i in Beziehung, die zur $FOLLOW_1$-Menge von i beitragen. Die Funktion g wird hier mithilfe der als berechnet vorausgesetzten ε-*ffi*-Mengen definiert.

Definition 8.3.9 (RLL(1)-Grammatik)
Eine rrkfG $G = (V_N, V_T, p, S)$ heißt **RLL(1)-Grammatik**, wenn für alle erweiterten kontextfreien Items
$[X \to \cdots.(r_1|\cdots|r_n)\cdots]$ gilt:
$FIRST_1(r_i) \oplus_1 FOLLOW_1([X \to \cdots.(r_1|\cdots|r_n)\cdots]) \cap$
$FIRST_1(r_j) \oplus_1 FOLLOW_1([X \to \cdots.(r_1|\cdots|r_n)\cdots]) = \emptyset$ für alle $i \neq j$,
und für alle erweiterten kontextfreien Items $[X \to \cdots.(r)^*\cdots]$ gilt:
$FIRST_1(r) \cap FOLLOW_1([X \to \cdots.(r)^*\cdots]) = \emptyset$ und $eps(r) = false$. □

Sind die $FIRST_1$- und $FOLLOW_1$-Mengen für eine rrkfG berechnet, und ist die RLL(1)-Eigenschaft als erfüllt festgestellt, so kann ein RLL(1)-Parser für die Ausgangsgrammatik erzeugt werden. Dabei bieten sich zwei wesentlich verschiedene Darstellungen des Parsers an. Die erste besteht aus einem für alle Grammatiken festen Treiber und einer zur jeweiligen Grammatik gehörenden Tabelle. Der Treiber indiziert die Tabelle mit dem aktuellen Item und dem nächsten Eingabesymbol, genauer gesagt mit ganzzahligen Codierungen dieser beiden Objekte. Das selektierte Feld in der Tabelle gibt entweder das nächste Item an oder zeigt einen Syntaxfehler an. Die zweite mögliche Darstellung ist die durch ein Programm. Das Programm besteht im wesentlichen aus einer Menge simultan rekursiver Prozeduren, einer pro Nichtterminal. Die Prozedur für das Nichtterminal X ist zuständig für die Analyse von Wörtern für X. Wir beginnen mit der Tabellenversion von RLL(1)-Parsern.

RLL(1)-Parser für rechtsreguläre kontextfreie Grammatiken (Tabellenversion)

Der RLL(1)-Parser ist ein deterministischer Kellerautomat. Die Parsertabelle M stellt eine Abbildung $m : It_G \times V_T \rightsquigarrow It_G \cup \{error\}$ dar. Die Parsertabelle wird dann konsultiert, wenn mittels Vorausschauens in die restliche Eingabe eine Entscheidung getroffen werden muß. Deshalb muß M nur Zeilen enthalten für

- Items, in denen eine Alternative ausgewählt werden muß, und
- Items, in denen eine Iteration bearbeitet werden muß;

d.h. die Abbildung m ist definiert für Items der Form $[X \to \cdots.(r_1|\cdots|r_n)\cdots]$ und der Form $[X \to \cdots.(r)^*\cdots]$.
Der RLL(1)-Parser wird in einer Anfangskonfiguration $(\#[S' \to .S], w\#)$ gestartet. Das aktuelle Item, das oberste auf dem Keller, bestimmt, ob die Parsertabelle konsultiert werden muß oder nicht. Muß die Tabelle befragt werden, so gibt $M[\rho, a]$ – falls nicht *error* – das nächste aktuelle Item für das aktuelle Item ρ und das aktuelle Eingabesymbol a an. Falls $M[\rho, a] = error$ ist, so liegt ein Syntaxfehler vor. In der Konfiguration $(\#[S' \to S.], \#)$ schließlich akzeptiert der Parser die gelesene Eingabe.

Die sonstigen Übergänge sind:
$\delta([X \to \cdots.a\cdots], a) = [X \to \cdots a.\cdots]$
$\delta([X \to \cdots.Y\cdots], \varepsilon) = [X \to \cdots.Y\cdots][Y \to .p(Y)]$
$\delta([X \to \cdots.Y\cdots][Y \to p(Y).], \varepsilon) = [X \to \cdots Y.\cdots]$
Außerdem gäbe es noch einige Übergänge etwa von $[X \to \cdots(\cdots|r_i.|\cdots)\cdots]$ zu $[X \to \cdots(\cdots|r_i|\cdots).\cdots]$, die weder Symbole lesen, noch Nichtterminale expandieren, noch zu Nichtterminalen reduzieren. Sie können schon zur Generierungszeit ein für alle mal ausgeführt werden, indem man die Übergangsfunktion gemäß der folgenden Tabelle modifiziert:
(1) $[X \to \cdots(\cdots|r_i.|\cdots)\cdots]$ ⇒ (2) $[X \to \cdots(\cdots|r_i|\cdots).\cdots]$,
(3) $[X \to \cdots(r.)^*\cdots]$ ⇒ (4) $[X \to \cdots.(r)^*\cdots]$
(5) $[X \to \cdots.(r_1\cdots r_n)\cdots]$ ⇒ (6) $[X \to \cdots(.r_1\cdots r_n)\cdots]$

Führt ein Übergang von δ zu (1), so wird er in das erweiterte kontextfreie Item (2) überführt. Führt er zu (3), so wird er in (4) geführt und aus (5) direkt in (6).

Nun folgt der Algorithmus zur Erzeugung der RLL(1)-Parsertabellen.

Algorithmus RLL(1)-GEN

Eingabe: RLL(1)-Grammatik G, $FIRST_1$ und $FOLLOW_1$ für G.

Ausgabe: Parsertabelle M für RLL(1)-Parser für G.

Methode: Für alle Items der Form $[X \to \cdots.(r_1|\cdots|r_n)\cdots]$ setze
$M([X \to \cdots.(r_1|\cdots|r_n)\cdots], a) = [X \to \cdots(\cdots|.r_i|\cdots)\cdots]$, für
$a \in FIRST_1(r_i)$ und falls zusätzlich $\varepsilon \in FIRST_1(r_i)$ dann auch für $a \in FOLLOW_1([X \to \cdots.(r_1|\cdots|r_n)\cdots])$.
Für alle Items der Form $[X \to \cdots.(r)^*\cdots]$ setze
$M([X \to \cdots.(r)^*\cdots], a) =$

$$\begin{cases} [X \to \cdots(.r)^*\cdots] & \text{falls } a \in FIRST_1(r) \\ [X \to \cdots(r)^*.\cdots] & \text{falls } a \in FOLLOW_1([X \to \cdots.(r)^*\cdots]) \end{cases}$$

Setze alle noch nicht gefüllten Einträge auf *error*.

