39     'rule' aus(ausd(Z1, Z2, minus))    : aus(Z1) aus(Z2) PutS("sub" ) Nl
40     'rule' aus(ausd(Z1, Z2, mal))      : aus(Z1) aus(Z2) PutS("mult") Nl
41     'rule' aus(ausd(Z1, Z2, durch))    : aus(Z1) aus(Z2) PutS("div" ) Nl
42 -----
43 -- Praedikate, die in der Datei text_io.c als C-Funktionen realisiert sind:
44 'action' PutS(STRING) -- Gibt einen String zur Standardausgabe aus
45 'action' PutI(INT)    -- Gibt eine Ganzzahl als Dezimalzahl aus
46 'action' Tab         -- Gibt ein Tab-Zeichen aus
47 'action' Nl          -- Gibt ein newline-Zeichen aus

```

```

48 -----
49 'root' expr1( -> Z) -- Lies einen Eindruck ein und erzeuge Zwischendarst.
50     aus (Z -> ) -- Uebersetze Zwischendarst. und gib aus
51     PutS("put") N1 -- Gib letzten Befehl aus
52 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 8:** OP ist ein ("Aufzählungs-") Typ mit vier Werten.

**Zeile 14:** Jeder Wert (oder: Term, oder: Baum) vom Typ ZAUS dient dazu, einen Ausdruck darzustellen. Z. B. wird der Ausdruck  $10 + 20 * 30 / 40$  durch folgenden Term dargestellt:

```
ausd(zahl(10), ausd(ausd(zahl(20), zahl(30), mal), zahl(40), durch), plus)
```

**Zeile 18:** Im Programm `calcul_1.g` hatte das Zwischensymbol `expr1` einen Ausgabeparameter vom Typ INT (nämlich den Wert des betreffenden Ausdrucks). Hier hat das Zwischensymbol `expr1` einen Ausgabeparameter vom Typ ZAUS, nämlich die Zwischendarstellung des betreffenden Ausdrucks.

**Zeile 23:** Entsprechendes gilt auch für das Zwischensymbol `expr2`.

**Zeile 28:** Entsprechendes gilt auch für das Zwischensymbol `expr3`.

**Zeile 36:** Das Aktions-Prädikat `aus` dient dazu, einen Term des Typs ZAUS in ein Programm der Stapelmaschine zu übersetzen und auszugeben. Nur der letzte Befehl des Programms (`put`) wird nicht durch das Prädikat `aus`, sondern direkt im Wurzelprädikat ausgegeben (siehe Zeile 51).

**Aufgabe 16.1.:** Erzeugen Sie das Programm `calcul_2.exe` (z. B. indem Sie die Stapeldatei `build.bat` im Verzeichnis `calcul_2` aufrufen) und testen Sie es mit möglichst vielen (richtigen und falschen) Eingaben.

### 17. calcul\_3.g: Ein Interpreter für erweiterte Ausdrücke

**Aufgabe 17.1.:** Erweitern Sie den Interpreter `calcul_1.g` zu einem Interpreter `calcul_3.g`, so dass in den zu interpretierenden Ausdrücken auch die Potenzfunktion (`**`) und eine Rest-Funktion (`rem` wie "remainder") vorkommen darf. Für die Rest-Funktion `rem` soll allgemein gelten:  $X \text{ rem } Y$  ist gleich  $X - ((X / Y) * Y)$  für beliebige Ganzzahlen  $X$  und  $Y$ . Konkret bedeutet das z. B.

```
17 rem 5 ist gleich 2
-17 rem 5 ist gleich -2
17 rem -5 ist gleich 2
-17 rem -5 ist gleich -2
```

Die Potenzfunktion `**` soll hier natürlich eine Ganzzahl als Ergebnis liefern (und nicht etwa eine mehr oder weniger ungenaue Bruchzahl). Hier ein kleiner Ausschnitt aus der Wertetabelle der Potenzfunktion ( $X$  hoch  $Y$ ) für Ganzzahlen:

|      |     | X=-4 | X=-3 | X=-2 | X=-1 | X= 0  | X=+1 | X=+2 | X=+3 | X=+4 |     |      |
|------|-----|------|------|------|------|-------|------|------|------|------|-----|------|
|      |     | ...  | ...  | ...  | ...  | ...   | ...  | ...  | ...  | ...  |     |      |
| Y=+4 | ... | +256 | +81  | +16  | +1   | 0     | +1   | +16  | +81  | +256 | ... | Y=+4 |
| Y=+3 | ... | -64  | -27  | -8   | -1   | 0     | +1   | +8   | +27  | +64  | ... | Y=+3 |
| Y=+2 | ... | +16  | +9   | +4   | +1   | 0     | +1   | +4   | +9   | +16  | ... | Y=+2 |
| Y=+1 | ... | -4   | -3   | -2   | -1   | 0     | +1   | +2   | +3   | +4   | ... | Y=+1 |
| Y= 0 | ... | +1   | +1   | +1   | +1   | undef | +1   | +1   | +1   | +1   | ... | Y= 0 |
| Y=-1 | ... | 0    | 0    | 0    | -1   | undef | +1   | 0    | 0    | 0    | ... | Y=-1 |
| Y=-2 | ... | 0    | 0    | 0    | +1   | undef | +1   | 0    | 0    | 0    | ... | Y=-2 |
| Y=-3 | ... | 0    | 0    | 0    | -1   | undef | +1   | 0    | 0    | 0    | ... | Y=-3 |
| Y=-4 | ... | 0    | 0    | 0    | +1   | undef | +1   | 0    | 0    | 0    | ... | Y=-4 |
|      |     | ...  | ...  | ...  | ...  | ...   | ...  | ...  | ...  | ...  |     |      |
|      |     | X=-4 | X=-3 | X=-2 | X=-1 | X= 0  | X=+1 | X=+2 | X=+3 | X=+4 |     |      |

Hier eine Lösung für die Aufgabe 17.1.:

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- calcul_3.g: ein Interpreter fuer Ausdruecke.
3  -- Die Ausdruecke duerfen folgende Bestandteile enthalten:
4  -- Dezimale Ganzzahlliterale mit oder ohne Vorzeichen, die zweistelligen
5  -- Operationen +, -, *, /, ** (Potenzierung!), rem (remainder, Rest)
6  -- und (runde) Klammern. Der Wert des Ausdrucks wird ausgegeben.
7  -----
8  'root' expr1( -> X)
9      PutI (X -> ) Nl
10 -----
11 'nonterm' expr1(-> Wert: INT)
12   'rule' expr1(-> X)           : expr2(-> X)
13   'rule' expr1(-> X+Y)        : expr1(-> X) "+" expr2(-> Y)
14   'rule' expr1(-> X-Y)        : expr1(-> X) "-" expr2(-> Y)
15
16 'nonterm' expr2(-> Wert: INT)
17   'rule' expr2(-> X)           : expr3(-> X)
18   'rule' expr2(-> X*Y)        : expr2(-> X) "*" expr3(-> Y)
19   'rule' expr2(-> X/Y)        : expr2(-> X) "/" expr3(-> Y)
20   'rule' expr2(-> X-((X/Y)*Y)): expr2(-> X) "rem" expr3(-> Y)
21
22 'nonterm' expr3(-> Wert: INT)
23   'rule' expr3(-> X)           : expr4(-> X)
24   'rule' expr3(-> Z)           : expr4(-> X) "***" expr3(-> Y)
25                                 potenz(X, Y -> Z)
26
27 'nonterm' expr4(-> Wert: INT)
28   'rule' expr4(-> X)           : GanzLit(-> X)
29   'rule' expr4(-> - X)         : "-" expr4(-> X)
30   'rule' expr4(-> + X)         : "+" expr4(-> X)
31   'rule' expr4(-> X)           : "(" expr1(-> X) ")"
32
33 'token' GanzLit(-> INT) -- Wird naeher beschrieben in einer .t-Datei
34 -----
35 'action' potenz(Basis:INT, Hochzahl:INT -> Ergebnis:INT)
36   'rule' potenz(X, Y -> 0) : eq(X, 0) le(Y, 0)
37                               PutS("Fehler beim Potenzieren: ")
38                               PutI(X) PutS(" hoch ") PutI(Y)
39                               PutS(" ist undefiniert!") Nl
40   'rule' potenz(X, Y -> 0) : eq(X, 0) ge(Y, +1)
41   'rule' potenz(X, Y -> +1) : ne(X, 0) eq(Y, 0)
42   'rule' potenz(X, Y -> 1/Z) : ne(X, 0) le(Y, -1)
43                               potenz(X, -Y -> Z)
44   'rule' potenz(X, Y -> X*Z) : ne(X, 0) ge(Y, +1)
45                               potenz(X, Y-1 -> Z)
46 -----
47 -- Die folgenden Praedikate wurden in C realisiert (siehe Datei text_io.c):
48 'action' PutS(STRING) -- Gibt eine Zeichenkette zur aktuellen Ausg. aus
49 'action' PutI(INT)    -- Gibt eine Ganzzahl als Dezimalzahl aus
50 'action' Nl           -- Gibt ein newline-Zeichen aus
51 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 20:** Die `rem`-Funktion kann man mithilfe der Funktionen `-`, `*` und `/` beschreiben. Man braucht also kein spezielles Prädikat dafür zu vereinbaren. Die `rem`-Funktion gehört üblicherweise zur "Punkt-

rechnung", d.h. das Operationszeichen "rem" bindet genauso stark wie \* und / und damit stärker als die "Strichrechnungen" + und -.

**Zeile 22:** Die Potenzfunktion \*\* soll stärker binden als die "Punktrechnungen" \*, / und rem, aber weniger stark als die Vorzeichen - und + (z. B. soll  $-2^{**3}$  soviel wie  $(-2)^{**4}$  und nicht etwa  $-(2^{**4})$  bedeuten). Deshalb wurde das Zwischensymbol `expr3` "umfunktioniert" und an seiner Stelle ein neues Zwischensymbol `expr4` eingeführt.

**Zeile 24 bis 25:** Diese Regel drückt aus, dass die Potenz-Funktion \*\* rechtsassoziativ sein soll, d.h.  $4^{**3}^{**2}$  bedeutet soviel wie  $4^{** (3^{**2})}$  und nicht etwa  $(4^{**3})^{**2}$ . Anders als die rem-Funktion kann man die Potenz-Funktion \*\* nicht direkt durch die Funktionen +, -, \* und / ausdrücken. Deshalb muss ein spezielles Prädikat vereinbart und hier aufgerufen werden. Dieses Prädikat wurde `potenz` genannt.

**Zeile 35:** Das Aktions-Prädikat `potenz` hat zwei Eingabeparameter vom Typ `INT` und einen Ausgabeparameter ebenfalls vom Typ `INT`.

**Zeile 36 bis 39:**  $X$  hoch  $Y$  ist undefiniert, wenn  $X$  gleich 0 und  $Y$  kleinergleich 0 ist. In diesem Fall wird zwar 0 als Ergebnis geliefert (obwohl es kein mathematisch korrektes Ergebnis gibt), aber vorher wird eine Fehlermeldung zur aktuellen Ausgabe (zum Bildschirm) ausgegeben.

**Zeile 40:**  $X$  hoch  $Y$  ist gleich 0, wenn  $X$  gleich 0 und  $Y$  größergleich +1 ist.

**Zeile 41:**  $X$  hoch  $Y$  ist gleich +1, wenn  $X$  ungleich 0 und  $Y$  gleich 0 ist.

**Zeile 42 bis 43:**  $X$  hoch  $Y$  ist gleich  $1/Z$ , wenn  $X$  ungleich 0 und  $Y$  kleinergleich -1 ist und  $Z$  gleich  $X$  hoch  $-Y$  ist. Man beachte, dass  $1/Z$  fast immer gleich 0 ist, weil / auch hier die Ganzzahldivision bezeichnet. Ausnahme: Wenn  $X$  gleich 1 und  $Y$  gleich -1 ist, dann ist  $Z$  gleich 1 hoch 1, d.h. gleich 1, und  $1/Z$  ist auch gleich 1.

**Zeile 44 bis 45:**  $X$  hoch  $Y$  ist gleich  $X * Z$ , wenn  $X$  ungleich 0 und  $Y$  größergleich +1 ist und  $Z$  gleich  $X$  hoch  $Y-1$  ist, d.h.  $X^Y$  ist gleich  $X * X^{Y-1}$ .

**Aufgabe 17.2.:** Erzeugen Sie das Programm `calcul_3.exe` (z. B. indem Sie die Stapeldatei `build.bat` im Verzeichnis `calcul_3` aufrufen) und testen Sie es mit möglichst vielen (richtigen und falschen) Eingaben.

## 18. calcul\_4.g: Polynome ableiten

Der Compiler `calcul_4.g` übersetzt keine "Quellprogramme" in "Zielprogramme", sondern Polynome in ihre erste Ableitung. Das Prädikat, mit dem ein Polynom abgeleitet wird, heißt hier `differ` (wie "differenzieren"). Das Programm `calcul_4.g` soll vor allem zeigen, wie kurz und elegant man dieses Prädikat `differ` in Gentle programmieren kann.

Die Polynome, die vom Compiler `calcul_4` eingelesen werden, dürfen nur eine Variable enthalten und die muss `x` heißen. Die Variable `x` darf positive und negative Exponenten haben (z. B.  $x^{**2}$ ,  $x^{**-3}$ ,  $x^{**+5}$ ,  $x^{**-1}$  etc.). Als Koeffizienten sind nur Ganzzahlen erlaubt, die "einfach so", ohne ein spezielles Zeichen für die Multiplikation, vor die Potenzen von `x` geschrieben werden (z. B.  $3x^{**2}$ ,  $-5x^{**3}$ ,  $-5x^{**-3}$  etc.).

| Beispiele: | Polynom                        | Erste Ableitung           |
|------------|--------------------------------|---------------------------|
|            | $x^{**3}$                      | $3x^{**2}$                |
|            | $5x^{**4} - 3x^{**2} + 5x - 7$ | $20x^{**3} - 6x + 5$      |
|            | $7x^{**-2} + 3x^{**4} - 27$    | $-14x^{**-3} + 12x^{**3}$ |

Vielleicht möchte der Leser zuerst selbst probieren, einen Compiler zu schreiben, der Polynome in ihre erste Ableitung übersetzt? Eine wichtige Teilaufgabe besteht darin, eine effiziente Zwischendarstellung für Polynome zu entwerfen.

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- calcul_4.g: Compiler, der Polynome in ihre erste Ableitung uebersetzt.
3  -- Beispiel: Das Polynom  $-5x^{**3} + 7x^{**2} - 6x + 13$  wird uebersetzt in seine
4  -- erste Ableitung  $-15x^{**2} + 14x - 6$ .
5  -----
6  'type' Z_Poly -- Zwischendarstellungen fuer Polynome
7      nix
8      poly(Faktor: INT, Hochzahl: INT, Rest: Z_Poly)
9
10 -- Beispiel: Das Polynom  $+3x^{**5} - 4x + 17$  hat folgende Zwischendarstellung:
11 -- poly(3, 5, poly(-4, 1, poly(17, 0, nix)))
12 -----
13 -- Ein Polynom einlesen und in die Zwischendarstellung uebersetzen:
14
15 'nonterm' Poly(->Zwischendarstellung: Z_Poly)
16     Poly(-> poly(F, H, R )): Nom(->F, H) Poly(-> R). -- Mehrere Nome.
17     Poly(-> poly(F, H, nix)): Nom(->F, H). -- Nur ein Nom.
18
19 'nonterm' Nom(->Faktor: INT, Hochzahl: INT)
20     Nom(-> F, H) : Zahl(-> F) "x" "*" Zahl(->H). -- z. B.  $-7x^{**+3}$ 
21     Nom(-> 1, H) : "x" "*" Zahl(->H). -- z. B.  $x^{**+3}$ 
22     Nom(-> F, 1) : Zahl(-> F) "x". -- z. B.  $-7x$ 
23     Nom(-> F, 0) : Zahl(-> F). -- z. B.  $-7$ 
24
25 'nonterm' Zahl(->INT)
26     Zahl(-> -N): "-" GanzLit(-> N). -- Ein Ganzzahlliteral mit "-" davor
27     Zahl(-> N): "+" GanzLit(-> N). -- Ein Ganzzahlliteral mit "+" davor
28     Zahl(-> N): GanzLit(-> N). -- Ein Ganzzahlliteral ohne Vorzeichen
29     Zahl(-> 1): "+". -- +x bedeutet soviel wie +1x
30     Zahl(-> -1): "-". -- -x bedeutet soviel wie -1x
31
32 'token' GanzLit(-> INT) -- Wird durch eine .t-Datei naeher beschrieben
33 -----
34 -- Ein Polynom in Zwischendarstellung differenzieren:
35
36 'action' differ(Z_Poly -> Z_Poly)
37     differ(nix -> nix): .
38     differ(poly(F, H, R1) -> poly(F*H, H-1, R2)): differ(R1 -> R2).
39 -----
40 -- Ein Polynom in Zwischendarstellung ausgeben:
41
42 'action' aus(Zwischendarstellung: Z_Poly)
43     aus(nix) : Nl.
44     aus(poly(F, H, R)): eq(F, 0) aus(R).
45     aus(poly(F, H, R)): eq(H, 0) Put(F) PutS(" ") aus(R).
46     aus(poly(F, H, R)): eq(H, 1) Put(F) PutS("x") PutS(" ") aus(R).
47     aus(poly(F, H, R)): Put(F) PutS("x**") PutI(H) PutS(" ") aus(R).
48
49 'action' Put(INT) -- Gibt ausdruecklich auch ein Vorzeichen aus
50     Put(N): lt(N, 0) PutS("-") PutI(-N).
51     Put(N): ge(N, 0) PutS("+") PutI(N).
52 -----
53 -- Ausgabe-Aktionen, die in C realisiert wurden (siehe Datei text_io.c)

```

```

54 'action' PutS(String) -- Gibt eine Zeichenkette zur aktuellen Ausgabe aus
55 'action' PutI(Int) -- Gibt eine Ganzzahl zur aktuellen Ausgabe aus
56 'action' Nl -- Gibt ein newline-Zeichen zur aktuellen Ausgabe aus
57 -----
58 -- Das "Hauptprogramm" des Compilers calcul_4:
59
60 'root'
61 Poly ( -> ZP1) -- Ein Polynom einlesen und in ZP1 umwandeln
62 aus (ZP1 -> ) -- Das Polynom ZP1 wieder ausgeben
63 differ(ZP1 -> ZP2) -- Das Polynom ZP1 ableiten zu ZP2
64 aus (ZP2 -> ) -- Die Ableitung ZP2 ausgeben
65 -----1-----2-----3-----4-----5-----6-----7-----

```

In diesem Programm und dem Kommentar dazu wird eine Potenz der Variablen  $x$  (zusammen mit einem Faktor davor, falls vorhanden) als ein Nom bezeichnet. Ein Polynom besteht natürlich aus einem oder mehreren Nomen.

Beispiele für Nome:  $-3x^{**}+2$ ,  $x^{**}3$ ,  $+5x$ ,  $x$ ,  $-3$

Man beachte: In einem Gentle-Programm kann man die Regeln (rules) der Prädikate wahlweise mit dem Schlüsselwort 'rule' beginnen oder mit einem Punkt . abschließen. Im Programm calcul\_4.g wurden alle Regeln mit einem abschließenden Punkt . und ohne das Schlüsselwort 'rule' notiert.

**Zeile 6:** In der hier verwendeten Zwischendarstellung ist ein Polynom entweder leer (nix) oder es besteht aus einem Faktor, einer Hochzahl und einem Rest. Der Rest ist wieder ein Polynom in Zwischendarstellung. Die Variable  $x$  taucht in der Zwischendarstellung nicht auf, da sie "selbstverständlich" ist.

**Achtung:** Diese Zwischendarstellung für Polynome garantiert *nicht*, dass jede Potenz von  $x$  nur einmal vorkommt oder dass die Potenzen "in einer vernünftigen Reihenfolge" angeordnet sind. Z. B. ist der Z\_Poly-Term

poly(2, 3, poly(5, 1, poly(4, 3, nix))) eine Zwischendarstellung des Polynoms  $2x^{**}3 + 5x + 4x^{**}3$ . Eine "normiertere" Zwischendarstellung des gleichen Polynoms wäre poly(6, 3, poly(5, 1, nix)).

**Zeile 16:** Ein Poly besteht aus einem Nom gefolgt von einem Poly oder

**Zeile 17:** nur aus einem Nom.

**Zeile 20:** Ein Nom mit Faktor und Hochzahl.

**Zeile 21:** Ein Nom ohne Faktor, aber mit Hochzahl entspricht einem Nom mit Faktor 1.

**Zeile 22:** Ein Nom ohne Faktor und ohne Hochzahl entspricht einem Nom mit Faktor 1 und Hochzahl 1.

**Zeile 23:** Ein Nom ohne die Variable  $x$  entspricht einem Nom mit  $x^{**}0$ .

**Zeile 26 bis 30:** Auch eine Zahl kann man auf verschiedene Weise notieren (mit Vorzeichen, ohne Vorzeichen etc.).

**Zeile 32:** Welche Lexeme (Zeichenfolgen im Quellprogramm) als Ganzzahl-Literale erkannt werden sollen, wird (aus Gründen der Effizienz) nicht hier im Gentle-Programm beschrieben, sondern in einer Spezifikation für den Scanner-Generator flex (d.h. in einer sogenannten .t-Datei). Immer wenn der Scanner ein solches Lexem erkennt, liefert er ein GanzLit-Token und einen INT-Wert an den Parser. Der INT-Wert ist natürlich der Wert des erkannten Ganzzahl-Literals.

**Zeile 36 bis 38:** Mit dem Aktions-Prädikat `differ` kann man Polynome (in Zwischendarstellung) ableiten "differenzieren"). Dazu genügen zwei kurze (und hoffentlich leicht zu verstehende) Regeln.

**Aufgabe 18.1.:** Erzeugen Sie das Programm `calcul_4.exe` (z. B. indem Sie die Stapeldatei `build.bat` im Verzeichnis `calcul_4` aufrufen) und testen Sie es mit möglichst vielen (richtigen und falschen) Eingaben.

## 19. Grundterme, Terme und Musterabgleich (pattern matching)

In diesem Abschnitt werden die folgenden beiden Typen als konkrete Beispiele benutzt:

```
1 'type' FARBE rot blau gruen
2 'type' BAUM
3   leer
4   b(Vorn: FARBE, Hinten: FARBE, Links: BAUM, Rechts: BAUM)
```

Ein Baum ist hier also entweder `leer`, oder er besteht aus einem Knoten mit zwei Farben (Vorn und Hinten) und zwei Unterbäumen (Links und Rechts).

**Zur Erinnerung:** In Gentle müssen Funktoren wie `rot`, `blau`, `gruen`, `leer` und `b` mit einem kleinen Buchstaben beginnen. Dagegen müssen Variablen-Namen mit einem großen Buchstaben beginnen. In einem Gentle-Programm kann man den Typ eines Variablen-Namens immer eindeutig aus dem Zusammenhang schließen, in dem er vorkommt. In diesem Abschnitt sollen `F`, `F1`, `F2` etc. immer Variablen vom Typ `FARBE` und `B`, `B1`, `B2` etc. Variablen vom Typ `BAUM` sein.

Ein Term ist ein Ding, welches nach den üblichen Regeln aus Funktoren, Klammern und Variablen-Namen gebildet wurde. Jeder Term gehört zu einem bestimmten Typ.

**Beispiel 19.1.:** Einige Terme des Typs `BAUM`:

```
1   leer
2   b(blau, rot, leer, leer)
3   b(rot, rot, b(blau, rot, leer leer), leer)
4   B
5   b(rot, blau, B, leer)
6   b(F, gruen, leer, B)
7   b(F1, F2, B1, B2)
8   b(rot, rot, b(F1, F2, leer, B1), b(rot, F3, B2, B3))
```

Man beachte, dass eine Variable vom Typ `BAUM` auch als Term vom Typ `BAUM` gilt (siehe 4.).

Einen Term, in dem keine Variable vorkommt, bezeichnet man auch als *Grundterm*. Im Beispiel oben sind 1. bis 3. Grundterme, die anderen Terme 4. bis 8. enthalten dargegebene Variablen und sind somit keine Grundterme.

Sei `T` ein Term. Wenn man in `T` einige (0 oder mehr) Variable durch einen Term des entsprechenden Typs ersetzt, erhält man einen Spezialfall von `T`. Wenn man in `T` jede Variable durch einen Grundterm des entsprechenden Typs ersetzt, erhält man einen *Grundspezialfall* von `T`. Die Spezialfälle eines Terms wurden hier nur erwähnt, um "ein tieferes Verständnis" zu erleichtern. Im folgenden werden wir uns ausschließlich mit den Grundspezialfällen eines Terms befassen.

**Beispiel 19.2.:**

1. Der Term `b(rot, F, leer, leer)` hat genau drei Grundspezialfälle, nämlich die Terme `b(rot, rot, leer, leer)`, `b(rot, blau, leer, leer)` und `b(rot, gruen, leer, leer)`.

2. Zu den Grundspezialfällen des Terms  $b(F1, F2, \text{leer}, \text{leer})$  gehören unter anderem die Terme  $b(\text{rot}, \text{rot}, \text{leer}, \text{leer})$ ,  $b(\text{blau}, \text{gruen}, \text{leer}, \text{leer})$  und  $b(\text{gruen}, \text{gruen}, \text{leer}, \text{leer})$ . Insgesamt hat der Term  $b(F1, F2, \text{leer}, \text{leer})$  genau neun Grundspezialfälle.

3. Von dem Term  $b(\text{rot}, \text{rot}, B, \text{leer})$  gibt es unendlich viele Grundspezialfälle. Dazu gehören die drei Terme  $b(\text{rot}, \text{rot}, \text{leer}, \text{leer})$ ,  $b(\text{rot}, \text{rot}, b(\text{rot}, \text{rot}, \text{leer}, \text{leer}), \text{leer})$  und  $b(\text{rot}, \text{rot}, b(\text{blau}, \text{blau}, b(\text{gruen}, \text{gruen}, \text{leer}, \text{leer}), b(\text{rot}, \text{rot}, \text{leer}, \text{leer})), \text{leer})$ .

Als Grenzfall gilt: Zu einem Grundterm  $G$  gibt es immer genau *einen* Grundspezialfall, nämlich  $G$  selbst.

### Aufgabe 19.1.:

1. Wieviele Grundspezialfälle hat der Term  $b(F, \text{blau}, \text{leer}, B)$ ?
2. Geben Sie drei dieser Grundspezialfälle an.

**Aufgabe 19.2.:** Geben Sie alle Grundspezialfälle des Terms  $b(\text{gruen}, \text{blau}, \text{leer}, b(\text{rot}, \text{rot}, \text{leer}, \text{leer}))$  an.

**Fachbegriff:** Manchmal sagen wir auch: Ein Term beschreibt die Menge seiner Grundspezialfälle. Ist  $T$  ein Term, dann soll  $GT(T)$  die Menge seiner Grundspezialfälle bezeichnen. Der Term  $T$  beschreibt also die Menge  $GT(T)$ .

### Aufgabe 19.3.:

1. Geben Sie einen Term an, der alle nicht-leeren Bäume beschreibt (d.h. der alle Grundterme vom Typ BAUM beschreibt, die ungleich leer sind).
2. Geben Sie einen Term an, der alle "Bäume der Tiefe 1" beschreibt. Das sind die Bäume, die aus genau einem Knoten bestehen. Da jeder Knoten zwei Farben enthält, enthält auch ein Baum der Tiefe 1 genau zwei Farben.

Angenommen, wir haben einen Term  $T$  und einen Grundterm  $G$ . Dann kann es interessant sein zu fragen: Ist  $G$  ein Grundspezialfall von  $T$ ? Kurz:  $G \in GT(T)$ ?

**Beispiel 19.3.:** Ist  $G$  ein Grundspezialfall von  $T$ ?

1. Der Grundterm  $b(\text{rot}, \text{gruen}, \text{leer}, \text{leer})$  ist ein Grundspezialfall des Terms  $b(\text{rot}, F, \text{leer}, \text{leer})$ . Denn wenn man in dem Term  $b(\text{rot}, F, \text{leer}, \text{leer})$  die Variable  $F$  durch den Grundterm  $\text{gruen}$  ersetzt, erhält man den Grundterm  $b(\text{rot}, \text{gruen}, \text{leer}, \text{leer})$ .
2. Sei  $G$  der Grundterm  $b(\text{blau}, \text{gruen}, \text{leer}, \text{leer})$  und  $T$  der Term  $b(\text{rot}, F, \text{leer}, \text{leer})$ .  $G$  ist kein Grundspezialfall von  $T$ , denn egal wodurch man im Term  $T$  die Variable  $F$  ersetzt, man erhält nie den Grundterm  $G$ .
3. Sei  $G$  der Grundterm  $b(\text{blau}, \text{gruen}, \text{leer}, \text{leer})$  und sei  $T$  der Term  $b(F1, F2, \text{leer}, \text{leer})$ . Dann gilt  $G \in GT(T)$ , denn wenn man in  $T$  die Variablen  $F1$  durch  $\text{blau}$  und  $F2$  durch  $\text{gruen}$  ersetzt, erhält man  $G$ .

Eine Variablenbelegung für einen Term  $T$  gibt für jede Variable von  $T$  an, durch welchen Term man sie ersetzen ("belegen") soll. Ein Grundterm  $G$  ist also genau dann ein Grundspezialfall von  $T$ , wenn es eine Variablenbelegung gibt, die den Term  $T$  in den Grundterm  $G$  überführt.

In jedes Gentle-Programm wird (vom Gentle-Compiler) ein Algorithmus eingebaut, der einen Term  $T$  und einen Grundterm  $G$  als Eingabe hat. Dieser Algorithmus stellt fest, ob  $G$  ein Grundspezialfall von  $T$  ist und wenn ja, dann berechnet er die entsprechende Variablenbelegung. Dieser Algorithmus wird auch als Unifizierungs-Algorithmus bezeichnet, weil er versucht, zwei Terme zu unifizieren ("gleich zu machen").

Im Zusammenhang mit dem Unifizierungs-Algorithmus bezeichnet man den Term  $T$  bisweilen auch als Muster (pattern) und den Grundterm  $G$  als Wert und spricht von einem Musterabgleich (pattern matching): Es wird geprüft, ob der Wert  $G$  dem Muster  $T$  entspricht (oder andersherum: ob das Muster  $T$  auf den Wert  $G$  paßt) und wenn ja, wird eine entsprechende Variablenbelegung berechnet.

#### Beispiel 19.4.: Musterabgleich

1. Das Muster  $b(\text{rot}, F, \text{leer}, \text{leer})$  paßt auf den Wert  $b(\text{rot}, \text{gruen}, \text{leer}, \text{leer})$ . Als Variablenbelegung ergibt sich: "Ersetze  $F$  durch  $\text{gruen}$ ".
2. Das Muster  $b(\text{rot}, F, \text{leer}, \text{leer})$  paßt nicht auf den Wert  $b(\text{blau}, \text{gruen}, \text{leer}, \text{leer})$ .
3. Das Muster  $b(F1, F2, \text{leer}, \text{leer})$  paßt auf den Wert  $b(\text{blau}, \text{gruen}, \text{leer}, \text{leer})$ . Als Variablenbelegung ergibt sich: "Ersetze  $F1$  durch  $\text{blau}$  und  $F2$  durch  $\text{gruen}$ ".

## 20. Grundlegendes über Prädikate

Ein Prädikat in Gentle hat eine gewisse Ähnlichkeit mit einer Prozedur in konventionellen Sprachen wie Pascal, Ada oder C:

1. Ein Prädikat muss vereinbart werden und kann dann beliebig oft aufgerufen werden.
2. Ein Prädikat hat (null oder mehr) formale Eingabeparameter und (null oder mehr) formale Ausgabeparameter. Jedem Parameter ist ein Typ zugeordnet.
3. Wenn man ein Prädikat aufruft, dann muss man für jeden Eingabeparameter einen Ausdruck des betreffenden Typs und für jeden Ausgabeparameter eine Variable (oder ein komplizierteres Muster) des betreffenden Typs als aktuellen Parameter angeben.
4. Häufig (aber nicht immer) ruft man ein Prädikat auf, damit aus bestimmten Werten für die Eingabeparameter die entsprechenden Werte der Ausgabeparameter berechnet werden.

Gleichzeitig hat ein Prädikat auch eine gewisse Ähnlichkeit mit einer Funktion in konventionellen Sprachen wie Pascal, Ada oder C, und zwar mit einer Funktion, die einen Wahrheitswert (`true` oder `false`) als Ergebnis liefert. Man sagt auch: Ein Aufruf eines Prädikates kann gelingen (das entspricht einem Funktionsaufruf, der den Wert `true` liefert) bzw. misslingen (das entspricht einem Funktionsaufruf, der den Wert `false` liefert). Englisch: An invocation of a predicate either succeeds or fails. Manchmal ruft man ein Prädikat nur auf, um "zu sehen", ob der Aufruf gelingt oder misslingt (und nicht, um aus den Werten der Eingabeparameter entsprechende Werte der Ausgabeparameter berechnen zu lassen). Ob ein bestimmter Aufruf eines Prädikates gelingt oder misslingt, hängt natürlich (in aller Regel) von den aktuellen Eingabeparametern des Aufrufs ab.

In Gentle unterscheidet man 6 Arten von Prädikaten:

1. Zwischensymbol-Prädikate (nonterm predicates)
2. Token-Prädikate (token predicates)

- |    |                             |                        |
|----|-----------------------------|------------------------|
| 3. | <b>Aktions-Prädikate</b>    | (action predicates)    |
| 4. | <b>Bedingungs-Prädikate</b> | (condition predicates) |
| 5. | <b>Feger-Prädikate</b>      | (sweep predicates)     |
| 6. | <b>Auswahl-Prädikate</b>    | (choice predicates)    |

In diesem Abschnitt werden nur die beiden grundlegenden Arten von Prädikaten (Aktions- und Bedingungs-Prädikate) behandelt. Die Besonderheiten der anderen vier Arten werden in anderen Abschnitten erläutert.

Die Aufrufe eines Aktions-Prädikates sollten immer gelingen. Wenn ein Aufruf eines Aktions-Prädikates doch einmal misslingt, dann gilt das als schwerwiegender Programmfehler und das betreffende Gentle-Programm wird sofort mit einer entsprechenden Fehlermeldung ("no rule applicable in line ...") abgebrochen.

Wenn dagegen ein Aufruf eines Bedingungs-Prädikates misslingt, dann ist das im allgemeinen "völlig normal" und entspricht in etwa der Situation, dass in einem konventionellen Programm die Bedingung einer if-Anweisung den Wert false hat.

Indem der Programmierer ein Prädikat als Aktions-Prädikat vereinbart, drückt er also aus, dass (seiner Ansicht und Absicht nach) alle Aufrufe dieses Prädikates gelingen werden. Vereinbart er ein Prädikat dagegen als Bedingungs-Prädikat, dann drückt er damit aus, dass er mit dem Mißlingen bestimmter Aufrufe dieses Prädikates rechnet.

Als Beispiele werden im folgenden einige Prädikate vereinbart, mit denen man Listen von Ganzzahlen bearbeiten kann. Die folgende Typvereinbarung wird in allen Beispielen vorausgesetzt und benützt:

```
1 'type' LISTE
2   leer
3   list(Element: INT, Rest: LISTE)
```

**Beispiele für Werte vom Typ LISTE:**

```
leer                -- Eine Liste mit null Elementen
list(25, leer)      -- Eine Liste mit einem Element (25)
list(-17, list(25, leer)) -- Eine Liste mit zwei Elementen (-17 und 25)
list(3, list(-17, list(25))) -- Eine Liste mit drei Elementen (3, -17 und 25)
```

Besonders wichtig sind in diesem Abschnitt die Begriffe *Wert* (value), *Ausdruck* (expression) und *Muster* (pattern). Statt einer vollständigen und formalen Definition hier eine unvollständige und informelle Erläuterung, die "den Sinn" dieser Begriffe deutlich machen soll.

Ausdrücke und Muster sind *syntaktische* Gebilde, d.h. sie können im Text eines Gentle-Programms vorkommen. Werte sind dagegen semantische Gebilde, d.h. sie werden erst während der Ausführung eines Programms vom Ausführer berechnet. Werte können deshalb grundsätzlich nicht in einem Programmtext vorkommen, sie können dort nur durch entsprechende Ausdrücke beschrieben werden. Ein Wert kann keine Variablen-Namen und keine Funktions-Namen wie +, -, \* oder / enthalten, sondern muss "vollständig ausgerechnet" sein. Ein Ausdruck kann (muss aber nicht) Funktions-Namen und Variablen-Namen enthalten. Ein Muster kann keine Funktions-Namen, wohl aber Variablen-Namen enthalten.

Ausdrücke und Muster können völlig gleich aussehen. Häufig folgt nur aus dem jeweiligen Zusammenhang, ob eine bestimmte Zeichenkette in einem Gentle-Programm ein Ausdruck ist oder ein Muster. Z. B. kann die Zeichenkette `list(G1, leer)` ein Muster oder ein Ausdruck sein, je nach Zusammen-

hang. Trotzdem sind Ausdrücke und Muster völlig verschiedene (in einem gewissen Sinne sogar "gegenteilige" oder "komplementäre") Gebilde.

Ist `list(G1, leer)` ein Ausdruck, dann muss für die Variable `G1` schon ein Wert festgelegt worden sein. Der Ausdruck `list(G1, leer)` gehört zum Typ `LISTE` und dient dazu, einen bestimmten Wert von diesem Typ zu beschreiben (siehe unten Beispiel 20.1.).

Ist `list(G1, leer)` dagegen ein Muster, dann darf für die Variable `G1` noch kein Wert festgelegt worden sein. Vielmehr dient das Muster dann dazu, für die Variable `G1` mithilfe eines Musterabgleichs einen Wert festzulegen (oder: zu definieren). An einem Musterabgleich ist außer dem Muster auch ein Wert beteiligt. Muster und Wert müssen vom selben Typ sein (z. B. vom Typ `LISTE`). Ein Musterabgleich ist nur zwischen bestimmten Mustern und Werten eines Typs möglich, zwischen anderen Mustern und Werten ist ein Abgleich nicht möglich (siehe unten Beispiel 20.2.).

#### Beispiel 20.1.: Der Wert eines Ausdrucks

1. Wenn die Variable `G1` den Wert `-17` hat, dann beschreibt (oder: hat) der Ausdruck `list(G1, leer)` den Wert `list(-17, leer)`.
2. Wenn die Variable `G1` den Wert `+23` hat, dann beschreibt der Ausdruck `list(G1, leer)` den Wert `list(+25, leer)`.
3. Wenn die Variable `G1` den Wert `-17` und die Variable `R1` den Wert `list(+25, leer)` hat, dann hat der Ausdruck `list(G1, list(G1, R1))` den Wert `list(-17, list(-17, list(+25, leer)))`.
4. Wenn die Variable `L` den Wert `list(-17, leer)` hat, dann beschreibt der Ausdruck `L` natürlich diesen Wert `list(-17, leer)`.
5. Der Ausdruck `list(-17, leer)` beschreibt (unabhängig vom Wert irgendwelcher Variablen) den Wert `list(-17, leer)`. Trotzdem gilt: Der Ausdruck `list(-17, leer)` ist eine Zeichenkette, die in einem Gentle-Programm vorkommen kann. Dagegen kann man sich unter dem Wert `list(-17, leer)` konkret eine Bitkette im Speicher eines Rechners oder auf einer Datenleitung vorstellen. Eine solche Bitkette kann nicht in einem Gentle-Programm vorkommen.

#### Beispiel 20.2: Musterabgleich (pattern matching):

1. Ein Abgleich zwischen dem Muster `list(G1, leer)` und dem Wert `list(-17, leer)` ist möglich und legt für die Variable `G1` den Wert `-17` fest.
2. Ein Abgleich zwischen dem Muster `list(G1, R1)` und dem Wert `list(-17, leer)` ist möglich und legt für die Variable `G1` den Wert `-17` und für die Variable `R1` den Wert `leer` fest.
3. Ein Abgleich zwischen dem Muster `list(G1, R1)` und dem Wert `list(1, list(2, list(3, leer)))` ist möglich und legt für `G1` den Wert `1` und für `R1` den Wert `list(2, list(3, leer))` fest.
4. Ein Abgleich zwischen dem Muster `list(G1, R1)` und dem Wert `leer` ist nicht möglich. Durch den Versuch eines solchen Abgleichs werden keine Werte für die Variablen `G1` und `R1` festgelegt.
5. Ein Abgleich zwischen dem Muster `L` und dem Wert `list(-17, leer)` legt für die Variable `L` natürlich den Wert `list(-17, leer)` fest. Offenbar kann das Muster `L` mit jedem Wert (des betreffenden Typs) abgeglichen werden.

6. Ein Abgleich zwischen dem Muster `list(-17, leer)` und dem Wert `list(-17, leer)` ist möglich, legt aber für keine Variable einen Wert fest (weil in dem Muster `list(-17, leer)` kein Variablenname vorkommt).

7. Ein Abgleich zwischen dem Muster `list(-17, leer)` und dem Wert `list(-16, leer)` ist nicht möglich (und legt natürlich auch für keine Variable einen Wert fest).

Jetzt sollen die Begriffe *Wert*, *Ausdruck* und *Muster* anhand eines konkreten Prädikates genauer (und hoffentlich etwas anschaulicher) erläutert werden. Hier die Vereinbarung des Prädikates:

```
4 'action'  Vertausche(Vorher: LISTE -> Nachher: LISTE)
5   'rule'  Vertausche(list(E1, list(E2, R2)) -> list(E2, list(E1, R2))): .
6   'rule'  Vertausche(  L                               ->                               L                               ): .
```

In Zeile 4 wird festgelegt, dass `Vertausche` ein Aktions-Prädikat ist und zwei Parameter hat: Einen Eingabeparameter namens `Vorher` vom Typ `LISTE` und einen Ausgabeparameter namens `Nachher`, ebenfalls vom Typ `LISTE`. Der Pfeil `->` dient dazu, die Eingabeparameter von den Ausgabeparametern zu trennen. Ein Aktions- oder Bedingungs-Prädikat kann beliebig viele Ein- und beliebig viele Ausgabeparameter haben (das schließt auch die Anzahl 0 ein).

Das Prädikat `Vertausche` soll in seinem Eingabeparameter (der ja eine Liste von Ganzzahlen ist) die ersten beiden Elemente miteinander vertauschen und das Ergebnis als Wert seines Ausgabeparameters festlegen (siehe Zeile 5). Eine solche Vertauschung ist natürlich nur möglich, wenn die `Vorher`-Liste mindestens zwei Elemente hat. Falls das nicht der Fall ist (d.h. falls die `Vorher`-Liste leer ist oder nur ein Element enthält), wird die `Vorher`-Liste unverändert als Wert der `Nachher`-Liste festgelegt (siehe Zeile 6).

Eine Regel (rule) besteht aus einer rechten Seite und einer linken Seite, die durch einen Doppelpunkt `:` voneinander getrennt werden können. Die Vereinbarung des Prädikates `Vertausche` umfaßt zwei Regeln (in Zeile 5 und 6). Diese Regeln sind besonders einfach: Ihre rechten Seiten sind leer (d.h. zwischen dem Doppelpunkt `:` und dem abschließenden Punkt `.` stehen keine Prädikataufrufe). Die linke Seite einer Regel wird häufig auch als der Kopf (head) der Regel und die rechte Seite als der Schwanz (tail) der Regel bezeichnet.

Im Kopf einer Regel muss (hinter dem Namen des Prädikates, das gerade vereinbart wird) für jeden Eingabeparameter ein Muster (des betreffenden Typs) und für jeden Ausgabeparameter ein Ausdruck (des betreffenden Typs) angegeben werden.

In der ersten Regel für das Prädikat `Vertausche` (siehe Zeile 5) ist also `list(E1, list(E2, R2))` (links vom Pfeil `->`) ein Muster und `list(E2, list(E1, R2))` (rechts vom Pfeil `->`) ein Ausdruck. Das Muster dient dazu, Werte für die Variablen `E1`, `E2` und `R2` festzulegen. Der Ausdruck setzt voraus, dass für die Variablen `E2`, `E1` und `R2` schon Werte festgelegt wurden und beschreibt unter dieser Voraussetzung einen bestimmten Wert vom Typ `LISTE`.

In der zweiten Regel für das Prädikat `Vertausche` (siehe Zeile 6) ist `L` (links vom Pfeil `->`) ein Muster und `L` (rechts vom Pfeil `->`) ein Ausdruck. Das Muster `L` dient dazu, für die Variable `L` einen Wert festzulegen. Der Ausdruck `L` setzt voraus, dass für die Variable `L` schon ein Wert festgelegt wurde und beschreibt unter dieser Voraussetzung einen bestimmten Wert (nämlich den Wert von `L`).

Angenommen, `TL1` (wie "Testliste 1") ist eine Variable vom Typ `LISTE` und hat den Wert `list(-17, list(3, leer))`. Dann kann man das Prädikat `Vertausche` z. B. folgendermaßen aufrufen:

```
7 Vertausche(list(25, TL1) -> list(ERG1, ERG2))
```

In jedem Aufruf muss für jeden Eingabeparameter ein Ausdruck (des betreffenden Typs) und für jeden Ausgabeparameter ein Muster (des betreffenden Typs) angegeben werden.

Im Beispiel-Aufruf des Prädikates `Vertausche` (siehe Zeile 7) ist also `list(25, TL1)` (links vom Pfeil `->`) ein Ausdruck und `list(ERG1, ERG2)` (rechts vom Pfeil `->`) ist ein Muster.

Man sieht: Für die Köpfe der Regeln des Prädikates und für die Aufrufe des Prädikates gelten "komplementäre Festlegungen". Einem Eingabeparameter des Prädikates entspricht in einem Regelkopf ein Muster und in einem Aufruf ein Ausdruck. Einem Ausgabeparameter entspricht in einem Regelkopf ein Ausdruck und in einem Aufruf ein Muster.

Hier die einzelnen Aktionen, die obiger Aufruf des Prädikates `Vertausche` auslöst:

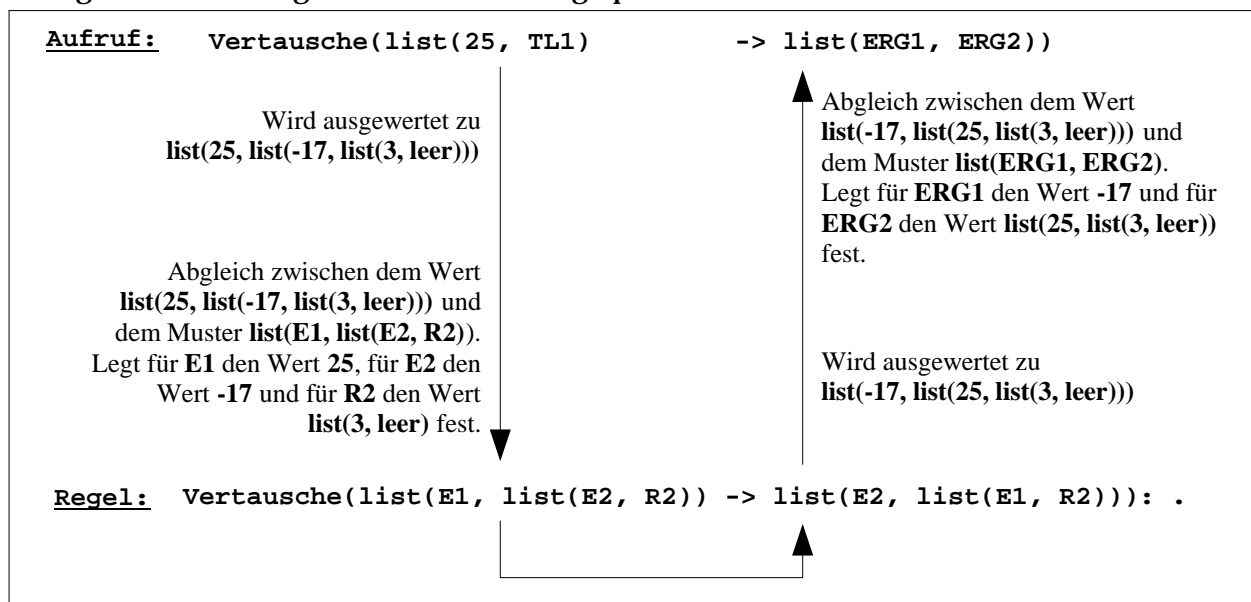
**Aktion 1:** Der aktuelle Eingabeparameter (d.h. der Ausdruck `list(25, TL1)`) wird ausgewertet. Als Ergebnis ergibt sich der Wert `list(25, list(-17, list(3, leer)))`.

**Aktion 2:** Der aktuelle Eingabeparameter (d.h. der Wert `list(25, list(-17, list(3, leer)))`) wird mit dem formalen Eingabeparameter der Regel (d.h. mit dem Muster `list(E1, list(E2, R2))`) abgeglichen. Dieser Musterabgleich legt für die `INT`-Variablen `E1` und `E2` die Werte `25` bzw. `-17` und für die `LISTE`-Variable `R2` den Wert `list(3, leer)` fest.

**Aktion 3:** Der formale Ausgabeparameter (d.h. der Ausdruck `list(E2, list(E1, R2))`) wird ausgewertet. Als Ergebnis ergibt sich der Wert `list(-17, list(25, list(3, leer)))`.

**Aktion 4:** Dieser Wert wird mit dem aktuellen Ausgabeparameter des Aufrufs (d.h. mit dem Muster `list(ERG1, ERG2)`) abgeglichen. Dieser Musterabgleich legt für die Variable `ERG1` den Wert `-17` und für `ERG2` den Wert `list(25, list(3, leer))` fest.

Die folgende Darstellung soll diese Aktionen graphisch veranschaulichen:



Wenn einer der beiden Musterabgleiche (Aktion 2 bzw. 4) nicht möglich ist, kann die aktuelle Regel des Prädikates nicht angewendet werden und der Ausführer versucht, die nächste Regel anzuwenden.

Wenn keine Regel des Prädikates angewendet werden kann, misslingt der Aufruf des Prädikates. Das geschieht beim Prädikat `Vertausche` aber nur dann, wenn man es etwa so aufruft:

```
8 Vertausche(leer -> list(ERG1, ERG2))
```

Mit der zweiten Regel wird als Ergebnis die Liste `leer` berechnet. Auf diese Liste passt aber das Muster

`list(ERG1, ERG2)` nicht und somit misslingt der Aufruf. Da `Vertausche` ein Aktions-Prädikat ist (und kein Bedingungs-Pädikat) würde ein solches Misslingen als schwerer Programmierfehler gewertet und den Abbruch der Programmausführung zur Folge haben.

**Aufgabe 20.1.:** Beschreiben Sie genau die Aktionen, die durch den Aufruf

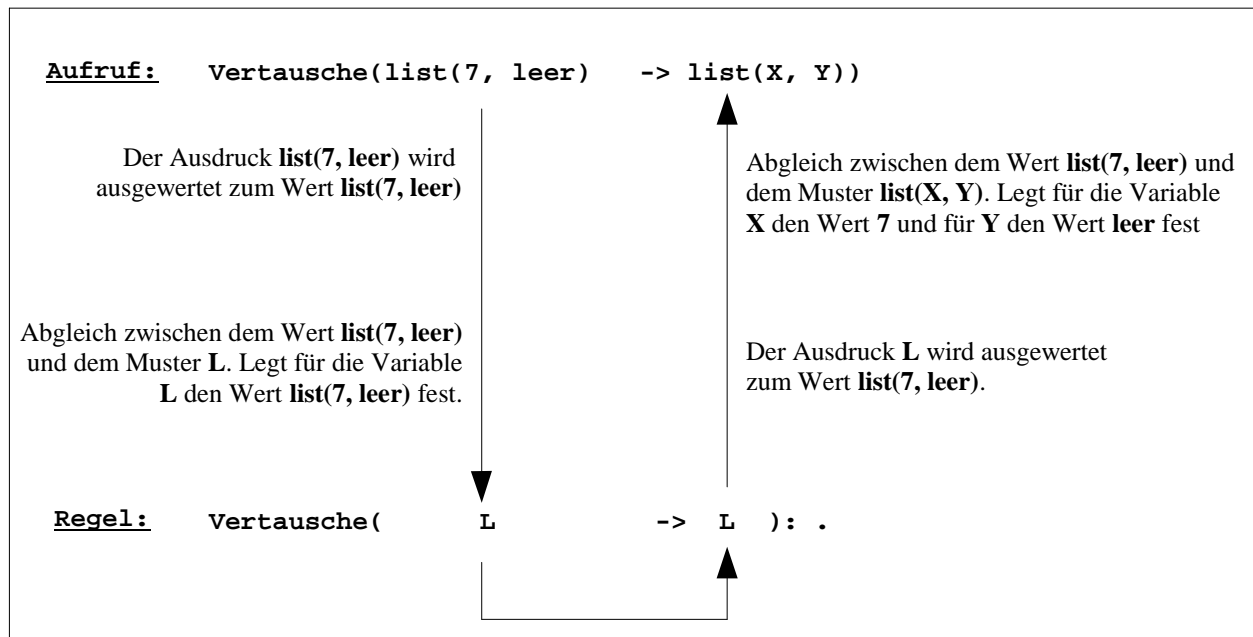
```
9 Vertausche(list(7, leer) -> list(X, Y))
```

ausgelöst werden. Benützen Sie dabei so häufig wie möglich die Fachbegriffe Wert, Ausdruck, Muster und Variable.

**Lösung zu Aufgabe 20.1.:**

1. Der aktuelle Eingabeparameter (d.h. der Ausdruck `list(7, leer)` in Zeile 8) wird ausgewertet. Als Ergebnis ergibt sich der Wert `list(7, leer)`.
2. Ein Abgleich zwischen diesem Wert und dem formalen Eingabeparameter im Kopf der ersten Regel (d.h. ein Abgleich mit dem Muster `list(E1, list(E2, leer))` in Zeile 6) ist nicht möglich. Also ist die erste Regel nicht anwendbar.
3. Ein Abgleich zwischen dem Wert `list(7, leer)` und dem formalen Eingabeparameter im Kopf der zweiten Regel (d.h. ein Abgleich mit dem Muster `L` links vom Pfeil `->` in Zeile 6) ist möglich und legt für die Variable `L` den Wert `list(7, leer)` fest.
4. Die Auswertung des formalen Ausgabeparameters (d.h. die Auswertung des Ausdrucks `L` rechts vom Pfeil `->` in Zeile 6) ergibt den Wert `list(7, leer)`.
5. Dieser Wert wird mit dem aktuellen Ausgabeparameter (d.h. mit dem Muster `list(X, Y)` in Zeile 8) abgeglichen und legt für die Variable `X` den Wert `7` und für `Y` den Wert `leer` fest.

Die folgende Graphik stellt nur die erfolgreiche Anwendung der zweiten Regel (die Aktionen 1., 3., 4. und 5.) dar. Das Scheitern des Versuchs, die erste Regel anzuwenden (Aktion 2.) wird nicht dargestellt):



Die Ausführung eines Aufrufs des Prädikates `Vertausche` ist vor allem deshalb so einfach und übersichtlich (-: ?), weil die Vereinbarung des Prädikates nur zwei Regeln umfaßt und in beiden Regeln die rechte Seite leer ist. Als nächstes soll ein Prädikat mit einer Regel vereinbart werden, deren rechte Seite nicht leer ist. Zuvor brauchen wir ein paar Hilfsbegriffe und Hilfsprädikate.

Wenn man aus einer (ausreichend langen) Liste `L` die ersten `n` Elemente entfernt, erhält man eine Liste, die wir im folgenden als den Rest `n` von `L` bezeichnen wollen. Ist z. B. `L` gleich der Liste `list(2, list(4, list(6, leer)))`, dann ist der Rest 1 von `L` gleich `list(4, list(6, leer))` und der Rest 3 von `L` ist gleich `leer`.

Die Prädikate `Rest1`, `Rest2` und `Rest3` sollen den betreffenden Rest einer (ausreichend langen) Liste berechnen. Auf "zu kurze Liste" sollen diese Prädikate nicht anwendbar sein. Deshalb empfiehlt es sich, sie als Bedingungs-Prädikate (und nicht als Aktions-Prädikate) zu vereinbaren.

**Aufgabe 20.2.:** Vereinbaren Sie drei Bedingungs-Prädikate namens `Rest1`, `Rest2` und `Rest3`, die den Rest 1 bzw. den Rest 2 bzw. den Rest 3 einer Liste (vom Typ `LISTE`) berechnen.

**Lösung zu Aufgabe 20.2.:**

```

1 'condition' Rest1(LISTE -> LISTE)
2   'rule' Rest1(list(E1, R1) -> R1): .
3
4 'condition' Rest2(LISTE -> LISTE)
5   'rule' Rest2(list(E1, list(E2, R2)) -> R2): .
6
7 'condition' Rest3(LISTE -> LISTE)
8   'rule' Rest3(list(E1, list(E2, list(E3, R3))) -> R3): .

```

Auch diese Prädikate haben nur Regeln mit leeren rechten Seiten (ähnlich wie das Prädikat `Vertausche`). Aber es gibt zahlreiche Varianten des Prädikates `Rest3`, deren Regeln nicht-leere rechte Sei-

ten besitzen. Die folgenden Prädikate `Rest3a` bis `Rest3d` "leisten" genau das Gleiche wie `Rest3`, berechnen den Wert ihres Ausgabeparameters aber auf andere Weise als `Rest3`:

```

 9 'condition' Rest3a(LISTE -> LISTE)
10 'rule'      Rest3a(L0 -> R3):
11             Rest1(L0 -> R1)
12             Rest1(R1 -> R2)
13             Rest1(R2 -> R3).
14
15 'condition' Rest3b(LISTE -> LISTE)
16 'rule'      Rest3b(L0 -> R3):
17             Rest1(L0 -> R1)
18             Rest2(R1 -> R3).
19
20 'condition' Rest3c(LISTE -> LISTE)
21 'rule'      Rest3c(list(E1, R1) -> R3):
22             Rest2(R1 -> R3).
23
24 'condition' Rest3d(LISTE -> LISTE)
25 'rule'      Rest3d(list(E1, list(E2, R2)) -> R3):
26             Rest1(R2 -> R3).

```

Die Vereinbarung des Prädikates `Rest3a` (Zeile 17 bis 21) umfaßt nur eine einzige Regel (Zeile 9 bis 13). Auf der rechten Seite dieser Regel (nach dem Doppelpunkt `:` in Zeile 10, d.h. in den Zeilen 11 bis 13) wird das Prädikat `Rest1` dreimal aufgerufen. Für diese Aufrufe gelten die oben beschriebenen Festlegungen, die zu den Festlegungen für die linke Seite der Regel (vor dem Doppelpunkt in Zeile 10) "komplementär" sind:

Auf der linken Seite der Regel (in Zeile 10) ist `L0` ein Muster und `R3` ist ein Ausdruck.

Auf der rechten Seite der Regel (in Zeile 19) ist `L0` ein Ausdruck und `R1` ist ein Muster.

In Zeile 20 ist `R1` ein Ausdruck und `R2` ein Muster und

in Zeile 21 ist `R2` ein Ausdruck und `R3` ein Muster.

Man sieht: Der Wahsinn hat Methode. Alle Muster dienen dazu, den Wert ihrer Variablen festzulegen. Alle Ausdrücke setzen voraus, dass für ihre Variablen schon Werte festgelegt wurden und beschreiben unter dieser Annahme einen Wert.

Angenommen, `TL2` (wie "Testliste 2") ist auch eine Variable vom Typ `LISTE` und hat den Wert `list(25, list(-17, list(3, leer)))`. Dann kann man das Prädikat `Rest3a` z. B. folgendermaßen aufrufen:

```
27 Rest3a(list(8, TL2) -> list(X, Y))
```

Dieser Aufruf wird folgendermaßen ausgeführt:

1. Der aktuelle Eingabeparameter (d.h. der Ausdruck `list(8, TL2)`) wird ausgewertet zum Wert `list(8, list(25, list(-17, list(3, leer))))`.
2. Dieser Wert wird mit dem formalen Eingabeparameter der ersten und einzigen `Rest3a`-Regel (d.h. mit dem Muster `L0` in Zeile 18) abgeglichen. Für die Variable `L0` wird dadurch der Wert `list(8, list(25, list(-17, list(3, leer))))` festgelegt.
3. Dann wird das Prädikat `Rest1` aufgerufen (in Zeile 11). Dazu wird der aktuelle Eingabeparameter des `Rest1`-Aufrufs (d.h. der Ausdruck `L0` in Zeile 11) ausgewertet zum Wert `list(8, list(25, list(-17, list(3, leer))))`. Dieser Wert wird mit dem formalen Eingabeparameter der ersten und einzigen `Rest1`-Regel (d.h. mit dem Muster `list(E1, R1)` in Zeile 10) abgeglichen. Für die Variable `E1` wird der Wert 8 und für `R1` der Wert

`list(25, list(-17, list(3, leer)))` festgelegt. Der formale Ausgabeparameter der `Rest1`-Regel (d.h. der Ausdruck `R1` in Zeile 11) wird ausgewertet zu `list(25, list(-17, list(3, leer)))`. Dieser Wert wird mit dem aktuellen Ausgabeparameter des `Rest1`-Aufrufs (d.h. mit dem Muster `R1` in Zeile 11) abgeglichen. Für die Variable `R1` (in Zeile 11) wird dadurch der Wert `list(25, list(-17, list(3, leer)))` festgelegt.

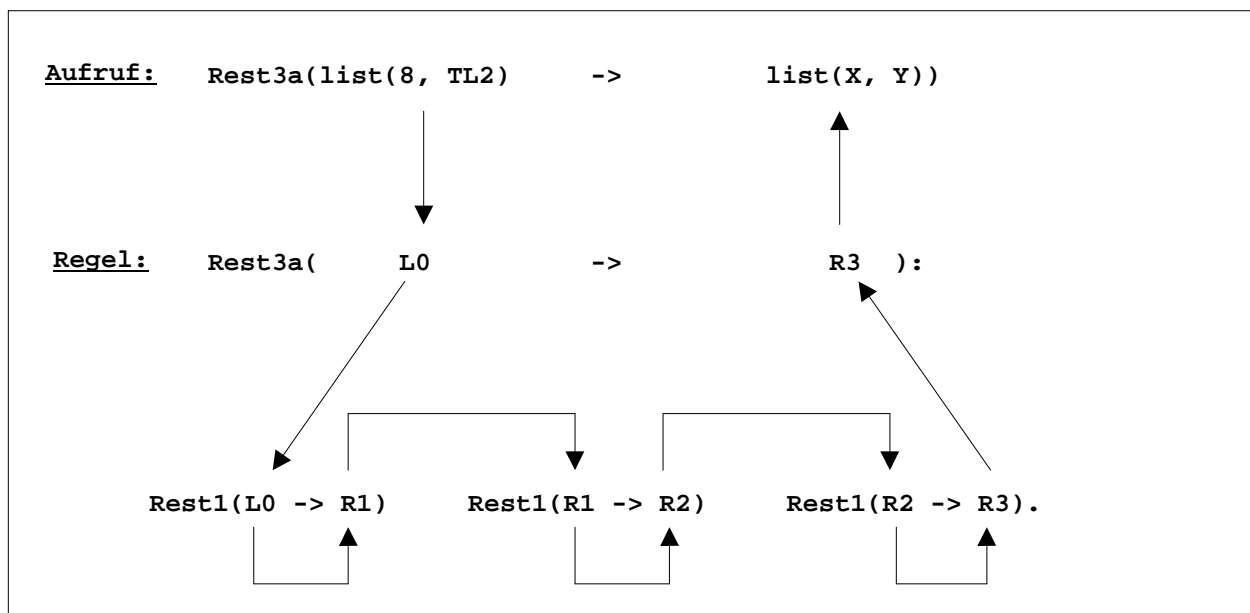
4. Dann wird das Prädikat `Rest1` zum zweiten Mal aufgerufen (in Zeile 12). Dazu wird ... (bitte ergänzen Sie). Für die Variable `R2` (in Zeile 12) wird der Wert `list(-17, list(3, leer))` festgelegt.

5. Dann wird das Prädikat `Rest1` zum dritten Mal aufgerufen (in Zeile 13). Dazu wird ... . Für die Variable `R3` (in Zeile 13) wird der Wert `list(3, leer)` festgelegt.

6. Der formale Ausgabeparameter der `Rest3a`-Regel (d.h. der Ausdruck `R3` in Zeile 10) wird ausgewertet zum Wert `list(3, leer)`.

7. Dieser Wert wird mit dem aktuellen Ausgabeparameter des `Rest3a`-Aufrufs (d.h. mit dem Muster `list(X, Y)` in Zeile 27) abgestimmt. Für die Variable `X` wird dadurch der Wert `3` und für `Y` der Wert `leer` festgelegt.

Hier eine graphische Darstellung dieser Aktionen. Wenn man den Pfeilen folgt, kommt man immer abwechseln an einem Ausdruck, an einem Muster, an einem Ausdruck, ... an einem Muster vorbei:



Von einer Variablen in einem Muster sagt man auch, sie stehe dort auf einer *definierenden Position*, weil das Muster dazu dient, für die Variable einen Wert festzulegen oder zu definieren. Eine Variable in einem Ausdruck steht dagegen auf einer *anwendenden Position*, weil dort vorausgesetzt wird, dass die Variable schon einen Wert hat und dieser Wert angewendet wird (um damit einen weiteren Wert zu beschreiben). Die obige Abbildung macht (hoffentlich) deutlich, in welcher Reihenfolge die einzelnen Ausdrücke und Muster in einer Regel "besucht" werden. Beim Programmieren einer Prädikat-Regel muss man folgende Bestimmungen beachten:

1. Variablen sind Regel-lokal. Die Variablen in einer Regel haben nichts zu tun mit den Variablen in anderen Regeln (desselben Prädikates oder anderer Prädikate). Wie man *globale* Variablen vereinbart und benützt, wird in einem anderen Abschnitt erläutert.
2. In einer Regel darf keine Variable mehr als einmal definiert werden (d.h. keine Variable darf mehr als einmal in den Mustern der Regel vorkommen).
3. Nachdem sie definiert wurde, darf eine Variable beliebig oft angewendet werden (d.h. in Ausdrücken der Regel vorkommen). Was "nachdem" heißt, geht (hoffentlich) aus obiger Graphik hervor.

**Aufgabe 20.3.:** Sei weiterhin angenommen, dass TL2 (wie "Testliste 2") eine Variable vom Typ LISTE ist und den Wert `list(25, list(-17, list(3, leer)))` hat. Welche Aktionen löst dann folgender Aufruf des Prädikates Rest3b aus?

