Bogenfix

Erzeugt von Doxygen 1.6.1

Thu Jun 30 12:53:42 2011

Inhaltsverzeichnis

Kapitel 1

Ausstehende Aufgaben

Global node_table_remove Testen, ob bei Unterschreitung eines Belegungsfaktors von 1/4 ein node_table_shrink Vorteile bei Performance und/oder Speicherverbrauch bewirkt.

Global rtsearch Bei der Betrachtung eines Nichtterminalknotens bietet sich eine Möglichkeit zur Fehlererkennung an: Wenn für den selben Wert von `i` von zwei verschiedenen Nachfolgern aus assoziierte Endknoten gemeldet werden, ist die Grammatik nicht LR(k)! Bequem zu testen bspw. über einen Bitvergleich in `prefixes_num`!

Global yyparse Ist eine clean-Routine auch für den Ableitungswald nötig? Eigentlich nicht, weil diese Funktionalität ja nur zu Diagnosezwecken eingebaut wurde und im Produktiveinsatz zwecks Zeitersparnis nicht aktiviert würde.

Kapitel 2

Datenstruktur-Verzeichnis

2.1 Datenstrukturen

Hier folgt die Aufzählung aller Datenstrukturen mit einer Kurzbeschreibung:

Kapitel 3

Datei-Verzeichnis

3.1 Auflistung der Dateien

Hier folgt die Aufzählung aller Dateien mit einer Kurzbeschreibung:

Kapitel 4

Datenstruktur-Dokumentation

4.1 item_t Strukturreferenz

Datenstruktur zur Repräsentation von Parser-Items.

Datenfelder

- int rule_number

  Nummer der Grammatikregel, aus der das Item entstanden ist.

- enum symbol_type dot_symbol_type

  Art des Symbols, das direkt hinter dem Punkt folgt (Nichtterminalsymbol, Terminalsymbol oder ε).

- unsigned long dot_symbol

  Nummer des Symbols hinter dem Punkt. Undefiniert bei ε.

- int first_k_count

  Anzahl der Elemente in der FIRST_k-Menge der rechten Seite des Items.

- unsigned long(* first_k_item)[LOOKAHEAD_LENGTH]

  Zeiger auf das Array, dass die FIRST_k-Menge der rechten Seite des Items enthält.

4.1.1 Ausführliche Beschreibung

Datenstruktur zur Repräsentation von Parser-Items. Der Parsergenerator BoGenLR erzeugt aus der Menge der Produktionen der Grammatik ein Array `items`[] mit Einträgen von diesem Typ. Jedem Item wird eine eindeutige Nummer zugeordnet, die zugleich sein Arrayindex ist.

Beim Generieren der Items wurde die Reihenfolge so gewählt, dass für ein Item $[A \rightarrow \alpha \cdot x\beta]$ bzw. $[A \rightarrow \alpha \cdot B\beta]$ mit Nummer l das Nachfolgeitem die Nummer $l+1$ erhält. Dies ist das Item, das nach Lesen von x bzw. nach Reduktion eines Nachfolgerknotens von B entsteht.

Nur zur Dokumentation hier aufgenommen, wird normalerweise vom Generator eingefügt.

4.1.2 Dokumentation der Datenelemente

4.1.2.1 unsigned long item_t::dot_symbol

Nummer des Symbols hinter dem Punkt. Undefiniert bei ε.

4.1.2.2 enum symbol_type item_t::dot_symbol_type

Art des Symbols, das direkt hinter dem Punkt folgt (Nichtterminalsymbol, Terminalsymbol oder ε).

4.1.2.3 int item_t::first_k_count

Anzahl der Elemente in der FIRST_k-Menge der rechten Seite des Items.

4.1.2.4 unsigned long(∗ item_t::first_k_item)[LOOKAHEAD_LENGTH]

Zeiger auf das Array, dass die FIRST_k-Menge der rechten Seite des Items enthält. Die rechte Seite eines Items $[A \to \alpha \cdot a\beta], a \in \Sigma$, ist $a\beta$; die rechte Seite von $[A \to \alpha \cdot B\beta], B \in N$, ist β.

4.1.2.5 int item_t::rule_number

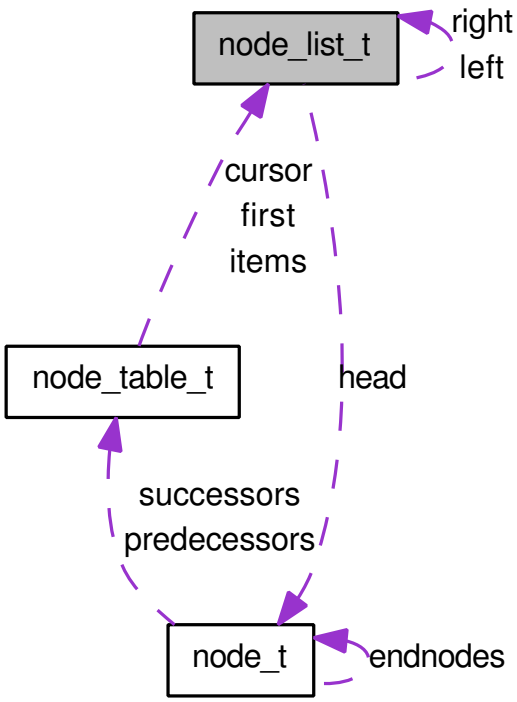
Nummer der Grammatikregel, aus der das Item entstanden ist.

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- bogenfix.c

4.2 node_list_t Strukturreferenz

Datentyp für Listen von Knoten. Zusammengehörigkeiten von node_list_t:



Datenfelder

- struct node_t ∗ head

  Zeiger auf den gespeicherten Knoten.

- struct node_list_t ∗ left

  Zeiger auf das linke Nachbarelement.

- struct node_list_t ∗ right

  Zeiger auf das rechte Nachbarelement.

4.2.1 Ausführliche Beschreibung

Datentyp für Listen von Knoten. Eine Knotenliste besteht aus einem Knotenzeiger *head*, gefolgt von Zeigern *left* und *right* auf die linken und rechten Nachbarlisten.

4.2.2 Dokumentation der Datenelemente

4.2.2.1 struct node_t∗ node_list_t::head [read]

Zeiger auf den gespeicherten Knoten.

4.2.2.2 struct node_list_t* node_list_t::left [read]

Zeiger auf das linke Nachbarelement.

4.2.2.3 struct node_list_t* node_list_t::right [read]

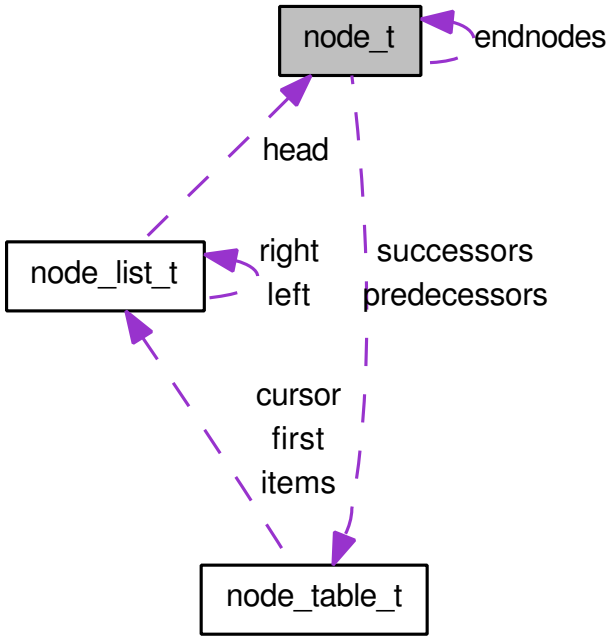
Zeiger auf das rechte Nachbarelement.

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- bogenfix.c

4.3 node_t Strukturreferenz

Datentyp für die Knoten des Graphen. Zusammengehörigkeiten von node_t:



Datenfelder

- signed long content

  Die Nummer des repräsentierten Items oder Symbols.

- struct node_table_t ∗ predecessors

  Hashtafel der direkten Vorgängerknoten im Graphen.

- struct node_table_t ∗ successors

  Hashtafel der direkten Nachfolgerknoten im Graphen.

- unsigned long last_visited

  Laufende Nummer des Suchlaufs, in dem der Knoten zuletzt besucht wurde.

- struct node_t ∗ endnodes [LOOKAHEAD_LENGTH]

  Für die rückwärts-topologische Suche: Array aus Zeigern auf die Endknoten, die zu den bisher hergeleiteten Präfixen des Lookahead-Strings korrespondieren.

- unsigned long prefixes

  Für die rückwärts-topologische Suche: Zusätzliche Integer- Darstellung eines Präfixvektors.

4.3.1 Ausführliche Beschreibung

Datentyp für die Knoten des Graphen. Der erweiterte LR(k)-Parser verwaltet einen Graphen, dessen Knoten Items oder Nichtterminalsymbole repräsentieren können.

4.3.2 Dokumentation der Datenelemente

4.3.2.1 signed long node_t::content

Die Nummer des repräsentierten Items oder Symbols. Wenn der Knoten ein Item repräsentiert, enthält `content` die Nummer dieses Items als positive Zahl; wenn der Knoten für ein Nichtterminal steht, enthält `content` die mit dem Operator $\sim$ negierte Nummer des Symbols.

4.3.2.2 struct node_t* node_t::endnodes[LOOKAHEAD_LENGTH] [read]

Für die rückwärts-topologische Suche: Array aus Zeigern auf die Endknoten, die zu den bisher hergeleiteten Präfixen des Lookahead-Strings korrespondieren. Dabei bedeutet für einen Knoten v ein Endknoten-Eintrag endnodes[i] = w != `NULL`: Zwischen v und w konnte ein Präfix der Länge i des Lookaheads hergeleitet werden, und der Endknoten w ist mit diesem Präfix assoziiert.

4.3.2.3 unsigned long node_t::last_visited

Laufende Nummer des Suchlaufs, in dem der Knoten zuletzt besucht wurde. Um während der rückwärts-topologischen Suche festzustellen, ob der Präfixvektor des betrachteten Knotens aktuell ist, wird während jedes Aufrufs der Suchfunktion ein globaler Zähler `search_id` inkrementiert und ggf. mit `node->last_visited` abgeglichen.

4.3.2.4 struct node_table_t* node_t::predecessors [read]

Hashtafel der direkten Vorgängerknoten im Graphen.

4.3.2.5 unsigned long node_t::prefixes

Für die rückwärts-topologische Suche: Zusätzliche Integer- Darstellung eines Präfixvektors. Wird benutzt, um Endlosschleifen in Zyklen des Graphen zu vermeiden, indem beim Besuchen eines Itemknotens v geprüft wird, ob sich der Präfixvektor des nachfolgenden Nichtterminalknotens v seit dem letzten Besuch verändert hat. Nur in diesem Fall lohnt sich ein erneuter Besuch von v. Für einen Nichtterminalknoten wird als Bitmaske gespeichert, welche Präfixe des Lookahead zum aktuellen Zeitpunkt bis zu diesem Knoten hergeleitet werden können. Für Itemknoten wird gespeichert, wie der Präfixvektor des unmittelbar nachfolgenden Nichtterminalknotens zu dem Zeitpunkt aussah, als der Itemknoten zuletzt untersucht wurde. Bedeutung der Werte:

- 0 - Kein Element des Vektors ist gesetzt.
- 1 - Nur das Präfix Epsilon (1 = 2^0)
- 2 - Nur das Präfix der Länge 1 (2 = 2^1)
- 3 - Das Präfix Epsilon und das der Länge 1 (3 = 2^0 + 2^1)
- 4 - Nur das Präfix der Länge 2 (4 = 2^2)

- ...

4.3.2.6 struct node_table_t* node_t::successors [read]

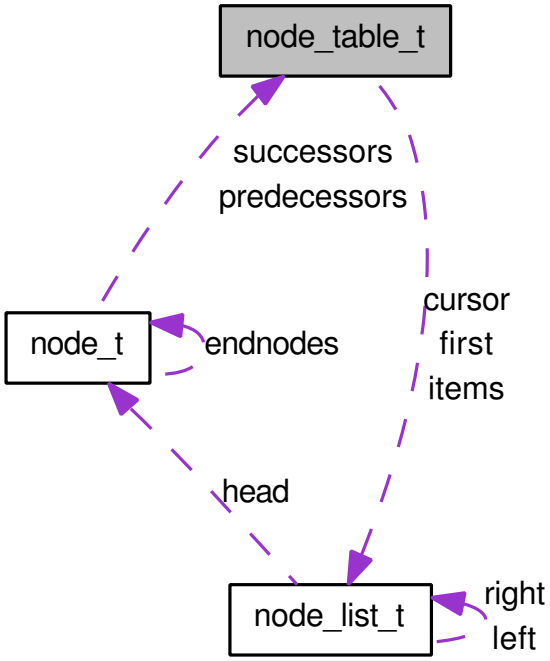
Hashtafel der direkten Nachfolgerknoten im Graphen.

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- bogenfix.c

4.4 node_table_t Strukturreferenz

Datentyp für eine Hashtabelle von Knoten. Zusammengehörigkeiten von node_table_t:



Datenfelder

- size_t length

  Maximallänge der Tabelle.

- size_t item_count

  Anzahl Elemente in der Tabelle.

- struct node_list_t * cursor

  Zeiger für die aktuelle Position in einer Iteration durch die Tabelle.

- struct node_list_t * first

  Zeiger auf das "erste" Element der Tabelle.

- struct node_list_t ** items

  Dynamisches Array aus Knotenlisten.

4.4.1 Ausführliche Beschreibung

Datentyp für eine Hashtabelle von Knoten. Aus Effizienzgründen werden die verschiedenen Knotenmengen, die während eines Parserlaufes verwaltet werden müssen, durch Hashtabellen repräsentiert.

4.4.2 Dokumentation der Datenelemente

4.4.2.1 struct node_list_t* node_table_t::cursor [read]

Zeiger für die aktuelle Position in einer Iteration durch die Tabelle.

4.4.2.2 struct node_list_t* node_table_t::first [read]

Zeiger auf das "erste" Element der Tabelle. Dies ist das Element, mit dem eine Iteration per `node_table_iter` startet, nachdem der Cursor mit `node_table_reset_cursor` initialisiert wurde. Dies ist auch das Element, das bei `node_table_pop` geliefert und aus der Tabelle entfernt wird.

4.4.2.3 size_t node_table_t::item_count

Anzahl Elemente in der Tabelle. Entspricht der Anzahl Felder im Array `items`. Wenn die Elementanzahl `item_count` diesen Wert erreicht, wird die Tabelle mittels `node_table_double` vergrößert.

4.4.2.4 struct node_list_t** node_table_t::items [read]

Dynamisches Array aus Knotenlisten. Ein Eintrag in der Hashtabelle ist ein Zeiger auf den ersten Knoten der internen Liste, dessen Hashwert der Tabellenposition entspricht.

4.4.2.5 size_t node_table_t::length

Maximallänge der Tabelle.

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- bogenfix.c

4.5 nonterminal_t Strukturreferenz

Datentyp zur Repräsentation der Nichtterminalsymbole (= Variablen) der Grammatik.

Datenfelder

- int item_count

  Anzahl Elemente des Arrays, auf das der Zeiger items zeigt.

- int ∗ items

  Zeiger auf ein Array mit den Nummern derjenigen Items, für die bei einer Expansion anhand des aktuell betrachteten Symbols neue Nachfolgerknoten in den Graphen eingefügt werden müssen.

- char ∗ name

  Zeiger auf einen String mit dem Klartextnamen des Symbols.

4.5.1 Ausführliche Beschreibung

Datentyp zur Repräsentation der Nichtterminalsymbole (= Variablen) der Grammatik. Der Generator `bogenlr` erzeugt ein Array `nonterminals`[] von diesem Typ.

Nur zur Dokumentation hier aufgenommen, wird normalerweise vom Generator eingefügt.

4.5.2 Dokumentation der Datenelemente

4.5.2.1 int nonterminal_t::item_count

Anzahl Elemente des Arrays, auf das der Zeiger *items* zeigt.

4.5.2.2 int∗ nonterminal_t::items

Zeiger auf ein Array mit den Nummern derjenigen Items, für die bei einer Expansion anhand des aktuell betrachteten Symbols neue Nachfolgerknoten in den Graphen eingefügt werden müssen.

4.5.2.3 char∗ nonterminal_t::name

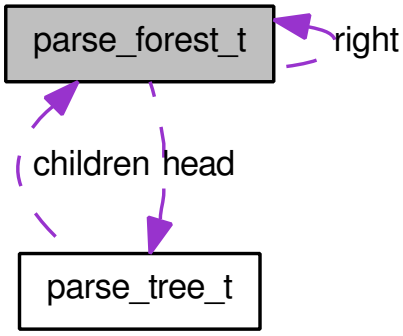
Zeiger auf einen String mit dem Klartextnamen des Symbols. Wird bei definiertem Makro `PARSE_TREE` benutzt, um beim Erzeugen eines Ableitungsbaumes die inneren Knoten mit den Namen der korrespondierenden Nichtterminalsymbolen zu beschriften.

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- bogenfix.c

4.6 parse_forest_t Strukturreferenz

Datenstruktur für Wälder aus partiellen Ableitungsbäumen. Zusammengehörigkeiten von parse_forest_t:



Datenfelder

- struct parse_tree_t ∗ head

  Der am weitesten rechts stehende Baum des Waldes.

- struct parse_forest_t ∗ right

  Zeiger auf den Restwald.

4.6.1 Ausführliche Beschreibung

Datenstruktur für Wälder aus partiellen Ableitungsbäumen. Ein Wald von (partiellen) Ableitungsbäumen besteht aus einem am weitesten rechts gelegenen Baum als Kopfelement und einem Zeiger auf den Restwald links.

Wird nur compiliert, falls das Makro `PARSE_TREE` definiert ist.

4.6.2 Dokumentation der Datenelemente

4.6.2.1 struct parse_tree_t∗ parse_forest_t::head [read]

Der am weitesten rechts stehende Baum des Waldes.

4.6.2.2 struct parse_forest_t∗ parse_forest_t::right [read]

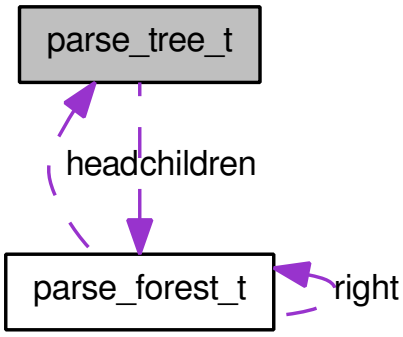
Zeiger auf den Restwald.

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- bogenfix.c

4.7 parse_tree_t Strukturreferenz

Datenstruktur für Ableitungsbäume. Zusammengehörigkeiten von parse_tree_t:



Datenfelder

- signed long root

  Wurzelsymbol.

- struct parse_forest_t ∗ children

  Liste der Nachfolgerknoten.

4.7.1 Ausführliche Beschreibung

Datenstruktur für Ableitungsbäume. Ein (partieller) Ableitungsbaum besteht aus einer Wurzel, die die Integerrepräsentation eines Symbols enthält, und einer Liste der Nachkommen. Das Symbol ε wird durch den Wert 0 codiert, Terminalsymbole durch positive Ganzzahlen, Nichtterminalsymbole durch negative, und zwar als "∼<symbolnummer>". Ausserdem könnte man anhand der Position im Baum bestimmen, ob der aktuelle Knoten ein Terminal- oder Nichtterminalsymbol enthält: Blattknoten können nur Terminalsymbole inkl. ε enthalten, innere Knoten nur Nichtterminalsymbole.

Wird nur compiliert, falls das Makro `PARSE_TREE` definiert ist.

4.7.2 Dokumentation der Datenelemente

4.7.2.1 struct parse_forest_t∗ parse_tree_t::children [read]

Liste der Nachfolgerknoten.

4.7.2.2 signed long parse_tree_t::root

Wurzelsymbol.

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- bogenfix.c

4.8 rule_t Strukturreferenz

Datenstruktur zur Repräsentation der Produktionen der Grammatik.

Datenfelder

- signed long lhs

  Nummer des Nichtterminalsymbols auf der linken Regelseite.

- int rhs_count

  Anzahl der Symbole auf der rechten Regelseite.

- signed long ∗ rhs

  Zeiger auf ein Array mit den Symbolen der rechten Regelseite.

4.8.1 Ausführliche Beschreibung

Datenstruktur zur Repräsentation der Produktionen der Grammatik. Wird nur compiliert, wenn das Makro `PARSE_TREE` definiert ist.

Der Generator erzeugt ein Array `rules`[] von diesem Typ. Die Nummer jeder Regel ist zugleich ihr Arrayindex und wird durch die Reihenfolge ihres Auftretens in der Grammatikdatei bestimmt.

Nur zur Dokumentation hier aufgenommen, wird normalerweise vom Generator eingefügt.

4.8.2 Dokumentation der Datenelemente

4.8.2.1 signed long rule_t::lhs

Nummer des Nichtterminalsymbols auf der linken Regelseite.

4.8.2.2 signed long∗ rule_t::rhs

Zeiger auf ein Array mit den Symbolen der rechten Regelseite. Dabei stehen positive Zahlen für Terminalsymbole und negative für die mit dem Operator ∼ negierte Nummer eines Nichtterminalsymbols.

4.8.2.3 int rule_t::rhs_count

Anzahl der Symbole auf der rechten Regelseite.

Die Dokumentation für diese Struktur wurde erzeugt aufgrund der Datei:

- bogenfix.c

Kapitel 5

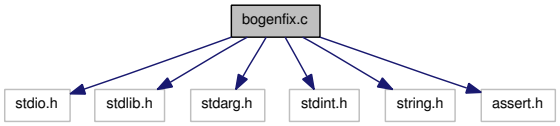
Datei-Dokumentation

5.1 bogenfix.c-Dateireferenz

Parserskelett für BoGenLR (Bonn Generator for LR(k) Parsers). `#include <stdio.h>`

```
#include <stdlib.h>
#include <stdarg.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>
```

Include-Abhängigkeitsdiagramm für bogenfix.c:



Datenstrukturen

- struct nonterminal_t

  Datentyp zur Repräsentation der Nichtterminalsymbole (= Variablen) der Grammatik.

- struct item_t

  Datenstruktur zur Repräsentation von Parser-Items.

- struct rule_t

  Datenstruktur zur Repräsentation der Produktionen der Grammatik.

- struct node_t

  Datentyp für die Knoten des Graphen.

- struct node_list_t

  Datentyp für Listen von Knoten.

- struct node_table_t

  Datentyp für eine Hashtabelle von Knoten.

- struct parse_tree_t

  Datenstruktur für Ableitungsbäume.

- struct parse_forest_t

  Datenstruktur für Wälder aus partiellen Ableitungsbäumen.

Makrodefinitionen

- #define malloc(x) (xmalloc((x)))

  Abfangen von fehlgeschlagenen Speicheranforderungen.

- #define DEBUG_PRINTF(...) (fprintf(stderr, __VA_ARGS__))
- #define HASHLENGTH 7
- #define LIST_PRINT(x) (list_print((x)))
- #define NODE_TABLE_CHECK(tbl, flag) (node_table_check((tbl), (flag)))
- #define NODE_TABLE_PRINT(x) (node_table_print((x)))
- #define PARSE_FOREST_READ(x, y) (parse_forest_read((x), (y)))
- #define PARSE_FOREST_REDUCE(x, y) (parse_forest_reduce((x), (y)))
- #define PARSE_FOREST_PRINT(x) (parse_forest_print((x)))

Aufzählungen

- enum symbol_type { Epsilon, Nonterminal, Terminal }

  Legt fest, welche Arten von Symbolen direkt nach dem Punkt in einem Item auftreten können.

- enum searchflag_t { Read, Reduce, Both }

  Aufzählungstyp für die gewünschte Suchmenge bei der rückwärts-topologischen Suche.

Funktionen

- static void * xmalloc (size_t size)

  Abfangen von fehlgeschlagenen Speicheranforderungen.

- static struct node_table_t * node_table_init (void)

  Erzeugen einer frischen Hashtabelle der Größe 0.

- static struct node_t * create_node (signed long content)

  Erzeugen eines neuen Knotens.

- static struct node_list_t * pool_get (void)

  Anfordern eines neuen Listenelementes aus dem Pool.

- static void pool_release (struct node_list_t *elem)

Zurückgeben eines aktuell nicht benötigten Listenelementes an den Pool.