Beispiel 8.3.15 (Fortführung von Beispiel 8.3.10)
Die Parsertabelle zur Grammatik G_e. (Aus Umbruchgründen sind Zeilen und Spalten vertauscht.)

	$[E \to T.\{\{+\|-\}T\}^*]$	$[T \to F.\{\{*\|/\}F\}^*]$
+	$[E \to T\{\{.+\|-\}T\}^*]$	$[T \to F\{\{*\|/\}F\}^*.]$
−	$[E \to T\{\{+\|.-\}T\}^*]$	$[T \to F\{\{*\|/\}F\}^*.]$
#	$[E \to T\{\{+\|-\}T\}^*.]$	$[T \to F\{\{*\|/\}F\}^*.]$
)	$[E \to T\{\{+\|-\}T\}^*.]$	$[T \to F\{\{*\|/\}F\}^*.]$
*	*error*	$[T \to F\{\{.*\|/\}F\}^*]$
/	*error*	$[T \to F\{\{*\|./\}F\}^*]$

Beachten Sie, daß bei der Aufstellung der Tabelle eine Kompression vorgenommen wurde. Aus dem Item $[E \to T.\{\{+|-\}T\}^*]$ wurde unter + gleich in das Item $[E \to T\{\{.+|-\}T\}^*]$ übergegangen. Analog für − und für das Item $[T \to F.\{\{*|/\}F\}^*]$ unter $*$ und /. Dadurch kann man sich alle Items der Form $[E \to T\{.\{+|-\}T\}^*]$ bzw. $[T \to F\{.\{*|/\}F\}^*]$ und zur Übersetzungszeit jeweils einen Ableitungsschritt sparen. □

Recursive descent RLL(1)-Parser

Eine populäre Darstellung für RLL(1)-Parser ist die in Form eines Programms. Diese Darstellung kann sowohl automatisch aus einer RLL(1)-Grammatik und ihren $FIRST_1$- und $FOLLOW_1$-Mengen erzeugt werden, wie wir gleich sehen werden, als auch „zu Fuß" programmiert werden. Letzteres ist die naheliegende Parserimplementierungsmethode, wenn kein Parsergenerator zur Verfügung steht.

Gegeben sei eine rechtsreguläre kontextfreie Grammatik $G = (V_N, V_T, p, S)$ mit $V_N = \{X_0, \ldots, X_n\}$, $S = X_0$, $p = \{X_0 \mapsto \alpha_0, X_1 \mapsto \alpha_1, \ldots, X_n \mapsto \alpha_n\}$. Aus der rechtsregulären kontextfreien Grammatik G und den berechneten $FIRST_1$- und $FOLLOW_1$-Mengen wird jetzt durch rekursive Funktionen *p_progr* und *progr* der Parser, ein sogenannter recursive descent-Parser, erzeugt. Für jede Produktion, d.h. für jedes Nichtterminal X, wird eine Prozedur mit Namen X erzeugt. Die Konstruktoren für reguläre Ausdrücke der rechten Seiten werden in Konstrukte wie case-, while-, repeat-Anweisungen, Tests auf Vorhandensein von Terminalsymbolen und rekursive Aufrufe von Prozeduren für Nichtterminale übersetzt. Hierbei werden die $FIRST_1$- und $FOLLOW_1$-Mengen von Vorkommen von regulären Ausdrücken benötigt, z.B. um die richtige von mehreren Alternativen auszuwählen. So ein Vorkommen eines regulären (Unter-) Ausdrucks entspricht genau einem erweiterten kontextfreien Item. Deshalb wird die Funktion *progr* rekursiv über erweiterte kontextfreie Items der Grammatik G definiert. In der Fallunterscheidung für Alternativen wird die folgende Funktion *FiFo* benutzt. $FiFo([X \to \cdots.\beta\cdots]) = FIRST_1(\beta) \oplus_1 FOLLOW_1([X \to \cdots.\beta\cdots])$.

```
program parser;
    var nextsym: symbol;
    proc scan;
        (* liest nächstes Eingabesymbol in nextsym *)
    proc error(meldung: string);
        (* gibt Fehlermeldung aus und stoppt Parserlauf *)
    proc accept;
        (* meldet Ende der Analyse, stoppt Parserlauf *)
    p_progr(X_0 → α_0);
    p_progr(X_1 → α_1);
        ⋮
    p_progr(X_n → α_n);
```



```
begin
    scan;
    X_0;
    if nextsym = "#"
        then accept
        else error("...")
    fi
end
```

p_progr$(X \to \alpha)$ =
    proc X;
     begin
        progr$([X \to .\alpha])$
     end;

progr$([X \to \cdots.(\alpha_1|\alpha_2|\cdots|\alpha_{k-1}|\alpha_k)\cdots])$ =
    case *nextsym* **in**
    FiFo$([X \to \cdots(.\alpha_1|\alpha_2|\cdots|\alpha_{k-1}|\alpha_k)\cdots])$:
        progr$([X \to \cdots(.\alpha_1|\alpha_2|\cdots|\alpha_{k-1}|\alpha_k)\cdots])$;
    FiFo$([X \to \cdots(\alpha_1|.\alpha_2|\cdots|\alpha_{k-1}|\alpha_k)\cdots])$:
        progr$([X \to \cdots(\alpha_1|.\alpha_2|\cdots|\alpha_{k-1}|\alpha_k)\cdots])$;
        ⋮
    FiFo$([X \to \cdots(\alpha_1|\alpha_2|\cdots|.\alpha_{k-1}|\alpha_k)\cdots])$:
        progr$([X \to \cdots(\alpha_1|\alpha_2|\cdots|.\alpha_{k-1}|\alpha_k)\cdots])$;
    otherwise *progr*$([X \to \cdots(\alpha_1|\alpha_2|\cdots|\alpha_{k-1}|.\alpha_k)\cdots])$;
    endcase

progr$([X \to \cdots.(\alpha_1\alpha_2\cdots\alpha_k)\cdots])$ =
    progr$([X \to \cdots(.\alpha_1\alpha_2\cdots\alpha_k)\cdots])$;
    progr$([X \to \cdots(\alpha_1.\alpha_2\cdots\alpha_k)\cdots])$;
        ⋮
    progr$([X \to \cdots(\alpha_1\alpha_2\cdots.\alpha_k)\cdots])$;

progr$([X \to \cdots.(\alpha)^*\cdots])$ =
    while *nextsym* **in** $FIRST_1(\alpha)$ **do**
        progr$([X \to \cdots.\alpha\cdots])$
    od

progr$([X \to \cdots.(\alpha)^+\cdots])$ =
    repeat *progr*$([X \to \cdots.\alpha\cdots])$
    until *nextsym* **not in** $FIRST_1(\alpha)$

progr$([X \to \cdots.\varepsilon\cdots])$ = ;

Für $a \in V_T$ ist
progr$([X \to \cdots.a\cdots])$ =
    if *nextsym* $= a$ **then** *scan*
    else *error*
    fi

Für $Y \in V_N$ ist
$progr([X \to \cdots .Y \cdots]) = Y$

Wie arbeitet ein solcher Parser? Die Prozedur X zu einem Nichtterminal X ist dafür zuständig, Wörter für X zu erkennen. Wenn sie aufgerufen wird, ist bereits das erste Symbol des zu erkennenden Wortes durch den integrierten Scanner/Sieber, die Prozedur *scan*, gelesen. Hat sie ein Wort für X gefunden und kehrt zurück, so hat sie ihrerseits bereits das auf dieses Wort folgende Symbol gelesen. Im nächsten Abschnitt wird eine mögliche Modifikation zum Zwecke der Fehlerbehandlung beschrieben.

