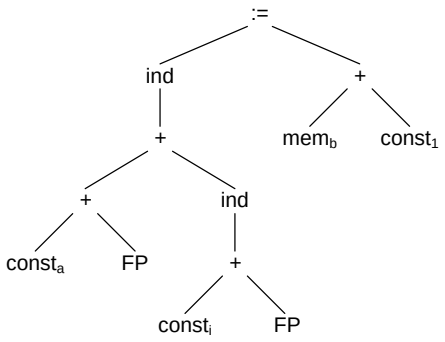
7.6 Generatoren für die Maschinencode-Erzeugung

Ziel: Die automatische Generierung eines Maschinencode-Erzeugers aus einer Beschreibung der Zielmaschine.
Dazu existieren mehrere wichtige Ansätze:

- über Baum-Grammatiken (allgemeiner Ansatz von Cattell [27])
- über SDTS und LR-Grammatiken für sequentiell dargestellte Syntaxbäume (Ansatz von Graham und Glanville [29])
- über Baum-Matching in Verbindung mit dynamischer Programmierung (Ansatz von Aho, Ganapathi, Tjiang, [22])

Bei allen Ansätzen müssen die Syntaxbäume relativ weit „aufgefächert“ sein, denn in allen Fällen wird die Arbeitsweise einzelner Maschinenbefehle durch Bäume bzw. Zeichenketten dargestellt.

Beispiel 7.11:
Der Syntaxbaum für `a[i] := b + 1` wird durch eine explizite Speicherzuordnung relativ groß. Es wird angenommen, dass `b` eine globale Variable ist und dass `a` und `i` lokal auf dem Laufzeit-Stack gespeichert sind. Das Feld `a` habe als unteren Index die 0. Zudem seien alle Variablen vom Typ `character` bzw. `integer`.
$\texttt{const}_a$ und $\texttt{const}_i$ sind die Abstände der Speicherorte von `a` und `i` relativ zum Framepointer `FP`, $\texttt{mem}_b$ ist die Speicheradresse von `b`.



7.6.1 Verwendung von Baum-Grammatiken bzw. Baum-Übersetzungs-Schemata

Die prinzipielle Vorgehensweise ist wie bei üblichen kontextfreien Grammatiken, nur werden statt Zeichenketten Bäume erzeugt. Für die Übersetzung benötigt man Produktionen der Form:

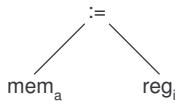
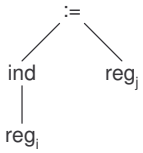
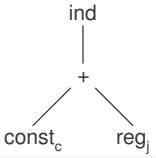
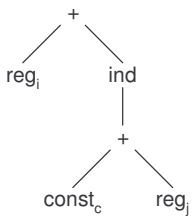
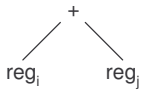
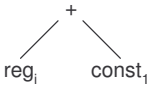
$$A \longrightarrow T\{\text{Regel}\} \quad \text{oder}$$
$$A \longrightarrow T\{\text{Kosten}\}\{\text{Regel}\}.$$

Dabei ist A ein nichtterminales Symbol, T ist ein Baum, dessen Blätter mit terminalen oder nichtterminalen Symbolen markiert sind, und alle internen Knoten von T sind mit terminalen Symbolen markiert.
Die „Regel“ gibt die semantische Aktion an. In unserer Anwendung wird jede Produktion die Abarbeitung eines Befehls darstellen. Die semantische Aktion besteht dann aus der Ausgabe des Befehls. Häufig sind derartige Baumgrammatiken, die die Wirkungsweise der Maschinenbefehle eines Rechners beschreiben, stark mehrdeutig. Um die verschiedenen Möglichkeiten bewerten zu können, wird den Produktionen auch noch eine Kostenfunktion zugeordnet, die dann zur Auswahl der „richtigen“ Produktionen benutzt wird.

Beispiel 7.12:

Bem.: Die tiefgestellten Buchstaben haben die Bedeutung von Zusatzbedingungen bzw. semantischen Prädikaten (Attributen).