- static void pool_release_list (struct node_list_t *first)
  Zurückgeben einer kompletten Liste an den Pool.
- static void list_add (struct node_list_t **list, struct node_t *node)
  Anhängen eines neuen Elementes an den Anfang einer Knotenliste.
- static int list_enqueue (struct node_list_t **list, struct node_t *node)
  Duplikatfreies Anhängen eines neuen Elementes an das Ende einer Knotenliste.
- static int list_contains_node_num (struct node_list_t *list, signed long node_number)
  Überprüft, ob in einer Knotenliste ein Knoten mit einer bestimmten Nummer enthalten ist.
- static struct node_t * list_pop (struct node_list_t **list)
  Entnimmt den ersten Knoten einer Liste.
- static int list_remove (struct node_list_t **list, struct node_t *node)
  Sucht einen Knoten anhand seines Zeigers und entfernt ihn aus einer Knotenliste.
- static void list_print (struct node_list_t *list)
  Ausgabe einer Knotenliste auf der Standardausgabe.
- static void list_clear (struct node_list_t **list)
  Vollständiges Leeren einer Knotenliste.
- static unsigned int compute_hash (struct node_t *node, size_t length)
  Berechnen des Hashwertes für einen Knoten.
- static int list_contains_sublist (struct node_list_t *lst, struct node_list_t *sublst)
  Test, ob eine Liste in einer anderen enthalten ist.
- static void node_table_check (struct node_table_t *tbl, int flag)
  Integritätstest für eine Knoten-Hashtafel.
- static void node_table_clear (struct node_table_t *tbl)
  Leeren einer Hashtabelle von Knoten.
- static void node_table_free (struct node_table_t *tbl)
  Vollständiges Löschen einer Knotentabelle.
- static void node_table_add (struct node_table_t *tbl, struct node_t *node)
  Hinzufügen eines neuen Knotens zu einer Hashtabelle.
- static void node_table_double (struct node_table_t *tbl)
  Verdoppeln einer vollen Hashtabelle.
- static void node_table_remove (struct node_table_t *tbl, struct node_t *node)
  Entfernen eines Knotens aus einer Hashtabelle.

- static int node_table_empty (struct node_table_t *tbl)
  Test, ob eine Hashtafel leer ist.
- static struct node_t * node_table_find (struct node_table_t *tbl, signed int content)
  Suchen eines Knotens in einer Knotentabelle anhand seines content-Feldes.
- static struct node_t * node_table_pop (struct node_table_t *tbl)
  Entnimmt einen beliebigen Knoten aus einer Hashtabelle.
- static void node_table_reset_cursor (struct node_table_t *tbl)
  Zurücksetzen des Cursors in einer Hashtabelle.
- static struct node_t * node_table_iter (struct node_table_t *tbl)
  Iterator über einer Hashtabelle.
- static struct node_table_t * node_table_copy (struct node_table_t *tbl)
  Kopieren einer bestehenden Hashtabelle.
- static void node_table_print (struct node_table_t *tbl)
  Ausgabe eine Hashtafel von Knoten auf stderr.
- static void parse_forest_read (struct parse_forest_t **forest, unsigned long symbol)
  Bei Leseschritt Erschaffung eines neuen partiellen Ableitungsbaums.
- static void parse_forest_reduce (struct parse_forest_t **forest, int item_number)
  Bei Reduktion Zusammenfassen mehrerer partieller Ableitungsbäume zu einem neuen.
- static void parse_tree_print (struct parse_tree_t *tree, unsigned long father)
  Ausgabe eines partiellen Ableitungsbaums auf `stdout`.
- static void parse_forest_print_rec (struct parse_forest_t *forest)
  Rekursive Ausgabe eines Baumes im Parsewald und zuvor aller linker Nachbarn.
- static void parse_forest_print (struct parse_forest_t *forest)
  Ausgabe eine Parsewaldes.
- static void End (void)
  Sauberer Abschluss der Ableitungsbaum-Ausgabe.
- static void clean (void)
  Speicherbereinigung am Ende des Parsevorgangs.
- static int call_lexer (void)
  Kapselung des yylex-Aufrufs.
- static void graph_adjust (struct node_t **node)
  Die Graphjustierung entfernt einen Knoten samt Vorgängern.
- static void expand (void)
  Durchführung eines Expansionsschrittes.

- static struct node_t * rtsearch (enum searchflag_t searchflag)

  Durchführung der rückwärts-topologischen Suche nach reduzier- oder lesbaren Endknoten.

- static void test_new_endnode (struct node_t **node)

  Hilfsfunktion, um einen neu entstandenen Endknoten nach einem Reduktions-/Leseschritt zu überprüfen.

- static void treat_predecessors (struct node_t **nt_node)

  Hilfsfunktion, um die Vorgängerknoten eines reduzierten Endknotens zu behandeln.

- static int reduce_read (void)

  Durchführung eines Reduktions-/Leseschrittes.

- static int yyparse (void)

  Rahmenprozedur für den Parser.

Variablen

- static signed long const startitem_number = 0

  Reservierte Item-Nummer.

- static signed long const acceptitem_number = 1

  Reservierte Item-Nummer.

- static int eof_reached = 0

  Flag, ob das Ende der Eingabe erreicht ist.

- static unsigned long phase_number = 0

  Zähler für die aktuelle Phase.

- static unsigned long search_id = 0

  Laufende Nummer für rückwärts-topologische Suchen.

- static unsigned long lookahead [LOOKAHEAD_LENGTH]

  Puffer für die nächsten k ungelesenen Eingabesymbole.

- static struct node_list_t * graph = NULL

  Der Parser verwaltet einen Graph aus Item- und Nichtterminalknoten.

- static struct node_table_t * expandable_nodes = NULL

  Hashtafel für expandierbare Endknoten.

- static struct node_table_t * readable_nodes = NULL

  Hashtafel für lesbare Endknoten.

- static struct node_list_t * new_readable_nodes = NULL

  Temporäre Liste für lesbare Endknoten, die nach einem Reduktions-/Leseschritt neu entstanden sind:.

- static struct node_table_t * reducible_nodes = NULL
  Hashtafel für reduzierbare Endknoten.
- static struct node_table_t * expanded_nodes = NULL
  Hashtafel bereits expandierter Knoten.
- static struct node_table_t * expanded_nonterminals = NULL
  Hashtafel bereits expandierter Nichtterminalknoten.
- static struct parse_forest_t * parse_forest = NULL
  Der Wald von partiellen Ableitungsbäumen wird über den globalen Zeiger `parse_forest` angesprochen.
- static unsigned long node_number = 0
  Für die Ausgabe des Parsewaldes: globale Zählvariable für die Knoten.

5.1.1 Ausführliche Beschreibung

Parserskelett für BoGenLR (Bonn Generator for LR(k) Parsers). In dieser Datei ist der C-Code für den feststehenden Teil des Resultatparsers festgelegt, d.h. die Datenstrukturen und Funktionen, die von einer konkreten Grammatik unabhängig sind. Sämtlicher weiterer Code wird anhand der Grammatikdatei erstellt, die dem Programm `bogenlr` als Eingabe übergeben wird.

Einige Makros, die in der Grammatikdatei oder beim Compileraufruf definiert werden können, und ihre Effekte:

- USE_DEBUG_PRINTF: Erzeugt ausführliche Ausgaben zum Parsevorgang auf stderr.
- DEBUG_ALL: Aktiviert erweiterte DEBUG-Ausgaben und fügt einigen Funktionen Integritätstests der verwendeten Datenstrukturen hinzu.
- PARSE_TREE: Der Parser gibt nach erfolgreichem Parsevorgang den Ableitungsbaum für die Eingabe auf stdout aus. Ausgabeformat ist die dot-Sprache, eine Beschreibungssprache für Graphen zu Benutzung mit dem Programm `dot` aus dem graphviz-Paket.
- PARSE_TREE_PHASES: Gibt nicht nur den einen Baum aus, sondern nach jeder Phase den Wald aus bisher erzeugten partiellen Ableitungsbäumen.
- DOXYGEN_ONLY: Compiliert Codeteile mit, die eigentlich an anderer Stelle erzeugt werden, bspw. im Parsergenerator bogenlr. Dient dazu, dass auch diese Codeabschnitte in der automatisch erzeugten Dokumentation erläutert werden.
- HASHLENGTH: Überschreibt den Defaultwert der Startlänge für Hashtabellen, der in dieser Skelettdatei auf 7 festgelegt ist.
- DNDEBUG: Abschalten der `assert()` -Überprüfungen.
- SUPPRESS_OUTPUT: Ausschalten der Erfolgsmeldung nach stderr.

5.1.2 Makro-Dokumentation

5.1.2.1 #define DEBUG_PRINTF(...) (fprintf(stderr, __VA_ARGS__))

Diagnose-Informationen nach `stderr` werden nur dann ausgegeben, wenn beim Übersetzen das Makro `USE_DEBUG_PRINTF` definiert ist. Ansonsten wird das Makro `DEBUG_PRINTF` zu `(void(0))` expandiert.

5.1.2.2 #define HASHLENGTH 7

Startwert für die Größe der Hashtabellen, in denen die Knoten des Parsegraphen verwaltet werden.

5.1.2.3 #define LIST_PRINT(x) (list_print((x)))

Falls das Makro `USE_DEBUG_PRINTF` nicht definiert ist, expandiert `LIST_PRINT` zu `(void)0`.

5.1.2.4 #define malloc(x) (xmalloc((x)))

Abfangen von fehlgeschlagenen Speicheranforderungen. Falls das Makro `DEBUG_ALL` definiert ist, werden Aufrufe von `malloc` vom Makro `xmalloc` abgefangen.

5.1.2.5 #define NODE_TABLE_CHECK(tbl, flag) (node_table_check((tbl), (flag)))

Falls das Makro `DEBUG_ALL` nicht definiert ist, expandiert das Makro `NODE_TABLE_CHECK` zu `(void)0`.

5.1.2.6 #define NODE_TABLE_PRINT(x) (node_table_print((x)))

Falls das Makro `USE_DEBUG_PRINTF` nicht definiert ist, expandiert `NODE_TABLE_PRINT` zu `(void)0`.

5.1.2.7 #define PARSE_FOREST_PRINT(x) (parse_forest_print((x)))

Falls das Makro `PARSE_TREE` nicht definiert ist, expandiert `PARSE_FOREST_PRINT` zu `(void)0`.

5.1.2.8 #define PARSE_FOREST_READ(x, y) (parse_forest_read((x), (y)))

Falls das Makro `PARSE_TREE` nicht definiert ist, expandiert `PARSE_FOREST_READ` zu `(void)0`.

5.1.2.9 #define PARSE_FOREST_REDUCE(x, y) (parse_forest_reduce((x), (y)))

Falls das Makro `PARSE_TREE` nicht definiert ist, expandiert `PARSE_FOREST_REDUCE` zu `(void)0`.

5.1.3 Dokumentation der Aufzählungstypen

5.1.3.1 enum searchflag_t

Aufzählungstyp für die gewünschte Suchmenge bei der rückwärts-topologischen Suche. Durch übergabe des entsprechenden Parameters an die Funktion `rtsearch` wird festgelegt, ob nur lesbare, nur reduzierbare, oder ob beide Endknotentypen durchsucht werden sollen.

Aufzählungswerte:

Read

Reduce

Both

```
00230 {
00231   Read, Reduce, Both
00232 };
```

5.1.3.2 enum symbol_type

Legt fest, welche Arten von Symbolen direkt nach dem Punkt in einem Item auftreten können. Möglich sind: Nichtterminalsymbole, Terminalsymbole und das leere Wort ε.

Nur zur Dokumentation hier aufgenommen, wird normalerweise vom Generator eingefügt.

Aufzählungswerte:

Epsilon

Nonterminal

Terminal

```
00143 {
00144   Epsilon, Nonterminal, Terminal
00145 };
```

5.1.4 Dokumentation der Funktionen

5.1.4.1 static int call_lexer (void) [static]

Kapselung des yylex-Aufrufs. Ruft `yylex` auf, sofern das letzte gelesene Token > 0 war, sonst wird direkt 0 zurückgegeben.

```
02575 {
02576   int result = 0;
02577   if (eof_reached == 0) {
02578     result = yylex();
02579     if (result == 0) {
02580       eof_reached = 1;
02581     }
02582   }
02583
02584   return result;
02585 } // int call_lexer
```

5.1.4.2 static void clean (void) [static]

Speicherbereinigung am Ende des Parsevorgangs. Die Funktion `clean` wird am Ende der Funktion `yyparse` aufgerufen, um den Speicher wieder freizugeben, der durch den Graphen und die verschiedenen anderen Knotenmengen belegt wird.

```
02447 {
02448   // Zeiger auf den aktuell behandelten Knoten:
02449   struct node_t *node = NULL;
02450   // Der aktuell betrachtete Nachfolger:
02451   struct node_t *succ_node = NULL;
02452   // Der aktuell betrachtete Vorgänger:
```

```
02453   struct node_t *pred_node = NULL;
02454   // Menge der entfernbaren Nachfolgerknoten:
02455   struct node_table_t *removable_nodes = node_table_init();
02456
02457   DEBUG_PRINTF("\nStarte die abschliessende Speicherbereinigung!\n");
02458
02459   /* Zuerst werden alle Knotenmengen geleert, die ausser dem Graphen
02460    * verwaltet werden. Da hier nur Knoten eingetragen sein sollen, die sich
02461    * auch tatsächlich noch im Graphen befinden, werden hier _nur_ die
02462    * Tabelleneinträge entfernt, nicht aber die tatsächlichen Knoten!
02463    */
02464   node_table_free(expandable_nodes);
02465   expandable_nodes = NULL;
02466   node_table_free(expanded_nodes);
02467   expanded_nodes = NULL;
02468   node_table_free(expanded_nonterminals);
02469   expanded_nonterminals = NULL;
02470   node_table_free(readable_nodes);
02471   readable_nodes = NULL;
02472   node_table_free(reducible_nodes);
02473   reducible_nodes = NULL;
02474
02475   /* Anschliessend wird der Graph durchmustert, um alle darin befindlichen
02476    * Knoten zu löschen. Dazu wird als erstes für alle Startknoten, d.h.
02477    * für jeden Knoten node in der Liste graph der Anfangsknoten, die Menge
02478    * seiner Nachfolger betrachtet.
02479    * Für jeden Knoten succ_node in der Nachfolgermenge wird node aus
02480    * succ_node->predecessors gelöscht. Anschliessend wird succ_node in
02481    * die Menge der löschbaren Knoten aufgenommen.
02482    */
02483   while (graph != NULL) {
02484     node = list_pop(&graph);
02485
02486     // Behandlung der Nachfolgerknoten
02487     while (node_table_empty(node->successors) == 0) {
02488       succ_node = node_table_pop(node->successors);
02489
02490       // Entferne node aus der Vorgängermenge von succ_node
02491       node_table_remove(succ_node->predecessors, node);
02492
02493       /* Entferne die Kanten zwischen succ_node und all seinen direkten
02494        * Vorgängern, d.h. Entferne succ_node aus den Nachfolgermengen
02495        * all seiner Vorgänger und entferne alle Vorgänger aus der
02496        * Vorgängermenge von succ_node:
02497        */
02498       while (node_table_empty(succ_node->predecessors) == 0) {
02499         pred_node = node_table_pop(succ_node->predecessors);
02500         node_table_remove(pred_node->successors, succ_node);
02501       }
02502
02503       // Trage succ_node bei den löschbaren Knoten ein:
02504       node_table_add(removable_nodes, succ_node);
02505     }
02506     node_table_free(node->successors);
02507
02508     /* Nachdem er von allen Nachfolgern gekappt wurde, kann der Knoten
02509      * node gelöscht werden:
02510      */
02511     free(node);
02512   } // while (graph != NULL)
02513
02514   /* Nun werden analog alle Knoten in der Tabelle deleteable_nodes
02515    * behandelt. Auf Vorgänger im Graphen muss dabei keine Rücksicht
02516    * genommen werden, weil beim Eintragen eines Knotens in die Menge
02517    * der Löschkandidaten alle Verbindungen zu Vorgängern entfernt wurden.
02518    */
02519   while (node_table_empty(removable_nodes) == 0) {
```

```
02520     node = node_table_pop(removable_nodes);
02521
02522     // Behandlung der Nachfolgerknoten:
02523     while (node_table_empty(node->successors) == 0) {
02524       succ_node = node_table_pop(node->successors);
02525
02526       // Entferne node aus der Vorgängerliste von succ_node:
02527       node_table_remove(succ_node->predecessors, node);
02528
02529       /* Entferne die Kanten zwischen succ_node und all seinen direkten
02530        * Vorgängern, d.h. Entferne succ_node aus den Nachfolgermengen
02531        * all seiner Vorgänger und entferne alle Vorgänger aus der
02532        * Vorgängermenge von succ_node:
02533        */
02534       while (node_table_empty(succ_node->predecessors) == 0) {
02535         pred_node = node_table_pop(succ_node->predecessors);
02536         node_table_remove(pred_node->successors, succ_node);
02537       }
02538
02539       // Trage succ_node bei den löschbaren Knoten ein:
02540       node_table_add(removable_nodes, succ_node);
02541     }
02542
02543     /* Nachdem alle Nachfolger behandelt sind, kann der
02544      * Knoten node gelöscht werden:
02545      */
02546     node_table_free(node->predecessors);
02547     node_table_free(node->successors);
02548     free(node);
02549
02550   } // end while (node_table_empty(removable_nodes) == 0)
02551
02552 #ifndef DEBUG_ALL
02553   /* Nun muss noch der Pool für Listenelemente bereinigt werden: */
02554   struct node_list_t *pool_tmp = NULL;
02555   while (pool != NULL) {
02556     pool_tmp = pool;
02557     pool = pool_tmp->right;
02558     free(pool_tmp);
02559   } // while (pool != NULL)
02560 #endif
02561
02562   DEBUG_PRINTF("Speicherbereinigung abgeschlossen!\n");
02563
02564 } // void clean
```

5.1.4.3 static unsigned int compute_hash (struct node_t * *node*, size_t *length*) [inline, static]

Berechnen des Hashwertes für einen Knoten.

Parameter:

node Zeiger auf den Knoten, dessen Hashwert bestimmt werden soll

length Länge der Tabelle, in der der Knoten eingetragen ist/werden soll. Wird benutzt als Modulo-Faktor für den Hashwert.

Rückgabe:

Hashwert als unsigned int

```
00955 {
00956   return ( ((uintptr_t)node >> 4) % length);
00957 } // static unsigned int compute_hash
```

5.1.4.4 static struct node_t* create_node (signed long *content*) [static, read]

Erzeugen eines neuen Knotens.

Parameter:

content Die Nummer des Items oder die (negierte) Nummer des Nichtterminalsymbols, für das der Knoten steht.

Rückgabe:

Einen Zeiger auf einen neuen Knoten vom Typ `struct node_t`, dessen weitere Felder mit `NULL` bzw. `0` initialisiert wurden.

```
00494 {
00495   struct node_t *node = malloc(sizeof(struct node_t));
00496   node->content = content;
00497   node->predecessors = node_table_init();
00498   node->successors = node_table_init();
00499   node->last_visited = 0;
00500   memset(node->endnodes, 0, LOOKAHEAD_LENGTH * sizeof(struct node_t*));
00501   node->prefixes = 0;
00502
00503   return node;
00504 } // struct node_t *create_node
```

5.1.4.5 static void End (void) [static]

Sauberer Abschluss der Ableitungsbaum-Ausgabe. Falls das Makro `PARSE_TREE` gesetzt ist, erzeugt der Parser bei erfolgreichem Parsevorgang einen Ableitungsbaum für die Eingabe im `dot`− Format. Falls `PARSE_TREE_PHASES` gesetzt ist, wird sogar für jede Phase der aktuelle Wald auf partiellen Ableitungsbäumen ausgegeben. Damit die Ausgabe auf jeden Fall eine gültige Eingabe für `dot` darstellt, muss sie mit einer geschweiften Klammer abgeschlossen werden.

```
02420 {
02421   printf("}");
02422 }
```

5.1.4.6 static void expand (void) [static]