Es folgt die Erzeugung des recursive descent-Parsers für die obige rechtsreguläre kontextfreie Grammatik G.

Beispiel 8.3.16 (Fortführung von Beispiel 8.3.10)
Für die erweiterte Ausdrucksgrammatik ergibt sich der folgende Parser. Dabei wird für Terminalsymbole ihre Stringdarstellung benutzt. Man beachte aber die Bemerkungen im Kapitel Lexikalische Analyse zu Symbolen und Darstellungen von Symbolen.

```
program parser;
var nextsym: string;
proc scan;
  {liest nächstes Eingabesymbol in nextsym}

proc error (meldung: string);
  {gibt Fehlermeldung aus und stoppt Parserlauf}

proc accept;
  {meldet erfolgreiche Analyse, stoppt Parserlauf}

proc S;
  begin E end;

proc E;
  begin
    T;
    while nextsym ∈ {"+"|"-"} do
      case nextsym in
        {"+"}: if nextsym = "+" then scan else error("+ erwartet") fi;
        otherwise if nextsym = "-" then scan else error("- erwartet") fi;
      endcase;
      T
    od;
  end;
```



```
proc T;
  begin
    F;
    while nextsym ∈ {"*"|"/"} do
      case nextsym in
        {"*"}: if nextsym = "*" then scan else error("* erwartet") fi;
        otherwise if nextsym = "/" then scan else error("/ erwartet") fi;
      endcase;
      F
    od;
  end;

proc F;
  begin
      case nextsym in
        {"("}:  E;
                if nextsym = ")" then scan else error(" ) erwartet") fi;
        otherwise if nextsym ="id" then scan else error("id erwartet") fi;
      endcase;
  end;

begin
scan;
S;
if nextsym = "#"
then accept
else error("# erwartet")
fi
end.
```

Durch die schematische Erzeugung dieses Programms entstehen einige Ineffizienzen. Diese können teilweise durch weniger naive Generierungsschemata vermieden werden, siehe Übung.

8.3.6 Fehlerbehandlung in LL(k)-Parsern

LL(k)-Parser haben die Eigenschaft des fortsetzungsfähigen Präfixes; d.h. jeder von einem LL(k)-Parser bestätigte Anfang eines Eingabewortes hat mindestens eine Fortsetzung zu einem Satz der Sprache. Obwohl Parser i.a. nur Fehlersymptome und nicht die Fehler selbst finden, legt es die obige Eigenschaft nahe, auf Korrekturen in dem bereits gelesenen Teil der Eingabe zu verzichten und statt dessen durch Veränderung oder teilweises Überlesen der restlichen Eingabe wieder eine Parser-Konfiguration zu suchen, aus der eine Analyse der restlichen Eingabe möglich ist. Das vorgestellte Verfahren bemüht sich, durch möglichst

geschicktes Überlesen eines Anfangs der restlichen Eingabe wieder eine geeignete Kombination von Kellerinhalt und restlicher Eingabe herzustellen.

Eine naheliegende Vorgehensweise wäre es, eine Endklammer oder ein Trennsymbol für das aktuelle Nichtterminal zu suchen und alle dazwischenliegenden Eingabesymbole zu überlesen oder beim Finden eines aussagekräftigen Endesymbols für ein Konstrukt Einträge im Keller so lange zu löschen, bis das zu diesem Endesymbol korrespondierende Nichtterminal oben auf dem Keller erscheint. Das hieße in Pascal oder ähnlichen Sprachen etwa:

- Bei der Analyse einer Anweisung könnte man ein Semikolon suchen,
- bei der Analyse einer Deklaration ein Komma oder ein Semikolon,
- bei bedingten Anweisungen, bzw. Schleifen ein **fi** bzw. **od**, wenn diese Symbole in der Sprache existieren,
- zu einem offenen **begin** für eine Anweisungsfolge würde man ein **end** suchen.

Dieser sogenannte **Panik-Modus** hat jedoch mehrere gravierende Nachteile. Selbst wenn das gesuchte Symbol im Programm vorhanden ist, kann der Parser größere Folgen von Wörtern ohne Analyse überlesen, bis er das Symbol findet. Wenn das Symbol sogar fehlt oder nicht zu der aktuellen Inkarnation des aktuellen Nichtterminals gehört, so gerät der Parser meist aus dem Tritt.

Der vorgestellte Fehlerbehandlungsalgorithmus geht deshalb differenzierter vor: Er verfügt über zwei Modi, den **Parser-Modus** und den **Fehler-Modus**; im Parser-Modus für ein Nichtterminal X ist er dabei, ein Wort für X zu analysieren und hat seit dem Beginn dieser Analyse noch keinen Fehler entdeckt oder einen entdeckten Fehler (vermeintlich) vollständig behandelt. Der Parser-Modus für X wird verlassen, wenn ein Wort für X gefunden wurde. Wurde der Parser-Modus in der Anfangskonfiguration gestartet, ist also $X = S$, so ist jetzt das Ende der syntaktischen Analyse erreicht, vorausgesetzt die Eingabe ist erschöpft. Der Parser-Modus kann allerdings auch im Fehlermodus rekursiv gestartet worden sein; dann kehrt er auch dorthin zurück. In den Fehlermodus geht er bei Auftreten eines Fehlers. Er überliest dann die nächsten Eingabesymbole bis er zu der aktuellen Analysesituation passende Endesymbole findet. Damit er das oben kritisierte nichtanalysierende Überlesen großer Eingabeteile vermeidet, geht er rekursiv in den Parser-Modus für ein Nichtterminal X über, wenn er ein charakteristisches Anfangssymbol für X findet. Wie er zwischen Parser- und Fehlermodus wechselt, kann man in Abbildung 8.17 sehen. Dort sind a_X, a_Y bzw. a_Z Anfangssymbole und b_X, b_Y bzw. b_Z Endesymbole für die Nichtterminale X, Y bzw. Z.

Das Zurückschalten vom Fehler-Modus in den Parser-Modus beim Finden eines charakteristischen Endesymbols nennen wir **Fortsetzen bei einem Fortsetzungssymbol**, das rekursive Umschalten vom Fehler-Modus in den Parser-Modus beim Finden eines charakteristischen Anfangsymbols **Aufsetzen bei einem Aufsetzsymbol**.