Beispiel 7.13:
Es soll jetzt ein Ausschnitt einer Baumgrammatik für einige Maschinenbefehle eines hypothetischen Rechners angegeben werden. In den Regeln wird hier zur besseren Lesbarkeit statt einer ausführbaren Anweisung der erzeugte Assemblercode angegeben.

Nr.	**Produktion**	**Regel**, (erzeugter Befehl)
1	$\texttt{reg}_i \rightarrow \texttt{const}_c$	{ `LD Ri,#c` }
2	$\texttt{reg}_i \rightarrow \texttt{mem}_a$	{ `LD Ri,a` }
3	$\texttt{reg}_{FP} \rightarrow$ `FP`	
4	`mem`→ 	{ `ST a,Ri` }
5	`mem`→ 	{ `ST *Ri,Rj` }
6	$\texttt{reg}_i \rightarrow$ 	{ `LD Ri,c(Rj)` }
7	$\texttt{reg}_k \rightarrow$ 	{ `ADD Rk,Ri,c(Rj)` }
8	$\texttt{reg}_k \rightarrow$ 	{ `ADD Rk,Ri,Rj` }
9	$\texttt{reg}_i \rightarrow$ 	{ `INC Ri` }

Bem.: `FP` bezeichnet den Framepointer, der hier immer in einem fest zugeordneten Register $\texttt{reg}_{FP}$ gespeichert ist. In diesem Beispiel wird keine Kostenfunktion verwendet, sondern es wird beim Parsing die heuristische Regel verwendet, dass bei mehreren Möglichkeiten immer die Produktion mit der größeren rechten Seite verwendet wird.

Nun stellt sich die Frage: Wie übersetzt man mit Hilfe einer derartigen Grammatik? Es ist klar, dass man auch hier den gegebenen Baum parsen muss und dann, ähnlich einer attributierten Grammatik (oder einem SDTS), die entsprechenden semantischen Aktionen oder Regeln ausführen muss.
Man kann sich vorstellen, dass Parsingverfahren für Baum-Grammatiken signifikant komplexer sind als Parsingverfahren für Zeichenketten. Trotzdem beruhen sie auf den gleichen Prinzipien. So könnte man etwa nach dem Bottom-Up Verfahren vorgehen. Also wird man versuchen, den vorgelegten Syntaxbaum von unten nach oben mit Hilfe der gegebenen Regeln zu reduzieren.
Dabei stellen sich folgende Fragen:

1) In welcher Reihenfolge sollen die Reduktionen durchgeführt werden?

2) Was ist zu tun, wenn mehrere Regeln anwendbar sind?

3) Was ist zu tun, wenn keine Regel anwendbar ist?

4) Wie verhütet man unendliche Schleifen in der Reduktionsphase?

zu 1): Meistens wird dieses Verfahren mit Varianten der dynamischen Programmierung verbunden. Dadurch wird ein Bottom-Up-Vorgehen festgelegt und das ganze wird dann zu einem Pattern-Matching-Problem über Bäumen.

zu 2): Man kann z.B. eine heuristische Regel wie „wähle einen möglichst großen Teilbaum“ verwenden, um zu einem möglichst kurzen Maschinenprogramm zu gelangen. Bessere Ergebnisse erzielt man aber bei der Anwendung der dynamischen Programmierung. Hierbei betrachtet man die Kosten von Übersetzungen von Teilbäumen für unterschiedliche Registerzahlen und benutzt diese Teillösungen, um eine Gesamtlösung zusammenzusetzen.

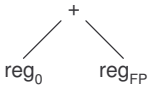
zu 3): Im schlimmsten Fall bleibt keine andere Möglichkeit als Backtracking.

zu 4): Dieses Problem kann man etwa mit einer Kostenfunktion und dynamischer Programmierung lösen.