```
28 Rest3b(list(8, TL2) -> list(A, B))
```

**Aufgabe 20.4.:** Ebenso für den folgenden Aufruf des Prädikates Rest3c:

```
29 Rest3c(list(8, TL2) -> list(C, D))
```

**Aufgabe 20.5.:** Ebenso für den folgenden Aufruf des Prädikates Rest3d:

```
30 Rest3d(list(8, TL2) -> list(E, F))
```

## 21. Und/Oder-Bäume, tiefes und flaches Wiederaufsetzen

Die Ausführung eines Prädikataufrufes kann man sich als ein Durchsuchen eines sogenannten *Und/Oder-Baumes* vorstellen. An der Wurzel eines solchen Baumes steht der auszuführende Prädikataufruf. In der "Schicht unterhalb der Wurzel" entspricht jeder Knoten einer Regel (des aufgerufenen Prädikates). Diese Regel-Knoten sollte man sich durch Oder verknüpft denken, denn der Prädikataufruf ist erfolgreich ausgeführt, wenn eine der Regeln erfolgreich ausgeführt wurde (Regel 1 oder Regel 2 oder ... oder Regel n):

```
Aufruf -|- Regel 1
         |- Regel 2
         |
         |...
         |- Regel n
```

Unter jeder Regel R stehen auf der nächst tieferen Schicht die Prädikataufrufe (und ähnliche Konstrukte), die die rechte Seite von R bilden. Diese Prädikataufrufe sollte man sich durch Und verknüpft denken, denn die Regel R ist erfolgreich ausgeführt, wenn alle Prädikataufrufe auf ihrer rechten Seite erfolgreich ausgeführt wurden (Aufruf 1 und Aufruf 2 und ... und Aufruf m):

```
Regel -|- Aufruf 1
        |- Aufruf 2
        |
        |...
        |- Aufruf m
```

Unter jedem Aufruf A stehen auf der nächst tieferen Schicht wieder die Regeln des aufgerufenen Prädikates und diese Regeln sind mit Oder verknüpft und unter jeder Regel stehen auf der nächst tieferen Schicht die Aufrufe, die die rechte Seite der Regel bilden und diese Aufrufe sind mit Und verknüpft u.s.w.

Ein Blatt eines solchen Und/Oder-Baumes ist entweder eine Regel, deren rechte Seite leer ist oder ein Aufruf eines "primitiven Prädikates", dessen Bedeutung nicht durch Regeln (sondern z. B. durch eine C-Routine) definiert ist.

Die folgenden Gentle-Typen und -Prädikate sollen dazu dienen, einige Und/Oder-Bäume als konkrete Beispiele zu konstruieren:

```

1 'type' PERSON a b c d e           -- Anton, Berta, Claus, Doris und Ernst
2 'condition' Mag(PERSON -> PERSON) -- Wer mag wen?
3   'rule' Mag(a -> b): .           -- Regel Mag1
4   'rule' Mag(b -> a): .           -- Regel Mag2
5   'rule' Mag(d -> c): .           -- Regel Mag3
6   'rule' Mag(d -> e): .           -- Regel Mag4
7   'rule' Mag(e -> d): .           -- Regel Mag5
8 'condition' IsR(PERSON)           -- Ist reich
9   'rule' IsR(b): .               -- Regel IsR1
10  'rule' IsR(c): .               -- Regel IsR2
11 'condition' HaG(PERSON)          -- Hat Glück
12  'rule' HaG(P0):                -- Regel HaG1
13    IsR(P0).                     -- Aufruf HaG1.1
14  'rule' HaG(P0):                -- Regel HaG2
15    Mag(P0 -> P1)                -- Aufruf HaG2.1
16    Mag(P1 -> P2)                -- Aufruf HaG2.2
17    eq(P0, P2).                  -- Aufruf HaG2.3

```

Das Prädikat `Mag` soll beschreiben, welche Person welche andere Person (bzw. welche anderen Personen, Mehrzahl) mag. Man beachte, dass Doris sowohl den Claus als auch den Ernst mag (siehe Regel `Mag3` und `Mag4`). Anhand der Regeln `Mag3` und `Mag4` soll der Unterschied zwischen zwei wichtigen Suchstrategien in Und/Oder-Bäumen illustriert werden (der Unterschied zwischen "tiefem Wiederaufsetzen" und "flachem Wiederaufsetzen", the difference between deep and shallow backtracking). Dieser Unterschied ist gleichzeitig ein wichtiger Unterschied zwischen der Sprache Prolog und der Sprache Gentle. In einem Gentle-Programm wäre die Regel `Mag4` sinnlos, in einem Prolog-Programm könnte eine entsprechende Regel dagegen durchaus sinnvoll sein. Dieser Unterschied zwischen Gentle und Prolog (d.h. der Unterschied zwischen flachem und tiefen Wiederaufsetzen) wird im folgenden genauer behandelt.

Das Prädikat `IsR` ("Ist reich") besagt, dass Berta und Claus reich sind (und die anderen drei Personen nicht reich sind). Zum Prädikat `HaG` ("Hat Glück") gehören zwei Regeln. Die erste (`HaG1`) besagt, dass eine Person Glück hat, wenn sie reich ist. Die zweite Regel (`HaG2`) drückt aus, dass eine Person `P0` Glück hat, wenn `P0` eine Person `P1` mag und `P1` wiederum `P0` mag.

Als erstes Beispiel soll der Prädikataufruf `HaG(a)` ("Hat Anton Glück?") betrachtet werden. Die Ausführung dieses Aufrufs kann durch den folgenden Und/Oder-Baum dargestellt werden:

```

           oder      und      oder
1  HaG(a) -|- HaG1 -|- IsR(a) -----|- IsR1 -- false
2           |           |           |-----|- IsR2 -- false
3           |           |           |
4           |- HaG2 -|- Mag(a -> P1) -|- Mag1 -- true (P1=b)
5           |           |           |
6           |           |- Mag(b -> P2) -|- Mag1 -- false
7           |           |           |-----|- Mag2 -- true (P2=a)
8           |           |           |
9           |           |-- eq(a, a) ----|- true

```

Der Aufruf `HaG(a)` kann erfolgreich ausgeführt werden, wenn die Regel `HaG1` oder die Regel `HaG2` erfolgreich ausgeführt werden kann. Die Regel `HaG1` kann nur dann erfolgreich ausgeführt werden, wenn der Aufruf `IsR(a)` erfolgreich ausgeführt werden kann. Der Aufruf `IsR(a)` kann erfolgreich

ausgeführt werden, wenn die Regel  $IsR1$  oder die Regel  $IsR2$  erfolgreich ausgeführt werden kann. Das ist aber nicht der Fall, also können der Aufruf  $IsR(a)$  und damit die Regel  $HaG1$  nicht erfolgreich ausgeführt werden. Die Regel  $HaG2$  kann nur dann erfolgreich ausgeführt werden, wenn der Aufruf  $Mag(a \rightarrow P1)$  und der Aufruf  $Mag(P1 \rightarrow P2)$  und der Aufruf  $eq(a, P2)$  erfolgreich ausgeführt werden können u.s.w. Als Endergebnis ergibt sich, dass Anton tatsächlich Glück hat (weil er Berta mag und Berta ihn auch mag).

**Aufgabe 21.1.:** Beschreiben Sie ganz entsprechend die Ausführung des Prädiaktaufrufs  $HaG(e)$  ("Hat Ernst Glück?") durch einen Und/Oder-Baum und in Worten.

Das hier besonders wichtige Beispiel ist der Prädiktaufruf  $HaG(d)$  ("Hat Doris Glück?"). Der folgende Und/Oder-Baum stellt die Ausführung dieses Aufrufs nach der Strategie *flaches Wiederaufsetzen* (shallow backtracking) dar. Diese Strategie wird in Gentle (für die Ausführung von Aktions- und Bedingungs-Prädiakten) verwendet:

```

      oder      und      oder
10  HaG(d) -|- HaG1 -|- IsR(d) -----|- IsR1 -- false
11                |- IsR2 -- false
12
13                -|- HaG2 -|- Mag(d -> P1) -|- Mag1 -- false
14                |- Mag2 -- false
15                |- Mag3 -- true   (P0=d, P1=c)
16
17                -|- Mag(c -> P2) -|- Mag1 -- false
18                |- Mag2 -- false
19                |- Mag3 -- false
20                |- Mag4 -- false
21                |- Mag5 -- false

```

Man erkennt hier: Die Regel  $HaG1$  kann nicht ausgeführt werden ("Doris ist nicht reich"). Deshalb wird versucht, die Regel  $HaG2$  auszuführen (siehe Zeile 13). Der Aufruf  $Mag(d \rightarrow P1)$  kann aufgrund der Regel  $Mag3$  ausgeführt werden und belegt die Variable  $P1$  mit dem Wert  $Carl$  ("Doris mag Carl", siehe Zeile 15). Mit dieser Belegung scheitert dann aber die Ausführung des nächsten Aufrufs  $Mag(c \rightarrow P2)$ , denn "Claus mag niemanden" (siehe Zeile 17 bis 21). Damit ist die Regel  $HaG2$  gescheitert. Da es für das Prädikat  $HaG$  keine weiteren Regeln ( $HaG3$ ,  $HaG4$  etc.) gibt, die man jetzt noch probieren könnte, ist damit die Ausführung des Aufrufs  $HaG(d)$  an der Wurzel des Und/Oder-Baumes endgültig gescheitert ("Doris hat kein Glück").

Dieses Scheitern des Aufrufs  $HaG(d)$  ist eine Konsequenz der verwendeten Strategie *flaches Wiederaufsetzen*. Bei dieser Strategie betrachtet man von jedem Aufruf nur "das *erste* Gelingen" (falls die Ausführung des Aufrufs überhaupt gelingen kann). Der Aufruf  $Mag(d \rightarrow P1)$  könnte aber nicht nur aufgrund der Regel  $Mag3$  gelingen ("Doris mag Claus"), sondern auch aufgrund der Regel  $Mag4$  ("Doris mag Ernst"). Bei der Strategie *tiefes Wiederaufsetzen* (deep backtracking) betrachtet man von jedem Aufruf nicht nur die erste gelungene Ausführung, sondern *alle* möglichen gelungenen Ausführungen. Mit dieser Strategie hat Doris doch noch Glück, denn sie mag ja (außer Carl auch noch) den Ernst und er mag sie. Der folgende Und/Oder-Baum stellt die Ausführung des Aufrufs  $HaG(d)$  nach der Strategie tiefes Wiederaufsetzen (deep backtracking) dar. Diese Strategie wird in Prolog verwendet:

```

      oder      und      oder
22  HaG(d) -|- HaG1 -|- IsR(d) -----|- IsR1 -- false
23          |         |         |         |         |- IsR2 -- false
24          |         |         |         |         |
25          |- HaG2 -|- Mag(d -> P1) -|- Mag1 -- false
26          |         |         |         |         |- Mag2 -- false
27          |         |         |         |         |- Mag3 -- true   (P0=d, P1=c)
28          |         |         |         |         |
29          |         |- Mag(c -> P2) -|- Mag1 -- false
30          |         z         |         |         |- Mag2 -- false
31          |         u         |         |         |- Mag3 -- false
32          |         r         |         |         |- Mag4 -- false
33          |         ü         |         |         |- Mag5 -- false
34          |         c         |         |         |
35          |         k- Mag(d -> P1) -|- Mag4 -- true   (P1=e)
36          |         |         |         |         |
37          |         |- Mag(e -> P2) -|- Mag1 -- false
38          |         |         |         |         |- Mag2 -- false
39          |         |         |         |         |- Mag3 -- false
40          |         |         |         |         |- Mag4 -- false
41          |         |         |         |         |- Mag5 -- true   (P2=d)
42          |         |         |         |         |
43          |         |- eq(d, d) -----|- true

```

Wenn die Ausführung des Aufrufs `Mag(c ->P2)` in Zeile 33 scheitert, dann wird sozusagen zur Zeile 27 zurückgegangen. Dort wurde mit der Regel `Mag3` die erste Möglichkeit gefunden, den Aufruf `Mag(d -> P1)` erfolgreich auszuführen. Jetzt wird (in Zeile 35) die nächste Möglichkeit (`Mag4`: "Doris mag Ernst") genommen und damit "weitergemacht".

Den Unterschied zwischen flachem Wiederaufsetzen (wie in Gentle) und tiefem Wiederaufsetzen (wie in Prolog) kann man auch so zusammenfassen: Beim flachen Wiederaufsetzen stellt jedes Prädikat eine *Funktion* dar, während beim tiefen Wiederaufsetzen ein Prädikat im allgemeinen eine *Relation* beschreibt. Wenn `Mag` eine *Funktion* ist, dann kann Doris nur *einen* mögen. Ist `Mag` dagegen eine *Relation*, dann kann Doris *mehrere* Personen mögen.

Eine Funktion ist eine linkstotale, rechtseindeutige Relation. Eine nullstellige Funktion ist ein Wert. Eine nullstellige Relation ist eine Menge von Werten. Z. B. stellt das folgende Prädikat in Gentle den Wert `b` dar. Ein entsprechendes Prädikat in Prolog würde dagegen die Menge `{b, c}` beschreiben:

```

1 'condition' ReM(-> PERSON)      -- Reiche Menschen
2 'rule' ReM(-> b): .             -- Berta ist ein reicher Mensch
3 'rule' ReM(-> c): .             -- Claus ist ein reicher Mensch

```

## 22. altcon\_1.g: Die Alternativen-Anweisung und die bedingte Anweisung

Mit der *Alternativen-Anweisung* (alternative statement) und mit der *bedingten Anweisung* (conditional statement) kann man bestimmte Probleme in einem Gentle-Programm besonders einfach und übersichtlich lösen. Man kann diese Anweisungen aber in jedem Fall durch andere Befehle (meistens durch die Vereinbarung zusätzlicher Prädikate) ersetzen. In diesem Sinne sind die Alternativen-Anweisung und die bedingte Anweisung eigentlich "nicht unbedingt nötig".

```

1  -----1*****2*****3*****4*****5*****6*****7*****
2  -- altcon_1.g: Zeigt die Anwendung der Alternativen-Anweisung
3  -- "(| ... || ... || ... .. || ... |)" und der bedingten Anweisung
4  -- "[| ... |]" der Sprache Gentle
5  -----
6  -- Die Praedikate meld1 und meld2 leisten beide das Gleiche: Sie geben
7  -- eine Fehlermeldung aus. In meld2 wurden unschoene Wiederholungen durch
8  -- Anwendung der Alternativ-Konstrukte von Gentle vermieden.
9  -----
10 'action' meld1(Text: STRING, Gewicht: INT)
11     meld1(T, G): le(G, 0)
12                 PutS("Meldung: ") PutS(T) Tab PutS("Warnung")           Nl.
13     meld1(T, G): eq(G, 1)
14                 PutS("Meldung: ") PutS(T) Tab PutS("Leichter Fehler") Nl.
15     meld1(T, G): ge(G, 2)
16                 le(G, 4)
17                 PutS("Meldung: ") PutS(T) Tab PutS("Schwerer Fehler") Nl.
18     meld1(T, G): ge(G, 5)
19                 PutS("Meldung: ") PutS(T) Tab PutS("Schwerer Fehler") Nl
20                 PutS("Die Anlage wird abgeschaltet!")           Nl.
21 -----
22 'action' meld2(Text: STRING, Gewicht: INT)
23     meld2(T, G): PutS("Meldung: ") PutS(T) Tab
24                 (| le(G, 0) PutS("Warnung")
25                 || eq(G, 1) PutS("Leichter Fehler")
26                 || ge(G, 2) PutS("Schwerer Fehler")
27                 [| ge(G, 5) Nl PutS("Die Anlage wird abgeschaltet!") |]
28                 |)
29                 Nl.
30 -----
31 -- Pradikate, die in der Datei text_io.c definiert sind:
32 'action' PutS(STRING) -- Gib eine Zeichenkette vom Typ STRING aus
33 'action' Tab          -- Gib ein Tabulator-Zeichen aus
34 'action' Nl          -- Gib ein newline-Zeichen aus
35 -----
36 'root'
37     meld1("Die Wassertemperatur steigt an      ", 0)
38     meld1("In einer Dampfleitung quietscht es  ", 1)
39     meld1("Geringe Wassermengen sind ausgetreten", 2)
40     meld1("Die Versorgungsspannung faellt ab   ", 5)
41     Nl
42     meld2("Die Wassertemperatur steigt an      ", 0)
43     meld2("In einer Dampfleitung quietscht es  ", 1)
44     meld2("Geringe Wassermengen sind ausgetreten", 2)
45     meld2("Die Versorgungsspannung faellt ab   ", 5)
46 -----1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 10 bis 20: Kritik an der Vereinbarung des Prädikates `meld1`: Die vier Regeln dieses Prädikates enthalten unschöne "Wiederholungen der gleichen Prädikataufrufe" (`PutS("Meldung: ")`), `PutS`**

(T) und N1). Zwischen Regel 3 und Regel 4 gibt es sogar noch mehr Übereinstimmungen. Solche "Wiederholungen" machen nicht nur beim Schreiben und Lesen Mühe, sondern vor allem bei der Wartung des Programms (wenn man aus Versehen einige, aber nicht alle Wiederholungen eines Befehls verbessert oder ändert).

**Zeile 22:** Das Prädikat `meld2` leistet das Gleiche wie `meld1`, in seiner Vereinbarung wurden aber unnötige Befehlswiederholungen mithilfe der Alternativen-Anweisung und der bedingten Anweisung vermieden. Das Prädikat `meld2` hat nur eine einzige Regel (Zeile 23 bis 29).

**Zeile 24 bis 28:** Eine Alternativen-Anweisung beginnt mit `( |`, endet mit `| )` und sollte mindestens zwei Alternativen enthalten (sonst ist die Anweisung nicht sinnvoll). Die Alternativen müssen durch `| |` voneinander getrennt werden. Eine Alternative besteht aus einer Folge von Anweisungen (Prädikat-Aufrufen, geschachtelten Alternativ-Anweisungen, bedingten Anweisungen etc.). Die Anweisungen einer Alternativen werden in der Reihenfolge ausgeführt, in der sie notiert sind. Wenn die Ausführung aller Anweisungen gelingt, dann ist damit auch die Ausführung der Alternativen gelungen. Sonst wird die Ausführung der Alternativen bei der ersten misslungenen Anweisung abgebrochen und gilt als misslungen. Die Alternativen einer Alternativen-Anweisung werden in der Reihenfolge ausgeführt, in der sie notiert sind. Wenn die Ausführung einer Alternativen gelingt, wird die Ausführung der gesamten Alternativen-Anweisung abgebrochen und gilt als gelungen. Wenn alle Alternativen misslingen, dann ist damit auch die Ausführung der Alternativen-Anweisung misslungen.

**Zeile 27:** Eine bedingte Anweisung beginnt mit `[ |` und endet mit `| ]`. Sie enthält eine Anweisungs-Folge ("eine Alternative"). Die Anweisungen werden in der Reihenfolge ausgeführt, in der sie notiert sind. Wenn die Ausführung einer Anweisung misslingt, wird die Ausführung der bedingten Anweisung abgebrochen, gilt aber trotzdem als gelungen.

### 23. `condit_1.g`: Bedinungs-Prädikate (conditional predicates) und Listen

Das folgende Programm `condit_1.g` soll vor allem zeigen, wie man in einem Gentle-Programm *Bedinungs-Prädikate* (condition predicates) einsetzen kann. Das Programm `condit_1.g` ist kein Compiler oder Interpreter, sondern dient dazu, ein (unrealistisch einfaches) Telefonverzeichnis zu verwalten. Die Namen und Telefon-Nummern, die in dieses Verzeichnis eingetragen werden, sind im Programm `condit_1.g` fest vorgegeben und können vom Benutzer nicht verändert werden. Dadurch ist das Programm zwar nicht besonders nützlich, aber sehr einfach zu bedienen.

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- condit_1.g: Ein (sehr einfaches) Telefonverzeichnis wird als
3  -- (unsortierte, lineare) Liste realisiert.
4  -----
5  'type' LISTE
6      nix
7      k(Name: STRING, TelNr: INT, Rest_der_Liste: LISTE)
8  -----
9  -- Zum Typ LISTE gehoeren u.a. folgende Terme:
10 -- nix
11 -- k("Otto", 123, nix)
12 -- k("Bert", 234, k("Otto", 123, nix))
13 -- k("Ralf", 345, k("Bert", 234, k("Otto", 123, nix)))

```

```

14 -----
15 'condition' HatNr(Name: STRING, Verzeichnis: LISTE -> TelNr: INT)
16 -- Gelingt und liefert eine TelNr, wenn der Name im Verzeichnis steht:
17 -----
18 'rule' HatNr(NAME, k(NAM1, NR, RL) -> NR):
19     eq(NAME, NAM1).
20 -- Gefunden, der NAME hat im Verzeichnis die NR:
21 'rule' HatNr(NAME, k(NAM1, NR1, RL) -> NR):
22     HatNr(NAME, RL -> NR).
23 -- Im Rest RL der Liste weitersuchen:
24 -----
25 'action' TragEin(Name: STRING, TelNr: INT, VAlt: LISTE -> VNeu: LISTE)
26 -- Liefert ein Verzeichnis VNeu, in dem der Name mit der TelNr einge-
27 -- tragen ist, wenn der Name noch nicht im Verzeichnis VAlt steht.
28 -- Liefert sonst das unveraenderte Verzeichnis VAlt:
29 -----
30 'rule' TragEin(NAME, NR, L -> L):
31     HatNr(NAME, L -> NR1).
32 -- Der Name wird NICHT eingetragen, wenn er schon im Verzeichnis steht.
33 'rule' TragEin(NAME, NR, L -> k(NAME, NR, L)): .
34 -- Eintrag in eine beliebige Liste L.
35 -----
36 'action' gib_alle_aus(Verzeichnis: LISTE)
37 -- Gibt alle Eintraege eines Verzeichnisses aus:
38 -----
39 'rule' gib_alle_aus(nix): . -- leeres Verzeichnis, gib nichts aus!
40 'rule' gib_alle_aus(k(NAME, NR, RL)):
41     PutS(NAME) Tab PutI(NR) Nl -- Gib ersten Eintrag aus
42     gib_alle_aus(RL). -- Gib den Rest RL der Liste aus
43 -----
44 'action' gib_einen_aus(Verzeichnis: LISTE, Name: STRING)
45 -- Wenn es im Verzeichnis einen Eintrag mit dem Namen und einer TelNr
46 -- gibt, wird der Name und die TelNr ausgegeben. Sonst wird der Name
47 -- in einer Fehlermeldung ausgegeben:
48 -----
49 'rule' gib_einen_aus(L, NAME):
50     HatNr(NAME, L -> NR)
51     PutS(NAME) PutS(" hat die Nr. ") PutI(NR) Nl
52 'rule' gib_einen_aus(L, NAME):
53     PutS(NAME) PutS(" steht nicht im Verzeichnis!") Nl
54 -----
55 -- Die folgenden Aktions-Praedikate werden hier nur vereinbart. Sie
56 -- wurden als C-Funktionen in der Datei text_io.c realisiert:
57 -----
58 'action' PutS(STRING) -- Gibt eine Zeichenkette zur aktuellen Ausgabe aus
59 'action' PutI(INT) -- Gibt eine Ganzzahl zur aktuellen Ausgabe aus
60 'action' Tab -- Gibt ein Tab-Zeichen zur aktuellen Ausgabe aus
61 'action' Nl -- Gibt ein newline-Z. zur aktuellen Ausgabe aus
62 -----
63 -- Das Wurzel-Praedikat (root predicate) ist das "Hauptprogramm":
64 -----
65 'root'
66 PutS("----- condit_1-Anfang") Nl
67 TragEin("Otto", 123, nix -> VERZ1)
68 TragEin("Bert", 234, VERZ1 -> VERZ2)
69 TragEin("Ralf", 345, VERZ2 -> VERZ3)
70 TragEin("Susi", 456, VERZ3 -> VERZ4)
71 TragEin("Carl", 567, VERZ4 -> VERZ5)
72 TragEin("Anna", 678, VERZ5 -> VERZ6)

```

```

73  gib_alle_aus (VERZ6)  N1
74  gib_einen_aus(VERZ6, "Ralf" )
75  gib_einen_aus(VERZ6, "Anne" )
76  gib_einen_aus(VERZ6, "Anna" )
77  PutS("----- condit_1-Ende")  N1
78  --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 05 bis 07:** Eine Liste ist entweder leer (`nix`, Zeile 6) oder sie besteht aus einem Namen, einer Telefon-Nr und einem Rest der Liste.

**Zeile 15:** `HatNr` ist ein Bedingungs-Prädikat (condition predicate). Charakteristisch für ein Bedingungs-Prädikat ist, dass seine Aufrufe (invocations) gelingen oder misslingen können. Auch der Aufruf eines Aktions-Prädikates kann misslingen, aber dann liegt ein schwerwiegender Programmierfehler vor und die Ausführung des betreffenden Gentle-Programms wird sofort mit einer Fehlermeldung abgebrochen. Dagegen ist es ganz normal, wenn ein Aufruf eines Bedingungs-Prädikates misslingt.

Das Prädikat `HatNr` hat einen Namen und ein Verzeichnis als Eingabe-Parameter. Ein Aufruf von `HatNr` gelingt, wenn der Name im Verzeichnis vorkommt. In diesem Fall liefert das Prädikat `HatNr` als Ausgabe-Parameter die Telefon-Nr, die hinter dem Namen im Verzeichnis steht. Wenn der Name nicht im Verzeichnis steht, dann misslingt der Aufruf des Prädikates `HatNr`.

**Zeile 25:** Mit dem Aktions-Prädikat `TragEin` kann man einen Namen und eine Telefon-Nr in ein Verzeichnis eintragen, aber nur, wenn der Name noch nicht in dem Verzeichnis vorkommt. Ob der Name schon in dem Verzeichnis vorkommt oder nicht, wird mithilfe des Bedingungs-Prädikates `HatNr` festgestellt.

**Zeile 30 bis 31:** Diese Regel (für das Aktions-Prädikat `TragEin`) kann nur angewendet werden, wenn der Aufruf des Bedingungs-Prädikates `HatNr` gelingt. In diesem Fall wird die Liste `L` (d.h. das Telefonverzeichnis) unverändert als Ausgabeparameter zurückgeliefert. Falls der Aufruf von `HatNr` misslingt, wird die nächste Regel (siehe Zeile 33) angewendet.

**Zeile 33:** Der `NAME` und die `NR` werden in das Telefonverzeichnis eingetragen. Als Ergebnis wird die Liste `k(NAME, NR, L)` geliefert.

**Zeile 36:** Mit dem Aktions-Prädikat `gib_alle_aus` kann man alle Einträge einer Liste ausgeben.

**Zeile 44:** Mit dem Aktions-Prädikat `gib_einen_aus` kann man einen Eintrag einer Liste ausgeben. Als Eingabe-Parameter muss man die Liste und den Namen des auszugebenden Eintrags angeben. Ausgegeben wird der Name und die zugehörige Telefon-Nr oder der Name und der Text "steht nicht im Verzeichnis".

**Zeile 49 bis 51:** Nur wenn der Aufruf des Bedingungs-Prädikates `HatNr` in Zeile 50 gelingt, kann diese Regel des Prädikates `gib_einen_aus` angewendet werden. Mißlingt der Aufruf von `HatNr`, wird die nächste Regel für `gib_einen_aus` (siehe Zeile 52 bis 53) angewendet.

**Zeile 67 bis 72:** Eine sehr wichtige und grundlegende Regel der Sprache Gentle besagt, dass man einer Variablen nur *einmal* einen Wert zuweisen darf. Deshalb müssen hier 6 verschiedene Variablen `VERZ1` bis `VERZ6` vereinbart und mit einem Wert versehen werden. `VERZ1` ist ein Telefonverzeichnis mit einem einzigen Eintrag (für `Otto`). `VERZ2` ist ein Verzeichnis mit zwei Einträgen (für `Otto` und `Bert`) etc. und `VERZ6` ist ein Verzeichnis mit sechs Einträgen (für `Otto`, `Bert`, `Ralf`, `Susi`, `Carl` und `Anna`). Würde man z. B. in Zeile 68 schreiben:

```
68  TragEin("Bert", 234, VERZ1 -> VERZ1)
```

so wäre das ein Versuch, der Variablen `VERZ1` ein zweites Mal einen Wert zuzuweisen und somit ein schwerwiegender Programmierfehler. Die Regel, dass man einer Variablen nur einmal einen Wert zu-

weisen und diesen Wert dann nie mehr ändern darf, unterscheidet funktionale und deklarative Programmiersprachen (wie z. B. Lisp, Prolog, SQL und Gentle) von imperativen Sprachen (wie z. B. Fortran, C, Java, Pascal oder Ada).

Hier die Ausgabe des Programms `condit_1`:

```

1 ----- condit_1-Anfang
2 Anna      678
3 Carl      567
4 Susi      456
5 Ralf      345
6 Bert      234
7 Otto      123
8
9 Ralf hat die Nr. 345
10 Anne steht nicht im Verzeichnis!
11 Anna hat die Nr. 678
12 ----- condit_1-Ende

```

Man beachte, dass die Einträge in folgender Reihenfolge in das Telefonverzeichnis eingefügt wurden: Otto, Bert, Ralf, Susi, Carl und Anna (siehe Zeile 67 bis 72 des Programms `condit_1`). In der Ausgabe (siehe Zeile 2 bis 07 der Ausgabe) erscheinen die Einträge genau in der umgekehrten Reihenfolge. Eine Sortierung der Einträge wurde an keiner Stelle des Programm vorgenommen.

