GRAY – ein Generator für rekursiv absteigende Ybersetzer

M. A. Ertl

Inhaltsverzeichnis

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
Praktikum, Institut fuer Computersprachen, TU Wien, 1989

1 Motivation

Forth wird oft als Sprache zur Schaffung anwendungsspezifischer Sprachen bezeichnet, als Meta-Sprache[3]. Meist wird die neue Sprache einfach durch die Definition entsprechender Forth-Worte implementiert. Forth's Flexibilität unterstützt diese Methode, hilfreich sind insbesondere Immediate-Wörter (Wörter, die zur Übersetzungszeit nicht compiliert, sondern ausgeführt werden, ähnlich Makros in Lisp) und die für den Forth-Compiler benötigten Wörter. Vorteilhaft ist dabei, daß kein Parser benötigt wird, daß sich die Schnittstelle zu Forth einfach gestaltet und somit vorhandene Worte und Programme in der neuen Sprache benutzt werden können, und daß Software Tools auch auf Programme in der neuen Sprache angewandt werden können. Andererseits muß dabei, ähnlich wie in Lisp und in Prolog, meist eine unkonventionelle Syntax in Kauf genommen werden, wie Gray, der Forth-Assembler [3, 4], und nicht zuletzt Forth selbst zeigen.

Mit einigen Tricks kann man hier Abhilfe schaffen, wobei auch die Vorteile der Methode schrittweise verlorengehen, will man jedoch eine vorgegebene Syntax implementieren, ist ein Parser-Generator wohl besser geeignet. Rekursiv absteigende Parser sind zwar leicht zu schreiben, aber nicht sehr wartungsfreundlich, und – speziell in Forth – schwer lesbar. Besonders bei größeren Grammatiken zahlt sich die automatische Übersetzung aus.

Auch bei Einsatz eines Parser-Generators kommen Forth's Vorteile bei der Metaprogrammierung zum Vorschein, z.B. kann Forth's Dictionary-Verwaltung für die Symboltabelle verwendet werden.

2 Entwurf

Mein Hauptziel war es, einen brauchbaren Parser-Generator zu bauen. Daneben sollte er möglichst einfach, portabel, und klar programmiert sein. Ich entschied mich daher dafür, rekursiv absteigende Parser zu erzeugen, da die Übersetzung von EBNF o.ä. relativ einfach ist. Damit können L-attributierte Grammatiken verarbeitet werden, während das bei LR-Parser-Generatoren nur mit Tricks möglich ist.

Aufgrund schlechter Erfahrungen mit Prolog's DCGs betrieb ich relativ großen Aufwand bei der Fehlersuche, ließ aber im Sinne der Einfachkeit, Portabilität und Klarheit Optimierungen, Fehlerbehandlung im resultierenden Parser und automatische Grammatiktransformationen weg.

Neben der üblichen Syntax-Konstruktionen wie Nonterminal, Terminal, Alternative, Verkettung, Leerwort und semantischen Aktionen führte ich noch einige praktische Konstruktionen ein, die direkt Forths Kontrollstrukturen entsprechen, und die hie und da in ähnlichen Programmen zu finden sind:

- für `if`...`endif` die Option,
- für `begin`...`while`...`repeat` die beliebige Wiederholung,
- für `begin`...`until` die mindestens einmalige Wiederholung.

Neben der Hauptaufgabe der Übersetzung sollen auch Fehler und Fehlermöglichkeiten erkannt und gemeldet werden, insbesondere Linksrekursionen und Konflikte.

Ein Konflikt liegt bei Top-Down-Parsern dann vor, wenn eine Entscheidung ansteht und mehrere Alternativen mit dem gleichen Token beginnen können. Das ist genau dann der Fall, wenn die jeweiligen First-Mengen (die, wenn sie ϵ enthalten, noch mit der Follow-Menge vereinigt werden müssen) nicht disjunkt sind.

Da ich mir die komplizierte Follow-Mengen-Berechnung sparen wollte, benutzte ich zunächst ein auf der zweiten Überlegung beruhendes Vorwärtsverfahren. Eine naive Implementation führte allerdings selbst bei kleinen Grammatiken zu explodierendem Speicherbedarf und einigen anderen unangenehmen Folgen. So stellte ich Gray schließlich auf konventionelle Konfliktentdeckung mit Follow-Mengen-Berechnung um.

Auch wenn Konflikte entdeckt werden, wird ein Parser erzeugt. Der Parser verkörpert die konventionellen Eindeutigkeitsregeln (siehe Kapitel A.3)

Die Berechnungen lassen sich recht gut als eine Art attributierte Grammatik darstellen (Abbildung 1). Die Berechnungen für die Nonterminale verwandeln den AST in einen gerichteten Graphen (ASG), was eine Sonderbehandlung bei der Berechnung von ererbten Attributen nötig macht.

2.1 Die Attribute

generated Der vom Konstrukt generierte Forth-Code

first Die First-Menge eines Konstrukts ohne ϵ

maybe-empty zeigt an, ob das Konstrukt ϵ ableiten kann, ob also die First-Menge ϵ enthält.

follow Die Follow-Menge des Konstrukts, ein ererbtes Attribut. Da ein Knoten mehrere Mengen erben kann, ist der Wert des Attributs die Vereinigung dieser Mengen, bzw. die kleinste Menge, die Übermenge aller geerbten Mengen ist (angezeigt durch $\supseteq$).

ambiguity Die Konflikt-Menge für das Konstrukt

Die Attribute *action* und *first* von *terminal* werden bei der Erzeugung des *terminals* bestimmt, die Attribute von *nonterm* sind gleich den Attributen der rechten Seite der Gray-Regel (nicht AG-Regel) für das Nonterminal, *nonterm.exec* ist der Aufruf des für die Regel erzeugten Codes.

Die Implementation wird im Kapitel A.7 beschrieben.