Durchführung eines Expansionsschrittes. Die Funktion `expand` führt für alle expandierbaren Endknoten des aktuellen Graphen den Expansionsschritt durch. Sollte in der aktuellen Phase bereits eine Expansion für eines der hier betrachteten Nichtterminalsymbole stattgefunden haben, so wird nur eine Kante vom expandierbaren Knoten zum Knoten für dieses Nichtterminalsymbol eingefügt. Ansonsten wird solch ein Nichtterminalknoten neu erzeugt, und es werden Items für alle Alternativen hinzugefügt, die zum aktuellen Zeitpunkt zu einer akzeptierenden Berechnung führen könnten.

```
02746 {
02747   // Universelle Laufvariable:
02748   int i = 0;
02749   // Flag, ob die aktuelle Expansion gelungen ist:
02750   int expand_ok = 0;
02751   // Der jeweils aktuelle zu expandierende Endknoten:
02752   struct node_t *node = NULL;
02753   // Der jeweils aktuelle Nichtterminalknoten:
02754   struct node_t *current_nt_node = NULL;
```

```
02755   // Der aktuell neu erzeugte Knoten:
02756   struct node_t *new_node = NULL;
02757   // Inhalt des Arrayeintrages für das aktuelle Nichtterminalsymbol:
02758   struct nonterminal_t nt;
02759
02760   if (node_table_empty(expandable_nodes) != 0) {
02761     DEBUG_PRINTF("  Keine expandierbaren Knoten gefunden!\n");
02762     return;
02763   }
02764
02765   while (node_table_empty(expandable_nodes) == 0) {
02766
02767     /* Entnimm einen Knoten aus der Menge der expandierbaren Knoten und
02768      * füge ihn in die Menge der Knoten ein, die in der aktuellen Phase
02769      * expandiert wurden:
02770      */
02771     DEBUG_PRINTF("  Menge expandierbarer Knoten:\n");
02772     DEBUG_PRINTF("  ");
02773     NODE_TABLE_PRINT(expandable_nodes);
02774
02775     node = node_table_pop(expandable_nodes);
02776
02777     DEBUG_PRINTF("  ");
02778     NODE_TABLE_PRINT(expandable_nodes);
02779     DEBUG_PRINTF("  Liste expandierter Knoten:\n");
02780     DEBUG_PRINTF("  ");
02781     NODE_TABLE_PRINT(expanded_nodes);
02782
02783     node_table_add(expanded_nodes, node);
02784
02785     DEBUG_PRINTF("  ");
02786     NODE_TABLE_PRINT(expanded_nodes);
02787     DEBUG_PRINTF("  Expandiere Knoten: %ld\n", node->content);
02788
02789     expand_ok = 0;
02790
02791     /* Sei [A -> alpha1 . B alpha2] das aktuell betrachtete expandierbare
02792      * Item. Falls eine Expansion anhand B in dieser Phase schon
02793      * durchgeführt wurde, müssen keine neuen Knoten in den Graphen
02794      * eingefügt werden, sondern nur eine Kante vom aktuellen Item zum
02795      * Nichtterminalknoten für B.
02796      */
02797     unsigned long current_nonterminal = items[node->content].dot_symbol;
02798     current_nt_node =
02799       node_table_find(expanded_nonterminals, ~current_nonterminal);
02800     if (current_nt_node != NULL) {
02801       DEBUG_PRINTF("  Anhand des NT-Symbols %ld wurde schon expandiert!\n",
02802           current_nonterminal);
02803       expand_ok = 1;
02804     }
02805
02806     /* Falls die Expansion jedoch zum ersten Mal für diese Phase
02807      * stattfindet, müssen ein Nichtterminalknoten für B und einige
02808      * Itemknoten für seine Alternativen [B -> beta_j] in den Graphen
02809      * eingefügt werden.
02810      */
02811     else {
02812       /* Erzeuge einen Nichtterminalknoten für current_nonterminal,
02813        * verbinde ihn aber noch nicht mit anderen Knoten im Graphen.
02814        */
02815       DEBUG_PRINTF("    Erzeuge Nichtterminalknoten für Symbol %ld\n",
02816           current_nonterminal);
02817       current_nt_node = create_node(~current_nonterminal);
02818
02819       /* Nun müssen alle Alternativen für current_nonterminal untersucht
02820        * und ggf. Knoten dafür in den Graphen eingefügt werden.
02821        */
```

```
02822       nt = nonterminals[current_nonterminal-1];
02823       DEBUG_PRINTF("    Starte Test der %d Alternativen\n", nt.item_count);
02824       for (i=0; i<nt.item_count; i++) {
02825
02826         DEBUG_PRINTF("      Prüfe Alternative %d/%d: Item Nr. %d => ",
02827           i+1, nt.item_count, nt.items[i]);
02828
02829         // Falls ein reduzierbarer Knoten entsteht:
02830         if (items[nt.items[i]].dot_symbol_type == Epsilon) {
02831           DEBUG_PRINTF("neuer reduzierbarer Knoten\n");
02832
02833           /* Der neue Knoten wird erzeugt und mit dem Nichtterminalknoten
02834            * verbunden.
02835            */
02836           new_node = create_node(nt.items[i]);
02837           node_table_add(new_node->predecessors, current_nt_node);
02838           node_table_add(current_nt_node->successors, new_node);
02839
02840           /* Im nächsten Reduce-Read-Schritt muss der neue Knoten
02841            * als reduzierbar berücksichtigt werden:
02842            */
02843           node_table_add(reducible_nodes, new_node);
02844
02845           expand_ok = 1;
02846         } // if (Epsilon)
02847
02848         // Falls ein lesbarer Endknoten entsteht:
02849         else if ( items[nt.items[i]].dot_symbol_type == Terminal ) {
02850
02851           // Entspricht das dot_symbol dem nächsten Symbol der Eingabe?
02852           if ( items[nt.items[i]].dot_symbol
02853                == lookahead[phase_number % LOOKAHEAD_LENGTH] ) {
02854             DEBUG_PRINTF("neuer lesbarer Knoten (%ld)\n",
02855                 items[nt.items[i]].dot_symbol);
02856
02857             /* Der neue Knoten wird in den Graphen eingefügt und mit dem
02858              * Nichtterminalknoten verbunden:
02859              */
02860             new_node = create_node(nt.items[i]);
02861             node_table_add(new_node->predecessors, current_nt_node);
02862             node_table_add(current_nt_node->successors, new_node);
02863
02864             /* Im nächsten Reduce-Read-Schritt muss der neue Knoten
02865              * als lesbar berücksichtigt werden:
02866              */
02867             node_table_add(readable_nodes, new_node);
02868
02869             expand_ok = 1;
02870
02871           } // if (passendes dot_symbol)
02872
02873           /* Falls das Terminalsymbol nicht dem nächsten Eingabesymbol
02874            * entspricht, kann eine Expansion dieser Alternative nicht zu
02875            * einer akzeptierenden Berechnung führen.
02876            */
02877           else {
02878             DEBUG_PRINTF("UNPASSENDES dot_symbol (%ld)\n",
02879                 items[nt.items[i]].dot_symbol);
02880           } // if (passendes dot_symbol) {...} else
02881
02882         } // elseif (Terminal)
02883
02884         // Falls ein expandierbarer Endknoten ensteht:
02885         else if (items[nt.items[i]].dot_symbol_type == Nonterminal) {
02886           DEBUG_PRINTF("neuer expandierbarer Knoten (%ld)\n",
02887               items[nt.items[i]].dot_symbol);
02888
```

```
02889           /* Falls ein entsprechender Knoten schon in dieser Phase
02890            * expandiert wurde, muss er nur noch mit seinem neuen
02891            * zusätzlichen Vorgängerknoten B_i verbunden werden:
02892            */
02893           new_node = node_table_find(expanded_nodes, nt.items[i]);
02894           if (new_node != NULL) {
02895             DEBUG_PRINTF("      ... existiert schon (schon expandiert)\n");
02896             node_table_add(new_node->predecessors, current_nt_node);
02897             node_table_add(current_nt_node->successors, new_node);
02898           }
02899
02900           /* Ansonsten wurde der Knoten entweder zwar schon erschaffen,
02901            * aber noch nicht expandiert, oder aber der Knoten existiert
02902            * noch nicht.
02903            */
02904           else {
02905             /* Falls der Knoten in dieser Phase erschaffen, aber noch
02906              * nicht expandiert wurde, so wurde er aber schon zur
02907              * Expansion vorgemerkt. Auch dann muss er nur noch
02908              * mit seinem neuen zusätzlichen Vorgänger verbunden werden:
02909              */
02910             new_node = node_table_find(expandable_nodes, nt.items[i]);
02911             if (new_node != NULL) {
02912               DEBUG_PRINTF
02913                 ("      ... existiert schon (noch nicht expandiert)\n");
02914               node_table_add(new_node->predecessors, current_nt_node);
02915               node_table_add(current_nt_node->successors, new_node);
02916             }
02917
02918             /* Ansonsten wurde solch ein Knoten in der aktuellen Phase noch
02919              * nicht erzeugt. Dies wird nun erledigt, und der Knoten wird
02920              * als expandierbar vorgemerkt.
02921              */
02922             else {
02923               new_node = create_node(nt.items[i]);
02924               node_table_add(new_node->predecessors, current_nt_node);
02925               node_table_add(current_nt_node->successors, new_node);
02926
02927               node_table_add(expandable_nodes, new_node);
02928             }
02929           } // if (new_node != NULL) {...} else
02930
02931           expand_ok = 1;
02932
02933         } // else if (Nonterminal)
02934
02935         /* Wenn keiner der vorigen Fälle eintritt, liegt ein Programmier-
02936          * fehler vor, da es für einen gültigen Knoten nur die drei
02937          * Möglichkeiten Epsilon, Terminal und Nonterminal als
02938          * dot_symbol_type gibt!
02939          */
02940         else {
02941           DEBUG_PRINTF
02942             ("FEHLER beim Expansionsschritt: Ungültiger Knoten!\n");
02943           exit(EXIT_FAILURE);
02944         } // else
02945
02946       } // for (i=0; i<nt.item_count; i++)
02947
02948     } // if (current_nt_node != NULL) ... else
02949
02950     /* Falls für mindestens eine Alternative des aktuellen
02951      * Nichtterminalsymbols ein Knoten in den Graphen eingefügt wurde,
02952      * war die Expansion erfolgreich. Der Nichtterminalknoten wird durch
02953      * eine Kante mit dem soeben fertig expandierten Knoten verbunden.
02954      */
02955     if (expand_ok) {
```

```
02956       DEBUG_PRINTF("  Expansion war erfolgreich!\n");
02957       node_table_add(expanded_nonterminals, current_nt_node);
02958       node_table_add(node->successors, current_nt_node);
02959       node_table_add(current_nt_node->predecessors, node);
02960     }
02961
02962     /* Falls für den aktuellen expandierbaren Endknoten keine Alternative
02963      * gefunden wurde, die zu einer akzeptierenden Berechnung führen kann,
02964      * müssen der neue Nichtterminalknoten current_nt_node und der
02965      * expandierbare Endknoten node aus dem Graphen entfernt werden, ebenso
02966      * wie alle Vorgänger, die keine anderen Nachfolger haben.
02967      */
02968     else {
02969       /* Der Nichtterminalknoten muss wieder aus dem Graph entfernt werden.
02970        * Dazu kann sein Speicherbereich einfach freigegeben werden, da er
02971        * noch mit keinem anderen Knoten im Graphen verbunden wurde.
02972        */
02973       DEBUG_PRINTF("    Entferne irrelevanten Nichtterminalknoten ... ");
02974       node_table_free(current_nt_node->predecessors);
02975       node_table_free(current_nt_node->successors);
02976       free(current_nt_node);
02977       current_nt_node = NULL;
02978       DEBUG_PRINTF("erledigt!\n");
02979
02980       /* Der Knoten node war zu Beginn des Expansionsversuchs in die Liste
02981        * der expandierten Endknoten eingetragen worden. Dies muss nun
02982        * zurückgenommen werden.
02983        */
02984       node_table_remove(expanded_nodes, node);
02985
02986       /* Nun können node und evtl. einige seiner Vorgängerknoten aus dem
02987        * Graphen entfernt werden:
02988        */
02989       DEBUG_PRINTF("    Entferne Expansionskandidaten %ld ... ",
02990           node->content);
02991       graph_adjust(&node);
02992       DEBUG_PRINTF("erledigt!\n");
02993
02994     } // if (expand_ok) else
02995
02996   } // while (node_table_empty(expandable_nodes) == 0)
02997
02998 } // void expand
```

5.1.4.7 static void graph_adjust (struct node_t ** *node*) [static]

Die Graphjustierung entfernt einen Knoten samt Vorgängern. Die Funktion `graph_adjust` entfernt einen Endknoten aus dem Graphen sowie alle seine Vorgänger, die keine sonstigen Nachfolger haben.

Parameter:

node Zeiger auf einen Zeiger (Call by Reference) auf den zu entfernenden Knoten.

```
02600 {
02601   assert((*node) != NULL);
02602
02603   /* Der Endknoten *node hat zwar keine Nachfolger, wohl aber in der
02604    * Regel Vorgänger, die noch auf Löschbarkeit überprüft werden
02605    * müssen.
02606    * Vorgängerknoten, die als löschbar identifiziert werden, werden in
02607    * einer Tabelle gesammelt:
02608    */
02609   struct node_table_t *preds = node_table_init();
```

```
02610   // Zeiger auf den aktuell untersuchten Vorgängerknoten:
02611   struct node_t *pred_node = NULL;
02612   // Zeiger auf den aktuell zu löschenden Knoten:
02613   struct node_t *kill_node = NULL;
02614   // Flag für Erfolg oder Misserfolg einer Knotenlöschung:
02615   int remove_success = 0;
02616
02617   /* Für alle Vorgänger von *node: Entnimm sie aus der predecessor-
02618    * Menge von node und entferne node aus ihren successor-Mengen.
02619    */
02620   while (node_table_empty((*node)->predecessors) == 0) {
02621     /* Nimm den nächsten Vorgängerknoten von *node: */
02622     pred_node = node_table_pop((*node)->predecessors);
02623
02624     /* Entferne *node aus der successor-Menge von pred_node: */
02625     node_table_remove(pred_node->successors, *node);
02626
02627     /* Falls pred_node keine anderen Nachfolger als *node im Graph hatte,
02628      * gehört er selbst zur Menge der entfernbaren Knoten.
02629      */
02630     if (node_table_empty(pred_node->successors) != 0) {
02631       node_table_add(preds, pred_node);
02632     }
02633   }
02634
02635   /* Bevor der Knoten endgültig entfernt werden kann, muss er ggf. noch aus
02636    * einigen Knotentabellen entfernt werden.
02637    * Die Funktion graph_adjust wird nur mit Knoten aufgerufen, die nicht
02638    * für das Startitem [S' -> .S] oder das Accept-Item [S' -> S.] stehen.
02639    * Jeder andere Itemknoten kann in einer der Tabellen für expandierbare,
02640    * expandierte, lesbare oder reduzierbare Knoten enthalten sein:
02641    */
02642   if ( (*node)->content >= 0) {
02643     node_table_remove(expandable_nodes, *node);
02644     node_table_remove(expanded_nodes, *node);
02645     node_table_remove(reducible_nodes, *node);
02646     node_table_remove(readable_nodes, *node);
02647   }
02648   /* Ein Nichtterminalknoten kann sich höchstens in der Liste der
02649    * expandierten Nichtterminalknoten befinden:
02650    */
02651   else {
02652     node_table_remove(expanded_nonterminals, *node);
02653   }
02654
02655   /* Nachdem alle Verweise von und auf *node entfernt sind, kann node
02656    * selbst aus dem Speicher entfernt werden:
02657    */
02658   node_table_free((*node)->predecessors);
02659   node_table_free((*node)->successors);
02660   free(*node);
02661   *node = NULL;
02662
02663
02664   /* Nun müssen noch die Vorgänger analog bearbeitet werden. */
02665   while (node_table_empty(preds) == 0) {
02666
02667     // Entnimmt einen zu löschenden Knoten aus der Menge:
02668     kill_node = node_table_pop(preds);
02669
02670     /* Für alle Vorgänger von kill_node: Entnimm sie aus der
02671      * predecessor-Menge von kill_node und entferne kill_node aus
02672      * ihren successor-Mengen.
02673      */
02674     while (node_table_empty(kill_node->predecessors) == 0) {
02675       /* Nimm den nächsten Vorgängerknoten von kill_node: */
02676       pred_node = node_table_pop(kill_node->predecessors);
```

```
02677
02678       /* Entferne den Eintrag für kill_node aus der successor-
02679        * Menge von temp_node:
02680        */
02681       node_table_remove(pred_node->successors, kill_node);
02682
02683       /* Falls pred_node keine anderen Nachfolger als kill_node im Graph
02684        * hatte, gehört er jetzt selbst zur Menge der entfernbaren Knoten.
02685        */
02686       if (node_table_empty(pred_node->successors) != 0)
02687         node_table_add(preds, pred_node);
02688     } // while (node_table_empty(kill_node->predecessors) == 0)
02689
02690     /* Bevor der Knoten endgültig entfernt werden kann, muss es ggf. noch
02691      * aus einigen Tabellen entfernt werden.
02692      * Ein Itemknoten kann nicht für das Accept-Item [S' -> S.] stehen,
02693      * da dieses keine Nachfolger hätte. Steht kill_node für das Startitem
02694      * [S' -> .S], so kann der Graph ansonsten nur noch aus Knoten für das
02695      * Accept-Item bestehen.
02696      */
02697     if ((kill_node)->content == startitem_number)
02698       remove_success = list_remove(&graph, kill_node);
02699
02700     /* Jeder andere Itemknoten kann in einer der Tabellen für
02701      * expandierbare, expandierte, lesbare oder reduzierbare Knoten
02702      * enthalten sein.
02703      */
02704     else if ((kill_node)->content >= 0) {
02705       node_table_remove(expandable_nodes, kill_node);
02706       node_table_remove(expanded_nodes, kill_node);
02707       node_table_remove(reducible_nodes, kill_node);
02708       node_table_remove(readable_nodes, kill_node);
02709     }
02710
02711     /* Ein Nichtterminalknoten kann sich höchstens in der Menge der
02712      * expandierten Nichtterminalknoten befinden:
02713      */
02714     else {
02715       node_table_remove(expanded_nonterminals, kill_node);
02716     }
02717
02718     /* Nachdem alle Verweise von und auf kill_node entfernt sind, kann
02719      * der Knoten selbst nun aus dem Speicher entfernt werden:
02720      */
02721     node_table_free(kill_node->predecessors);
02722     node_table_free(kill_node->successors);
02723     free(kill_node);
02724     kill_node = NULL;
02725
02726   } // while (NULL != preds)
02727
02728 } // void graph_adjust
```

5.1.4.8 static void list_add (struct node_list_t ∗∗ *list*, struct node_t ∗ *node*) [static]

Anhängen eines neuen Elementes an den Anfang einer Knotenliste. Dabei wird nicht darauf geachtet, dass die Liste duplikatfrei bleibt. Sollte der Zeiger ∗list, über den die Liste betreten wird, aus der Mitte einer größeren Liste stammen, wird das neue Element auch mit dem bisherigen linken Nachbarn von ∗list verbunden.

Parameter:

list Zeiger auf einen Zeiger auf die Liste, in die der neue Knoten eingefügt werden soll (Call by Reference).

node Zeiger auf den Graphknoten, der neu in die Liste eingefügt werden soll.

```
00630 {
00631   /* Falls node auf keinen Knoten zeigt, liegt ein Fehler vor: */
00632   assert(node != NULL);
00633
00634   struct node_list_t *lst_new = pool_get();
00635
00636   lst_new->head = node;
00637
00638   /* Verbinde das neue Listenelement mit seinen neuen Nachbarn: */
00639   if (*list != NULL) {
00640     lst_new->left = (*list)->left;
00641     (*list)->left = lst_new;
00642     if (lst_new->left != NULL)
00643       lst_new->left->right = lst_new;
00644   }
00645   else
00646     lst_new->left = NULL;
00647
00648   lst_new->right = *list;
00649
00650   *list = lst_new;
00651 } // void list_add
```

5.1.4.9 static void list_clear (struct node_list_t ** *list*) [static]

Vollständiges Leeren einer Knotenliste. Der Aufruf `list_clear (list)` entfernt alle Listeneinträge, so dass `*list` anschließend ein Nullzeiger ist.

Parameter:

list Zeiger auf einen Zeiger auf die zu leerende Knotenliste (Call by Reference).

```
00920 {
00921   /* Falls die Liste schon leer ist, gibt es nichts zu tun: */
00922   if (*list == NULL)
00923     return;
00924
00925   /* Falls das erste Element der zu löschenden Liste aus der Mitte einer
00926    * größeren Liste stammt, löse erst diese Verbindung:
00927    */
00928   if ((*list)->left != NULL) {
00929     (*list)->left->right = NULL;
00930     (*list)->left = NULL;
00931   }
00932
00933   /* Nun gib die Liste an den Pool zurück: */
00934   pool_release_list(*list);
00935   *list = NULL;
00936 } // void list_clear
```

5.1.4.10 static int list_contains_node_num (struct node_list_t * *list*, signed long *node_number*) [static]

Überprüft, ob in einer Knotenliste ein Knoten mit einer bestimmten Nummer enthalten ist. Der Aufruf `list_contains_node_num (lst, node_number)` prüft, ob sich in der Liste, auf die *lst* zeigt, ein Eintrag für den Knoten mit Index *node_number* befindet.

Parameter:

list Zeiger auf die zu durchsuchende Knotenliste.

node_number Index des Knotens, der gesucht wird. Wenn *node_number* eine positive Zahl ist, wird ein Itemknoten gesucht, sonst ein Nichtterminalknoten.

Rückgabe:

`1`, falls der gesuchte Knoten sich in *lst* befindet; `0`, sonst.

```
00744 {
00745   while (list != NULL) {
00746     if (list->head->content == node_number)
00747       return 1;
00748     else
00749       list = list->right;
00750   }
00751
00752   /* An diesen Punkt gelangt die Funktion nur, wenn die Liste erfolglos
00753    * durchsucht wurde:
00754    */
00755   return 0;
00756 } // int list_contains_node_num
```

5.1.4.11 static int list_contains_sublist (struct node_list_t * *lst*, struct node_list_t * *sublst*) [static]

Test, ob eine Liste in einer anderen enthalten ist. Bei einem Aufruf `list_contains_sublist(lst, sublst)` wird geprüft, ob die Liste `sublst` Teil von `lst` ist.

Diese Funktion wird nur compiliert, wenn das Makro `DEBUG_ALL` definiert ist und dient als Hilfsfunktion für `node_table_check`.

Parameter:

lst Liste, in der nach *sublst* gesucht werden soll.

sublst Liste, auf deren Enthaltensein in `lst` getestet wird.

Rückgabe:

0, falls *sublst* in *lst* enthalten ist; 1, sonst,

```
00977 {
00978   /* Wenn schon das erste Element von lst identisch mit sublst ist,
00979    * sind wird schon fertig:
00980    */
00981   if (lst == sublst)
00982     return 0;
00983
00984   /* Ansonsten gehe die Liste lst durch und teste für jedes Element, ob
00985    * sein Nachfolger NULL oder sublst entspricht.
00986    */
00987   while (lst != NULL) {
00988     if (lst->right == sublst)
00989       return 0;
00990
00991     lst = lst->right;
00992   } // while (lst != NULL)
00993
00994   /* Wenn die Funktionsausführung an diese Stelle gelangt, wurde lst
```

```
00995    * komplett durchsucht, ohne sublst zu finden.
00996    */
00997   return 1;
00998 } // static int list_contains_sublist
```

5.1.4.12 static int list_enqueue (struct node_list_t ** *list*, struct node_t * *node*) [static]

Duplikatfreies Anhängen eines neuen Elementes an das Ende einer Knotenliste. Sollte sich das Element bereits in der Liste befinden, wird nichts unternommen.

Parameter:

list Zeiger auf einen Zeiger auf die Liste, in die der neue Knoten eingefügt werden soll (Call by Reference).

node Zeiger auf den Graphknoten, der neu in die Liste eingefügt werden soll.

Rückgabe:

`0`, falls der Knoten eingefügt werden konnte, `1`, falls sich `node` schon vorher in der Liste befindet.

```
00670 {
00671   /* Falls node auf keinen Knoten zeigt, liegt ein Fehler vor: */
00672   assert(node != NULL);
00673
00674   /* Falls die Liste bisher leer ist, bildet das neue Element den
00675    * Listenanfang.
00676    */
00677   if (*list == NULL) {
00678     struct node_list_t *lst_new = pool_get();
00679     lst_new->head = node;
00680     lst_new->left = NULL;
00681     lst_new->right = NULL;
00682     *list = lst_new;
00683     return 0;
00684   }
00685
00686   /* Falls die Liste nicht leer ist: Suche das letzte Element und füge
00687    * das neue dahinter an.
00688    */
00689   else {
00690
00691     // Zeiger auf das aktuell betrachtete Listenelement:
00692     struct node_list_t *lst_tmp = *list;
00693
00694     /* Sollte sich das neue Element bereits am Listenanfang befinden, wird
00695      * die Funktion ohne Aktion abgebrochen.
00696      */
00697     if (lst_tmp->head == node)
00698       return 1;
00699
00700     while (lst_tmp->right != NULL) {
00701       lst_tmp = lst_tmp->right;
00702
00703       /* Falls das aktuelle Listenelement gleich dem neu einzufügenden
00704        * ist, endet die Funktion ohne Aktion.
00705        */
00706       if (lst_tmp->head == node)
00707         return 1;
00708     }
00709
00710     /* Wenn die Ausführung der Funktion an diese Stelle gelangt, ist das
00711      * Ende der Liste gefunden (lst_tmp zeigt auf das letzte Listenelement,
```

```
00712       * lst_tmp->right ist NULL), und das neue Element ist noch nicht in der
00713       * Liste enthalten.
00714       */
00715      struct node_list_t *lst_new = pool_get();
00716      lst_new->head = node;
00717      lst_new->left = lst_tmp;
00718      lst_new->right = NULL;
00719      lst_tmp->right = lst_new;
00720      return 0;
00721
00722    } // if (*list == NULL) ... else
00723
00724 } // void list_enqueue
```

5.1.4.13 static struct node_t* list_pop (struct node_list_t ** *list*) [static, read]

Entnimmt den ersten Knoten einer Liste. Die Funktion `list_pop` liefert einen Zeiger auf den ersten Knoten einer Knotenliste und entfernt seinen Eintrag aus dieser Liste.

Parameter:

list Zeiger auf einen Zeiger auf die behandelte Liste. (Call by Reference).

Rückgabe:

Zeiger auf das erste Listenelement, falls es existiert; Sonst: `NULL`.

```
00772 {
00773   if (*list == NULL)
00774     return NULL;
00775   else {
00776     struct node_list_t *old_first = *list;
00777     struct node_t *node = old_first->head;
00778     *list = old_first->right;
00779     /* Falls sich das Listenelement, über den die Liste betreten wird, in
00780      * ihrer Mitte befindet, muss auch der left-Zeiger angepasst werden!
00781      */
00782     if ((*list) != NULL)
00783       (*list)->left = old_first->left;
00784
00785     pool_release(old_first);
00786
00787     return node;
00788   }
00789 } // struct node_t *list_pop
```

5.1.4.14 static void list_print (struct node_list_t * *list*) [static]

Ausgabe einer Knotenliste auf der Standardausgabe. Der Aufruf `list_print (list)` gibt zu Diagnosezwecken eine einfache textuelle Darstellung der Knotenliste *list* aus.

Die Funktion wird hinter dem Makro `LIST_PRINT` versteckt und nur ausgeführt, falls das Makro `USE_DEBUG_PRINTF` definiert ist. Ansonsten wird `LIST_PRINT` zu `(void) 0` expandiert.

Parameter:

list Zeiger auf die Knotenliste, die ausgegeben werden soll.

```
00887 {
00888   DEBUG_PRINTF("[");
00889   while (list != NULL) {
00890     if (list->head->content >= 0)
00891       DEBUG_PRINTF(" %ld", list->head->content);
00892     else
00893       DEBUG_PRINTF(" n%ld", ~(list->head->content));
00894     list = list->right;
00895   }
00896   DEBUG_PRINTF("]\n");
00897 } // void list_print
```

5.1.4.15 static int list_remove (struct node_list_t ** *list*, struct node_t * *node*) [static]

Sucht einen Knoten anhand seines Zeigers und entfernt ihn aus einer Knotenliste. Die Funktion `list_remove` erhält als Argumente die Adresse eines Zeigers auf eine Knotenliste und einen Zeiger auf einen Knoten. Wenn sich der Knoten in der Liste befindet, wird sein erstes Vorkommen aus der Liste entfernt.

Parameter:

list Zeiger auf einen Zeiger auf die Knotenliste, aus der der Knoten erntfernt werden soll (Call by Reference).

node Zeiger auf den Knoten, der aus der Liste ausgetragen werden soll.

