30.11.1995

Entwurf und Implementierung des funktionalen Parser-Generators CoP

Jürg Fässler

Technischer Bericht 95.27

Die grundlegenden Impulse für die in diesem Bericht geschilderten Arbeiten gehen vom Projekt AEMES[1] - **A**N **E**XTENSIBLE **M**AINTENANCE **E**NGINEERING **S**YSTEM - aus. Dieses Projekt läuft seit September 1992 unter der Leitung von Professor Richter und hat den Entwurf und die Implementierung einer erweiterbaren Wartungsumgebung für COBOL-74-Programme zum Ziel. In diesem Bericht werden wir zeigen, warum ein Parser-Generator für eine Wartungsumgebung wie AEMES eine Schlüsselrolle spielt und deshalb absolut unerlässlich ist. Wir werden auch begründen, warum wir nicht auf bestehende Technologien zurückgreifen konnten beziehungsweise wollten und deshalb, basierend auf funktionalen Konzepten, einen Parser-Generator in Eigenentwicklung realisiert haben. CoP - A **C**OMBINATOR BASED **P**ARSER GENERATOR - verwendet Backtracking-Verfahren und kann deshalb Parser für nahezu sämtliche kontextfreien Sprachen erzeugen. Dabei haben wir Wert darauf gelegt, dass der Spezifikation eines Parsers ein strenger Formalismus zugrundeliegt, der zusätzlich leicht verständlich ist.

Schlüsselwörter: Backtracking, Combinators, Continuations, funktionale Programmierung, Parser-Generator, Standard ML, Syntax-Analyse.

1. Das Projekt AEMES wird unterstützt von der Kommission zur Förderung der wissenschaftlichen Forschung (KWF) und den externen Projektpartnern Schweizerische Lebensversicherungs- und Rentenanstalt und Bull Schweiz AG

Inhaltsverzeichnis

1.0 Einleitung

1.1 Wartungsproblematik

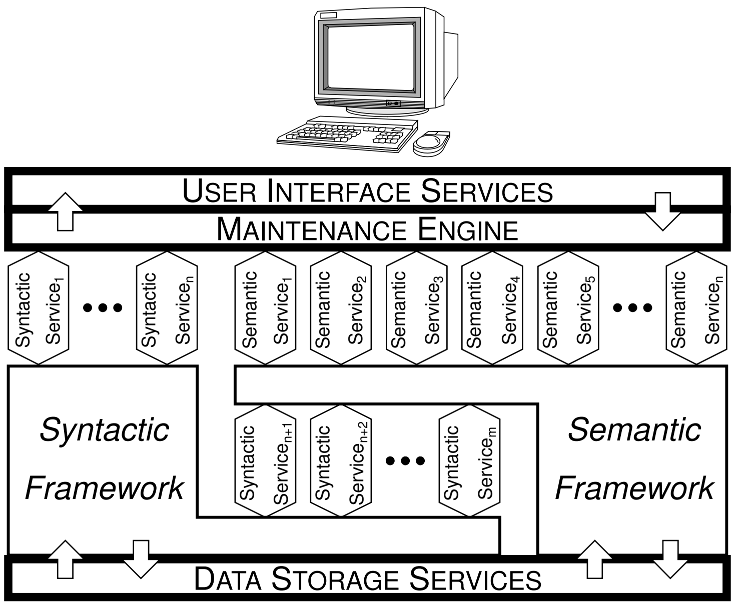
Programmwartung ist ein vielschichtiges Thema und umfasst sowohl Management-Aktivitäten als auch organisatorische, methodische und technische Aspekte (Lehner 1991). Wartung im Zusammenhang mit Software bedeutet immer auch Programmänderungen. Um solche Änderungen erfolgreich durchführen zu können, muss der Programmierer die betreffenden Programme weitgehend verstehen. Das ist aus vielerlei Gründen eine schwierige Aufgabe. Die zu wartenden Programme sind häufig zehn und mehr Jahre alt. Sie wurden während ihres langen "Lebens" immer wieder geändert, so dass ihre ursprüngliche Struktur nicht mehr klar erkennbar ist. Im Laufe der Jahre entstand ein schwer durchschaubares Konglomerat verschiedenster Funktionalitäten, Programmiertechniken und Programmierstile. In der Regel fehlt eine zuverlässige und leicht verständliche Dokumentation. Die Programmierer, welche für die ursprünglichen Versionen verantwortlich waren, arbeiten schon längst nicht mehr im betreffenden Betrieb und können nicht mehr gefragt werden. Ihr Know-how ist unwiederbringlich verloren gegangen oder steckt bis zu einem gewissen Grad in den zu wartenden Programmen. Diese Programme sind oft die einzige Informationsquelle, auf die für Wartungsaufgaben zurückgegriffen werden kann. Programme sind zwar formale Spezifikationen, welche die Problemlösung exakt beschreiben, eignen sich aber in erster Linie für die Interpretation durch den Computer und weniger durch den Menschen. Wie verschiedene Untersuchungen zeigen, verbringt ein Wartungsprogrammierer den grössten Teil seiner Zeit damit, um existierende Programme zu verstehen und nicht etwa um neue zu schreiben oder um Änderungen durchzuführen. Wenn es gelingt, den Aufwand für das Verstehen von Programmen zu reduzieren, ist die Wartungsproblematik zwar noch nicht gelöst, aber doch beträchtlich entschärft. Genau hier setzt AEMES an, um wichtige Hilfestellungen zu leisten.

1.2 Das Projekt AEMES

Der Schwerpunkt von AEMES ist ein technischer und liegt bei der Analyse von COBOL-74-Programmen, wie sie zu Hunderttausenden existieren und immense Wartungsprobleme verursachen. AEMES erhebt nicht den Anspruch, neue Wartungsmodelle zu entwickeln, sondern beabsichtigt, bereits bestehende Methoden und Techniken in die Praxis umzusetzten und konsequent anzuwenden. Es gibt keine vollautomatischen Verfahren zur Programmwartung, obwohl seit einiger Zeit Forschung in diesem Bereich betrieben wird. Mit Methoden der Künstlichen Intelligenz wird versucht, dem Programmierer einen virtuellen Assistenten zur Seite zu stellen, der ihm Routinearbeiten selbständig abnimmt. Solchen Projekten war bisher nur bescheidener Erfolg beschieden. Der Mensch bleibt auf unabsehbare Zeit der wichtigste Faktor in der Programmwartung. Das kommt auch daher, weil der Programmtext allein zum Verstehen eines Programms nicht ausreicht. Ein Programmierer muss zusätzlich über den Kontext Bescheid wissen, in dem ein Programm abläuft. Dazu gehören technische Kenntnisse der zugrundeliegenden Hardware und des Betriebssystems, aber auch Wissen über die modellierten Geschäftsprozesse. Es ist sehr schwierig, eventuell sogar unmöglich, solches Kontextwissen zu formalisieren, damit es für den Computer handhabbar wird. Das heisst, der Computer kann dem Programmierer den Prozess des Programmverstehens nicht abnehmen. Man kann den Computer aber dazu nutzen, um das Verstehen von Programmen erheblich zu erleichtern. AEMES bietet dem Wartungsprogrammierer eine mausgesteuerte grafische Benutzeroberfläche, die das interaktive Navigieren innerhalb von Programmen und das Lokalisieren von kritischen Programmstücken ermöglicht. Das hilft ihm, sich in den zu wartenden Programmen schneller zurechtzufinden und verkürzt so den Prozess des Programmverstehens erheblich. AEMES stellt den Programmierer gewissermassen ins Zentrum der Wartungsaktivität und gibt ihm Mittel in die Hand, die ihn bei seiner Arbeit unterstützen. Diese Symbiose zwischen technischen Hilfsmitteln einerseits und menschlicher Intelligenz und Intuition andererseits, bildet das Fundament, auf dem AEMES aufbaut.

Eines der wichtigsten Entwurfskriterien von AEMES ist die Erweiterbarkeit, wie der Projektname trefflich ausdrückt. Man kann nicht im voraus sämtliche Tools zur Verfügung stellen, die ein Wartungsprogrammierer je brauchen wird. Eine Wartungsumgebung ist deshalb offen zu konzipieren, damit sie nachträglich leicht neuen Bedürfnissen angepasst werden kann. Das bedeutet nicht, dass sich jedes x-beliebige Werkzeug in AEMES integrieren lässt. Vielmehr werden durch AEMES auf verschiedenen Ebenen Rahmenwerke (*Frameworks*) zur Verfügung gestellt, welche die Grundstruktur von Werkzeugen vorgeben. Betrachten wir die Architektur von AEMES, so erkennen wir drei Hauptkomponenten, nämlich die Benutzerschnittstelle, die Datenverwaltung und gewissermassen das Herzstück, die Wartungswerkzeuge, welche wir unter dem Begriff *Maintenance Engine* zusammenfassen (Abbildung 1).

ABBILDUNG 1. DIE ARCHITEKTUR VON AEMES



1.2.1 User Interface Services

Die Aufgabe der Benutzerschnittstelle besteht darin, dem Wartungsprogrammierer eine konsistente und einheitliche Sicht der verschiedenen Werkzeuge zu präsentieren. Konsistent bedeutet in diesem Zusammenhang: Alle Anzeigefenster mit verschiedenen Repräsentationen desselben Programms sind ständig up to date zu halten. Das ist nicht nur bei Programmänderungen wichtig, sondern auch dann, wenn sich beim Navigieren die Sicht auf bestimmte Programmausschnitte ändert. Um der Individualität der einzelnen Benutzer gerecht zu werden, bestehen Möglichkeiten, AEMES dem persönlichen Arbeitsstil anzupassen. Die besten Werkzeuge nützen nichts, wenn sie schwerfällig zu bedienen sind und dem Benutzer eine Arbeitsweise aufzwingen, die für ihn nicht intuitiv ist. Der Benutzer muss sich auf seine eigentliche Arbeit, die Analyse von Programmen, konzentrieren können und darf nicht durch die Bedienung der Wartungsumgebung ständig abgelenkt werden. In diesem Zusamenhang kommt der Benutzerschnittstelle grösste Bedeutung zu. Um den Anforderungen von AEMES gerecht zu werden, wurde WUI - THE WIREABLE USER INTERFACE - entwickelt (Kiser 1995). Das WUI ist ein Ansatz zur Integration von Werkzeugen unter Gewährleistung einer einheitlichen Benutzerschnittstelle. Dem Wartungsprogrammierer werden Möglichkeiten geboten, die Wartungsumgebung weitgehend frei zu konfigurieren. Das geschieht über einen Verdrahtungsmechanismus, der es erlaubt, Werkzeuge mit einfacher Funktionalität zu komplexeren zusammenzufassen.

1.2.2 Maintenance Engine

Eine Wartungsumgebung darf nicht einfach ein Haufen von verschiedenen Werkzeugen sein. Es geht nicht darum, möglichst viele, sondern einen ausgewogenen Satz von Werkzeugen bereitzustellen. AEMES unterscheidet zwischen Diensten (*Services*) und Werkzeugen (*Tools*). Dienste stellen Basisfunktionalität zur Verfügung, die von der Maintenance Engine zu Werkzeugen kombiniert werden. Die Maintenance Engine basiert auf zwei Rahmenwerken, welche syntaktische beziehungsweise semantische Dienste anbieten. Der Benutzer arbeitet generell mit Werkzeugen und nicht mit Diensten. Er kann aber über das WUI selbständig verschiedene Dienste zu Werkzeugen kombinieren, wenn dies Sinn macht. Falls die Wartungsumgebung um neue Dienste erweitert werden soll, ist das nicht die primäre Aufgabe des Benutzers, sondern von Spezialisten auf dem Gebiet der syntaktischen beziehungsweise semantischen Analyse von Programmen.

Syntaktisches Rahmenwerk: Syntaktische Dienste basieren auf den syntaktischen Strukturen einer Programmiersprache, ohne deren Bedeutung zu berücksichtigen. Dazu gehören *textuelle Suchfunktionen*, *Browser*, *Crossreferencer* usw. Eine Sonderstellung unter den syntaktischen Diensten nimmt der *Parser* ein. Der Parser liefert die interne Darstellung von Programmen, auf der fast alle anderen Dienste und Werkzeuge aufbauen. Der Parser ist gewissermassen eine Schlüsseltechnologie für Wartungsumgebungen. Wenn wir - wie im Fall AEMES - eine erweiterbare Wartungsumgebung realisieren, benötigen wir sogar einen Parser-Generator, und zwar aus verschiedenen Gründen:

1. Es gilt als ein gutes Entwurfsprinzip, Spezifikation und Implementierung zu trennen, denn Spezifikationen sind einfacher zu verstehen und abzuändern als deren Implementierung. Ein Parser-Generator kommt uns hier sehr entgegen. Er verlangt als Eingabe eine formale Spezifikation einer Programmiersprache (Grammatik) und erzeugt daraus die Implementierung, also den Parser selbst.
2. Eine Grammatik beschreibt nicht nur die syntaktische Struktur einer Programmiersprache, sondern natürlich gleichzeitig die Struktur des erzeugten konkreten Syntaxbaums eines geparsten Programms. Weil wir in AEMES den konkreten Syntaxbaum (Parse-Baum) als interne Darstellung verwenden, kommt der Grammatik noch eine besondere Bedeutung zu. Sie dient als wichtige formale Grundlage bei der Imple-

mentierung von sämtlichen Diensten und Werkzeugen und ist damit ein Garant für die Erweiterbarkeit der gesamten Wartungsumgebung.

3. Parser sind meistens fest in Compiler integriert und deshalb für andere Anwendungen nicht direkt zugänglich. Für Wartungsumgebungen muss also ein Parser nachgebaut werden. Das ist für Programmiersprachen mit einer wohldefinierten Grammatik in der Regel kein Problem. Die Parser früher Programmiersprachen, zu denen auch COBOL gehört, wurden nach Ad-hoc-Methoden entwickelt. Das heisst, es existiert keine einheitliche, zuverlässige und vollständige Grammatik. Ein Nachbau des Parsers kann deshalb nur experimentell - d.h. im *Trial-And-Error-Verfahren* - erfolgen. Ein Parser-Generator unterstützt dieses Vorgehen, ist es doch viel einfacher, die Grammatik zu ändern, als den Parser selbst. Durch den Einsatz eines Parser-Generators ist es uns gelungen, COBOL-74 weitgehend zu reformalisieren und eine saubere Grammatik dafür zu entwickeln.
4. AEMES ist für die Analyse von COBOL-74-Programmen vorgesehen, will aber eine Erweiterung hinsichtlich anderer Dialekte oder sogar Programmiersprachen a priori nicht ausschliessen. Deshalb werden die verschiedenen Rahmenwerke von AEMES so konzipiert, dass sie möglichst einfach neuen Gegebenheiten angepasst werden können. Für das syntaktische Rahmenwerk kann dies nur die Bereitstellung eines Parser-Generators bedeuten.

Man kann mit Fug und Recht behaupten: Ohne Parser keine Wartungsumgebung und ohne Parser-Generator keine erweiterbare Wartungsumgebung.

Semantisches Rahmenwerk: Die Dienste, die vom semantischen Rahmenwerk angeboten werden, beruhen nicht nur auf den syntaktischen Strukturen einer Programmiersprache, sondern beziehen auch die Bedeutung (Semantik) dieser Strukturen in die Analyse ein. Zu solchen Diensten gehören unter anderem *Daten-* und *Kontrollflussanalysen*, aber auch die Generierung von *Aufrufgraphen*. Bei AEMES gehen wir davon aus, dass die Betriebsumgebung und die Wartungsumgebung voneinander getrennt sind. Im Gegensatz zur Wartungsumgebung, welche auf einer Arbeitsstation läuft, handelt es sich im Betrieb um eine reine Host-Umgebung. Das hat die wichtige Konsequenz, dass AEMES nur statische Analysen durchführen kann. Testläufe sind also nicht möglich.