3 Besonderheiten durch die Implementation in Forth

Forths auffallendste Eigenschaft ist der Stack. Dieser hatte zwar auf die Implemnentation keinen besonderen Einfluß, ist aber für die Aktionen sehr vorteilhaft, die auf ihre Daten nicht mit `$1` etc. zugreifen müssen, sondern sie auf dem Stapel erwarten und die Ergebnisse dort zurücklassen.

Forths Interaktivität erleichtert nicht nur das Programmtesten, sondern Forth-Programme können als Folge auch auf den Forth-Compiler zugreifen, was die Metaprogrammierung erleichtert. Gray erzeugt daher nicht eine Datei, die

$$E \rightarrow E_1E_2 \quad \left\{ \begin{array}{lcl} E.generated & = & E_1.generated\ E_2.generated \\ E.first & = & \left\{ \begin{array}{ll} E_1.first \cup E_2.first & \text{if } E_1.maybe\text{-}empty \\ E_1.first & \text{otherwise} \end{array} \right. \\ E.maybe\text{-}empty & = & E_1.maybe\text{-}empty \wedge E_2.maybe\text{-}empty \\ E_1.follow & \supseteq & \left\{ \begin{array}{ll} E_2.first \cup E.follow & \text{if } E_2.maybe\text{-}empty \\ E_2.first & \text{otherwise} \end{array} \right. \\ E_2.follow & \supseteq & E.follow \\ E.ambiguity & = & \emptyset \end{array} \right.$$

$$E \rightarrow E_1|E_2 \quad \left\{ \begin{array}{lcl} E.generated & = & \left\{ \begin{array}{ll} \begin{array}{l} E_2.first - E_1.first\ \texttt{sym-in if}\ E_2.generated \\ \qquad\qquad \texttt{else}\ E_1.generated\ \texttt{endif} \end{array} & \text{if } E_1.maybe\text{-}empty \\ \begin{array}{l} E_1.first\ \texttt{sym-in if}\ E_1.generated \\ \qquad\qquad \texttt{else}\ E_2.generated\ \texttt{endif} \end{array} & \text{otherwise} \end{array} \right. \\ E.first & = & E_1.first \cup E_2.first \\ E.maybe\text{-}empty & = & E_1.maybe\text{-}empty \vee E2.maybe\text{-}empty \\ E_1.follow & \supseteq & E.follow \\ E_2.follow & \supseteq & E.follow \\ E.ambiguity & = & (E_1.first \cap E_2.first) \cup \left\{ \begin{array}{ll} E.first \cap E.follow & \text{if } E.maybe\text{-}empty \\ \emptyset & \text{otherwise} \end{array} \right. \end{array} \right.$$

$$E \rightarrow \quad \left\{ \begin{array}{lcl} E.generated & = & \\ E.first & = & \emptyset \\ E.maybe\text{-}empty & = & true \\ E.ambiguity & = & \emptyset \end{array} \right.$$

$$E \rightarrow terminal \quad \left\{ \begin{array}{lcl} E.generated & = & terminal.action \\ E.first & = & terminal.first \\ E.maybe\text{-}empty & = & false \\ E.ambiguity & = & \emptyset \end{array} \right.$$

$$E \rightarrow E_1? \quad \left\{ \begin{array}{lcl} E.generated & = & E.first\ \texttt{sym-in if}\ E_1.generated\ \texttt{endif} \\ E.first & = & E_1.first \\ E.maybe\text{-}empty & = & true \\ E_1.follow & \supseteq & E.follow \\ E.ambiguity & = & E.follow \cap E_1.first \end{array} \right.$$

$$E \rightarrow E_1* \quad \left\{ \begin{array}{lcl} E.generated & = & \texttt{begin}\ E_1.first\ \texttt{sym-in while}\ E_1.generated\ \texttt{repeat} \\ E.first & = & E_1.first \\ E.maybe\text{-}empty & = & true \\ E_1.follow & \supseteq & E.follow \cup E_1.first \\ E.ambiguity & = & E.follow \cap E_1.first \end{array} \right.$$

$$E \rightarrow E_1+ \quad \left\{ \begin{array}{lcl} E.generated & = & \texttt{begin}\ E_1.generated\ E_1.first\ \texttt{sym-in not until} \\ E.first & = & E_1.first \\ E.maybe\text{-}empty & = & E_1.maybe\text{-}empty \\ E_1.follow & \supseteq & E.follow \cup E_1.first \\ E.ambiguity & = & E.follow \cap E_1.first \end{array} \right.$$

$$E \rightarrow \{code\} \quad \left\{ \begin{array}{lcl} E.generated & = & code.generated \\ E.first & = & \emptyset \\ E.maybe\text{-}empty & = & true \\ E.ambiguity & = & \emptyset \end{array} \right.$$

$$E \rightarrow nonterm \quad \left\{ \begin{array}{lcl} E.generated & = & nonterm.exec \\ E.first & = & nonterm.first \\ E.maybe\text{-}empty & = & nonterm.maybe\text{-}empty \\ nonterm.follow & \supseteq & E.follow \\ E.ambiguity & = & \emptyset \end{array} \right.$$

Abbildung 1: Attributierte Grammatik für Grays Ausdrücke dann von einem Compiler weiterverarbeitet wird, sondern erzeugt den Parser direkt im Speicher. Für die sonst nötige Symboltabelle wird einfach Forths Dictionary verwendet.

Sehr praktisch ist auch der primitive Parser. Alles, was sich zwischen zwei Leerzeichen befindet, ist ein Wort. Daher können beliebige Symbole definiert werden, die bei Gray die Syntax von Grammatiken konstituieren. Das erspart nicht nur die Entwicklung eines eigenen Parsers, sondern ermöglicht es auch, Forth und die Grammatik ohne komplizierte Schnittstelle zu kombinieren, was sowohl für die Schnittstelle zum Scanner als auch für die Erweiterbarkeit wichtig ist. Als nachteilig erweist sich bei dieser Vorgangsweise die leicht unkonventionelle Syntax und der geringe Schutz vor Fehlern, an den Forth-Programmierer allerdings gewöhnt sind.