Beispiel 7.14:
Betrachten wir den Syntaxbaum aus Beispiel 7.11 und die Beispielgrammatik aus Beispiel 7.13. Man versucht nun, den Syntaxbaum Bottom-Up und von links nach rechts zu parsen. Dabei muss man die folgenden Regeln nacheinander anwenden:

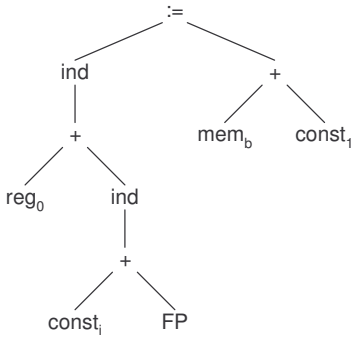
1) Regel (1): $\text{reg}_0 \rightarrow \text{const}_a$ erzeugt {`LD R0,#a`}

2) Regel (3): $\text{reg}_{FP} \rightarrow \text{FP}$

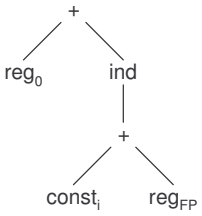
3) Regel (8): $\text{reg}_0 \rightarrow$ 

erzeugt {`ADD R0,R0,FP`}

Damit hat man als Zwischenergebnis den Baum:

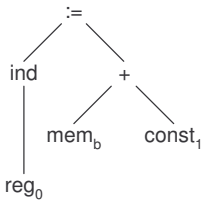


Nach der erneuten Anwendung von Regel (3) sind jetzt zwei Regeln anwendbar: die Regel (6) und die Regel (7). Da die rechte Seite der Regel (7) „größer“ ist (einen größeren Teilbaum überdeckt), wird diese Regel verwendet:

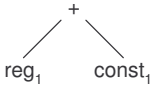
4) Regel (7): $\text{reg}_0 \rightarrow$ 

erzeugt {`ADD R0,R0,i(FP)`}.

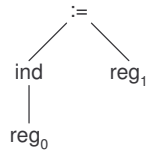
Damit hat man folgendes Zwischenresultat:



5) Regel (2): $\text{reg}_1 \rightarrow \text{mem}_b$ erzeugt {`LD R1,b`}

6) Regel (9): $\text{reg}_1 \rightarrow$ 

erzeugt {`INC R1`}

7) Regel (5): mem → 

erzeugt {`ST *R0,R1`}

Man erhält also folgendes Maschinencode-Programm:

```
LD   R0,#a
ADD  R0,R0,FP
ADD  R0,R0,i(FP)
LD   R1,b
INC  R1
ST   *R0,R1
```

7.6.2 Verwendung eines üblichen LR-Parsers

Ein anderer Ansatz [29] transformiert den vorliegenden Eingabebaum in eine Zeichenkette und verwendet die üblichen Parsermethoden, z.B. den Parsergenerator `yacc`. Bei den Baum-Produktionen müssen die rechten Seiten natürlich ebenfalls transformiert werden, so dass „übliche“ kontextfreie Produktionen entstehen.

Beispiel 7.15:
Wir betrachten wieder den Syntaxbaum aus Beispiel 7.11. Als Transformation verwenden wir die sequentielle Darstellung des Baumes in Präfix-Notation. Da der Verzweigunggrad der internen Knoten implizit gegeben ist, müssen keine weiteren Informationen abgespeichert werden. Die sequentielle Darstellung des Syntaxbaums wäre dann:

$$\texttt{:= ind + + const}_a\ \texttt{FP ind + const}_i\ \texttt{FP + mem}_b\ \texttt{const}_1$$

Die transformierte Baum-Grammatik wird als SDTS umgeschreiben:

Nr.	Produktion	Regel (erzeugter Befehl)
(1)	$\texttt{reg}_i \rightarrow \texttt{const}_c$	`LD Ri,#c`
(2)	$\texttt{reg}_i \rightarrow \texttt{mem}_a$	`LD Ri,a`
(3)	$\texttt{reg}_{FP} \rightarrow \texttt{FP}$	
(4)	$\texttt{mem} \rightarrow \texttt{:= mem}_a\ \texttt{reg}_i$	`ST a,Ri`
(5)	$\texttt{mem} \rightarrow \texttt{:= ind reg}_i\ \texttt{reg}_j$	`ST *Ri,Rj`
(6)	$\texttt{reg}_i \rightarrow \texttt{ind + const}_c\ \texttt{reg}_j$	`LD Ri,c(Rj)`
(7)	$\texttt{reg}_k \rightarrow \texttt{+ reg}_i\ \texttt{ind + const}_c\ \texttt{reg}_j$	`ADD Rk,Ri,c(Rj)`
(8)	$\texttt{reg}_k \rightarrow \texttt{+ reg}_i\ \texttt{reg}_j$	`ADD Rk,Ri,Rj`
(9)	$\texttt{reg}_i \rightarrow \texttt{+ reg}_i\ \texttt{const}_1$	`INC Ri`