Wir werden jetzt jeweils zwei Klassen von Aufsetzsymbolen und Fortsetzungssymbolen kennenlernen und das Generierungsschema für recursive descent-

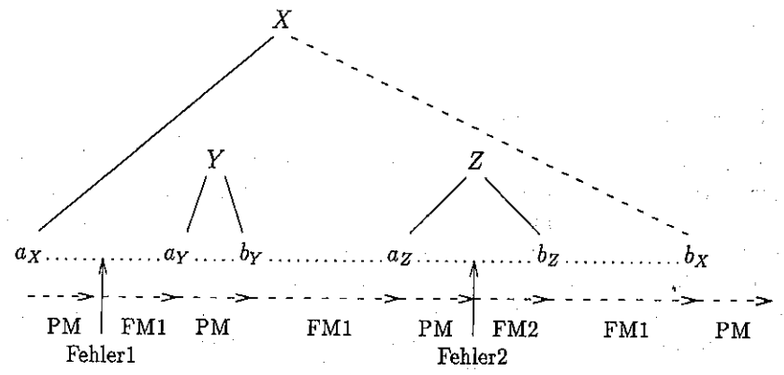

Abb. 8.17: Fehlerbehandlung; FM steht für Fehler-Modus, PM für Parser-Modus.

RLL(1)-Parser schrittweise für diese vier Klassen so modifizieren, daß die generierten RLL(1)-Parser das Fortsetzen bzw. Aufsetzen im Fehlerfall beherrschen. Dabei benutzt der Generator Benutzerangaben, für welche Nichtterminale bzw. Vorkommen von Nichtterminalen diese Fehlerbehandlungen wirksam werden sollen.

Das Generierungsschema erzeugt eine boolesche Funktion X für ein Nichtterminal X. Die Funktion ist für die Erkennung von Wörtern für X zuständig, aber auch für die Behandlung von Fehlern, die durch Tests im Rumpf von X entdeckt werden. Ein Ziel der Modifikation des Generierungsschemas ist das folgende:

Eine Fehlermeldung wird nur in X ausgegeben, d.h. Folgefehlermeldungen für den gleichen Fehler aus verschiedenen Parserprozeduren sollen vermieden werden. Den Parserfunktionen, die X aufgerufen haben, wird mitgeteilt, ob in X ein Fehler entdeckt und gemeldet wurde, der sie auch betreffen kann. Dies würde besagen, daß der Fehler-Modus in X eingenommen wurde und nach Verlassen von X weiterbesteht. Um dieses Ziel zu erreichen, werden die Parserprozeduren in boolesche Funktionen umgewandelt. Eine Parserfunktion X kehrt mit dem Ergebnis *true* zurück, wenn die Funktion, welche X aufgerufen hat, nach der Rückkehr aus X im Parser-Modus weiterfahren soll. Kehrt X mit *false* zurück, so liegt eine nicht abgeschlossene Behandlung eines Fehlers vor. Der Aufrufer muß im Fehler-Modus fortfahren. Man beachte dieses Zusammenspiel von lokaler und nicht lokaler Fehlerbehandlung: Die Funktion, die den Fehler entdeckt hat, meldet den Fehler und versucht weiter fortzusetzen. Gelingt es ihr, so kehrt sie – trotz entdeckter Fehler – mit *true* zurück; gelingt es ihr nicht, so teilt sie dies dem Aufrufer durch das Ergebnis *false* mit. Daraufhin versucht sich der Aufrufer an der Behandlung des Fehlers.

Das Ende des Rumpfes jeder Parserfunktion mit gewünschter Fehlerbehandlung sieht folgendermaßen aus:

fehler: (* Ausgabe einer für die Situation bezeichnenden Fehlermeldung *)
behandl: (* Fortsetzen oder Aufsetzen *)

Fortsetzen bei *LAST*-Symbolen

Nehmen wir an, der Parser ist dabei, ein Wort für ein Nichtterminal X zu analysieren. Dabei stößt er auf einen Fehler, d.h. eine Situation, in der das nächste Eingabesymbol nicht zum aktuellen Zustand paßt. Eine naheliegende Fehlerbehandlung wäre es, den Fehler zu melden und die nächsten Eingabesymbole zu überlesen, bis ein Symbol gefunden wird, welches das letzte Symbol in einem Wort für X sein kann. In einer Pascal-Anweisungsfolge würde man das Symbol **end** suchen, in einer Parameterliste die Klammer ')' usw. Dazu müssen wir in Analogie zu $FIRST_1$ die mengenwertige Funktion $LAST_1$ definieren.

Definition 8.3.10 (*LAST*)
Für $w \in V_T^*$ und $k \in \mathbb{N}$ sei $w : k$ der **k-Suffix** von w. Dann sei $LAST_k(X) = \{w : k \mid X \underset{R}{\overset{*}{\Longrightarrow}} w\}$ für ein Nichtterminal X. □

Um $LAST_k$ für die hier behandelten rechtsregulären kontextfreien Grammatiken zu berechnen, muß man wieder eine induktive Definition finden, d.h. eine Definition über die Struktur der regulären Ausdrücke. Dies ist eine leichte Übung für den Leser.

Die Implementierung dieser Fortsetzung bei *LAST*-Symbolen ist einfach; der Fehlerbehandlungsteil in der Funktion X ist:

```
fehler:   (* melde Fehler *)
behandl:  while nextsym not in LAST1(X) do
              scan
          od;
          scan;
          return(true)
```

Nach Finden eines *LAST*-Symbols für X wird angenommen, daß ein erfolgreiches Fortsetzen im Parser-Modus möglich ist und keine weitere Fehlerbehandlung durch den Aufrufer von X erforderlich ist.

Der Benutzer des RLL(1)-Parsergenerators bzw. der Schreiber der Eingabegrammatik muß die Nichtterminale angeben, bei denen eine Fortsetzung über *LAST*-Symbole versucht werden soll. Dies sind solche, die über charakteristische *LAST*-Symbole verfügen. Ideal dafür sind Klammern, die eindeutig einem Nichtterminal zugeordnet werden können, wie **if-fi**-Klammerung für bedingte Anweisungen, **do-od** für Schleifenrümpfe, **begin-end** für Anweisungslisten, '(' und ')' für Listen von Ausdrücken und formale Parameterspezifikationen o.ä. Vorsicht ist geboten bei rekursiven Nichtterminalen. Hier kann es geschehen, daß eine Endklammer der falschen Inkarnation des Nichtterminals zugeordnet wird. Deshalb

sollte man nur solche rekursiven Nichtterminale für die Fortsetzung bei *LAST*-Symbolen auszeichnen, bei der auch der Anfang ihrer Wörter erkannt werden kann.