1.2.3 Data Storage Services

Es ist keine Frage, dass eine Wartungsumgebung wie AEMES eine grosse Menge von Daten erzeugt, die konsistent verwaltet werden muss. Dazu bieten sich Datenbanken als Lösung an. Datenbanken für CASE-Umgebungen, zu denen man auch AEMES zählen darf, nennt man üblicherweise Repositories. In einem Repository werden sämtliche Daten, die während der Entwicklung eines Projekts anfallen, gespeichert. Der Begriff Repository beschreibt leider weniger eine Technologie, sondern ist eher ein Marketing-Schlagwort. Deshalb haben wir bisher bewusst auf den Einsatz eines Repository's verzichtet, bis wir selbst, mit Hilfe unseres Prototyps, mehr über die Art und Struktur der anfallenden Daten gelernt haben. Zur Zeit verwenden wir direkt das UNIX-Filesystem zur Datenhaltung, sind uns aber bewusst, dass dies den Anforderungen in der Praxis nicht genügen kann. Wir verfolgen das Grundprinzip, möglichst wenig verschiedene Repräsentationen von Programmen effektiv zu speichern. Wenn immer möglich wird versucht, Informationen *on the fly* aus ein paar Grundrepräsentationen zu extrahieren. Als wichtigste Daten haben sich bisher die zu wartenden Programme selbst und die daraus abgeleiteten Parse-Bäume erwiesen, welche als Grundlage der meisten Werkzeuge dienen. Natürlich müssen auch Meta-Informationen verwaltet werden. Dazu gehören unter anderem Parser-Spezifikationen und semantische Informationen über die Programmiersprache, welche den zu analysierenden Programmen zugrundeliegt.

Wir haben in diesem Abschnitt die drei Hauptkomponenten von AEMES vorgestellt und versucht, einen Einblick in das Projekt und die damit verbundenen Probleme zu vermitteln. Im weiteren Verlauf werden wir uns auf das syntaktische Rahmenwerk beschränken und uns mit der Thematik der syntaktischen Analyse von Programmen im allgemeinen und derjenigen von COBOL-Programmen im speziellen befassen.

2.0 Grundlagen der syntaktischen Programm-Analyse

Wenn wir interaktiv mit Computern arbeiten, dann arbeiten wir gleichzeitig - meist ohne uns dessen bewusst zu sein - ständig mit den verschiedensten Parsern. Am ehesten realisieren wir noch beim Aufruf eines Compilers, dass vor der Code-Erzeugung eine syntaktische Analyse des Quellprogramms stattfindet. Aber auch jeder Befehl, den wir über die Tastatur eingeben und an das Betriebssystem absetzen, wird geparst. Dass eine interne Instanz unsere Eingaben überprüft, merken wir in der Regel erst dann, wenn diese mit einer Fehlermeldung zurückgewiesen werden. Bei diesen internen Instanzen handelt es sich um Parser. Parser müssen mindestens in der Lage sein, korrekte von inkorrekten Eingaben zu unterscheiden. Meistens generieren sie aber zusätzlich eine interne Darstellung, die für eine weitere verarbeitung durch den Computer besser geeignet ist als die ursprüngliche Eingabe. Parser sind also in der Informatik eine häufige Erscheinung und werden so selbstverständlich benutzt, dass man sich kaum je mit ihrer Komplexität auseinandersetzt. Das hat einen guten Grund. Für viele der heute verwendeten Programmiersprachen stellt die syntaktische Analyse tatsächlich kein Problem mehr dar. Für eine grosse Klasse formaler Sprachen existieren eine Vielzahl von Methoden und Generatoren, die es erlauben, aus einer formalen Sprachbeschreibung vollautomatisch einen Parser zu generieren. Leider gilt diese Aussage nicht für alle Programmiersprachen. Dauerbrenner wie FORTRAN und COBOL entziehen sich noch heute weitgehend einer einfachen formalen Beschreibung und somit der automatischen Generierung eines Parsers. Die Probleme rühren vor allem daher, dass sich solche Sprachen nur bedingt denjenigen Sprachklassen zuordnen lassen, für die einfache und effiziente Parsing-Methoden bekannt sind.

Die syntaktische Analyse von Programmen hat zwei grundsätzliche Aspekte. Erstens geht es darum, die Menge von Eingabesequenzen, die zu einer formalen Sprache gehören, exakt zu beschreiben. Zweitens muss ein Algorithmus gefunden werden, der effizient entscheidet, ob ein beliebiges Wort zur definierten Sprache gehört oder nicht und darüber hinaus mindestens in der Lage ist, eine interne Darstellung der korrekten Eingabe zu erzeugen. Der erste Aspekt ist ein theoretischer und sehr genau erforscht. Man weiss heute, wie eine formale Sprache zu definieren ist, damit sie einfach und effizient geparst werden kann. Wir geben nachfolgend einen kurzen Überblick. Der zweite Aspekt ist eher praktischer Natur. Im Prinzip gibt es kein Universalrezept dafür, wie ein Parser für eine beliebige formale Sprache konkret zu implementieren ist. Für eine grosse Klasse von formalen Sprachen existieren bereits ausgereifte Parsing-Methoden oder sogar Parser-Generatoren. Wenn man keine speziellen Anforderungen hat, genügen diese oft. Einer der bekanntesten Generatoren ist LEX/YACC, der auf sämtlichen UNIX-Systemen zur Verfügung steht. Für einfachere formale Sprachen ist es oft angebracht, den Parser von Hand zu schreiben. Welche Strategie man schliesslich wählt, hängt sehr von der vorgegebenen Problemstellung und der zugrundeliegenden formalen Sprache ab. An den nachfolgenden Beispielen wollen wir zeigen, dass sich an einen Parser je nach Verwendungszweck verschiedene Anforderungen stellen, welche einen grossen Einfluss auf die zu wählende Parsing-Technik haben:

Compiler: Der Parser wird in diesem Fall für eine ganz bestimmte Anwendung mit einem im voraus klar definierten Ziel geschrieben. Der Benutzer braucht über interne Darstellungen nichts zu wissen. Er ist nur am Endresultat, dem ausführbaren Programm interessiert. Der Benutzer tendiert dazu, den Compiler mit seinen Phasen (Scanning, Parsing, ..., Code-Generierung) als Black-Box, also gewissermassen als Funktion von

Quell-Code nach ausführbarem Code zu betrachten. Über den inneren Aufbau, also über die technischen Überlegungen, Hilfsmittel und Tricks, die eingesetzt wurden, um einen schnellen Compiler zu konstruieren, der effizienten Code erzeugt, braucht der Anwender in der Regel nichts zu wissen. Das gibt dem Compiler-Entwickler einen grossen Spielraum bei der Optimierung des Parsers, denn er kann die Grammatik beliebig transformieren, sofern er die Syntax der zugrundeliegenden Programmiersprache nicht ändert.

Compiler benötigen eine vollständige syntaktische Analyse. Die zugehörigen Parser erzeugen deshalb eine grosse Menge von internen Informationen, die über einen relativ kurzen Zeitabschnitt gebraucht werden. Direkt nach und zum Teil schon während der Code-Generierung werden nicht mehr gebrauchte Informationen elimiert. Ganz generell ist die Geschwindigkeit des Parsers viel wichtiger als der benötigte Speicherplatz, welcher ja nur kurzzeitig verwendet wird.

Wartungsumgebung: Von einem Parser wird hier grössere Flexibilität vorausgesetzt, denn er muss eine einheitliche interne Darstellung für die verschiedensten Dienste und Werkzeuge liefern, auch für solche, die vielleicht im Moment noch gar nicht bekannt sind. Für die interne Darstellung muss deshalb eine formale Beschreibung vorhanden sein, die so allgemein ist, dass sie die Weiterentwicklung der Wartungsumgebung nicht behindert oder sogar verhindert. Es ist deshalb naheliegend, die allgemeinste dieser Darstellungen zu wählen, und das ist der Parse-Baum. Dies hat zusätzlich den Vorteil, dass die Grammatik nicht nur den Parser selbst, sondern gleichzeitig die interne Darstellung formal beschreibt. Diese Voraussetzungen lassen aber wenig Spielraum für Optimierungen des Parsers, denn eine Transformation der Grammatik - auch wenn sie die Syntax der Programmiersprache nicht ändert - hat in der Regel eine Änderung der internen Darstellung zur Folge.

Eine vollständige syntaktische Analyse ist je nach Werkzeug nicht in jedem Fall notwendig. Werkzeuge benötigen die aus den Programmen extrahierten Informationen während eines relativ langen Zeitraums. Interne Repräsentationen werden in der Regel für die Verwendung in verschiedenen Werkzeugen aufbewahrt. Die Minimierung des benötigten Speicherplatzes ist daher wichtiger als die Effizienz des Parsers.

Wir beabsichtigen nicht, an dieser Stelle eine vollständige Einführung in die syntaktische Analyse von formalen Sprachen zu geben. Wir denken aber, dass es nötig ist, einige Begriffe klarzustellen und diejenigen Aspekte herauszuheben, die den Entwurf und die Implementierung des Parser-Generators CoP massgeblich beeinflusst haben.

2.1 Grammatiken und Formale Sprachen

Ein Parser hat grundsätzlich die Aufgabe, einen Text auf syntaktische Korrektheit zu überprüfen. Deshalb benötigt man bei deren Implementierung eine formale Beschreibung (Grammatik) aller syntaktisch korrekten Eingabesequenzen (Wörter). Zur Programmierung von Computern verwendet man aus Komplexitätsgründen nicht natürliche Sprachen wie Deutsch oder Englisch, sondern formale Sprachen, wie z.B. C, PASCAL oder COBOL. Wir erläutern an dieser Stelle einige Begriffe, die im Zusammenhang mit formalen Sprachen von Bedeutung sind und die wir deshalb im weiteren Verlauf dieses Berichts häufig verwenden werden:

Alphabet: Unter einem Alphabet versteht man eine endliche Menge von unterscheidbaren Symbolen. Die ASCII-Zeichen bilden zum Beispiel eine solche Menge.

Wort: Ein Wort über einem Alphabet ist eine endliche Sequenz von Symbolen aus diesem Alphabet. Eine beliebige Kette von ASCII-Zeichen, auch wenn sie Leerzeichen enthält, bildet ein solches Wort.

Sprache: Eine (formale) Sprache bezeichnet eine Menge von Wörtern über einem fixen Alphabet. Zum Beispiel bezeichnen alle PASCAL-Programme die formale Sprache PASCAL.

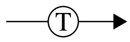
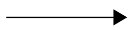
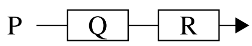
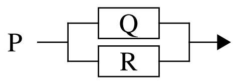
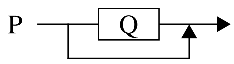
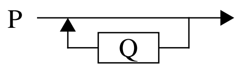
Grammatik: Eine Grammatik ist ein 4-Tupel (N,T,P,S) und dient zur Definition der Syntax von formalen Sprachen. Dabei bedeuten:
N : Menge von Nonterminalsymbolen
T : Menge von Terminalsymbolen (Alphabet)
P : Menge von Produktionen (Regeln)
S : Startsymbol $\in$ N

Eine genaue und umfassende Einführung in die Theorie der formalen Sprachen findet man z.B. in (Hopcroft and Ullmann 1979).

2.2 Notationen zur Definition von Grammatiken

Zur Definition von Grammatiken existieren verschiedene Notationen. Wir haben die drei häufigsten in einer Tabelle zusammengefasst (Tabelle 1).

Tabelle 1. Grammatik-Notationen

Funktion	BNF	EBNF	Syntax-Diagramme
Terminalsymbol	"T"	"T"	
Nonterminalsymbol	<NT>	NT	
Leeres Wort	ε		
Sequenz	<P> ::= Q R	P = Q R .	
Alternative	<P> ::= Q | R	P = Q | R .	
Gruppierung		P = (Q) .	
Option		P = [Q] .	
Repetition		P = { Q } .	

Alle drei Notationen sind gleich mächtig. Man kann damit die Syntax von beliebigen kontextfreien Sprachen formal beschreiben. Die Backus-Naur-Form (BNF) wurde zum ersten Mal im Zusammenhang mit ALGOL-60 verwendet (Backus, Bauer et al. 1960). Die nachträgliche Erweiterung zur EBNF (Extended BNF) erhöht zwar nicht die Mächtigkeit aber die Ausdruckskraft von Grammatikspezifikationen. Sowohl die BNF als auch die EBNF sind formale Sprachen und mächtig genug, um ihre eigene Syntax zu beschreiben:

2.2.1 BNF-Grammatik in BNF-Notation

```
<BNF-Grammar>  :== <BNF-Grammar> <Production> | ε
<Production>   :== <Nonterminal> ":==" <Expression>
<Expression>   :== <Term> | <Expression> "|" <Term>    ← Alternative
<Term>         :== <Factor> | <Term> <Factor>          ← Sequenz
<Factor>       :== <Nonterminal> | <Terminal> | ε
<Nonterminal>  :== IDENTIFIER
<Terminal>     :== STRING
```

2.2.2 EBNF-Grammatik in EBNF-Notation

```
EBNF-Grammar   = { Production } .
Production     = Nonterminal "=" Expression .
Expression     = Term { "|" Term } .                   ← Alternative
Term           = Factor { Factor } .                   ← Sequenz
Factor         = Nonterminal
               | Terminal
               | "(" Expression ")"                    ← Gruppierung
               | "[" Expression "]"                    ← Option
               | "{" Expression "}" .                  ← Repetition
Nonterminal    = IDENTIFIER .
Terminal       = STRING .
```

Die Symbole, welche von der (E)BNF-Notation selbst verwendet werden, nennt man Metasymbole. Terminalsymbole schreibt man in Anführungszeichen, um sie von Metasymbolen zu unterscheiden.

EBNF-Spezifikationen sind kompakter und besitzen eine grössere Ausdruckskraft als BNF-Spezifikationen. Formale Sprachdefinitionen in BNF sind um ein Vielfaches umfangreicher als solche in EBNF. BNF-Spezifikationen sind zudem in der Regel hochrekursiv und deshalb oft schwer verständlich. Wenn es aber darum geht, Grammatiken auf bestimmte Eigenschaften zu analysieren oder in Parser umzusetzen, dann ist die einfachere BNF-Darstellungen eindeutig im Vorteil. Es besteht daher ein *Trade-Off* zwischen einfacher Lesbarkeit, welche dem Entwickler einer Grammatik entgegenkommt und einfacher Analysierbarkeit, welche für die Implementierung und Optimierung eines Parsers von Vorteil ist. Der Parser-Generator CoP ist in der Lage, mit beiden Notationen zu arbeiten. Mehr noch, man kann eine EBNF-Grammatik in eine BNF-Grammatik transformieren, ohne dass dies einen Einfluss auf die erzeugte internen Darstellung hat. Das bedeutet, der Parser baut auf der einfacheren BNF-Repräsentation auf, ohne dass Wartungsprogrammierer oder Entwickler von Wartungswerkzeugen, die sich lieber an die ausdrucksstärkere EBNF-Notation halten, etwas davon merken. Wir werden auf dieses Thema später zurückkommen.

Es existiert ein einfacher Algorithmus, um von einer EBNF- zu einer BNF-Darstellung zu gelangen. Dazu müssen Gruppierungen, Optionen und Repetitionen einer EBNF-Spezifikation durch die Möglichkeiten der BNF ausgedrückt werden. Die umgekehrte Richtung ist allerdings nicht mehr so einfach.

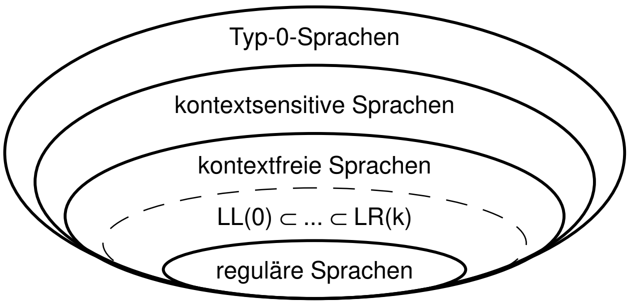
```
EBNF:                  BNF:

P = α ( β ) γ .⇒       <P>    ::= α <Q> γ
                       <Q>    ::= β

P = α [ β ] γ .⇒       <P>    ::= α <Q> γ
                       <Q>    ::= β | ε

P = α { β } γ .⇒       <P>    ::= α <Q> γ
                       <Q>    ::= β <Q> | ε
```

α, β und γ sind beliebige Sequenzen, die sowohl Terminal als auch Nonterminalsymbole enthalten können oder entsprechen dem leeren Wort ε. Falls α, β, γ oder die neu erzeugte Produktion <Q> ebenfalls Gruppierungen, Optionen oder Repetitionen enthalten, dann müssen sie nach der gleichen Methode umgeformt werden.