Rückgabe:

Integerwert für Erfolg oder Misserfolg der Aktion. Dabei wird `0` zurückgegeben, wenn der Knotenzeiger erfolgreich aus der Liste entfernt wurde, `1`, falls die Liste leer ist, und 2, falls der Knotenzeiger nicht enthalten ist.

```
00812 {
00813   /* Falls node auf keinen Knoten zeigt, liegt ein Fehler vor: */
00814   assert(node != NULL);
00815
00816   /* Falls die Liste leer ist, gibt es nichts zu entfernen: */
00817   if ( (*list) == NULL ) {
00818     return 1;
00819   }
00820
00821   // Zeiger auf das aktuell betrachtete Listenelement:
00822   struct node_list_t *tmp = *list;
00823
00824   /* Falls schon das erste Listenelement das zu Löschende ist, wird es
00825    * entfernt und der Listenanfang auf \c right umgebogen. Für den
00826    * Fall, dass der übergebene Listenzeiger aus der Mitte einer größeren
00827    * Liste stammt, wird auch der left-Zeiger angepasst.
00828    */
00829   if (tmp->head == node) {
00830     if (tmp->left != NULL)
00831       tmp->left->right = tmp->right;
00832     if (tmp->right != NULL)
00833       tmp->right->left = tmp->left;
00834
00835     *list = tmp->right;
00836     pool_release(tmp);
00837
00838     return 0;
00839   }
00840
00841   /* Falls es nicht das erste Element ist: Ueberprüfe iterativ den
00842    * Nachfolger des aktuellen Elementes:
```

```
00843    */
00844   // Zeiger auf den Nachfolger des aktuellen Listenelementes:
00845   struct node_list_t *tmp2 = NULL;
00846   while (tmp->right != NULL) {
00847     if (tmp->right->head == node) {
00848       /* Merke Dir die Restliste tmp->right als tmp2. Verbinde dann tmp
00849        * und tmp->right->right, so dass tmp2 "übergangen" wird.
00850        * Nun kann tmp2 entfernt werden.
00851        */
00852       tmp2 = tmp->right;
00853       tmp->right = tmp2->right;
00854       if (tmp->right != NULL)
00855         tmp->right->left = tmp;
00856
00857       pool_release(tmp2);
00858
00859       return 0;
00860     }
00861     else
00862       tmp = tmp->right;
00863   }
00864
00865   /* An diesen Punkt gelangt die Funktion nur, wenn die Liste vollständig
00866    * erfolglos durchsucht wurde:
00867    */
00868   return 2;
00869 } // int remove_node (...)
```

5.1.4.16 static void node_table_add (struct node_table_t * *tbl*, struct node_t * *node*) [static]

Hinzufügen eines neuen Knotens zu einer Hashtabelle. Ein Aufruf `node_table_add(tbl, node)` fügt den Knoten, auf den *node* zeigt, in die Hashtabelle ein, auf die *tbl* zeigt. Dabei wird davon ausgegangen, dass es sich um eine gültig vorinitialisierte Hashtabelle handelt, ansonsten wird das Programm abgebrochen. Sollte *node* sich bereits in der Tabelle befinden, wird die Funktion ohne Aktion beendet.

Parameter:

tbl Zeiger auf die Hashtabelle, der ein neuer Knoten hinzugefügt werden soll.

node Zeiger auf den Knoten, der neu in die Tabelle aufgenommen werden soll.

```
01298 {
01299   assert(node != NULL && tbl != NULL);
01300
01301   NODE_TABLE_CHECK(tbl, 0);
01302
01303   /* Falls die Tabelle bisher nur mit Nullwerten initialisiert, aber noch
01304    * nicht benutzt wurde, wird sie nun mit der Standardgröße HASHLENGTH
01305    * initialisiert:
01306    */
01307   if (tbl->length == 0) {
01308     /* Alle Einträge werden zunächst auf NULL gesetzt. Die Länge des Arrays
01309      * tbl->items wird durch das Makro HASHLENGTH festgelegt.
01310      */
01311     tbl->length = HASHLENGTH;
01312     tbl->item_count = 0;
01313     tbl->cursor = NULL;
01314     tbl->first = NULL;
01315     tbl->items =
01316       calloc(HASHLENGTH, sizeof(struct node_list_t*));
01317   } // if (tbl->length == 0);
01318
01319   unsigned int node_hash = compute_hash(node, tbl->length);
```

```
01320
01321   /* Versuche, den neuen Knoten in das Tabellenfeld mit der Nummer
01322    * node_hash einzutragen. Falls dort noch kein anderer Knoten mit dem
01323    * Hashwert node_hash eingetragen war oder unter den dort eingetragenen
01324    * Knoten der neue Knoten node noch nicht enthalten war, wird
01325    * add_success auf 1 gesetzt, sonst auf 0.
01326    */
01327   int add_success = 0;
01328
01329   /* Falls noch kein Knoten mit demselben Hashwert existiert, wird das
01330    * Feld tbl->items[node_hash] neu bestückt und der neue Eintrag auch mit
01331    * dem Zeiger first verbunden:
01332    */
01333   if (tbl->items[node_hash] == NULL) {
01334     struct node_list_t *lst_new = pool_get();
01335     lst_new->head = node;
01336     lst_new->left = NULL;
01337     lst_new->right = tbl->first;
01338
01339     if (tbl->first != NULL)
01340       tbl->first->left = lst_new;
01341
01342     tbl->first = lst_new;
01343     tbl->items[node_hash] = lst_new;
01344     add_success = 1;
01345   } // if (tbl->items[node_hash] == NULL)
01346
01347   /* Wenn es schon Knoten mit demselben Hashwert in tbl gibt, muss geprüft
01348    * werden, ob vielleicht schon node selbst eingetragen ist. Dann gibt es
01349    * nichts zu tun. Ansonsten muss ein neues Listenelement für node an
01350    * geeigneter Stelle eingefügt werden.
01351    */
01352   else {
01353     /* Ist es schon der erste Knoten? */
01354     if (tbl->items[node_hash]->head == node)
01355       add_success = 0;
01356
01357     /* Falls es nicht der erste Knoten ist, untersuche alle Nachfolger mit
01358      * demselben Hashwert.
01359      */
01360     else {
01361       struct node_list_t *lst_ptr = tbl->items[node_hash];
01362       int found = 0;
01363
01364       /* Solange es noch Nachfolgeknoten mit demselben Hashwert gibt, müssen
01365        * diese daraufhin untersucht werden, ob sie identisch mit node sind.
01366        * Falls einer gefunden wird, findet keine Einfügung statt,
01367        * falls nicht, wird der neue Knoten an passender Stelle eingefügt.
01368        */
01369       while ( (lst_ptr->right != NULL)
01370           && (compute_hash(lst_ptr->right->head, tbl->length) == node_hash)
01371           && (found != 1) )
01372       {
01373         /* Das Schleifeninnere wird also nur betreten, wenn der Knoten im
01374          * Listenfeld lst_ptr->right existiert und auch den Hashwert
01375          * node_hash hat. Falls dieser Knoten identisch mit node ist,
01376          * findet keine Einfügung statt:
01377          */
01378         if (lst_ptr->right->head == node)
01379           found = 1;
01380
01381         /* Ansonsten wird die Untersuchung beim Nachfolgerknoten wiederholt:
01382          */
01383         else
01384           lst_ptr = lst_ptr->right;
01385       } // while ( ... )
01386
```

```
01387       /* Falls nun found == 0, so verweist der Zeiger lst_ptr auf den
01388        * letzten Knoten in der Knotenliste von tbl, der den Hashwert
01389        * node_hash hat.
01390        * Der rechte Nachbar lst_ptr->right ist entweder NULL oder zeigt
01391        * auf einen Knoten mit anderem Hashwert. Es muss ein neues
01392        * Listenelement für node angelegt werden, das zwischen lst_ptr
01393        * und lst_ptr->right eingetragen wird.
01394        */
01395       if (found == 0) {
01396         struct node_list_t *lst_new = pool_get();
01397         lst_new->head = node;
01398         lst_new->left = lst_ptr;
01399         lst_new->right = lst_ptr->right;
01400
01401         lst_ptr->right = lst_new;
01402
01403         if (lst_new->right != NULL)
01404           lst_new->right->left = lst_new;
01405
01406         add_success = 1;
01407       } // if (found == 0)
01408
01409       /* Falls aber found == 1, so befindet sich ein Eintrag für node
01410        * bereits in tbl, und es findet keine Einfügung statt.
01411        */
01412       else {
01413         add_success = 0;
01414       }
01415
01416     } // if (tbl->items[node_hash]->head == node) ... else
01417
01418   } // if (tbl->items[node_hash] == NULL) ... else
01419
01420   /* Falls tatsächlich eine Einfügung stattgefunden hat, muss getestet
01421    * werden, ob die Tabelle vergrößert werden muss:
01422    */
01423   if (add_success == 1) {
01424     tbl->item_count++;
01425     NODE_TABLE_CHECK(tbl, 1);
01426     if (tbl->item_count >= tbl->length)
01427       node_table_double(tbl);
01428   } // if (add_success == 1)
01429
01430 } // static void node_table_add
```

5.1.4.17 static void node_table_check (struct node_table_t * *tbl*, int *flag*) [static]

Integritätstest für eine Knoten-Hashtafel. Für eine Hashtafel *tbl* wird getestet, ob die Einträge in `tbl->items` und in der Liste `tbl->first` zueinander passen. Dadurch fällt bspw. auf, falls in `tbl->items` Listenelemente eingtragen sind, die eigentlich nicht Teil der Tabelle sein dürfen.

Diese Funktion wird nur compiliert, falls das Makro `DEBUG_ALL` definiert ist.

Parameter:

tbl Zeiger auf die zu überprüfende Tabelle.

flag Zahl, die als Flag benutzt wird. `flag == 1` bewirkt, dass eine Ausgabezeile generiert wird mit den Angaben:

- `tbl->length`
- `count` (im Normalfall `==` `tbl->item_count`, also Anzahl gefundener Tabellenelemente

- `line_count` (Anzahl belegter Tabellenzeilen)
- Durchschnittliche Anzahl von Elementen je belegter Tabellenzeile `count / line_count`.

```
01026 {
01027   /* Für jeden Listenzeiger in tbl->items wird geprüft, ob er auch von
01028    * tbl->first aus erreichbar ist. Wenn nicht, so liegt ein Fehleintrag
01029    * in der Tabelle vor!
01030    */
01031   unsigned int i = 0;
01032   int found_first = 0;
01033   size_t line_count = 0, count = 0;
01034   struct node_list_t *lst = NULL;
01035   for (i=0; i<tbl->length; i++) {
01036     lst = tbl->items[i];
01037     if (lst != NULL) {
01038       count++;
01039       line_count++;
01040       if (compute_hash(lst->head, tbl->length) != i)
01041         abort();
01042       if (lst == tbl->first)
01043         found_first = 1;
01044       if (list_contains_sublist(tbl->first, lst)  != 0)
01045         abort();
01046
01047       while ( lst->right != NULL
01048           && (compute_hash(lst->right->head, tbl->length) == i) )
01049       {
01050         count++;
01051         lst = lst->right;
01052       }
01053     }
01054   } // for (i=0; i<tbl->length; i++)
01055
01056   if (flag == 1)
01057     printf("stat: length: %zu item_count: %zu line_count: %zu relativ: %f\n",
01058       tbl->length, tbl->item_count, line_count,
01059       ((double)count / (double)line_count));
01060
01061   if (count != tbl->item_count)
01062     abort();
01063
01064   if (found_first == 0 && count > 0)
01065     abort();
01066
01067   count = 0;
01068   lst = tbl->first;
01069   while (lst != NULL) {
01070     count++;
01071     lst = lst->right;
01072   }
01073   if (count != tbl->item_count)
01074     abort();
01075
01076 } // static void node_table_check
```

5.1.4.18 static void node_table_clear (struct node_table_t * *tbl*) [static]

Leeren einer Hashtabelle von Knoten. Ein Aufruf von `node_table_clear(tbl)` entfernt die Tabelle *tbl* nicht vollständig aus dem Speicher, sondern leert nur das enthaltene Array der Listenzeiger.

Parameter:

tbl Zeiger auf die zu bereinigenden Knotentabelle.

```
01125 {
01126   /* Falls die Tabelle noch nicht initialisiert war oder keine Einträge
01127    * enthält, gibt es nichts zu bereinigen:
01128    */
01129   if (tbl == NULL)
01130     return;
01131   if (tbl->item_count == 0)
01132     return;
01133
01134   NODE_TABLE_CHECK(tbl, 0);
01135
01136   list_clear(&(tbl->first));
01137   memset(tbl->items, 0, tbl->length * sizeof(struct node_list_t*));
01138   tbl->item_count = 0;
01139
01140   NODE_TABLE_CHECK(tbl, 0);
01141 } // static void node_table_clear
```

5.1.4.19 static struct node_table_t* node_table_copy (struct node_table_t * *tbl*) [static, read]

Kopieren einer bestehenden Hashtabelle.

Parameter:

tbl Zeiger auf die Tabelle, von der eine Kopie erzeugt werden soll.

Rückgabe:

Zeiger auf die Kopie.

```
01704 {
01705   if (tbl == NULL)
01706     return NULL;
01707   if (tbl->item_count == 0) {
01708     struct node_table_t *new_table = node_table_init();
01709     return new_table;
01710   }
01711
01712   NODE_TABLE_CHECK(tbl, 0);
01713
01714   struct node_table_t *new_tbl = malloc(sizeof(struct node_table_t));
01715   new_tbl->length = tbl->length;
01716   new_tbl->item_count = tbl->item_count;
01717   new_tbl->first = NULL;
01718   new_tbl->cursor = NULL;
01719   new_tbl->items =
01720     calloc(tbl->length, sizeof(struct node_list_t*));
01721
01722   /* Nun klone alle Elemente aus tbl und trage die neu entstehenden
01723    * Listeneinträge in new_tbl->items ein!
01724    */
01725   tbl->cursor = tbl->first;
01726   struct node_t *node = NULL;
01727   struct node_list_t *lst_new = NULL;
01728   unsigned int node_hash = 0;
01729   while (tbl->cursor != NULL) {
01730     node = tbl->cursor->head;
01731     node_hash = compute_hash(node, tbl->length);
01732
01733     /* Falls in new_tbl noch kein Knoten mit dem Hashwert node_hash
01734      * existiert, wird das Feld new_tbl->items[node_hash] neu bestückt
01735      * und der neue Eintrag auch mit dem Listenzeiger new_tbl->first
```

```
01736       * verbunden:
01737       */
01738      if (new_tbl->items[node_hash] == NULL) {
01739        lst_new = pool_get();
01740        lst_new->head = node;
01741        lst_new->left = NULL;
01742        lst_new->right = new_tbl->first;
01743
01744        if (new_tbl->first != NULL)
01745          new_tbl->first->left = lst_new;
01746        new_tbl->first = lst_new;
01747
01748        new_tbl->items[node_hash] = lst_new;
01749      }
01750
01751      /* Falls es in new_tbl schon Einträge für andere Knoten mit dem
01752       * Hashwert node_hash gibt, wird der Eintrag für node davor gesetzt.
01753       * Dabei muss node auch mit einem ggf. vorhandenen linken Nachbarn
01754       * verbunden werden. Falls der neue Nachfolger von node das bisherige
01755       * first-Element von new_tbl war, muss auch dieser Zeiger angepasst
01756       * werden.
01757       */
01758      else {
01759        lst_new = pool_get();
01760        lst_new->head = node;
01761        lst_new->left = new_tbl->items[node_hash]->left;
01762        lst_new->right = new_tbl->items[node_hash];
01763
01764        if (lst_new->left != NULL)
01765          lst_new->left->right = lst_new;
01766        lst_new->right->left = lst_new;
01767
01768        if (new_tbl->first == lst_new->right)
01769          new_tbl->first = lst_new;
01770
01771        new_tbl->items[node_hash] = lst_new;
01772      }
01773
01774      tbl->cursor = tbl->cursor->right;
01775    } // while (tbl->cursor != NULL)
01776
01777    NODE_TABLE_CHECK(new_tbl, 0);
01778    return new_tbl;
01779  } // static struct node_table_t* node_table_copy
```

5.1.4.20 static void node_table_double (struct node_table_t * *tbl*) [static]

Verdoppeln einer vollen Hashtabelle. Wenn der Belegungsfaktor der Hashtabelle den Wert 1 erreicht, wird node_table_double aufgerufen, um sie in eine neue Tabelle mit doppelter Größe umzukopieren.

Parameter:

tbl Zeiger auf die Hashtabelle, die verdoppelt werden soll.

```
01195 {
01196   assert(tbl != NULL);
01197
01198   if (tbl->length == 0)
01199     return;
01200
01201   NODE_TABLE_CHECK(tbl, 0);
01202
01203   unsigned int old_length = tbl->length;
```

```
01204   /* Zeiger auf den alten Inhalt der Hashtabelle: */
01205   struct node_list_t *lst_old = tbl->first;
01206
01207   /* Erschaffe zunächst eine neue Tabelle mit doppelter Größe wie
01208    * tbl->length und trage sie als neue tbl->items:
01209    */
01210   tbl->length = 2 * old_length;
01211   tbl->cursor = NULL;
01212   tbl->first = NULL;
01213   free(tbl->items);
01214   tbl->items =
01215     calloc(tbl->length, sizeof(struct node_list_t*));
01216
01217   /* Nun werden alle Elemente von lst_old in die neue Tabelle tbl->items
01218    * übertragen.
01219    * Dabei muss im Gegensatz zu node_table_add keine Duplikatprüfung
01220    * stattfinden, da hier ja nur eine bereits als duplikatfrei bekannte
01221    * Tabelle umkopiert wird. Auch wird tbl->item_count nicht angefasst,
01222    * da sich die Belegung der Tabelle ja nicht ändert.
01223    */
01224   struct node_t *node = NULL;
01225   unsigned int node_hash = 0;
01226   struct node_list_t *lst_new = NULL;
01227
01228   while (lst_old != NULL) {
01229     node = list_pop(&(lst_old));
01230     node_hash = compute_hash(node, tbl->length);
01231
01232     /* Falls noch kein Knoten mit dem Hashwert node_hash existiert, wird
01233      * das Feld tbl->items[node_hash] neu bestückt und der neue Eintrag
01234      * auch mit dem Listenzeiger tbl->first verbunden:
01235      */
01236     if (tbl->items[node_hash] == NULL) {
01237       lst_new = pool_get();
01238       lst_new->head = node;
01239       lst_new->left = NULL;
01240       lst_new->right = tbl->first;
01241
01242       if (tbl->first != NULL)
01243         tbl->first->left = lst_new;
01244       tbl->first = lst_new;
01245
01246       tbl->items[node_hash] = lst_new;
01247     } // if (tbl->items[node_hash] == NULL)
01248
01249     /* Falls es schon Einträge für andere Knoten mit dem Hashwert
01250      * node_hash in tbl gibt, wird der Eintrag für node davor gesetzt.
01251      * Dabei muss node auch mit einem ggf. vorhanden linken Nachbarn
01252      * verbunden werden. Falls der neue rechte Nachbar von node das
01253      * bisherige first-Element von tbl war, muss auch dieser Zeiger
01254      * aktualisiert werden:
01255      */
01256     else {
01257       lst_new = pool_get();
01258       lst_new->head = node;
01259       lst_new->left = tbl->items[node_hash]->left;
01260       lst_new->right = tbl->items[node_hash];
01261
01262       tbl->items[node_hash]->left = lst_new;
01263
01264       if (lst_new->left != NULL)
01265         lst_new->left->right = lst_new;
01266
01267       if (tbl->first == lst_new->right)
01268         tbl->first = lst_new;
01269
01270       tbl->items[node_hash] = lst_new;
```

```
01271     } // if (tbl->items[node_hash] == NULL) ... else
01272
01273   } // while (lst_old != NULL)
01274
01275   NODE_TABLE_CHECK(tbl, 1);
01276
01277 } // static void node_table_double
```

5.1.4.21 static int node_table_empty (struct node_table_t * *tbl*) [inline, static]

Test, ob eine Hashtafel leer ist.

Parameter:

tbl Zeiger auf eine Hashtafel, die auf Leersein getestet werden soll.

Rückgabe:

`0`, falls die Tabelle Elemente enthält, `1`, falls die Tafel initialisiert ist, aber 0 Elemente enthält, oder `2`, falls die Tafel noch nicht initialisiert ist.

```
01518 {
01519   if (tbl == NULL)
01520     return 2;
01521   else if (tbl->item_count == 0)
01522     return 1;
01523   else
01524     return 0;
01525 } // static int node_table_empty
```

5.1.4.22 static struct node_t* node_table_find (struct node_table_t * *tbl*, signed int *content*) [static, read]

Suchen eines Knotens in einer Knotentabelle anhand seines content-Feldes.

Parameter:

tbl Zeiger auf die Knotentabelle, in der gesucht wird.

content Inhalt des zu findenden Knotens.

Rückgabe:

Zeiger auf den gesuchten Knoten, falls er sich in der Tabelle befindet; `NULL`, sonst.

```
01539 {
01540   if (tbl == NULL)
01541     return NULL;
01542   if (tbl->item_count == 0)
01543     return NULL;
01544   else {
01545     struct node_list_t *lst = tbl->first;
01546     while (lst != NULL) {
01547       if (lst->head->content == content)
01548         return lst->head;
01549       else
01550         lst = lst->right;
01551     }
01552     /* Absicherung, falls die Schleife komplett durchlaufen wird, ohne
```

```
01553       * einen Knoten zu finden; sollte eigentlich nie vorkommen:
01554       */
01555      return NULL;
01556    }
01557 } // static struct node_t* node_table_find
```

5.1.4.23 static void node_table_free (struct node_table_t * *tbl*) [static]

Vollständiges Löschen einer Knotentabelle. Nach einem Aufruf von `node_table_free(tbl)` sollte im aufrufenden Code der Zeiger *tbl* sofort wieder neu belegt werden, ggf. mit `NULL`, da der Zeiger sonst nicht mehr zwingend auf ein gültiges Datum zeigt.

Parameter:

tbl Zeiger auf die zu löschende Tabelle.

```
01156 {
01157   if (tbl == NULL)
01158     return;
01159
01160   NODE_TABLE_CHECK(tbl, 0);
01161
01162   /* Falls die Tabelle noch Einträge enthält, müssen diese zuerst
01163    * entfernt werden:
01164    */
01165   if (tbl->item_count > 0) {
01166     list_clear(&(tbl->first));
01167     /* Ein Nullsetzen von tbl->items per memset() wie bei node_table_clear
01168      * ist nicht nötig, da der Speicher für die Hashtabelle als nächstes
01169      * vollständig freigegeben wird:
01170      */
01171   }
01172
01173   free(tbl->items);
01174   free(tbl);
01175 } // static void node_table_free
```

5.1.4.24 static struct node_table_t * node_table_init (void) [static, read]

Erzeugen einer frischen Hashtabelle der Größe 0. Erzeugt wird eine leere Hashtabelle für Knoten. Die Erzeugung erfolgt lazy, d.h. alle Elemente werden mit Nullwerten initialisiert. Das Array `items` aus Listenzeigern wird erst beim ersten Zugriff mittels `node_table_add` tatsächlich mit der Größe initialisiert, die durch das Makro `HASHLENGTH` festgelegt ist. Dann erst wird auch das Feld `length` auf `HASHLENGTH` gesetzt.

Rückgabe:

Frische Hashtabelle vom Typ node_table_t.

```
01101 {
01102   struct node_table_t *tbl = malloc(sizeof(struct node_table_t));
01103   tbl->length = 0;
01104   tbl->item_count = 0;
01105   tbl->first = NULL;
01106   tbl->cursor = NULL;
01107   tbl->items = NULL;
01108
01109   NODE_TABLE_CHECK(tbl, 0);
01110   return tbl;
01111 } // static void node_table_init
```

5.1.4.25 static struct node_t* node_table_iter (struct node_table_t * *tbl*) [static, read]

Iterator über einer Hashtabelle. Die Funktion iteriert über einer Hashtafel. Bei Aufruf von `node_table_iter(tbl)` wird ein Zeiger auf denjenigen Knoten in *tbl* zurückgegeben, auf den der interne Zeiger `cursor` verweist. Das Element wird dabei nicht aus der Tabelle entfernt, der Zeiger wird um ein Element weitergeschoben. Ist das Ende der Tabelle erreicht, zeigt der Cursor auf NULL, und dies ist auch der Rückgabewert. Zurückgesetzt wird der Cursor mit der Funktion `node_table_reset_cursor`.

