GRAY – ein Generator für rekursiv absteigende Ybersetzer

M. Anton Ertl
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
anton@mips.complang.tuwien.ac.at
http://www.complang.tuwien.ac.at/anton/
Tel.: (+43-1) 58801 4474
Fax.: (+43-1) 505 78 38

Zusammenfassung

Der Parser-Generator Gray übersetzt Grammatiken in ausführbaren Forth-Code für einen Parser. Als Besonderheit ist dabei die problemlose Kombination semantischen Aktionen und Erweiterungen der BNF zu nennen, die sich aus der Verwendung des Stacks zur Kommunikation zwischen den semantischen Aktionen ergibt. Dieser Artikel beschreibt den Entwurf und die Verwendung von Gray.

1 Motivation

Forth wird oft als Sprache zur Schaffung anwendungsspezifischer Sprachen bezeichnet, als Meta-Sprache [Zec84]. Meist wird die neue Sprache einfach durch die Definition entsprechender Forth-Wörter implementiert. Forth's Flexibilität unterstützt diese Methode, hilfreich sind insbesondere Immediate-Wörter und die für den Forth-Compiler benötigten Wörter (z.B. `POSTPONE`). Vorteilhaft ist dabei, daß kein Parser benötigt wird, daß sich die Schnittstelle zu Forth einfach gestaltet und somit vorhandene Wörter und Programme in der neuen Sprache benutzt werden können, und daß Forth's Werkzeuge (Debugger etc.) 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, viele Forth-Assembler [Zec84, Per86], 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 schwer lesbar. Besonders bei größeren Grammatiken zahlt sich die automatische Übersetzung aus.

2 Andere Arbeiten

Es gibt eine große Menge von Parser-Generatoren für verschiedene Zielsprachen [SR].

Für Forth gibt es Parser-Interpreter von Mikael Patel und Brad Rodriguez [Rod90], die auf Top-down-Parsen mit Backtracking basieren.

[IH90] beschreibt zwei Parser-Generatoren: Der erste erzeugt rekursiv absteigende Parser, wie Gray; der zweite besteht aus einem Pascal-Programm, das eine Tabelle für einen tabellengesteuerten Top-down-Parser erzeugt, und aus einem Interpreter für diese Tabelle in Forth.

FOSM [Cha91, Cha92, Jak96] ist ein Stringmustererkenner, der mit Backtracking arbeitet. Er ist zwar sehr ausdrucksstark, sollte dafür aber nicht auf sehr lange Strings angewendet werden.

3 Terminologie

Ein *Parser* überprüft, ob die Eingabe*sequenz* in einer *Sprache* enthalten ist. Die Sprache wird durch eine *Grammatik* beschrieben. Die Grammatik enthält grammatikalische *Ausdrücke*, die Teile der Sprache beschrieben. Manchmal ist es nützlich, die Beziehung zwischen Grammatik(ausdrücken) und Sequenzen nicht unter dem Blickwinkel des Prüfens von Sequenzen, sondern unter dem des Erzeugens von Sequenzen aus der Grammatik zu betrachten; man sagt dann, die Grammatik *leitet* die Sequenz *ab*. Die Sequenzen bestehen aus *Terminalsymbolen*. In der Grammatik stehen *Nonterminale* für bestimmte Grammatikausdrücke. Die Überprüfung einer Sequenz alleine ist nicht sehr nützlich; daher werden auch noch *Aktionen* während des Parsens durchgeführt, die aus dem Parser einen *Compiler* oder *Interpreter* machen.

4 Entwurf

Mein Hauptziel war es, einen brauchbaren Parser-Generator zu bauen. Daneben sollte er möglichst einfach, portabel, und klar programmiert sein. Ich ent-

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
Forth-Tagung 1997, Ludwigshafen

schied mich daher dafür, rekursiv absteigende Parser [ASU86] zu erzeugen, da die Übersetzung von EBNF o.ä. relativ einfach ist.

Im Gegensatz zur sonstigen Forth-Praxis investierte ich relativ viel Arbeit in die Erkennung von möglichen Fehlern in der Grammatik (insbesondere Linksrekursionen und Konflikte), weil solche Fehler sich beim Testen gerne verstecken (wie ich aus Erfahrungen mit Prolog's DCGs wußte). Im Sinne der Einfachkeit, Portabilität und Klarheit ließ ich Optimierungen, Fehlerbehandlung im resultierenden Parser und automatische Grammatiktransformationen weg.