## 24. condit\_2.g: Bedingungs-Prädikate (conditional predicates) und Bäume

Im vorigen Programm `condit_1.g` wurde ein Telefonverzeichnis als (unsortierte) lineare Liste realisiert. Im folgenden Programm `condit_2.g` wird ein Telefonverzeichnis als (sortierter) binärer Baum realisiert. Ansonsten leistet `condit_2.g` genau so wenig wie `condit_1.g`.

```

1 -----1*****2*****3*****4*****5*****6*****7*****
2 -- condit_2.g: Ein (sehr einfaches) Telefonverzeichnis wird als
3 -- sortierter, binaerer Baum realisiert.
4 -----
5 'type' BAUM
6   nix
7   k(Linker: BAUM, Name: STRING, TelNr: INT, Rechter: BAUM)
8 -----
9 -- Zum Typ BAUM gehoeren u.a. die folgenden fuenf Terme:
10 --   nix
11 --   k(nix, "Bert", 234, nix)
12 --                                     k(nix, "Ralf", 345, nix)
13 -- k(k(nix, "Bert", 234, nix), "Otto", 123, nix)
14 -- k(k(nix, "Bert", 234, nix), "Otto", 123, k(nix, "Ralf", 345, nix))
15 -----
16 'condition' HatNr(Name: STRING, Verzeichnis: BAUM -> TelNr: INT)
17 -- Gelingt und liefert eine TelNr, wenn der Name im Verzeichnis steht:
18 -----
19 'rule' HatNr(NAME, k(BL, NAM1, NR, BR) -> NR):
20       eq(NAME, NAM1).
21 -- Gefunden, der NAME hat im Verzeichnis die NR:
22 'rule' HatNr(NAME, k(BL, NAM1, NR1, BR) -> NR):
23       lt(NAME, NAM1)
24       HatNr(NAME, BL -> NR).
25 -- Im linken Unterbaum BL weitersuchen:

```

```

26 'rule' HatNr(NAME, k(BL, NAM1, NR1, BR) -> NR):
27     gt(NAME, NAM1)
28     HatNr(NAME, BR -> NR).
29 -- Im rechten Unterbaum BR weitersuchen:
30 -----
31 -- Das Aktions-Praedikat TragEin gelingt und liefert ein Verzeichnis VNeu
32 -- (in dem der Name mit der TelNr eingetragen ist), wenn der Name noch
33 -- nicht im Verzeichnis VAlt eingetragen ist:
34 -----
35 'action' TragEin(Name: STRING, TelNr: INT, Valt: BAUM -> VNeu: BAUM)
36 -- Wenn der Name noch nicht im Verzeichnis Valt steht, wird ein
37 -- Verzeichnis VNeu geliefert, in dem der Name und die TelNr eingetragen
38 -- sind. Sonst wird das Verzeichnis Valt unveraendert geliefert:
39 -----
40 'rule' TragEin(NAME, NR, B -> B):
41     HatNr(NAME, B -> NR1).
42 -- Den Namen NICHT noch einmal eintragen, wenn er schon eine Nr hat.
43 'rule' TragEin(NAME, NR, nix -> k(nix, NAME, NR, nix)): .
44 -- Eintrag in einen leeren Baum nix.
45 'rule' TragEin(NAME, NR, k(BL, NAM1, NR1, BR) -> k(BN, NAM1, NR1, BR)):
46     lt(NAME, NAM1)
47     TragEin(NAME, NR, BL -> BN).
48 -- Eintrag in den linken Unterbaum BL.
49 'rule' TragEin(NAME, NR, k(BL, NAM1, NR1, BR) -> k(BL, NAM1, NR1, BN)):
50     gt(NAME, NAM1)
51     TragEin(NAME, NR, BR -> BN).
52 -- Eintrag in den rechten Unterbaum BR.
53 -----
54 'action' gib_alle_aus(Verzeichnis: BAUM, Ebene: INT)
55 -- Gibt alle Eintraege eines Verzeichnisses (und vor jedem die Nummer
56 -- seiner Ebene im Baum) aus:
57 -----
58 'rule' gib_alle_aus(nix, N): . -- leeres Verzeichnis, gib nichts aus!
59 'rule' gib_alle_aus(k(BL, NAME, NR, BR), EB):
60     gib_alle_aus(BL, EB+1) -- Linken Unterbaum
61     PutS(NAME) Tab PutI(NR) Tab -- Wurzel
62     PutS("Ebene ") PutI(EB) Nl
63     gib_alle_aus(BR, EB+1). -- Rechten Unterbaum
64 -----
65 -- Das Aktions-Praedikat gib_einen_aus gibt einen Eintrag eines Telefon-
66 -- verzeichnisses bzw. eine Fehlermeldung aus:
67 -----
68 'action' gib_einen_aus(Verzeichnis: BAUM, Name: STRING)
69 -- Wenn der Name mit einer TelNr im Verzeichnis steht, wird der Name und
70 -- die TelNr ausgegeben. Sonst wird der Name in einer Fehlermeldung
71 -- ausgegeben:
72 -----
73 'rule' gib_einen_aus(B, NAME):
74     HatNr(NAME, B -> NR)
75     PutS(NAME) PutS(" hat die Nr. ") PutI(NR) Nl
76 'rule' gib_einen_aus(B, NAME):
77     PutS(NAME) PutS(" steht nicht im Verzeichnis!") Nl
78 -----
79 -- Die folgenden Aktions-Praedikate werden hier nur vereinbart. Sie
80 -- wurden als C-Funktionen in der Datei text_io.c realisiert:
81 -----
82 'action' PutS(STRING) -- Gibt eine Zeichenkette zur aktuellen Ausgabe aus
83 'action' PutI(INT) -- Gibt eine Ganzzahl zur aktuellen Ausgabe aus

```

```

84 'action'  Tab          -- Gibt ein Tab-Zeichen zur aktuellen Ausgabe aus
85 'action'  Nl          -- Gibt ein newline-Z. zur aktuellen Ausgabe aus
86 -----
87 -- Das Wurzel-Praedikat (root predicate) ist das "Hauptprogramm":
88 -----
89 'root'
90 PutS("----- condit_2-Anfang") Nl
91 TragEin("Otto", 123, nix -> VERZ1)
92 TragEin("Bert", 234, VERZ1 -> VERZ2)
93 TragEin("Ralf", 345, VERZ2 -> VERZ3)
94 TragEin("Susi", 456, VERZ3 -> VERZ4)
95 TragEin("Carl", 567, VERZ4 -> VERZ5)
96 TragEin("Anna", 678, VERZ5 -> VERZ6)
97 gib_alle_aus (VERZ6, 0) Nl
98 gib_einen_aus(VERZ6, "Ralf")
99 gib_einen_aus(VERZ6, "Anne")
100 gib_einen_aus(VERZ6, "Anna")
101 PutS("----- condit_2-Ende") Nl
102 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 05 bis 07:** Ein Baum ist entweder leer (Zeile 06) oder er besteht aus einem Knoten. Dieser Knoten hat vier Komponenten: Einen linken (Unter-) Baum, einen Namen, eine Telefon-Nr und einen rechten (Unter-) Baum. Im Programm `condit_2.g` werden immer nur sortierte Bäume erzeugt, d.h. für jeden Knoten eines solchen Baumes gilt:

Im linken Unterbaum des Knotens sind alle Namen kleiner als der Name im Knoten und im rechten Unterbaum des Knotens sind alle Namen größer als der Name im Knoten.

**Zeile 16:** Das Bedingungs-Prädikat `HatNr` entspricht natürlich genau dem gleichnamigen Prädikat im vorigen Beispiel `condit_1.g`.

**Zeile 19 bis 20:** Wenn diese Regel für das Prädikat `HatNr` angewendet werden kann, dann wurde der NAME im Baum `k(BL, NAM1, NR, BR)` gefunden und hat die Telefon-Nr NR.

**Zeile 22 bis 24:** Wenn diese Regel angewendet wird, muss im linken Unterbaum BL des Baumes `k(BL, NAM1, NR, BR)` weitergesucht werden, weil der gesuchte NAME kleiner ist als NAM1.

**Zeile 26 bis 28:** Wenn diese Regel angewendet wird, muss im rechten Unterbaum BR des Baumes `k(BL, NAM1, NR, BR)` weitergesucht werden, weil der gesuchte NAME größer ist als NAM1.

**Zeile 35:** Mit dem Aktions-Prädikat `TragEin` kann man einen Namen und eine Telefon-Nr in ein Verzeichnis eintragen, aber nur, wenn der Name noch nicht in dem Verzeichnis vorkommt. Ob der Name schon in dem Verzeichnis vorkommt oder nicht, wird mithilfe des Bedingungs-Prädikates `HatNr` festgestellt.

**Zeile 40 bis 41:** Diese Regel (für das Prädikat `TragEin`) kann nur angewendet werden, wenn der Aufruf des Prädikats `HatNr` (in Zeile 41) gelingt. In diesem Fall wird der Baum B (d.h. das Telefonverzeichnis) unverändert als Ausgabeparameter zurückgeliefert. Falls der Aufruf von `HatNr` misslingt, wird versucht, die nächste Regel (Zeile 43 bis 44) anzuwenden.

**Zeile 43 bis 44:** Diese Regel (für das Prädikat `TragEin`) kann nur angewendet werden, wenn ein Eintrag in ein leeres Verzeichnis (`nix`) eingefügt werden soll. Dies ist die einzige der vier Regeln für `TragEin`, die tatsächlich einen Eintrag in einen Baum einfügt. Als Ausgabe-Parameter wird der Baum `k(nix, NAME, NR, nix)` zurückgeliefert.

**Zeile 45 bis 47:** Der NAME und die NR müssen im linken Unterbaum BL des Baumes `k(BL, NAM1, NR1, BR)` eingetragen werden, weil der NAME kleiner ist als NAM1.

**Zeile 49 bis 51:** Der NAME und die NR müssen im rechten Unterbaum BR des Baumes `k(BL, NAM1, NR1, BR)` eingetragen werden, weil der NAME größer ist als NAM1.

**Zeile 54:** Mit dem Aktions-Prädikat `gib_alle_aus` kann man alle Einträge eines Telefonverzeichnisses ausgeben. Da das Telefonverzeichnis hier als Baum realisiert wurde, entspricht jeder Knoten des Baumes einem Eintrag im Verzeichnis. Für jeden Eintrag (Knoten) wird nicht nur sein Name und seine Telefon-Nr ausgegeben, sondern auch seine "Ebene im Baum". Der Wurzelknoten des gesamten Baumes befindet sich dabei auf Ebene 0, die (maximal 2) Knoten, die direkt am Wurzelknoten hängen, bilden die Ebene 1, die (maximal 4) Knoten, die an direkt an den Knoten der Ebene 1 hängen, bilden die Ebene 2 etc.. Allgemein: Die maximal  $2^n$  Knoten, die direkt an den Knoten der Ebene  $n-1$  hängen, bilden die Ebene  $n$ .

Hier die Ausgabe des Programms `condit_2`:

```

1 ----- condit_2-Anfang
2 Anna      678   Ebene 2
3 Bert      234   Ebene 1
4 Carl      567   Ebene 2
5 Otto      123   Ebene 0
6 Ralf      345   Ebene 1
7 Susi      456   Ebene 2
8
9 Ralf hat die Nr. 345
10 Anne steht nicht im Verzeichnis!
11 Anna hat die Nr. 678
12 ----- condit_2-Ende
```

Man beachte, dass die sechs Einträge des Telefonverzeichnisses hier (aufsteigend) sortiert ausgegeben wurden (siehe Zeile 2 bis 7), d.h. in der Reihenfolge Anna, Bert, Carl, ... Otto, obwohl sie "völlig unsortiert" in das Telefonverzeichnis eingetragen wurden.

**Aufgabe 24.1.:** Betrachten Sie noch einmal die Ausgabe des Programms `condit_2`, insbesondere die Zeilen 2 bis 7. Wie sah der binäre Baum aus, durch den das Telefonverzeichnis realisiert wurde? Sie wissen von jedem Eintrag, auf welcher Ebene er stand und dass der Baum sortiert war. Damit sollte es Ihnen möglich sein, den Baum graphisch darzustellen.

**Aufgabe 24.2.:** Das Prädikat `gib_alle_aus` (siehe Zeile 54 bis 63 des Programms `condit_2`) gibt alle Einträge eines Telefonverzeichnisses so aus, dass die Namen in *aufsteigend* sortierter Reihenfolge erscheinen (zuerst Anna und zuletzt Otto). Ändern Sie das Prädikat so, dass es die Einträge in *absteigender* Reihenfolge ausgibt (zuerst Otto und zuletzt Anna).

## 25. sweep\_v1.g: Feger-Prädikate (sweep predicates) und Bäume

Das folgende Beispiel soll zeigen, wie man sogenannte *Feger-Prädikate* (sweep predicates) in einem Gentle-Programm anwenden kann. Feger-Prädikate leisten nichts Neues, man kann sie in jedem Fall durch andere Prädikate ersetzen. In bestimmten Fällen ist ein Feger-Prädikat aber eine bequeme und leicht lesbare Abkürzung für ein "kompliziertes System anderer Prädikate". Ein Feger-Prädikat kommt vor allem dann in Frage, wenn man "in einer großen und komplizierten" Datenstruktur, die aus vielen verschiedenartigen Knoten besteht, nur einige wenige Knoten bearbeiten will. Das Feger-Prädikat erleichtert es einem, die wenigen zu bearbeitenden Knoten zu finden und erspart einem das Hinschreiben "vieler trivialer Regeln" für die nicht zu bearbeitenden Knoten.

Auch `sweep_v1.g` dient nur dazu, bestimmte Konstrukte der Gentle-Sprache (nämlich Feger-Prädikate, sweep predicates) vorzuführen und leistet darüber hinaus nichts Sinnvolles. Das Programm baut einen binären Baum auf, der aus roten und schwarzen Knoten besteht. Ein (Unter-) Baum mit einem roten Wurzelknoten wird als roter Baum bezeichnet (auch wenn alle anderen Knoten schwarz sind), einen schwarzen (Unter-) Baum erkennt man an seinem schwarzen Wurzelknoten. Ein roter Knoten mit zwei roten Unterbäumen wird als rrr-Knoten bezeichnet, ein schwarzer Knoten mit einem roten und einem schwarzen Unterbaum wird als srs-Knoten bezeichnet etc.. Nach dem Aufbau des rot-schwarzen Binärbaumes werden nur die rsr-Knoten (d.h. die roten Knoten mit einem schwarzen und einem roten Unterbaum) "bearbeitet", d.h. gezählt bzw. ausgegeben. Das Zählen der rsr-Knoten erfolgt zweimal: einmal mit einem "normalen" Aktions-Prädikat und dann mit einem Feger-Prädikat. Damit soll deutlich gemacht werden, dass Feger-Prädikate in bestimmten Fällen "eleganter und leichter lesbar" sind als andere Prädikate.

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- sweep_v1.g: Einige Praedikate zum Bearbeiten von Baeumen sollen den
3  -- Sinn von Feger-Praedikaten illustrieren.
4  -----
5  'type'  RB -- Rote Baeume
6  -- Ein roter Baum ist entweder leer (rnix), oder er besteht aus
7  -- einem roten Knoten mit zwei Unterbaeumen daran. Die Unterbaeume
8  -- koennen rot oder schwarz sein.
9  rnix          -- leerer   roter Baum
10 rrr(RB, RB, INT) -- mit zwei roten                Unterbaeumen
11 rrs(RB, SB, INT) -- mit einem roten    und einem schwarzen Unterbaum
12 rsr(SB, RB, INT) -- mit einem schwarzen und einem    roten Unterbaum
13 rss(SB, SB, INT) -- mit zwei schwarzen                Unterbaeumen
14 -----
15 'type'  SB -- Schwarze Baeume
16 -- Ein schwarzer Baum ist entweder leer (snix) oder er besteht aus
17 -- einem schwarzen Knoten, mit zwei Unterbaeumen daran. Die Unterbaeume
18 -- koennen schwarz oder rot sein.
19 snix          -- leerer   schwarzer Baum
20 sss(SB, SB, INT) -- mit zwei schwarzen                Unterbaeumen
21 ssr(SB, RB, INT) -- mit einem schwarzen und einem roten    Unterbaum
22 srs(RB, SB, INT) -- mit einem roten    und einem schwarzen Unterbaum
23 srr(RB, RB, INT) -- mit zwei roten                Unterbaeumen
24 -----
25 -- Ein rsr-Knoten ist ein roter Knoten, der links einen schwarzen und
26 -- rechts einen roten Unterbaum besitzt.
27 -----
28 'action' anz_rsr_r(Baum: RB, Bisher: INT -> Ergebnis: INT)

```

```

29 -- Liefert die Anzahl der rsr-Knoten eines roten Baumes
30 'rule' anz_rsr_r(rnix, NO -> NO ): .
31 'rule' anz_rsr_r(rrr(RB1, RB2, N), NO -> N2):
32     anz_rsr_r(RB1, NO -> N1)
33     anz_rsr_r(RB2, N1 -> N2)
34 'rule' anz_rsr_r(rrs(RB1, SB2, N), NO -> N2):
35     anz_rsr_r(RB1, NO -> N1)
36     anz_rsr_s(SB2, N1 -> N2)
37 'rule' anz_rsr_r(rsr(SB1, RB2, N), NO -> N2 + 1):
38     anz_rsr_s(SB1, NO -> N1)
39     anz_rsr_r(RB2, N1 -> N2)
40 'rule' anz_rsr_r(rss(SB1, SB2, N), NO -> N2):
41     anz_rsr_s(SB1, NO -> N1)
42     anz_rsr_s(SB2, N1 -> N2)
43 -----
44 'action' anz_rsr_s(Baum: SB, Bisher: INT -> Ergebnis: INT)
45 -- Liefert die Anzahl der rsr-Knoten eines schwarzen Baumes
46 'rule' anz_rsr_s(snix, NO -> NO ): .
47 'rule' anz_rsr_s(sss(SB1, SB2, N), NO -> N2):
48     anz_rsr_s(SB1, NO -> N1)
49     anz_rsr_s(SB2, N1 -> N2)
50 'rule' anz_rsr_s(ssr(SB1, RB2, N), NO -> N2):
51     anz_rsr_s(SB1, NO -> N1)
52     anz_rsr_r(RB2, N1 -> N2)
53 'rule' anz_rsr_s(srs(RB1, SB2, N), NO -> N2):
54     anz_rsr_r(RB1, NO -> N1)
55     anz_rsr_s(SB2, N1 -> N2)
56 'rule' anz_rsr_s(srr(RB1, RB2, N), NO -> N2):
57     anz_rsr_r(RB1, NO -> N1)
58     anz_rsr_r(RB2, N1 -> N2)
59 -----
60 'sweep' anz_rsr(Baum: ANY, Bisher: INT -> Ergebnis: INT)
61 -- Liefert die Anzahl der rsr-Knoten eines Baumes. Leistet also
62 -- gleichviel wie die Praedikate anz_rsr_r und anz_rsr_s zusammen!
63 'rule' anz_rsr(rsr(RB1, SB2, N), NO -> N2 + 1):
64     anz_rsr (RB1, NO -> N1)
65     anz_rsr (SB2, N1 -> N2)
66 -----
67 'sweep' rsr_tiefe(Baum: ANY, Ebene: INT)
68 -- Liefert "die Tiefe" aller rsr-Knoten eines Baumes. Bei der
69 -- Berechnung der Tiefe zaehlen nur rsr-Knoten, alle anderen werden
70 -- "ignoriert".
71 'rule' rsr_tiefe(rsr(RB1, SB2, N), EB):
72     PutS("Ein rsr-Knoten auf rsr-Ebene ") PutI(EB) N1
73     rsr_tiefe(RB1, EB+1)
74     rsr_tiefe(SB2, EB+1)
75 -----
76 'sweep' aus_rsr(Baum: ANY)
77 -- Gibt die INT-Komponenten aller rsr-Knoten eines Baumes aus:
78 'rule' aus_rsr(rsr(RB1, SB2, N)):
79     PutS("INT-Komponente eines rsr-Knotens : ") PutI(N) N1
80     aus_rsr(RB1)
81     aus_rsr(SB2)
82 -----
83 'action' mach_einen_testbaum(->RB)
84 -- Liefert einen (roten) Baum, der dann als "Testdatum" fuer die
85 -- Praedikate anz_rsr_r, anz_rsr, rsr_tiefe etc. dienen kann.
86 'rule' mach_einen_testbaum(->ERG_BAUM):

```

```

87     where(rsr(sss(srr(rnix, rnix, 1),
88                 srr(snix, rnix, 2),
89                 3),
90             rrr(rsr(snix, rnix, 4),
91                 rsr(snix,
92                     rss(snix, snix, 5),
93                     6),
94                 7),
95                 8)
96     ->   ERG_BAUM)
97 -----
98 -- Die folgenden Aktions-Praedikate werden hier nur vereinbart. Sie
99 -- wurden als C-Funktionen in der Datei text_io.c realisiert:
100 -----
101 'action'  PutS(String) -- Gibt eine Zeichenkette zur aktuellen Ausgabe aus
102 'action'  PutI(Int)    -- Gibt eine Ganzzahl zur aktuellen Ausgabe aus
103 'action'  Tab         -- Gibt ein Tab-Zeichen zur aktuellen Ausgabe aus
104 'action'  Nl          -- Gibt ein newline-Z. zur aktuellen Ausgabe aus
105 -----
106 -- Das Wurzel-Praedikat (root predicate) ist das "Hauptprogramm":
107 -----
108 'root'
109   PutS("----- sweep_v1-Anfang") Nl
110   mach_einen_testbaum(->RB1)
111   anz_rsr_r(RB1, 0 -> ERG1)
112   PutS("Anzahl rsr-Knoten (action-Version): ") PutI(ERG1) Nl
113   anz_rsr (RB1, 0 -> ERG2)
114   PutS("Anzahl rsr-Knoten ( sweep-Version): ") PutI(ERG2) Nl
115   rsr_tiefe(RB1, 0)
116   aus_rsr (RB1)
117   PutS("----- sweep_v1-Ende") Nl
118 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 05 bis 23:** Jeder Wert des Typs RB ist ein roter Baum, jeder Wert des Typs SB ist ein schwarzer Baum. Jeder Baum kann sowohl rote als auch schwarze Unterbäume haben. Man beachte, dass es zwei verschiedene "leere Bäume" gibt: einen leeren roten Baum (`rnix`) und einen leeren schwarzen Baum (`snix`).

Angenommen, wir haben einen Baum, der aus beliebig vielen roten und/oder schwarzen Knoten besteht und wir wollen wissen, wieviele `rsr`-Knoten dieser Baum enthält. Dann können wir zum Zählen dieser Knoten geeignete Aktions-Prädikate vereinbaren. Da wir mit schwarzen und roten (Unter-) Bäumen rechnen müssen, brauchen wir entsprechend auch zwei (Aktions-) Prädikate (die hier `anz_rsr_r` und `anz_rsr_s` genannt wurden).

**Zeile 28 bis 42:** Das Aktions-Prädikat `anz_rsr_r` hat zwei Eingabe-Parameter (Baum vom Typ RB, d.h. einen roten Baum, und Bisher vom Typ INT) und einen Ausgabe-Parameter (Ergebnis vom Typ INT). Es addiert die Anzahl der `rsr`-Knoten in dem roten Baum zur Bisher-Knotenzahl und liefert die Summe als Ergebnis. Das Prädikat `anz_rsr_r` ruft das Prädikat `anz_rsr_s` und sich selbst (rekursiv) auf, um die Anzahl der `rsr`-Knoten in den Unterbäumen des roten Baumes zu zählen.

Zum Prädikat `anz_rsr_r` gehören 5 Regeln (rules). Eigentlich ist nur die vierte Regel (für Bäume mit einem `rsr`-Wurzelknoten, siehe Zeile 37 bis 39) interessant, denn nur in dieser Regel wird `+1` gerechnet. Die anderen vier Regeln sind "trivial", können aber Blasen am Daumen des Programmierers verursachen.

**Zeile 44 bis 58:** Das Aktions-Prädikat `anz_rsr_s` hat zwei Eingabe-Parameter (Baum vom Typ `SB`, d.h. einen schwarzen Baum und `Bisher` vom Typ `INT`) und einen Ausgabe-Parameter (Ergebnis vom Typ `INT`). Es addiert die Anzahl der `rsr`-Knoten in dem schwarzen Baum zur `Bisher`-Knotenzahl und liefert die Summe als Ergebnis. Das Prädikat `anz_rsr_s` ruft das Prädikat `anz_rsr_r` und sich selbst (rekursiv) auf, um die Anzahl der `rsr`-Knoten in den Unterbäumen des schwarzen Baumes zu zählen.

Zum Prädikat `anz_rsr_s` gehören 5 Regeln, die alle "trivial" sind, beim Lesen des Programms aber trotzdem einige Zeit kosten können, bis man sie verstanden hat.

**Zeile 60 bis 65:** Das Feger-Prädikat (sweep predicate) `anz_rsr` leistet genau das gleiche, wie die beiden Aktions-Prädikate `anz_rsr_r` und `anz_rsr_s` zusammen, erzeugt aber keine Blasen am Daumen des Programmierers und kostet beim Lesen nur wenig Zeit. Das Prädikat `anz_rsr` hat zwei Eingabe-Parameter (`Baum` und `Bisher`) und einen Ausgabe-Parameter (`Ergebnis`). Die Typ-Angabe `ANY` besagt, dass der Baum-Parameter von einem beliebigen Typ ("any type") sein darf. Damit wird das starke Typensystem der Sprache Gentle an dieser Stelle gezielt "gelockert", um eine kurze und leichter lesbare Notation zu ermöglichen. Zum Feger-Prädikate `anz_rsr` gehört hier nur eine Regel (siehe Zeile 63 bis 65), nämlich die "interessante Regel", in der `+1` gerechnet wird. Die "trivialen Regeln" braucht man nicht hinzuschreiben, sie verstehen sich bei einem Feger-Prädikat von selbst.

**Zeile 67 bis 74:** Wenn man in einem Baum vom Wurzelknoten zu einem bestimmten `rsr`-Knoten `K` geht und dabei alle `rsr`-Knoten zählt, an denen man vorbeikommt, dann erhält man die `rsr`-Tiefe des `rsr`-Knotens `K`. Das Feger-Prädikat `rsr_tiefe` gibt die `rsr`-Tiefen aller `rsr`-Knoten eines (roten oder schwarzen) Baumes aus.

**Zeile 76 bis 81:** Aus den Vereinbarungen der Typen `RB` und `SB` (siehe Zeile 05 bis 13 bzw. 15 bis 23) geht hervor, dass jeder Knoten eines Baumes auch eine `INT`-Komponente besitzt. Mit dem Feger-Prädikat `aus_rsr` kann man die `INT`-Komponenten aller `rsr`-Knoten eins (roten oder schwarzen) Baumes ausgeben lassen.

**Zeile 83 bis 96:** Das Prädikat `mach_einen_testbaum` dient dazu, einen bestimmten (im Prinzip aber beliebigen) Baum zur erzeugen, der dann als Testdatum für die oben vereinbarten Prädikate `anz_rsr_r`, `anz_rsr`, `rsr_tiefe` und `aus_rsr` dienen kann. Will man die Prädikate mit einem anderen Baum testen, dann kann man dazu das Prädikat `mach_einen_testbaum` entsprechend ändern.

Das Aktions-Prädikat `where` ist in Gentle vorvereinbart. Es hat genau einen Eingabe-Parameter und einen Ausgabe-Parameter, die von einem beliebigen Typ sein dürfen, aber beide vom selben Typ sein müssen. Das Prädikat `where` weist seinem Ausgabe-Parameter den Wert seines Eingabe-Parameters zu. Hier wird dem Ausgabe-Parameter `ERG_BAUM` ein Term (`rsr(sss(...), rrr(...), 8)`) zugewiesen, der einen roten Baum (vom Typ `RB`) darstellt. Dieser Baum hat insgesamt 8 nicht-leere Unterbäume und 9 `nix`-Unterbäume (`rnix` bzw. `snix`). Es empfiehlt sich, diesen Baum mit Bleistift (und Radiergummis) graphisch darzustellen, um sich mit seinen Knoten und seiner Struktur vertraut zu machen.

Hier die Ausgabe des Programms sweep\_v1:

```

1 ----- sweep_v1-Anfang
2 Anzahl rsr-Knoten (action-Version): 3
3 Anzahl rsr-Knoten ( sweep-Version): 3
4 Ein rsr-Knoten auf rsr-Ebene 0
5 Ein rsr-Knoten auf rsr-Ebene 1
6 Ein rsr-Knoten auf rsr-Ebene 1
7 INT-Komponente eines rsr-Knotens : 8
8 INT-Komponente eines rsr-Knotens : 4
9 INT-Komponente eines rsr-Knotens : 6
10 ----- sweep_v1-Ende

```

**Wichtige Anmerkung:** Ein Feger-Prädikat darf nicht beliebig viele Eingabe- und/oder Ausgabe-Parameter haben. Vielmehr ist die Anzahl und die Art der Parameter genau festgelegt und entsprechend unterscheidet man drei Arten von Feger-Prädikaten:

```

1 'sweep' feger_art1(Ein1: ANY, Ein2: T -> Aus1: T)
2 'sweep' feger_art2(Ein1: ANY, Ein2: T)
3 'sweep' feger_art3(Ein1: ANY)

```

Dabei steht **T** hier für einen beliebigen Typen, den man in der Vereinbarung des Feger-Prädikates aber konkret angeben muss. Ein Feger-Prädikat hat also entweder

- zwei Eingabe-Parameter und einen Ausgabe-Parameter (feger\_art1) oder
- zwei Eingabe-Parameter (feger\_art2) oder
- einen Eingabe-Parameter (feger\_art3)

und der erste Eingabe-Parameter muss immer den "Joker-Typ" ANY haben. Im Programm sweep\_v1.g gibt es für jede Art von Feger-Prädikat genau ein konkretes Beispiel:

**Art von Feger-Prädikat:**            **Beispiel im Programm sweep\_v1:**

```

feger_art1            anz_rsr
feger_art2            rsr_tiefe
feger_art3            aus_rsr

```

Diese Einschränkung auf drei Arten von Feger-Prädikaten ist notwendig, damit der Gentle-Compiler alle trivialen Regeln automatisch ergänzen kann. Außerdem ist die Einschränkung sinnvoll, weil man in der Praxis genau diese drei Arten von Feger-Prädikaten braucht.

## 26. Spezielle Muster und Terme

Viele Terme (Grundterme und Terme mit Variablen) können sowohl als *Muster* als auch als *Ausdruck* verwendet werden. Der Leser eines Gentle-Programms muss jeweils aus dem Zusammenhang eines Terms schließen, ob es sich um ein Muster handelt oder um einen Ausdruck. Das gilt z. B. für alle Terme eines vom Programmierer definierten Typs. Für Terme der vordefinierten Typen `INT` und `STRING` gelten etwas abweichende Regeln (siehe unten).

Einige Zeichenketten dürfen nur als Muster, andere nur als Ausdrücke verwendet werden. Diese speziellen Muster und Ausdrücke werden im folgenden beschrieben.

Das Egal-Zeichen (ein einzelner Unterstrich `_`) darf nur als Muster oder als Teil eines Musters verwendet werden, sozusagen als "Ersatz für einen Variablen-Namen". Es bedeutet: "An dieser Stelle darf irgendein Wert stehen, auf den später aber keine Bezug mehr genommen wird und mit dem deshalb keine Variable belegt wird" oder kurz: "Es ist egal, welcher Wert an dieser Stelle steht".

### Beispiel 26.1.: Muster mit Egal-Zeichen

Für das Prädikat `Rest2` wurde im vorigen Abschnitt folgende Regel vereinbart:

```
'rule' Rest2(list(E1, list(E2, L)) -> L): .
```

Die Variablen `E1` und `E2` werden zwar definiert, aber nirgends angewendet. Hier eine elegantere (und möglicherweise etwas einfacher zu lesende) Formulierung der gleichen Regel:

```
'rule' Rest2(list(_, list(_, L)) -> L): .
```

Mit dem Egal-Zeichen kann man also Variablen vermeiden, die zwar definiert aber nicht angewendet werden (d.h. die in einem Muster, aber in keinem Ausdruck vorkommen).

In einem *Ausdruck* darf ein Egal-Zeichen grundsätzlich nicht vorkommen, auch wenn aus dem Zusammenhang eindeutig hervorzugehen scheint, für welchen Wert es steht.

### Beispiel 26.2.: Das Egal-Zeichen ist *kein* Variablen-Name

Die folgende Regel ist syntaktisch falsch, weil ein Egal-Zeichen im Ausdruck rechts vom Pfeil `->` vorkommt:

```
'rule' Identitaet(list(_, leer) -> list(_, leer)): .
```

Die folgende Regel ist syntaktisch korrekt:

```
'rule' Identitaet(list(E, leer) -> list(E, leer)): .
```

In einem Gentle-Programm dürfen zwei Arten von Literalen vorkommen: Ganzzahl-Literale (z. B. `0`, `-47`, `+123`) und Zeichenketten-Literale (z. B. `"Hallo!"`, `"ABC"`, `" "`, `" "" "`). Ein Literal ist in Gentle grundsätzlich ein *Ausdruck* und darf nicht als Muster oder Teil eines Musters verwendet werden.

### Beispiel 26.2.: Verbotene Literal-Muster

Die folgenden beiden Prädikatvereinbarungen sind nicht erlaubt, weil darin Literale als Muster verwendet werden:

```
1 'condition' InWorten(INT -> STRING)
2   'rule'    InWorten(0 -> "Null"): .
3   'rule'    InWorten(1 -> "Eins"): .
4   'rule'    InWorten(2 -> "Zwei"): .
5   ...
```

```

6
7 'condition' Translate (STRING -> STRING)
8 'rule' Translate("zero" -> "Null"): .
9 'rule' Translate("one" -> "Eins"): .
10 'rule' Translate("two" -> "Zwei"): .
11 ...

```

Die folgenden beiden Prädikatvereinbarungen sind dagegen erlaubt:

```

1 'condition' InWorten (INT -> STRING)
2 'rule' InWorten(N -> "Null"): eq(N, 0).
3 'rule' InWorten(N -> "Eins"): eq(N, 1).
4 'rule' InWorten(N -> "Zwei"): eq(N, 2).
5 ...
6
7 'condition' Translate (STRING -> STRING)
8 'rule' Translate(S -> "Null"): eq(S, "zero").
9 'rule' Translate(S -> "Eins"): eq(S, "one").
10 'rule' Translate(S -> "Zwei"): eq(S, "two").
11 ...

```

Die Idee, Literale als Muster zu verwenden, ist nicht von vornherein schlecht oder gefährlich. Möglicherweise wird eine zukünftige Version von Gentle Literale als Muster zulassen.

Neben der Möglichkeit, ganz normale *Prädikate* zu vereinbaren, gibt es in Gentle auch ein paar sehr merkwürdige und exotische Gebilde namens +, -, \* und /, die man auch als vordefinierte *Funktionen* bezeichnet. Diese Funktionen kann man (ähnlich wie Prädikate) aufrufen. Allerdings muss man dazu eine sehr unnatürliche Notation verwenden, die aus dem Mittelalter stammt und als *Infixnotation* bezeichnet wird. Hier ein paar Beispiele für Funktions-Aufrufe in Infixnotation:

1 + 1, A - B - C, A + 17 \* B / 5 etc. Der exotische Charakter dieser sogenannten *Funktionen* wird noch dadurch verschärft, dass der Name + zwei verschiedene Funktionen bezeichnet: Eine einstellige Funktion ("positives Vorzeichen") und eine zweistellige Funktion ("Addition zweier Ganzzahlen"). Ganz entsprechendes gilt für den Namen - ("Minus").

Die sechs sogenannten Funktionen mit den vier Namen +, -, \* und / sind gedacht als Ersatz für die Prädikate