und man erzeugt dann, etwa mit Hilfe von `yacc` oder `bison`, einen Bottom-Up Parser für diese Grammatik und übersetzt die Eingabe in der üblichen Art und Weise.

Probleme:

1) Die aus den Maschinenbefehlen konstruierte Grammatik ist meist hochgradig mehrdeutig. Hier hilft eventuell die heuristische Regel, möglichst viel in einem Schritt zu reduzieren. Also wird bei shift-reduce-Konflikten auf shift entschieden und bei reduce-reduce-Konflikten die längere Produktion bevorzugt.

2) Die Reihenfolge der Auswertung von Teilbäumen ist durch das Parsingverfahren festgelegt (links → rechts).

3) Es kann immer noch passieren, daß der Parsingprozess blockiert oder in eine Schleife gerät.

4) Die Registerzuweisung ist offen und muss von außen gesteuert werden.

Vorteile::

- Es ist prinzipiell relativ einfach, einen nach dieser Methode arbeitenden Maschinencode-Erzeuger auf eine neue Zielmaschine umzuprogrammieren.
- Unter gewissen Voraussetzungen ist es möglich zu beweisen, dass stets korrekter Code erzeugt wird, dass der Übersetzer nicht blockiert und nicht in eine Schleife gelangen kann.

Erweitert man die Grammatik, genauer gesagt, die Attributierung, so ist es möglich, gewisse Schwierigkeiten des ursprünglichen Ansatzes zu umgehen. So ist es zum Beispiel möglich, nur gewisse Zahlbereiche für Operanden zuzulassen. Hier wird als Beispiel angenommen, dass kleine Summanden in den Maschinenbefehl „einkodiert“ werden können, größere dagegen ein Erweiterungswort benötigen.

$$\texttt{reg}_i \;\rightarrow\; \texttt{+ reg}_i\ \texttt{const}_c \quad \{\ \texttt{if 1}\leq\texttt{c}\leq\texttt{8 then ADDQ \#c, Ri else ADD \#c, Ri}\ \}$$

Man sollte die Werte der semantischen Prädikate natürlich mit in die Entscheidung zur Auswahl der passenden Produktion zur Reduktion einfließen lassen. Durch diese Erweiterung ist es aber meist nicht mehr möglich, die korrekte und störungsfreie Übersetzung zu beweisen.

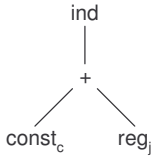
7.6.3 Der Ansatz von Aho, Ganapathi und Tjiang

Der Ansatz von Aho, Ganapathi und Tjiang beruht wiederum auf Baumgrammatiken. Die Autoren definieren ein sogenanntes **Baum-Übersetzungs-Schema** (tree-translation-scheme) und geben eine Methode an, wie mit Hilfe von Pattern-Matching und dynamischer Programmierung ein solches System implementiert werden kann. Dieses System namens ***twig*** ist zur Generierung von Maschinencode-Erzeugern für mehrere Compiler benutzt worden.
Eine allgemeine Produktion hat hat bei diesem Ansatz folgendes Aussehen:

$$\texttt{A} \rightarrow \texttt{T} \qquad \{\texttt{cost}\} = \{\texttt{action}\}$$

Dabei ist `A` ein nichtterminaler Knoten, `T` ein Baum, `cost` ist ein Code-Fragment, das zur Berechnung der Kosten dieses Musters aufgerufen wird. Fehlt dieses Feld, so werden Einheitskosten angenommen. `action` ist ein Code-Fragment, das bei Akzeptieren dieser Produktion ausgeführt wird und üblicherweise den Maschinencode generiert.