Im folgenden bezeichnen wir die Menge der Nichtterminale, für die der Schreiber der Grammatik G eine Fortsetzung bei *LAST*-Symbolen wünscht, mit $LAST(G)$.

Nichtlokale Fortsetzung bei *LAST*-Symbolen

Die bisher geschilderte Art der Fehlerbehandlung setzt ein offensichtlich zu großes Vertrauen in das Vorhandensein des *LAST*-Symbols für das aktuelle Nichtterminal. Damit gibt es zwei Probleme; erstens kann der Syntaxfehler gerade darin bestehen, daß dieses *LAST*-Symbol fehlt; zweitens könnte das aktuelle Nichtterminal gerade gar kein charakteristisches *LAST*-Symbol besitzen, wie etwa das Nichtterminal für arithmetische Ausdrücke.

Die nächste Verbesserung versetzt Parserfunktionen in die Lage, auch nach *LAST*-Symbolen „umfassender" Nichtterminale zu suchen; d.h. wir betrachten jetzt Nichtterminale (genauer Vorkommen von Nichtterminalen), die in dem schon konstruierten Anfang der Linksableitung das aktuelle Nichtterminal produziert haben. Die zugehörigen Inkarnationen der Parserfunktionen sind noch nicht abgearbeitet und werden jetzt in die Fehlerbehandlung miteinbezogen, wenn diese „lokal" in der aktuellen Funktion nicht möglich ist. Dazu wird jede Parserfunktion um einen Parameter vom Typ *set of symbol* erweitert. Darin wird jeder ihrer Inkarnationen mitgeteilt, welche *LAST*-Symbole die umfassenden Nichtterminalvorkommen erwarten. Im Fehlerfall beendet die Parserfunktion das Überlesen der Eingabesymbole beim Finden eines der *LAST*-Symbole. Ist es ein eigenes *LAST*-Symbol, so kehrt sie mit Ergebnis *true*, andernfalls mit Ergebnis *false* zurück. Dann testen die Aufrufer in der Reihenfolge der Rückkehr, ob das gefundene Symbol für sie ein *LAST*-Symbol ist.

Damit ergibt sich ein neues Generierungsschema für RLL(1)-Parser mit Fortsetzung bei *LAST*-Symbolen. Zu diesem und den folgenden Generierungsschemata sind ein paar Vorbemerkungen notwendig. Wir benutzen zwei Funktionen *p_progr* und *progr*. *p_progr* wird auf Produktionen angewendet und ruft *progr* auf. *progr* ist induktiv über erweiterte kontextfreie Items definiert. *p_progr*$(X \to \alpha)$ erzeugt den „Rahmen" der Funktion X, d.h. den Kopf der Funktionsdeklaration und eine oder mehrere Variablendeklarationen, und den Fehlerbehandlungsteil. In den erzeugten Text werden einige Namen, von Variablen und Marken, definiert bzw. deklariert, die in dem von *progr* erzeugten Rumpfteil von X benutzt werden. Dies sind die Variablen s (und später noch b) und die Marken *fehler* und *behandl*. Die in den verschiedenen Fällen der Definition von *progr* benutzten (freien) Namen beziehen sich also immer auf die Deklaration in der aktuell erzeugten Funktionsdeklaration.

In dem Generierungsschema wird wieder die mengenwertige Funktion *FiFo* benutzt. Ihre Definition - zur Erinnerung - ist:

$$FiFo\,([X \to \alpha.\beta\gamma]) = FIRST_1(\beta) \oplus_1 FOLLOW_1([X \to \alpha.\beta\gamma])$$

Die entsprechenden Mengen, wie auch die Mengen $LAST_1(X)$, die in den erzeugten Texten vorkommen, sind immer vorberechnete, d.h. konstante Mengen.

(a) Für ein Nichtterminal $X \in LAST(G)$
p_progr$(X \to \alpha) =$
func X (*lasts:* **set of symbol**) **bool**;
var *s:* **set of symbol**;
begin
$s :=$ *lasts* $\cup LAST_1(X)$;
progr$([X \to .\alpha])$;
return(true);
fehler: *error*(" ... ");
behandl: **while** *nextsym* **not in** *s* **do** *scan* **od**;
if *nextsym* **in** $LAST_1(X)$
then *scan;* **return(true)**
else return(false)
fi;
end

(b) Für Nichtterminal $X \notin LAST(G)$
p_progr$(X \to \alpha) =$
func X (*lasts:* **set of symbol**) **bool**;
var *s:* **set of symbol**;
begin
$s :=$ *lasts*;
progr$([X \to .\alpha])$;
return(true);
fehler: *error*(" ... ");
behandl: **return(false)**
end

(c) Für $X \in LAST(G)$
progr$([Y \to \cdots .X \cdots]) =$
if *nextsym* **in** $FiFo([Y \to \cdots .X \cdots])$
then
if not $X(s)$
then goto *behandl*
fi
else goto *fehler*
fi;
progr$([Y \to \cdots X. \cdots])$

(d) Für $X \notin LAST(G)$
progr$([Y \to \cdots .X \cdots]) =$
if not $X(s)$
then goto *behandl*
fi;
progr$([Y \to \cdots X. \cdots])$

(e) Für $a \in V_T$
progr$([X \to \cdots .a \cdots]) =$
if *nextsym* = "a"
then *scan*
else goto *fehler*
fi;
progr$([X \to \cdots a. \cdots])$

(f) *progr*$([X \to \alpha.]) = \varepsilon$

Das Hauptprogramm des RLL(1)-Parsers ist:

program *parser;*
var *nextsym:* **symbol**;
proc *scan;*
(*liest nächstes Eingabesymbol in *nextsym**)
proc *error(meldung:* **string**);
(*gibt Fehlermeldung aus; stoppt jetzt den Parserlauf nicht mehr.*)
proc *accept;*
(*meldet Ende der Analyse; stoppt den Parserlauf*)
p_progr$(X_0 \to \alpha_0)$;
p_progr$(X_1 \to \alpha_1)$;
⋮
p_progr$(X_n \to \alpha_n)$;
begin
scan;
if *nextsym* **in** $FIRST_1(X_0)$ (*nur generiert, wenn $X_0 \in LAST(G)$*)
then
if not $X_0(\{"\#"\})$
then goto *behandl*
fi
else goto *fehler*
fi;
if *nextsym* = "#"
then *accept*
else goto *fehler*
fi;
fehler: error(" ... ");
behandl: **while** *nextsym* **not in** {"#"} **do** *scan* **od**
end

Fortsetzung bei *FOLLOW*-Symbolen

Eine Fehlerbehandlung nur über Fortsetzung bei *LAST*-Symbolen ist unzureichend. Ausdrücke haben i.a. keine charakteristischen Endesymbole. Sie kommen aber in verschieden Kontexten vor, in denen Symbole auf sie folgen müssen, die sehr aussagekräftig für die syntaktische Position sind. Dazu gehören Bedingungen in if-Anweisungen (gefolgt von **then**), in while-Schleifen (gefolgt von **do**), geschachtelte, geklammerte Ausdrücke (gefolgt von ')'), Ausdrücke in Parameterlisten (gefolgt von ',' oder ')') und Anweisungen in Anweisungsfolgen (gefolgt von ';', **end**, **fi** oder **od**). Deshalb nehmen wir eine Fortsetzung bei Folgesymbolen hinzu; dabei sind die Folgesymbole für ein Vorkommen eines Nichtterminals die Symbole, die auf dieses Vorkommen in Satzformen folgen können. Der Gram-

matikschreiber gibt wieder an, für welche Vorkommen von Nichtterminalen er eine Fortsetzung bei Folgesymbolen wünscht. Die entsprechende Menge ist mit $FOLLOW(G)$ bezeichnet.