```

1 'action' Plus (Summand1: INT, Summand2: INT -> Summe: INT)
2 'action' Minus (Minuend: INT, Subtrahend: INT -> Differenz: INT)
3 'action' Mal (Faktor1: INT, Faktor2: INT -> Produkt: INT)
4 'action' Durch (Dividend: INT, Divisor: INT -> Quotient: INT)
5 'action' Pos (Argument: INT -> Ergebnis: INT)
6 'action' Neg (Argument: INT -> Ergebnis: INT)

```

mit denen man zwei Ganzzahlen vom Typ INT addieren, subtrahieren, multiplizieren, dividieren bzw. eine Ganzzahl unverändert lassen (Pos) oder "vorzeichenmäßig umdrehen" (Neg) kann. Glücklicherweise gibt es in Gentle nur die hier erwähnten sechs (in Ziffern: 6) Funktionen und auch ein "wildentschlossener Programmierer" kann keine weiteren Funktionen hinzufügen (auch dann nicht, wenn er bereit ist, in C zu programmieren).

**Anmerkung:** Es soll Programmierer geben, die *Funktionen* mit ihren zweideutigen Namen und ihrer schwer lesbaren Infixnotation für "einfacher und natürlicher" halten, als *Prädikate* (mit ihren eindeutigen Namen und ihrer leicht lesbaren Präfixnotation). Vermutlich ist eine solche Vorliebe für Funktionen auf eine harte Jugend zurückzuführen, in der nur Fortran, C oder Java gesprochen wurde.

Funktionsaufrufe sind in einem *Muster* grundsätzlich nicht erlaubt und werden nur in *Ausdrücken* geduldet.

**Beispiel 26.3.:** Funktionsaufrufe sind nur in Ausdrücken erlaubt, aber nicht in Mustern:

```

1 'action'  Minus1(INT -> INT)
2   'rule'  Minus1(N + 1 -> N   ): . -- falsch: Funktionsaufruf in Muster
3   'rule'  Minus1(N       -> N -1): . -- erlaubt: Funktionsaufruf in Ausdruck
4
5 'action'  Wurzel(INT -> INT)
6   'rule'  Wurzel(N * N -> N): .   -- falsch: Funktionsaufruf in Muster
7
8 'action'  Quadrat(INT -> INT)
9   'rule'  Quadrat(N -> N * N): .  -- erlaubt: Funktionsaufruf in Ausdruck

```

**Aufgabe 26.1.:** Vereinbaren Sie ein Prädikat namens `Wurzel`, welches aus einer (nicht-negativen) Ganzzahl die ganzzahlige Wurzel zieht (Beispiele: Die ganzzahlige Wurzel aus 9 ist 3, die ganzzahlige Wurzel aus 8 oder 5 oder 4 ist 2, die ganzzahlige Wurzel aus 3 ist 1 etc.).

**Anmerkung:** Die vorige Aufgabe mit dem `Wurzel`-Prädikat soll eine Anregung sein, die Möglichkeiten der "Prädikat-Programmierung" in Gentle zu erkunden. In einem "ernsthaften Gentle-Programm" würde man ein `Wurzel`-Prädikat nicht in Gentle, sondern in C schreiben.

**Benannte Muster** (oder: Muster mit Namen) sind in bestimmten Spezialfällen eine bequeme und gut lesbare "Abkürzung" für kompliziertere Formulierungen. Als Beispiel soll das folgende Bedingungs-Prädikat dienen:

```
'condition' Mindestens2(Eingabe: LISTE -> Ausgabe: LISTE)
```

Ein Aufruf dieses Prädikates soll nur dann gelingen, wenn die Eingabe-LISTE mindestens 2 Elemente hat. In diesem Fall soll die Eingabe-LISTE unverändert zum Wert der Ausgabe-LISTE gemacht werden. Hier eine mögliche Regel für dieses Prädikat:

**Beispiel 26.4.:** Eine etwas komplizierte Formulierung

```
'rule' Mindestens2(list(E1, list(E2, R2)) -> list(E1, list(E2, R2))): .
```

Für einen Leser dieser Vereinbarung ist es nicht ganz trivial sich zu vergewissern, dass links und rechts vom Pfeil `->` genau der gleiche Term steht (links als Muster und rechts als Ausdruck).

**Beispiel 26.4.:** Eine einfachere Formulierung mit einem benannten Muster:

```
'rule' Mindestens2(L:list(E1, list(E2, R2)) -> L): .
```

Das Muster `list(E1, list(E2, R2))` wurde hier mit dem (Variablen-) Namen `L` versehen. Wenn ein aktueller Eingabeparameter mit dem Muster abgeglichen werden kann, dann wird der gesamte Eingabeparameter als Wert der Variablen `L` festgelegt. Außerdem werden natürlich (wie immer) entsprechende "Teile des Eingabeparameters" als Werte für die Variablen `E1`, `E2` und `R2` festgelegt. Da die Variablen `E1`, `E2` und `R2` in diesem Beispiel nirgendwo angewendet werden, kann man sie auch durch Egal-Zeichen ersetzen, etwa so:

```
'rule' Mindestens2(L:list(_, list(_, _)) -> L): .
```

**Beispiel 26.5.:** Das Prädikat `Add` addiert die vordersten beiden Zahlen einer Liste und fügt die Summe als vorderstes Element in die Liste ein (ohne die addierten Zahlen zu entfernen):

```

1 'condition' Add(LISTE -> LISTE)
2   'rule'    Add(L: list(E1, list(E2, _)) -> list(E1+E2, L)): .

```

## 27. Unveränderbare Variablen und veränderbare Variablen

In allen konventionellen Programmiersprachen kann man sogenannte *Konstanten* vereinbaren, z. B. so:

```
MAX : constant integer := 17; -- Eine Konstante in Ada
int const MAX = 17;         -- Eine Konstante in C
final int MAX = 17;         -- Eine Konstante in Java
```

Wenn der betreffende Ausführer diese Vereinbarung abarbeitet (während der Ausführung des umfassenden Programms) erzeugt er eine Konstante namens MAX mit dem Wert 17. Der Wert einer solchen Konstanten kann während ihrer gesamten Lebenszeit nicht verändert werden.

In Pascal und Modula kann man nur sogenannte *statische* Konstanten vereinbaren. Den Wert einer statischen Konstanten kann man ermitteln, indem man den ("statischen") Programmtext durchliest (und ohne zu wissen, welche Daten das Programm später mal einlesen wird). In C, Java und Ada kann man auch *dynamische* Konstanten vereinbaren. Der Wert einer solchen dynamischen Konstanten kann z. B. von ("dynamisch") eingelesenen Werten oder von berechneten Werten abhängen. Wird eine Konstantenvereinbarung mehrmals ausgeführt, so wird jedesmal eine neue Konstante erzeugt. Alle aufgrund einer Konstantenvereinbarung erzeugten Konstanten können verschiedene (oder gleiche) Werte haben, wie das folgende Beispiel deutlich machen soll.

**Beispiel 27.1.:** Eine Vereinbarung, viele Konstanten (ein Beispiel in C)

```
1 int quadratSumme(int n1, int n2) {
2     // Liefert das Quadrat der Summe n1 + n2 als Ergebnis:
3     const int sum = n1 + n2;
4     return sum * sum;
5 }
```

Bei jedem Aufruf der Funktion `quadratSumme` wird eine neue (dynamische) Konstante namens `sum` erzeugt. Die Werte verschiedener `sum`-Konstanten können sich ohne weiteres voneinander unterscheiden.

**Anmerkung:** In "alltäglichen Gesprächen unter Programmierern" ist es leider üblich, von der Konstanten `sum` (Einzahl!) zu sprechen. Realistischer und klarer wäre es, von *den* Konstanten `sum` (Mehrzahl!) oder von *einer* (der möglicherweise zahlreichen) Konstanten `sum` zu reden (z. B. "Alle Konstanten namens `sum` gehören zum Typ `int`" statt "Die Konstante `sum` gehört zum Typ `int`").

Alle Konstanten namens `sum`, die der Ausführer aufgrund der obigen Vereinbarungen erzeugt, entsprechen zusammengenommen ziemlich genau dem, was Mathematiker schon seit langem als eine (Einzahl!) *Variable* bezeichnen. Wenn ein mathematischer Text z. B. mit dem Satz beginnt: "Sei  $N$  eine ganze Zahl", dann bedeutet das etwa folgendes:

Man darf für  $N$  eine beliebige ganze Zahl als Wert festlegen und im folgenden Text alle Vorkommen von  $N$  durch diesen Wert ersetzen. Man darf aber den Wert, den man für  $N$  gewählt hat, nicht "mitten im Text" verändern. Nur wenn man den Text "erneut von Anfang an liest", darf man für  $N$  erneut einen Wert festlegen. Kurz: Für mathematische Variablen gibt es keine Zuweisungs-Anweisung, mit der man ihren einmal festgelegten Wert "mitten in ihrem Gültigkeitsbereich" verändern könnte.

Die Funktion `quadratSumme` hat Ähnlichkeit mit einem mathematischen Text und `sum` entspricht einer mathematischen Variablen: Jedesmal, wenn die Funktion `quadratSumme` ausgeführt wird, berechnet der Ausführer einen neuen Wert für `sum`, aber dieser Wert ist während der Ausführung der Funktion unveränderbar.

Ein typisches Gentle-Programm besteht zu einem wesentlichen Teil aus Vereinbarungen und Aufrufen von Prädikaten. Diese Vereinbarungen und Aufrufe können Muster und Ausdrücke enthalten, die ihrerseits Variablen enthalten können. All diese Variablen sind mathematische Variablen, d.h. sie haben große Ähnlichkeit mit den dynamischen Konstanten SUM im Beispiel 27.1.

Die meisten Probleme, die beim Bau eines Compilers auftauchen, kann man mit mathematischen Variablen (d.h. mit Konstanten) elegant und effizient lösen. Es gibt aber ein paar Probleme, bei denen das nicht der Fall ist. Für diese relativ seltenen "Sonderfälle" gibt es in Gentle auch Variablen, deren Wert man mithilfe einer Zuweisungs-Anweisung verändern kann. Deshalb wird im folgenden zwischen *unveränderbaren Variablen* und *veränderbaren Variablen* unterschieden.

Bevor eine unveränderbare Variable erzeugt wird, kann ihr Wert beliebig "festgelegt" (d.h. eingelesen oder berechnet) werden. Die Beliebigkeit und Dynamik dieser Festlegung eines Wertes rechtfertigt die Bezeichnung "Variable". Sobald eine unveränderbare Variable erzeugt und mit einem Wert versehen wurde, ist sie konstant, d.h. ihr Wert kann dann während ihrer gesamten Lebensdauer nicht mehr geändert werden. Veränderbare Variablen haben große Ähnlichkeit mit den C-Konstanten namens SUM im Beispiel 27.1. Für unveränderbare Variablen gibt es keine Zuweisungs-Anweisung. Unveränderbare Variablen kann man auch als *mathematische Variablen* oder als *dynamische Konstanten* oder als *Informatik-Konstanten* bezeichnen.

Den Wert einer veränderbaren Variablen kann man beliebig oft z. B. mithilfe von Zuweisungs-Anweisungen verändern. Wenn eine Anweisung den Wert einer solchen Variablen verändert, dann sagt man auch: Die Anweisung hat einen Seiteneffekt auf die Variable. Veränderbare Variablen haben große Ähnlichkeit mit den Gebilden, die man in konventionellen Programmiersprachen als Variablen bezeichnet. Veränderbare Variablen kann man auch als *Informatik-Variablen* oder als *programmiersprachliche Variablen* oder als *Wertebehälter* bezeichnen.

Bei den veränderbaren Variablen unterscheidet man in Gentle zwei Arten:

**Veränderbare Variablen mit Namen** (oder:vereinbarte Variablen): Eine solche Variable wird vom Ausführer erzeugt und mit einem Namen versehen, wenn er ihre Vereinbarung abarbeitet. Wenn man den momentanen Wert einer solchen Variablen "lesen" oder verändern will, muss man in dem betreffenden Befehl den Namen der Variablen nennen ("anwenden").

**Veränderbare Variablen mit Zeiger** (oder: allokierte Variablen): Eine solche Variable wird vom Ausführer erzeugt, wenn er einen sogenannten *Allokator* auswertet (entspricht dem new in Pascal, Modula, Ada und Java bzw. den Funktionen malloc, calloc etc. in älteren C-Dialekten). Der Aufruf eines Allokators liefert einen Zeigerwert, der auf die Variable zeigt. Wenn man den momentanen Wert einer solchen Variablen "lesen" oder verändern will, muss man in dem betreffenden Befehl den Zeigerwert angeben, der auf die Variable zeigt.

Einen Zeigerwert kann man nur als Wert einer (veränderbaren oder unveränderbaren) Variablen "aufbewahren". Eine solche Variable bezeichnet man als *Zeigervariable*.

**Anmerkung:** Der Begriff *Zeiger* ist auf eine häufig verwirrende Weise mehrdeutig: Er läßt offen, ob man einen *Zeigerwert* oder eine *Zeigervariable* meint. Wenn man sich genau ausdrücken will, sollte man deshalb die Begriffe *Zeigerwert* und *Zeigervariable* verwenden.

Für die Sichtbarkeit und Lebensdauer von veränderbaren Variablen gilt in Gentle Folgendes:

**Veränderbare Variablen mit Namen** kann man nur "auf der obersten Ebene eines Gentle-Programms" vereinbaren (und nicht als lokale Variablen eines Prädikates oder einer Regel etc.). Daraus folgt: Solche Variablen werden zu Beginn einer Programmausführung erzeugt und leben bis zum Ende der Programmausführung. Außerdem sind sie in allen Prädikaten ("überall und jederzeit") sichtbar. Deshalb werden sie auch als *globale Variablen* bezeichnet.

**Veränderbare Variablen mit Zeiger** können "jederzeit" während der Ausführung eines Gentle-Programms erzeugt werden (nicht nur zu Beginn einer Programmausführung) und leben bis zum Ende der Programmausführung. Wenn man den Zeigerwert, der auf eine solche allokierte Variable zeigt, zum Wert einer Variablen  $Z$  macht, dann kann man überall dort auf die allokierte Variable zugreifen, wo  $Z$  sichtbar ist. Dabei ist  $Z$  typischerweise eine veränderbare Variable (d.h. eine globale Variable), kann aber auch eine unveränderbare Variable (d.h. eine regellokale Variable) sein.

Das folgende Beispiel-Programm `variav_v1` soll zeigen, wie man in einem Gentle-Programm mit veränderbaren Variablen umgehen kann, d.h. wie man solche Variablen erzeugen lassen, wie man ihnen Werte zuweisen und wie man ihre momentanen Werte "lesen" kann:

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- variav_v1.g: Demonstriert die Verwendung von ver?nderbaren Variablen mit
3  -- Namen (Name1 und Alter1) und ver?nderbaren Variablen mit Zeigern
4  -----
5  'var' Name1 : STRING -- eine ver?nd. Variable namens Name1 vom Typ STRING
6  'var' Alter1: INT   -- eine ver?nd. Variable namens Alter1 vom Typ INT
7  -----
8  -- Der folgende Typ PERSON ist ein Zeigertyp. Jeder Wert des Typs PERSON
9  -- zeigt auf einen Verbund mit zwei Komponenten namens Name (vom Typ
10 -- STRING) und Alter (vom Typ INT):
11
12 'table' PERSON(Name: STRING, Alter: INT)
13 -----
14 -- Das folgende Praedikat NeuePerson erzeugt einen neuen Verbund mit zwei
15 -- Komponenten, initialisiert die Komponenten und liefert als Ausgabe-
16 -- parameter einen Zeiger, der auf den Verbund zeigt:
17
18 'action' NeuePerson(Heisst: STRING, Jahre: INT -> PERSON)
19   'rule' NeuePerson(NAM, ALT -> PERS):
20     PERS :: PERSON      -- Erzeuge einen neuen Verbund
21     PERS'Name  <- NAM   -- initialisiere die Name-Komponente
22     PERS'Alter <- ALT   -- initialisiere die Alter-Komponente
23 -----
24 'root '
25   Name1      <- "Anna"  -- Zuweisung eines Wertes an die Variable Name1
26   Alter1     <- 11      -- Zuweisung eines Wertes an die Variable Alter1
27   Name1      -> NAM1    -- Erzeugung einer Konstanten namens NAM1
28   Alter1     -> ALT1    -- Erzeugung einer Konstanten namens ALT1
29   print(NAM1 ->)
30   print(ALT1 ->)
31   Name1      <- "Berta" -- Zuweisung eines Wertes an die Variable Name1
32   Alter1     <- 22      -- Zuweisung eines Wertes an die Variable Alter1
33   Name1      -> NAM2    -- Erzeugung einer Konstanten namens NAM2
34   Alter1     -> ALT2    -- Erzeugung einer Konstanten namens ALT2
35   print(NAM2 ->)
36   print(ALT2 ->)
37   NeuePerson("Claudia", 33 -> P1) -- Erzeugung einer Konstanten namens P1
38   NeuePerson("Detlev", 44 -> P2) -- Erzeugung einer Konstanten namens P2
39   P1'Name    -> NAM3    -- Erzeugung einer Konstanten namens NAM3

```

```

40 P1'Alter -> ALT3 -- Erzeugung einer Konstanten namens ALT3
41 P2'Name -> NAM4 -- Erzeugung einer Konstanten namens NAM4
42 P2'Alter -> ALT4 -- Erzeugung einer Konstanten namens ALT4
43 print(NAM3 ->)
44 print(ALT3 ->)
45 print(NAM4 ->)
46 print(ALT4 ->)
47 P1'Name <- "Erich" -- Zuweisung eines Wertes an die Variable P1'Name
48 P1'Alter <- 55 -- Zuweisung eines Wertes an die Variable P1'Alter
49 P1'Name -> NAM5 -- Erzeugung einer Konstanten namens NAM5
50 P1'Alter -> ALT5 -- Erzeugung einer Konstanten namens ALT5
51 print(NAM5 ->)
52 print(ALT5 ->)
53 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 05 bis 06:** So vereinbart man veränderbare Variablen mit Namen.

**Zeile 12:** Eigentlich bezeichnet `PERSON` nicht nur *einen* Typ sondern *zwei* Typen:

1. Einen Verbundtyp (record type). Jeder Wert dieses Verbundtyps besteht aus zwei Komponenten namens `Name` und `Alter`. Diese Verbundkomponenten sind vom Typ `STRING` bzw. `INT`.
2. Einen Zeigertyp (pointer type). Jeder Wert dieses Zeigertyps zeigt auf einen Verbund (record) des Verbundtyps 1.

Je nachdem, in welchem Zusammenhang das Wort `PERSON` verwendet wird, ist der eine oder der andere Typ oder sind beide Typen gleichzeitig gemeint. Siehe unten.

**Zeile 18:** Das Aktionsprädikat `NeuePerson` hat zwei nicht weiter aufregende Eingabeparameter (vom Typ `STRING` bzw. `INT`) und einen etwas interessanteren Ausgabeparameter vom Zeiger-Typ `PERSON` (nicht vom Verbund-Typ `PERSON`).

**Zeile 19:** `NAM`, `ALT` und `PERS` sind, wie immer, unveränderbare Variablen. `NAM` und `ALT` spielen hier die Rolle von Mustern, `PERS` die Rolle eines Ausdrucks.

**Zeile 20:** Der doppelte Doppelpunkt (Quadpunkt) `::` bezeichnet in Gentle einen Allokator (ähnlich dem `new` in anderen Sprachen). Der Ausführer erzeugt eine neue Variable vom Verbund-Typ `PERSON` und liefert einen Zeigerwert vom Zeiger-Typ `PERSON`, der auf den Verbund zeigt, als Ergebnis. Mit diesem Zeigerwert erzeugt er eine unveränderbare Variable namens `PERS`.

**Zeile 21:** So kann man einer allokierten Variablen (genauer: einer Komponente einer solchen Variablen) einen (neuen) Wert zuweisen. `PERS'Name` bezeichnet die `Name`-Komponente des Verbundes, auf den `PERS` zeigt. Eine Zuweisung (an eine veränderbare Variable) wird in Gentle immer durch einen nach links gerichteten Pfeil `<-` bezeichnet.

**Zeile 25:** Auch hier ein nach links gerichteter Pfeil, also eine Zuweisung (an die vereinbarte Variable `Name1`)

**Zeile 27:** Ein nach rechts gerichteter Pfeil `->` bewirkt in Gentle immer, dass die rechts vom Pfeil genannten unveränderbaren Variablen erzeugt und mit den links vom Pfeil beschriebenen Werten versehen werden. Hier wird also eine unveränderbare Variable namens `NAM1` erzeugt und mit dem Wert der Variablen `Name1` versehen. Eine solche Erzeugung einer unveränderbaren Variablen ist konzeptuell etwas ziemlich anderes als eine Zuweisung (an eine veränderbare Variable).

**Zeile 28:** Hier wird eine unveränderbare Variable namens `ALT1` erzeugt und mit dem Wert der veränderbaren Variablen `Alter1` versehen. Dies ist in Gentle die einzige erlaubte Methode, um den Wert einer veränderbaren Variablen "zu lesen und weiter zu verarbeiten". Es ist z. B. nicht erlaubt, den Namen einer vereinbarten Variablen direkt als Parameter eines `print`-Befehls zu verwenden (`print(Name1)` oder `print(Alter1)`). Das mag im ersten Moment etwas umständlich erscheinen, dient

aber dazu, "die Welt der unveränderbaren Variablen" und "die Welt der veränderbaren Variablen" einfach und sauber (ohne komplizierte "Ausnahmeregeln") voneinander zu trennen.

**Zeile 29 bis 30:** Was würden Sie hier zum Bildschirm ausgeben, wenn Sie der Ausführer wären?

**Zeile 31 bis 34:** Diese Befehle sollen deutlich machen, dass man die Werte der vereinbarten Variablen `Name1` und `Alter1` durch Zuweisungen ("Linkspfeile") beliebig oft verändern kann.

**Zeile 35 bis 36:** Was würden Sie ausgeben?

**Zeile 37:** Das Prädikat `NeuePerson` wird aufgerufen. Der Aufruf gelingt garantiert und erzeugt eine unveränderbare Variable namens `P1`. Diese Variable enthält einen Zeigerwert (vom Zeiger-Typ `PERSON`), der auf eine Verbund-Variable (vom Verbund-Typ `PERSON`) zeigt. Können Sie erkennen, welche Werte die Komponenten dieser Verbund-Variablen haben?

**Zeile 38:** Ganz entsprechend wird eine unveränderbare Variable namens `P2` erzeugt.

**Zeile 39:** Hier wird eine Verbundkomponente "gelesen" und mit ihrem Wert wird eine unveränderbare Variable namens `NAM3` erzeugt.

**Zeile 40 bis 42:** Entsprechend werden unveränderbare Variablen namens `ALT3`, `NAM4` und `ALT4` erzeugt.

**Zeile 43 bis 46:** Was würden Sie ausgeben?

**Zeile 47 bis 48:** Diese Zuweisungen sollen zeigen, dass man auch die Komponenten-Werte einer allokierten Verbundvariablen beliebig häufig verändern kann.

**Zeile 49 bis 50:** Wenn man die veränderten Komponenten-Werte (der allokierten Variable auf die `P1` zeigt) lesen und weiter verarbeiten will, dann muss man zuerst unveränderbare Variablen (`NAM5` und `ALT5`) mit diesen Werten erzeugen lassen, da führt in Gentle kein Weg dran vorbei.

**Zeile 51 bis 52:** Was würden Sie ausgeben?

Hier die Ausgabe eines maschinellen Gentle-Ausführers, der obiges Programm ausgeführt hat. Die Zeilen-Nummern wurden nachträglich hinzugefügt. Hätten Sie die gleichen Werte ausgegeben?

```
1 "Anna "  
2  
3 "Berta "  
4  
5 "Claudia "  
6  
7 "Detlev "  
8  
9 "Erich "
```

## 28. Vordefinierte Prädikate

Die folgenden neun Namen bezeichnen in Gentle vordefinierte Prädikate:

```
eq ne gt ge lt le print where @
```

Da einige dieser Namen überladen sind (ein überladener Name bezeichnet mehrere verschiedene Prädikate), gibt es in Gentle mehr als 9 vordefinierte Prädikate. Die genaue Anzahl hängt davon ab, wieviele Typen ein Programmierer in seinem Gentle-Programm vereinbart hat.

### 28.1. Gleich und ungleich

Für jeden Typ  $T$  (d.h. für die vordefinierten Typen `INT`, `STRING` und `POS` und für jeden vom Programmierer vereinbarten Typ) gibt es zwei Bedingungs-Prädikate namens `eq` und `ne`:

```
1 'condition' eq(T, T ->)
2 'condition' ne(T, T ->)
```

mit zwei Eingabeparametern vom Typ  $T$  (und null Ausgabeparametern). Ein Aufruf von `eq` gelingt, wenn beide Parameter gleich sind. Ein Aufruf von `ne` gelingt, wenn die beiden Parameter nicht gleich sind. Im Beispiel 28.2.1. (siehe unten) werden diese Prädikate angewendet.

### 28.2. Größer, größergleich, kleiner, kleinergleich

Nur für die vordefinierten Typen `INT` und `STRING` gibt es weitere vier Bedingungs-Prädikate namens `gt` (greater than), `ge` (greater or equal), `lt` (less than) und `le` (less or equal):

```
1 'condition' gt(INT, INT ->)      -- groesser
2 'condition' ge(INT, INT ->)      -- groessergleich
3 'condition' lt(INT, INT ->)      -- kleiner
4 'condition' le(INT, INT ->)      -- kleinergleich
5
6 'condition' gt(STRING, STRING ->) -- groesser
7 'condition' ge(STRING, STRING ->) -- groessergleich
8 'condition' lt(STRING, STRING ->) -- kleiner
9 'condition' le(STRING, STRING ->) -- kleinergleich
```

Ein Aufruf des Prädikats `gt` gelingt, wenn der erste Parameter größer ist als der zweite Parameter. Für die anderen Prädikate gilt entsprechendes. Zeichenketten vom Typ `STRING` werden lexikographisch verglichen (d.h. eine Zeichenkette  $Z_1$  ist kleiner als eine Zeichenkette  $Z_2$ , wenn in einem Lexikon  $Z_1$  vor  $Z_2$  stehen müsste). Hier ein paar Beispiele zur lexikographischen Ordnung:

"AAA" ist kleiner als "BB" ist kleiner als "BBA" ist kleiner als "BBB" ist kleiner als "BBCAAA"

Die Sortierfolge einzelner Zeichen wird durch ihren ASCII-Code (eine Ganzzahl zwischen 0 und 255) bestimmt. Z. B. sind die Ziffern (0, 1, ..., 9) kleiner als die großen Buchstaben (A, B, ..., Z) und die großen Buchstaben sind kleiner als die kleinen Buchstaben (a, b, ..., z).

**Beispiel 28.2.1.:** Die Vergleichsprädikate:

```
1 'action' GGT(Zahl1:INT, Zahl2:INT -> GroessterGemeinsamerTeiler:INT)
2 'rule' GGT(N1, N2 -> N1): eq(N1, N2)
3 'rule' GGT(N1, N2 -> N3): le(N1, N2) GGT(N1, N2-N1 -> N3)
4 'rule' GGT(N1, N2 -> N3): gt(N1, N2) GGT(N1-N2, N2 -> N3)
```

**Achtung:** Die beiden Eingabeparameter `Zahl1` und `Zahl2` sollten Ganzzahlen größer als 0 sein, sonst funktioniert dieses Prädikat `GGT` nicht richtig.

### 28.3. Die `print`-Prädikate können alles mögliche ausgeben

Für jeden Typ `T` (d.h. für die vordefinierten Typen `INT`, `STRING` und `POS` und für jeden vom Programmierer vereinbarten Typ) gibt es ein Aktions-Prädikat namens `print` :

```
'action' print(T->)
```

mit dem man einen Wert vom Typ `T` zur Standardausgabe (das ist meistens der Bildschirm) ausgeben kann. Die Prädikate namens `print` sind in erster Linie für Testausgaben während der Entwicklung eines Gentle-Programms gedacht und nicht für "ernsthafte Ausgaben des fertigen Gentle-Programms".

Werte des Typs `STRING` werden von `print` in Anführungszeichen eingeschlossen ausgegeben. Z. B. läßt der Aufruf `print("Hallo!")` auf dem Bildschirm "Hallo!" erscheinen (und nicht nur Hallo!).

Wenn man Werte eines selbstdefinierten Typs (d.h. Terme) mit `print` ausgibt, werden diese in einer bestimmten Standardform ("mit systematischer Einrückung") ausgegeben. An dieser Standardform kann man nichts ändern.

Wenn die Möglichkeiten, die die `print`-Prädikate bieten, nicht ausreichen, kann man z. B. die Ausgabe-Prädikate `PutS`, `PutI`, `PutP` etc. im Modula `text_io.o` benutzen oder selbst Ausgabe-Prädikate in C programmieren.

### 28.4. Die `where`-Prädikate können unveränderbare Variablen erzeugen

Für jeden Typ `T` (d.h. für die vordefinierten Typen `INT`, `STRING` und `POS` und für jeden vom Programmierer vereinbarten Typ) gibt es ein Aktions-Prädikat namens `where`, dessen Vereinbarung eine einzige "triviale" Regel umfaßt:

```
1 'action' where(T -> T)
2   'rule' where(EIN -> AUS)
```

Mit einem Aufruf eines `where`-Prädikates kann man eine oder mehrer (unveränderbare) Variablen erzeugen.

**Beispiel 28.4.1.:** Das `where`-Prädikat für einen selbstvereinbarten Typ:

```
1 'type' LISTE
2   leer
3   list(Element: INT, Rest: LISTE)
```

Aufgrund dieser Typvereinbarung wird, so kann man sich vorstellen, vom Gentle-Compiler automatisch ein entsprechendes `where`-Prädikat vereinbart. Die folgenden Aufrufe dieses `where`-Prädikates könnten z. B. im Wurzel-Prädikat (`root predicate`) des betreffenden Gentle-Programms stehen:

```
1 where(list(7, leer) -> TL3)
2 where(list(13, TL3) -> list(N1, list(N2, R)))
```

Der Aufruf in Zeile 11 erzeugt eine Variable namens `TL3` mit dem Wert `list(7, leer)`. Die Variable `TL3` ist vom Typ `LISTE`. Der Aufruf in Zeile 12 erzeugt drei Variablen: Eine Variable namens `N1` vom Typ `INT` mit dem Wert 13, eine Variable namens `N2` vom Typ `INT` mit dem Wert 7 und eine Variable namens `R` vom Typ `LISTE` mit dem Wert `leer`.

## 28.5. Das An-der-Position-Prädikat @

Das vordefinierte Aktions-Prädikat mit dem einprägsamen Namen @ (englisch: "at", deutsch: "bei" oder "an der Position" oder "Affenschwanz") dient dazu, die Position eines bestimmten Lexems in einem Quellprogramm festzuhalten. Das @-Prädikat hat (null Eingabeparameter und) einen Ausgabeparameter vom vordefinierten Typ POS:

```
'action' @(-> POS)
```

Wenn man einen Wert vom Typ POS mit dem Prädikat PutP (aus dem Modul text\_io.o) ausgibt, erscheint z. B. folgendes auf dem Bildschirm:

```
file 1, line 3, col 17
```

Die Position eines Lexems braucht man häufig dazu, aussagekräftige Fehlermeldungen wie die folgenden zu erzeugen:

```
file 1, line 3, col 17: Der Bezeichner OTTO ist undefiniert
```

Hier ein kleines Beispielpogramm, in dem das @-Prädikat heftig angewendet wird:

```
1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- posit_v1.g: Demonstriert das vordefinierte Praedikat namens @ (welches
3  -- die Position eines Lexems aus dem Quellprogramm festhaelt). posit_v1 ist
4  -- ein Parser, der "Quellprogramme" der folgenden Form akzeptiert:
5  -- Breite: 123 Hoehe: 456
6  -- Die Positionen der vier Lexeme werden ermittelt und ausgegeben
7  -----
8  'nonterm' EINGABE
9    'rule' EINGABE:
10     @(->POS1) BREITE @(->POS2) HOEHE @(->POS3)
11     PutS("EINGABE: Eingabe beginnt bei ")
12     PutP(POS1) Nl
13     PutS("EINGABE: Breite beginnt bei ")
14     PutP(POS2) Nl
15     PutS("EINGABE: Hoehe beginnt bei ")
16     PutP(POS3) Nl
17 'nonterm' BREITE
18 'rule' BREITE:
19   "Breite:" @(->POS1) GanzLit @(->POS2)
20   PutS("BREITE: Breite beginnt bei ")
21   PutP(POS1) Nl
22   PutS("BREITE: Literal beginnt bei ")
23   PutP(POS2) Nl
24 'nonterm' HOEHE
25 'rule' HOEHE:
26   "Hoehe:" @(->POS1) GanzLit @(->POS2)
27   PutS("HOEHE: Hoehe beginnt bei ")
28   PutP(POS1) Nl
29   PutS("HOEHE: Literal beginnt bei ")
30   PutP(POS2) Nl
31 'token' GanzLit
32 -----
33 'action' PutS(STRING) -- Gibt eine Zeichenkette vom Typ STRING aus
34 'action' PutP(POS)    -- Gibt einen Wert des Typs POS (Position) aus
35 'action' Nl          -- Gibt ein newline-Zeichen aus
36 -----
37 'root'
38   EINGABE
```