Parameter:

tbl Zeiger auf die Hashtafel, in der iteriert wird.

Rückgabe:

Zeiger auf den Knoten, auf den der Cursor zeigt.

```
01675 {
01676   if (tbl == NULL)
01677     return NULL;
01678
01679   if (tbl->item_count == 0)
01680     return NULL;
01681
01682   if (tbl->cursor == NULL)
01683     return NULL;
01684
01685   /* Wenn keiner der obigen Fälle eintritt, zeigt der Cursor auf ein
01686    * gültiges Tabellenelement. Gib dies zurück und setze zuvor den Cursor
01687    * auf das nächste Element, sofern es existiert.
01688    */
01689   struct node_t *node = tbl->cursor->head;
01690   tbl->cursor = tbl->cursor->right;
01691   return node;
01692 } // static struct node_t* node_table_iter
```

5.1.4.26 static struct node_t* node_table_pop (struct node_table_t * *tbl*) [static, read]

Entnimmt einen beliebigen Knoten aus einer Hashtabelle. Achtung: Da nicht auszuschließen ist, dass das gerade angeforderte Element auch dasjenige ist, auf das gerade `cursor` zeigt, kann eine verschränkte Benutzung von `node_table_iter` und `node_table_pop` zu unvorhersehbaren und unerwünschten Ergebnissen führen!

Parameter:

tbl Zeiger auf die Tabelle, aus der ein Knoten entnommen werden soll.

Rückgabe:

Zeiger auf den ersten gefundenen Knoten.

```
01574 {
01575   if (tbl == NULL) {
01576     return NULL;
01577   }
01578   else if (tbl->item_count == 0) {
01579     return NULL;
01580   }
01581   else {
```

```
01582     NODE_TABLE_CHECK(tbl, 0);
01583
01584     /* Es wird das Element aus der Tabelle entfernt, auf das der Zeiger
01585      * tbl->first verweist.
01586      */
01587     struct node_t *node = tbl->first->head;
01588
01589     /* Falls tbl->first keinen Nachfolgerknoten hat, wird soeben der
01590      * einzige Knoten der ganzen Tabelle entnommen. Es gibt also auch
01591      * keinen weiteren Knoten mit dem Hashwert node_hash, somit muss
01592      * das entsprechende Feld auf NULL gesetzt werden.
01593      */
01594     unsigned int node_hash = compute_hash(node, tbl->length);
01595     if (tbl->first->right == NULL) {
01596       tbl->items[node_hash] = NULL;
01597       tbl->item_count = 0;
01598     } // if (tbl->first->right == NULL)
01599
01600     /* Falls es noch Knoten in der Tabelle gibt, aber keinen mehr mit
01601      * dem Hashwert node_hash von node, muss tbl->items[node_hash]
01602      * ebenfalls auf NULL gesetzt werden, und der first-Zeiger wird um ein
01603      * Element weiter geschoben. Ansonsten kann auch der Zeiger auf
01604      * tbl->items[i] um ein Element weitergeschoben werden.
01605      */
01606     else {
01607       unsigned int next_hash =
01608         compute_hash(tbl->first->right->head, tbl->length);
01609       if ( next_hash != node_hash )
01610         tbl->items[node_hash] = NULL;
01611       else
01612         tbl->items[node_hash] = tbl->first->right;
01613       tbl->item_count--;
01614     } // if (tbl->first->right == NULL) ... else
01615
01616     /* Nun wird das erste Element der Liste first an den Pool
01617      * zurückgegeben. */
01618     struct node_list_t *old_first = tbl->first;
01619     tbl->first = old_first->right;
01620     if (tbl->first != NULL)
01621       tbl->first->left = NULL;
01622     pool_release(old_first);
01623
01624     /* Somit ist der Knoteneintrag jetzt sauber aus der Hashtabelle
01625      * entfernt, und der gewünschte Knoten kann als Rückgabewert
01626      * geliefert werden.
01627      */
01628     NODE_TABLE_CHECK(tbl, 1);
01629     return node;
01630   }
01631
01632 } // static struct node_t* node_table_pop
```

5.1.4.27 static void node_table_print (struct node_table_t * *tbl*) [static]

Ausgabe eine Hashtafel von Knoten auf stderr. Der Aufruf `node_table_print (tbl)` gibt zu Diagnosezwecken eine einfache textuelle Darstellung der Knotentabelle *tbl* aus.

Die Funktion wird hinter dem Makro `NODE_TABLE_PRINT` versteckt und nur ausgeführt, falls das Makro `USE_DEBUG_PRINTF` definiert ist. Ansonsten wird `NODE_TABLE_PRINT` zu `(void(0))` expandiert.

Parameter:

tbl Zeiger auf die Knotentabelle, die ausgegeben werden soll.

```
01797 {
01798   if (tbl == NULL) {
01799     DEBUG_PRINTF("[]");
01800     return;
01801   }
01802
01803   LIST_PRINT(tbl->first);
01804 } // static void node_table_print
```

5.1.4.28 static void node_table_remove (struct node_table_t * *tbl*, struct node_t * *node*) [static]

Entfernen eines Knotens aus einer Hashtabelle. Der Aufruf `node_table_remove(tbl, node)` entfernt den Knotenzeiger *node* aus der Tabelle, auf die *tbl* zeigt. Dabei wird davon ausgegangen, dass *tbl* auf eine gültig initialisierte Hashtabelle zeigt, ansonsten wird das Programm abgebrochen. Wenn der Knoten nicht in der Tabelle enthalten war, endet die Funktion ohne Aktion.

Noch zu erledigen

Testen, ob bei Unterschreitung eines Belegungsfaktors von 1/4 ein node_table_shrink Vorteile bei Performance und/oder Speicherverbrauch bewirkt.

Parameter:

tbl Zeiger auf die Tabelle, aus der der Knoten entfernt werden soll.

node Knotenzeiger, der aus *tbl* entfernt werden soll.

```
01454 {
01455   assert(node != NULL && tbl != NULL);
01456
01457   NODE_TABLE_CHECK(tbl, 0);
01458
01459   if (tbl->length == 0 || tbl->item_count == 0) {
01460     return;
01461   }
01462
01463   unsigned int node_hash = compute_hash(node, tbl->length);
01464
01465   /* Versuche, den Knoten aus der Liste, die an das Tabellenfeld
01466    * mit der Nummer node_hash angehängt ist, zu entfernen. Wenn
01467    * die Entfernung erfolgreich ist, gibt list_remove den Wert 0
01468    * zurück, sonst 1 oder 2.
01469    */
01470   /* Aufgrund der angepassten Implementierung von list_remove muss an dieser
01471    * Stelle auch nach Änderung der Implementierung auf eine doppelt
01472    * verkettete Liste nichts geändert werden. Es muss einzig beachtet
01473    * werden, dass nun ggf. das first-Element gelöscht wird, sodass dieser
01474    * Zeiger noch korrigiert werden muss.
01475    */
01476   if (tbl->first->head == node)
01477     tbl->first = tbl->first->right;
01478   /* TODO: Evtl. ist es besser, den Knoten hier "per Hand" zu löschen und
01479    * nicht mittels list_remove, sonst laufe ich in manchen Fällen bis ans
01480    * Listenende, also über den letzten Knoten mit Hashwert node_hash hinaus!
01481    * Das führt zwar auch zu einem korrekten Ergebnis, verbraucht aber
01482    * unnötig Zeit!
01483    */
01484   int remove_success = list_remove(&(tbl->items[node_hash]), node);
01485   if ( tbl->items[node_hash] != NULL &&
01486        (compute_hash(tbl->items[node_hash]->head, tbl->length) != node_hash)
01487     )
```

```
01488     tbl->items[node_hash] = NULL;
01489
01490   /* Wenn die Entfernung erfolgreich war, muss der Belegzähler der Tabelle
01491    * dekrementiert werden. Ausserdem findet ein Test statt, ob die Tabelle
01492    * verkleinert werden kann:
01493    */
01494   if (remove_success == 0) {
01495     tbl->item_count--;
01496 /*
01497     if (tbl->item_count < (tbl->length / 4))
01498       node_table_shrink(tbl);
01499 */
01500   }
01501
01502   NODE_TABLE_CHECK(tbl, 1);
01503
01504 } // static void node_table_remove
```

5.1.4.29 static void node_table_reset_cursor (struct node_table_t * *tbl*) [static]

Zurücksetzen des Cursors in einer Hashtabelle. Die Funktion setzt den internen Cursor zurück, mit dessen Hilfe die Funktion `node_table_iter` eine Hashtabelle iteriert. Nach Aufruf von `node_table_reset_cursor` zeigt `tbl->cursor` auf das erste Element der Tabelle. Dabei bedeutet "erstes" Element nicht, dass der Knoten den niedrigsten Hashwert hat o.ä., sondern nur, dass der Zeiger `tbl->first` darauf verweist.

Parameter:

tbl Zeiger auf die Tabelle, deren Cursor zurückgesetzt werden soll.

```
01650 {
01651   if (tbl == NULL)
01652     return;
01653
01654   tbl->cursor = tbl->first;
01655 } // static void node_table_reset_cursor
```

5.1.4.30 static void parse_forest_print (struct parse_forest_t * *forest*) [static]

Ausgabe eine Parsewaldes. Der Aufruf `parse_forest_print(forest)` gibt die Bäume des Waldes *forest* von links nach rechts aus. Bei Aufruf am Ende eines erfolgreichen Parsevorganges besteht dieser Wald aus einem einzigen Baum, der die eindeutige Rechtsableitung der Eingabe repräsentiert. Ausgabeformat ist ein dot-konforme Eingabe.

Parameter:

forest Zeiger auf den auszugebenden Wald.

Wird nur compiliert, falls das Makro `PARSE_TREE` definiert ist.

```
02375 {
02376   if (forest == NULL) {
02377     return;
02378   }
02379   else {
02380     node_number++;
02381     printf("{");
```

```
02382
02383     node_number++;
02384     /* Sofern vorhanden, gib erst rekursiv alle weiter links gelegenen
02385      * Bäume aus. Danach, gib den rechten Baum forest->head aus.
02386      */
02387     parse_forest_print_rec(forest);
02388
02389     printf("}\n");
02390   }
02391 } // static void parse_forest_print
```

5.1.4.31 static void parse_forest_print_rec (struct parse_forest_t * *forest*) [static]

Rekursive Ausgabe eines Baumes im Parsewald und zuvor aller linker Nachbarn. Der Aufruf `parse_forest_print_rec forest` gibt zuerst rekursiv alle weiter links im Wald gelegenen Bäume aus, danach `forest->head`.

Parameter:

forest Zeiger auf den rechtesten Baum des auszugebenden Waldstückes.

Wird nur compiliert, falls das Makro `PARSE_TREE` definiert ist.

```
02337 {
02338   if (forest == NULL) {
02339     return;
02340   }
02341   else {
02342     /* Sofern vorhanden, gib erst rekursiv alle weiter links im Wald
02343      * gelegenen Bäume aus. Danach, gib den rechten Baum forest->head
02344      * aus.
02345      */
02346     parse_forest_print_rec(forest->right);
02347
02348     node_number++;
02349     printf("\n\t{\n");
02350
02351     /* Unsichtbarer Startknoten für den Baum forest->right: */
02352     printf("\t\t%ld [shape=none, label=\"\"];\n", node_number);
02353     parse_tree_print(forest->head, node_number);
02354
02355     printf("\t}\n");
02356   }
02357 } // stativ void parse_forest_print_rec
```

5.1.4.32 static void parse_forest_read (struct parse_forest_t ** *forest*, unsigned long *symbol*) [static]

Bei Leseschritt Erschaffung eines neuen partiellen Ableitungsbaums. Bei einem Leseschritt wird dem bisherigen Parsewald ein neuer Baum ganz rechts hinzugefügt. Dieser besteht zunächst nur aus einem Wurzelknoten, in dem das aktuell gelesene Symbol enthalten ist.

Parameter:

forest Zeiger auf einen Zeiger (Call by Reference) auf den Parsewald, an den der neue Baum rechts angefügt werden soll.

symbol Integerdarstellung des gerade gelesenen Symbols.

Wird nur compiliert, falls das Makro PARSE_TREE definiert ist.

```
01895 {
01896   DEBUG_PRINTF("  parse_forest_read %ld\n", symbol);
01897
01898   // Kreiere einen neuen Baum für symbol:
01899   signed long sym = (signed long) symbol;
01900   struct parse_tree_t *new_tree = malloc(sizeof(struct parse_tree_t));
01901   new_tree->root = sym;
01902   new_tree->children = NULL;
01903
01904   // Kreiere für tree einen neuen Wald-Eintrag:
01905   struct parse_forest_t *new_forest =
01906     malloc(sizeof(struct parse_forest_t));
01907     new_forest->head = new_tree;
01908     new_forest->right = NULL;
01909
01910   // Falls der Wald bisher leer ist, wird der neue Baum sein erster!
01911   if (*forest == NULL) {
01912     *forest = new_forest;
01913   }
01914
01915   /* Ansonsten hänge den neuen Baum direkt ans rechte Ende des Waldes
01916    * und mache den alten Wald zu seinem right-Element.
01917    */
01918   else {
01919     new_forest->right = *forest;
01920     *forest = new_forest;
01921   } // if (*forest == NULL) ... else
01922
01923 } // static void parse_forest_read
```

5.1.4.33 static void parse_forest_reduce (struct parse_forest_t ** *forest*, int *item_number*) [static]

Bei Reduktion Zusammenfassen mehrerer partieller Ableitungsbäume zu einem neuen. Wenn ein Endknoten w anhand der Regel $A \rightarrow \alpha$ reduziert wird, muss das am weitesten rechts stehende Waldstück gefunden werden, dessen Wurzelsymbole konkateniert die Symbolfolge α ergeben. An dieser Stelle wird ein neuer Wurzelknoten für A eingefügt und die Bäume für α zu seinen Nachkommen gemacht.

Parameter:

forest Zeiger auf einen Zeiger (Call by Reference) auf den Parsewald, in dem die zusammenzufassenden Bäume liegen.

item_number Nummer des Items, anhand dessen die Reduktion durchgeführt wurde.

Wird nur compiliert, falls das Makro PARSE_TREE definiert ist.

```
01945 {
01946   DEBUG_PRINTF("  parse_forest_reduce: Item %d, Regel %d\n",
01947       item_number, items[item_number].rule_number);
01948
01949   /* Als erstes muss die passende Regel im Array rules aller Grammatikregeln
01950    * gefunden werden. Beachte dabei: Regel Nr. 1 hat im Array den Index 0,
01951    * usw. Das liegt daran, dass für die eigentliche Regel 0, S' -> S,
01952    * kein Eintrag existiert, denn anhand ihrer darf niemals reduziert
01953    * werden!
01954    */
01955   struct rule_t rule = rules[items[item_number].rule_number - 1];
01956
01957   /* Falls eine Reduktion von Epsilon durchgeführt wurde, muss ganz
```

```
01958     * rechts ein neuer Baum angefügt werden:
01959     */
01960    if (rule.rhs_count == 0) {
01961      DEBUG_PRINTF("    Kreiere Epsilon-Baum\n");
01962
01963      // Erzeuge einen neuen Teilbaum für die Epsilon-Reduktion:
01964      struct parse_tree_t *eps = malloc(sizeof(struct parse_tree_t));
01965        eps->root = 0;
01966        eps->children = NULL;
01967      struct parse_tree_t *eps_root = malloc(sizeof(struct parse_tree_t));
01968        eps_root->root = ~(rule.lhs);
01969        eps_root->children = malloc(sizeof(struct parse_forest_t));
01970          eps_root->children->head = eps;
01971          eps_root->children->right = NULL;
01972
01973      /* Hänge den neuen Baum mit Wurzel eps_root ganz rechts an den
01974       * globalen Parsewald:
01975       */
01976      struct parse_forest_t *new_forest =
01977        malloc(sizeof(struct parse_forest_t));
01978        new_forest->head = eps_root;
01979        new_forest->right = *forest;
01980      *forest = new_forest;
01981
01982    } // if (rule.rhs_count == 0)
01983
01984    /* Falls die rechte Seite der reduzierten Regel alpha != Epsilon ist,
01985     * so muss im Wald das am weitesten rechts liegende Waldstück gefunden
01986     * werden, dessen Wurzeln die Symbole aus alpha enthalten. Diese
01987     * Bäume werden dann unter einer neuen Wurzel zusammengefasst.
01988     * Da der Wald von rechts nach links durchmustert wird, müssen
01989     * die Symbole auf der rechten Seite von rule auch von rechts nach links
01990     * betrachtet werden!
01991     */
01992    else {
01993      DEBUG_PRINTF("    Suche zusammenzufügende Bäume\n");
01994      // Universelle Zählvariable:
01995      int i = 0;
01996
01997      // Flag, ob ein Baum gefunden wurde, dess Wurzel zum rechten Symbol
01998      // der rechten Regelseite passt:
01999      int found_right = 0;
02000
02001      // Flag, ob die Bäume gefunden wurden, die zur rechten Regelseite passen
02002      int found_string = 0;
02003
02004      // Temporärer Zeiger auf den aktuellen betrachteten Waldabschnitt:
02005      struct parse_forest_t *tmp_forest = *forest;
02006
02007      /* Zuerst muss der Fall betrachtet werden, dass das gesuchte Wald-
02008       * stück am rechten Rand des Gesamtwaldes liegt. In diesem Fall gibt
02009       * es nämlich keinen weiter rechts gelegenen Baum, dessen right-Zeiger
02010       * nach der Reduktion angepasst werden muss!
02011       */
02012      DEBUG_PRINTF("      Rechter Baum:\n");
02013      DEBUG_PRINTF("        Symbol %d: %ld ... ",
02014          rule.rhs_count, rule.rhs[rule.rhs_count - 1]);
02015      if (tmp_forest->head->root == rule.rhs[rule.rhs_count - 1]) {
02016        DEBUG_PRINTF("passt\n");
02017
02018        /* Nun wird getestet, ob die weiteren Baumwurzeln zu den restlichen
02019         * Symbolen in rule.rhs passen.
02020         */
02021        for (i=rule.rhs_count - 2; i>=0; i--) {
02022          DEBUG_PRINTF("        Symbol %d: %ld ... ",
02023              i+1, rule.rhs[i]);
02024
```

```
02025         tmp_forest = tmp_forest->right;
02026
02027         /* Falls es weniger Baumwurzeln als Symbole in der reduzierten
02028          * Regel gibt, liegt ein Programmierfehler vor:
02029          */
02030         assert(tmp_forest != NULL);
02031
02032         /* Falls das aktuelle Paar aus Symbol und Baumwurzel zusammen
02033          * passt, wird die Schleife fortgesetzt, ansonsten abgebrochen:
02034          */
02035         if (tmp_forest->head->root == rule.rhs[i]) {
02036           DEBUG_PRINTF("passt zu Wurzel %ld!\n",
02037               tmp_forest->head->root);
02038         }
02039         else {
02040           DEBUG_PRINTF("passt nicht zu Wurzel %ld!\n",
02041               tmp_forest->head->root);
02042           break;
02043         }
02044
02045       } // for (i=rule.rhs_count - 2; i>=0; i--)
02046
02047       /* Im Erfolgsfall ist die FOR-Schleife bis i = -1 durchgelaufen, und
02048        * tmp_forest zeigt auf den an weitesten links stehenden Eintrag des
02049        * gesuchten Waldstückes. Danach kann noch ein Restwald folgen,
02050        * der später noch zum right-Element des neu erschaffenen Baumes
02051        * gemacht werden muss.
02052        */
02053
02054       /* Falls der passende Waldabschnitt gefunden wurde, fasse die Bäume
02055        * unter einer neuen Wurzel zusammen und pflanze den Resultatbaum an
02056        * der entsprechenden Stelle wieder im Wald ein.
02057        */
02058       if (i == -1) {
02059
02060         DEBUG_PRINTF("      Passendes Waldstück gefunden!\n");
02061         found_string = 1;
02062
02063         /* Das gefundene Waldstück, beginnend mit dem rechten Baum des
02064          * Gesamtwaldes, stellt die Kindknoten der neu zu erschaffenden
02065          * Baumwurzel dar.
02066          */
02067         struct parse_forest_t *first_child = *forest;
02068
02069         /* Erschaffe nun eine neue Baumwurzel für das Symbol rule.lhs: */
02070         struct parse_tree_t *new_tree =
02071           malloc(sizeof(struct parse_tree_t));
02072           new_tree->root = ~(rule.lhs);
02073           new_tree->children = first_child;
02074
02075         /* Erstelle einen neuen Waldeintrag für den neuen Baum und
02076          * mache ihn zum ersten Eintrag des Gesamtwaldes.
02077          */
02078         struct parse_forest_t *new_forest =
02079           malloc(sizeof(struct parse_forest_t));
02080           new_forest->head = new_tree;
02081           new_forest->right = tmp_forest->right;
02082         *forest = new_forest;
02083
02084         /* Der letzte Kindeintrag von new_tree, auf den der Zeiger
02085          * tmp_forest noch zeigt, muss noch von seinem bisherigen
02086          * Nachfolgereintrag im Gesamtwald getrennt werden.
02087          * Anschließend kann der Zeiger tmp_forest verworfen werden!
02088          */
02089         tmp_forest->right = NULL;
02090         tmp_forest = NULL;
02091
```

```
02092       } // if (i == -1)
02093
02094       /* Falls das passende Waldstück noch nicht vom Waldanfang aus
02095        * gefunden wurde, muss tmp_forest zurückgesetzt werden, damit
02096        * die Weitersuche an der korrekten Stelle startet!
02097        */
02098       else {
02099         DEBUG_PRINTF("      Passendes Waldstück NICHT gefunden!\n");
02100         tmp_forest = *forest;
02101         found_string = 0;
02102       } // if (i == -1) else
02103
02104     } // if (rechtes Symbol passt zu rechter Baumwurzel)
02105
02106     else {
02107       DEBUG_PRINTF("passt nicht zu %ld!\n",
02108           tmp_forest->head->root);
02109     } // if (rechtes Symbol passt zu rechter Baumwurzel) else
02110
02111     /* Sollte das relevante Waldstück nicht am rechten Rand des Waldes
02112      * beginnen, wird weitergesucht, bis entweder des Waldstück gefunden
02113      * ist oder der Wald komplett erfolglos durchmustert wurde.
02114      * Achtung: Im Erfolgsfall muss man sich noch merken, welcher
02115      * Waldeintrag der direkte Vorgänger des rechtesten passenden Baumes
02116      * ist, um dessen right-Zeiger am Ende korrekt umbiegen zu können!
02117      */
02118     while (found_string == 0) {
02119
02120       /* Suche das rechte Symbol von rule.rhs in tmp_forest: */
02121       found_right = 0;
02122       while ( (tmp_forest->right != NULL) && (found_right == 0) ) {
02123         DEBUG_PRINTF("      Nächster Baum:\n");
02124         DEBUG_PRINTF("        Symbol %d: %ld ... ",
02125             rule.rhs_count, rule.rhs[rule.rhs_count - 1]);
02126
02127         /* Sobald ein Baum gefunden wurde, der zum rechten Symbol in
02128          * rule.rhs passt, kann die Schleife beendet werden:
02129          */
02130         if (tmp_forest->right->head->root == rule.rhs[rule.rhs_count - 1])
02131         {
02132           DEBUG_PRINTF("passt zu Wurzel %ld!\n",
02133               tmp_forest->right->head->root);
02134           found_right = 1;
02135         }
02136
02137         /* Ansonsten wird im nächsten Schleifendurchlauf der
02138          * nächste Baum des Waldes getestet:
02139          */
02140         else {
02141           DEBUG_PRINTF("passt nicht zu Wurzel %ld!\n",
02142               tmp_forest->right->head->root);
02143           tmp_forest = tmp_forest->right;
02144         }
02145
02146       } // while ( (tmp_forest->right != NULL) && (found_right == 0) )
02147
02148       /* Falls tmp_forest ganz durchmustert wurde, ohne das rechte Symbol
02149        * der reduzierten Regel zu finden, liegt ein Programmierfehler vor!
02150        */
02151       assert(tmp_forest->right != NULL);
02152
02153       /* Falls die Suche nach einem passenden ersten Waldeintrag
02154        * erfolgreich war, zeigt tmp_forest nun auf den direkten Vorgänger
02155        * des rechtesten Baumes im Wald, der zu rule passen könnte. Diese
02156        * spezielle Position im Wald muss aus zwei Gründen in einem
02157        * eigenen Zeiger candidate_before festgehalten werden:
02158        * 1.) Im Erfolgsfall muss ich den right-Zeiger von candidate_before
```

```
02159         *     auf die Wurzel des neuen Baumes umbiegen.
02160         * 2.) Im Falle eines Misserfolges muss tmp_forest auf diese Stelle
02161         *     gesetzt werden, damit die Suche von dort aus fortgesetzt
02162         *     werden kann.
02163         */
02164        struct parse_forest_t *candidate_before = tmp_forest;
02165
02166        /* Nun soll tmp_forest auf den ersten Baum des aktuelle
02167         * betrachteteten Waldstückes zeigen, damit die weitere Suche
02168         * analog zum obigen Fall stattfinden kann:
02169         */
02170        tmp_forest = tmp_forest->right;
02171
02172        /* Nun muss getestet werden, ob auch die Wurzeln der Nachfolgerbäume
02173         * zu den restlichen Symbole in rule.rhs passen:
02174         */
02175        for (i=rule.rhs_count - 2; i>=0; i--) {
02176          DEBUG_PRINTF("        Symbol %d: %ld ... ",
02177              i+1, rule.rhs[i]);
02178
02179          tmp_forest = tmp_forest->right;
02180
02181          /* Falls es weniger Baumwurzeln gibt als Symbole in der reduzierten
02182           * Regel, liegt ein Programmierfehler vor:
02183           */
02184          assert(tmp_forest != NULL);
02185
02186          /* Falls das aktuelle Paar aus Symbol und Baumwurzel zusammen
02187           * passt, wird die Schleife fortgesetzt, ansonsten abgebrochen:
02188           */
02189          if (tmp_forest->head->root == rule.rhs[i]) {
02190            DEBUG_PRINTF("passt zu Wurzel %ld!\n",
02191                tmp_forest->head->root);
02192          }
02193          else {
02194            DEBUG_PRINTF("passt nicht zu Wurzel %ld!\n",
02195                tmp_forest->head->root);
02196            break;
02197          }
02198
02199        } // for (i=rule.rhs_count - 2; i>=0; i--)
02200
02201        /* Im Erfolgsfall ist die FOR-Schleife bis i = -1 durchgelaufen, und
02202         * tmp_forest zeigt auf den am weitesten links stehenden Eintrag des
02203         * gesuchten Waldstückes. Danach dann noch ein Restwald folgen,
02204         * der später noch zum right-Element des neu erschaffenen Baumes
02205         * gemacht werden muss.
02206         */
02207
02208        /* Falls der passende Waldabschnitt gefunden wurde, fasse die Bäume
02209         * unter einer neuen Wurzel zusammen und pflanze den Resultatbaum an
02210         * der entsprechenden Stelle wieder im Wald ein.
02211         */
02212        if (i == -1) {
02213
02214          DEBUG_PRINTF("      Passendes Waldstück gefunden!\n");
02215          found_string = 1;
02216
02217          /* Das gefundene Waldstück, rechts beginnend mit dem Eintrag
02218           * candidate_before->right, stellt die Kindknoten der neu zu
02219           * erschaffenden Baumwurzel dar:
02220           */
02221          struct parse_forest_t *first_child = candidate_before->right;
02222
02223          /* Erschaffe nun eine neue Baumwurzel für das Symbol rule.lhs: */
02224          struct parse_tree_t *new_tree =
02225            malloc(sizeof(struct parse_tree_t));
```

```
02226           new_tree->root = ~(rule.lhs);
02227           new_tree->children = first_child;
02228
02229         /* Erschaffe einen neuen Waldeintrag für den neuen Baum und
02230          * pflanze ihn zwischen candidate_before und tmp_forest->right ein:
02231          */
02232         struct parse_forest_t *new_forest =
02233           malloc(sizeof(struct parse_forest_t));
02234           new_forest->head = new_tree;
02235           new_forest->right = tmp_forest->right;
02236         candidate_before->right = new_forest;
02237
02238         /* Der letzte Kindeintrag von new_tree, auf den der Zeiger
02239          * tmp_forest noch zeigt, muss noch von seinem bisherigen
02240          * Nachfolgereintrag im Gesamtwald getrennt werden.
02241          * Anschließend kann der Zeiger tmp_forest verworfen werden!
02242          */
02243         tmp_forest->right = NULL;
02244         tmp_forest = NULL;
02245
02246       } // if (i == -1)
02247
02248       /* Falls das passende Waldstück noch nicht gefunden wurden, muss
02249        * tmp_forest zurückgesetzt werden, damit die weitere Suche
02250        * an der richtigen Stelle startet:
02251        */
02252       else {
02253         DEBUG_PRINTF("      Passendes Waldstück NICHT gefunden!\n");
02254         tmp_forest = candidate_before->right;
02255         found_string = 0;
02256       } // if (i == -1) else
02257
02258     } // while (found_string == 0)
02259
02260   } // if (rule.rhs_count == 0) else
02261
02262 } // static void parse_forest_reduce
```