Einzigartig ist Forths Möglichkeit, Definitionswörter zu definieren. Von dieser Möglichkeit wurde Gebrauch gemacht, um Kontext-Variablen, -Konstanten und Methoden zu definieren, die einfach über ihren Namen aufgerufen werden können. Auch Terminale und Nonterminale werden mit selbstgeschriebenen Definitionswörter erzeugt.

An Forth wird oft das weitgehende Fehlen von Sicherheitsvorkehrungen – insbesondere jeglicher Art von Typ-Überprüfung – kritisiert, was jedoch keine auffallenden Nachteile mit sich zog. Typenfehler geben sich schnell zu erkennen und sind interaktiv leicht zu lokalisieren; Erfahrung ist auch hier vonnöten. Weiters stellt sich die Frage, ob eine Typenprüfung nicht viele Möglichkeiten eliminieren würde – bei Gray würde ihr wahrscheinlich die Implementation der Syntax zum Opfer fallen.

4 Erfahrungen und Resultate

Die erste funktionsfähige Version war in kurzer Zeit implementiert. Es folgten viele kleine und größere Verbesserungen, die allerdings immer nur lokale Änderungen im Programm zur Folge hatten, was darauf hindeutet, daß der Grundentwurf gut war (oder daß ich vor gröberen Änderungen zurückschreckte). Gray scheint jedenfalls die Ausnahme zu sein, die "Plan to throw one away" bestätigt. Daran dürften objekt-orientierte Programmiertechniken, viele kleine Definitionen, modulare Programmierung schuld sein.

An Gray fällt vor allem der geringe Speicherbedarf auf – etwas mehr als 5Kbytes auf einem 16-bit-forth, wobei die Definitionsnamen und ein 1Kbyte großer Datenbereich inbegriffen sind. Tabelle 1 zeigt Daten über Grays Effizienz zur Laufzeit.

	calc	oberon
ASG-Knoten	47	367
Regeln	4	56
ASG-Größe (bytes)	1176	8104
Erzeugter Code (bytes)	290	3551
Erzeugte Mengen	13	271
Erzeugt gesamt (bytes)	706	5719
Zeit (Sekunden)	3.4	24.5

Tabelle 1: Meßwerte für Gray auf einem 4Mhz 6502-System unter fig-forth

A Benutzerhandbuch

Vorausgesetzt werden die Kenntnis von Forth, von Syntax-beschreibungen in BNF o.ä., und eine Ahnung von Compilerbau.

A.1 Wofür kann man Gray gebrauchen

A.1.1 Syntaktische Analyse einer Programmiersprache

Gray kann für die Grammatiken der meisten Programmiersprachen Parser erzeugen. Womöglich muß die Grammatik noch etwas umgeformt werden, zum Beispiel durch Elimination von Linksrekursionen oder Linksfaktorisierung.

A.1.2 Lexikalische Analyse

Da Grays Code etwa so gut ist wie handgemachter, liegen brauchbare Scanner durchaus im Bereich des Möglichen. Allerdings ergibt sich die Frage, ob man dabei nicht mit Kanonen auf Spatzen schießt – bei Scannern ist keine komplizierte Firstmengenberechnung nötig. Weiters sind die Grammatiken nach der meist notwendigen Linksfaktorisierung wohl nicht viel lesbarer als Forth-Code und schließlich muß noch ein Interface zu den I/O-Routinen geschrieben werden. In Forth bieten sich andere, ungewohnte Möglichkeiten an, deren Einsatz erwogen werden sollte, z.B. der Einsatz des Dictionary.

A.2 Grammatiken

Eine Grammatik beschreibt die Syntax einer Sprache. Tabelle 2 zeigt Grays Grammatik-Ausdrücke (Derzeit müssen allerdings alle Operatoren verdoppelt werden, z.B. "`))`" statt "`)`"). Ein Parser-Generator übersetzt eine Grammatik in einen Parser, der alle Sätze der Sprache und nur diese parsen kann. Man sagt auch, daß die Grammatik die Sätze erzeugt oder ableitet. So kann z.B. α `?` kein oder ein α parsen, α `?` leitet also ϵ (nichts läßt sich schwer hinschreiben) bzw. α ab.

Konstruktion	Schreibweise	parst	Beispiel
Verkettung	`( `α β … `)`	α, dann β, dann …	`( "begin" wortfolge "until" )`
Alternative	`( `α` | `β` | `…` )` [a]	α oder β oder …	`( wort | zahl )`
Leerwort	`eps`	ϵ (nichts)	`eps`
Option	α `?`	null oder ein α	`( "else" wortfolge ) ?`
*-Wiederholung	α `*`	null oder mehrere α	`wort *`
+-Wiederholung	α `+`	ein oder mehrere α	`zeichen +`
Nonterminal	*name*	siehe Kapitel A.2.2	`wortfolge`
Terminal	*name*	siehe Kapitel A.2.1	`"begin"`
Aktion	{ *Forth-Code* }	ϵ (nichts)	{ `+` }

[a] In der Alternative muß die Verkettung nicht geklammert werden, `( `α β` | `γ` )` ist das gleiche wie `( ( `α β` ) | `γ` )`

Tabelle 2: Grays grammatikalische Ausdrücke (α, β und γ bezeichnen solche Ausdrücke)

A.2.1 Terminale und die Schnittstelle zur Eingabe

Die von einem Parser verarbeiteten Grundeinheiten sind Symbole, die von einer darunterliegenden Eingabe-Routine geliefert werden. Diese Symbole können – je nach Komplexität der Eingabe-Routine – einzelne Zeichen oder z.B. Wörter. Soweit es Gray angeht, gibt es für jede Art von Symbolen ein Token, das sie von anderen Symbolen unterscheidet.