2.3 Sprachklassen

Mitte der Fünfzigerjahre publizierte der Linguist N. Chomsky eine Klassierung (formaler) Sprachen, die heute noch aktuell ist. Er unterscheidet vier Klassen, zwischen denen echte Teilmengenbeziehungen bestehen (Abbildung 2).

ABBILDUNG 2. SPRACHKLASSEN NACH CHOMSKY



Diese Klassierung scheint zunächst ziemlich willkürlich. Sie bekommt aber einen Sinn, wenn man die virtuellen Maschinen betrachtet, die zur Erzeugung beziehungsweise Analyse einer Sprache aus der jeweiligen Klasse vorausgesetzt werden. So benötigt man zum Beispiel für reguläre Sprachen lediglich einen (deterministischen) endlichen Automaten und für Typ-0-Sprachen die volle Mächtigkeit der Turing-Maschine. Je umfassender die Klasse, desto komplexer und aufwendiger die Analyseverfahren. Auch wenn diese Klassierung beim theoretischen Verständnis von formalen Sprachen grosse Dienste leistet, hilft sie bei der konkreten Realisierung von Parsern wenig.

Seit den Sechzigerjahren werden formale Sprachen theoretisch weitgehend verstanden. Die Entwicklung einer grossen Menge verschiedener Parsing-Methoden und Parser-Generatoren war die Folge. Es hat sich herausgestellt, dass aus Komplexitätsgründen für die Realisierung von Parsern nur die regulären und die kontextfreien Sprachen von Bedeutung sind. Es ist sogar so, dass man sich in der Praxis üblicherweise auf Subklassen der kontextfreien Sprachen beschränkt. Zu den wichtigsten Subklassen gehören z.B. die LL(k)-Sprachen, welche sich durch Top-Down-Parser analysieren lassen und die mächtigeren LR(k)-Sprachen, zu deren Analyse Bottom-Up-Verfahren benötigt werden. LR(k)-Sprachen sind deterministisch. Sie definieren also genau diejenigen kontextfreien Sprachen, die sich eindeutig parsen lassen und für die noch effiziente Parsing-Methoden existieren. Dabei bezeichnet k die Anzahl Symbole, die in der Eingabesequez vorausgeschaut werden muss (look-ahead), um deterministisch - das heisst ohne Backtracking - eine eindeutige syntaktische Analyse zu erhalten. In der Regel beschränkt man sich auf LR(1), oder Subklassen davon , wie z.B. LALR oder SLR, zumal man zeigen kann, dass sich jede LR(k)-Grammatik in eine LR(1)-Grammatik umformen lässt. Ganz generell kann man zusammenfassen: Wenn sich eine Sprache durch eine LR(k)-Grammatik oder eine Subklasse davon beschreiben lässt, dann kann man auf bewährte Lösungen zurückgreifen. Wenn die Grammatik einer formalen Sprache nicht zu diesen “einfachen” Klassen gehört, dann hat man ein ernsthaftes Problem.

2.4 Syntax-Analyse von COBOL

Beschränkungen auf Subklassen der kontextfreien Sprachen haben auf die Ausdruckskraft von Programmiersprachen kaum einen negativen Einfluss, erleichtern aber deren syntaktische Analyse erheblich. Leider kamen diese Erkenntnisse für Sprachen wie FORTRAN und COBOL zu spät. Das ist ein Umstand, der schon deshalb zu bedauern ist, weil gerade diese beiden Sprachen noch heute weitverbreitet sind. Es ist leider auch zu beobachten, dass für Programmiersprachen, die schon syntaktisch schlecht definiert sind, zusätzlich die Beschreibung der Semantik zu wünschen übrig lässt. Das Resultat dieser Mängel ist eine schlechte Wartbarkeit der in diesen Sprachen geschriebenen Programme. Das wiederum hat zur Folge, dass die Nachfrage nach Wartungswerkzeugen für diese Sprachen sehr gross ist. Die schlechten formalen Grundlagen erschweren die Entwicklung von mächtigen und zuverlässigen Werkzeugen zur Unterstützung der Wartung.

Im Projekt AEMES sehen wir uns mit der Programmiersprache COBOL-74 konfrontiert, deren Grammatik zwar kontextfrei notiert werden kann, die aber nicht zu denjenigen Subklassen der kontextfreien Sprachen gehört, für die einfache Parsing-Methoden oder sogar Parser-Generatoren existieren.

2.4.1 Probleme

Wegen der Unvollständigkeit der Syntaxdefinition von COBOL - viele Teile sind auch in den Standardisierungs-Dokumenten nur informell beschrieben - kann man keine exakten Aussagen über die zugrundeliegende Sprachklasse machen. Das ist wohl mit ein Grund, weshalb von COBOL eine Vielzahl von herstellerspezifischen Dialekten existieren, die in erheblichen Teilen voneinander abweichen können. Bei COBOL handelt es sich weniger um eine Programmiersprache, als vielmehr um eine Sprachfamilie. Im Grunde genommen ist jeder COBOL-Compiler selbst die einzige zuverlässige syntaktische (und auch semantische) Sprachbeschreibung eines spezifischen COBOL-Dialekts. Eine COBOL-Grammatik ist darüberhinaus riesig und unübersichtlich. Sie umfasst zwischen dreissig bis vierzig A4-Seiten, im Gegensatz etwa zur Programmiersprache OBERON, deren Grammatik auf einer einzigen A4-Seite Platz findet. Zusätzlich erschwerend ist der Umstand, dass weder COBOL-Grammatiken noch COBOL-Parser frei verfügbar sind.

Zu diesen eher allgemeinen Problemen kommen noch eine ganze Anzahl technischer hinzu. COBOL ist nicht formatfrei. Das heisst, die Spaltenposition, an der ein bestimmtes Symbol steht, ist mit massgebend für die richtige Interpretation dieses Symbols. Gewisse Symbole können nur mit Hilfe von Kontextinformationen eindeutig identifiziert werden. Zum Beispiel kann die Zeichenfolge "`999`" je nach Umfeld ein PIC-String, ein Integer, ein Section- oder ein Paragraph-Name sein. Diese Mehrdeutigkeiten erschweren den Einsatz von konventionellen Scanning-Verfahren. Leerzeichen sind an gewissen Stellen von grösster Bedeutung und an anderen völlig irrelevant. Man beachte, dass die Zeichenfolge "`9-A`" ein Identifier "`9 - A`" aber ein arithmetischer Ausdruck ist, der aus einem Integer, einem Schlüsselwort und einem Identifier besteht. Ein weiteres Problem entsteht daraus, dass syntaktische Einheiten (z.B. IF-Anweisung) in der Regel nicht durch ein Schlüsselwort (z.B. END) abgeschlossen sind. Sie enden dann einfach dort, wo die nächste syntaktische Einheit beginnt. Das heisst, der Parser muss oft über viele Eingabesymbole vorausschauen, um eine korrekte Fortsetzung zu finden.

2.4.2 Lösungsansatz

Die speziellen Eigenschaften von COBOL haben uns zur Eigenentwicklung eines entsprechenden Parsing-Verfahrens veranlasst. Anstatt einen COBOL-Parser in mühsamer Fleissarbeit selbst zu schreiben, haben wir einen Parser-Generator entwickelt, der auch mit COBOL fertig wird und gleichzeitig einen strengen aber einfachen Formalismus bei der Beschreibung der Grammatik erlaubt. Dabei wurde von Anfang an eine saubere und konsistente Lösung angestrebt und nicht versucht, gängige Verfahren mit Hilfe von "Tricks" so zu erweitern, dass damit eine grössere Sprachklasse behandelt werden kann. Die Grundüberlegung besteht darin, eine Methode zu verwenden, von der man im voraus weiss, dass sie mit beliebigen kontextfreien Sprachen zurechtkommt. Es ist einfacher, ein umfassendes Konzept einzuschränken, als ein zu eingeschränktes Konzept im nachhinein zu erweitern. Der Lösungsansatz heisst Backtracking.

Warum Backtracking? Die COBOL-Grammatik gibt es nicht. Vielmehr existieren die verschiedensten, zum grossen Teil unvollständigen Beschreibungen, auf die man bei der Implementierung eines COBOL-Parsers zurückgreifen muss. Eine COBOL-Grammatik kann deshalb nur experimentell entwickelt werden. Das ist ein sehr komplexes Unterfangen. Wir benötigen deshalb einen Parser-Generator, der auf Methoden beruht, die bei der Entwicklung der Grammatik möglichst wenig einschränken. Backtracking-Verfahren garantieren hier die grösste Freiheit, denn sie geben keine speziellen Randbedingungen an Grammatiken vor. Was der Entwickler als Grammatik notiert, kann auch in einen Parser für die entsprechende Sprache transformiert werden[1]. Das ist nicht der Fall, wenn man sich auf Subklassen der kontextfreien Sprachen beschränkt. Dort sieht man sich ständig mit dem Problem konfrontiert, dass die Grammatik auf bestimmte Eigenschaften überprüft und gegebenenfalls umgeformt werden muss. Umformungen, wenn sie überhaupt möglich sind, haben oft den Nachteil, dass die Grammatik unübersichtlich und damit schwer verständlich wird. Weil bei AEMES die Grammatik nicht nur den Parser, sondern auch die interne Darstellung beschreibt, wollen wir natürlich genau diesen Umstand um jeden Preis vermeiden. Backtracking-Verfahren erlauben dem Entwickler, die Grammatik in der am einfachsten lesbaren Form aufzuschreiben. CoP, der Parser-Generator von AEMES, verbietet Grammatik-Transformationen nicht, will sie aber auf

1. Für das von uns gewählte Verfahren stimmt das nicht ganz. Linksrekursive Produktionen sind nämlich verboten. Das ist aber eine Randbedingung, die praktisch nicht einschränkt.

keinen Fall erzwingen. Sorgfältiges Umschreiben einer gegebenen Grammatik kann nämlich, ohne die Lesbarkeit zu beeinträchtigen, zu effizienteren Parsern führen.

Beim Backtracking-Verfahren zur syntaktischen Analyse, das wir verwenden, übernimmt der Computer die Sucharbeit automatisch und findet theoretisch immer eine Lösung, falls mindestens eine solche existiert. Man kann das Verfahren sogar so anpassen, dass bei mehrdeutigen Grammatiken sämtliche Lösungen gefunden werden. Die Suchräume können allerdings eine solche Grössen annehmen, dass innerhalb eines vernünftigen Zeitraums keine Lösung gefunden wird. Backtracking-Verfahren ermöglichen elegante Lösungen. Sie haben aber, in vielen Fällen zu Recht, den schlechten Ruf der Ineffizienz. Durch geeignete Heuristiken lässt sich diese Problematik aber stark reduzieren. Die Kunst bei der Verwendung von Backtracking-Verfahren besteht darin, den möglichen Lösungsraum geschickt einzuschränken. Falls man dies nämlich ungeschickt tut, ist es möglich, dass die gewünschte Lösung nicht mehr gefunden werden kann. Im übrigen sind Backtracking-Parser “selbstverbessernd”. Das heisst, die Effizienz des Parsers hängt direkt von der Güte der zugrundeliegenden Grammatik ab. Wenn eine eindeutige Grammatik vorliegt, ist der erzeugte Parser automatisch schneller als bei einer mehrdeutigen Grammatik.

Funktionale Programmiersprachen sind für die Realisierung unserer Konzepte hervorragend geeignet. In den nachfolgenden Abschnitten werden wir zunächst die theoretischen Grundlagen erläutern und danach die konkrete Realisierung von CoP diskutieren. Dabei werden wir auch auf das Thema der Heuristiken eingehen und zeigen, dass es möglich ist, mit den hier vorgestellten Konzepten einen Parser-Generator zu implementieren, der effizient und mächtig genug ist, um in einer Wartungsumgebungen wie AEMES eingesetzt zu werden.

3.0 Funktionale Programmierung und Syntax-Analyse

Wie wir sehen werden, sind funktionale Parser inhärent backtracking-fähig und lassen sich relativ leicht erweitern, so dass sie die gesamte Klasse der kontextfreien Sprachen analysieren können. Darüberhinaus bieten funktionale Sprachen Möglichkeiten, die es erlauben, die (E)BNF-Spezifikation einer formalen Sprache praktisch eins zu eins in den entsprechenden funktionalen Parser zu übersetzen.

3.1 Grundlagen funktionaler Programmiersprachen

Funktionale Programmiersprachen basieren auf dem Lambda-Kalkül, welches Syntax und Semantik von Lambda-Ausrücken definiert. Syntaktisch gesehen ist ein Lambda-Ausdruck:

1. eine Variable: `x, y, z, ...`
2. eine Funktionsapplikation: `F G`

   Wobei `F` ist eine Lambda-Abstraktion und `G` ein beliebiger Lambda-Ausdruck ist.
3. eine Lambda-Abstraktion: `λ x . F`

   `F` ist ein Lambda Ausdruck. Wenn die Variable `x` in `F` frei erscheint, dann wird `x` durch die Lambda-Abstraktion gebunden.

Die Semantik ist durch Transformationsregeln gegeben, die es ermöglichen, mit Lambda-Ausdrücken zu rechnen. Wir verzichten in diesem Bericht auf nähere Einzelheiten und verweisen auf (Gordon 1988), der eine gute und leicht verständliche Einführung in das Lambda-Kalkül bietet.

3.1.1 Funktionale Notation

CoP ist in der funktionalen Programmiersprache Standard ML implementiert (Milner, Tofte et al. 1990). Um die grundlegenden Konzepte funktionaler Parser zu zeigen, ist es einfacher, wenn wir uns auf eine konzisere Notation abstützen. Zu diesem Zweck und um der Lambda-Notation eine grössere Ausdruckskraft zu verleihen, erweitern wir diese zu einer kleinen funktionalen Programmiersprache. Die Syntax in EBNF lautet wie folgt:

```
λ-Program      = { λ-Definition } .

λ-Definition   = "def" [ "rec" ] Constant "=" λ-Expression .

λ-Expression   = Constant
               | λ-Expression λ-Expression
               | "λ" Variable "." λ-Expression
               | "if" λ-Expression "then" λ-Expression
                 "else" λ-Expression "end"
               | "let" λ-Definition { λ-Definition }
                 "in" λ-Expression "end" .
```

Die Semantik dieser kleinen Programmiersprache geben wir informell an Hand von Beispielen. Es ist übrigens nicht schwierig, sämtliche nachfolgenden Definitionen und Beispiele in eine realen funktionalen Programmiersprache, wie zum Beispiel Standard ML, umzusetzen.

3.1.2 Beispiel

Zur Erhöhung der Verständlichkeit notieren wir bei der Definition von Konstanten deren Datentyp. Zum Beispiel können wir eine Konstante `plus` definieren, die wir danach als Funktion verwenden können:

```
type plus : int → int → int
def  plus = λ x . λ y . x + y
```

Man kann mit Lambda-Ausdrücken rechnen:

```
plus 7 4
= (λ x . λ y . x + y) 7 4
= (λ y . x + y) [ 7 / x ] 4          ← Substitution: 7 für x
= (λ y . 7 + y) 4
= (7 + y) [ 4 / y ]                  ← Substitution: 4 für y
= 7 + 4
= 11
```

Ein wichtiger Aspekt, welcher die Eleganz von funktionalen Sprachen massgeblich bestimmt, ist die Tatsache, dass Funktionen Erstklassobjekte sind. Das heisst, sie können sowohl als Argumente wie auch als Funktionswerte anderer Funktionen auftreten, und zwar ohne Einschränkungen. Funktionen, welche andere Funktionen als Argumente oder Rückgabewerte akzeptieren, nennt man Funktionen höherer Ordnung. Von solchen Möglichkeit machen funktionale Parser intensiv Gebrauch. Man betrachte noch einmal das obige Beispiel. `plus 7` liefert als Resultat eine Funktion, nämlich `(λ y . 7 + y)`. `plus` ist also eine Funktion höherer Ordnung und liefert als Resultat eine Funktion vom Datentyp `int → int`. Diese hat die Semantik, dass sie jeden Integer-Wert, auf den sie angewendet wird, um `7` inkrementiert.