```
39 --*****1*****2*****3*****4*****5*****6*****7*****
```

Diesem Parser `posit_v1` kann man z. B. das folgende "Quellprogramm" vorlegen:

```
40 Breite:
41     127
42 Hoehe: 350
```

Dabei gehören die Zeilennummern natürlich nicht zum Quellprogramm, sondern wurden nachträglich hinzugefügt. Der Buchstabe `B` am Anfang des Wortes `Breite:` ist also das erste Zeichen des Quellprogramms. Hier die Ausgabe des Parsers `posit_v1` für diese Eingabe:

```
43 BREITE: Breite beginnt bei file 1, line 1, col 1.
44 BREITE: Literal beginnt bei file 1, line 2, col 4.
45 HOEHE:  Hoehe beginnt bei file 1, line 3, col 1.
46 HOEHE:  Literal beginnt bei file 1, line 3, col 8.
47 EINGABE: Eingabe beginnt bei file 1, line 1, col 1.
48 EINGABE: Breite beginnt bei file 1, line 1, col 1.
49 EINGABE: Hoehe beginnt bei file 1, line 3, col 1.
```

Für die Eingabe

```
50 Breite:
51     123
52     Hoehe:
53     456
```

sieht die Ausgabe des Parsers `posit_v1` so aus:

```
54 BREITE: Breite beginnt bei file 1, line 1, col 1.
55 BREITE: Literal beginnt bei file 1, line 2, col 2.
56 HOEHE:  Hoehe beginnt bei file 1, line 3, col 3.
57 HOEHE:  Literal beginnt bei file 1, line 4, col 4.
58 EINGABE: Eingabe beginnt bei file 1, line 1, col 1.
59 EINGABE: Breite beginnt bei file 1, line 1, col 1.
60 EINGABE: Hoehe beginnt bei file 1, line 3, col 3.
```

## 29. Verwaltung einer Symboltabelle

Viele Compiler übersetzen Quellprogramme mit Hilfe einer sogenannten Symboltabelle. Mit "Symbolen" sind in diesem Zusammenhang vor allem die Bezeichner gemeint, die im Quellprogramm vereinbart und angewendet ("benutzt") werden. Typischerweise wird eine Symboltabelle folgendermaßen verwaltet: Wenn der Compiler eine Vereinbarung erkennt, trägt er den vereinbarten Bezeichner in seine Symboltabelle ein und verbindet ihn mit bestimmten Zusatzinformationen (z. B. mit dem Typ der vereinbarten Variablen oder mit dem Typ und dem Wert der vereinbarten Konstanten oder mit der formalen Parameterliste der vereinbarten Prozedur oder mit der Position im Quellprogramm, an der der Bezeichner vereinbart wurde etc.). Manchmal ergänzt der Compiler diese Zusatzinformationen später noch, z. B. indem er dem Bezeichner eine bestimmte Maschinenadresse zuordnet. Wenn der Compiler eine Anwendung eines Bezeichners erkennt, prüft er anhand der Symboltabelle, ob der Bezeichner korrekt vereinbart wurde und eventuell, ob der Typ des Bezeichners zu der Anwendung paßt etc.

In "einfachen Programmiersprachen" darf jeder Bezeichner höchstens einmal vereinbart werden. Entsprechend muss ein Compiler für eine solche einfache Sprache auch nur eine "einfache Symboltabelle" verwalten. Bei komplizierteren Sprachen besteht ein Programm aus Blöcken, die ineinander geschachtelt sein können (dies gilt z. B. für Pascal, Modula und Ada und ansatzweise auch für C). Ein Bezeichner kann dann mehrfach (in jedem Block mit einer neuen Bedeutung) vereinbart werden. Entsprechend komplizierter ist in solchen Fällen die Verwaltung der Symboltabelle.

Die folgenden beiden Compiler `syntab_1.g` und `syntab_2.g` leisten exakt das Gleiche, d.h. sie übersetzen die gleichen Quellprogramme in die gleichen Zielprogramme. Beide Compiler verwalten eine einfache Symboltabelle und beide benützen dazu im wesentlichen zwei Prädikate namens `DefMeaning` (um einen Bezeichner und bestimmte Informationen dazu in die Symboltabelle einzutragen) und `HasMeaning` (um Informationen zu einem Bezeichner aus der Symboltabelle zu holen). Im Compiler `syntab_1.g` sind die beiden Prädikate `DefMeaning` und `HasMeaning` in Gentle programmiert. Das hat den Vorteil, dass der Leser die Vereinbarungen der Prädikate (hoffentlich) relativ leicht lesen und verstehen kann. Allerdings ist `syntab_1.g` kein Vorbild, was die Effizienz (genauer: die Schnelligkeit der Zugriffe auf die Symboltabelle) betrifft. Im Compiler `syntab_2.g` wird eine Implementierung der Prädikate `DefMeaning` und `HasMeaning` in der Sprache C verwendet (siehe Modul `idents.c`). Diese effiziente Verwaltung einer Symboltabelle ist für die Verwendung in "ernsthaften Compilern" gedacht und geeignet und wird als Teil des Gentle-Systems mitgeliefert.

Im nächsten Abschnitt wird mit dem Compiler `syntab_3.g` gezeigt, wie man eine blockorientierte Symboltabelle verwalten kann. Auch in diesem Beispiel wird der C-Modul `idents.c` als Grundlage verwendet.

```

1  -----1*****2*****3*****4*****5*****6*****7*****
2  -- syntab_1.g: Verwaltung einer einfachen Symboltabelle
3  -- Version 1: Alle Typen und Praedikate sind in Gentle programmiert
4  -----
5      -- Typen und Praedikate fuer die Symboltabelle
6      -- Ein Wert vom Typ INFO enthaelt alle Informationen, die zu einem
7      -- Bezeichner in der Symboltabelle gespeichert werden sollen:
8  'type' INFO
9      info(Position: POS)
10     -- EineSymboltabelle ist eine lineare Liste von Eintraegen. Jeder
11     -- Eintrag enthaelt einen Bezeichner und eine Info-rmation.
12  'type' SYMTAB
13     leer
14     list(Bezeichner: STRING, Info: INFO, Rest: SYMTAB)
15     -- Auf die globale Symboltabelle greifen (nur) die Praedikate
16     -- HasMeaning und DefMeaning zu:
17  'var' GlobSymTab: SYMTAB -- Die globale Symboltabelle
18
19  'condition' HasMeaning(Bezeichner: STRING -> Info: INFO)
20     -- Missglueckt, wenn der Bezeichner noch nicht in der globalen
21     -- Symboltabelle steht. Liefert sonst die Info des Bezeichners:
22  'rule' HasMeaning(BEZ -> INF):
23     GlobSymTab -> LST
24     HasMeaningR(BEZ, LST -> INF).
25
26  'condition' HasMeaningR(Bezeichner: STRING, STab: SYMTAB -> Info: INFO)
27     -- Rekursives Hilfspraedikat fuer HasMeaning
28  'rule' HasMeaningR(BEZ1, list(BEZ2, INF, _ ) -> INF):
29     eq(BEZ1, BEZ2).
30  'rule' HasMeaningR(BEZ1, list( _ , _ , ST) -> INF):
31     HasMeaningR(BEZ1, ST -> INF).
32

```

```

33 'action' DefMeaning(Bezeichner: STRING, Info: INFO)
34   -- Traegt einen Bezeichner und seine Info in die globale Symboltabelle
35   -- ein. Mehrfachs eintragen des gleichen Bezeichners ist moeglich,
36   -- aber HasMeaning findet dann nur den letzten Eintrag:
37   'rule' DefMeaning(BEZ, INF):
38     GlobSymTab -> ALTE_ST
39     GlobSymTab <- list(BEZ, INF, ALTE_ST)
40 -----
41 'action' TragEinOderMelde(Bezeichner: STRING, Info: INFO)
42   -- Wenn der Bezeichner schon in der Symboltabelle steht, wird eine
43   -- Fehlermeldung ausgegeben. Sonst wird der Bezeichner zusammen mit
44   -- seiner Info in die Symboltabelle eingetragen:
45   'rule' TragEinOderMelde(BEZ, info(P0)):      -- Wenn
46     HasMeaning(BEZ -> info(P1))              -- BEZ schon eingetragen ist,
47     PutP(P0) PutS(" : Bez. ") PutS(BEZ)      -- dann
48     PutS(" schon definiert in ") PutP(P1) Nl  -- Fehlermeldung ausgeben,
49   'rule' TragEinOderMelde(BEZ, INF):          -- sonst
50     DefMeaning(BEZ, INF).                    -- BEZ eintragen.
51
52 'action' MeldeWennUndef(Bezeichner: STRING, Position: POS)
53   -- Wenn der Bezeichner nicht in der Symboltabelle steht, wird eine
54   -- Fehlermeldung ausgegeben:
55   'rule' MeldeWennUndef(BEZ, P):              -- Wenn BEZ eingetragen ist,
56     HasMeaning(BEZ -> _).                    -- nix machen,
57   'rule' MeldeWennUndef(BEZ, P):              -- sonst
58     PutP(P) PutS(" : Bez. ") PutS(BEZ)      -- Fehlermeldung
59     PutS(" ist undefiniert") Nl.             -- ausgeben.
60 -----
61 -- Die kontextfreie Grammatik der Quellsprache (ein Quell-Programm besteht
62 -- aus zwei Listen von Bezeichnern, die durch einen Doppelpunkt getrennt
63 -- sind. Ein Bezeichner in der ersten Liste gilt als Vereinbarung, ein
64 -- Bezeichner in der zweiten Liste als Anwendung des Bezeichners):
65
66 'nonterm' init -- "Pseudo-Zwischensymbol" zum Initialisieren der globalen
67   'rule' init: GlobSymTab <- leer -- Symboltabelle
68
69 'nonterm' Programm
70   'rule' Programm: VereinbarungTeil ":" AnweisungsTeil.
71 'nonterm' VereinbarungTeil
72   'rule' VereinbarungTeil: Vereinbarung VereinbarungTeil.
73   'rule' VereinbarungTeil: .
74 'nonterm' AnweisungsTeil
75   'rule' AnweisungsTeil :Anweisung AnweisungsTeil.
76   'rule' AnweisungsTeil :.
77 'nonterm' Vereinbarung
78   'rule' Vereinbarung:
79     Bezeich(-> BEZ) @(-> P)
80     TragEinOderMelde(BEZ, info(P)).
81 'nonterm' Anweisung
82   'rule' Anweisung:
83     Bezeich(-> BEZ) @(-> P)
84     MeldeWennUndef(BEZ, P).
85 'token' Bezeich(->STRING)
86 -----
87 'action' PutS(STRING) -- in text_io.o definiert
88 'action' PutP(POS) -- in text_io.o definiert
89 'action' Nl -- in text_io.o definiert
90 -----
91 'root'

```

```

92  init      -- Die globale Symboltabelle initialisieren
93  Programm  -- Ein Quellprogramm parsen und Kontextbedingungen pruefen
94  --*****1*****2*****3*****4*****5*****6*****7*****

```

Eine Eingabe fuer das Programm (oder: ein Quellprogramm für den Compiler) `syntab_1`:

```

1  anna bert celia
2  detlef emil bert anna :
3  celia anna celia karlos
4  detlef detlef anna deltef

```

Und hier die Ausgabe des Programms `syntab_1`:

```

1  file 1, line 2, col 13.: Bez. bert schon definiert in file 1, line 1, col 6.
2  file 1, line 2, col 18.: Bez. anna schon definiert in file 1, line 1, col 1.
3  file 1, line 3, col 18.: Bez. karlos ist undefiniert
4  file 1, line 4, col 20.: Bez. deltef ist undefiniert

```

Der folgende Compiler `syntab_2.g` leistet exakt das Gleiche wie der Compiler `syntab_1.g`, verwaltet seine Symboltabelle aber wesentlich effizienter:

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- syntab_2.g: Verwaltung einer einfachen Symboltabelle
3  -- Version 2: Die Typen und Praedikate fuer die Symboltabelle sind in
4  -- der Sprache C (Modul idents.c und Datei id_i_ada.t) programmiert.
5  -----
6  -- Typen und Praedikate fuer die Symboltabelle
7  -- Ein Wert vom Typ INFO enthaelt alle Informationen, die zu einem
8  -- Bezeichner in der Symboltabelle gespeichert werden sollen:
9  'type' INFO          -- Informationen zu einem Bezeichner
10  info(Position: POS) -- koennte auch viele Komponenten haben
11
12  -- Der Typ IDENT und die Praedikate HasMeaning und DefMeaning
13  -- werden im C-Modul idents.o definiert:
14  'type' IDENT        -- Zeiger in eine Hash-Tabelle
15  'condition' HasMeaning(Bezeichner: IDENT      -> Info: INFO)
16  'action'    DefMeaning(Bezeichner: IDENT, Info: INFO ->          )
17  -----
18  'action' TragEinOderMelde(Bezeichner: IDENT, Info: INFO)
19  -- Wenn der Bezeichner schon in der Symboltabelle steht, wird eine
20  -- Fehlermeldung ausgegeben. Sonst wird der Bezeichner zusammen mit
21  -- seiner Info in die Symboltabelle eingetragen:
22  'rule' TragEinOderMelde(BEZ, info(P0)):      -- Wenn
23  HasMeaning(BEZ -> info(P1))                  -- BEZ schon eingetragen ist,
24  id_to_string(BEZ -> S_BEZ)                   -- dann
25  PutP(P0) PutS(" : Bez. ") PutS(S_BEZ)        -- Fehlermeldung
26  PutS(" schon definiert in ") PutP(P1) Nl     -- ausgeben
27  'rule' TragEinOderMelde(BEZ, INF):           -- sonst
28  DefMeaning(BEZ, INF).                       -- BEZ mit INF eintragen.
29
30  'action' MeldeWennUndef(Bezeichner: IDENT, Position: POS)
31  -- Wenn der Bezeichner nicht in der Symboltabelle steht, wird eine
32  -- Fehlermeldung ausgegeben:
33  'rule' MeldeWennUndef(BEZ, P):               -- Wenn
34  HasMeaning(BEZ -> _).                       -- BEZ eingetragen ist: nix
35  'rule' MeldeWennUndef(BEZ, P):             -- sonst
36  id_to_string(BEZ -> S_BEZ)                  -- Fehlermeldung
37  PutP(P) PutS(" : Bez. ") PutS(S_BEZ)        -- aus-
38  PutS(" ist undefiniert") Nl.                -- geben.
39  -----

```

```

40 -- Die kontextfreie Grammatik der Quellsprache (ein Quell-Programm besteht
41 -- aus zwei Listen von Bezeichnern, die durch einen Doppelpunkt getrennt
42 -- sind. Ein Bezeichner in der ersten Liste gilt als Vereinbarungen, ein
43 -- Bezeichner in der zweiten Liste als Anwendung des Bezeichners):
44
45 'nonterm' Programm
46   'rule' Programm: VereinbarungsTeil ":" AnweisungsTeil.
47 'nonterm' VereinbarungsTeil
48   'rule' VereinbarungsTeil: Vereinbarung VereinbarungsTeil.
49   'rule' VereinbarungsTeil: . -- leerer VereinbarungsTeil
50 'nonterm' AnweisungsTeil
51   'rule' AnweisungsTeil : Anweisung AnweisungsTeil.
52   'rule' AnweisungsTeil : . -- leerer AnweisungsTeil
53 'nonterm' Vereinbarung
54   'rule' Vereinbarung : -- Eine Vereinbarung ist ein
55       Bezeich(-> BEZ) @(-> P) -- Bezeichner BEZ an einer Position P
56       TragEinOderMelde(BEZ, info(P)).
57 'nonterm' Anweisung
58   'rule' Anweisung : -- Eine Anweisung ist ein
59       Bezeich(-> BEZ) @(-> P) -- Bezeichner BEZ an einer Position P
60       MeldeWennUndef(BEZ, P).
61 'token' Bezeich(->IDENT)
62 -----
63 'action' id_to_string(IDENT -> STRING) -- in idents.o definiert
64 'action' PutS (STRING) -- in text_io.o definiert
65 'action' PutP (POS) -- in text_io.o definiert
66 'action' Nl -- in text_io.o definiert
67 -----
68 'root'
69   Programm -- Ein Quellprogramm parsen und Kontextbedingungen pruefen
70 --*****1*****2*****3*****4*****5*****6*****7*****

```

Die folgende DOS-Stapeldatei namens `build.bat` dient zum Erzeugen des Compilers `symtab_2`. Sie zeigt im Wesentlichen, dass der Modul `idents.o` (die Übersetzung des C-Moduls `idents.c`) eingebunden werden muss, da er die Vereinbarungen der Prädikate `DefMeaning` und `HasMeaning` enthält (siehe Zeile 9 und 15):

```

1 rem -----
2 rem build.bat fuer symtab_2
3 rem -----
4 gentle symtab_2.g
5 reflex Bezeich=id_i_ada.t
6 flex gen.l
7 bison gen.y
8
9 @copy %genlib%\idents.o > nul
10 @copy %genlib%\errmsg.o > nul
11 @copy %genlib%\main1.o > nul
12 @copy %genlib%\grts.o > nul
13 @copy %genlib%\text_io.o > nul
14
15 gcc symtab_2.c lex.yy.c gen.tab.c idents.o errmsg.o main1.o grts.o text_io.o
16 ren a.exe symtab_2.exe
17
18 symtab_2 "anna bert anna: bert carl anna bert"
19 rem -----

```

Eine Eingabe fuer das Programm (oder: ein Quellprogramm für den Compiler) `symtab_2`:

```

1 anna bert celia
2 detlef emil bert anna :
3 celia anna celia karlos
4 detlef detlef anna deltef

```

Und hier die Ausgabe des Programms `syntab_2`:

```

1 file 1, line 2, col 13.: Bez. bert schon definiert in file 1, line 1, col 6.
2 file 1, line 2, col 18.: Bez. anna schon definiert in file 1, line 1, col 1.
3 file 1, line 3, col 18.: Bez. karlos ist undefiniert
4 file 1, line 4, col 20.: Bez. deltef ist undefiniert

```

### 30. Verwaltung einer "blockorientierten Symboltabelle"

Das folgende Beispiel `syntab_3.g` soll zeigen, wie ein Compiler für eine blockorientierte Sprache eine entsprechend "blockorientierte Symboltabelle" effizient verwalten kann. Der dazu verwendete Algorithmus stammt von F. W. Schröder und benützt als Grundlage den in C geschriebenen Gentle-Standardmodul `idents.c`.

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- syntab_3.g: Verwaltung einer zusammengesetzten Symboltabelle
3  -- (fuer eine Block-orientierte Sprache)
4  -----
5      -- Typen und Praedikate fuer die Symboltabelle
6      -- Ein Wert vom Typ INFO enthaelt alle Informationen, die zu einem
7      -- Bezeichner in der Symboltabelle gespeichert werden sollen:
8  'type' INFO                -- Informationen zu einem Bezeichner
9      info(Position: POS, Wert: INT)
10     leer
11 'type' EINTRAG              -- Eintrag in einer Liste verdeckter Bezeichner
12     eint(Bezeichner: IDENT, VerdeckteBedeutung: INFO)
13 'type' LISTE_VB            -- Liste verdeckter Bezeichner
14     leer
15     list(Elem: EINTRAG, Rest: LISTE_VB)
16 'type' STAPEL_LVB          -- Stapel von Listen verdeckter Bezeichner
17     leer
18     push(Elem: LISTE_VB, Rest: STAPEL_LVB)
19
20     -- Der Typ IDENT und die folgenden Praedikate werden
21     -- im C-Modul idents.c definiert:
22 'type' IDENT                -- Zeiger in eine Hash-Tabelle
23 'condition' HasMeaning (Bezeichner: IDENT -> Bedeutung: INFO)
24 'action'    id_to_string(Bezeichner: IDENT ->                STRING)
25 'action'    DefMeaning (Bezeichner: IDENT, Bedeutung: INFO)
26 'action'    UndefMeaning(Bezeichner: IDENT)
27
28     -- Der globale Stapel ethaelt alle verdeckten Bedeutungen:
29 'var' GlobalerStapel : STAPEL_LVB
30
31 'nonterm' init                -- muss nonterm sein wegen YACC
32 'rule' init: GlobalerStapel <- leer.
33 -----

```

```

34 -- Praedikate, die dirket mit der Symboltabelle zu tun haben:
35
36 'nonterm' BetreteSichtbarkeitsBereich -- muss nonterm sein wegen YACC
37 'rule' BetreteSichtbarkeitsBereich:
38     GlobalerStapel -> LS
39     GlobalerStapel <- push(leer, LS)
40
41 'nonterm' VerlasseSichtbarkeitsBereich -- muss nonterm sein wegen YACC
42 'rule' VerlasseSichtbarkeitsBereich:
43     GlobalerStapel -> push(LVB, REST_STAPEL)
44     GlobalerStapel <- REST_STAPEL
45     EntDecke(LVB)
46
47 'action' EntDecke(VerdeckteBezeichner: LISTE_VB)
48 'rule' EntDecke(list(eint(BEZ, leer), REST_LISTE)):
49     UndefMeaning(BEZ)
50     EntDecke(REST_LISTE).
51 'rule' EntDecke(list(eint(BEZ, INF_ALT), REST_LISTE)):
52     DefMeaning(BEZ, INF_ALT)
53     EntDecke(REST_LISTE).
54 'rule' EntDecke(leer): .
55
56 'action' VerdeckeOderMelde(Bezeichner: IDENT, Bedeutung: INFO)
57 'rule' VerdeckeOderMelde(BEZ, INF_NEU): -- Verdecke!
58     GlobalerStapel -> push(LVB, REST_STAPEL)
59     VerdecktNochNichts(BEZ, LVB)
60     BisherigeBedeutung(BEZ -> INF_ALT)
61     GlobalerStapel <- push(list(eint(BEZ, INF_ALT), LVB), REST_STAPEL)
62     DefMeaning(BEZ, INF_NEU)
63 'rule' VerdeckeOderMelde(BEZ, info(P, _)): -- Melde!
64     id_to_string(BEZ -> S_BEZ)
65     PutP(P) PutS(": Der Bezeichner ") PutS(S_BEZ)
66     PutS(" wird mehrfach vereinbart") Nl
67
68 'condition' VerdecktNochNichts(Bezeichner: IDENT, SchonVerdekt: LISTE_VB)
69 'rule' VerdecktNochNichts(BEZ1, list(eint(BEZ2, _), REST_LISTE)):
70     ne(BEZ1, BEZ2)
71     VerdecktNochNichts(BEZ1, REST_LISTE).
72 'rule' VerdecktNochNichts(_, leer): .
73
74 'action' BisherigeBedeutung(Bezeichner: IDENT -> Bedeutung: INFO)
75 -- liefert fuer jeden Bezeichner eine bisherige Bedeutung, notfalls
76 -- die Bedeutung leer:
77 BisherigeBedeutung(BEZ -> INF):
78     HasMeaning(BEZ -> INF).
79 BisherigeBedeutung(BEZ -> leer): .
80
81 'action' MeldeDassUndef(Bezeichner: IDENT, Position: POS)
82 'rule' MeldeDassUndef(BEZ, P):
83     id_to_string(BEZ -> S_BEZ)
84     PutP(P) PutS(": Der Bezeichner ")
85     PutS(S_BEZ)
86     PutS(" ist undefiniert") Nl
87 -----

```

```

88 -- Abstrakte Syntax (interne Darstellung eines Programms)
89 'type' ANWEISUNG
90   wrt(Bezeichner: IDENT, Wert: INT)
91 'type' ANWEISUNGS_LISTE
92   leer
93   list(Elem: ANWEISUNG, Rest: ANWEISUNGS_LISTE)
94 -----
95 -- Konkrete Syntax und Umwandlung in abstrakte Syntax:
96
97 'nonterm' Block(->ANWEISUNGS_LISTE)
98   'rule' Block(-> AWL):
99     "declare"
100    BetreteSichtbarkeitsBereich    -- Pseudo-nonterm
101    Vereinbarungsteil
102    "begin"
103    Anweisungsteil(-> AWL)
104    "end" ";"
105    VerlasseSichtbarkeitsBereich  -- Pseudo-nonterm
106 'nonterm' Vereinbarungsteil
107   'rule' Vereinbarungsteil:
108     Vereinbarung Vereinbarungsteil
109   'rule' Vereinbarungsteil: .    -- leerer Vereinbarungsteil
110 'nonterm' Anweisungsteil(-> ANWEISUNGS_LISTE)
111   'rule' Anweisungsteil(-> GESAMT):
112     Anweisung(-> AWL1) Anweisungsteil (-> AWL2)
113     conc(AWL1, AWL2 -> GESAMT).
114   'rule' Anweisungsteil(-> leer): . -- leerer Anweisungsteil
115 'nonterm' Vereinbarung
116   'rule' Vereinbarung:
117     Bezeich(-> BEZ) @(-> P) "==" GanzLit(-> INT1) ";"
118     VerdeckeOderMelde(BEZ, info(P, INT1)).
119 'nonterm' Anweisung(-> ANWEISUNGS_LISTE)
120   'rule' Anweisung(-> list(wrt(BEZ, INT1), leer)):
121     "write" Bezeich(->BEZ) @(->P) ";"
122     (| HasMeaning(BEZ -> info(_, INT1))
123      || MeldeDassUndef(BEZ, P)
124      where(-1 -> INT1)
125      |)
126   'rule' Anweisung(-> AWL):
127     Block(-> AWL).
128
129 'token'   Bezeich(->IDENT)
130 'token'   GanzLit(->INT)
131 -----
132 -- Hilfspraedikat zum Konkatenieren von zwei Anweisungslisten:
133 'action' conc(ANWEISUNGS_LISTE, ANWEISUNGS_LISTE -> ANWEISUNGS_LISTE)
134   'rule' conc(list(E1, REST1), LIST2 -> list(E1, LIST3)):
135     conc(REST1, LIST2 -> LIST3).
136   'rule' conc(leer, LIST2 -> LIST2): .
137 -----
138 'action' GibZielprogrammAus(ANWEISUNGS_LISTE)
139   'rule' GibZielprogrammAus(list(wrt(BEZ, INT1), REST_LISTE)):
140     id_to_string(BEZ -> S_BEZ)
141     PutS("Put ") PutI(INT1) PutS(" -- ") PutS(S_BEZ) Nl
142     GibZielprogrammAus(REST_LISTE).
143   'rule' GibZielprogrammAus(leer): Nl.
144 -----
145 'action' PutS(STRING)                -- in text_io.o definiert

```

```

146 'action' PutI(INT)           -- in text_io.o definiert
147 'action' PutP(POS)         -- in text_io.o definiert
148 'action' Nl                -- in text_io.o definiert
149 -----
150 'root'
151   init                      -- initialisiere die globale Symboltabelle
152   Block(->AWL)              -- lies ein Quellprogramm ein
153   GibZielprogrammAus(AWL)
154 --*****1*****2*****3*****4*****5*****6*****7*****
155

```

Hier ein Quellprogramm für den Compiler `syntab_3`:

```

1 declare
2   anna := 10;
3   bert := 20;
4 begin
5   write anna;           -- put 10
6   write bert;          -- put 20
7   declare
8     anna := 30;
9     carl := 40;
10  begin
11    write bert;         -- put 20
12    write anna;        -- put 30
13    write carl;        -- put 40
14    declare
15      anna := 50;
16      carl := 60;
17    begin
18      write bert;      -- put 20
19      write anna;      -- put 50
20      write carl;      -- put 60
21    end;
22  end;
23  write anna;          -- put 10
24  write bert;          -- put 20
25 end;

```

Und hier das Zielprogramm, in das der Compiler `syntab_3` obiges Quellprogramm übersetzt:

```

1 Put 10 -- anna
2 Put 20 -- bert
3 Put 20 -- bert
4 Put 30 -- anna
5 Put 40 -- carl
6 Put 20 -- bert
7 Put 50 -- anna
8 Put 60 -- carl
9 Put 10 -- anna
10 Put 20 -- bert

```

Hier ein fehlerhaftes Quellprogramm als Eingabe für den Compiler `syntab_3`:

```
1 declare
2   anna := 10;
3   bert := 20;
4   anna := 30;           -- Fehlermeldung
5 begin
6   write bert;          -- put 20
7   write carl;         -- Fehlermeldung
8   declare
9     anna := 40;
10    bert := 50;
11    bert := 60;       -- Fehlermeldung
12  begin
13    write carl;      -- Fehlermeldung
14  end;
15 end;
```

Und hier die Ausgabe von `syntab_3` für das fehlerhafte Quellprogramm:

```
1 file 1, line 4, col 4.: Der Bezeichner anna wird mehrfach vereinbart
2 file 1, line 7, col 10.: Der Bezeichner carl ist undefiniert
3 file 1, line 11, col 7.: Der Bezeichner bert wird mehrfach vereinbart
4 file 1, line 13, col 13.: Der Bezeichner carl ist undefiniert
5 Put 20 -- bert
6 Put -1 -- carl
7 Put -1 -- carl
```

### 31. Wahl-Prädikate (choice predicates)

Wahl-Prädikate sind dazu gedacht, bestimmte, relativ komplizierte *Optimierungsprobleme* bei der *Codegenerierung* leichter lösbar zu machen. Der Programmierer kann jede Regel eines Wahl-Prädikates mit einem *Preis* versehen. Bei der Ausführung eines Aufrufs eines Wahl-Prädikates wird nicht die erste Lösung genommen, sondern es werden (in einem bestimmten Sinne) alle Lösungen erzeugt und für jede Lösung ihr Preis berechnet. Dann wird die billigste Lösung (oder die erste von mehreren gleich billigen Lösungen) gewählt. Gewisse Einschränkungen garantieren, dass die Zeit für die Suche nach einer billigsten Lösung nicht quadratisch oder sogar exponentiell mit der Anzahl der Regeln wächst, sondern nur linear. Die wichtigste Einschränkung kann man ganz grob etwa so wiedergeben: Eine billigste Lösung muss aus billigsten Teillösungen bestehen (oder: "lokal schlechte Teillösungen" dürfen zusammengenommen keine "globale gute Lösung" ergeben).

**Anmerkung:** Auf der Suche nach Beispielen für die Anwendung von Wahl-Prädikaten hat der Autor mehrmals "zu einfache Probleme" konstruiert, die sich mit einfacheren Mitteln (normalen Aktions- und Bedingungs-Prädikaten) eleganter lösen lassen. Wahl-Prädikate sind nur in bestimmten, relativ komplizierten Fällen sinnvoll. "Einfache Code-Generierungs-Probleme" sollte man ohne Wahl-Prädikate lösen.

Im folgenden Beispiel sollen bestimmte Ausdrücke in Befehle für eine hypothetische Stapel-und-Akkumulator-Maschine übersetzt werden. Die Ausdrücke dürfen nur aus Variablen, Minuszeichen und Klammern bestehen, z. B. so:

$A-B$ ,  $(A-B)-(C-D)$ ,  $A-B-C-D-(E-F-G-H)$ ,  $ANNA-(BERTA-ANNA)$  etc.

**Anmerkung:** Das Minuszeichen wird in diesem Beispiel dem Pluszeichen vorgezogen, weil bei reinen Plus-Ausdrücken Klammern weniger Sinn machen als bei reinen Minus-Ausdrücken.  $A+B+C$  hat häufig die gleich Bedeutung wie  $(A+B)+C$  und wie  $A+(B+C)$ , aber  $(A-B)-C$  bedeutet etwas ganz anderes als  $A-(B-C)$ .

Die Stapel-und-Akkumulator-Maschine (SUA-Maschine) kann wahlweise die obersten beiden Werte auf ihrem Stapel durch ihre Differenz ersetzen (Stapel-Subtraktion, `sub_stk`) oder den Wert einer Variablen von ihrem Akkumulator subtrahieren (Akku-Subtraktion, `sub_aku`). Man kann einen Minus-Ausdruck im allgemeinen also in verschiedene Befehlsfolgen für die SUA-Maschine übersetzen. Oft (aber nicht immer) ist das Rechnen im Akkumulator "billiger" als das Rechnen auf dem Stapel, aber für einige Ausdrücke ist ein Stapel notwendig (z. B. für Ausdrücke wie  $(A-B)-(C-D)$ ).

Wichtig ist die folgende (hoffentlich) anschauliche Sprechweise: Ein zu übersetzender Ausdruck wird *akkumulierbar* (bzw. *stapelbar*) genannt, wenn er in eine Befehlsfolge übersetzt werden kann, die ein Ergebnis im Akkumulator (bzw. auf dem Stapel) erzeugt. Viele Ausdrücke sind sowohl akkumulierbar als auch stapelbar, aber die verschiedenen Übersetzungen des gleichen Ausdrucks sind unterschiedlich "teuer". Gesucht ist eine möglichst billige Übersetzung.

Hier eine Beschreibung aller Befehle der Zielmaschine. `BEZ` bezeichnet dabei stets einen beliebigen Bezeichner (identifier):

|                         |                  |   |
|-------------------------|------------------|---|
| <code>push</code>       | <code>BEZ</code> | Legt den Wert der Variablen <code>BEZ</code> als oberstes Element auf den Stapel.   |
| <code>pop</code>        | <code>BEZ</code> | Entfernt den obersten Wert vom Stapel und legt ihn in die Variable <code>BEZ</code> .   |
| <code>sub_stk</code>    |                  | Entfernt den obersten und den zweitobersten Wert vom Stapel, subtrahiert den obersten Wert vom zweitobersten Wert und legt das Ergebnis (die Differenz) wieder auf den Stapel |
| <code>pop_load</code>   |                  | Entfernt den obersten Wert vom Stapel und legt ihn in den Akkumulator.  |
| <code>load</code>       | <code>BEZ</code> | Lädt den Wert der Variablen <code>BEZ</code> in den Akkumulator.  |
| <code>store</code>      | <code>BEZ</code> | Kopiert den Inhalt des Akkumulators in die Variable <code>BEZ</code> . Der Inhalt des Akkumulators wird dadurch nicht verändert.  |
| <code>sub_aku</code>    | <code>BEZ</code> | Subtrahiert den Wert der Variablen <code>BEZ</code> vom Akkumulator.  |
| <code>neg_aku</code>    |                  | Dreht das Vorzeichen des Akkumulators um (oder: multipliziert den Akkumulator mit -1).  |
| <code>store_push</code> |                  | Kopiert den momentanen Inhalt des Akkumulators oben auf den Stapel. Der Inhalt des Akkumulators wird dadurch nicht verändert.   |

Die Maschinenbefehle, in die ein Ausdruck übersetzt wird, müssen (so sei vereinbart) den Wert des Ausdrucks in eine ausgezeichnete Variable namens `ERGEB` bringen. Wenn der Wert auf dem Stapel berechnet wird, muss er abschließend mit einem `pop`-Befehl nach `ERGEB` gebracht werden. Wird der Wert des Ausdrucks dagegen im Akkumulator berechnet, muss er mit einem `store`-Befehl nach `ERGEB` gebracht werden.

Um die Vereinbarung und Initialisierung von Variablen kümmern wir uns in diesem Beispiel überhaupt nicht. Wir nehmen einfach an, dass alle benutzten Variablen existieren und schon irgendwie einen vernünftigen Wert bekommen haben.

Der Compiler `choice_1.g` (siehe unten) übersetzt jeweils einen Ausdruck seiner Quellsprache (einen "konkreten Ausdruck") zuerst in eine Zwischendarstellung (in einen "abstrakten Ausdruck") und dann in eine Befehlsfolge für die SUA-Maschine.

**Beispiel:** Dem konkreten Ausdruck  $(ANNA - BERT) - CARL$  entspricht der abstrakte Ausdruck `sub(sub(bez(ANNA), bez(BERT)), bez(CARL))` (siehe unten Zeile 23 bis 25). Dieser abstrakte Ausdruck kann z. B. in die folgende Befehlsfolge für die SUA-Maschine übersetzt werden:

```
push    ANNA
push    BERT
sub_stk
push    CARL
sub_stk
pop     ERGEB
```

Eine zweite mögliche Übersetzung ist die Befehlsfolge:

```
load    ANNA
sub_aku BERT
sub_aku CARL
store   ERGEB
```

Der folgende Compiler wählt die zweite Übersetzung, weil sie billiger ist.

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- choice_1.g: Ein (hoffentlich) einfaches Beispiel fuer die Anwendung
3  -- von Wahl-Praedikaten (choice predicates).
4  -----
5  -- Quellsprache: Ausdruecke, die aus Variablen, Minuszeichen und Klammern
6  -- bestehen (z.B. A-B oder (A-B)-(C-D-E) oder ANNA-(BERT-CARL)-BERT ).
7  -- Zielsprache: Befehle fuer eine "Stapel-und-Akkumulator-Maschine" (die
8  -- Zahlen in Klammern geben die "Kosten einer Befehlsausfuehrung" an)
9  -- push      V  -- legt den Wert der Variablen V auf den Stapel      (10)
10 -- pop       V  -- bringt oberstes Stapelelement in die Variable V   (10)
11 -- sub_stk   -- ersetzt die beiden obersten Stapelelemente durch
12 --           -- ihre Differenz ("zweitoberstes minus oberstes")     (20)
13 -- pop_load  -- laedt das oberste Stapelelement in den Akku          (20)
14 -- load     V  -- laedt den Wert der Variablen V in den Akku          (10)
15 -- store    V  -- speichert den Inhalt des Akku in die Variable V     (10)
16 -- sub_aku  V  -- subtrahiert den Wert der Variablen V vom Akku       (10)
17 -- neg_aku  -- dreht das Vorzeichen des Akku um ("mal -1")          (10)
18 -- store_push -- legt den Inhalt des Akku auf den Stapel             (20)
19 -- Gesucht ist eine Uebersetzung in eine moeglichst "billige" Befehlsfolge.
20 -----
21 -- Abstrakte Syntax:
22
23 'type' AAUSD -- abstrakter Ausdruck
24   sub(AAUSD, AAUSD)
25   bez(STRING)
26 -----
27 -- Konkrete Syntax und Umwandlung in abstrakte Syntax:
28
29 'nonterm' Ausdruck1(-> AAUSD)
30   'rule' Ausdruck1(-> sub(AA1, AA2)):
31     Ausdruck1(-> AA1) "-" Ausdruck2(-> AA2).
32   'rule' Ausdruck1(-> AA):
33     Ausdruck2(-> AA).
34 'nonterm' Ausdruck2(-> AAUSD)
35   'rule' Ausdruck2(-> bez(BEZ)):
36     Bezeich (-> BEZ).
37   'rule' Ausdruck2(-> AA):
38     "(" Ausdruck1(-> AA) ")".
39 'token' Bezeich(-> STRING)
40 -----
41 -- Uebersetzung in eine Befehlsfolge mit Ergebnis im Akkumulator:
42
43 'choice' AkkuBar(Ausdruck: AAUSD -> Kosten: INT)
44   'rule' AkkuBar(sub(bez(BEZ1), bez(BEZ2)) -> 20):
45     GibAus("load" , BEZ1, 10)
46     GibAus("sub_aku", BEZ2, 10)
47     $20
48   'rule' AkkuBar(sub(AA1, bez(BEZ2)) -> K1+10):
49     AkkuBar(AA1 -> K1)
50     GibAus("sub_aku", BEZ2, 10)
51     $10
52   'rule' AkkuBar(sub(bez(BEZ1), AA2) -> K2+20):
53     AkkuBar(AA2 -> K2)
54     GibAus("sub_aku", BEZ1, 10)
55     GibAus("neg_aku", "" , 10)
56     $20

```

```

57 'rule' AkkuBar(AA -> K+20):
58     StapelBar(AA -> K)
59     GibAus("pop_load", "", 20)
60     $20
61 -----
62 -- Uebersetzung in eine Befehlsfolge mit Ergebnis auf dem Stapel:
63
64 'choice' StapelBar(Ausdruck: AAUSD -> Kosten: INT)
65     'rule' StapelBar(bez(BEZ) -> 10):
66         GibAus("push", BEZ, 10)
67         $10
68     'rule' StapelBar(sub(AA1, AA2) -> K1+K2+20):
69         StapelBar(AA1 -> K1)
70         StapelBar(AA2 -> K2)
71         GibAus("sub_stk", "", 20)
72         $20
73     'rule' StapelBar(AA -> K+20):
74         AkkuBar(AA -> K)
75         GibAus("store_push", "", 20)
76         $20
77 -----
78 -- Uebersetzung in eine Befehlsfolge mit Ergebnis in ERGEB:
79
80 'choice' Uebersetzbar(Ausdruck: AAUSD -> Kosten: INT)
81     -- Ein abstrakter Ausdruck ist uebersetzbar, wenn er akkumulierbar oder
82     -- stapelbar ist. Alle abstrakten Ausdruecke sind uebersetzbar!
83     'rule' Uebersetzbar(AA -> K+10):
84         AkkuBar(AA -> K)
85         GibAus("store", "ERGEB", 10)
86         $10
87     'rule' Uebersetzbar(AA -> K+10):
88         StapelBar(AA -> K)
89         GibAus("pop", "ERGEB", 10)
90         $10
91 -----
92 -- Eine Zeile des Zielprogramms ausgeben:
93 'action' GibAus(Operator: STRING, Operand: STRING, Kosten: INT)
94     'rule' GibAus(S1, S2, K):
95         PutS(S1) Tab PutS(S2) Tab PutS("-- Kosten: ") PutI(K) Nl
96 -----
97 -- Um eine kleine Macke von Gentle (Version 230) zu umgehen (Wahl-
98 -- Praedikate duerfen nicht direkt im root-Praedikat aufgerufen werden):
99 'action' Uebersetze(Ausdruck: AAUSD -> Kosten: INT)
100     'rule' Uebersetze(AA -> K):
101         Uebersetzbar(AA -> K).
102 -----
103 -- Hilfspraedikate aus dem C-Modul text_io.o:
104 'action' PutS(STRING)
105 'action' PutI(INT)
106 'action' Nl
107 'action' Tab
108 -----
109 'root'
110     Ausdruck1 (                -> AbstrakterAusdruck)
111     Uebersetze(AbstrakterAusdruck -> Gesamtkosten    )
112     Tab PutS("Gesamtkosten:") Tab PutI(Gesamtkosten) Nl
113 --*****1*****2*****3*****4*****5*****6*****7*****

```

**Zeile 43:** "AkkuBar" soll an "akkumulierbar" erinnern. Ein abstrakter Ausdruck ist akkumulierbar, wenn er in eine Befehlsfolge übersetzt werden kann, die seinen Wert in den Akkumulator bringt. AkkuBar ist ein Wahlprädikat (choice predicate). Deshalb darf am Ende jeder Regel ein "Preis" stehen (siehe z. B. Zeile 47, 51 etc.). Der Preis gilt für eine Anwendung der Regel. Der Preis einer "Gesamtlösung" ist die Summe der Preise aller Regelanwendungen, die für diese Lösung nötig sind. Das Prädikat `AkkuBar` hat einen Ausgabeparameter vom Typ `INT` und liefert damit den Preis seiner Anwendung. Das geschieht nur hier in diesem "Demonstrations-Beispiel". In einem "ernsthaften Compiler" würde man die Kosten nicht als Parameter "weiterreichen" und alle damit zusammenhängen Parameter und Befehle könnten entfallen.

**Zeile 44:** Akkumulierbar ist z. B. der abstrakte Ausdruck `sub( bez(anna), bez(bert) )` (der dem konkreten Ausdruck `anna - bert` entspricht).

**Zeile 48 und 52:** Akkumulierbar sind auch solche abstrakten Subtraktionen, bei denen einer der beiden Operanden ein Bezeichner (und nicht etwa ein komplizierterer Ausdruck) ist.

**Zeile 54:** Diese Subtraktion subtrahiert "falsch rum" (den Wert der Variablen `BEZ1` vom Wert des Ausdrucks `AA2`, statt umgekehrt). Deshalb muss mit dem Maschinenbefehl `neg_aku` (siehe nächste Zeile 55) das Vorzeichen des Ergebnisses (im Akkumulator) umgedreht werden.

**Zeile 57:** Schließlich sind alle abstrakten Ausdrücke akkumulierbar, die stapelbar sind. Allerdings muss an die Befehlsfolge, die ein Ergebnis auf dem Stapel erzeugt, ein `pop_load`-Befehl angehängt werden und der kostet 20 \$. Die Umwandlung eines stapelbaren Ausdrucks in einen akkumulierbaren Ausdruck hat also einen bestimmten Preis.

**Zeile 64:** Auch `StapelBar` ist ein Wahlprädikat und jede Regel hat einen Preis. Außerdem liefert das Prädikat die Kosten seiner Anwendung in seinem Ausgabeparameter (was nur in diesem Demonstrations-Beispiel passiert und in einem ernsthaften Compiler wegfallen würde).

**Zeile 65:** Ein abstrakter Ausdruck, der nur aus einem Bezeichner besteht, ist stapelbar und "das Stapeln" eines solchen Ausdrucks kostet 10 \$.

**Zeile 68:** Ein abstrakter Ausdruck der Form `sub(AA1, AA2)` ist stapelbar, wenn seine Teilausdrücke `AA1` und `AA2` stapelbar sind. Die Kosten für das Stapeln eines solchen Ausdrucks berechnen sich wie folgt: `K1` \$ für das Stapeln des Teilausdrucks `AA1`, `K2` \$ für das Stapeln des Teilausdrucks `AA2` und 20 \$ für den anschließenden `sub_stk` Befehl.

**Zeile 73:** Schließlich sind alle Ausdrücke stapelbar, die akkumulierbar sind. Allerdings muss an die Befehlsfolge, die ein Ergebnis im Akkumulator erzeugt, ein `store_push`-Befehle (Kosten: 20 \$) angehängt werden. Auch die Umwandlung eines akkumulierbaren Ausdrucks in einen stapelbaren Ausdruck hat ihren Preis.

**Zeile 80:** Auch `Uebersetzbar` ist ein Wahlprädikat. Dieses Prädikat stellt den Code-Generator des Compilers dar. Natürlich sind alle abstrakten Ausdrücke übersetzbar und `Uebersetzbar` ist deshalb ein Aktions-Prädikat (und kein Bedingungs-Prädikat).

**Zeile 83:** Wenn ein abstrakter Ausdruck `AA` akkumulierbar ist (zu einem Preis von `K` \$), dann kann man seine Übersetzung (mit Ergebnis im Akkumulator) durch Anhängen eines Maschinenbefehls `store ERGEB` (zum Preise von 10 \$) vollenden. Die gesamte Codeerzeugung kostet dann offenbar `K + 10` \$.

**Zeile 85:** Wenn ein abstrakter Ausdruck `AA` stapelbar ist (zu einem Preis von `K` \$), dann kann man seine Übersetzung (mit Ergebnis auf dem Stapel) durch Anhängen eines Befehls `pop ERGEB` (zum Preise von 10 \$) vollenden. Die gesamte Codeerzeugung kostet dann offenbar `K + 10` \$.

Hier sechs Ausdrücke und die Befehlsfolgen, in die der obige Compiler `choice_1.g` sie übersetzt:

Anna-Bert-Carl-Doris-Emil

```

1 load      Anna      -- Kosten: 10
2 sub_aku   Bert       -- Kosten: 10
3 sub_aku   Carl       -- Kosten: 10
4 sub_aku   Doris     -- Kosten: 10
5 sub_aku   Emil      -- Kosten: 10
6 store    ERGEB     -- Kosten: 10
7          Gesamtkosten:60

```

Anna-((Bert-(Carl-Doris))-Emil)

```

1 load      Carl       -- Kosten: 10
2 sub_aku   Doris     -- Kosten: 10
3 sub_aku   Bert      -- Kosten: 10
4 neg_aku             -- Kosten: 10
5 sub_aku   Emil      -- Kosten: 10
6 sub_aku   Anna      -- Kosten: 10
7 neg_aku             -- Kosten: 10
8 store    ERGEB     -- Kosten: 10
9          Gesamtkosten:80

```

(Anna-Bert)-(Carl-Doris)

```

10 push     Anna      -- Kosten: 10
11 push     Bert      -- Kosten: 10
12 sub_stk           -- Kosten: 20
13 push     Carl      -- Kosten: 10
14 push     Doris     -- Kosten: 10
15 sub_stk           -- Kosten: 20
16 sub_stk           -- Kosten: 20
17 pop      ERGEB     -- Kosten: 10
18          Gesamtkosten:110

```

(Anna-Bert-Carl)-(Doris-Emil-Fanny)

```

19 load     Anna      -- Kosten: 10
20 sub_aku  Bert      -- Kosten: 10
21 sub_aku  Carl      -- Kosten: 10
22 store_push           -- Kosten: 20
23 load     Doris     -- Kosten: 10
24 sub_aku  Emil      -- Kosten: 10
25 sub_aku  Fanny     -- Kosten: 10
26 store_push           -- Kosten: 20
27 sub_stk           -- Kosten: 20
28 pop      ERGEB     -- Kosten: 10
29          Gesamtkosten:130

```

(Anna-Bert)-(Carl-Doris-Emil)-(Fanny-Gerd)

```

30 push     Anna      -- Kosten: 10
31 push     Bert      -- Kosten: 10
32 sub_stk           -- Kosten: 20
33 load     Carl      -- Kosten: 10
34 sub_aku  Doris     -- Kosten: 10
35 sub_aku  Emil      -- Kosten: 10
36 store_push           -- Kosten: 20
37 sub_stk           -- Kosten: 20
38 push     Fanny     -- Kosten: 10
39 push     Gerd      -- Kosten: 10

```

```

40 sub_stk          -- Kosten: 20
41 sub_stk          -- Kosten: 20
42 pop             ERGEB -- Kosten: 10
43                 Gesamtkosten:180
44

```

(Anna-Bert)-(Carl-Doris)-Emil-Fritz

```

45 push           Anna   -- Kosten: 10
46 push           Bert   -- Kosten: 10
47 sub_stk        -- Kosten: 20
48 push           Carl   -- Kosten: 10
49 push           Doris  -- Kosten: 10
50 sub_stk        -- Kosten: 20
51 sub_stk        -- Kosten: 20
52 pop_load       -- Kosten: 20
53 sub_aku        Emil   -- Kosten: 10
54 sub_aku        Fritz  -- Kosten: 10
55 store          ERGEB  -- Kosten: 10
56                 Gesamtkosten:150

```

**Aufgabe 31.1.:** Experimentieren Sie mit dem Compiler `choice_1`. Geben Sie ihm weitere Ausdrücke ein (z. B.  $(A-B-C)-D$ ,  $(A-B)-(C-D)-(E-F)$ ,  $(A-B)-C-(E-F)$  etc. und erklären Sie die Ausgabe des Compilers. Können Sie die Ausgabe des Compilers für eine beliebige Eingabe exakt voraussagen?

**Aufgabe 31.2.:** Variieren Sie den Compiler `choice_1`, indem Sie die "Preise" für einige Regeln von Wahlprädikaten verändern. Was passiert z. B., wenn Sie den Preis für eine Akkumulator-Subtraktion (`sub_aku`, siehe Zeile 46, 50 und 54) erhöhen, z. B. auf 20, 25 oder 30 \$? Welche Ausdrücke werden dann vom Compiler anders übersetzt als früher? Verändern Sie auch die Preise für andere Maschinenbefehle und versuchen Sie, das veränderte Verhalten des Compilers vorauszusagen.

**Aufgabe 31.3.:** Angenommen, die Stapel-und-Akkumulator-Maschine (SUA-Maschine) wird um einen neuen Maschinenbefehl erweitert: `sub_stk ANNA` subtrahiert den Wert der Variablen ANNA vom Inhalt des obersten Stapelelementes (`sub_stk` ohne Argument subtrahiert weiterhin das oberste Stapelelement vom zweitobersten Stapelelement). Legen Sie einen fairen Preis für die diesen neuen Maschinenbefehl fest (z. B. 15 \$) und erweitern Sie den Compiler `choice_1` entsprechend zu einem Compiler `choice_2`. Welche Ausdrücke werden von `choice_2` anders übersetzt als von `choice_1`?

**Aufgabe 31.4.:** Erweitern Sie die Stapel-und-Akkumulator-Maschine (SUA-Maschine) um weitere Maschinenbefehle zum Addieren, Multiplizieren und Dividieren von Ganzzahlen. Schreiben Sie einen Compiler `choice_3`, der entsprechend "reichhaltigere Ausdrücke" (in denen zusätzlich zum Minuszeichen `-` auch die Operationszeichen `+`, `*` und `/` vorkommen dürfen) übersetzen kann.

## 32. Erzeugung eines Compilers mit Gentle

Das Gentle-System besteht aus einer Reihe von Programmen, die aufeinander abgestimmt sind und miteinander kooperieren. Zu diesen Programmen gehören der Gentle-Compiler `gentle`, ein Hilfsprogramm namens `reflex`, der Scanner-Generator `flex`, der Parser-Generator `bison`, ein C-Compiler und ein Binder, z. B. das GNU-C-System `gcc`.

Jedes dieser Programme erwartet gewisse Eingabedateien und erzeugt gewisse Ausgabedateien. Diese "Kommunikation über Dateien" gibt dem Benutzer die Möglichkeit, die Erzeugung seines Compilers Schritt für Schritt zu verfolgen und, wenn er möchte, die Zwischenergebnisse zu modifizieren. Obwohl solche "Eingriffe auf niedriger Ebene" nur in Ausnahmefällen nötig sind, ist es für den Benutzer doch gut zu wissen, dass sie möglich sind.

Anhand eines einfachen Beispiels soll im folgenden erläutert werden, welche Dateien bei der Erzeugung eines Compilers von welchem Programm des Gentle-Systems verarbeitet ("eingelassen") bzw. erzeugt ("ausgegeben") werden. Ausgangspunkt ist der folgende, in der Sprache Gentle geschriebene Compiler `fanta.g`:

```

1  --*****1*****2*****3*****4*****5*****6*****7*****
2  -- fanta.g: Ein Mini-Compiler, der ein "Quell-Programm" wie etwa
3  -- begin OTTO := 17 end
4  -- in einen entsprechendes "Ziel-Programm"
5  -- MOV OTTO,17
6  -- uebersetzt.
7  -----
8  -- Abstrakte Syntax:
9  'type' AB_ZUWEIS -- abstrakte Zuweisungen
10  zug(Variable: STRING, Wert: INT)
11  -----
12  -- Konkrete Syntax und Umwandlung in abstrakte Syntax:
13  'nonterm' Programm(->AB_ZUWEIS) -- Einziges Zwischensymbol-Praedikat
14  'rule' Programm(->zug(VAR, ZAHL)): -- Einzige Regel fuer Programm
15  "begin" Bezeich(->VAR) -- "begin" ist ein Token-Muster
16  "!=" GanzLit(->ZAHL) -- "!=" ist ein Token-Muster
17  "end". -- "end" ist ein Token-Muster
18  'token' Bezeich(-> STRING) -- Ein Token-Praedikat
19  'token' GanzLit(->INT) -- Ein Token-Praedikat
20  -----
21  -- Hilfspraedikate aus dem C-Modul text_io:
22  'action' PutS(STRING) -- Gibt eine Zeichenkette vom Typ STRING aus
23  'action' PutI(INT) -- Gibt eine Ganzzahl vom Typ INT aus
24  'action' NL -- Gibt ein newline-Zeichen aus
25  'action' Tab -- Gibt ein Tabulator-Zeichen aus
26  -----
27  -- Das Hauptprogramm:
28  'root'
29  Programm(->zug(V, Z)) -- Quellprogramm einlesen.
30  PutS("MOV") Tab -- Das entsprechende
31  PutS(V) PutS(",") -- Zielprogramm
32  PutI(Z) NL -- ausgeben.
33  --*****1*****2*****3*****4*****5*****6*****7*****

```

Man beachte: Obwohl dieses Beispiel sehr einfach ist, enthält `fanta.g` doch immerhin ein Zwischensymbol-Prädikat (nonterm predicate), aus dem das Gentle-System einen Parser erzeugen muss. Außerdem werden in `fanta.g` drei Tokenmuster ("begin", "!=" und "end") und zwei Token-Prädika-

te (Bezeich und GanzLit) benützt. Die Token-Prädikate müssen vom Programmierer in zusätzlichen Dateien (siehe unten `ilit_ada.t` und `id_s_ada.t`) beschrieben werden. Aus den drei Tokenmustern und den beiden Token-Prädikaten muss das Gentle-System einen entsprechenden Scanner erzeugen.

Die folgende DOS-Stapeldatei `build.bat` beschreibt alle Arbeitsschritte, die zur Erzeugung des fanta-Compilers nötig sind:

```

1 rem -----
2 rem build.bat fuer fanta
3 rem -----
4 gentle fanta.g
5 reflex GanzLit=ilit_ada.t Bezeich=id_s_ada.t
6 flex gen.l
7 bison gen.y
8
9 @copy %genlib%\errmsg.o > nul
10 @copy %genlib%\mainl.o > nul
11 @copy %genlib%\grts.o > nul
12 @copy %genlib%\text_io.o > nul
13
14 gcc fanta.c lex.yy.c gen.tab.c text_io.o errmsg.o mainl.o grts.o
15 ren a.exe fanta.exe
16 fanta "begin OTTO := 123 end"
17 rem -----

```

**Zeile 04:** Die Datei `fanta.g` wird mit dem Compiler `gentle` in eine C-Datei namens `fanta.c` übersetzt. Außerdem erzeugt der `gentle`-Compiler die Dateien `gen.lit`, `gen.tkn`, `gen.h` und `gen.y`.

**Zeile 05:** Dem Programm `reflex` wird mitgeteilt, dass die Datei `ilit_ada.t` als Beschreibung des Token-Prädikates `GanzLit` und die Datei `id_s_ada.t` als Beschreibung des Token-Prädikates `Bezeich` nehmen soll. Daraufhin faßt `reflex` die Dateien `ilit_ada.t`, `id_s_ada.t`, `gen.lit`, `gen.tkn` und alle Dateien (im aktuellen Arbeitsverzeichnis), deren Name mit `.b` endet (d.h. die Dateien `comments.b` und `layout.b`) zu einer Datei namens `gen.l` zusammen.

**Zeile 06:** Die Datei `gen.l` sollte jetzt die Spezifikation eines Scanners enthalten. Der Scanner-Generator `flex` erzeugt daraus eine entsprechende C-Datei namens `lex.yy.c`.

**Zeile 07:** Die (vom `gentle`-Compiler erzeugte) Datei `gen.y` sollte die Spezifikation eines Parsers enthalten. Der Parser-Generator `bison` erzeugt daraus eine entsprechende C-Datei `gen.tab.c`.

**Zeile 09 bis 12:** Die mit dem Gentle-System mitgelieferten Standard-Module `errmsg.o`, `mainl.o`, `grts.o` und `text_io.o` werden in das aktuelle Arbeitsverzeichnis kopiert, damit das Kommando in Zeile 14 nicht zu lang wird. Wenn einem anstelle von DOS ein Betriebssystem zur Verfügung steht, ist dieser "Trick" natürlich nicht nötig. Die Datei `text_io.o` wird benötigt, weil im Gentle-Programm `fanta.g` die Prädikate `PutS`, `PutI` etc. benützt werden, die in dieser Datei realisiert sind. Die beiden Dateien `errmsg.o` und `mainl.o` sind nötig, weil aus `fanta.g` auch ein Parser erzeugt wird. Die Datei `grts.o` (gentle run time system) ist in jedem Fall nötig.

**Zeile 14:** Der C-Compiler-und-Binder `gcc` erzeugt aus den angegebenen Dateien ein Maschinenprogramm namens `a.exe`.

**Zeile 15:** Die Datei `a.exe` wird umbenannt in `fanta.exe`.

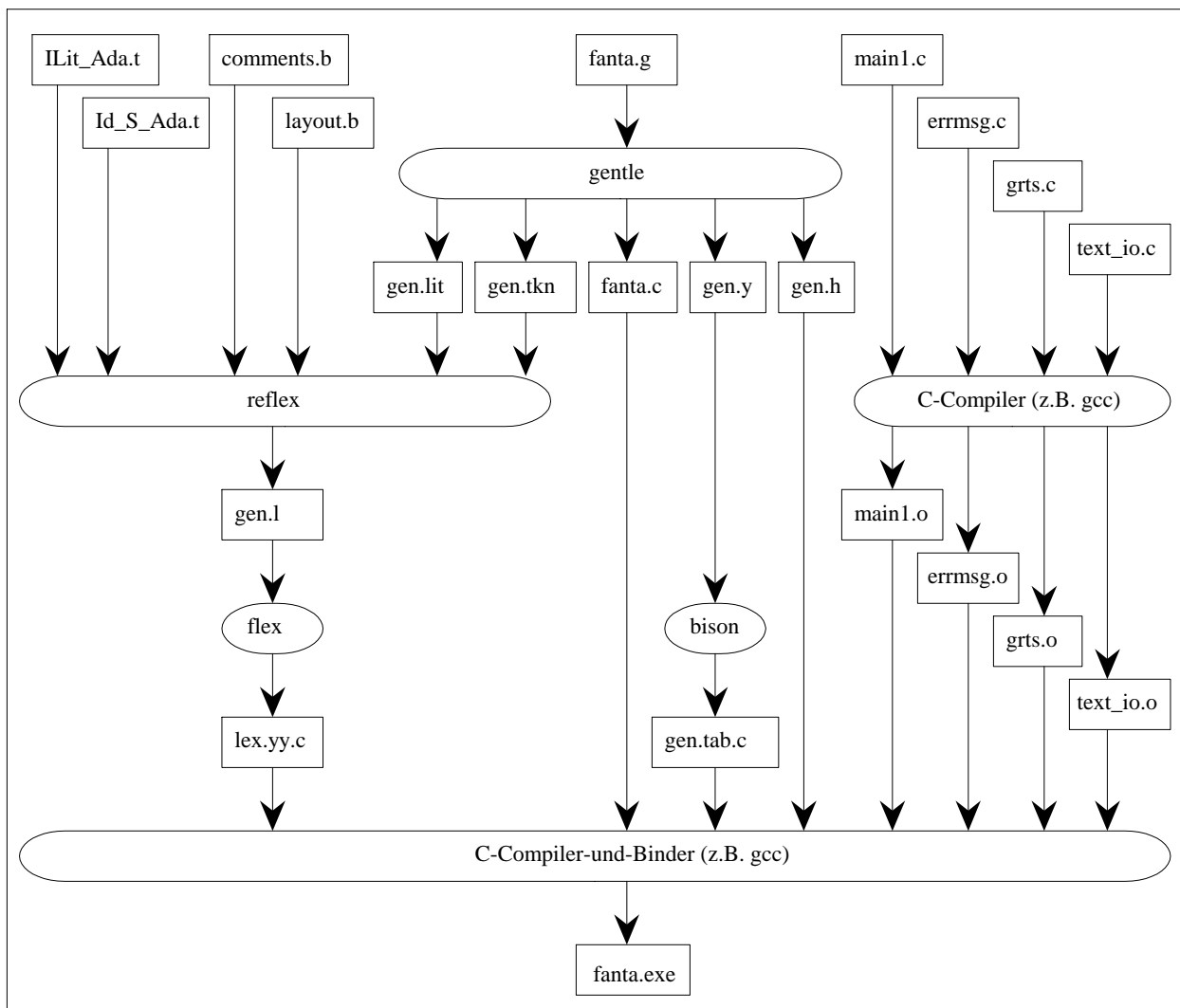
**Zeile 16:** Der Compiler `fanta.exe` wird aufgerufen. Auf der Kommandozeile wird ihm das Quellprogramm "begin OTTO := 123 end" übergeben.

Das folgende Diagramm zeigt, welche Dateien bei der Erzeugung des fanta-Compilers von welchem Programm des Gentle-Systems eingelesen bzw. ausgegeben werden:

Erläuterungen zum Diagramm: Die Programme, aus denen das Gentle-System besteht (der Gentle-Compiler `gentle`, der Scanner-Generator `flex`, der Parser-Generator `bison` etc.) sind in abgerundete Kästchen eingeschlossen. Die Dateien, die von den einzelnen Programmen eingelesen bzw. ausgegeben werden (z. B. `ilit_ada.t`, `comments.b`, `fanta.g`, `gen.lit` etc.), sind in eckige Kästchen eingeschlossen. Die Pfeile sollen für jedes Programm deutlich machen, welche Dateien es einliest und welche Dateien es ausgibt.

Erläuterungen zu den einzelnen Dateien:

Die Datei `ilit_ada.t`: Enthält eine Spezifikation für den Scanner-Generator `flex`. Diese Spezifi-



kation beschreibt, für welche Lexeme ("Zeichenketten") in einem Quellprogramm der Scanner (des fanta-Compilers) ein GanzLit-Token an den Parser (des fanta-Compilers) liefern soll.