Die Eingabe-Routine sollte immer ein Symbol im voraus lesen, damit der Parser aufgrund des zugehörigen Tokens Entscheidungen treffen kann.

Das Interface zur Eingabe besteht aus zwei Teilen:

- In die Variable `test-vector` sollte ein Wort *test?* mit dem Stack-Effekt (set — f) gespeichert werden. *test?* prüft, ob das Token des folgenden Symbols in `set` enthalten ist.[1]

- Mit

  token[2] `singleton ' `*check&read* `terminal` *name*

  kann ein Terminal definiert werden. *name* kann in einer Grammatik dort verwendet werden, wo das zu *token* gehörige Symbol geparst werden soll.

  check&read (f —)[3] wird in den Parser eingebaut und wird beim Parsen aufgerufen, wenn das Symbol geparst werden soll. Wegen der Vorausschau muß *check&read* dann allerdings schon das nächste Symbol lesen und klassifizieren. Daneben überprüft es noch, ob ein Syntax-Fehler vorliegt (dann ist f false; f ist das Ergebnis von *token* `singleton` *test?*) und reagiert entsprechend (siehe Kapitel A.6). Beispiele sind in den Kapiteln B und C zu finden.

[1] Um auf Mitgliedschaft zu prüfen, steht `member?` (set token — f) zur Verfügung

[2] Tokens müssen ganze Zahlen zwischen 0 und einer maximalen Größe sein, die vor der ersten Mengenoperation (z.B. `singleton`) mit `max-member` (u —) deklariert werden muß.

[3] *check&reads* für spezielle Symbole (z.B. Zahlen) werden u.U. entsprechende Aktionen setzen, z.B. den Wert der Zahl stapeln

A.2.2 Nonterminale und Regeln

Mit

$$\alpha \texttt{ <- } name \tag{1}$$

oder

$$\texttt{nonterminal } name \tag{2}$$

$$\alpha \; name \texttt{ rule} \tag{3}$$

kann man *name* als Abkürzung für α definieren; das bedeutet, daß *name* nach seiner Deklaration (1, 2) überall statt α verwendet werden kann.[4]

name ist übrigens ein Nonterminal, (1) und (3) sind Regeln.

Nonterminale dienen nicht nur als Abkürzungen, sondern ermöglichen auch rekursive Definitionen, z.B.

```
nonterminal struktur
struktur * <- struktur-folge
( "if" struktur-folge ( "else" struktur-folge ) ? "endif"
| "while" ...
⋮
| wort ) struktur rule
```

A.2.3 Aktionen

Aus den bisher beschriebenen Ausdrücken können zwar Parser erzeugt werden, für einen Interpreter oder Compiler fehlen allerdings noch die semantischen Aktionen. Syntaktisch verhalten sie sich wie das Leersymbol `eps`, aber wenn sie geparst werden (siehe auch Kapitel A.3), führen sie Forth-Code aus. Wenn z.B. `num` eine Zahl parst und sie stapelt, dann parst

```
( num { . } )
```

eine Zahl und druckt sie aus.

Der Parameter-Stack darf nach Belieben verwendet werden. Dementsprechend sollte für jede Regel der Stack-Effekt dokumentiert werden. Den Return-Stack sollte man nur innerhalb einer Aktion verwenden.

A.2.4 Parser

Mit

$$\alpha \texttt{ parser } name$$

kann aus einem grammatikalischen Ausdruck α, dessen Nonterminale alle durch Regeln definiert sind, ein Parser erzeugt werden, der mit *name* aufgerufen werden kann.

[4] Man könnte statt `<-` auch `constant` verwenden, Gray erzeugt aber dann meistens mehr Code

A.3 Eindeutigkeitsregeln

Gray erzeugt prädiktive Parser; sie versuchen, jeweils aus dem ersten Token zu schließen, welcher Ausdruck jetzt geparst werden soll. Bei vielen Grammatiken ist das nicht möglich – es liegt eine Mehrdeutigkeit vor.

Von Gray erzeugte Parser parsen immer den ersten Ausdruck, der mit dem nächsten Terminalsymbol beginnen kann. Gibt es keinen solchen Ausdruck, so wird die erste (bei Schachtelungen die äußerste) ϵ-Ableitung verwendet. Welche ϵ-Ableitung angewandt wird, ist dann wichtig, wenn damit Aktionen verbunden sind. Wiederholungen werden ausgeführt, solange es geht.

In `( a ? | a b | { ... } )` (a,b sind Terminale) wird in jedem Fall der erste Zweig gewählt.

Bei Option und *-Wiederholung werden mögliche ϵ-Ableitungen im Operanden ignoriert.

α `+` verhält sich wie `(` α α `* )`, α wird also jedenfalls einmal durchlaufen, wobei gegebenenfalls auch eine ϵ-Ableitung ausgeführt wird.

A.4 Warnungen und Fehlermeldungen

Bei fast allen Meldungen wird angegeben, wo sie aufgetreten ist. Bei Verkettung und Alternative wird immer “`)`” oder “`|`” als Fehlerpunkt angezeigt.

A.4.1 Fehler während des Einlesens der Grammantik

no operand Zwischen den Klammern und/oder senkrechten Strichen steht kein Ausdruck. Abhilfe: Fügen Sie einen Ausdruck, z.B. das Leerwort `eps` ein.

multiple rules for nonterminal Für jedes Nonterminal darf es nur eine Regel geben. Abhilfe: Verwenden Sie die Alternative.