5.1.4.34 static void parse_tree_print (struct parse_tree_t * *tree*, unsigned long *father*) [static]

Ausgabe eines partiellen Ableitungsbaums auf `stdout`. Der Funktionsaufruf `parse_tree_print(tree)` gibt die Wurzel des Baumes *tree* aus und rekursiv alle seine Kindknoten. Das Ausgabeformat ist konform zum Pogramm `dot` aus dem Paket `graphviz`.

Parameter:

tree Zeiger auf den Baum, der ausgegeben werden soll.

father Nummer des Vaterknotens.

Wird nur compiliert, falls das Makro `PARSE_TREE` definiert ist.

```
02279 {
02280   if (tree != NULL) {
02281
02282     node_number++;
02283     unsigned long my_number = node_number;
02284
02285     /* Für ein Terminalsymbol, gib seine Nummer aus; für ein
02286      * Nichtterminalsymbol, gib seinen Namen aus der
02287      * Grammatikdatei aus:
02288      */
02289     if (tree->root >= 0)
```

```
02290       printf("\t\t%ld [label=\"Token %ld\", shape=box];\n",
02291           node_number, tree->root);
02292     else
02293       printf("\t\t%ld [label=\"%s\"];\n",
02294          node_number, nonterminals[~(tree->root)-1].name);
02295
02296     /* Erschaffe eine Kante zwischen Vater- und Kindknoten: */
02297     printf("\t\t%ld -> %ld;\n", father, node_number);
02298
02299     /* Drucke die (von rechts nach links gespeicherten) Kinder von
02300      * links nach rechts aus). Dazu müssen sie allerdins zuerst
02301      * umgeordnet werden.
02302      */
02303     struct parse_forest_t *children = tree->children;
02304     struct parse_forest_t *print_children = NULL;
02305     struct parse_forest_t *tmp = NULL;
02306     while (children != NULL) {
02307       tmp = malloc(sizeof(struct parse_forest_t));
02308       tmp->head = children->head;
02309       tmp->right = print_children;
02310       print_children = tmp;
02311       children = children->right;
02312     } // while (children != NULL)
02313
02314     while (print_children != NULL) {
02315       parse_tree_print(print_children->head, my_number);
02316       print_children = print_children->right;
02317     }
02318
02319   } // if (tree != NULL)
02320 } // static void parse_tree_print
```

5.1.4.35 static struct node_list_t* pool_get (void) [static, read]

Anfordern eines neuen Listenelementes aus dem Pool. Anstatt bei jeder Eintragung eines Knotens in eine Liste Speicher für ein neues Listenelement per `malloc` anzufordern, wird zunächst im Pool nachgesehen, ob es noch ein ungenutztes Listenelement gibt. Dieses kann allerdings noch ungültige Daten enthalten. Im aufrufenden Code muss daher für eine sinnvolle Belegung gesorgt werden.

Rückgabe:

Zeiger auf ein Listenelement vom Typ `struct node_list_t`.

```
00521 {
00522 #ifdef DEBUG_ALL
00523   return malloc(sizeof(struct node_list_t));
00524 #else
00525   /* Falls der Pool kein Element mehr enthält, muss Speicher für ein
00526    * neues angefordert werden:
00527    */
00528   if (pool == NULL) {
00529     return malloc(sizeof(struct node_list_t));
00530   }
00531   else
00532   {
00533     struct node_list_t *elem = pool;
00534     pool = elem->right;
00535     return elem;
00536   }
00537 #endif
00538 } // static struct node_list_t* pool_get
```

5.1.4.36 static void pool_release (struct node_list_t * *elem*) [static]

Zurückgeben eines aktuell nicht benötigten Listenelementes an den Pool. Anstatt den Speicher für ein aktuell nicht benötigtes Listenelement per `free` sofort wieder an das System zurückzugeben, werden die Listenelemente in einem Pool gesammelt, aus dem bei Bedarf schnell wieder Speicher zugewiesen werden kann. Dabei wird nicht darauf geachtet, ob das Listenelement noch mit linken und rechten Nachbarn verbunden ist; darauf muss im aufrufenden Code geachtet werden.

Parameter:

elem Zeiger auf das Listenelement, das wieder in den Pool aufgenommen wird.

```
00558 {
00559 #ifdef DEBUG_ALL
00560   free(elem);
00561 #else
00562   if (elem != NULL)
00563   {
00564     /* Setze elem an den Anfang des Pools: */
00565     elem->left = NULL;
00566     elem->right = pool;
00567     if (pool != NULL)
00568       pool->left = elem;
00569     pool = elem;
00570   }
00571 #endif
00572 } // static void pool_release
```

5.1.4.37 static void pool_release_list (struct node_list_t * *first*) [static]

Zurückgeben einer kompletten Liste an den Pool. Vergleichbar mit `pool_release` werden auch hier die Elemente der zu löschenden Liste nicht per `free` an das System zurückgegeben, sondern im Pool gesammelt. Im Unterschied zu `pool_release` werden hier alle Listenelemente bis zum Listenende an den Pool zurückgegeben.

Parameter:

first Zeiger auf den Anfang der zurückzugebenden Liste.

```
00588 {
00589   if (first == NULL)
00590     return;
00591 #ifdef DEBUG_ALL
00592   struct node_list_t *next = first->right;
00593   free(first);
00594   while (next != NULL) {
00595     first = next;
00596     next = first->right;
00597     free(first);
00598   }
00599 #else
00600   /* Suche zunächst das Ende der zu löschenden Liste: */
00601   struct node_list_t *last = first;
00602   while(last->right != NULL) {
00603     last = last->right;
00604   }
00605
00606   /* Nun zeigt last auf das letzte Element der zu löschenden Liste. */
00607   if (pool != NULL)
00608     pool->left = last;
```

```
00609   last->right = pool;
00610   pool = first;
00611 #endif
00612 } // static void pool_release_list
```

5.1.4.38 static int reduce_read (void) [static]

Durchführung eines Reduktions-/Leseschrittes. Die Funktion `reduce_read` sorgt für die Durchführung von Reduktions-/Leseschritten und wird aufgerufen, wenn der Graph des Parsers nur noch reduzierbare und/oder lesbare Endknoten enthält. Im Falle eines Leseschrittes wird die aktuelle Phase beendet und ein neues Eingabesymbol in den Lookahead-Puffer eingefügt.

Rückgabe:

Integerwert, der Erfolg oder Misserfolg der Aktion durch die Werte 0 (Erfolg) oder 1 (Misserfolg) kennzeichnet.

```
03841 {
03842   // Zeiger für evtl. zu löschenden Endknoten:
03843   struct node_t *kill_node = NULL;
03844
03845   /* Suche in der Menge der reduzier- und lesbaren Knoten nach einem Knoten,
03846    * für den ein passender Pfad existiert.
03847    */
03848   struct node_t *suitable_node = rtsearch(Both);
03849
03850   /* Falls das Suchergebnis leer ist, kann die Eingabe nicht Element der
03851    * vom Parser erkannten Sprache sein.
03852    */
03853   if (suitable_node == NULL) {
03854     DEBUG_PRINTF("  Kein Endknoten mit passendem Pfad gefunden!\n");
03855     return 1;
03856   }
03857
03858   /* Falls das Suchergebnis ein reduzierbarer Endknoten w ist,
03859    * wird genau dieser Knoten jetzt reduziert:
03860    */
03861   if ( items[suitable_node->content].dot_symbol_type == Epsilon ) {
03862
03863     DEBUG_PRINTF("  Gültiger reduzierbarer Endknoten %ld wurde gefunden!\n",
03864         suitable_node->content);
03865
03866     /* Vermerke die Reduktion im Wald der partiellen Ableitungsbäume:
03867      */
03868     PARSE_FOREST_REDUCE(&parse_forest, suitable_node->content);
03869
03870     // Alle lesbaren Endknoten müssen entfernt werden:
03871     while (node_table_empty(readable_nodes) == 0) {
03872       kill_node = node_table_pop(readable_nodes);
03873       graph_adjust(&kill_node);
03874     }
03875
03876     // Alle reduzierbaren Endknoten ausser w werden entfernt:
03877     while (node_table_empty(reducible_nodes) == 0) {
03878       kill_node = node_table_pop(reducible_nodes);
03879       if (kill_node == suitable_node)
03880         kill_node = NULL;
03881       else
03882         graph_adjust(&kill_node);
03883     }
03884
03885     /* Bemerkung: Aufgrund der Invariante, die der Parser unterhält,
```

```
03886      * enthält der Graph unmittelbar vor einem Reduktions-/Leseschritt
03887      * keine expandierbaren Endknoten. Die Menge expandable_nodes sollte
03888      * also leer sein und kann daher beim Entfernen aller verbliebenen
03889      * Endknoten übergangen werden.
03890      */
03891
03892     /* Betrachte den Nichtterminalknoten A, der im Graph einziger direkter
03893      * Vorgänger von w ist. Entferne die Kante zwischen A und w.
03894      */
03895     struct node_t *nt_node =
03896       node_table_pop(suitable_node->predecessors);
03897     node_table_remove(nt_node->successors, suitable_node);
03898
03899     /* Falls ein Itemknoten mehr als einen Vorgängerknoten hatte, liegt
03900      * ein Programmierfehler vor:
03901      */
03902     assert(node_table_empty(suitable_node->predecessors) != 0);
03903
03904     // Entferne w aus dem Graphen:
03905     node_table_free(suitable_node->predecessors);
03906     node_table_free(suitable_node->successors);
03907     free(suitable_node);
03908     suitable_node = NULL;
03909
03910     /* Nun müssen noch die bisherigen Vorgängerknoten von A behandelt
03911      * werden, da aus ihnen die neuen Endknoten des Graphen entstehen.
03912      */
03913     treat_predecessors(&nt_node);
03914
03915     /* Falls die Vorgängerbehandlung fehlerfrei durchgelaufen ist, ist der
03916      * Reduktionsschritt abgeschlossen.
03917      */
03918     return 0;
03919
03920   } // if (Epsilon)
03921
03922   /* Falls das Suchergebnis ein lesbarer Endknoten ist:
03923    * Nun findet ein Leseschritt für alle lesbaren Endknoten statt;
03924    * die aktuelle Phase wird beendet.
03925    */
03926   else {
03927
03928     DEBUG_PRINTF("  Gültiger lesbarer Endknoten %ld wurde gefunden!\n",
03929         suitable_node->content);
03930
03931     /* Vermerke den Leseschritt im Wald der partiellen Ableitunsgbäume:
03932      */
03933     PARSE_FOREST_READ(&parse_forest,
03934         lookahead[phase_number % LOOKAHEAD_LENGTH]);
03935
03936     /* Alle reduzierbaren Endknoten und ggf. einige ihrer Vorgängerknoten
03937      * müssen entfernt werden:
03938      */
03939     while (node_table_empty(reducible_nodes) == 0) {
03940       kill_node = node_table_pop(reducible_nodes);
03941       graph_adjust(&kill_node);
03942     }
03943
03944     /* Bemerkung:
03945      * 1.) Aufgrund der Invariante, die der Parser unterhält, kann
03946      * die Menge der expandierbaren Endknoten hier ignoriert werden, da sie
03947      * leer sein sollte.
03948      * 2.) Wenn ein lesbarer Endknoten gefunden wurde, für den es einen
03949      * passenden Pfad gibt, muss das nicht auch für jeden anderen lesbaren
03950      * Endknoten gelten. Dennoch wird der Leseschritt an dieser Stelle für
03951      * alle lesbaren Endknoten durchgeführt. Knoten, für die dies ein
03952      * Fehler war, werden später im Parsevorgang aussortiert.
```

```
03953       */
03954
03955      /* Beende die aktuelle Phase durch Lesen des nächsten Eingabesymbols
03956       * und Bereinigen der temporären Listen:
03957       */
03958      DEBUG_PRINTF("  Token %ld gelesen!\n",
03959          lookahead[phase_number % LOOKAHEAD_LENGTH]);
03960      lookahead[phase_number % LOOKAHEAD_LENGTH] = call_lexer();
03961      DEBUG_PRINTF("  Lege neues Eingabesymbol %ld in Puffer!\n",
03962        lookahead[phase_number % LOOKAHEAD_LENGTH]);
03963      phase_number = phase_number + 1;
03964      DEBUG_PRINTF("  Bereinige expanded_nodes\n");
03965      node_table_clear(expanded_nodes);
03966      DEBUG_PRINTF("  Bereinige expanded_nonterminals\n");
03967      node_table_clear(expanded_nonterminals);
03968
03969      /* Bewege in allen lesbaren Endknoten den Punkt um eine Stelle nach
03970       * rechts, d.h. ersetze jedes Item [A -> alpha . a_{i+1} beta] durch
03971       * sein Nachfolgeitem [A -> alpha a_{i+1} . beta].
03972       * Untersuche dabei jeden dieser neuen Endknoten auf Expandier-,
03973       * Reduzier- oder Lesbarkeit in der nächsten Phase.
03974       */
03975      while (node_table_empty(readable_nodes) == 0) {
03976        suitable_node = node_table_pop(readable_nodes);
03977        DEBUG_PRINTF("    Bearbeite lesbaren Endknoten %ld\n",
03978            suitable_node->content);
03979        suitable_node->content = suitable_node->content + 1;
03980        test_new_endnode(&suitable_node);
03981      }
03982
03983      /* Nun sind alle Kandidaten für neue Endknoten kategorisiert in
03984       * lesbare, reduzierbare, expandierbare und irrelevante Knoten.
03985       * Neue lesbare Endknoten wurden von der Funktion test_new_endnode
03986       * in die Liste new_readable_endnodes eingetragen. Diese Liste
03987       * kann jetzt in die Tabelle readable_endnodes überführt werden.
03988       */
03989      struct node_t *read_node = NULL;
03990      while (new_readable_nodes != NULL) {
03991        read_node = list_pop(&new_readable_nodes);
03992        node_table_add(readable_nodes, read_node);
03993      }
03994      read_node = NULL;
03995
03996      DEBUG_PRINTF("    Alle lesbaren Endknoten bearbeitet!\n");
03997
03998      DEBUG_PRINTF(
03999          "Abschliessende Behandlung neuer reduzierbarer Knoten:\n");
04000      /* Falls bei der Bearbeitung der lesbaren Endknoten neue reduzierbare
04001       * Endknoten entstanden sind, müssen diese vor Ende der Phase
04002       * bearbeitet werden, um die Invariante vor dem ersten Expansionsschritt
04003       * der neuen Phase aufrecht zu erhalten.
04004       */
04005      if (node_table_empty(reducible_nodes) == 0) {
04006
04007        /* Suche in der Menge der reduzierbaren Endknoten nach einem
04008         * Knoten, für den ein passender Pfad existiert.
04009         */
04010        suitable_node = rtsearch(Reduce);
04011        /* Falls das Suchergebnis leer ist, können die aktuell vorhandenen
04012         * reduzierbaren Endknoten nicht zu einer akzeptierenden
04013         * Berechnung korrespondieren und können aus dem Graphen entfernt
04014         * werden.
04015         */
04016        if (suitable_node == NULL) {
04017          DEBUG_PRINTF(
04018              "  Kein gültiger reduzierbarer Endknoten gefunden!\n");
04019
```

```
04020         // Alle reduzierbaren Endknoten werden entfernt:
04021         while (node_table_empty(reducible_nodes) == 0) {
04022           kill_node = node_table_pop(reducible_nodes);
04023           graph_adjust(&kill_node);
04024         }
04025       } // if (suitable_node == NULL)
04026 
04027       /* Falls ein reduzierbarer Endknoten mit passendem Pfad gefunden
04028        * wurde, muss die entsprechende Reduktion sofort durchgeführt
04029        * werden.
04030        */
04031       else if ( items[suitable_node->content].dot_symbol_type == Epsilon ) {
04032         DEBUG_PRINTF(
04033             "  Gültiger reduzierbarer Endknoten %ld wurde gefunden!\n",
04034             suitable_node->content);
04035 
04036         /* Vermerke die Reduktion im Wald der partiellen Ableitungsbäume:
04037          */
04038         PARSE_FOREST_REDUCE(&parse_forest, suitable_node->content);
04039 
04040         // Alle lesbaren Endknoten müssen entfernt werden:
04041         while (node_table_empty(readable_nodes) == 0) {
04042           kill_node = node_table_pop(readable_nodes);
04043           graph_adjust(&kill_node);
04044         }
04045 
04046         /* Alle reduzierbaren Endknoten ausser suitable_node werden
04047          * entfernt:
04048          */
04049         while (node_table_empty(reducible_nodes) == 0) {
04050           kill_node = node_table_pop(reducible_nodes);
04051           if (kill_node == suitable_node)
04052             kill_node = NULL;
04053           else
04054             graph_adjust(&kill_node);
04055         }
04056 
04057         /* Der Graph kann nun auch neue expandierbare Endknoten enthalten,
04058          * die ebenfalls entfernt werden müssen:
04059          */
04060         while (node_table_empty(expandable_nodes) == 0) {
04061           kill_node = node_table_pop(expandable_nodes);
04062           graph_adjust(&kill_node);
04063         }
04064 
04065         /* Betrachte den Nichtterminalknoten A, der im Graph einziger
04066          * direkter Vorgänger von suitable_node ist, und entferne die
04067          * Kante zwischen den beiden:
04068          */
04069         struct node_t *nt_node =
04070           node_table_pop(suitable_node->predecessors);
04071         node_table_remove(nt_node->successors, suitable_node);
04072 
04073         /* Falls ein Itemknoten mehr als einen Vorgänger hatte, liegt ein
04074          * Programmierfehler vor:
04075          */
04076         assert(node_table_empty(suitable_node->predecessors) != 0);
04077 
04078         // Entferne den reduzierbaren Knoten aus dem Graphen:
04079         node_table_free(suitable_node->predecessors);
04080         node_table_free(suitable_node->successors);
04081         free(suitable_node);
04082         suitable_node = NULL;
04083 
04084         /* Nun müssen noch die bisherigen Vorgängerknoten von A behandelt
04085          * werden, da aus ihnen die neuen Endknoten des Graphen entstehen.
04086          */
```

```
04087         treat_predecessors(&nt_node);
04088
04089       } // else if (Epsilon)
04090
04091       /* Ansonsten ist das Suchergebnis ungültig, denn wenn die
04092        * Suche auf lesbare Endknoten eingeschränkt wurde, darf das
04093        * Ergebnis nur ein Knoten mit dot_symbol_type == Epsilon sein!
04094        */
04095       else {
04096         fprintf(stderr, "treat_predecessors: Falscher Knoten wurde gefunden!\n");

04097         abort();
04098       } // else if (Epsilon) ... else
04099
04100     } // if (node_table_empty(reducible_nodes) == 0)
04101
04102     DEBUG_PRINTF("\nWurzelknoten des Graphen nach Ende der Phase:\n");
04103     LIST_PRINT(graph);
04104
04105 #ifdef PARSE_TREE_PHASES
04106     node_number++;
04107     printf("\n  { %ld [label=\"Parsewald nach Phase %ld\"] }\n",
04108       node_number, phase_number);
04109     PARSE_FOREST_PRINT(parse_forest);
04110 #endif
04111
04112     return 0;
04113
04114   } // if (Epsilon) ... else
04115
04116 } // int reduce_read
```

5.1.4.39 static struct node_t* rtsearch (enum searchflag_t *searchflag*) [static, read]

Durchführung der rückwärts-topologischen Suche nach reduzier- oder lesbaren Endknoten. Diese Funktion implementiert die Variante der "zyklenignorierenden Suche". Alle reduzierbaren und lesbaren Endknoten werden daraufhin untersucht, ob für sie ein passender Pfad gefunden wird. Sobald der erste Endknoten mit einem solchen Pfad gefunden wurde, ist die Suche beendet.