4.1 Syntax

Als Schreibweise für die Grammatiken wählte ich eine um Operatoren der regulären Ausdrücke erweiterte BNF (auch bekannt als *regular right part grammar* RRPG). Da die zusätzlichen Operatoren (+,∗,?) Postfix-Operatoren sind und diese Operatoren sich in Forth-Kontrollstrukturen übersetzen lassen, sind diese Konstruktionen sehr einfach zu implementieren.

Die Grammatik-Konstruktionen Verkettung und Alternative bereiteten etwas mehr Probleme, da sie in den Lehrbüchern ohne Operatoren bzw. mit infix-Operatoren dargestellt sind. Für diese Konstruktionen verwendete ich als Grundlage Postfix-Operatoren und fügte als syntaktischen Zucker noch eine Klammerschreibweise hinzu, die eine lehrbuchnahe Schreibweise erlaubt (abgesehen vom Zwang zur expliziten Klammerung solcher Konstruktionen). Eine weitere Abweichung von den Lehrbüchern ist die umgekehrte Schriebweise der Regeln (`Ausdruck <- Nonterminal` statt *Nonterminal* → *Ausdruck*), aber daran sind Forth-Benutzer ja gewöhnt.

Um Konflikte mit existierenden Wörtern zu vermeiden, wurden die Operatoren und Klammern in Doppelschreibweise ausgeführt (z.B. `((`, `++` statt `(`, `+`). Ursprünglich war geplant, die Operatoren statt dessen in ein eigenes Vocabulary[1] zu stellen und den richtigen Operatorsatz über die Suchreihenfolge (search order) festzulegen, sobald diese Methode portabel würde. Als es mit dem Erscheinen des ANS Forth Standards soweit war, behielt ich allerdings die Doppelschreibweise bei, einerseits aus Gründen der Kontinuität und Kompatibilität, andererseits weil die Lösung über die Suchordnung wieder andere Probleme aufgeworfen hätte.

4.2 Erzeugter Code

Die verschiedenene Grammatikkonstruktionen werden letztendlich wie folgt in ausführbaren Forth-Code übersetzt:

Regel: Definition.

Nonterminal: Aufruf der Definition für die Regel des Nonterminals.

[1] heute auch als Wordlist bekannt.

Terminalsymbol: Aufruf der Aktion des Terminalsymbols.

`eps`: Nichts.

Verkettung `((` α β `))`**:** code(α) code(β).

Alternative `((` α `||` β `))`**:** ... `if` code(α) `else` code(β) `then`.

Option α `??`**:** ... `if` code(α) `then`.

Wiederholung α `**`**:** `begin` ... `while` code(α) `repeat`.

Wiederholung α `++`**:** `begin` code(α) ... `until`.

Aktion {{ *`code`* }}**:** Aufruf einer anonymen Definition (mit `:noname` definiert), die *code* enthält.

Die schwierigen Teile („...“) habe ich weggelassen, nämlich die Berechnung des Flags für die Entscheidung, wie weitergeparst werden soll; im Prinzip schaut der Parser nach, ob α mit dem nächsten Terminalsymbol der Eingabe anfangen kann; wenn ja, entscheidet sich der Parser dafür, α zu parsen, sonst wählt er die andere Alternative.