A.4.2 Fehlermeldungen während der Parser-Erzeugung

no rule for nonterminal Das Nonterminal wurde zwar deklariert und verwendet, aber es wurde keine Regel definiert.

left recursion In der Grammatik existiert eine Linksrekursion, d.h. beim Parsen eines Nonterminals kann das Nonterminal noch einmal auftreten, ohne daß dazwischen ein Terminalsymbol geparst wurde. Das würde eine Endlosrekursion auslösen und ist daher verboten. Siehe Kapitel A.5.1

A.4.3 Fehler, die nicht auftreten sollten

you found a bug Deutet auf einen Fehler in Gray hin, oder darauf, daß Sie beim Herumspielen etwas kaputtgemacht haben.

A.4.4 Warnungen

deuten darauf hin, daß es mehrere Wege gibt, die Sprache zu parsen. Durch die Eindeutigkeitsregeln (siehe Kapitel A.3) wird zwar ein Weg ausgewählt, aber womöglich ein anderer als der gewünschte. Probleme können sich dadurch bei Aktionen ergeben.

Oft kann eine Grammatik geschrieben werden, deren Parser die gleiche Sprache versteht und die eindeutig ist (siehe Kapitel A.5).

warning: two branches may be empty In der Alternative können mehrere Zweige ϵ ableiten. Die von Gray erzeugten Parser wählen immer den ersten Zweig.

warning: unnecessary option Die Option wurde auf einen Ausdruck angewandt, der schon von sich aus ϵ ableiten kann.

Wenn der Ausdruck nur ϵ parsen kann, wird er nicht durchlaufen.

warning: *-repetition of optional term Ein Ausdruck wurde *-wiederholt, der ϵ ableiten kann.

Wenn der Ausdruck nur ϵ parsen kann, wird er nicht durchlaufen.

conflict: *Konflikt-Menge*[5] Konflikte sind schwerwiegender als die anderen Warnings, da sie dazu führen können, daß nicht alle Sätze der Sprache geparst werden können, daß also der Parser die Sprache nicht versteht.

Ein Konflikt liegt dann vor, wenn der Parser eine Entscheidung treffen muß, wie nun weiter geparst werden soll, diese Entscheidung aus dem nächsten Token aber nicht hervorgeht. Es gibt dann eine Menge von Tokens, die beide Möglichkeiten einleiten können, die Konfliktmenge.

Beispiel:

```
( "a" ? "a" )
```

sollte "a" und "aa" parsen können. Wenn der Parser aber ein "a" sieht, weiß er nicht, ob es das erste oder das zweite ist.

A.5 Umformung von Grammatiken

Um die durch Linksrekursionen und Konflikte entstehenden Probleme zu vermeiden, kann die Grammatik so verändert werden, daß die Sprache erhalten bleibt, die Probleme aber verschwinden. Ich beschränke mich hier auf ein paar Beispiele, Algorithmen sind in [1] zu finden.

A.5.1 Elimination von Linksrekursionen

Eine Grammatik ist linksrekursiv, wenn beim Parsen eines Nonterminals das Nonterminal noch einmal vorkommen kann, ohne daß inzwischen ein Terminal geparst werden muß. Da Linksrekursionen zu Endlosrekursion führen kann, ist sie verboten.

Einfache Linksrekursionen folgen dem Schema

`( N` α `|` β `) <- N`

`N` steht also für β, $\beta\alpha$, $\beta\alpha\alpha$, …, und kann durch

`(` β α `* ) <- N`

ersetzt werden.

[5] Die Konfliktmenge wird als Zahlenfolge ausgegeben; ist eine andere Art der Ausgabe erwünscht, so ist die Exekutionsadresse des entsprechenden Ausgabe-Worts (token —) in die Variable `print-token` zu speichern.

A.5.2 Linksfaktorisieren

In

$(\ \alpha \ \beta \ \mid \ \alpha \ \gamma \ \mid \ \delta \ )$

besteht ein Konflikt zwischen dem ersten und dem zweiten Zweig der Alternative. Dieser Konflikt kann gelöst werden, indem die Entscheidung zwischen den beiden Möglichkeiten verschoben wird, bis klar ist wie sie zu treffen ist:

$(\ \alpha \ (\ \beta \ \mid \ \gamma \ ) \ \mid \ \delta \ )$

Normalerweise sind die Konflikte etwas komplizierter und erfordern u.U. gröbere Umformungen der Grammatik, sodaß sie nachher nicht mehr besonders ansehnlich ist.

A.6 Fehlerbehandlung

Wie soll der Parser auf Syntax-Fehler reagieren? Die einfachste Methode ist, den Parse-Vorgang abzubrechen, und eine möglichst sinnvolle Fehlermeldung auszugeben.

Eine einfache Möglichkeit für diese Fehlermeldung ist die Ausgabe der Symbole, die der Parser erwartet hat. Diese Menge ist leicht zu erhalten, indem die überprften Mengen (die Argumente von *test?*) vereinigt werden. Ein Beispiel ist in Kapitel B zu finden.

Es gibt noch viele andere Arten der Fehlerbehandlung, deren Beschreibung allerdings den Rahmen dieses Handbuches sprengen würde. Ich verweise daher auf die Literatur [1] und führe hier nur noch eine einfache Technik an, um bei häufig vorkommenden Fehlern nicht sofort abbrechen zu müssen: Fehler-Regeln.

Die Grammatik der Sprache wird so erweitert, daß auch Programme, die häufige Fehler enthalten, geparst werden können. Allerdings sollten die Fehler Fehlermeldungen auslösen.