3.2 Funktionale Parsing-Techniken

Die Grundidee des funktionalen Ansatzes besteht darin, die Metasymbole der (E)BNF als Funktionen höherer Ordnung zu betrachten, welche es erlauben, einfache Parser-Funktionen zu immer komplexeren zu kombinieren, bis schliesslich der vollständige Parser für eine formale Sprache aufgebaut ist. Es gibt verschiedene Möglichkeiten, wie man dies tun kann. Wir zeigen nachfolgend zwei davon. Der direkte Ansatz, den wir zuerst diskutieren werden, ist naheliegend und wird in ähnlicher Form an verschiedenen Stellen beschrieben (Burge 1975; Reade 1989; Paulson 1991). Danach werden wir begründen, warum wir für CoP ein eigenes Verfahren entwickelt haben. Dieses Verfahren beruht auf Continuations.

3.2.1 Direkter Ansatz

3.2.1.1 Direkter Ansatz mit limitiertem Backtracking

Weil wir hier nur das Prinzip funktionaler Parser zeigen wollen, gehen wir von der vereinfachenden Annahme aus, dass die Eingabesequenz als Liste von Zeichen vorliegt, also zum Beispiel `["A"," ","B"]` für das Wort `"A B"`. Der Parser, den wir zuerst konstruieren werden, soll als Resultat ein Paar (kartesisches Produkt) der Form (Verdikt, Restsequenz) zurückgeben. Die erste Komponente gibt an, ob die syntaktische Analyse erfolgreich war oder nicht. Die zweite Komponente enthält den noch ungeparsten Rest der Eingabesequenz. Einen Parser mit diesen Eigenschaften nennt man üblicherweise *Recognizer*, denn er erkennt zwar korrekte beziehungsweise inkorrekte Eingabesequenzen, erzeugt aber keine interne Darstellung, wie z.B. einen Parse-Baum. Wir definieren den Datentyp $\texttt{Parser}_d$ folgendermassen:

```
type Parserd = char list → bool * char list
```

Mit dem Index **d** geben wir an, dass es sich um den **d**irekten Ansatz zur Definition von funktionalen Parsern handelt. Eine erfolgreiche syntaktische Analyse liefert den Wert:

```
(true,[])
```

Dies bedeutet, dass erstens kein Syntaxfehler aufgetreten ist und zweitens die Eingabesequenz vollständig geparst werden konnte. Als fehlerhafte Resultate gelten zum Beispiel:

`(false,...)` oder `(true,[...])`

Das erste Resultat signalisiert einen eindeutigen Syntaxfehler. Das zweite zeigt an, dass zwar eine gewisse Anfangssequenz richtig geparst werden konnte, aber gleichzeitig noch ungeparste Zeichen in der Eingabe verbleiben, was ebenfalls als Syntaxfehler zu interpretieren ist.

Der einfachste Parser ist derjenige, der ein einzelnes Terminalsymbol erkennen kann. Wir setzten die vordefinierten Funktionen `head` und `tail` voraus. `head` liefert das erste Element einer beliebigen Liste und `tail` die Restliste ohne das erste Element.

```
type head : 'a list → 'a
head []              = undefined
head ["A","B","C"] = "A"

type tail : 'a list → 'a list
tail []              = undefined
tail ["A","B","C"] = ["B","C"]
```

Wir definieren den Terminalsymbol-Parser `TERM wie folgt`:

```
type TERMd : char → Parserd
def  TERMd = λ termsym . λ input .
     if input = [] then
       (false,[])
     else
       if head input = termsym then
         (true,tail input)
       else
         (false,input)
       end
     end
```

Wir können jetzt bereits ganz einfache Parser schreiben, welche in der Lage sind, genau ein einzelnes Zeichen zu erkennen:

```
(TERMd "A") ["A","B",...] = (true,["B",...])

(TERMd "B") ["A","B",...] = (false,["A","B",...])
```

Für die Spezifikation mächtigerer Parser benötigen wir Funktionen, die einfache Parser zu komplexeren kombinieren können. Funktionen mit solchen Eigenschaften nennt man Parser-Combinators. Da die Anwendung eines Parsers als Resultat das Paar (Verdikt, Restsequenz) liefert, benötigen wir zusätzlich die beiden vordefinierten Funktionen `first` und `second`, welche uns erlauben, auf die einzelnen Komponenten eines Paares zuzugreifen:

```
type first : 'a × 'b → 'a
first (true,["B",...]) = true

type second : 'a × 'b → 'b
second (true,["B",...]) = ["B",...]
```

In Anlehnung an die (E)BNF-Notation definieren wir zunächst einen solchen Combinator $\mathrm{ALT_d}$, der die alternative Anwendung von zwei Parsern ermöglicht:

```
type ALTd : Parserd → Parserd → Parserd
def  ALTd = λ Q . λ R . λ input .
     let
       type res : bool * char list
       def  res = Q input
     in
       if first res then
         res
       else
         R input
       end
     end
```

Der Combinator $\mathrm{ALT_d}$ erwartet als Argumente zwei Parser `Q` und `R` und liefert als Resultat einen neuen Parser. Dieser wendet zunächst den Parser `Q` auf `input` an. Falls kein Syntaxfehler auftritt, also das Verdikt gleich `true` ist, wird von der Alternative `R` keinen Gebrauch gemacht, sondern das Resultat von `Q` zurückgegeben. Andernfalls wird der Parser `R` auf `input` angewendet. Das von `R` erzeugte Resultat, ob positiv oder negativ, wird dann zum Schlussresultat das neuen Parsers. Der Combinator $\mathrm{ALT_d}$ ist für das Backtracking verantwortlich, indem er nämlich je nach Fall eine oder beide der Optionen `Q` beziehungsweise `R` anwendet. Ausgehend von einer simplen Grammatik können wir nun bereits einen kleinen Parser definieren. Für jede Grammatik-Regel (Produktion) entsteht ein funktionaler Parser. Das Terminalsymbol links vom Gleichheitszeichen liefert den Namen der Funktion. Der Ausdruck auf der rechten Seite wird mit Hilfe von Parser-Combinators und eventuell anderen, auf gleiche Art definierten Parsern zusammengesetzt. Die Grammatik

```
S = "A" | "B" .
```

wird in den funktionalen Parser

```
type Sd : Parserd
def  Sd = ALTd (TERMd "A") (TERMd "B")
```

umgesetzt.

Dazu ein paar Beispiele:

```
Sd ["A","B","C",...] = (true,["B","C",...])

Sd ["B","C",...]     = (true,["C",...])

Sd ["C",...]         = (false,["C",...])
```

Analog zu den obigen Beispielen definieren wir für jedes Metasymbol der (E)BNF einen entsprechenden Parser-Combinator. Die Sequenz wird in (E)BNF nicht durch ein Metasymbol, sondern einfach durch die Hintereinanderstellung von Terminal- und Nonterminalsymbolen ausgedrückt. Der Parser-Combinator SEQ_d hat demzufolge keine explizite Entsprechung in der (E)BNF. SEQ_d wendet den zweiten Parser R nur dann auf die Restsequenz an, wenn der erste Parser Q keinen Syntaxfehler liefert. Andernfalls wird die Fehlermeldung des ersten Parsers zurückgegeben.

```
type SEQd : Parserd → Parserd → Parserd
def  SEQd = λ Q . λ R . λ input .
     let
       type res : bool * char list
       def  res = Q input
     in
       if first res then
         R (second res)
       else
         res
       end
     end
```

EPS_d ist ein spezieller Parser, der immer erfolgreich ist. Das leere Wort ε kann in jedem Fall geparst werden. Die Eingabesequenz bleibt einfach unverändert.

```
type EPSd : Parserd
def  EPSd = λ input . (true,input)
```

Mit den bisher definierten Parser-Combinators sind wir bereits in der Lage, für sämtliche in BNF definierten Grammatiken einen Parser zu schreiben. Das heisst allerdings noch nicht, dass diese Parser in jedem Fall korrekt arbeiten, wie wir später noch sehen werden. Dass die Erweiterung zur EBNF keine mächtigeren Grammatiken liefert, zeigen die Implementierungen von OPT_d und REP_d, welche sich durch die Konstrukte der BNF ausdrücken lassen.

```
type OPTd : Parserd → Parserd
def  OPTd = λ Q . ALTd Q EPSd

type    REPd : Parserd → Parserd
def rec REPd = λ Q . ALTd (SEQd Q (REPd Q)) EPSd
```

Es ist erstaunlich, dass mit so wenigen Grundfunktionen mächtige Parser konstruiert werden können, die zudem die zugrundeliegende (E)BNF-Spezifikation praktisch eins zu eins widerspiegeln. Trotzdem ist Euphorie verfrüht. Die oben definierten Parser-Combinators haben einen gravierenden Mangel. Die nachfolgende einfache Grammatik S - S ist das Startsymbol - lässt nämlich ein Wort zu, das obwohl syntaktisch korrekt, mit diesem Ansatz nicht geparst werden kann.

```
S = A "Y" .
A = "X" "Y" | "X" .
```

Der Parser S_d für die Grammatik S sieht wie folgt aus:

```
type Sd : Parserd
def  Sd = SEQd Ad (TERMd "Y")

type Ad : Parserd
def  Ad = ALTd (SEQd (TERMd "X") (TERMd "Y")) (TERMd "X")
```

Das nachfolgende syntaktisch korrekte Beispiel lässt sich mit S_d nicht parsen:

```
Sd ["X","Y"] = (false,[])
```

Der Parser S_d versagt in diesem Fall, obwohl die Eingabesequenz "XY" eindeutig zur Sprache S gehört. Der Parser verfügt zwar grundsätzlich über Backtracking-Fähigkeiten, es gibt aber offenbar Fälle, wo diese versagen. Was genau läuft schief?

Der Syntaxfehler wird durch den Combinator SEQ_d im Parser S_d festgestellt. Dieser wendet zuerst den Parser A_d auf die Eingabesequenz "XY" an und erwartet danach als Resteingabe die Sequenz "Y". Innerhalb des Parsers A_d konsumiert die erste Alternative des Combinators ALT_d bereits die gesamte Eingabesequenz, was zu diesem Zeitpunkt und aus lokaler Sicht völlig korrekt ist. Die zweite Alternative, die global gesehen die richtige wäre, kommt in diesem Beispiel gar nie zum Zug. Der Combinator SEQ_d im Parser S_d wendet nun sein zweites Argument auf eine leere Eingabesequenz an, was natürlich zu einem Syntaxfehler führt. Die direkte Methode, die wir hier gezeigt haben, besitzt leider nur limitierte Backtracking-Fähigkeiten, die von einer lokalen Sicht des Parsing-Vorgangs geprägt sind.

3.2.1.2 Direkter Ansatz mit vollem Backtracking

Falls wir das oben diskutierte Problem lösen wollen, dann müssen wir die Parser-Combinators so erweitern, dass sie beliebige kontextfreie Grammatiken parsen können.

Der Parser-Combinator ALT_d ist nur dann backtracking-fähig, wenn er einen Syntaxfehler selbst entdeckt. Sobald die erste Alternative als syntaktisch korrekt akzeptiert wird, kommt die zweite nie mehr zum Zug. ALT_d und die darauf aufbauenden Parser haben deshalb nur eine lokale Sicht des gesamten Parsing-Vorgangs. Wenn mehrere Alternativen gleichzeitig möglich sind, dürfen wir uns in Zukunft nicht auf eine bestimmte beschränken, sondern müssen zunächst alle akzeptieren. Wir erweitern deshalb die Definition des Datentyps $Parser_d$ wie folgt:

```
type Parserdc = char list → (bool * char list) list
```

Mit dem Index **dc** bezeichnen wir jetzt den **d**irekten Ansatz für beliebige kontextfreie Grammatiken (**c**fg). Weil es nun möglich ist, dass ein Parser nicht nur ein einzelnes Resultat, sondern eine ganze Menge davon erzeugt, müssen wird den Datentyp des Resultats zu einer Liste erweitern. Genaugenommen müsste man nicht Listen, sondern Mengen verwenden. Die Parser-Combinators sind entsprechend anzupassen. Obwohl $TERM_{dc}$ auch jetzt nur ein einzelnes Resultat liefert, muss dieses in einer Liste verpackt zurückgegeben werden:

```
type TERMdc : char → Parserdc
def  TERMdc = λ termsym . λ input .
     if input = [] then
       [ (false,[]) ]
     else
       if head input = termsym then
         [ (true,tail input) ]
       else
         [ (false,input) ]
       end
     end
```

Der Combinator $\texttt{ALT}_{dc}$ verfolgt jetzt in jedem Fall beide Alternativen. Die beiden als Liste zurückgegeben Einzelresultate werden mit Hilfe der vordefinierten Funktion `concatenate` zu einer einzigen zusammengefügt.

```
type concatenate : 'a list → 'a list → 'a list
concatenate ["A","B"] ["C","D"] = ["A","B","C","D"]
```

Obwohl der Combinator $\texttt{ALT}_{dc}$ einfacher aussieht als $\texttt{ALT}_{d}$, darf man sich nicht darüber hinwegtäuschen lassen, dass $\texttt{ALT}_{dc}$ erheblich mehr Rechenzeit benötigt:

```
ALTdc : Parserdc → Parserdc → Parserdc
def ALTdc = λ Q . λ R . λ input . concatenate (Q input) (R input)
```

Der Combinator $\texttt{SEQ}_{dc}$ wendet den zweiten Parser `R` auf jedes einzelne syntaktisch korrekte Resultat der vom ersten Parser `Q` erzeugten Liste an. Die Implementierung von $\texttt{SEQ}_{dc}$ wird viel komplexer als diejenige von $\texttt{SEQ}_{d}$, und wir benötigen einige Hilfsfunktionen. Zunächst setzten wir voraus, dass eine vordefinierte Funktion `insert` existiert, welche ein Element am Anfang einer Liste einfügt:

```
type insert : 'a → 'a list → 'a list
insert "A" ["B","C"] = ["A","B","C"]
```

Die Funktion `map` wendet eine beliebige Funktion auf sämtliche Elemente einer Liste an:

```
type    map : ('a → 'b) → 'a list → 'b list
def rec map = λ f . λ lst .
        if lst = [] then
          []
        else
          insert (f (head lst)) (map f (tail lst))
        end

map (λ x . 2 * x) [1,2,3,4,5] = [2,4,6,8,10]
```

Die Funktion `flatten` hängt eine Liste, deren Elemente selbst Listen sind, zu einer einfachen Liste zusammen:

```
type    flatten : 'a list list → 'a list
def rec flatten = λ lstlst .
        if lstlst = [] then
          []
        else
          concatenate (head lstlst) (flatten (tail lstlst))
        end

flatten [["A","B"],[],["C","D","E"]] = ["A","B","C","D","E"]
```

Mit Hilfe der oben definierten Hilfsfunktionen scheint die Struktur von $\texttt{SEQ}_{dc}$ nicht wesentlich komplizierter zu sein als diejenige von $\texttt{SEQ}_{d}$. $\texttt{SEQ}_{dc}$ erzeugt aber wesentlich mehr Daten und benötigt auch mehr Rechenzeit.

```
type SEQdc : Parserdc → Parserdc → Parserdc
def  SEQdc = λ Q . λ R . λ input .
     let
       type applyR : (bool * char list) → (bool * char list) list
       def  applyR = λ res .
            if first res then
```

```
            R (second res)
          else
            [ res ]
          end
      in
        flatten (map applyR (Q input))
      end
```

Der Combinator $\mathtt{EPS}_{dc}$ ändert sich kaum. Er muss lediglich sein Resultat als Liste zurückgeben:

```
type EPSdc : Parserdc
def  EPSdc = λ input . [ (true,input) ]
```

$\mathtt{OPT}_{dc}$ und $\mathtt{REP}_{dc}$ lassen sich auch hier aus den oben definierten Konstrukten der BNF ableiten:

```
type OPTdc : Parserdc → Parserdc
def  OPTdc = λ Q . ALTdc Q EPSdc

type    REPdc : Parserdc → Parserdc
def rec REPdc = λ Q . ALTdc (SEQdc Q (REPdc Q)) EPSdc
```

Mit diesen Combinators sind wir in der Lage, Parser zu definieren, die sämtliche kontextfreien Sprachen analysieren können. Wir müssen allerdings eine Restriktion bei der Notation in Kauf nehmen. Linksrekursive Produktionen sind verboten. Wir werden später auf dieses grundsätzliche Problem von Top-Down-Parsern kurz eingehen. Obwohl sich die strukturelle Komplexität der Parser-Combinators nicht wesentlich erhöht hat, darf man doch nicht darüber hinwegsehen, dass ein Parsing-Vorgang jetzt wesentlich mehr Speicherplatz und Laufzeit benötigen kann. Dafür erhalten wir als Gegenleistung bei mehrdeutigen Grammatiken sämtliche Lösungen. Auch das problematische Beispiel, das wir oben vorgestellt haben, kann jetzt mit Hilfe der neuen Parser-Combinators korrekt geparst werden. Der Aufruf

```
Sdc ["a","b"]
```

hat zur Folge, dass zunächst $\mathtt{A}_{dc}$ aktiviert wird:

```
Adc ["a","b"] = [(true,[]),(true,["b"])]
```

$\mathtt{A}_{dc}$ liefert eine Liste mit beiden aus seiner Sicht richtigen Teillösungen. Diese werden an den Aufrufer $\mathtt{S}_{dc}$ zurückgegeben. Innerhalb von $\mathtt{S}_{dc}$ wird jetzt das zweite Argument von $\mathtt{SEQ}_{dc}$ auf die Resultatliste von $\mathtt{A}_{dc}$ angewendet. Das erste Element der Resultatliste führt nach wie vor zu einem Syntaxfehler. Das zweite liefert jetzt aber eine syntaktisch korrekte Ableitung. Wir erhalten:

```
[(false,[]),(true,[])]
```

Die Lösungsmenge beinhaltet nicht nur sämtliche korrekten syntaktischen Analysen, sondern auch alle wegen Syntaxfehlern abgebrochenen Versuche.