4.3 Struktur

Beim Einlesen einer Grammatik bauen die Wörter `++`, `))` etc. einen abstrakten Syntaxbaum für die Grammatik auf. Die verschiedenen Arten von Knoten dieses Baumes 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
```

Alle Knoten haben u.a. folgende Felder:

first-set Der Code, der für den Knoten (und seine Kinder) erzeugt wird, kann eine bestimmte Menge von Sequenzen parsen. Die Menge der Terminalsymbole, mit denen diese Sequenzen beginnen, ist das first-set.

maybe-empty Wenn unter den Sequenzen auch die leere Sequenz ist.

follow-set Die Menge der Terminalsymbole, die hinter den von diesem Knoten geparsten Sequenzen stehen können.

First-set und maybe-empty werden verwendet, um beim Parsen bei einer Wahlmöglichkeit (Alternative, Option, Wiederholungen) zwischen den verschiedenen Möglichkeiten zu entscheiden. Follow-set dient bei Gray ausschließlich dazu, um beim Erzeugen des Parsers Konflikte zu erkennen.

Die folgenden Wörter operieren auf allen Arten von Knoten:

compute Berechnet first-set und maybe-empty des Knotens.

generate Erzeugt den Code für den Knoten und seine Kinder.

propagate Der erste Durchgang von Gray: Berechnet alle follow-sets.

pass2 Der zweite Durchgang: Berechnet die nötigen First-Mengen, überprüft auf Linksrekursionen und Konflikte, und erzeugt den Code für die Regeln.

Diese Struktur eignet sich gut für die objektorientierte Programmierung. Gray enthält eine entsprechende, für seinen Zweck spezialisierte Forth-Erweiterung (15 Zeilen inkl. Kommentar).

Um undurchsichtige Stapel-Manipulations-Exzesse zu vermeiden, setzte ich einen Kontext-Stapel ein. Auf diesen kommt der Knoten des jeweils zu bearbeitenden Ausdrucks, auf dessen Felder (z.B. first-set) dann einfach über ihren Namen zugegriffen werden kann (wie bei Smalltalks Instanzvariablen). Ähnlich wie Gray verwenden einige objekt-orientierte Forth-Erweiterungen spezielle Objekt-Stacks.

4.4 Geschichte

Ich implementierte Gray im Sommer 1989 innerhalb weniger Tage. 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 und modulare Programmierung schuld sein.

Ich portierte Gray dann im darauffolgenden Winter auf das Forth-System TILE und machte Gray öffenlich verfügbar. Im Sommer 1994 erfolgte dann die Anpassung an ANS Forth.

5 Besonderheiten durch Forth

Abgesehen von den üblichen Besonderheiten von Forth, wie der Postfix-Notation, der Interaktivität, und dem Fehlen von Typprüfungen wirkt sich die Verwendung von Forth in folgender Weise besonders aus:

5.1 Stack

Forths auffallendste Eigenschaft ist der Stack. Dieser hatte zwar auf die Implementation keinen besonderen Einfluß, ist aber beim Schreiben eines Compilers sehr vorteilhaft:

In den Lehrbüchern werden die beim Parsen durchgeführten Aktionen als attributierte Grammatik dargestellt. In attributierten Grammatiken erfolgt der Informationsfluß über benannte Attribute[2] und hält sich genau an die Struktur der Grammatik. Wenn man, z.B. durch Konflikte, gezwungen ist, die Grammatik umzuformen, wird es sehr aufwendig, die richtigen Attributierungsregeln hinzuschreiben. Außerdem kann der Formalismus der attributierten Grammatik nicht mit Konstruktionen wie Wiederholung und Option erweitert werden.

In Gray erfolgt der Informationsfluß dagegen über den Stack, was eine gewisse Unabhängigkeit des Informationsflußes von der Struktur der Grammatik erlaubt. Daher ist auch die Einführung von Erweiterungen der BNF kein Problem.

Als Beispiel nehmen wir Ausdrücke der Form $x - x - x ...$, die implizit so geklammert werden sollten: $(...(x - x) - x)...)$. Das sollte sich in der Struktur des abstrakten Syntaxbaums wiederspiegeln. Eine attributierte Grammatik dafür wäre:

```
expr: expr '-' term
     {expr.tree :=
      make_node('-',expr.1.tree,term.tree);}
    | term { expr.tree := term.tree; }
```

Diese Grammatik enthält eine Linksrekursion. Will man diese entfernen (z.B., weil man einen LL(1)-Parser-Generator verwendet), müssen auch die Attributierungsregeln umgeschrieben werden, und zwar in eine wesentlich kompliziertere Form:

```
expr: term expr1 { expr1.in = term.tree;
                   expr.tree = expr1.tree; }

expr1: '-' term expr1
       { expr1.1.in =
         make_node('-', expr1.in, term.tree);
         expr1.tree = expr1.1.tree;
     | eps { expr1.tree = expr1.in }
```

Bei Gray ist die Lösung dagegen viel einfacher:

```
(( term
   (( '-' term {{ [char] - make-node }} )) **
)) <- expr ( -- tree )
```

[2] yacc ist da etwas unbequemer, da muß man auf die Attribute mit $$, $1 etc. zugreifen.

Nun stellt sich natürlich die Frage, ob der Informationsfluß nicht undurchschaubar wird, wenn er von der Struktur der Grammatik nicht vollkommen abhängig ist. Die Abhilfe für dieses Problem ist in Forth-Kreisen schon lange bekannt: der Stack-Kommentar. Jede Regel sollte mit einem Stack-Kommentar ausgestattet sein, der den Stack-Effekt der Regel beschreibt. Wenn nötig, kann man auch Stack-Kommentare in die Regeln hineinschreiben, die den Stack-Zustand an diesen Stellen beschreiben.

Wenn in den Aktionen Code erzeugt wird, sollten auch noch (entsprechend gekennzeichnete) Stack-Kommentare hinzugefügt werden, die den Stack-Effekt dieses Codes zur Laufzeit beschreiben.

5.2 Metaprogrammierung

Forth-Programme haben Zugriff auf Forth-Compiler-Wörter, was das Schreiben von Compilern sehr erleichtert. Gray selbst erzeugt daher nicht eine Datei, die dann von einem Forth-Compiler weiterverarbeitet wird, sondern erzeugt den Parser direkt im Speicher. Für die sonst nötige Symboltabelle wird einfach Forths Dictionary verwendet.

Genauso wird ein mit Gray geschriebener Compiler meistens seinen Code direkt als ausführbaren Forth-Code erzeugen und das Dictionary für die Symboltabellen verwenden. Der resultierende Compiler wird üblicherweise recht schnell sein (wenn er nicht selbst viel Zeit verbraucht. Die Geschwindigkeit des erzeugten Codes hängt vom verwendeten Forth-System ab und dürfte bei derzeit erhältlichen Native-Code-Systemen ca. zweimal langsamer sein als Code, der von einem optimierenden C-Compiler erzeugt wird [EM95]; bei threaded-code-Systemen dürfte der erzeugte Code ca. 5–10 mal langsamer sein als der von einem optimierenden C-Compiler erzeugte.

Ein weiterer Vorteil von Forth bei der Metaprogrammierung ist die fehlende Typüberprüfung. Die meisten (alle?) Typsysteme sehen solche Verwendungen einfach nicht vor, und würden sie verbieten. Bei Gray würde wahrscheinlich zumindest die Implementation der Syntax einer Typprüfung zum Opfer fallen.

5.3 Sonstiges

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.

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

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

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.

6 Erfahrungen und Resultate

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. Abb. 1 zeigt Daten über Grays Effizienz. Auf modernen Maschinen ist die verbrauchte Zeit nicht mehr sinnvoll meßbar. Dafür ist die ANS Forth-Version viel gefräßiger, da mit jedem Knoten des Baums gleich die ganze Zeile mitgespeichert wird, in der der Knoten vorkommt (ANS Forth bietet keinen Zugriff auf die aktuelle Zeilennummer).

Die mir bekannten Anwendungen sind:

- Bei Gforth wird ein kleiner Compiler verwendet, der die Spezifikation der Primitives in verschiedene Files, übersetzt, unter anderem in C-Code [Ert93]. Dieser Compiler ist mit Hilfe von Gray in Forth geschrieben. Eine erste Version dieses Compilers war in Emacs-Lisp implementiert; da sich diese Sprache aber mit jeder Emacs-Version änderte, wurde er dann in Forth neu geschrieben. Die Forth-Version ist zwar um einiges länger als die Elisp-Version, aber auch beträchtlich schneller und verständlicher.
- Marcel Hendrix hat einen Compiler für eine kleine Pascal-ähnliche Sprache mit Gray und iForth implementiert.

7 Zusammenfassung

Gray ist ein Parser-Generator, der RRPG-Grammatiken in rekursiv absteigende Parser in Form von exekutierbarem Forth-Code übersetzt. Gray ist frei erhältlich auf http://www.complang.tuwien.ac.at/forth/gray4.tar.gz (bzw. gray4.zip).

Konstruktion	Schreibweise	parst	Beispiel
Verkettung	`((` α β `... ))`	α, dann β, dann ...	`(( "begin" wortfolge "until" ))`
Alternative	`((` α `||` β `|| ... ))`[3]	α oder β oder ...	`(( wort || zahl ))`
Leerwort	`eps`	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 Abschnitt A.2.2	`wortfolge`
Terminal	*name*	siehe Abschnitt A.2.1	`"begin"`
Aktion	{{ *Forth-Code* }}	nichts	`{{ + }}`

Abbildung 2: Grays grammatikalische Ausdrücke

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. Abb. 2 zeigt Grays Grammatik-Ausdrücke. 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 nichts bzw. α ab.

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

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 sein 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.[4]

- Mit

  token[5] `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 —)[6] 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 Abschnitt A.6). Ein Beispiel ist in Abschnitt B zu finden.

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

[5] 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.

[6] *check&read*s 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

α `<-` *name* (1)

oder

`nonterminal name` (2)

α `name rule` (3)

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

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"
|| ...
|| 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 Abschnitt 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

α `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.

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 ein Konflikt 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.

Beispiel: 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 niemals verwendet (und eine damit verbundene Aktion niemals ausgeführt). In `(( a || {{ ... }} )) ??` wird die Aktion niemals ausgeführt.

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

A.4 Warnungen und Fehlermeldungen

Bei fast allen Meldungen wird angegeben, auf welchen Teil der Grammatik sie sich beziehen. 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 ein, z.B. das Leerwort `eps`.

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 Abschnitt 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 Abschnitt A.3) wird zwar ein Weg ausgewählt, aber womöglich ein anderer als der gewünschte. Bei *warnings* können nur Probleme der Art auftreten, daß die falsche ϵ-Ableitung gewählt wird und damit eventuell die falsche Aktion ausgeführt wird.

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

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

warning: unnecessary option Die Option wurde auf einen Ausdruck angewandt, der schon von sich aus ϵ ableiten kann. Die ϵ-Ableitung des Ausdrucks wird niemals verwendet, sondern ggf. die der Option.

warning: *-repetition of optional term Ein Ausdruck wurde *-wiederholt, der ϵ ableiten kann. Die ϵ-Ableitung des Ausdrucks wird niemals verwendet, sondern ggf. die der *-Wiederholung.

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

conflict: *Konflikt-Menge*[7] 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. Ein von Gray erzeugter Parser glaubt in diesem Fall, daß es das erste ist; er kann also „a" nicht parsen.

[7] 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 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 [ASU86] 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.

A.5.2 Linksfaktorisieren

In

`((` α β `||` α γ `||` δ `))`

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:

`((` α `((` β `||` γ `)) ||` δ `))`

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

A.6 Fehlerbehandlung im erzeugten Parser

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 überprüften Mengen (die Argumente von *test?*) vereinigt werden. Ein Beispiel ist in Abschnitt 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 [ASU86] und führe hier nur noch eine einfache Technik an, um bei bestimmten, 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
   (( (( ";"
      || {{ ." ; missing" pascal-error }}
      )) statement )) **
)) <- StatementSequence
```