Beispiel: In Pascal und ähnlichen Sprachen pflegt der Strichpunkt zwischen den Statements vergessen zu werden. Ursprünglich lautet die Syntax

```
( statement ( ";" statement ) * ) <- StatementSequence
```

Soll der Benutzer nicht wegen jedes vergessenen Strichpunkts zum Editor geschickt werden, kann die Regel wie folgt geändert werden:

```
( statement
  ( ( ";" | { ." ; inserted" pascal-error } ) statement ) *
) <- StatementSequence
```

A.7 Eingeweide von Gray

Viele Forth-Programmierer werden Gray erweitern oder verändern wollen. Damit sie sich leichter zurechtfinden, folgt hier ein kleiner Überblick. Beim Einlesen der Grammatik bauen die Wörter `terminal`, `?`, `<-`, `)` etc. zunächst einen abstrakten Syntax-Graphen (ASG) der Grammatik im Speicher auf. Für die meisten Konstruktionen wird dabei ein Knoten erzeugt, Verkettung und Alternative dagegen werden mit Hilfe der zweistelligen Operatoren `concat` und `alt` in $n-1$ zweistellige Knoten übersetzt, was die weitere Arbeit erleichtert.

Durch Aufruf von `parser` wird in zwei Durchgängen aus dem ASG der Parser erzeugt. Der erste – `propagate` – sorgt für die Berechnung der Follow-Mengen[6], die nur für die Erkennung von Konflikten gebraucht werden. Der zweite Durchgang – `pass2` – berechnet die restlichen nötigen First-Mengen[7], meldet Linksrekursionen, Konflikte und sonstigen Fehlermöglichkeiten, und erzeugt Code für alle Regeln. Abschließend erzeugt `parser` noch den Code für das Startwort.

Für die Berechnung der Firstmenge eines ASG-Knoten und für die Code-Erzeugung gibt es kleine Unterpässe (`compute` und `generate`), die jeweils die Teile des ASG durchwandern, die sie brauchen. `compute` sorgt auch für das Aufdecken von Linksrekursionen: Derselbe (Nonterminal-)Knoten kommt genau dann mehrmals in der Berechnung vor, wenn eine Linksrekursion vorliegt. Daher wird `compute` für alle Knoten aufgerufen.

Um undurchsichtige Stapel-Manipulations-Exzesse zu vermeiden, setzte ich einen Kontext-Stapel ein. Auf diesen kommt der ASG-Knoten des jeweils zu bearbeitenden Ausdrucks, auf dessen Felder (Kontext-Variablen und -Konstanten) dann einfach über ihren Namen zugegriffen werden kann. Den aktuellen Knoten erhält man mit `this`. Kontext-Variablen und -Konstanten verhalten sich also ähnlich wie Smalltalks Instance Variables, der Zugriff kann allerdings überall erfolgen und Kontext-Stacks müssen explizit manipuliert werden.

Als weitere Anleihe bei der objekt-orientierten Programmierung habe ich Methoden und Maps[5] verwendet. Die Wörter `compute`, `generate`, `propagate` und `pass2` haben sowohl eine allgemeine, für alle Knotenarten gültige Aufgabe, als auch eine spezielle, von der Knotenart abhängige. Der spezielle Code wird über eine Vektor-Tabelle (Map) aufgerufen, auf die das Feld Methods des aktuellen Knotens zeigt. Diesen Vorgang führen die mit dem Feld-definierenden Wort `method` definierten Wörter automatisch durch.

Die Knoten lassen sich in eine Klassenstruktur einteilen, was sich in gemeinsamen Datenstrukturen und Code ausdrückt:

```
syntax-expr
   terminal
   eps
   nonterminal
   action
   unary
      option&repetition
         option
         repetition
            *repetition
            +repetition1
   binary
      concatenation
      alternative
```

[6] Die Follow-Menge eines Grammatikalischen Ausdrucks enthält die Tokens der Terminalsymbole, die dem Ausdruck folgen dürfen.

[7] Die First-Menge eines Ausdrucks enthält die Tokens der Terminale, mit denen der Ausdruck beginnen kann. Wenn der Ausdruck ϵ ableiten kann, enthält die First-Menge auch noch ϵ. In Gray wird das allerdings mit einem Extra-flag namens `maybe-empty` angezeigt. Der Parser benötigt First-Mengen, um entscheiden zu können, wie geparst werden soll (bei Alternative, Option und Wiederholungen).

A.7.1 Beispiel einer Erweiterung

In Pascal und ähnlichen Sprachen gibt es viele Syntax-Ausdrücke der Art

(α (β α) *)

die wir Listen nennen wollen. Der geübte Software-Ingenieur hebt die Gemeinsamkeiten der Ausdrücke natürlich heraus, indem er einen neuen Operator einführt: α β `&`. Dieser kann ganz einfach als Abkürzung für die obere Syntax definiert werden:

```
: & ( syntax-expr1 syntax-expr2 -- syntax-expr3 )
 over concat[8] * concat ;
```

Bei der Anwendung werden zwei Pointer auf *syntax-expr1* erzeugt, also ein DAG. Das ist erlaubt, Zyklen dagegen müssen Nonterminale enthalten, da sonst Endlosrekursionen in `generate` entstehen.

Das wäre nun ohne weiters brauchbar, aber als Beispiel für kompliziertere Erweiterungen ist in Abbildung 2 ein Programm zu finden, das

```
begin
 [ α generate ]
 ... test? while
 [ β generate ]
repeat
```

statt

```
[ α generate ]
begin
 ... test? while
 [ β generate ]
 [ α generate ]
repeat
```

erzeugt.

`&` erzeugt einen zweistelligen Knoten (das zusätzliche Feld wird später erklärt), dessen Map auf listenspezifische Wörter zeigt, sodaß eine Liste anders bearbeitet wird als eine Verkettung. Diese Wörter müssen nun definiert werden.

`generate-list` ist am einfachsten, denn wir wissen schon fast, was wir wollen. Die einzige offene Frage ist, was getestet werden soll, wann also ein Durchgang angehängt werden soll: genau dann, wenn voraussichtlich noch ein Durchgang angehängt werden kann, also wenn das nächste Token in der First-Menge von (β α) enthalten ist. Diese Menge ist die First-Menge von β, wenn β ϵ ableiten kann, scheint α durch und seine First-Menge muß dazugenommen werden. Während `generate` darf allerdings kein Speicher angefordert werden und daher sind auch Mengenoperationen verboten. Die Testmenge wird daher schon vorher in `pass2-list` berechnet und im Feld `test-set` gemerkt.