Jetzt wird das Generierungsschema für RLL(1)-Parser zum zweiten Mal geändert, um die Fortsetzung bei Folgesymbolen zu inkorporieren. Jede Parserfunktion X bekommt einen weiteren Parameter und zwar für die Übergabe von Mengen von Folgesymbolen für das jeweilige angewandte Vorkommen von X. Die Parserfunktion X für eine erweiterte kontextfreie Produktion $X \to \alpha$ wird dann folgendermaßen generiert:

(a) Für ein Nichtterminal $X \in LAST(G)$:

```
p_progr(X → α) =
    func X ( lasts, follows: set of symbol) bool;
      var s: set of symbol;
      begin
            s := lasts ∪ LAST1(X) ∪ follows;
            progr([X → .α]);
            return(true);
    fehler:  error("...");
    behandl: while nextsym not in s do scan od;
            if nextsym in LAST1(X)
            then scan; return(true)
            fi;
            if nextsym in follows then return(true)
            fi;
            return(false)
      end
```

(b) Für ein Nichtterminal $X \notin LAST(G)$:

```
p_progr(X → α) =
    func X ( lasts, follows: set of symbol) bool;
      var s: set of symbol;
      begin
            s := lasts ∪ follows;
            progr([X → .α]);
            return(true);
    fehler:  error("...");
    behandl: if follows ≠ ∅
            then while nextsym not in s do scan od;
                  if nextsym in follows then return(true)
                  fi
            fi;
            return(false)
      end
```


(c) Für ein Vorkommen eines Nichtterminals $X \notin FOLLOW(G)$ ist.

```
progr([Y → ....X...]) =
    if nextsym in FiFo([Y → ....X...])
    then
        if not X(s, ∅)
        then goto behandl
        fi
    else goto fehler
    fi;
    progr([Y → ...X....])
```

(c') Für ein Vorkommen von $X \in FOLLOW(G)$:

```
progr([Y → ....X...]) =
    if nextsym in FiFo([Y → ....X...])
    then
        if not X(s, FOLLOW1([Y → ....X...]))
        then goto behandl
        fi
    else goto fehler
    fi;
    progr([Y → ...X....])
```

Aufsetzen bei Anfangssymbolen

An der jetzt erreichten Lösung des Fehlerbehandlungsproblems stört noch, daß in einigen Situationen bei der Suche nach einem *LAST*- oder *FOLLOW*-Symbol zuviel überlesen wird. Ganze Anweisungen oder sogar Anweisungsfolgen können übersprungen werden, wenn das Ende einer bedingten Anweisung oder eines Schleifenrumpfes gesucht wird. Außerdem gibt es ein spezielles Problem mit rekursiven Nichtterminalen, für die die Fortsetzung bei *LAST*-Symbolen spezifiziert ist. Treten die entsprechenden Konstrukte geschachtelt auf, so wird die bisher entwickelte Strategie immer das erste Endesymbol nach der Fehlerstelle zum Fortsetzen benutzen, auch wenn es zu einer tieferen Schachtelung gehört. Dadurch ergäben sich Folgefehler.

Beispiel 8.3.17

... **while** ... **do** $a := a + 1$ ↑**while** ... **do** ... **od**; ... **od** ...

Der Fehler besteht im Fehlen des Semikolons nach der Wertzuweisung. Eine Fehlerbehandlung nur auf der Basis von Fortsetzung bei *LAST*-Symbolen würde das **od**-Symbol der inneren Schleife als Ende der äußeren Schleife auffassen und damit einen Folgefehler produzieren. Eine Fortsetzung bei Folgesymbolen würde den gleichen Fehler machen oder ein Semikolon aus dem Rumpf der inneren Schleife als Folgesymbol für die Wertzuweisung benutzen, mit im wesentlichen denselben Folgen. Diese falsche Fehlerbehandlung kann vermieden werden, wenn der Beginn der inneren Schleife erkannt wird und der Parser im Parser-Modus dort wieder aufsetzt. □

Jetzt wird die Fehlerbehandlung so modifiziert, daß beim Überlesen wieder in den Parser-Modus für ein Nichtterminal X übergegangen wird, wenn ein für X charakteristisches Anfangssymbol gefunden wird. Dabei werden allerdings nur solche Nichtterminale in Betracht gezogen, die vom aktuellen Nichtterminal erreichbar sind. D.h. die Fehlerbehandlung setzt nur dann bei einem Anfangssymbol für ein Nichtterminal X auf, wenn es im Syntaxbaum eine mögliche Verbindung von dem neuen X-Knoten zu einem **offenen Nichtterminalknoten** gibt. Ein offener Nichtterminalknoten ist ein Blatt mit Nichtterminalmarkierung im bereits konstruierten Fragment des Syntaxbaums.

Im obigen Beispiel wäre das Nichtterminal *whilestat* aus dem offenen Nichtterminal *statement* ableitbar. Damit wäre ein Aufsetzen bei Anfangssymbol **while** des Nichtterminals *whilestat* möglich und würde zu einer guten Fehlerbehandlung führen.

Hier ist eine Bemerkung zu der Art der Fehlerbehandlung angebracht, die der vorgestellte Algorithmus durchführt. Es sieht so aus, als ob im Fehler-Modus nur Symbole überlesen würden, was ja einem Löschen der Symbole aus der Eingabe gleich käme. Das Aufsetzen, welches jetzt hinzugenommen wird, entspricht allerdings häufig dem Einfügen eines oder mehrerer Symbole, in obigem Beispiel etwa dem Einfügen eines Semikolons.

Welche Anforderungen müssen Anfangssymbole erfüllen, wenn man in einer Fehlersituation bei ihnen aufsetzen will?

- Sie sollten ausschließlich als erste Symbole von Wörtern für Nichtterminale auftreten.
- Ist das aktuelle Nichtterminal X, und sind $Y_1, \ldots, Y_n$ die aus X ableitbaren Nichtterminale, so muß jedes Anfangssymbol von X eindeutig zu einem der Y_j $(1 \leq j \leq n)$ gehören.