B calc – Ein kleiner Interpreter

```
( a usage example: )
( you: ? 2*3-5/4= )
( calc: 5 )
( the grammar is a bit unconventional in
( its treatment of unary -. E.g., you have
( to write 3*(-5) instead of 3*-5 )

decimal
255 max-member ( the whole character set )
2variable input ( a string in 'addr count'
                ( representation )

10 stack expected

: sym ( -- c )
 input 2@ if
  c@
 else
  drop [char] =
 endif ;

: 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 cr
  true abort" syntax error"
 endif ;

: ?readnext ( f -- )
 ?syntax-error
 expected clear
 input 2@
 dup if
  1 chars - swap char+ swap
 endif
 input 2! ;

: init ( -- )
 bl word count input 2! ;

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

: x ( set1 -- set2 )
 ( read a char from the input
 ( and include it in the set )
 char 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 [char] 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 ? ( -- )
```

Literatur

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Cha91] Gordon Charlton. FOSM, a FOrth String Matcher. In *EuroForml '91 proceedings*, 1991.

[Cha92] Gordon Charlton. FOSM, a FOrth String Matcher, continued. In *EuroForth '92*, pages 113–122, Southampton, England, 1992. MicroProcessor Engineering.

[EM95] M. Anton Ertl and Martin Maierhofer. Translating Forth to efficient C. In *EuroForth '95 Conference Proceedings*, Schloß Dagstuhl, Germany, 1995.

[Ert93] M. Anton Ertl. A portable Forth engine. In *EuroFORTH '93 conference proceedings*, Mariánské Láznè (Marienbad), 1993.

[IH90] Tyler A. Ivanco and Geoffry Hunter. A user definable language interface for forth. *Journal of Forth Application and Research*, 6(1), 1990.

[Jak96] C.M. Jakeman. Portable back-tracking in ans forth. In *FORML '96 Proceedings*, 1996. ftp://ftp.taygeta.com/pub/Forth/Applications/fosm1v1.zip.

[Per86] Michael A. Perry. A 68000 Forth assembler. In Marlin Ouverson, editor, *Dr. Dobb's Toolbook of Forth*, chapter 23, pages 193–201. M&T Books, Redwood City, CA 94063, 1986.

[Rod90] Brad Rodriguez. A BNF parser in Forth. *SigForth Newsletter*, 2(2):13–15, December 1990.

[SR] David Muir Sharnoff and Steven Allen Robenalt. Catalog of compilers, interpreters, and other language tools. http://www.idiom.com/free-compilers.

[Zec84] Ronald Zech. *Die Programmiersprache FORTH.* Franzis, München, first edition, 1984. In German.