Der hier geschilderte Ansatz ist zwar sehr mächtig, aber für reale Programmiersprachen kein gangbarer Weg zur syntaktischen Analyse. Wir müssen uns vor Augen halten, dass wir nicht an Recognizern, sondern an Parsern interessiert sind, die zu jeder korrekten Lösung einen Parse-Baum und zu jeder inkorrekten eine aussagekräftige Fehlermeldung generieren. Ein Parser, der auf diesem Ansatz basiert, wird deshalb eine Unmenge von Daten erzeugen, die den Anwender häufig gar nicht interessieren, sondern lediglich zur Steuerung des Parsing-Vorgangs dienen.

Im nächsten Abschnitt werden wir zeigen, dass es elegantere Alternativen gibt, um das obige Problem zu lösen. Die Parser-Combinators können mit Hilfe von Continuations so erweitert werden, dass wir an jeder Stelle des Parsers eine globale Sicht des Problems haben und zwar so, dass wir nicht gezwungen sind, sämtliche möglichen Ableitungen zu generieren.

3.2.2 Continuations

Funktionale Programme bestehen aus Kombinationen von verschiedenen Funktionen. Die einfachste und uns wohl allen vertrauteste Art, Funktionen miteinander zu kombinieren, ist die Verkettung. `F` und `G` seien zwei beliebige Funktionen und `'a`, `'b` beziehungsweise `'c` beliebige Datentypen:

```
F   : 'b → 'c
G   : 'a → 'b
ARG : 'a
```

Wir können dann folgende Verkettung bilden:

`F (G ARG)` oder anders geschrieben `(F o G) ARG`

Auch bei der Verkettung kann man sagen, dass die einzelnen beteiligten Funktionen nur eine lokale Sicht des Problems haben. Die Funktion `G` berechnet aus `ARG` einen Wert und hat dann auf nachfolgende Berechnungen keinen Einfluss mehr. Zur Verdeutlichung des Datenflusses betrachte man die nachfolgende Grafik (Abbildung 3):

ABBILDUNG 3. DATENFLUSS BEI DER DIREKTEN VERKETTUNG

Wenn wir Funktionen höherer Ordnung einsetzen, bietet sich eine Vielzahl von Möglichkeiten, um interessante Effekte zu erzielen, die sich für die Realisierung von flexibleren Konzepten eignen.

3.2.2.1 Was sind Continuations

Mit Hilfe von Continuations können wir den oben geschilderten Sachverhalt ändern. Wir werden alle an einer Berechnung teilnehmenden Funktionen so erweitern, dass am Ende der gesamten Berechnung jeder einzelnen von ihnen das Schlussresultat präsentiert wird. Auf Grund dieses Schlussresultats können wir an jeder beliebigen Stelle des funktionalen Programms weitere Aktionen veranlassen. Jede Funktion erhält so während einer Berechnung die Kontrolle zweimal. Einmal, wenn sie ihren direkten Beitrag zur Berechnung liefert und ein zweites Mal, um das Schlussresultat zu überprüfen und eventuell weiterzuverarbeiten. Es ist einleuchtend, dass solche Möglichkeiten hervorragend geeignet sind, um Backtracking-Probleme zu lösen.

Bevor wir nun Parser-Combinators mit Hilfe von Continuations definieren, zeigen wir deren Prinzip. In Anlehnung an obiges Beispiel zur Verkettung von Funktionen, erweitern wir die beiden Grundfunktionen `F` und `G` folgendermassen:

```
type F' : ('c → Answer) → 'b → Answer
def  F' = λ cont_F . λ x . cont_F (F x)

type G' : ('b → Answer) → 'a → Answer
def  G' = λ cont_G . λ x . cont_G (G x)
```

`Answer` definiert den Datentyp des Schlussresultats und kann je nach Bedarf von beliebiger Struktur sein. Ein einfacher Datentyp wie `int` ist genauso möglich, wie eine zusammengesetzter. Bei den Funktionen $\mathtt{cont}_F$ und $\mathtt{cont}_G$ handelt es sich um sogenannte Continuations (Fortsetzungen). $\mathtt{cont}_F$ beinhaltet sämtliche auf `F` folgenden und $\mathtt{cont}_G$ sämtliche auf `G` folgenden Berechnungen. Offenbar umfasst `F'` die Continuation von `G'`, denn `F` wird ja nach `G` ausgeführt:

```
G' (F' ?)
= (λ cont_G . λ x . cont_G (G x)) ((λ cont_F . λ x . cont_F (F x)) ?)
```

Es stellt sich nun die Frage: Was ist die Continuation von `F'`? Bevor wir die Funktionen `F'` und `G'` miteinander kombinieren können, müssen wir zuerst eine weitere Funktion einführen, welche die Berechnungskette terminiert. Wir definieren zu diesem Zweck die Funktion `FIN` (Final Continuation) wie folgt:

```
type FIN : 'c → Answer
```

Die Implementierung der Funktion `FIN` ist problemabhängig. Wir können deshalb lediglich ihren Datentyp angeben. `FIN` hat im wesentlichen nur die Aufgabe, das Argument vom Datentyp `'c`, welches das Schlussresultat der gesamten Berechnung repräsentiert, in den Wert vom Datentyp `Answer` umzuwandeln. Wir können nun die ursprünglichen Funktionen `F` und `G` via `F'` und `G'` wie folgt verknüpfen:

```
(G' (F' FIN)) ARG
```

Durch einfache Umformungen zeigen wir, dass aus dem Continuation-Ausdruck, abgesehen von `FIN`, tatsächlich wieder der ursprüngliche Ausdruck `F (G ARG)` entsteht:

```
(G' (F' FIN)) ARG
= (G' ((λ cont_F . λ x . cont_F (F x)) FIN)) ARG
= (G' (λ x . FIN (F x))) ARG
= ((λ cont_G . λ x . cont_G (G x)) (λ x . FIN (F x))) ARG
= (λ x . (λ x . FIN (F x)) (G x)) ARG
= (λ x . FIN (F (G x))) ARG
= FIN (F (G ARG))
```

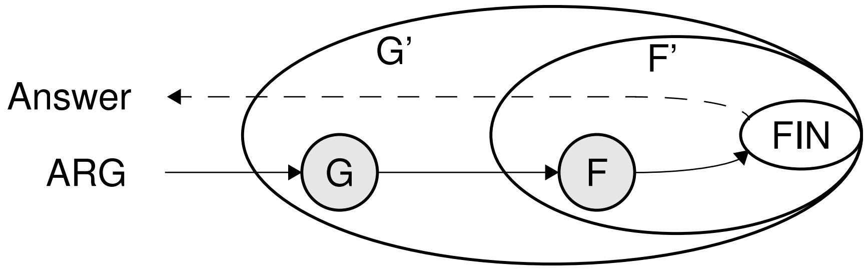
Mit Hilfe von Continuations können wir natürlich nicht nur zwei, sondern beliebig viele Funktionen miteinander kombinieren und müssen dabei nur auf die Datentypen achten:

```
(F_1 (F_2 (... (F_i (F_i+1 (... (F_n-1 (F_n FIN))...)))...))) ARG
```

Offenbar gilt für jedes i, mit 0 < i < n: Der gesamte Ausdruck $(\mathtt{F}_{i+1}\ \ldots)$ ist die Continuation der Funktion $\mathtt{F}_i$. `FIN` ist die Continuation der Funktion $\mathtt{F}_n$.

Was haben wir durch die Verwendung von Continuations gewonnen? Zunächst scheint es so, als ob wir die Komplexität beträchtlich erhöht hätten. Das stimmt aber nur bedingt. Es ist richtig, dass wir zusätzliche Funktionen einführen und somit das funktionale Programm vergrössern. Da diese zusätzlichen Funktionen alle nach einem einheitlichen Muster aufgebaut sind, werden Continuation-Programme in der Regel eher übersichtlicher. Das Arbeiten mit Continuations ist eine reine Gewohnheitssache. Mit Hilfe von Continuations ist es möglich, für funktionale Programme einen Kontrollfluss zu definieren und mit einfachen Mitteln zu manipulieren (Abbildung 4).

ABBILDUNG 4. DATEN- UND KONTROLLFLUSS BEI DER VERKETTUNG MIT CONTINUATIONS



Betrachten wir die Funktion `G'` etwas genauer.

```
type G' : ('b → Answer) → 'a → Answer
def  G' = λ cont_G . λ x . cont_G (G x)
```

Wir stellen fest, dass wir innerhalb von `G'` mehr Möglichkeiten haben, um in den Berechnungsablauf einzugreifen, als wenn wir `G` direkt verwenden. Nachdem wir $\mathtt{cont}_G$ `(G x)` ausgeführt haben, erhalten wir einen Wert vom Datentyp `Answer`. Dieser repräsentiert das Schlussresultat der gesamten Berechnungskette. Insbesondere ist auch die Auswirkung der Funktion `F` mitberücksichtigt, die ja innerhalb der Continuation $\mathtt{cont}_G$ zum Zuge kommt. Auf Grund des Schlussresultats, können wir verschiedene Aktionen veranlassen. Im einfachsten Fall geben wir das Schlussresultat einfach unverändert weiter. Wir können dieses aber auch manipulieren, bevor wir es weitergeben. Als weitere interessante Alternative bietet sich die Möglichkeit, $\mathtt{cont}_G$ beliebig oft mit immer anderen Werten aufzurufen, was für Backtracking-Probleme ideal ist. Es steht uns auch jederzeit frei, die Berechnungskette sofort abzubrechen. Wir brauchen dann nur anstatt $\mathtt{cont}_G$ zu aktivieren, direkt ein Resultat vom Datentyp `Answer` zurückzugeben.

Beim obigen Beispiel ist die Tatsache unbefriedigend, dass sämtliche Continuations einen unterschiedlichen Datentyp aufweisen. Das ist aber keine notwendige Voraussetzung. Es gibt eine Klasse von Funktionen, die sich durch Continuations desselben Datentyps `('a → Answer)` nachbilden lassen. Glücklicherweise gehören zu dieser Klasse auch die Parser-Combinators, wie wir im nächsten Abschnitt sehen werden.

3.2.3 Parser-Combinators basierend auf Continuations

3.2.3.1 Deterministischer Continuation-Ansatz

Aufbauend auf den oben vorgestellten Konzepten ist es nun ein leichtes, Parser-Combinators mit Hilfe von Continuations zu realisieren. Zunächst definieren wir die grundlegenden Datentypen. Wir gehen dabei von ähnlichen Annahme wie beim direkten Ansatz aus. Die Eingabesequenz liege als eine Liste von Zeichen vor. Weil wir uns auch hier mit einem Recognizer zufrieden geben wollen, genügt als Resultat ein Wert vom Datentyp `bool`. Wir definieren deshalb die Datentypen $\texttt{Answer}_c$, $\texttt{Continuation}_c$ und $\texttt{Parser}_c$ wie folgt:

```
type Answerc   = bool
type Continuationc= char list → Answerc
type Parserc   = Continuationc → Continuationc
```

Mit dem Index **c** drücken wir aus, dass wir den **C**ontinuation-Ansatz verwenden. Wir definieren wieder als erstes den Terminalsymbol-Parser.

```
type TERMc : char → Parserc
def  TERMc = λ termsym . λ cont . λ input .
     if input = [] then
       false
     else
       if head input = termsym then
         cont (tail input)
       else
         false
       end
     end
```

Als erstes fällt auf, dass der Combinator $\texttt{TERM}_c$ ein zusätzliches Argument `cont`, eine Continuation erhalten hat. `cont` kann man sich als Funktion vorstellen, die nach der erfolgreichen Anwendung von $\texttt{TERM}_c$ den gesamten Rest der Eingabe parst. Wenn innerhalb von $\texttt{TERM}_c$ ein Fehler auftaucht, dann wird das Parsing mit der Antwort `false` abgebrochen und die Continuation `cont` nicht aufgerufen. $\texttt{TERM}_c$ erzeugt als Resultat immer einen Wert vom Datentyp $\texttt{Answer}_c$, im Fehlerfalle direkt oder sonst über den Aufruf von `cont`. Diese Antwort wird von allen Teil-Parsern gesehen, die vor $\texttt{TERM}_c$ aktiv waren. Analog zu $\texttt{TERM}_c$ definieren wir die weiteren Parser-Combinators:

```
type ALTc : Parserc → Parserc → Parserc
def  ALTc = λ Q . λ R . λ cont . λ input .
     let
       type ans : Answerc
       def  ans = Q cont input
     in
       if ans then
         ans
       else
         R cont input
       end
     end
```

Der Combinator $\texttt{ALT}_c$ entscheidet sich jetzt nicht mehr aus lokaler Sicht, ob er die zweite Alternative `R` anwenden soll oder nicht. Das Resultat `ans` enthält nämlich das Schlussresultat, welches durch die Anwendung von `Q` und der auf $\texttt{ALT}_c$ folgenden Continuation `cont` erzeugt worden ist. Wenn `ans` gleich `true` ist, dann wissen wir mit Bestimmtheit, dass eine korrekte Syntaxanalyse möglich war und wir brauchen `R` nicht mehr anzuwenden.