`ilit_ada.t` beschreibt Ganzzahl-Literale (integer literals) wie sie auch in einem Ada-Programm er-

laubt sind (d.h. zwischen zwei Ziffern darf man einen Unterstrich `_` als "Verzierung" einfügen). Anstelle von `ilit_ada.t` könnte man auch die Datei `ilit_pas.t` verwenden (die Ganzzahl-Literale beschreibt, wie sie auch in einem Pascal-Programm erlaubt sind) oder eine eigene `flex`-Spezifikation (d.h. eine `.t`-Datei) schreiben. Wie man eine `flex`-Spezifikation schreibt, wird in jedem Buch über den Scanner-Generatro `flex` (bzw. `lex`) genau erläutert. Hier die Datei `ilit_ada.t`:

```

1 /* ----- */
2 /* ilit_ada.t beschreibt Ganzzahl-Literale, die aus dezimalen Ziffern */
3 /* und Unterstrichen bestehen duerfen (aehnlich wie in Ada). */
4 /* Ein Unterstrich darf aber nur zwischen zwei Ziffern stehen. */
5 /* Liefert den Wert des Literals als Ganzzahl vom Typ INT. */
6 /* ----- */
7 [0-9](_[0-9]|[0-9])* {
8     char *p = yytext;
9     int d = 0; /* Distanz, um die Zeichen verschoben werden */
10    while (*p != '\0') { /* Unterstriche "_" aus yytext entfernen */
11        if (*p == '_')
12            d++;
13        else
14            *(p-d) = *p; /* Zeichen um d Stellen nach links kopieren */
15        p++;
16    }
17    *(p-d) = '\0'; /* abschliessende Null nach yytext */
18
19    yylval.attr[1] = atoi (yytext);
20    yysetpos();
21    return GanzLit;
22 }
23 /* ----- */

```

Man beachte, dass in der `return`-Anweisung (Zeile 21) der Name `GanzLit` des entsprechenden Token-Prädikates angegeben werden muss (siehe oben die Datei `fanta.g`, Zeile 19).

Die Datei `id_s_ada.t`: Enthält ebenfalls eine Spezifikation für den Scanner-Generator `flex`. Diese Spezifikation beschreibt, für welche Lexeme in einem Quellprogramm der Scanner (des `fanta`-Compilers) ein `Bezeich`-Token an den Parser (des `fanta`-Compilers) liefern soll. `id_s_ada.t` beschreibt `Bezeichner` (identifizier) wie sie auch in einem Ada-Programm erlaubt sind. Anstelle von `id_s_ada.t` könnte man auch eine der Dateien `id_s_pas.t`, `id_s_eif.t` oder `id_s_ceh.t` verwenden (die `Bezeichner` beschreiben, wie sie auch in Pascal, Eiffel bzw. C-Programmen erlaubt sind) oder eine eigene `flex`-Spezifikation schreiben. Hier die Datei `id_s_ada.t`:

```

1 /* ----- */
2 /* id_s_ada.t: Bezeichner (identifizier) wie in Ada */
3 /* Liefert den Bezeichner als Zeichenkette vom Typ STRING */
4 /* ----- */
5 [A-Za-z](_[A-Za-z0-9]|[A-Za-z0-9])* {
6     yylval.attr[1] = strdup(yytext);
7     yysetpos();
8     return Bezeich;
9 }
10 /* ----- */

```

Man beachte, dass auch hier in der `return`-Anweisung (Zeile 08) der Name `Bezeich` des entsprechenden Token-Prädikates angegeben werden muss (siehe oben die Datei `fanta.g`, Zeile 18).

Die Datei `comments.b`: Mit dieser Datei wird dem Scanner-Generator `flex` mitgeteilt, welche Zeichenketten der von ihm erzeugte Scanner (in einem `fanta`-Quellprogramm) als Kommentare erkennen

soll. Wenn in dem Verzeichnis, in dem der fanta-Compiler erzeugt wird, keine Datei namens `comments.b` vorhanden ist, sind in den Quellprogrammen des fanta-Compilers keine Kommentare erlaubt. Hier die Datei `comments.b`:

```

1  /* -----
2     comments.b: Ermoeeglicht Kommentare im Ada-Stil (alle Zeichen
3     zwischen "--" und dem naechsten Zeilenende gelten als Kommentar)
4     -----
5  */
6  "--".*\n { yyPosToNextLine(); }
7  /* ----- */

```

Die Datei `layout.b`: Mit dieser Datei wird dem Scanner-Generator `flex` mitgeteilt, welche Zeichen der von ihm erzeugte Scanner zwischen zwei Lexemen (in einem fanta-Quellprogramm) zulaßen soll. Diese Trennzeichen werden im Englischen auch als `white space` bezeichnet, weil in vielen Sprachen Leerzeichen, Tabulator-Zeichen und Zeilenende-Zeichen als Trennzeichen zwischen Lexemen erlaubt sind, und diese Zeichen in einem Ausdruck des Quellprogramms als "weißer Zwischenraum" erscheinen. Wenn in dem Verzeichnis, in dem der fanta-Compiler erzeugt wird, keine Datei namens `layout.b` vorhanden ist, dann sind Leerzeichen, Tabulator-Zeichen und Zeilenende-Zeichen als Trennzeichen zwischen Lexemen erlaubt.

Wenn eine Datei namens `layout.b` vorhanden ist, diese Datei aber leer ist (d.h. wenn sie z. B. nur ein paar Leerzeichen enthält), dann sind nur Zeilenende-Zeichen zwischen Lexemen erlaubt. Hier die Datei `layout.b`:

```

1  /* -----
2     layout.b: Regelt "white space" fuer REFLEX und FLEX (bzw. LEX)
3     -----
4     Erlaubt (ausser Zeilenwechsel) nur Blanks und Bindestriche (-)
5     zwischen Lexemen.
6  */
7
8  [ -] { yypos += 1; }

```

Die Datei `gen.lit`: Enthält für jedes Zeichenketten-Literal, welches in der Datei `fanta.g` als Tokenmuster verwendet wurde, einen Eintrag:

```

9  "end" { yysetpos(); return yytoken_3; }
10 "!=" { yysetpos(); return yytoken_2; }
11 "begin" { yysetpos(); return yytoken_1; }

```

Die Datei `gen.tkn`: Enthält für jedes Tokenprädikat, welches in der Datei `fanta.g` erwähnt wurde, einen Eintrag:

```

1  Bezeich
2  GanzLit

```

Die Datei `gen.l` Wird vom Hilfsprogramm `reflex` aus den Dateien `ilit_ada.t`, `id_s_ada.t`, `comments.b`, `layout.b`, `gen.lit` und `gen.tkn` erzeugt. Enthält die Spezifikation eines Scanners (für den Scanner-Generator `flex`):

```

1  %{
2  #include "gen.h"
3  extern YYSTYPE yylval;
4  extern long yypos;
5  #define yysetpos() { yylval.attr[0] = yypos; yypos += yyleng; }
6  %}

```

```

7 %%
8 "end" { yysetpos(); return yykw_end; }
9 "!=" { yysetpos(); return yytk_COLON_EQUAL; }
10 "begin" { yysetpos(); return yykw_begin; }
11 /* ----- */
12 /* id_s_ada.t: Bezeichner (identifizier) wie in Ada */
13 /* Liefert den Bezeichner als Zeichenkette vom Typ STRING */
14 /* ----- */
15 [A-Za-z](_[A-Za-z0-9]|[A-Za-z0-9])* {
16     yylval.attr[1] = strdup(yytext);
17     yysetpos();
18     return Bezeich;
19 }
20 /* ----- */
21 /* ----- */
22 /* ilit_ada.t beschreibt Ganzzahl-Literale, die aus dezimalen Ziffern */
23 /* und Unterstrichen bestehen duerfen (aehnlich wie in Ada). */
24 /* Ein Unterstrich darf aber nur zwischen zwei Ziffern stehen. */
25 /* Liefert den Wert des Literals als Ganzzahl vom Typ INT. */
26 /* ----- */
27 [0-9](_[0-9]|[0-9])* {
28     char *p = yytext;
29     int d = 0; /* Distanz, um die Zeichen verschoben werden */
30     while (*p != '\0') { /* Unterstriche "_" aus yytext entfernen */
31         if (*p == '_')
32             d++;
33         else
34             *(p-d) = *p; /* Zeichen um d Stellen nach links kopieren */
35         p++;
36     }
37     *(p-d) = '\0'; /* abschliessende Null nach yytext */
38
39     yylval.attr[1] = atoi (yytext);
40     yysetpos();
41     return GanzLit;
42 }
43 /* ----- */
44 /* ----- */
45 comments.b: Ermoeeglicht Kommentare im Ada-Stil (alle Zeichen
46 zwischen "--" und dem naechsten Zeilenende gelten als Kommentar)
47 -----
48 /*
49 "--".*\n { yyPosToNextLine(); }
50 /* ----- */
51 \ { yypos += 1; }
52 \n { yyPosToNextLine(); }
53 \t { yypos += 1; }
54 . { yysetpos(); yyerror("illegal token"); }
55 %%
56 #ifndef yywrap
57 yywrap() { return 1; }
58 #endif

```

Die Datei `gen.h`: Eine Kopffdatei (a header file), die vom `gentle`-Compiler erzeugt wird, wenn er die Datei `fanta.g` übersetzt:

```

1 typedef struct {long attr[2];} yyATTRIBUTES;
2 #define YYSTYPE yyATTRIBUTES
3 extern YYSTYPE yylval;
4

```

```
5 #define Bezeich 257
6 #define GanzLit 258
7 #define yykw_end 259
8 #define yytk_COLON_EQUAL 260
9 #define yykw_begin 261
```

Die Datei `gen.y`: Wird vom `gentle-Compiler` erzeugt, wenn er die Datei `fanta.g` übersetzt. Enthält die Spezifikation eines Parsers (für den Parser-Generator `bison`):

```
1 %{
2 typedef long * yy;
3 #define yyu (-2147483647L)
4 static yy yynull;
5 extern yy yyh;
6 extern yy yyhx;
7 static yyErr(n,l)
8 {
9 yyAbort(n,"fanta", l);
10 }
11 /* start */
12 /* end */
13 typedef struct {long attr[2];} yyATTRIBUTES;
14 #define YYSTYPE yyATTRIBUTES
15 extern YYSTYPE yylval;
16
17 %}
18 %start ROOT_
19 %token Bezeich 257
20 %token GanzLit 258
21 %token yykw_end 259
22 %token yytk_COLON_EQUAL 260
23 %token yykw_begin 261
24 %%
25 Programm:
26     yykw_begin
27     Bezeich
28     yytk_COLON_EQUAL
29     GanzLit
30     yykw_end
31
32 {
33 YY yyb;
34 YY YY_0_1;
35 YY YY_0_1_1;
36 YY YY_0_1_2;
37 YY yyv_VAR;
38 YY YY_2_1;
39 YY yyv_ZAHL;
40 YY YY_4_1;
41 YY_2_1 = (yy)($2.attr[1]);
42 YY_4_1 = (yy)($4.attr[1]);
43 yyb = yyh;
44 yyh += 3; if (yyh > yyhx) yyExtend();
45 yyv_VAR = YY_2_1;
46 yyv_ZAHL = YY_4_1;
47 YY_0_1_1 = yyv_VAR;
48 YY_0_1_2 = yyv_ZAHL;
49 YY_0_1 = yyb + 0;
50 YY_0_1[0] = 1;
```

```

51 yy_0_1[1] = ((long)yy_0_1_1);
52 yy_0_1[2] = ((long)yy_0_1_2);
53 $$$.attr[1] = ((long)yy_0_1);
54 $$$.attr[0] = $1.attr[0];
55 }
56 ;
57 ROOT_:
58   Programm
59
60 {
61   YY yyb;
62   YY YY_1_1;
63   YY yyv_V;
64   YY YY_1_1_1;
65   YY yyv_Z;
66   YY YY_1_1_2;
67   YY YY_2_1;
68   YY YY_4_1;
69   YY YY_5_1;
70   YY YY_6_1;
71   YY_1_1 = (yy)($1.attr[1]);
72   if (yy_1_1[0] != 1) goto yyfl_9_1;
73   yy_1_1_1 = ((yy)yy_1_1[1]);
74   yy_1_1_2 = ((yy)yy_1_1[2]);
75   yyv_V = yy_1_1_1;
76   yyv_Z = yy_1_1_2;
77   yy_2_1 = ((yy)"MOV");
78   PutS(yy_2_1);
79   Tab();
80   YY_4_1 = yyv_V;
81   PutS(yy_4_1);
82   yy_5_1 = ((yy)",");
83   PutS(yy_5_1);
84   YY_6_1 = yyv_Z;
85   PutI(yy_6_1);
86   Nl();
87   $$$.attr[0] = $1.attr[0];
88   goto yyfl_9_1_A;
89   yyfl_9_1 : ;
90   yyErr(3,28);
91   yyfl_9_1_A : ;
92 }
93 ;

```

main1.c bzw. main1.o: Jeder in Gentle geschriebene Compiler muss durch ein C-Hauptprogramm (d.h. durch eine C-Funktion namens main) ergänzt werden. Die Datei main1.c enthält eine solche C-Funktion. Diese C-Funktion bewirkt, dass man dem fertigen fanta-Compiler Quellprogramme auf einem von drei "Wegen" übergeben kann:

1. Beim Aufruf des Compilers, indem man das Quellprogramm direkt als Zeichenkette angibt
2. Beim Aufruf des Compilers, indem man einen Dateinamen angibt. Das Quellprogramm muss dann in der entsprechenden Datei stehen.
3. Über die Standardeingabe (meistens ist das die Tastatur). In diesem Fall ruft man den Compiler ohne weitere Parameter auf, tippt dann das Quellprogramm ein und schließt mit einem Dateiende-Zeichen (<Strg>-Z bzw. <Strg>-D) ab.

```

1 // Datei main1.c
2 /* -----
3 Als Teil eines in GENTLE geschriebenen Programms COMPX erlaubt dieses
4 main-Programm, die Eingabe fuer COMPX auf eine der folgenden drei Arten
5 anzugeben:
6 1. COMPX pfname -- Als Eingabe dient die entsprechende Datei
7 2. COMPX abcdefgh -- Als Eingabe dient die Zeichenkette "abcdefgh"
8 3. COMPX -- Als Eingabe dient die Standardeingabe
9 Bei 2. wird die Zeichenkette "abcdefgh" in eine Datei namens TMP.TMP
10 geschrieben und diese Datei dient dann als Eingabe fuer COMPX.
11 ----- */
12 #include <stdio.h>
13
14 extern FILE *yyin;
15 // -----
16 main (argc, argv)
17     int     argc;
18     char **argv;
19
20 {
21     /* INITIALIZE */
22     if (argc > 2) {
23         printf("too many arguments\n");
24         exit(1);
25     } // if
26
27     if (argc == 2) {
28         yyin = fopen (argv[1], "r");
29         if (yyin == NULL) {
30             printf("Will use '%s' as input for '%s'\n\n", argv[1], argv[0]);
31             yyin = fopen("tmp.tmp", "w");
32             fputs(argv[1], yyin);
33             fclose(yyin);
34             yyin = fopen("tmp.tmp", "r");
35         }
36     } // if
37
38     /* INVOKE GENERATED PROGRAM */
39     ROOT();
40
41     /* FINALIZE */
42     exit(0);
43 } // main
44 // -----

```

Für einfache "Compiler", die gar keinen Parser enthalten, sollte man anstelle von `main1.c` die Datei `main0.c` verwenden. Die Datei `main2.c` funktioniert ähnlich, wie `main1.c`, erlaubt aber nur 2 Wege, über die man dem `fanta`-Compiler seine Quellprogramme übergeben kann (die Angabe als Zeichenkette auf der Kommandozeile beim Aufruf des Compilers ist dann nicht möglich).

`errmsg.c` bzw. `errmsg.o`: Jeder in Gentle geschriebene Compiler, der einen Parser enthält (d.h. in dessen Gentle-Teil mindestens ein Zwischensymbol-Prädikat (nonterm predicate) vereinbart wurde), muss durch eine C-Funktion namens `errmsg` ergänzt werden. Der vom Parser-Generator `bison` erzeugte Parser ruft diese Funktion auf, wenn er in einem Quellprogramm einen Syntaxfehler entdeckt. Die Datei `errmsg.c` enthält eine solche C-Funktion namens `errmsg`. Es gibt kaum einen Grund, diese Datei gegen eine andere auszutauschen.

`grts.c` bzw. `grts.o`: Jeder in Gentle geschriebene Compiler muss durch das Gentle Laufzeitsystem (gentle run time system, grts) ergänzt werden. Dieses Laufzeitsystem enthält einige Grundfunktionen, die jeder in Gentle geschriebene Compiler benötigt.

`text_io.c` bzw. `text_io.o`: Ein in Gentle geschriebener Compiler kann durch beliebig viele in C geschriebene Module ergänzt werden. Die Datei `text_io.c` ist ein Beispiel für einen solchen C-Modul. Sie enthält Funktionen namens `PutS`, `PutI`, `PutP` etc., mit denen man Werte vom Typ `STRING`, `INT` bzw. `POS` ausgeben kann. Siehe dazu auch oben die Zeilen 22 bis 25 in der Datei `fanta.g`.

Ein weiterer wichtiger C-Modul ist `idents.c`. Er enthält die Vereinbarung eines Typs `IDENT` und Funktionen zur Verwaltung einer Symboltabelle (implementiert durch eine Hash-Tabelle). Falls nötig, kann man weitere Module in C programmieren und in einem Gentle-Programm benutzen.

## 33. Sachwortverzeichnis

|                        |            |                        |            |                       |              |
|------------------------|------------|------------------------|------------|-----------------------|--------------|
| _ (Egal-Zeichen)       | 82         | leerer                 | 54, 79     | conditional statement | 69           |
| -> (Pfeil rechts)      | 11, 88     | Und/Oder-              | 65         | <b>D</b>              |              |
| : Doppelpunkt          | 7          | bedingte Anw.          | 69         | Darstellung           |              |
| . (Punkt)              | 28, 53, 59 | Bedingungs-Präd.       | 57, 72     | Zwischen-             | 13           |
| .b-Datei               | 114f.      | Beispielprogramme (.g) |            | Datei                 |              |
| .t-Datei               | 42, 53     | ahnen_v1               | 7          | .b-                   | 114f.        |
| (  (Alt.-Anw.)         | 70         | ahnen_v2               | 11         | .t-                   | 42, 53       |
| [  (bed. Anw.)         | 70         | ahnen_v3               | 13         | definierende Pos.     | 64           |
| @ (bei)                | 92         | ahnen_v4               | 15         | DefMeaning            | 37           |
| \$ (Dollar)            | 107        | ahnen_v5               | 16         | deklarative Prog.Spr. | 73           |
| ** (Potenzfunkt.)      | 49         | altcon_1               | 69         | differenzieren        | 51           |
| <- (Pfeil links)       | 88         | calcul_1               | 44         | Doppelpunkt :         | 7            |
| ) (Alt.-Anw.)          | 70         | calcul_2               | 47         | dynamische            |              |
| ]  (bed. Anw.)         | 70         | calcul_3               | 50         | Konstanten            | 85f.         |
| <b>A</b>               |            | calcul_4               | 52         | <b>E</b>              |              |
| ableiten               | 51         | choice_1               | 105        | Egal-Zeichen _        | 82           |
| abstrakte Syntax       | 13, 25     | condit_1               | 70         | Eiffel                | 33           |
| action predicate       | 57         | condit_2               | 73         | Eingabeparameter      | 11, 15       |
| Ada                    | 33         | digits_1               | 28         | Eins-zu-Eins          | 31           |
| akkumulierbar          | 103        | fanta                  | 110        | eq (equal)            | 90           |
| Aktions-Präd.          | 57         | hello_v1               | 5          | errmsg.c              | 9, 118       |
| Algorithmus            |            | idei                   | 38         | <b>F</b>              |              |
| Unifizierungs-         | 56         | ides                   | 32         | fail                  | 56           |
| alternative statement. | 69         | int                    | 41         | farbiger Baum         | 24           |
| Alternativen-Anw.      | 69         | sweep_v1               | 77         | Feger-Präd.           | 2, 77, 81    |
| Anweisung              |            | syntab_1               | 94         | flaches Wiederaufs.   | 68           |
| Alternativen-          | 69         | syntab_2               | 96         | flex                  | 8, 34, 110,  |
| bedingte               | 69         | syntab_3               | 98         | 112                   |              |
| anwenden               |            | variav_v1              | 87         | Funktion              | 56, 83       |
| eine Variable          | 13         | Bezeichner             | 32, 38     | funktionale Prog.Spr. | 73           |
| anwendende Position    | 64         | binärer Baum           | 24, 73     | Funktionsaufrufe      |              |
| Aufzählungstyp         | 22         | Binärzahl              | 28         | in Mustern            | 84           |
| Ausdruck               | 57, 59, 82 | bison                  | 8, 34, 110 | Funktor               | 14, 22f., 54 |
| Ausdruck               |            | blockorientiert        | 94, 98     | nullstelliger         | 24           |
| Wert eines             | 57         | build.bat              | 5          | <b>G</b>              |              |
| Ausgabeparameter       | 11, 15, 27 | <b>C</b>               |            | ge (greater or equal) | 90           |
| <b>B</b>               |            | C (die Sprache)        | 33         | gelingen              | 56           |
| backtracking           |            | C-Modul                | 119        | gen.h                 | 115          |
| deep                   | 68         | choice predicate       | 103        | gen.l                 | 8            |
| shallow                | 68         | comments.b             | 10, 114f.  | gen.y                 | 8, 116       |
| Baum                   | 48         | conc2                  | 15         | Generator             |              |
| binärer                | 24, 73     | condition              | 72, 75     | Lexer-                | 8            |
| farbiger               | 24         | condition predicate    | 72         | Scanner-              | 8            |

|                         |               |                       |                 |  |  |
|-------------------------|---------------|-----------------------|-----------------|--|--|
| gentle                  |               |                       |                 |  |  |
| Compiler                | 6, 8, 34, 110 |                       |                 |  |  |
| run time system         | 9             |                       |                 |  |  |
| Grammatik, kontextfreie | 4             |                       |                 |  |  |
| grts.c                  | 9, 119        |                       |                 |  |  |
| Grund                   |               |                       |                 |  |  |
| -Spezialfall            | 54            |                       |                 |  |  |
| -Term                   | 54            |                       |                 |  |  |
| gt (greater than)       | 90            |                       |                 |  |  |
| <b>H</b>                |               |                       |                 |  |  |
| HasMeaning              | 37            |                       |                 |  |  |
| header file gen.h       | 115           |                       |                 |  |  |
| Hewlett Packard         | 26            |                       |                 |  |  |
| Hex.-Zahlen             | 29            |                       |                 |  |  |
| <b>I</b>                |               |                       |                 |  |  |
| id_i_ada.t              | 40            |                       |                 |  |  |
| id_i_ceh.t              | 40            |                       |                 |  |  |
| id_i_eif.t              | 40            |                       |                 |  |  |
| id_i_pas.t              | 40            |                       |                 |  |  |
| id_s_ada.t              | 36, 113, 115  |                       |                 |  |  |
| id_s_ceh.t              | 36            |                       |                 |  |  |
| id_s_eif.t              | 36            |                       |                 |  |  |
| id_s_pas.t              | 35            |                       |                 |  |  |
| id_to_string            | 37            |                       |                 |  |  |
| idents.c                | 37            |                       |                 |  |  |
| ilit_ada.t              | 43            |                       |                 |  |  |
| ilit_pas.t              | 42            |                       |                 |  |  |
| imperative Prog.Spr.    | 73            |                       |                 |  |  |
| Infixnotation           | 26            |                       |                 |  |  |
| int                     | 11            |                       |                 |  |  |
| INT                     | 11            |                       |                 |  |  |
| invocation              | 56            |                       |                 |  |  |
| <b>J</b>                |               |                       |                 |  |  |
| Johann                  | 19            |                       |                 |  |  |
| John                    | 19            |                       |                 |  |  |
| Jugend, harte           | 83            |                       |                 |  |  |
| <b>K</b>                |               |                       |                 |  |  |
| Knuth, Donald           | 28            |                       |                 |  |  |
| Kommentare              | 5, 10         |                       |                 |  |  |
| Konkatenation           | 15            |                       |                 |  |  |
| konkrete Syntax         | 25            |                       |                 |  |  |
| Konstanten              | 85            |                       |                 |  |  |
| dynamische              | 85f.          |                       |                 |  |  |
| statische               | 85            |                       |                 |  |  |
| kontextfreie Grammatik  | 4             |                       |                 |  |  |
| Kopfdatei gen.h         | 115           |                       |                 |  |  |
|                         |               | <b>L</b>              |                 |  |  |
|                         |               | layout.b              | 10, 114         |  |  |
|                         |               | le (less or equal)    | 90              |  |  |
|                         |               | leer                  | 54, 72          |  |  |
|                         |               | Lexem                 | 31              |  |  |
|                         |               | Lexer                 | 8               |  |  |
|                         |               | Literale              |                 |  |  |
|                         |               | als Muster            | 82f.            |  |  |
|                         |               | lokal (Regel-)        | 65              |  |  |
|                         |               | lt (less than)        | 90              |  |  |
|                         |               | <b>M</b>              |                 |  |  |
|                         |               | main0.c               | 9, 118          |  |  |
|                         |               | main1.c               | 9, 117          |  |  |
|                         |               | Maria                 | 19              |  |  |
|                         |               | Mary                  | 19              |  |  |
|                         |               | Maschinen-Spr.        | 46              |  |  |
|                         |               | matching              |                 |  |  |
|                         |               | pattern               | 2, 54, 56, 58   |  |  |
|                         |               | misslingen            | 56              |  |  |
|                         |               | Modula                | 33              |  |  |
|                         |               | Muster                | 56f., 59, 82    |  |  |
|                         |               | Muster                |                 |  |  |
|                         |               | mit Namen             | 84              |  |  |
|                         |               | Token-                | 110             |  |  |
|                         |               | Musterabgleich        | 2, 54, 56, 58   |  |  |
|                         |               | <b>N</b>              |                 |  |  |
|                         |               | Name                  |                 |  |  |
|                         |               | eines Musters         | 84              |  |  |
|                         |               | ne (not equal)        | 90              |  |  |
|                         |               | nix                   | 53, 72          |  |  |
|                         |               | Nom                   |                 |  |  |
|                         |               | Poly-                 | 53              |  |  |
|                         |               | nonterm               | 4, 7            |  |  |
|                         |               | nonterm predicate     | 12, 27          |  |  |
|                         |               | nonterminal symbol    | 4, 7            |  |  |
|                         |               | Notation              |                 |  |  |
|                         |               | Infix-                | 26              |  |  |
|                         |               | polnische             | 26              |  |  |
|                         |               | Postfix-              | 26              |  |  |
|                         |               | Präfix-               | 26              |  |  |
|                         |               | <b>O</b>              |                 |  |  |
|                         |               | Optimierung           | 103             |  |  |
|                         |               | <b>P</b>              |                 |  |  |
|                         |               | Parameter             |                 |  |  |
|                         |               | Ausgabe-              | 11, 15, 27      |  |  |
|                         |               | Eingabe-              | 11, 15          |  |  |
|                         |               | Parser                | 7               |  |  |
|                         |               | Pascal                | 33              |  |  |
|                         |               | pattern matching      | 2, 54, 56, 58   |  |  |
|                         |               | Pfeil (->)            | 11, 88          |  |  |
|                         |               | Pfeil (<-)            | 88              |  |  |
|                         |               | polnische Notation    | 26              |  |  |
|                         |               | Poly                  |                 |  |  |
|                         |               | -Nom                  | 53              |  |  |
|                         |               | Polynom               | 51              |  |  |
|                         |               | POS                   | 22              |  |  |
|                         |               | Position              | 22              |  |  |
|                         |               | Position              |                 |  |  |
|                         |               | anwendende            | 64              |  |  |
|                         |               | definierende          | 64              |  |  |
|                         |               | Postfixnotation       | 26              |  |  |
|                         |               | Potenzfunktion        | 49              |  |  |
|                         |               | Prädikate             | 5, 56           |  |  |
|                         |               | @ (bei)               | 92              |  |  |
|                         |               | Aktions-              | 57              |  |  |
|                         |               | Bedingungs-           | 57, 62, 72      |  |  |
|                         |               | eq (equal)            | 90              |  |  |
|                         |               | Feger-                | 2, 77, 81       |  |  |
|                         |               | ge (greater or equal) | 90              |  |  |
|                         |               | gt (greater than)     | 90              |  |  |
|                         |               | le (less or equal)    | 90              |  |  |
|                         |               | lt (less than)        | 90              |  |  |
|                         |               | ne (not equal)        | 90              |  |  |
|                         |               | Token-                | 32, 38, 42, 111 |  |  |
|                         |               | Vereinb. von          | 28              |  |  |
|                         |               | vordefinierte         | 90              |  |  |
|                         |               | Wahl-                 | 103             |  |  |
|                         |               | where                 | 91              |  |  |
|                         |               | Wurzel                | 12              |  |  |
|                         |               | Zwischensymbol-       | 12, 27          |  |  |
|                         |               | Präfixnotation        | 26              |  |  |
|                         |               | predicates            |                 |  |  |
|                         |               | action                | 57              |  |  |
|                         |               | choice                | 103             |  |  |
|                         |               | condition             | 72              |  |  |
|                         |               | nonterm               | 12, 27          |  |  |
|                         |               | root                  | 12              |  |  |
|                         |               | sweep                 | 2, 77           |  |  |
|                         |               | print                 | 5, 14           |  |  |
|                         |               | Programmiersprachen   |                 |  |  |
|                         |               | deklarative           | 73              |  |  |

|                  |                |                    |            |                  |          |
|------------------|----------------|--------------------|------------|------------------|----------|
| funktionale      | 73             | SUA-Maschine       | 103        | <b>V</b>         |          |
| imperative       | 73             | succeed            | 56         | Variable         |          |
| Punkt .          | 28, 53, 59     | sweep              | 80         | anwenden         | 13       |
| PutS             | 17             | sweep predicate    | 2, 77      | Regel-lokale     | 65       |
| <b>R</b>         |                | symbol             |            | unveränd.        | 86       |
| rechtsassoziativ | 51             | nonterminal        | 4, 7       | veränd.          | 86       |
| record type      | 22             | Symbol             |            | vereinbaren      | 13, 72   |
| reflex           | 8, 34, 45      | End-               | 4, 7       | Zeiger-          | 86       |
| Regel-lokal      | 65             | Zwischen-          | 4, 7, 45   | variante VT      | 23       |
| Regeln           | 12, 79         | Symboltabelle      | 93         | veränd. Variable | 86       |
| Reihenfolge      | 64             | Symboltabelle      |            | veränd. Variable |          |
| rem (remainder)  | 49             | blockorientierte   | 94, 98     | mit Namen        | 87       |
| ren (rename)     | 39             | Syntax             |            | mit Zeiger       | 87       |
| Restfunktion rem | 49             | abstrakte          | 13, 25     | Verbundtyp       | 22       |
| rnix             | 79             | konkrete           | 25         | vereinbaren      |          |
| root             | 5              | <b>T</b>           |            | eine Variable    | 13, 72   |
| root predicate   | 12             | Term               | 48, 54     | einen Funktor    | 14       |
| rule             | 12, 28, 53, 79 | Term               |            | einen Typ        | 14       |
| <b>S</b>         |                | Spezialfall eines  | 54         | Vereinbarung     | 28       |
| Scanner          | 8, 31          | text_io.c          | 17, 119    | Viele-zu-Eins    | 31, 41   |
| scheitern        | 68             | tiefes Wiederaufs. | 68         | vordefinierte    |          |
| Schlüsselwort    |                | token              | 38         | Funktionen       | 83       |
| action           | 15             | Token              | 31         | Prädikate        | 90       |
| condition        | 72, 75         | -Muster            | 110        | Typen            | 22       |
| nonterm          | 4, 7           | -Prädikat          | 32, 38, 42 | <b>W</b>         |          |
| root             | 5              | Typ                |            | Wahl-Präd.       | 103      |
| rule             | 28, 53         | INT                | 11         | Wert             | 57, 59   |
| sweep            | 80             | POS                | 22         | where            | 91       |
| token            | 38             | vereinbaren        | 14         | Wiederaufsetzen  |          |
| Schröder         | 5              | type               | 14         | flaches          | 68       |
| snix             | 79             | record             | 22         | tiefes           | 68       |
| Spezialfall      | 54             | struct             | 22f.       | Wurzel-Prädikat  | 12       |
| Sprache          |                | union              | 23         | <b>Y</b>         |          |
| Maschinen-       | 46             | Typen              |            | yyerror          | 9        |
| stack machine    | 46             | Aufzählungs-       | 22         | yyin             | 9        |
| Stapel-Maschine  | 46             | rekursive          | 23         | yylval           | 35       |
| stapelbar        | 103            | variante VT        | 23         | yylval.attr      | 35       |
| statement        |                | Verbund-           | 22         | yysetpos         | 35       |
| alternative      | 69             | vordefinierte      | 22         | <b>Z</b>         |          |
| conditional      | 69             | <b>U</b>           |            | Zeigervariable   | 86       |
| statische        |                | unbuild.bat        | 6          | Zeigerwert       | 86       |
| Konstanten       | 85             | Und/Oder-Baum      | 65         | Zuweisung        | 89       |
| str_hand.c       | 16             | Unifizierung       | 56         | Zwischendarst.   | 13       |
| string_to_id     | 37             | union type         | 23         | Zwischensymbol   | 4, 7, 45 |
| struct type      | 22f.           | unveränd. Variable | 86         | -Prädikate       | 12, 27   |