Parameter:

searchflag Wert vom Typ `searchflag_t`, durch den festgelegt wird, ob nur reduzierbare Endknoten, nur lesbare Endknoten oder beide Typen berücksichtigt werden sollen.

Rückgabe:

Ein Zeiger auf einen reduzier- oder lesbaren Endknoten, für den es einen passenden Pfad gibt, falls solch ein Knoten existiert. Sonst: `NULL`.

Noch zu erledigen

Bei der Betrachtung eines Nichtterminalknotens bietet sich eine Möglichkeit zur Fehlererkennung an: Wenn für den selben Wert von `i` von zwei verschiedenen Nachfolgern aus assoziierte Endknoten gemeldet werden, ist die Grammatik nicht LR(k)! Bequem zu testen bspw. über einen Bitvergleich in `prefixes_num`!

```
03020 {
03021   search_id++;
03022
03023   // Universelle Zählvariablen:
```

```
03024   int i, j, k = 0;
03025
03026   // Aktuell untersuchter Knoten:
03027   struct node_t *current_node = NULL;
03028
03029   // Item, dessen rechte Seite aktuell betrachtet wird:
03030   struct item_t current_item;
03031
03032   // Menge der noch zu untersuchenden Knoten:
03033   struct node_table_t *search_nodes = node_table_init();
03034
03035   /* Temporärer Zeiger zum Iterieren über die Vorgängerknoten des
03036    * aktuell untersuchten Knotens:
03037    */
03038   struct node_t *pred_node = NULL;
03039
03040   /* Flag, ob ein Knoten etwas zur Herleitung des Lookaheads beitragen
03041    * konnte:
03042    */
03043   int ok = 0;
03044
03045   if (node_table_empty(readable_nodes) != 0)
03046     DEBUG_PRINTF("  rtsearch: Keine lesbaren Knoten!\n");
03047   if (node_table_empty(reducible_nodes) != 0)
03048     DEBUG_PRINTF("  rtsearch: Keine reduzierbaren Knoten!\n");
03049
03050   // Betrachte zuerst alle lesbaren Endknoten:
03051   if (searchflag == Read || searchflag == Both) {
03052
03053     node_table_reset_cursor(readable_nodes);
03054     current_node = node_table_iter(readable_nodes);
03055     while (current_node != NULL) {
03056
03057       current_item = items[current_node->content];
03058       ok = 0;
03059       DEBUG_PRINTF("  Betrachte lesbaren Endknoten %ld\n",
03060         current_node->content);
03061
03062       /* Falls der aktuelle Knoten zuvor innerer Knoten im Graph war, kann
03063        * er noch einen alten und somit jetzt ungültigen Präfixvektor
03064        * enthalten. Dieser muss erst gelöscht und neu initialisiert werden:
03065        */
03066       memset(current_node->endnodes, 0,
03067           LOOKAHEAD_LENGTH * sizeof(struct node_t*));
03068       current_node->prefixes = 0;
03069       current_node->last_visited = search_id;
03070
03071       // Betrachte alle Strings u' in FIRST_k(right(current_item))
03072       for (i=0; i < current_item.first_k_count; i++) {
03073
03074         /* Gib Vergleich zwischen aktuellem Lookahead und aktuell
03075          * betrachtetem String aus der FIRST_k-Menge des Knotens aus:
03076          */
03077         DEBUG_PRINTF("    Fk: ");
03078         for (j=0; j < LOOKAHEAD_LENGTH; j++) {
03079           DEBUG_PRINTF("%3ld, ", current_item.first_k_item[i][j]);
03080         }
03081         DEBUG_PRINTF("\n");
03082         DEBUG_PRINTF("    LA: ");
03083         for (j=0; j < LOOKAHEAD_LENGTH; j++) {
03084           DEBUG_PRINTF("%3ld, ",
03085               lookahead[ (phase_number + j) % LOOKAHEAD_LENGTH ]);
03086         }
03087         DEBUG_PRINTF("\n");
03088
03089         // Vergleiche den aktuellen String u' mit dem Lookahead-String u:
03090         for (j=0; j < LOOKAHEAD_LENGTH; j++) {
```

```
03091           if ( current_item.first_k_item[i][j] !=
03092               lookahead[ ( (phase_number + j) % LOOKAHEAD_LENGTH) ] )
03093             break;
03094         }
03095
03096         /* Falls die innere for-Schleife bis j == LOOKAHEAD_LENGTH
03097          * durchläuft, wurde der Lookaheadstring durch die rechte Seite
03098          * des aktuell betrachteten lesbaren Endknotens vollständig
03099          * hergeleitet. Somit ist die Suche abgeschlossen!
03100          */
03101         if (j == LOOKAHEAD_LENGTH) {
03102           DEBUG_PRINTF("    Suche erfolgreich beendet!\n");
03103           return current_node;
03104         } // if (j == LOOKAHEAD_LENGTH)
03105
03106         /* Ansonsten wurde entweder eine Stelle gefunden, in der u und u'
03107          * voneinander abweichen, oder u' endete vorzeitig, d.h.
03108          * current_item.first_k_item[i][j] == 0.
03109          */
03110         else {
03111           /* Falls |u'| < |u| gilt und die beiden Strings in den ersten
03112            * j = |u'| Stellen übereinstimmten, wird ein Eintrag für u' in
03113            * den Präfixvektor des aktuellen Knotens aufgenommen.
03114            * Dies deckt auch den Fall u' = Epsilon ab, in dem die Schleife
03115            * schon bei j == 0 terminiert.
03116            */
03117           if (current_item.first_k_item[i][j] == 0) {
03118             DEBUG_PRINTF(
03119                 "    FIRST_k-Eintrag verbraucht => Muss Vorgänger besuchen!\n");

03120             current_node->endnodes[j] = current_node;
03121             current_node->prefixes = current_node->prefixes | (1 << j);
03122             ok = 1;
03123           }
03124
03125           /* Ansonsten gibt es an der j+1-sten Position einen Unterschied
03126            * zwischen u' und u, so dass dieses Präfix bei der aktuellen
03127            * Suche verworfen werden kann!
03128            * In diesem Fall muss nichts unternommen werden. Sofern vorhanden,
03129            * wird der nächste String aus der FIRST_k-Menge von current_node
03130            * betrachtet.
03131            */
03132         } // if (j == LOOKAHEAD_LENGTH) ... else
03133
03134       } // for (i=0; i < current_item.first_k_count; i++)
03135
03136       /* Falls der Endknoten nicht zu einer akzeptierenden Berechnung führen
03137        * kann: Entfernen und Graphbereinigung.
03138        */
03139       if (ok == 0) {
03140         DEBUG_PRINTF("  Graphjustierung für Knoten %ld wird veranlasst ...",
03141             current_node->content);
03142         graph_adjust(&current_node);
03143         DEBUG_PRINTF("  Graphjustierung erledigt\n");
03144       }
03145
03146       /* Falls der Endknoten allein nicht zur Herleitung des Lookahead genügt,
03147        * aber er zu einer akzeptierenden Berechnung führen kann:
03148        * Füge jeden Vorgängerknoten von current_node zur Suchmenge hinzu,
03149        * sofern er sich dort noch nicht befindet:
03150        */
03151       else {
03152         node_table_reset_cursor(current_node->predecessors);
03153         pred_node = node_table_iter(current_node->predecessors);
03154         while (pred_node != NULL) {
03155           node_table_add(search_nodes, pred_node);
03156           pred_node = node_table_iter(current_node->predecessors);
```

```
03157         }
03158       }
03159
03160       // Nächster lesbarer Knoten für folgenden Schleifendurchlauf:
03161       current_node = node_table_iter(readable_nodes);
03162
03163     } // while (current_node != NULL) // readable_nodes
03164
03165   } // if (searchflag == Read || searchflag == Both
03166
03167   // Nun werden alle reduzierbaren Endknoten behandelt:
03168   if (searchflag == Reduce || searchflag == Both) {
03169
03170     node_table_reset_cursor(reducible_nodes);
03171     current_node = node_table_iter(reducible_nodes);
03172     while (current_node != NULL) {
03173
03174       DEBUG_PRINTF("  Betrachte reduzierbaren Endknoten %ld\n",
03175           current_node->content);
03176
03177       /* Falls der aktuelle Knoten zuvor innerer Knoten im Graph war, kann
03178        * er noch einen alten und somit jetzt ungültigen Präfixvektor
03179        * enthalten. Dieser muss erst gelöscht und neu initialisiert werden:
03180        */
03181       memset(current_node->endnodes, 0,
03182           LOOKAHEAD_LENGTH * sizeof(struct node_t*));
03183       current_node->prefixes = 0;
03184       current_node->last_visited = search_id;
03185
03186       /* Assoziiere das Lookahead-Präfix Epsilon mit dem reduzierbaren
03187        * Endknoten:
03188        */
03189       current_node->endnodes[0] = current_node;
03190       current_node->prefixes = 1;
03191
03192       /* Füge jeden Vorgängerknoten von current_node zur Suchmenge hinzu,
03193        * sofern er sich dort noch nicht befindet:
03194        */
03195       node_table_reset_cursor(current_node->predecessors);
03196       pred_node = node_table_iter(current_node->predecessors);
03197       while (pred_node != NULL) {
03198         node_table_add(search_nodes, pred_node);
03199         pred_node = node_table_iter(current_node->predecessors);
03200       }
03201
03202       // Nächster reduzierbarer Knoten für folgenden Schleifendurchlauf:
03203       current_node = node_table_iter(reducible_nodes);
03204
03205     } // while (current_node != NULL) // reducible_nodes
03206
03207   } // if (searchflag == Reduce || searchflag == Both)
03208
03209   /* Falls der Lookahead-String nicht durch die rechte Seite eines
03210    * Endknotens allein hergeleitet werden konnte: Behandle die
03211    * Vorgängerknoten.
03212    */
03213   while (node_table_empty(search_nodes) == 0) {
03214     current_node = node_table_pop(search_nodes);
03215     current_item = items[current_node->content];
03216     ok = 0;
03217
03218     /* Itemknoten:
03219      * Falls der aktuell untersuchte Knoten ein Item repräsentiert, so muss
03220      * getestet werden, ob die rechte Seite dieses Items etwas zur
03221      * Herleitung des Lookaheads beitragen kann.
03222      * Das Item kann nur die Form [A -> alpha . B beta] haben, mit B ein
03223      * Nichtterminalsymbol.
```

```
03224      */
03225     if (current_node->content >= 0) {
03226       DEBUG_PRINTF("  Betrachte Itemknoten %ld\n",
03227         current_node->content);
03228
03229       /* Falls des Knoten in diesem Suchlauf nicht schon einmal besucht
03230        * wurde, hat er noch keinen oder einen ungültigen Präfixvektor.
03231        * Sollte er in diesem Suchlauf schon besucht worden sein, wird sein
03232        * vorhandener Präfixvektor nur ergänzt.
03233        */
03234       if ( current_node->last_visited < search_id )  {
03235         memset(current_node->endnodes, 0,
03236             LOOKAHEAD_LENGTH * sizeof(struct node_t*));
03237         current_node->last_visited = search_id;
03238       }
03239
03240       /* Der Itemknoten v = [A -> alpha . B beta] hat genau einen
03241        * Nachfolgerknoten. Dieser repräsentiert das Nichtterminalsymbol B.
03242        * Für jeden Eintrag u' im Präfixvektor von B wird geprüft,
03243        * ob u' mithilfe der rechten Seite \beta von v zu einem Präfix
03244        * mit Länge >= |u'| von u verlängert werden kann.
03245        * Der Präfixvektor von B existiert auf jeden Fall und ist auch
03246        * garantiert aktuell, da v erst nach Besuch von B in die Suchmenge
03247        * aufgenommen wurde!
03248        */
03249       node_table_reset_cursor(current_node->successors);
03250       struct node_t *nt_node = node_table_iter(current_node->successors);
03251
03252       /* Wenn current_node als innerer Knoten keinen Nachfolger nt_node
03253        * hat, liegt ein Programmierfehler vor:
03254        */
03255       assert(nt_node != NULL);
03256
03257       current_node->prefixes = nt_node->prefixes;
03258
03259       /* Prüfe jeden der LOOKAHEAD_LENGTH Einträge im Präfixvektor von
03260        * nt_node auf Verlängerbarkeit:
03261        */
03262       for (i=0; i<LOOKAHEAD_LENGTH; i++) {
03263
03264         /* Ist der Eintrag mit Index i im Präfixvektor von B verlängerbar?
03265          */
03266         if (nt_node->endnodes[i] != NULL) {
03267           DEBUG_PRINTF("    Prüfe Präfixeintrag Nr. %d\n", i);
03268
03269           /* Teste jeden String u' in FIRST_k(beta) auf seinen Beitrag: */
03270           for (j=0; j < current_item.first_k_count; j++) {
03271
03272             DEBUG_PRINTF("    Vergleiche mit aktuellem FIRST_k-String: ");
03273             for (k=0; (k+i) < LOOKAHEAD_LENGTH; k++) {
03274               DEBUG_PRINTF("%ld, ", current_item.first_k_item[j][k]);
03275             }
03276             DEBUG_PRINTF("\n");
03277
03278             /* Vergleiche den aktuellen String mit dem Suffix des
03279              * Lookahead:
03280              */
03281             for (k=0; (k+i) < LOOKAHEAD_LENGTH; k++) {
03282               if ( current_item.first_k_item[j][k] !=
03283                   lookahead[( (phase_number + i + k) % LOOKAHEAD_LENGTH)] )
03284                 break;
03285             }
03286
03287             /* Falls die for-Schleife über k bis (k+i) == LOOKAHEAD_LENGTH
03288              * durchläuft, wurde der Lookaheadstring mithilfe der rechten
03289              * Seite des aktuellen Knotens vollständig hergeleitet.
03290              * Somit ist die Suche abgeschlossen!
```

```
03291             */
03292            if ( (k+i) == LOOKAHEAD_LENGTH ) {
03293              DEBUG_PRINTF("    Suche erfolgreich beendet!\n");
03294              return nt_node->endnodes[i];
03295            }
03296
03297            /* Ansonsten wurde entweder eine Stelle gefunden, in der u und
03298             * u' voneinander abweichen, oder u' endete vorzeitig, d.h.
03299             * current_item.first_k_item[j][k] == 0.
03300             */
03301            else {
03302              /* Falls i + k < |u| und die beiden Strings stimmten in den
03303               * ersten i+k Zeichen überein und u' endete vorzeitig, so
03304               * kann der bisher hergeleitete Teilstring des Lookaheads
03305               * noch forgesetzt werden.
03306               * Es wird ein Eintrag in den Präfixvektor des aktuellen
03307               * Knotens aufgenommen.
03308               * Dies deckt auch den Fall an, dass durch den aktuellen
03309               * String aus FIRST_k(beta) nur Epsilon hergeleitet wurde,
03310               * die Schleife also schon bei k=0 terminierte.
03311               */
03312              if (current_item.first_k_item[j][k] == 0) {
03313                DEBUG_PRINTF(
03314                    "    FIRST_k-Eintrag verbraucht => Muss Vorgänger besuchen!\
    n");
03315                current_node->endnodes[k+i] = nt_node->endnodes[i];
03316                ok = 1;
03317              }
03318
03319              /* Ansonsten gibt es an der i+k+1-sten Stelle von u einen
03320               * Unterschied zwischen u und dem bisher hergeleiteten
03321               * Präfix, sodass dieses bei der aktuellen Suche verworfen
03322               * werden kann!
03323               * In diesem Fall muss nichts unternommen werden. Sofern
03324               * vorhanden, wird der nächste String auf FIRST_k(beta)
03325               * betrachtet.
03326               */
03327
03328            } // if ( (k+1) == LOOKAHEAD_LENGTH ) ... else ...
03329
03330          } // for (j=0; ...)
03331
03332        } // if (nt_node->endnodes[i] != NULL) ...
03333
03334      } // for (i=0; ...
03335
03336      /* Falls der aktuelle Knoten nach aktueller Kenntnis nicht zu einer
03337       * akzeptierenden Berechnung führen kann, darf ich ihn dennoch zur
03338       * Zeit nicht löschen, da er evtl. später in derselben Suche
03339       * aufgrund eines Kreises nochmal besucht werden könnte!
03340       */
03341
03342      /* Falls der aktuelle Knoten zu einer akzeptierenden Berechnung
03343       * führen kann: Füge jeden Vorgängerknoten von current_node zur
03344       * Suchmenge hinzu, sofern er sich dort noch nicht befindet:
03345       */
03346      if (ok != 0) {
03347        node_table_reset_cursor(current_node->predecessors);
03348        pred_node = node_table_iter(current_node->predecessors);
03349        while (pred_node != NULL) {
03350          node_table_add(search_nodes, pred_node);
03351          pred_node = node_table_iter(current_node->predecessors);
03352        }
03353
03354        DEBUG_PRINTF("    Suchknoten: ");
03355        NODE_TABLE_PRINT(search_nodes);
03356
```

```
03357       } // end if (ok != 0)
03358
03359     } // if (current_node->content >= 0) // Itemknoten
03360
03361     /* Nichtterminalknoten:
03362      * Falls der aktuell untersuchte Knoten ein Nichtterminalsymbol
03363      * repräsentiert, so muss sein Präfixvektor um die Einträge aus
03364      * den Präfixvektoren all seiner Nachfolger ergänzt werden.
03365      */
03366     else {
03367       DEBUG_PRINTF("  Betrachte Nichtterminalknoten %ld\n",
03368         ~(current_node->content));
03369
03370       /* Falls des Knoten in diesem Suchlauf nicht schon einmal besucht
03371        * wurde, hat er noch keinen oder einen ungültigen Präfixvektor und
03372        * auch evtl. einen ungültigen Eintrag im Feld prefixes.
03373        * Sollte er in diesem Suchlauf schon besucht worden sein, wird sein
03374        * vorhandener Präfixvektor nicht gelöscht, sondern im
03375        * Folgenden nur ergänzt.
03376        */
03377       if ( current_node->last_visited < search_id )  {
03378         memset(current_node->endnodes, 0,
03379             LOOKAHEAD_LENGTH * sizeof(struct node_t*));
03380         current_node->prefixes = 0;
03381         current_node->last_visited = search_id;
03382       }
03383
03384       node_table_reset_cursor(current_node->successors);
03385       struct node_t *succ_node =
03386         node_table_iter(current_node->successors);
03387
03388       /* Für jeden Nachfolger v' von des Nichtterminalknotens
03389        * current_node, der während des aktuellen Suchlaufs schon besucht
03390        * wurde: Ergänze den Präfixvektor von current_node um die
03391        * Einträge im Präfixektor von v'.
03392        */
03393       while (succ_node != NULL) {
03394         if ( succ_node->last_visited == search_id) {
03395           for (i=0; i<LOOKAHEAD_LENGTH; i++) {
03396             if (succ_node->endnodes[i] != NULL) {
03397               current_node->endnodes[i] = succ_node->endnodes[i];
03398               current_node->prefixes = current_node->prefixes | (1 << i);
03399               ok = 1;
03400             }
03401           }
03402         } // if ( succ_node->last_visited == search_id )
03403
03404         succ_node = node_table_iter(current_node->successors);
03405       } // end while (succ_node != NULL)
03406
03415       /* Falls es mindestens einen Präfixeintrag zu übertragen gab, kann
03416        * der Knoten zu einer akzeptierenden Berechung beitragen. Jeder
03417        * Vorgängerknoten kann zur Suchmenge hinzugefügt werden, sofern er
03418        * sich dort noch nicht befindet UND sofern der Präfixvektor von
03419        * current_node sich seit dem letzten Besuch des Knotens verändert
03420        * hat.
03421        */
03422       if (ok != 0) {
03423         node_table_reset_cursor(current_node->predecessors);
03424         pred_node = node_table_iter(current_node->predecessors);
03425         while (pred_node != NULL) {
03426           DEBUG_PRINTF("    weitersuchen in %ld?", pred_node->content);
03427           if ( (pred_node->prefixes != current_node->prefixes)
03428               || (pred_node->last_visited != search_id) ) {
03429             pred_node->prefixes = current_node->prefixes;
03430             node_table_add(search_nodes, pred_node);
03431             DEBUG_PRINTF(" Ja\n");
```

```
03432           }
03433           else {
03434             DEBUG_PRINTF(" Nein\n");
03435           }
03436           pred_node = node_table_iter(current_node->predecessors);
03437         } // while (pred_node != NULL)
03438         DEBUG_PRINTF("    Suchknoten: ");
03439         NODE_TABLE_PRINT(search_nodes);
03440       } // if (ok != 0)
03441
03442     } // if (current_node->content >= 0) ... else // Nichtterminalknoten
03443
03444   } // while (node_tabe_empty(search_nodes) == NULL)
03445
03446
03447   /* Wenn die Funktionsausführung an diese Stelle gelangt, konnte der
03448    * Lookahead-String nicht hergeleitet werden:
03449    */
03450   return NULL;
03451 } // struct node_t *rtsearch
```

5.1.4.40 static void test_new_endnode (struct node_t ** *node*) [static]

Hilfsfunktion, um einen neu entstandenen Endknoten nach einem Reduktions-/Leseschritt zu überprüfen. Der Aufruf `test_new_endnode(node)` prüft, ob der neue Endknoten node expandierbar, reduzierbar oder lesbar ist und trägt ihn in die entsprechende Menge ein.

Parameter:

node Zeiger auf einen Zeiger (Call by Reference) auf den zu untersuchenden Knoten.

```
03739 {
03740
03741   /* Falls der neue Endknoten das Item [S' -> S.] repräsentiert, darf er
03742    * nur im Graph verbleiben, falls die Eingabe verbraucht ist. In diesem
03743    * Falle darf er allerdings nicht in die Menge der reduzierbaren Knoten
03744    * eingetragen werden!
03745    */
03746   if ( (*node)->content == acceptitem_number ) {
03747     if (lookahead[ (phase_number) % LOOKAHEAD_LENGTH ] == 0) {
03748       DEBUG_PRINTF("    Accept-Item gefunden!\n");
03749       return;
03750     }
03751     /* Wurde der Knoten für das Accept-Item erzeugt, bevor die gesamte
03752      * Eingabe verbraucht ist, kann er wieder aus dem Graphen entfernt
03753      * werden. Da er mit keinem anderen Knoten verbunden ist und in keine
03754      * andere Knotenmenge als graph eingetragen wurde, kann die Löschung
03755      * in diesem Fall ohne Aufruf von graph_adjust erfolgen.
03756      */
03757     else {
03758       DEBUG_PRINTF("    Verfrühtes Accept-Item gefunden!\n");
03759       list_remove(&graph, (*node));
03760       node_table_free((*node)->predecessors);
03761       node_table_free((*node)->successors);
03762       free(*node);
03763       *node = NULL;
03764       return;
03765     }
03766   } // if (acceptitem_number)
03767
03768   /* Falls der neue Knoten expandierbar ist, muss er für den
03769    * nächsten Expansionsschritt vorgemerkt werden:
03770    */
```

```
03771   if ( items[(*node)->content].dot_symbol_type == Nonterminal ) {
03772     DEBUG_PRINTF(
03773         "    Modifizierter/neuer Endknoten %ld ist expandierbar.\n",
03774         (*node)->content);
03775     node_table_add(expandable_nodes, *node);
03776   } // end if ( ... == Nonterminal )
03777
03778   /* Ansonsten ist der Knoten lesbar oder reduzierbar: */
03779   else {
03780
03781     /* Falls der neue Knoten reduzierbar ist, muss er für eine
03782      * Wiederholung des aktuellen Reduktions-/Leseschrittes
03783      * vorgemerkt werden:
03784      */
03785     if ( items[(*node)->content].dot_symbol_type == Epsilon ) {
03786       DEBUG_PRINTF(
03787           "    Modifizierter/neuer Endknoten %ld ist reduzierbar.\n",
03788           (*node)->content);
03789       node_table_add(reducible_nodes, *node);
03790     } // end if ( ... == Epsilon )
03791
03792     /* Ansonsten ist der Knoten lesbar: */
03793     else {
03794
03795       /* Wenn das nächste Lookahead-Symbol hinter dem Punkt steht,
03796        * wird der Knoten beim nächsten Leseschritt mit berücksichtigt.
03797        */
03798       if ( (items[(*node)->content].dot_symbol_type == Terminal)
03799         && (items[(*node)->content].dot_symbol == lookahead[
03800             (phase_number) % LOOKAHEAD_LENGTH])
03801          ) {
03802         DEBUG_PRINTF(
03803             "    Modifizierter/neuer Endknoten %ld ist lesbar.\n",
03804             (*node)->content);
03805         list_add(&new_readable_nodes, *node);
03806       }
03807
03808       /* Falls der Knoten lesbar ist, aber mit falschen Symbol hinter dem
03809        * Punkt, kann er nicht zu einer akzeptierenden Berechnung
03810        * korrespondieren und wird aus dem Graphen entfernt.
03811        */
03812       else {
03813         DEBUG_PRINTF(
03814             "    Modifizierter/neuer Endknoten %ld ist lesbar mit falschem Symbol
      .\n",
03815             (*node)->content);
03816         graph_adjust(node);
03817       }
03818
03819     } // end if ( ... == Epsilon ) else
03820
03821   } // end if ( ... == Nonterminal ) else
03822
03823 } // void test_new_endnode
```

5.1.4.41 static void treat_predecessors (struct node_t ** *nt_node*) [static]

Hilfsfunktion, um die Vorgängerknoten eines reduzierten Endknotens zu behandeln. Der Aufruf `treat_predecessors(nt_node)` erfolgt nach der Reduktion eines reduzierbaren Endknotens w. *nt_node* ist dabei der Nichtterminalknoten, der im Graph unmittelbarer Vorgänger von w war. Für jeden Vorgänger von *nt_node* wird nun überprüft, ob er sich als neuer Endknoten eignet.