Wir nehmen an, daß der Grammatikschreiber eine Teilmenge $BEG(G)$ der Nichtterminale für das Aufsetzen bei Anfangssymbolen ausgezeichnet hat.

Zur Formalisierung der Mengen von Anfangssymbolen brauchen wir ein paar Definitionen:

Definition 8.3.11 (Anfangssymbol)
Sei $G = (V_N, V_T, p, S)$ eine rrkfG. Sei $X \in V_N$. $prod(X) = \{Y \mid X \overset{*}{\underset{R}{\Longrightarrow}} \ldots Y \ldots\}$ sei die Menge der aus X produzierbaren Nichtterminale, $prod\_b(X) = prod(X) \cap BEG(G)$, die Menge solcher Nichtterminale, die gekennzeichnet sind mit „Aufsetzen bei Anfangssymbolen".

$$BEGSYMBS(X)_Y = FIRST_1(Y) - \bigcup_{Z \in prod\_b(X), Z \neq Y} FIRST_1(Z)$$

ist die Menge aller **Anfangssymbole für Y im Kontext X.**

$$BEGSYMBS(X) = \bigcup_{Y \in prod\_b(X)} BEGSYMBS(X)_Y$$

ist die Menge aller **Anfangssymbole im Kontext X.** □

Die Anfangssymbole von Y im Kontext X bilden also eine Teilmenge der Menge $FIRST_1(Y)$. Eliminiert werden alle Symbole, die in den $FIRST_1$-Mengen von anderen, aus X ableitbaren Nichtterminalen vorkommen. Spezifiziert der Beschreiber der Grammatik G unglücklicherweise etwa, daß die Nichtterminale *STAT* und *IFSTAT* beide in $BEG(G)$ sein sollen, so fällt **if** als Anfangssymbol weg, da es in den $FIRST_1$-Mengen beider Nichtterminale liegt. Danach würde das Symbol **if** nie zum Aufsetzen benutzt.

Sei X das aktuelle Nichtterminal. Wegen eines Syntaxfehlers geht der Parser in den Fehler-Modus über und startet die Fehlerbehandlung. Wie bisher schon sucht er *LAST*- oder *FOLLOW*-Symbole. In der nächsten Verfeinerung jedoch startet er eine Fortsetzung bei jedem Symbol aus $BEGSYMBS(X)$. Er geht also rekursiv in den Parser-Modus über, und zwar für das Nichtterminal $Y \in prod(X)$, für das *nextsym* in $BEGSYMBS(X)_Y$ ist. Nach Konstruktion sind $BEGSYMBS(X)_Y$ und $BEGSYMBS(X)_Z$ für $Y \neq Z$ disjunkt, und $BEGSYMBS(X)$ ergibt sich als Vereinigung aller Mengen $BEGSYMBS(X)_Y$. Damit bestimmt jedes Symbol aus $BEGSYMBS(X)$ die Fortsetzung mit genau einem Nichtterminal. Dies geschieht durch das folgende Programmstück (ein Macro)

recover(X)*:*
case *nextsym* **in**
  $BEGSYMBS(X)_{Y_1}$ *:* $Y_1(s, \emptyset, b)$
  $BEGSYMBS(X)_{Y_2}$ *:* $Y_2(s, \emptyset, b)$
  ⋮
  $BEGSYMBS(X)_{Y_n}$ *:* $Y_n(s, \emptyset, b)$
endcase

wenn $prod\_b(X) = \{Y_1, \ldots, Y_n\}$ ist. Wie man gleich deutlicher sieht, ist s wieder die durch die Aufrufe akkumulierte Menge der Endesymbole und b ist die Menge der akkumulierten Anfangssymbole. Die Tatsache, daß der Kontext für das ausgewählte Nichtterminal Y_i nicht klar ist, drückt sich in der leeren Menge von Folgesymbolen aus.

Es folgt das Generierungsschema für den RLL(1)-Parser mit integrierter Fortsetzung bei Anfangssymbolen. Als erstes wird die Anweisungsliste des Hauptprogramms angegeben.

```
begin   scan;
        if nextsym in FiFo([S' → .S])
        then
            if not S(∅,{#},BEGSYMBS(S))
            then goto behandl
            fi
        else goto fehler
        fi;
```

```
        if nextsym = "#"
        then accept
        else goto fehler
        fi;
fehler: error("...");
behandl: while nextsym ≠ "#" do
            if nextsym in BEGSYMBS(S)
            then recover(S)
            else scan
            fi
         od
end
```

Es folgen die Definitionen der Funktionen *p_progr* und *progr* für die Produktionen der rrkfG. Sie sind wieder rekursiv über erweiterte kontextfreie Items der Grammatik definiert.

(a) Für ein Nichtterminal $X \in LAST(G)$:

```
p_progr(X → α) =
    func X ( lasts, follows, begins: set of symbol) bool;
    var  s,b: set of symbol;
    begin       s := lasts ∪ LAST1(X) ∪ follows;
                b := begins ∪ BEGSYMBS(X);
                progr([X → .α]);
                return(true);
    fehler:     error("...");
    behandl:    while nextsym not in s do
                      if nextsym in b
                      then   if nextsym in BEGSYMBS(X)
                             then recover(X)
                             else return(false)
                             fi;
                      else scan
                      fi
                od;
                if nextsym in LAST1(X) then scan; return(true) fi;
                if nextsym in follows then return(true) fi;
                return(false)
    end
```

(b) Für ein Nichtterminal $X \notin LAST(G)$:

```
p_progr(X → α) =
    func X ( lasts, follows, begins: set of symbol) bool;
    var  b,s: set of symbol;
    begin       s := lasts ∪ follows;
```



```
                b := begins;
                if follows ≠ ∅ then b := b ∪ BEGSYMBS(X) fi;
                progr([X → .α]);
                return(true);
    fehler:     error("...");
    behandl:    if follows ≠ ∅
                then while nextsym in s do
                           if nextsym in b
                           then   if nextsym in BEGSYMBS(X)
                                  then recover(X)
                                  else return(false)
                                  fi
                           else scan
                           fi;
                     od;
                     if nextsym in follows then return(true) fi;
                fi;
                return(false)
    end
```

(c) Für ein Vorkommen eines Nichtterminals $X \notin FOLLOW(G)$ ist:

```
progr([Y → ···.X···]) =
    if nextsym in FiFo([Y → ···.X···])
    then
        if not X(s, ∅, b)
        then goto behandl
        fi
    else goto fehler
    fi;
    progr([Y → ···X.···])
```