Die Definition der übrigen Parser-Combinators ist trivial und widerspiegelt die Eleganz des Continuation-Ansatzes:

```
type SEQc : Parserc → Parserc → Parserc
def  SEQc = λ Q . λ R . λ cont . Q (R cont)

type EPSc : Parserc
def  EPSc = λ cont . cont

type OPTc : Parserc → Parserc
def  OPTc = λ Q . ALTc Q EPSc

type    REPc : Parserc → Parserc
def rec REPc = λ Q . ALTc (SEQc Q (REPc Q)) EPSc
```

Ist der Ansatz mit Continuations tatsächlich mächtiger als der direkte Ansatz? Um zu zeigen, dass wir das obige Problembeispiel parsen können, definieren wir zunächst eine geeignete End-Continuation (Final Continuation):

```
type FINc : Continuationc
def  FINc = λ input . input = []
```

In diesem Fall muss `FIN` überprüfen, ob der Parsing-Vorgang erfolgreich war oder nicht. `FIN` steht gewissermassen für den Endzustand. Wenn `FIN` aufgerufen wird, dann wissen wir, dass bisher kein Syntaxfehler aufgetreten ist. Um das Endresultat vom Datentyp `bool` zu erzeugen, brauchen wir also nur zu testen, ob die verbleibende Eingabesequenz leer ist. Wir können jetzt in Anlehnung an obiges Beispiel den Parser $\mathtt{S_c}$ definieren. Der Einfachheit halber wiederholen wir die Grammatik:

```
S = A "Y" .
A = "X" "Y" | "X" .
```

Der zugehörige Parser $\mathtt{S_c}$ unterscheidet sich strukturell nicht vom Parser $\mathtt{S_d}$:

```
type Sc : Parserc
def  Sc = SEQc Ac (TERMc "Y")

type Ac : Parserc
def  Ac = ALTc (SEQc (TERMc "X") (TERMc "Y")) (TERMc "X")
```

Im Gegensatz zum Parser $\mathtt{S_d}$ kann $\mathtt{S_c}$ das nachfolgende Beispiel korrekt parsen:

```
(Sc FIN) ["X","Y"] = true
```

Der Parsing-Vorgang ist jetzt allerdings nicht mehr so einfach zu durchschauen wie beim direkten Ansatz. Zur besseren Übersicht rechnen wir den Ausdruck $\mathtt{(S_c\ FIN)}$ aus und erhalten:

```
Sc FIN
= (SEQc Ac (TERMc "Y")) FIN
= ((λ Q . λ R . λ cont . Q (R cont)) Ac (TERMc "Y")) FIN
= ((λ R . λ cont . Ac (R cont)) (TERMc "Y")) FIN
= (λ cont . Ac ((TERMc "Y") cont)) FIN
= Ac ((TERMc "Y") FIN)
```

Wir stellen fest, dass es sich beim Ausdruck $\mathtt{((TERM_c\ "Y")\ FIN)}$ um die Continuation von $\mathtt{A_c}$ handelt. Der Parser $\mathtt{A_c}$ wird als erster aktiviert. Der Combinator $\mathtt{ALT_c}$ innerhalb von $\mathtt{A_c}$ wählt, ähnlich wie beim direkten Ansatz, zunächst die falsche Alternative. Danach ruft er seine Continuation, also $\mathtt{((TERM_c\ "Y")\ FIN)}$, mit der leeren Eingabesequenz auf, was zu einem Syntaxfehler führt und die Antwort `false` erzeugt. Diese Antwort wird vom Combinator $\mathtt{ALT_c}$ gesehen und veranlasst diesen, die zweite Alternative auszuprobieren, die in diesem Fall zum Erfolg führt. Wir erhalten das korrekte Resultat `true`.

3.2.3.2 Continuation-Ansatz für beliebige kontextfreie Grammatiken

Wir haben gesehen, dass mit dem Continuation-Ansatz immer eine Lösung gefunden wird, falls eine solche existiert. Wenn wir bei mehrdeutigen Grammatiken an sämtlichen Lösungen interessiert sind, dann können wir natürlich auch den Continuation-Ansatz für beliebige kontextfreien Grammatiken erweitern. Wir verwenden den Index **cc** mit dem wir jetzt den **C**ontinuation-Ansatz für beliebige kontextfreie Grammatiken (**c**fg) bezeichnen.

Auch jetzt müssen wir darauf vorbereitet sein, dass ein Parser möglicherweise mehrere Resultate liefern kann. Wir erweitern deshalb den Datentyp $\texttt{Answer}_c$ wie folgt:

```
type Answer_cc = bool list
```

Die übrigen Datentypen bleiben strukturell unverändert:

```
type Continuation_cc= char list → Answer_cc
type Parser_cc  = Continuation_cc → Continuation_cc
```

Die Parser-Combinators ändern sich kaum. $\texttt{TERM}_{cc}$ gibt jetzt sein (eindeutiges) Resultat in einer Liste verpackt zurück.

```
type TERM_cc : char → Parser_cc
def  TERM_cc = λ termsym . λ cont . λ input .
      if input = [] then
        [ false ]
      else
        if head input = termsym then
          cont (tail input)
        else
          [ false ]
        end
      end
```

$\texttt{ALT}_{cc}$ wendet in jedem Fall beide Alternativen an und vereinigt die beiden Resultatlisten zu einer einzigen:

```
type ALT_cc : Parser_cc → Parser_cc → Parser_cc
def  ALT_cc = λ Q . λ R . λ cont . λ input .
     concatenate (Q cont input) (R cont input)
```

Die übrigen Parser-Combinators bleiben erstaunlicherweise völlig unverändert:

```
type SEQ_cc : Parser_cc → Parser_cc → Parser_cc
def  SEQ_cc = λ Q . λ R . λ cont . Q (R cont)

type EPS_cc : Parser_cc
def  EPS_cc = λ cont . cont

type OPT_cc : Parser_cc → Parser_cc
def  OPT_cc = λ Q . ALT_cc Q EPS_cc

type    REP_cc : Parser_cc → Parser_cc
def rec REP_cc = λ Q . ALT_cc (SEQ_cc Q (REP_cc Q)) EPS_cc
```

Die End-Continuation `FIN` liefert jetzt ihr Resultat ebenfalls in einer Liste:

```
type FINcc : Continuationcc
def  FINcc = λ input . [ input = [] ]
```

Als Resultat einer syntaktischen Analyse erhalten wir eine Liste von `bool`-Werten.

3.3 Kritik

Bei beiden funktionalen Ansätzen, die wir oben gezeigt haben, handelt es sich um Top-Down-Verfahren. Diese haben eine generelle Einschränkung bei der Definition von Grammatiken. Linksrekursive Produktionen sind verboten. Man betrachte das nachfolgende Beispiel:

Grammatik:

```
S = S "b" | "a" .
```

Zugehöriger Parser:

```
type    S : Parser
def rec S = ALT (SEQ S (TERM "b")) (TERM "a")
```

Die Produktion `S` ist linksrekursiv. Der aus `S` abgeleitete Parser ist eine Funktion, die nicht terminiert. `S` ruft als erstes wieder sich selbst auf, ohne die Eingabesequenz zu verkleinern. Wir haben einen endlosen rekursiven Aufruf erzeugt. Man kann das Problem lösen, indem man die Grammatik umschreibt, so dass sich die Syntax der formalen Sprache nicht ändert, aber die Linksrekursion verschwindet. In (Aho, Sethi et al. 1986) wird ein Algorithmus vorgestellt, der auf BNF-Grammatiken angewendet werden kann. Von dieser automatischen Umformung ist allerdings abzuraten, denn sie verändert die Grammatik in weiten Teilen. Die ausdrucksstärkere EBNF-Notation bietet genug Möglichkeiten, um linksrekursive Produktionen zu umgehen. Bei der Eigenentwicklung unserer COBOL-74-Grammatik hat sich diese Einschränkung nie als Problem erwiesen.

Abgesehen von diesem kleinen Nachteil haben Top-Down-Verfahren den Vorteil, dass sie automatisch zu jeder Produktion einer Grammatik einen eigenen unabhängigen Parser liefern. Das ist eine notwendige, allerdings noch keine hinreichende Bedingung, um inkrementelle Parser zu implementieren. Parser für beliebige kontextfreie Sprachen sind weniger effizient als Parser für bestimmte Subklassen davon. Um diesen Nachteil auszugleichen, ist es wünschenswert, bei Änderungen nicht das gesamte Programm neu zu analysieren, sondern lediglich die geänderten Teile davon. Zur Zeit wird abgeklärt, ob und wie sich CoP zu einem Generator für inkrementelle Parser erweitern lässt.

4.0 CoP ein funktionaler Parser-Generator

In diesem Abschnitt werden wir CoP - A **Co**mbinator Based **P**arser Generator - vorstellen. CoP befindet sich derzeit noch im Prototyp-Stadium. Trotzdem ist es bereits möglich, einen Parser für die Procedure-Division von COBOL-74-Programmen zu generieren. Der generierte Parser wurde mit Erfolg an verschiedenen Programmen aus der Praxis getestet und hat sich dabei als erstaunlich leistungsstark erwiesen. Die zugrundeliegende COBOL-74-Grammatik ist in einer für CoP adaptierten Form der EBNF geschrieben. Sie wurde mit Hilfe von CoP entwickelt, ausführlich getestet und laufend verbessert. CoP erzeugt als interne Darstellung einen Parse-Baum, der zusätzlich Positionsinformationen der geparsten syntaktischen Strukturen enthält. Natürlich ist CoP nicht auf COBOL-74 beschränkt. Wir werden nachfolgend beschreiben, wie man von der Grammatik einer beliebigen kontextfreien Sprache vollständig automatisch zu einer internen Darstellung für Programme (Wörter) aus dieser Sprache kommt. Zunächst fassen wir aber die wichtigsten Entwurfskriterien von CoP zusammen.

4.1 Entwurfskriterien

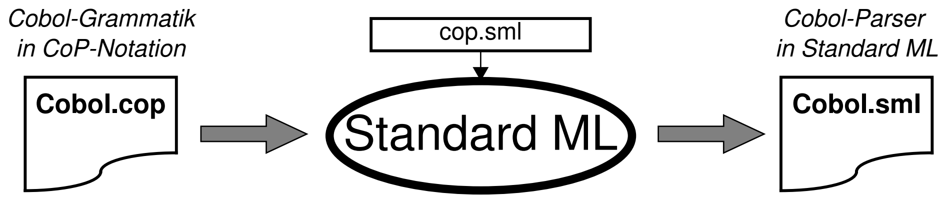
Der Parser-Generator CoP ist massgeschneidert für den Einsatz in Wartungsumgebungen und deshalb für die Konstruktion von Compilern nicht geeignet. Folgende Kriterien haben den Entwurf von CoP massgeblich beeinflusst:

1. CoP soll Parser für beliebige kontextfreie Sprachen generieren können. Dabei nehmen wir die Einschränkung in Kauf, dass linksrekursive Produktionen nicht zugelassen sind.
2. Bei mehrdeutigen Grammatiken soll CoP nur die erste Lösung finden. Diese Einschränkung ist vernünftig. Viele Programmiersprachen, z.B. auch PASCAL (IF-THEN-ELSE), besitzen eine mehrdeudige Grammatik, können aber doch eindeutig geparst werden, weil man sich im voraus auf eine bestimmte Lösung festlegt und den Parser dementsprechend implementiert. Was wir genau unter der ersten Lösung verstehen, werden wir später erläutern.
3. CoP soll die interne Struktur vollautomatisch aufbauen. Der Entwickler einer Grammatik kann sich ausschliesslich auf die syntaktischen Strukturen konzentrieren und muss sich nicht mit der Generierung von internen Darstellungen abgeben, wie dies z.B. bei der Verwendung von LEX/YACC nötig ist.
4. CoP soll sowohl mit EBNF- als auch mit BNF-Spezifikationen umgehen können.
5. Aus Effizienzgründen ist dem Parsing-Vorgang praktisch immer ein Scanning der Eingabe vorgeschaltet. CoP soll deshalb nicht nur einen Parser, sondern auch einen Scanner automatisch generieren.
6. Programm-Kommentare dürfen nicht einfach eliminiert werden, sondern sind in die interne Darstellung einzubauen.
7. Die von CoP erzeugten Parser sollen nicht nur Experimentierzwecken dienen, sondern mit Programmen aus der Praxis fertig werden.

4.2 Architektur von CoP

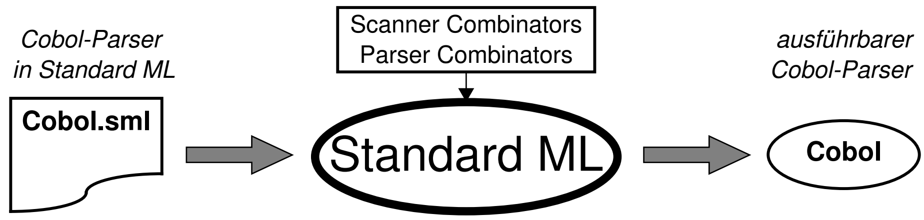
CoP ist vollständig in der funktionalen Programmiersprache Standard ML implementiert. Die Erzeugung eines Parsers verläuft in zwei Phasen. In der ersten, der Generierungs-Phase, wird der Interpreter von Standard ML mit dem Parser-Generator (cop.sml) initialisiert (siehe Abbildung 5). Danach kann eine beliebige Grammatik-Spezifikation eingelesen werden (hier z.B. **Cobol.cop**). Als Resultat erhält man einen Parser, der ebenfalls in Standard ML geschrieben ist (in diesem Fall **Cobol.sml**).

ABBILDUNG 5. GENERIERUNGS-PHASE



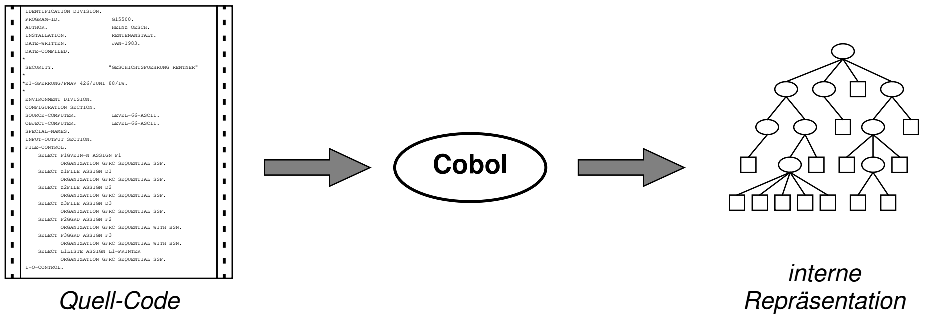
Der erzeugte Parser (**Cobol.sml**) ist noch nicht direkt ausführbar. Wir benötigen eine zweite Phase, die Kompilierungs-Phase, um einen ausführbaren Parser zu erhalten (siehe Abbildung 6). Dazu verwenden wir generische Parser- und Scanner-Combinators, wie wir sie im vorhergehenden Abschnitt definiert haben und übergeben diese zusammen mit dem generierten Parser dem Standard ML Interpreter, welcher schliesslich den ausführbaren Parser (**Cobol**) erzeugt. Generisch bedeutet hier, dass die Combinators völlig sprachunabhängig sind und unverändert für sämtliche erzeugten Parser eingesetzt werden können.

ABBILDUNG 6. KOMPILIERUNGS-PHASE



Der ausführbare Parser kann als Black-Box betrachtet werden, der ein syntaktisch korrektes Programm in eine interne Repräsentation übersetzt (siehe Abbildung 7). Die interne Repräsentation wird in eine Datei gespeichert, von wo aus sie für Weiterverarbeitungen durch beliebige Wartungswerkzeuge zugänglich ist.

ABBILDUNG 7. SYNTAX-ANALYSE



Bevor wir auf die einzelnen Phasen näher eingehen, zeigen wir die Definition der internen Repräsentation, also des Parse-Baums. Ab jetzt werden wir direkt Standard ML als funktionale Notation verwenden. Eine kurze, aber dennoch umfassende Einführung in Standard ML findet man in (Harper 1993).