Wenden wir uns der First-Mengen-Berechnung (`compute-list`) zu. Normalerweise ist sie die First-Menge von α, bei leerem α scheint aber die First-Menge von β durch und muß dazugenommen werden. Wenn β ϵ ableitet, kommt noch

[8] Ich verwende hier `concat`, da die Klammerschreibweise mehr Stapel-Jonglieren verlangt

```
binary-syntax-expr
 cell context-var test-set
constant list-syntax-expr

: compute-list ( -> first maybe-empty )
 operand1 compute dup if
  swap operand2 get-first union swap
 endif ;

: propagate-list ( follow -> )
 operand2 compute if
  operand1 get-first union
 endif
 union
 dup operand1 propagate ( follow1 )
 operand1 compute if
  union
 else
  swap drop
 endif
 operand2 propagate ;

: generate-list ( -> )
 [compile] begin
 operand1 generate
 test-set @ compile-test
 [compile] while
 operand2 generate
 [compile] repeat ;

: pass2-list ( -> )
 operand2 compute if
  operand1 get-first union
 endif
 dup test-set !
 follow-set @ check-conflict
 pass2-binary ;

create list-map
', compute-list
', propagate-list
', generate-list
', pass2-list

: && ( syntax-expr1 syntax-expr2 -> syntax-expr3 )
 list-map make-binary 0 , ;
```

Abbildung 2: Listen als neue Syntax-Konstruktion die First-Menge von α dazu usw., aber das ändert nichts mehr und findet sich daher auch nicht im Programm wieder. Der Ausdruck leitet genau dann ϵ ab, wenn α ϵ ableitet.

Bei `propagate-list` geht es genau umgekehrt wie bei `compute-list` – die Follow-Menge kommt von außen und muß – eventuell verändert – an die Operanden weitergegeben werden. Kann kein Operand ϵ ableiten, folgt auf β ein Token aus der First-Menge von α, auf α eines aus der First-Menge von β oder aus der Follow-Menge des Ausdrucks, die Follow-Menge von α ist also die Vereinigung dieser Mengen; Kann jedoch ein Operand ϵ ableiten, dann scheint seine Follow-Menge zum anderen Operanden durch.

In `pass2-list` müssen Konflikte erkannt und gemeldet werden und `pass2` für die Argumente aufgerufen werden. Da die letzte Aufgabe für alle zweistelligen Knoten gleich ist, wird sie von `pass2-binary` übernommen. Einen Konflikt gibt es dann, wenn es Tokens gibt, die beide Entscheidungen ermöglichen, wenn also die Follow-Menge nicht disjunkt zur Test-Menge ist.

B calc – Ein kleiner Interpreter

```
( a little calculator )

255 max-member ( the whole character set )
variable sym

10 stack expected

: testsym ( set -> f )
 dup expected push
 sym @ member? ;

' testsym test-vector !

: ?syntax-error ( f -> )
 not if
  empty begin
   expected top union
   expected pop
   expected clear? until
  ." expected:" ['] emit apply-to-members abort
 endif ;

: ?readnext ( f -> )
 ?syntax-error
 expected clear
 0 begin
  drop key
  dup 32 = not until
 sym ! ;

: init
 true ?readnext ;

: t ( -> ) ( use: t c name )
 ( make terminal name with the token c )
 [compile] ascii singleton ['] ?readnext terminal ;

: x ( set1 -> set2 )
 ( read a char from the input and include it in the set )
 [compile] ascii singleton union ;

( make a terminal that accepts all digits )
empty x 0 x 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8 x 9
  ' ?readnext terminal digit

t ( "("
t ) ")"
```

```
t + "+"
t - "-"
t * "*"
t / "/"
t = "="

nonterminal expr

(( {{ 0 }}
   (( {{ 10 * sym @ ascii 0 - + }} digit )) ++
)) <- num ( -> n )

(( num
|| "(" expr ")"
)) <- factor ( -> n )

(( factor (( "*" factor {{ * }}
          || "/" factor {{ / }}
          )) **
)) <- term ( -> n )

(( (( term
   || "-" term {{ 0 swap - }} ))
   (( "+" term {{ + }}
   || "-" term {{ - }} )) **
)) expr rule ( -> n )

(( {{ init }} expr "=" {{ . }} )) parser ? ( -> )
```