(c') Für ein Vorkommen von $X \in FOLLOW(G)$:

```
progr([Y → ···.X···]) =
    if nextsym in FiFo([Y → ···.X···])
    then
        if not X(s, FOLLOW1([Y → ···.X···]), b)
        then goto behandl
        fi
    else goto fehler
    fi;
    progr([Y → ···X.···])
```

Aufsetzen bei Vorgängersymbolen

Im letzten Abschnitt wurde die Fehlerbehandlung so erweitert, daß sie im Fehler-Modus wieder aufsetzte, d.h. in den Parser-Modus für ein Nichtterminal Y um-

schaltete, wenn ein für Y charakteristisches Anfangssymbol gefunden wurde. Es kommt jedoch häufig vor, daß Nichtterminale keine charakteristischen Anfangssymbole haben, die die oben beschriebenen Anforderungen erfüllen. Dafür gibt es eventuell Symbole, die einem speziellen Vorkommen des Nichtterminals immer vorangehen. Solche Symbole nennen wir **Vorgängersymbole** für ein Nichtterminalvorkommen. In Analogie zu *FOLLOW* nennen wir sie *PRECEDE*-Symbole. Den zwei Vorkommen von Typbeschreibungen in Pascal-Deklarationen etwa geht einmal ein Doppelpunkt und einmal ein Gleichheitszeichen voran. In der bedingten Anweisung geht den Vorkommen des Nichtterminals Anweisung einmal ein **then** und einmal ein **else** voraus.

Definition 8.3.12 (*PRECEDE*)
Sei $[X \to \alpha.\beta\gamma]$ ein erweitertes kontextfreies Item.

$$PRECEDE([X \to \alpha.\beta\gamma]) = \{a \in V_T \mid S \underset{R,lm}{\overset{*}{\Longrightarrow}} wX\delta \underset{R,lm}{\Longrightarrow} w\alpha\beta\gamma\delta \text{ und } a \in LAST_1(w\alpha)\}$$

ist die Menge aller Terminalsymbole, die in einer regulären Linkssatzform diesem Vorkommen von β vorangehen können. □

Die rekursive Definition von *PRECEDE*, die zur Berechnung notwendig ist, ist analog zu der von $FOLLOW_1$ aufgebaut. Sie bleibt dem Leser zur Übung überlassen.

Der Grammatikbeschreiber spezifiziert wieder eine Menge von Vorkommen von regulären Unterausdrücken, d.h. erweiterten kontextfreien Items, zum Aufsetzen bei Vorgängersymbolen. Wir nennen diese Menge $PREC(G)$.

Bei der Berechnung der Aufsetzsymbole, also der Anfangssymbole und der Vorgängersymbole müssen folgende Bedingungen erfüllt werden:

- führen die Spezifikationen des Grammatikbeschreibers dazu, daß ein Symbol gleichzeitig Anfangs- und Vorgängersymbol wäre, so wird dieses Symbol nicht als Aufsetzsymbol gewählt; jedes Aufsetzsymbol sollte also eindeutig für das Aufsetzen bei einem Anfangssymbol oder bei einem Vorgängersymbol stimmen;
- jedes Aufsetzsymbol sollte eindeutig ein Nichtterminal bestimmen, für welches rekursiv in den Parser-Modus gegangen wird.

Wir bestimmen in mehreren Schritten die Mengen $BEGS(X)_Y$ und $PRECS(X)_Y$ der Anfangssymbole bzw. Vorgängersymbole von Y im Kontext X.

1. Man berechne $B1(X)_Y = \begin{cases} FIRST_1(Y) & , \text{ falls } Y \in prod\_b(X) \\ \emptyset & , \text{ sonst} \end{cases}$

2. Man berechne $P1(X)_Y = \bigcup_{Z \in prod(X) \cup \{X\}} \{PRECEDE([Z \to \cdots.Y\cdots]) \mid [Z \to \cdots.Y\cdots] \in PREC(G)\}$
   $P1(X)_Y$ enthält alle Vorgängersymbole von Vorkommen von Y aus $PREC(G)$, für solche Nichtterminale, die von X aus erreichbar sind.

3. $B2(X)_Y = B1(X)_Y - \bigcup_Y P1(X)_Y$.
   Eliminiere alle *FIRST*-Symbole, die gleichzeitig *PRECEDE*-Symbole sind.

4. $P2(X)_Y = P1(X)_Y - \bigcup_Y B1(X)_Y$.
   Eliminiere alle *PRECEDE*-Symbole, die gleichzeitig *FIRST*-Symbole sind.

5. $BEGS(X)_Y = B2(X)_Y - \bigcup_{Z \neq Y} B2(X)_Z$.
   Eliminiere alle mehrfach vorkommenden Anfangssymbole.

6. $PRECS(X)_Y = P2(X)_Y - \bigcup_{Z \neq Y} P2(X)_Z$.
   Eliminiere alle mehrfach vorkommenden Vorgängersymbole.

7. $BEGSYMBS(X) = \bigcup_Y (BEGS(X)_Y \cup PRECS(X)_Y)$.
   Dies ist die neue, endgültige Definition der Aufsetzsymbole im Kontext X.

Das Generierungsschema für die RLL(1)-Parser, jetzt mit Aufsetzen bei Anfangs- und Vorgängersymbolen, bleibt unverändert bis auf das Programmstück (Macro) *recover*(X). Dies ändert sich zu:

recover(X):
case *nextsym* **in**
  $BEGS(X)_{Y_1}$: $Y_1(s, \emptyset, b)$
  ⋮
  $BEGS(X)_{Y_n}$: $Y_n(s, \emptyset, b)$
  $PRECS(X)_{Z_1}$: **begin** *scan;* $Z_1(s, \emptyset, b)$ **end**
  ⋮
  $PRECS(X)_{Z_m}$: **begin** *scan;* $Z_m(s, \emptyset, b)$ **end**
endcase

Damit ist die Fehlerbehandlung zu RLL(1)-Parsern vollständig beschrieben. Ausgehend von einer rechtsregulären kontextfreien Grammatik G und der Angabe der Mengen $LAST(G)$, $FOLLOW(G)$, $BEG(G)$ und $PREC(G)$ wird die Fehlerbehandlung durch einen modifizierten RLL(1)-Parsergenerator miterzeugt.

8.4 Bottom up-Syntaxanalyse

8.4.1 Einführung

Parser in der in diesem Abschnitt behandelten Klasse lesen ihre Eingabe ebenfalls von links nach rechts; bevor sie ein weiteres Eingabesymbol lesen, machen sie in dem bereits gelesenen und analysierten Teil alle gebotenen Reduktionen. Betrachtet man, wie sie den Syntaxbaum konstruieren, so beginnen sie mit dem Blattwort des Syntaxbaums, dem Eingabewort, konstruieren für immer größere Teile der gelesenen Eingabe Unterbäume des Syntaxbaums, indem sie bei Reduktionen Teilbäume unter dem neuen Nichtterminalknoten zusammenhängen, bis