4.3 Interne Repräsentation

Der Parser eines Compilers baut in der Regel den konkreten Syntaxbaum (Parse-Baum) nicht explizit auf. Das hat Gründe, die mit der Effizienz zusammenhängen. Der Parse-Baum ist eine sehr grosse Datenstruktur und enthält viele Informationen, die zur Code-Erzeugung nicht notwendig sind. So dienen z.B. Schlüsselwörter wie "BEGIN" und "END" nur der Steuerung des Parsing-Vorgangs und werden nicht direkt in Code umgesetzt. Für eine Wartungsumgebung wie AEMES ist der Parse-Baum hingegen die ideale interne Repräsentation. Erstens kann er aus einer Grammatik-Spezifikation vollständig automatisch aufgebaut werden. Zweitens ist der Parse-Baum - erweitert mit Positionsinformationen - eine eineindeutige interne Darstellung eines geparsten Programms. Drittens handelt es sich beim Parse-Baum um eine einfache Datenstruktur, welche für sämtliche formalen Sprachen gleichermassen geeignet ist. Wir geben die Definition in Standard ML:

Der Typ `Location` beschreibt eine beliebige Position innerhalb eines Programms. Die Komponenten des Tripels haben die Bedeutung: (Absolute Position, Zeile, Spalte).

```
type Location = int * int * int
```

Mit Hilfe einer `Annotation` können wir jede syntaktische Struktur eines geparsten Programms exakt lokalisieren. Wir notieren zu jeder Struktur ihre Anfangs- und Endposition. Solche Informationen sind z.B. für die Code-Erzeugung irrelevant, in einer Wartungsumgebung aber sehr hilfreich:

```
type Annotation = { FROM : Location, TO : Location }
```

Der Parse-Baum ist ein n-ärer Baum. Wir verwenden deshalb eine Listenstruktur als grundlegende Repräsentation:

```
datatype Tree = TREE        of Node list
and      Node = TERMINAL    of string * string * Annotation
              | NONTERMINAL of string * Tree   * Annotation
```

Ein Parse-Baum (`Tree`) ist eine Liste von Knoten (`Node list`). Es existieren zwei Klassen von Knoten (`Node`). Eine beschreibt Terminal- und die andere Nonterminalsymbole. Bei Terminalsymbolen (`TERMINAL`) speichern wir ein Tripel der Form (Typ, Wert, Position). Ein Integer mit dem Wert “123” wird z.B. folgendermassen erfasst:

```
TERMINAL("INTEGER","123",{FROM=(...),TO=(...)})
```

Für jedes Nonterminalsymbol (`NONTERMINAL`) existiert eine Grammatikregel (Produktion) der Form:

```
P = (E)BNF-Expression .
```

`(E)BNF-Expression`, die rechte Seite der Produktion `P`, liefert die Komponenten, aus denen `P` aufgebaut ist. Diese Komponenten werden durch eine Liste von Sub-Bäumen `S` repräsentiert. Der Parse-Baum von `P` ist ein Tripel (Typ, `S`, Position). Die obige Produktion wird folgendermassen in einen Parse-Baum abgebildet:

```
NONTERMINAL("P",TREE S,{FROM=(...),TO=(...)})
```

Wir wollen an einem kleinen Beispiel die Zusammenhänge verdeutlichen. Dazu setzen wir die folgende Grammatik voraus:

```
MoveStatement
  = "MOVE" IDENTIFIER "TO" IDENTIFIER {[","] IDENTIFIER} .
```

`IDENTIFIER` sei ein Terminalsymbol, das wir hier nicht näher spezifizieren. Beim nachfolgenden kurzen Text handle es sich um ein syntaktisch korrektes “`MoveStatement`”.

```
MOVE a TO b, c
```

Wir erhalten daraus die folgende interne Darstellung. Schlüsselwörter werden mit dem leeren String (`""`) identifiziert:

```
TREE[
  NONTERMINAL(
    "MoveStatement",
    TREE[
      TERMINAL("","MOVE",{FROM=(1,1,1),TO=(5,1,5)})
      TERMINAL("IDENTIFIER","a",{FROM=(6,1,6),TO=(7,1,7)})
      TERMINAL("","TO",{FROM=(8,1,8),TO=(10,1,10)})
      TERMINAL("IDENTIFIER","b",{FROM=(11,1,11),TO=(12,1,12)})
      TERMINAL("",",",{FROM=(12,1,12),TO=(13,1,13)})
      TERMINAL("IDENTIFIER","c",{FROM=(14,1,14),TO=(15,1,15)})
    ],
    {FROM=(1,1,1),TO=(15,1,15)}
  )
]
```

Schon bei diesem kleinen Beispiel wird klar, warum man diese Darstellung als interne Repräsentation bezeichnet. In Wirklichkeit hat man es nicht mit einzeiligen, sondern mit tausendzeiligen Programmen zu tun. Der erzeugte Parse-Baum wird riesig. Interne Darstellungen sind für einen menschlichen Betrachter schwer verständlich. Wir geben den Parse-Baum deshalb in einer besser lesbaren Form wieder:

```
+-[MoveStatement]  (1,1,1)-(15,1,15)
   |
   +-("MOVE")            (1,1,1)-(5,1,5)
   |
   +-<'a' : IDENTIFIER>  (6,1,6)-(7,1,7)
   |
   +-("TO")              (8,1,8)-(10,1,10)
   |
   +-<'b' : IDENTIFIER>  (11,1,11)-(12,1,12)
   |
   +-(",")               (12,1,12)-(13,1,13)
   |
   +-<'c' : IDENTIFIER>  (14,1,14)-(15,1,15)
```

Aus diesem Beispiel wird klar, wie Grammatik und Parse-Baum zusammenhängen. Die Grammatik beschreibt exakt die Struktur aller möglichen Parse-Bäume. Der Parse-Baum eines Programms ist eine konkrete Instanz. Bei der Entwicklung von Wartungswerkzeugen kann man sich mit Hilfe von CoP auf die gut lesbare Grammatik abstützen und braucht die Strukturinformationen nicht mühsam aus der internen Darstellung zusammenzusuchen.

4.4 Grammatik-Spezifikationen für CoP

Um eine internen Repräsentation für ein Programm zu erhalten, benötigen wir eine Grammatik der entsprechenden Programmiersprache. Wie man eine Grammatik formuliert, so dass sie von CoP akzeptiert wird, werden wir in diesem Abschnitt ausführlich beschreiben. Wie bereits erwähnt, kann CoP sowohl mit BNF- als auch mit EBNF-Spezifikationen umgehen. Grammatik-Spezifikationen für CoP besitzen aber einige Erweiterungen, die dem Entwickler erlauben, den Prozess des Parsens zu steuern und den Aufbau der internen Struktur zu beeinflussen.

Die Grammatik, welche CoP als Grundlage dient, um einen Parser zu generieren, ist selbst eine formale Sprache und lässt sich in EBNF notieren. Wir werden diese Grammatik schrittweise vorstellen und deren Semantik an Hand von Beispielen erläutern. Ein durch CoP erzeugter Parser hat einen Namen (`IDENTIFIER`) und besteht aus vier Komponenten:

```
Parser
  = "PARSER" IDENTIFIER
    LayoutDescription
    CharacterClassDescription
    LexicalSyntaxDescription
    ContextFreeSyntaxDescription
    "END" .
```

Im Abschnitt “Funktionale Parsing-Techniken” ab Seite 17 haben wir der Einfachheit halber vorausgesetzt, dass ein Parser ausschliesslich auf der (E)BNF-Grammatik aufbaut. Das ist in der Praxis eine unrealistische Annahme. In Wirklichkeit sind Parser aus Effizienzgründen mehrstufig gegliedert. CoP basiert auf den vier Stufen: Layout (`LayoutDescription`), Zeichenklassen (`CharacterClassDescription`), Scanner (`LexicalSyntaxDescription`) und Parser (`ContextFreeSyntaxDescription`).

4.4.1 Layout

Unter *Layout* fallen diejenigen Symbole (Zeichen), welche eigentlich nicht direkt zur formalen Sprache gehören, aber für die textuelle Gestaltung von Programmen von grosser Bedeutung sind. Dazu gehören in der Regel alle Leerzeichen, aber auch Programm-Kommentare (zur Zeit in CoP noch nicht implementiert).

```
LayoutDescription = "LAYOUT" [CharacterSet] "END" .
```

In CoP kann man derzeit nur diejenigen Symbole als Layout spezifizieren, welche als Leerzeichen zu interpretieren sind. Wir definieren zu diesem Zweck Produktionen, um Zeichenmengen zu beschreiben:

```
CharacterSet = CharacterSubrange {"|" CharacterSubrange} .

CharacterSubrange
  = IDENTIFIER | CharacterDefinition [".." CharacterDefinition] .

CharacterDefinition = CHARACTER | NUMBER .
```

Wir können die nicht druckbaren Symbole des ASCII-Alphabets folgendermassen spezifizieren:

```
LAYOUT
  0 .. ' ' | 127 .. 255
END
```

Der durch CoP erzeugte Parser wird sämtliche Layout-Symbole überspringen. Obwohl Layout-Symbole im Parse-Baum nicht erscheinen, spielen sie oft eine wichtige Rolle bei der Steuerung des Parsing-Vorgangs, wie wir an früherer Stelle bereits bemerkt haben (siehe Seite 14).

4.4.2 Zeichenklassen

Formale Sprachen basieren theoretisch mindestens auf einem Alphabet, in der Praxis sogar auf einer Hierarchie von Alphabeten. Für CoP sind die ASCII-Zeichen das grundlegende Alphabet. Über *Zeichenklassen* können eine ganze Anzahl von Teilmengen aus diesem Alphabet gebildet werden, die sich auch überlappen dürfen:

```
CharacterClassDescription
  = "CHARACTER" "CLASSES"
    {IDENTIFIER "=" CharacterSet "."}
    "END" .
```

Häufige Zeichenklassen, welche fast in allen Programmiersprachen vorkommen, sind z.B. `Digit` und `Letter`. Das nachfolgende Beispiel gibt eine kleine Auswahl möglicher Definitionen:

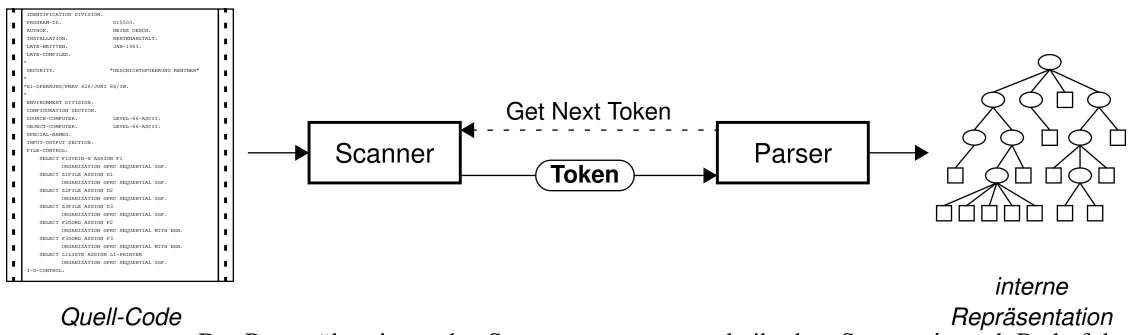
```
CHARACTER CLASSES
  Digit    = '0' .. '9' .
  Letter   = 'A' .. 'Z' | 'a' .. 'z' .
  HexDigit = Digit | 'A' .. 'F' .
  NewLine  = 10 .
END
```

Es ist erlaubt, einzelne Zeichen entweder über deren dezimalen ASCII-Code oder über deren ausdruckbaren Wert zu identifizieren. 65 und `'A'` sind z.B. identisch. Weil es sich bei den ASCII-Zeichen um eine geordnete Menge handelt, können wir zusammenhängende Teilmengen in abgekürzter Form schreiben (`X .. Y`). Die Verwendung von Zeichenklassen macht die Definition des Scanners übersichtlicher.

4.4.3 Scanner

Der *Scanner* ist die Vorstufe des Parsers. Er liest direkt die Eingabe und fasst Zeichen aus dem grundlegenden Alphabet zu Tokens zusammen. Tokens sind z.B. Schlüsselwörter, Identifiers, Integers usw. Tokens bestehen meist aus einer Sequenz von Symbolen des grundlegenden Alphabets und bilden selbst gewissermassen ein Alphabet auf höherer Ebene, nämlich das Alphabet, auf dem der Parsers aufbaut. Die Scanning-Phase wird aus Effizienzgründen eingeführt. Die Beschreibung von Tokens basiert in der Regel auf regulären Ausdrücken, welche sich in sehr effiziente Erkennungsalgorithmen transformieren lassen. Abbildung 8 zeigt die konventionelle Kooperation zwischen Scanner und Parser (Aho, Sethi et al. 1986).

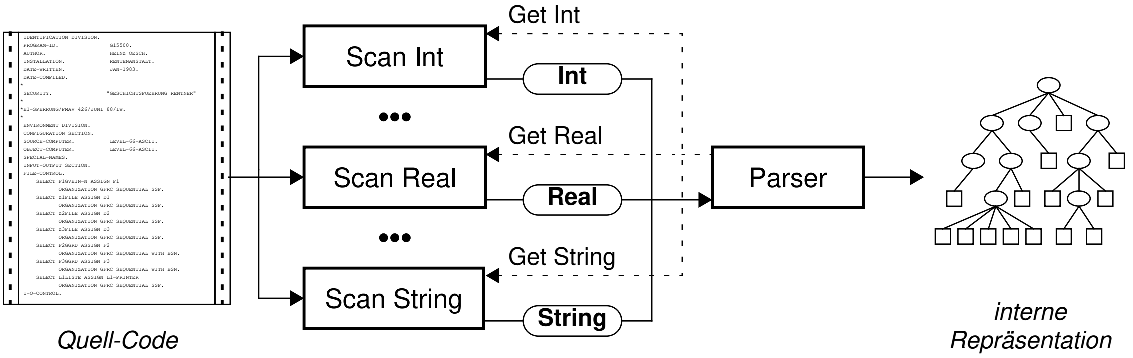
ABBILDUNG 8. KONVENTIONELLER SCANNER



Der Parser übernimmt den Steuerungsprozess und gibt dem Scanner je nach Bedarf den Auftrag, das nächste Token aus der Eingabe zu lesen. Auf Grund seiner Spezifikation, die übrigens auch in einer formalen Sprache geschrieben ist, entscheidet der Scanner selbständig, welches Token als nächstes in der Eingabe anliegt und gibt dieses zusammen mit dem Typ des Tokens an den Parser weiter, z.B. ("abc", IDENTIFIER). Dieses Verfahren hat den gleichen Effekt, wie wenn der Scanner im voraus eine Token-Liste generieren würde. Das ist dann sinnvoll, wenn der Scanner keine Kontextinformationen benötigt, sondern allein auf Grund seiner Spezifikation eindeutig entscheiden kann, welches Token als nächstes in der Eingabe steht.

Ein COBOL-Scanner ist oft nicht in der Lage, ohne Kontextinformationen selbständig zu entscheiden, zu welcher Klasse das nächste Token in der Eingabe gehört (siehe Seite 14). Die Verwendung von Kontextinformationen widerspricht aber der Idee eines konventionellen Scanners. Anstatt einen Scanner zu verwenden, der im voraus eine Token-Liste generiert, wählten wir für CoP den Ansatz, den Scanner vollständig in den Parser zu integrieren. Der Parser verfügt über die nötigen Kontextinformation und kann deshalb den Scanner unterstützen. Genaugenommen hat man bei CoP nicht mehr einen einzigen Scanner, der in der Lage ist, alle möglichen Tokens zu erkennt, sondern für jedes Token einen eigenen unabhängigen Scanner. Aus dieser Menge von Scannern wählt der Parser an jeder betreffenden Stelle des Parsing-Vorgangs den richtigen aus (siehe Abbildung 9).

ABBILDUNG 9. SCANNER VON COP:



Die Token-Beschreibung, oder in anderen Worten die lexikalische Syntax, wird in CoP durch drei Komponenten repräsentiert:

```
LexicalSyntaxDescription
  = "LEXICAL" "SYNTAX"
    "TOKENSETS"   {TokenSetDescription} "END"
    "AUXILIARIES" {AuxiliaryProduction} "END"
    "TOKENS"      {ScannerProduction}   "END"
    "END" .
```

CoP erlaubt die Definition von Token-Mengen (`TokenSetDescription`) in aufzählender Form:

```
TokenSetDescription = IDENTIFIER "=" STRING {"|" STRING} "." .
```

Token-Mengen können z.B. verwendet werden, um Schlüsselwortlisten zu bilden. Wozu das gut ist, werden wir bald sehen.

```
TOKENSETS
  KeyWord = "BEGIN" | "GOTO" | "END" .
END
```

Auxiliaries (`AuxiliaryProduction`) sind Hilfsdefinitionen, die nicht selbst als Token erscheinen, aber zur Definition von Tokens verwendet werden können.

```
AuxiliaryProduction = IDENTIFIER "=" Expression "." .
```

Eine solche Hilfsdefinition könnte z.B. folgendermassen aussehen:

```
AUXILIARIES
  Number = Digit {Digit} .
  Sign   = '+' | '-' .
END
```

Die Definition `Number` kann jetzt in einer Token-Definition beliebig oft verwendet werden. Tokens sind die Terminalsymbole des Parsers und deshalb besonders sorgfältig auszuwählen. Man spezifiziert sie gemäss folgender Produktion:

```
ScannerProduction
  = IDENTIFIER "=" Expression [";" "EXCLUDE" IDENTIFIER] "." .
```

Wir können nun Tokens, das grundlegende Parser-Alphabet bilden:

```
TOKENS
  INTEGER    = [Sign] Number .
  FIXREAL    = [Sign] [Number] '.' Number .
  IDENTIFIER = Letter {Letter | Digit} ; EXCLUDE KeyWords .
END
```

Sowohl `Sign` als auch `Number` sind keine Tokens und deshalb für den Parser nicht sichtbar. Sie erleichtern hier die Definition von `INTEGER` und `FIXREAL` und gewährleisten einen besseren Überblick. Wir sehen hier auch, wozu Token-Mengen verwendet werden. Oft möchte man bei Identifiers nicht alle durch die Syntaxregel spezifizierten Wörter zulassen, sondern eine gewisse Teilmenge, z.B. alle Schlüsselwörter ausschliessen. In CoP kann man dies mit Hilfe der `EXCLUDE`-Klausel tun.