Parameter:

nt_node Zeiger auf einen Zeiger (Call by Reference) auf den Nichtterminalknoten, aus dessen Vorgängern die neuen Endknoten des Graphen entstehen sollen.

```
03476 {
03477   // Zeiger auf den aktuell bearbeiteten Vorgängerknoten:
03478   struct node_t *pred_node = NULL;
03479
03480   /* Es sind Fälle möglich, in denen während der Vorgängerbehandlung
03481    * erneut Reduktionen stattfinden, die dann auch eine Wiederholung der
03482    * Vorgängerbehandlung implizieren. Daher wird die Bearbeitung von
03483    * Vorgängerknoten so oft iteriert, bis *nt_node nicht mehr auf einen
03484    * Nichtterminalknoten zeigt.
03485    */
03486   while ( (*nt_node) != NULL ) {
03487
03488     /* Falls nt_node keine anderen Nachfolger im Graphen als w hat, sind
03489      * die Vorgängerknoten von nt_node die neuen Endknoten des Graphen.
03490      * Ihr Punkt wird um eine Stelle nach rechts veschoben, und sie werden
03491      * darauf überprüft, ob sie Teil einer akzeptierenden Berechnung
03492      * sein können:
03493      */
03494     if ( node_table_empty((*nt_node)->successors) != 0 ) {
03495
03496       DEBUG_PRINTF("  nt_node %ld hat keine weiteren Nachfolger!\n",
03497           (*nt_node)->content);
03498       while ( node_table_empty((*nt_node)->predecessors) == 0 ) {
03499
03500         /* Betrachte den nächsten Vorgängerknoten v und entferne seine
03501          * Kante zu \c nt_node:
03502          */
03503         pred_node = node_table_pop((*nt_node)->predecessors);
03504         node_table_remove(pred_node->successors, (*nt_node));
03505
03506         /* Es ist möglich, dass v zu einem früheren Zeitpunkt der aktuellen
03507          * Phase schon einmal ein expandierbarer Endknoten war, von dem
03508          * aus eine Expansion durchgeführt wurde. Dabei wurde v in die
03509          * Menge expanded_nodes eingetragen. Da v nach dem Verschieben des
03510          * Punktes aber nicht mehr derselbe Knoten wie damals ist, muss
03511          * er jetzt aus expanded_nodes ausgetragen werden:
03512          */
03513         node_table_remove(expanded_nodes, pred_node);
03514
03515         /* Verschiebe den Punkt in v und überprüfe den Knoten: */
03516         pred_node->content = pred_node->content + 1;
03517         test_new_endnode(&pred_node);
03518
03519       } // end while ( node_table_empty((*nt_node)->predecessors) == 0)
03520
03521       /* Nachdem alle Vorgängerknoten behandelt wurden, kann auch nt_node
03522        * entfernt werden. Falls anhand dieses Symbols in der aktuellen Phase
03523        * bereits expandiert wurde, muss der Knoten zuvor auch aus der Liste
03524        * expanded_nonterminals ausgetragen werden.
03525        */
03526
03527       DEBUG_PRINTF("  Entferne erledigten Nichtterminalknoten %ld ...",
03528           (*nt_node)->content);
03529       node_table_remove(expanded_nonterminals, (*nt_node));
03530       node_table_free((*nt_node)->predecessors);
03531       node_table_free((*nt_node)->successors);
03532       free(*nt_node);
03533       *nt_node = NULL;
03534       DEBUG_PRINTF(" erledigt!\n");
03535
03536     } // if ( node_table_empty((*nt_node)->successors) != 0 )
03537
```

```
03538     /* Sollte nt_node noch andere Nachfolger als w haben, müssen Kopien der
03539      * Vorgänger von A angefertigt werden, aus denen dann die neuen
03540      * Endknoten entstehen:
03541      */
03542     else {
03543
03544       DEBUG_PRINTF(
03545           "  NT-Knoten %ld hat Nachfolger => Kopiere Vorgängerknoten!\n",
03546           (*nt_node)->content);
03547       /* Erzeuge für jeden Knoten v in der Vorgängermenge von nt_node eine
03548        * Kopie v' inklusive aller in v eingehenden Kanten, aber ohne die
03549        * ausgehenden Kanten.
03550        * Bewege dann in jeder Knotenkopie v' den Punkt um eine Stelle nach
03551        * rechts und überprüfe v' mit test_new_endnode.
03552        */
03553       node_table_reset_cursor((*nt_node)->predecessors);
03554       struct node_t *copy_node = node_table_iter((*nt_node)->predecessors);
03555       while ( copy_node != NULL ) {
03556         /* Erzeuge eine Kopie v' des Knotens copy_node mit um eine Stelle
03557          * verschobenem Punkt. Der (einzige) direkte Vorgänger von v
03558          * wird zugleich zum Vorgänger von v'.
03559          * Sollte v und somit auch v' keinen Vorgänger haben, muss v'
03560          * in die Liste der Startknoten des Graphen eingetragen werden.
03561          */
03562         DEBUG_PRINTF("  Kopiere Vorgänger %ld\n", copy_node->content);
03563         pred_node = create_node(copy_node->content + 1);
03564         if ( node_table_empty(copy_node->predecessors) != 0 ) {
03565           list_enqueue(&graph, pred_node);
03566         }
03567         else {
03568           DEBUG_PRINTF("  Verbinde neuen Knoten mit Vorgängern ...");
03569           pred_node->predecessors =
03570             node_table_copy(copy_node->predecessors);
03571           /* Der neue Itemknoten pred_node hat jetzt genau einen
03572            * Vorgängerknoten, einen Nichtterminalknoten pred_nt.
03573            * In dessen Nachfolgerliste muss pred_node noch eintragen
03574            * werden:
03575            */
03576           node_table_reset_cursor(copy_node->predecessors);
03577           struct node_t *pred_nt =
03578             node_table_iter(copy_node->predecessors);
03579           node_table_add(pred_nt->successors, pred_node);
03580           DEBUG_PRINTF(" erledigt!\n");
03581         }
03582
03583         // Ueberprüfe den neuen Knoten v':
03584         DEBUG_PRINTF("  Teste neuen Knoten!\n");
03585         test_new_endnode(&pred_node);
03586
03587         copy_node = node_table_iter((*nt_node)->predecessors);
03588       } // while ( copy_node != NULL)
03589
03590       /* Der Zeiger nt_node muss zurückgesetzt werden, da ansonsten auch
03591        * dann die äussere Schleife wiederholt wird, wenn im Folgenden
03592        * keine erneute Reduktion stattfindet.
03593        */
03594       *nt_node = NULL;
03595     } // if ( node_table_empty((*nt_node)->successors) != 0 ) ... else
03596
03597     /* Nun sind alle Kandidaten für neue Endknoten kategorisiert in
03598      * lesbare, reduzierbare, expandierbare und irrelevante Knoten.
03599      * Neue lesbare Endknoten wurden von der Funktion test_new_endnode
03600      * in die Liste new_readable_endnodes eingetragen. Diese Liste
03601      * kann jetzt in die Tabelle readable_endnodes überführt werden.
03602      */
03603     struct node_t *read_node = NULL;
03604     while (new_readable_nodes != NULL) {
```

```
03605       read_node = list_pop(&new_readable_nodes);
03606       node_table_add(readable_nodes, read_node);
03607     }
03608     read_node = NULL;
03609
03610     /* Nun muss überprüft werden, ob dem Graph neue reduzierbare Knoten
03611      * hinzugefügt wurden. Sollte dies der Fall sein, müssen sie auf
03612      * Gültigkeit getestet und ggf. reduziert werden, um vor dem
03613      * nächsten Expansionsschritt die Einhaltung der Invariante
03614      * sicherzustellen.
03615      */
03616     if ( node_table_empty(reducible_nodes) == 0 ) {
03617
03618       /* Suche in der Menge der reduzierbaren Endknoten nach einem
03619        * Knoten, für die ein passender Pfad existiert.
03620        */
03621       struct node_t *suitable_node = rtsearch(Reduce);
03622
03623       /* Zeiger für Knoten, die evtl. gelöscht werden müssen: */
03624       struct node_t *kill_node = NULL;
03625
03626       /* Sollte kein Knoten gefunden werden, für den ein passender Pfad
03627        * existiert, können die neu erzeugten reduzierbaren Knoten nicht
03628        * zu einer akzeptierenden Berechung führen. Sie können daher aus
03629        * dem Graph entfernt werden.
03630        */
03631       if ( suitable_node == NULL ) {
03632         DEBUG_PRINTF(
03633             "  Kein gültiger reduzierbarer Endknoten gefunden!\n");
03634
03635         // Alle reduzierbaren Endknoten werden entfernt:
03636         while (node_table_empty(reducible_nodes) == 0) {
03637           kill_node = node_table_pop(reducible_nodes);
03638           graph_adjust(&kill_node);
03639         }
03640       } // if ( suitable_node == NULL )
03641
03642       else {
03643
03644         /* Sollte es einen reduzierbaren Endknoten w mit gültigem Pfad
03645          * geben, so muss die Reduktion dieses Knotens durchgeführt und die
03646          * Vorgängerbehandlung wiederholt werden, bevor zum nächsten
03647          * Expansionsschritt übergegangen werden kann.
03648          */
03649         if ( items[suitable_node->content].dot_symbol_type == Epsilon ) {
03650           DEBUG_PRINTF(
03651               "  Gültiger reduzierbarer Endknoten %ld wurde gefunden!\n",
03652               suitable_node->content);
03653
03654           /* Vermerke die Reduktion im Wald der partiellen Ableitungsbäume:
03655            */
03656           PARSE_FOREST_REDUCE(&parse_forest, suitable_node->content);
03657
03658           // Alle lesbaren Endknoten müssen entfernt werden:
03659           while (node_table_empty(readable_nodes) == 0) {
03660             kill_node = node_table_pop(readable_nodes);
03661             graph_adjust(&kill_node);
03662           }
03663
03664           // Alle reduzierbaren Endknoten ausser w werden entfernt:
03665           while (node_table_empty(reducible_nodes) == 0) {
03666             kill_node = node_table_pop(reducible_nodes);
03667             if (kill_node == suitable_node)
03668               kill_node = NULL;
03669             else
03670               graph_adjust(&kill_node);
03671           }
```

```
03672
03673           /* Im Unterschied zur Situation bei einer Reduktion innerhalb der
03674            * Funktion reduce_read kann der Graph jetzt auch expandierbare
03675            * Endknoten enthalten, die noch zu löschen sind:
03676            */
03677           while (node_table_empty(expandable_nodes) == 0) {
03678             kill_node = node_table_pop(expandable_nodes);
03679             graph_adjust(&kill_node);
03680           }
03681
03682           /* Betrachte den Nichtterminalknoten A, der in Graph einziger
03683            * direkter Vorgänger von w ist, und entferne die Kante von
03684            * A nach w:
03685            */
03686           *nt_node =
03687             node_table_pop(suitable_node->predecessors);
03688           node_table_remove((*nt_node)->successors, suitable_node);
03689
03690           /* Falls ein Itemknoten mehr als einen Vorgänger hatte,
03691            * liegt ein Programmierfehler vor:
03692            */
03693           assert(node_table_empty(suitable_node->predecessors) != 0);
03694
03695           // Entferne w aus dem Graphen:
03696           node_table_free(suitable_node->predecessors);
03697           node_table_free(suitable_node->successors);
03698           free(suitable_node);
03699           suitable_node = NULL;
03700
03701           /* Nun müssen noch in einem erneuten Durchlauf der äusseren
03702            * while-Schleife die bisherigen Vorgängerknoten von A
03703            * behandelt werden, aus denen wieder neue Endknoten entstehen.
03704            */
03705
03706         } // if (Epsilon)
03707
03708         /* Ansonsten ist das Suchergebnis ungültig, denn wenn die
03709          * Suche auf lesbare Endknoten eingeschränkt wurde, darf das
03710          * Ergebnis nur ein Knoten mit dot_symbol_type == Epsilon sein!
03711          */
03712         else {
03713           fprintf(stderr, "treat_predecessors: Falscher Knoten wurde gefunden!\n"
     );
03714           abort();
03715         } // if ( Epsilon ) ... else
03716
03717       } // if ( suitable_node == NULL ) ... else
03718
03719     } // if ( node_tabe_empty(reducible_nodes) == 0 )
03720
03721   } // while ( (*nt_node) != NULL)
03722
03723 } // void treat_predecessors
```

5.1.4.42 static void* xmalloc (size_t *size*) [static]

Abfangen von fehlgeschlagenen Speicheranforderungen. Wird nur übersetzt, wenn das Makro DEBUG_ALL definiert ist.

Parameter:

size Größe des anzufordernden Speicherbereichs.

Rückgabe:

Zeiger auf den alloziierten Speicherbereich.

```
00059 {
00060   void *ptr = malloc(size);
00061
00062   if (ptr == NULL)
00063     abort();
00064
00065   return ptr;
00066 } // xmalloc (size_t size)
```

5.1.4.43 static int yyparse (void) [static]

Rahmenprozedur für den Parser. Die Funktion `yyparse` erwartet, dass eine Funktion `yylex` vorhanden ist, die bei jedem Aufruf ein Token in seiner Integerrepräsentation liefert. Für eine Eingabefolge x prüft `yyparse`, ob x Teil der Sprache L(G) ist, die durch die Grammatik G beschrieben wird, die dem Parser zugrunde liegt.

Rückgabe:

`0`, falls $x \in L(G)$; `1`, falls die Eingabe fehlerhaft ist (wie Bison).

Noch zu erledigen

Ist eine clean-Routine auch für den Ableitungswald nötig? Eigentlich nicht, weil diese Funktionalität ja nur zu Diagnosezwecken eingebaut wurde und im Produktiveinsatz zwecks Zeitersparnis nicht aktiviert würde.

```
04135 {
04136   DEBUG_PRINTF("Betrete Funktion yyparse\n");
04137 #ifdef PARSE_TREE
04138   atexit(End);
04139 #endif
04140   // Initialisierung der globalen Knotentabellen:
04141   DEBUG_PRINTF("Initialisiere globale Knotentabellen... ");
04142   expandable_nodes      = node_table_init();
04143   readable_nodes        = node_table_init();
04144   reducible_nodes       = node_table_init();
04145   expanded_nodes        = node_table_init();
04146   expanded_nonterminals = node_table_init();
04147   DEBUG_PRINTF("erledigt!\n");
04148
04149   unsigned long i; // Universelle Laufvariable
04150
04151   // Flag für Erfolg eines Reduktions-/Leseschrittes
04152   int reduce_read_ok = 0;
04153
04154   DEBUG_PRINTF("Fülle Lookahead-Array... ");
04155   /* Fülle den Eingabepuffer mit den ersten LOOKAHEAD_LENGTH Token,
04156    * die yylex liefert.
04157    * Sollte die Eingabe vorher enden, liefert yylex für alle weiteren
04158    * Aufrufe den Wert 0.
04159    */
04160   for (i = 0; i < LOOKAHEAD_LENGTH; i++) {
04161     lookahead[i] = call_lexer();
04162   }
04163   DEBUG_PRINTF("erledigt!\n");
04164
04165   DEBUG_PRINTF("Initialisiere Graph...\n");
```

```
04166   /* Zu Beginn der Programmausführung ist das einzige Element des Graphen
04167    * der Knoten für das expandierbare Item [S' -> .S]_0.
04168    */
04169   DEBUG_PRINTF("  Erzeuge Knoten für Startitem [S' -> .S]\n");
04170   struct node_t *startnode =
04171     create_node((signed long) startitem_number);
04172   DEBUG_PRINTF("  Trage Startknoten in Graph ein\n");
04173   list_add (&graph, startnode);
04174   DEBUG_PRINTF("  Trage Startknoten in Tabelle expandierbarer Knoten ein\n");
04175   node_table_add(expandable_nodes, startnode);
04176   DEBUG_PRINTF("...erledigt!\n");
04177
04178 #ifdef PARSE_TREE
04179   node_number = 0;
04180   printf("digraph {");
04181 #endif
04182
04183   /* Iteriere abwechselnd Expansions- und Reduktions-/Leseschritte, bis
04184    * alle Zeichen der Eingabe verbraucht sind oder in einem Reduktions-/
04185    * Leseschritt vorzeitig erkannt wird, dass die Eingabe ungültig ist.
04186    * Dabei gilt die Eingabe als verbraucht, sobald ein Token mit dem Wert
04187    * 0 gefunden wird.
04188    */
04189   DEBUG_PRINTF("Betrete Parserschleife!\n\n");
04190   reduce_read_ok = 0;
04191   while ( (lookahead[phase_number % LOOKAHEAD_LENGTH] > 0) &&
04192       (reduce_read_ok == 0) ) {
04193     DEBUG_PRINTF("\n");
04194
04195     DEBUG_PRINTF("Phase %2ld: Lookahead = ", (phase_number + 1));
04196     for (i=0; i<LOOKAHEAD_LENGTH; i++) {
04197       DEBUG_PRINTF("%ld, ",
04198         lookahead[ (phase_number+i) % LOOKAHEAD_LENGTH ]);
04199     }
04200     DEBUG_PRINTF("\n");
04201
04202     DEBUG_PRINTF("Phase %2ld: Expansionsschritt\n", (phase_number + 1));
04203     expand();
04204
04205     DEBUG_PRINTF("Phase %2ld: Reduktions-/Leseschritt\n",
04206       (phase_number + 1));
04207     reduce_read_ok = reduce_read();
04208   } // while (lookahead[phase_number % LOOKAHEAD_LENGTH])
04209
04210   /* Teste, ob die Parserschleife regulär beendet wurde oder aufgrund
04211    * eines fehlgeschlagenen Reduktions-/Leseschrittes:
04212    */
04213   if (reduce_read_ok == 0) {
04214     /* Falls die Eingabe verbraucht ist und der Graph das Item [S' -> S.]_n
04215      * enthält: akzeptiere.
04216      * Sonst: Beende mit Fehler.
04217      *
04218      * Wenn der Parsevorgang erfolgreich war, wurde eine der
04219      * Zusammenhangskomponenten von \G vollständig auf diesen einen Knoten
04220      * reduziert. Es genügt also, die Liste aller Anfangsknoten des Graphen
04221      * nach diesem Knoten zu durchsuchen!
04222      */
04223     if (list_contains_node_num(graph, acceptitem_number)) {
04224 #ifndef SUPPRESS_OUTPUT
04225       fprintf(stderr,
04226           "\nDie Eingabe wurde erfolgreich geparst! (%ld Token)\n",
04227           phase_number);
04228 #endif
04229       clean();
04230
04231 /* Drucke den finalen Ableitungsbaum nur dann aus, wenn nicht schon am
04232  * Ende jeder Phase der aktuelle Wald ausgegeben wird:
```

```
04233  */
04234 #ifndef PARSE_TREE_PHASES
04235       PARSE_FOREST_PRINT(parse_forest);
04236 #endif
04237
04242        return 0;
04243     }
04244     else {
04245       fprintf(stderr,
04246           "\nFEHLER bei Symbol Nr. %ld: Die Eingabe ist ungültig!\n",
04247         phase_number);
04248       clean();
04249       return 1;
04250     }
04251   }
04252   else {
04253     fprintf(stderr,
04254         "\nVorzeitiger Abbruch der Parserschleife bei Symbol %ld: %ld!\n",
04255         phase_number,
04256         lookahead[ (phase_number+i) % LOOKAHEAD_LENGTH ]);
04257     return 1;
04258   }
04259
04260 } // static int yyparse (void)
```

5.1.5 Variablen-Dokumentation

5.1.5.1 signed long const acceptitem_number = 1 [static]

Reservierte Item-Nummer. Der Item-Index 1 ist reserviert für das Item [S' -> S.], das das Ende des Parse-Vorgangs markiert.

5.1.5.2 int eof_reached = 0 [static]

Flag, ob das Ende der Eingabe erreicht ist. `eof_reached == 0`, solange `yylex` noch Eingabetoken > 0 liefert. `eof_reached == 1`, sonst.

5.1.5.3 struct node_table_t* expandable_nodes = NULL [static]

Hashtafel für expandierbare Endknoten.

5.1.5.4 struct node_table_t* expanded_nodes = NULL [static]

Hashtafel bereits expandierter Knoten. Um den Aufwand bei den Expansionsschritten zur verringern, wird in jeder Phase Buch darüber geführt, welche Itemknoten in der aktuellen Phase schon expandiert wurden.

5.1.5.5 struct node_table_t* expanded_nonterminals = NULL [static]

Hashtafel bereits expandierter Nichtterminalknoten. Um den Aufwand bei den Expansionsschritten zur verringern, wird in jeder Phase Buch darüber geführt, welche Nichtterminalknoten in der aktuellen Phase schon bei Expansionen erschaffen wurden.

5.1.5.6 struct node_list_t* graph = NULL [static]

Der Parser verwaltet einen Graph aus Item- und Nichtterminalknoten. Da der Graph aus mehreren Zusammenhangskomponenten bestehen kann, ist die Datenstruktur *graph* ein Zeiger auf eine **Liste** von Anfangsknoten.

5.1.5.7 unsigned long lookahead[LOOKAHEAD_LENGTH] [static]

Puffer für die nächsten k ungelesenen Eingabesymbole. Dabei ist k die Länge des Lookahead, wie sie durch das Makro `LOOKAHEAD_LENGTH` festgelegt ist. Um nach jedem Leseschritt das explizite Verschieben aller restlichen neun Array-Elemente zu vermeiden, werden die Array-Elemente immer zyklisch über den Index `(phase_number % k)` angesprochen.

5.1.5.8 struct node_list_t* new_readable_nodes = NULL [static]

Temporäre Liste für lesbare Endknoten, die nach einem Reduktions-/Leseschritt neu entstanden sind:.

5.1.5.9 unsigned long node_number = 0 [static]

Für die Ausgabe des Parsewaldes: globale Zählvariable für die Knoten. Wird nur compiliert, falls das Makro `PARSE_TREE` definiert ist.

5.1.5.10 struct parse_forest_t* parse_forest = NULL [static]

Der Wald von partiellen Ableitungsbäumen wird über den globalen Zeiger `parse_forest` angesprochen. Wird nur compiliert, falls das Makro `PARSE_TREE` definiert ist.

5.1.5.11 unsigned long phase_number = 0 [static]

Zähler für die aktuelle Phase. Entspricht der Anzahl bisher gelesener Token, da in jeder Phase genau ein Zeichen der Eingabe verbraucht wird.

5.1.5.12 struct node_table_t* readable_nodes = NULL [static]

Hashtafel für lesbare Endknoten.

5.1.5.13 struct node_table_t* reducible_nodes = NULL [static]

Hashtafel für reduzierbare Endknoten.

5.1.5.14 unsigned long search_id = 0 [static]

Laufende Nummer für rückwärts-topologische Suchen. Jeder Suchlauf bekommt eine eindeutige Nummer zugewiesen. In jedem Knoten des Graphen befindet sich ein Feld `last_visited`, das angibt, während welches Suchlaufs der Knoten zuletzt besucht wurde.

5.1.5.15 signed long const startitem_number = 0 [static]

Reservierte Item-Nummer. Der Item-Index 0 ist reserviert für das Item [S' -> .S], das den Beginn des Parse-Vorgangs markiert.

Index