Beispiel 7.16:
Die Produktion (6) aus Beispiel 7.13 würde hier wie folgt dargestellt:

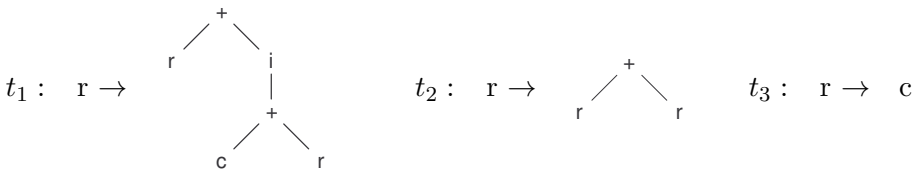
$\texttt{reg}_i \rightarrow$ 

$\{\ \texttt{3+cost.reg}_j\ \} = \{\ \texttt{LD Ri,c(Rj)}\}$

In die Berechnung der Kostenfunktion gehen in diesem Fall die notwendigen Kosten zur Berechnung des Wertes im Register `Rj` mit ein. Die Kosten werden zur Bestimmung der anzuwendenden Produktionen benutzt.
Interessant ist, dass hier nicht eines der üblichen Parsingverfahren modifiziert wird, sondern es wird ein Tree-Pattern-Matching angewendet. Dieses Verfahren basiert auf einer Idee von Hoffman und O' Donnell [31] und stellt eine Erweiterung des Aho-Corasick-Verfahrens [19] dar.

Idee: Die Bäume werden durch die Menge aller Wege von der Wurzel zum Blatt festgelegt, wobei die Verzweigungsnummern (die Nummer des Nachfolgers) auch mit notiert sind. Für die so erzeugten Pattern wird ein Aho-Corasick-Automat konstruiert, der eine parallele Suche nach all diesen Pattern im Baum erlaubt.

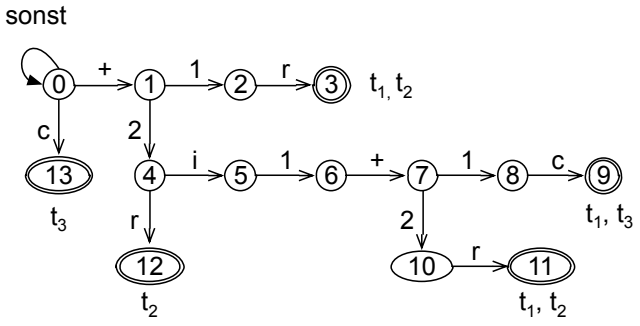
Beispiel 7.17:
Man betrachte die folgenden drei Produktionen einer Baumgrammatik:



Die Menge der Wege ist:

```
+ 1 r
+ 2 i 1 + 1 c
+ 2 i 1 + 2 r
+ 2 r
c
```

wobei zu bemerken ist, dass der Weg + 1 r sowohl zu t_1 als auch zu t_2 gehört. Der Aho-Corasick-Automat ist dann also:



Bem.: Die Werte der Fehler-Funktion sind aus Gründen der besseren Übersichtlichkeit in der obigen Abbildung nicht mit eingetragen!

Anwendung findet nun eine Technik von Hoffman und O' Donnell, um jetzt die Pattern im Baum zu erkennen. Man durchläuft den Eingabe-Baum von oben nach unten und berechnet für jeden Knoten im Baum den Zustand des Automaten für den Weg von der Wurzel bis zu

diesem Knoten. Aus diesen Informationen lässt sich dann (bei geeigneter Datenstruktur) einfach erkennen, an welchen Stellen rechte Seiten von Regeln im Baum auftreten. Die berechneten Kosten werden dann zur Auswahl der „günstigsten“ Regel benutzt und die Regel dann angewendet, d.h. die rechte Seite der Regel wird im Eingabe-Baum durch die linke ersetzt. Es muss jetzt noch berücksichtigt werden, dass durch den Ersetzungsprozess ein Teilbaum durch einen Knoten ersetzt wird und damit wieder neue Matching-Möglichkeiten entstehen.