C mini – Ein kleiner Compiler

```
( mini )
( you have to type one symbol and on character after the end of )
( the program because of the lookahead of parser and scanner )

( scanner ------------------------------------------------------ )
( it is implemented using gray to give an example )
( that's probably not the best way )
255 max-member ( the whole character set )

variable tokenval 0 tokenval !
: token ( -- ) ( use: token name ) ( name: -- n )
 ( defines a token that returns a unique value )
 tokenval @ constant
 1 tokenval +! ;

token ";"
token ","
token ":="
token "="
token "#"
token ">"
token "+"
token "-"
token "*"
token "("
token ")"
token Ident
token Number

vocabulary keywords keywords definitions
token PROGRAM
token VAR
token BEGIN
token END
token Read
token Write
token IF
token THEN
token WHILE
token DO
forth definitions

variable numval
variable identp
variable identlen

: adds ( addr1 c -> addr1+1 )
```

```
 ( accumulates char to string )
 over c! 1+ ;

: key/ident ( addr -- n )
 ( checks string at addr for keyword and returns token value )
 ['] keywords lookup if
  execute
 else
  drop Ident
 endif ;

variable ch

: testchar? ( set -- f )
 ch c@ member? ;
' testchar? test-vector !

: ?nextchar ( f -- )
 not abort" non-mini character or '=' missing after ':'"
 key ch c! ;

: .. ( c1 c2 -- set )
 ( creates a set that includes the characters c, c1<=c<=c2 )
 empty copy-set
 swap 1+ rot do
  i over add-member
 loop ;

: ` ( -- terminal ) ( use: ` c )
 ( creates anonymous terminal for the character c )
 [compile] ascii singleton ['] ?nextchar make-terminal ;

ascii a ascii z ..  ascii A ascii Z ..  union
  ' ?nextchar  terminal letter
ascii 0 ascii 9 ..  ' ?nextchar  terminal digit
0 32 ..  ' ?nextchar  terminal space

(( space **
   (( ` ;      {{ ";"  }}
   || ` ,      {{ ","  }}
   || ` : ` =  {{ ":=" }}
   || ` =      {{ "="  }}
   || ` #      {{ "#"  }}
   || ` >      {{ ">"  }}
   || ` +      {{ "+"  }}
   || ` -      {{ "-"  }}
   || ` *      {{ "*"  }}
   || ` (      {{ "("  }}
   || ` )      {{ ")"  }}
```

```
   || {{ 0 }}
      (( {{ 10 * ch c@ + ascii 0 - }} digit )) ++
      {{ numval !  Number }}
   || {{ here identp ! here ch c@ adds }} letter
      (( {{ ch c@ adds }} (( letter || digit )) )) **
      {{ 0 adds  here - identlen !  here key/ident }}
   ))
)) <- symbol

symbol parser scan

( table handler --------------------------------------------- )
( uses forths dictionary mechanism )

vocabulary variables ( for mini variables )

( enter )
: $var ( addr -- )
 ( defines variable whose name [0-terminated] addr points to )
 ( very tile-dependent )
 ['] variables lookup abort" variable already defined"
 >r 0 0 2 r> entry ;

( lookup )
: getvar ( addr -- word )
 ( get the execution address of the var with name at addr )
 ['] variables lookup not abort" variable undefined" ;

( code generator -------------------------------------------- )
( nearly everything already exists in forth )

: $prog ( addr -- )
 ( defines colon-def to whose name (0-terminated) addr points )
 >r here 0 1 r> entry ] ;

( parser ---------------------------------------------------- )
tokenval @ 1- max-member

variable sym

: testsym? ( set -- f )
 sym @ member? ;
' testsym? test-vector !

: ?nextsym ( f -- )
 not abort" syntax error"
 scan sym ! ;

: t ( n -- ) ( use: token t name )
```

```
( takes a token and makes a terminal asg node )
 singleton ['] ?nextsym terminal ;

( terminals have the same name as their tokens )
( just for convenience )
";" t ";"
"," t ","
":=" t ":="
"=" t "="
"#" t "#"
">" t ">"
"+" t "+"
"-" t "-"
"*" t "*"
"(" t "("
")" t ")"
Ident t Ident
Number t number
PROGRAM t PROGRAM
VAR t VAR
BEGIN t BEGIN
END t END
Read t "Read"
Write t "Write"
IF t IF
THEN t THEN
WHILE t WHILE
DO t DO

: <> ( n1 n2 -- f )
 = not ;

nonterminal Stat
nonterminal Expr

(( {{ numval @ }} number )) <- Number

(( Number {{ [compile] literal }}
|| {{ identp @ getvar (compile) compile @ }} Ident
|| "(" Expr ")"
)) <- Factor

(( Factor (( "*" Factor {{ compile * }} )) ** )) <- Term

(( Term
   (( (( "+" {{ ['] + }} || "-" {{ ['] - }} ))
      Term {{ (compile) }}
   )) **
)) Expr rule
```

```
(( Expr
   (( "=" {{ [’] = }} || "#" {{ [’] <> }} || ">" {{ [’] > }} ))
   Expr {{ (compile) }}
)) <- Cond

Stat ";" && ?? <- StatSeq

(( "Read" {{ identp @ getvar (compile) compile ! }} Ident
)) <- ReadStat

(( "Write" Expr )) <- WriteStat

(( {{ identp @ getvar }} Ident ":=" Expr {{ (compile) compile ! }}
)) <- AssStat

(( IF Cond {{ [compile] if }} THEN StatSeq END {{ [compile] endif }}
)) <- IfStat

(( {{ [compile] begin }} WHILE Cond {{ [compile] while }} DO
   StatSeq END {{ [compile] repeat }}
)) <- WhileStat

(( ReadStat || WriteStat || AssStat || IfStat || WhileStat
)) Stat rule

(( VAR {{ variables definitions }}
   (( {{ identp @ $var }} Ident )) "," &&
   {{ forth definitions }}
)) <- Decl

(( PROGRAM {{ identp @ identlen @ allot }} Ident  Decl ??
   {{ $prog }} BEGIN StatSeq {{ [compile] ; }} END
)) <- Program

Program parser parsemini

: mini ( -- ) ( use: mini program )
( get the first char and symbol, then parse mini )
 true ?nextchar true ?nextsym parsemini ;

( interpreter ---------------------------------------------------- )
( forth itself )
```

Literatur

[1] Alfred V. Aho, Ravi Sethi, Jeffrey B. Ullman, *Compilers. Principles, Techniques and Tools*, Addison-Wesley, 1986

[2] M.Brockhaus, A.Falkner, *Übersetzerbau*, Institut für praktische Informatik, Technische Universität Wien, 1988

[3] Ronald Zech, *Die Programmiersprache Forth*, Francis Verlag Gmbh, München, 1984

[4] Michael A. Perry, "A 68000 Forth Assembler", *Dr. Dobbs Toolbook of Forth*, Marlin Ouverson (ed), M&T Books, Redwood City, CA, 1986

[5] Craig Chambers, David Ungar, Elgin Lee, "An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language based on Prototypes", *OOPSLA '89 Proc.*, ACM Press 1989