Sowohl die Definitionen von Auxiliaries als auch diejenige von Tokens basieren auf regulären Ausdrücken (`Expression`). Es gibt spezielle Notationen, um reguläre Ausdrücke zu beschreiben. In CoP verwenden wir aus Konsistenzgründen die (E)BNF, also die gleiche Notation wie für die Parser-Spezifikation. Das ist zulässig, wenn wir bei der Definition der lexikalischen Syntax die Verwendung rekursiver Produktionen verbieten. Wir führen an dieser Stelle die Produktionen für die Spezifikation von (E)BNF-Ausrükken (`Expression`) ein.

```
Expression = Term {"|" Term} .

Term = Factor {Factor} .

Factor
  = "EPSILON"
  | LexicalTerminal
  | ContextFreeTerminal
  | NonTerminal
  | "(" Expression ")"
  | "[" Expression "]"
  | "[|" Expression "|]"
  | "{" Expression "}"
  | "{|" Expression "|}" .

LexicalTerminal = CharacterDefinition .

ContextFreeTerminal = STRING .

NonTerminal = IDENTIFIER .
```

Wir stellen fest, dass die obige Definition in etwa der Definition der (E)BNF-Notation entspricht (siehe Seite 11). Weil wir für den Scanner und den Parser dieselben Ausdrücke verwenden, gibt es gewisse Abweichungen. So existieren zwei verschiedene Klassen von Terminalsymbolen. `LexicalTerminal` darf nur für den Scanner und `ContextFreeTerminal` nur für den Parser verwendet werden. Für die Option “`[|` ... `|]`” und die Repetition “`{|` ... `|}`” gibt es zusätzliche Operatoren, die ausschliesslich für Parser-Spezifikationen vorgesehen sind. Wir werden darauf im nächsten Abschnitt zurückkommen. Da CoP sowohl die BNF als auch die EBNF unterstützt, sind die Möglichkeiten beider Notationen in den obigen Produktion vereinigt.

Bemerkungen zur Implementierung: Scanner sind einfache Parser für reguläre Ausdrücke. Die Parser-Combinators, die wir im Abschnitt “Funktionale Parsing-Techniken” ab Seite 17 behandelt haben, sind deshalb auch für die Implementierung von Scannern anwendbar. Dieser Ansatz ist zwar elegant, aber für Scanner nicht sehr effizient. Bekanntlich lassen sich reguläre Ausdrücke durch deterministische endliche Automaten (DEA) erkennen. Solche Automaten können einfach implementiert werden und führen zu effizienten Scannern. Es ist geplant, den Scanner-Generator von CoP so umzuschreiben, dass er anstatt Combinators einen DEA verwendet.

4.4.4 Parser

Den Aufbau des *Parsers* haben wir im Abschnitt “Funktionale Parsing-Techniken” ab Seite 17 ausführlich diskutiert. Wir können uns deshalb hier auf das wesentliche beschränken. In CoP wird ein Parser gemäss den untenstehenden Produktionen implementiert. Dabei dürfen die in der Scanner-Spezifikation definierten Tokens als Terminalsymbole verwendet werden.

```
ContextFreeSyntaxDescription
  = "CONTEXTFREE" "SYNTAX"
    "START" "AT" IDENTIFIER
    {ContextFreeProduction}
    "END" .

ContextFreeProduction
  = IDENTIFIER ["HIDE" | "RENAME" "AS" IDENTIFIER | "OMIT"]
    "=" Expression "." .
```

Die Reihenfolge, in der wir die Produktionen aufschreiben, ist irrelevant. Wir müssen aber CoP mitteilen, welche Produktion das Startsymbol repräsentiert.

Wir geben nachfolgend einen kleinen Überblick über die (E)BNF-Erweiterungen, die dem Entwickler erlauben, den Prozess des Parsens zu steuern und den Aufbau der internen Struktur zu beeinflussen. Wir machen aber darauf aufmerksam, dass die Verwendung dieser Erweiterungen nicht zu mächtigeren Parsern führen.

4.4.4.1 Steuerung des Parsing-Vorgangs

Wir haben bereits erwähnt, dass CoP einen Parser generiert, der bei mehrdeutigen Grammatiken die erste Lösung findet. Welche Lösung zuerst gefunden werden soll, bestimmt der Entwickler einer Grammatik selbst. Es existieren drei Parser-Combinators, bei denen sich der Kontrollfluss des Parsers in zwei Äste verzweigt, nämlich die Alternative, die Option, und die Repetition. Bei diesen drei Combinators spielt es eine Rolle, welcher der beiden Äste zuerst beschritten wird. CoP bietet dem Grammatik-Entwickler zusätzliche Operatoren. Es gelten folgende Regeln:

1. Alternative:
   - `P = Q | R .` $\Rightarrow$ Parse `Q`, falls ein Syntaxfehler auftritt und nur dann parse `R`
2. Option:
   - `P = [ Q ] .` $\Rightarrow$ `P = Q |` ε `.`
   - `P = [| Q |] .` $\Rightarrow$ `P =` ε `| Q .`
3. Repetition:
   - `P = { Q } .` $\Rightarrow$ `P = (Q P) |` ε `.`

- `P = {| Q |] .` $\Rightarrow$ `P = ε | (Q P) .`

Für die COBOL-74-Grammatik von AEMES haben diese Regeln wichtige Konsequenzen. Wir zeigen dies am Beispiel der IF-Anweisung, die wie bei vielen anderen Programmiersprachen mehrdeutig ist. Die Frage bei verschachtelten IF-THEN-ELSE-Konstrukten lautet bei solchen Programmiersprachen: Zu welchem IF gehört das ELSE?

Interpretation A:	Interpretation B:

<pre>IF (...)
   IF (...)
      ...
   ELSE
      ...</pre>
	<pre>IF (...)
	   IF (...)
	      ...
	ELSE
	   ...</pre>

Interpretation A:

```
IfStatement
  = "IF" ConditionalExpression StatementSequence
    [ "ELSE" StatementSequence ] .
```

Interpretation B:

```
IfStatement
  = "IF" ConditionalExpression StatementSequence
    [| "ELSE" StatementSequence |] .
```

Per Konvention gilt bei den meisten Programmiersprachen Interpretation A, so auch bei COBOL-74. Es ist sehr wichtig und nicht immer ganz einfach, der beabsichtigten Lösung den Vorrang zu geben.

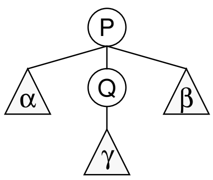
4.4.4.2 Steuerung der Parse-Baum-Konstruktion

Die Produktionen einer Grammatik beschreiben die Struktur aller möglichen Parse-Bäume. CoP bietet Möglichkeiten, um von dieser vorgegebenen Struktur während des Parsing-Vorgangs abzuweichen. Der Parsing-Vorgang wird dabei nicht beeinflusst, sondern nur die erzeugte interne Darstellung. Zur Zeit stehen die drei Modifikationsfunktionen `HIDE`, `RENAME` und `OMIT` zur Verfügung. Wir beschreiben deren Bedeutung an Hand von Beispielen.

Normalerweise werden Produktionen der Form:

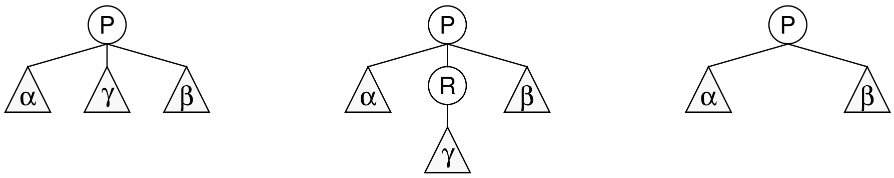
```
P = α Q β .
Q = γ.
```

in einen Parse-Baum mit folgender Struktur übersetzt:



Diesen Sachverhalt kann man mit Hilfe der Parse-Baum-Modifikationsfunktionen von CoP ändern:

```
P = α Q β .
   Q HIDE = γ.          Q RENAME AS R = γ.          Q OMIT = γ.
```



Wann diese Modifikationsfunktionen eingesetzt werden, bleibt dem Entwickler einer Grammatik überlassen. Für `HIDE` geben wir im nachfolgenden Abschnitt ein Anwendungsbeispiel.

Bemerkungen zur Implementierung: Der Parser-Generator CoP besteht aus einem dynamischen und einem statischen Teil. Der dynamische Teil ist für die Erzeugung von Parsern zuständig. Er verfügt selbst über einen eingebauten Scanner und Parser. Diese bereiten eine gegebene Grammatik so auf, dass sie vom Generator in den spezifizierten Parser umgesetzt werden kann. Der statische Teil besteht aus einer Sammlung von generischen Modulen, die völlig unverändert für sämtliche generierten Parser eingesetzt werden. Dazu gehören in erster Linie die Scanner- und Parser-Combinators, aber auch Funktionen zum Lesen und Schreiben von Dateien, zur Manipulation von Zeichenketten und zum Aufbau und zur Speicherung des Parse-Baums.

4.4.5 Beispiel

In diesem Abschnitt wollen wir an Hand eines kleinen Beispiels zeigen, wie eine Grammatik für CoP aussieht. Zu diesem Zweck definieren wir eine kleine formale Sprache in EBNF, welche arithmetische Ausdrücke beschreibt.

```
PARSER AExpEBNF
  LAYOUT
    0 .. ' ' | 127 .. 255
  END

  CHARACTER CLASSES
    Digit = '0' .. '9' .
  END

  LEXICAL SYNTAX
    TOKENSETS
    END

    AUXILIARIES
    END

    TOKENS
      NUMBER = Digit {Digit} .
    END
  END

  CONTEXTFREE SYNTAX
    START AT Exp

    Exp    = Term {("+" | "-") Term} .
    Term   = Factor {("*" | "/") Factor} .
```

```
    Factor = "-" Exp | "(" Exp ")" | NUMBER .
  END
END
```

Grammatik-Kategorien wie TOKENSETS und AUXILIARIES, die man nicht benötigt, müssen trotzdem notiert werden. Man lässt ihren Beschreibungsteil einfach leer. Der Parser AExpEBNF kann Ausdrücke der Form “1+2-3*4/5” parsen und in eine entsprechende interne Darstellung abbilden.

Wir haben an früherer Stelle erwähnt (siehe Seite 11), dass es mit Hilfe von CoP möglich ist, eine EBNF-Grammatik in eine BNF-Grammatik zu transformieren, ohne dass sich die Struktur der internen Darstellung ändert. Wir schreiben die Grammatik für AExpEBNF gemäss dem Algorithmus auf Seite 11 um und ergänzen die Grammatik mit der von CoP zur Verfügung gestellten Hilfsfunktion HIDE zur Modifiktion des Parse-Baums:

```
PARSER AExpBNF
  LAYOUT
    0 .. ' ' | 127 .. 255
  END

  CHARACTER CLASSES
    Digit = '0' .. '9' .
  END

  LEXICAL SYNTAX
    TOKENSETS
    END

    AUXILIARIES
    END

    TOKENS
      NUMBER = Digit {Digit} .
    END
  END

  CONTEXTFREE SYNTAX
    START AT Exp

    Exp         = Term Exp' .
    Exp'   HIDE = Op1 Term Exp' | EPSILON .
    Op1    HIDE = "+" | "-" .
    Term        = Factor Term' .
    Term'  HIDE = Op2 Factor Term' | EPSILON .
    Op2    HIDE = "*" | "/" .
    Factor      = "-" Exp | "(" Exp ")" | NUMBER .
  END
END
```

Als erstes stellt man fest, dass die BNF-Beschreibung der kontextfreien Syntax mehr als doppelt so lang wird, wie diejenige in EBNF. Trotzdem beschreiben die Grammatiken AExpEBNF und AExpBNF exakt dieselben Sprachen. Auf den ersten Blick nicht so klar ist die Tatsache, dass die beiden Parser AExpEBNF und AExpBNF für identische Eingaben auch genau dieselben internen Darstellungen erzeugen. Mit der Funktion HIDE können die neu erzeugten Produktionen gewissermassen zum Verschwinden gebracht werden, so dass sich keinen Eintrag im Parse-Baum erzeugen.

5.0 Stand und Ausblick

Wir haben in diesem Bericht motiviert, warum eine erweiterbare Wartungsumgebung wie AEMES über ein syntaktisches Rahmenwerk verfügen muss. Es hat sich dabei herausgestellt, dass ein Parser-Generator eine wichtige Komponente dieses Rahmenwerks ist. Die speziellen Randbedingung, die durch COBOL gegeben sind, verlangen nach unkonventionellen, aber dennoch sauberen Lösungen. Basierend auf funktionalen Konzepten haben wir gezeigt, wie man einen Generator implementiert, der effiziente (COBOL)-Parser liefert, die auch mit Programmen aus der Praxis fertig werden. Die generierten Parser sind für Anwendungen wie Programm-Analysen massgeschneidert und für die Verwendung in Compilern nicht geeignet. Sie haben genau einen Zweck, nämlich ein Quellprogramm in einen Parse-Baum zu transformieren. Diese spezifische Aufgabenstellung ermöglicht eine vollständige Automatisierung des Transformationsprozesses. Der Entwickler einer Grammatik kann sich primär auf die syntaktischen Strukturen konzentrieren. Er hat aber die Möglichkeit, bis zu einem gewissen Grad den Ablauf des Parsens zu steuern und den Aufbau des Parse-Baums zu beeinflussen, falls er dies wünscht. Die Parser-Spezifikationssprache von CoP wurde so gestaltet, dass sie leicht lesbar ist, denn sie dient gleichzeitig als formale Spezifikation der internen Darstellung und ist somit eine wichtige Grundlage bei der Entwicklung von Wartungswerkzeugen.

Der Parser-Generator CoP und die COBOL-74-Grammatik von AEMES werden weiterentwickelt. Aus den bisherigen Resultaten lassen sich eine ganze Anzahl von Aufgabenstellungen ableiten. Als nächstes soll die Data-Division geparst werden können. Das hat nicht nur Erweiterungen an der Grammatik, sondern auch am Parser-Generator zur Konsequenz. Programm-Kommentare müssen zukünftig erkannt und in den Parse-Baum integriert werden. Bei Programmänderungen wäre es von Vorteil, wenn nur die geänderten Stellen neu geparst werden müssten. Es ist abzuklären, wie die von CoP generierten Parser zu erweitern sind, dass sie inkrementell eingesetzt werden können.

Ein Parser-Generator ist eine notwendige, aber noch keine hinreichende Voraussetzung für eine erweiterbare Wartungsumgebung. Wenn sämtliche Werkzeuge und Dienste direkt auf der internen Darstellung operieren, dann ist die Erweiterbarkeit nicht mehr gewährleistet. Es ist deshalb abzuklären, wie eine einheitliche Abfrage- und Manipulations-Schnittstelle zu implementieren ist, die eine weitgehende Entkopplung von Werkzeugen und interner Darstellung ermöglicht.