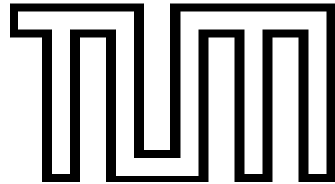


Technische
Universität
München

Fakultät für Informatik

Fortgeschrittenen-Praktikum

Browser für GOS

Christian Lesny
Herbststr. 21
84030 Ergolding
(lesny@informatik.tu-muenchen.de)

Aufgabensteller: Prof. Dr. Manfred Broy
Betreuer: Dr. Bernhard Rumpe

Inhaltsverzeichnis

1. Einführung und Motivation ... 5

1.1 Wozu Browser? ... 5

1.2 Zielsetzung ... 6

1.3 Aufbau der Ausarbeitung ... 7

1.4 Notation und Glosar ... 8

2. Programmiersprache GOS ... 10

2.1 Die funktionale Programmiersprache Gofer ... 10
2.1.1 Mathematische Funktionen und funktionale Sprache ... 10
2.1.2 Datentypen ... 11
2.1.3 Funktionsvereinbarung und -anwendung ... 12
2.1.4 Funktionen höherer Ordnung ... 13
2.1.5 Zusammenfassung ... 14

2.2 Objektorientierte Programmierung mit GOS ... 14
2.2.1 Objekte ... 14
2.2.2 Klassen ... 16
2.2.3 Vererbung ... 18
2.2.4 Polymorphismus, Overloading und späte Bindung ... 21
2.2.5 Zusammenfassung ... 22

3. GOS-Browser ... 24

3.1 Eingesetzte Programme ... 24

3.2 Allgemeine Beschreibung ... 25

3.3 Anwendung und Funktionsbeschreibung ... 27
3.3.1 Programmaufruf ... 27
3.3.2 Auswertungsbefehle ... 27

3.4 Anwendungsbeispiel ... 30

4. GOS-Grammatik 32

4.1 Beschreibung der Grammatik 32
4.1.1 Gofer-Grammatik 32
4.1.2 OO-Teil 35
4.1.3 Zusammenfassung 38

4.2 Notation 38

4.3 Definition der GOS-Grammatik 40

4.4 Layout 46
4.4.1 Kommentare 46
4.4.2 Layout-Rule 47

5. Implementierung 51

5.1 Aufbau und Funktionsweise des GOS-Browsers 51
5.1.1 Ablaufsteuerung 52
5.1.2 ParserManager-Komponente 52
5.1.3 Parser-Komponente 53
5.1.4 Scanner-Komponente 54
5.1.5 FileReader-Komponente 55
5.1.6 Abstrakter Syntaxbaum 55
5.1.7 Zusammenfassung 56

5.2 Programmstruktur 56
5.2.1 Quelldateien und Programmaufbau 57
5.2.2 Erstellungsprozeß 58

5.3 Parse-Framework 59
5.3.1 Allgemeines 59
5.3.2 Gofer-Datenstrukturen für den abstrakten Syntaxbaum 59
5.3.3 Klassenhierarchie zum abstrakten Syntaxbaum 68
5.3.4 Die Klasse FileReader 76
5.3.5 Die Scanner-Klasse 76
5.3.6 Die Parser-Klasse 78
5.3.7 Die Klasse ParserManager 78

5.4 Layout 80
5.4.1 Kommentare 80
5.4.2 Layout-Rule 80

5.5 Auswertungen 82
5.5.1 Arbeitsweise 82
5.5.2 Beispielauswertung 83

6. Zusammenfassung und Ausblick 84

A Technische Daten und Bedienhinweise **86**

B GOS-Browser Referenz **88**

C Gofer-Grammatik **90**

D Beschreibung Gofer-Layout **97**

E Anwendungsbeispiel **103**

Literaturverzeichnis **109**

Kapitel 1

Einführung und Motivation

Die Produktion großer Programme stellt hohe geistige Ansprüche an seine Entwickler. Besonders in der Phase der Implementierung werden die Grenzen menschlichen Denkens allzuoft erreicht. Deshalb müssen immer wieder Mittel und Wege gefunden werden, dieses Problem in den Griff zu bekommen. In der Software-Entwicklung ist der Rechner das wichtigste Arbeitsgerät. Mit seiner Hilfe wird der Entwickler mit Werkzeugen und Programmen unterstützt, ohne die eine befriedigende Arbeitsweise nicht möglich wäre. Die Wahl der Werkzeuge ist natürlich nicht nur von der Problemstellung, sondern auch von der gewählten Problemlösungsmethode, insbesondere von der oder den verwendeten Programmiersprachen abhängig. So sind auch für die Programmiersprache GOS spezielle Werkzeuge und Hilfsprogramme denkbar, die den Entwicklungsprozeß mit dieser Sprache unterstützen. Im Rahmen dieses Fortgeschrittenen Praktikums soll ein solches Werkzeug für die Programmiersprache GOS entwickelt werden.

1.1 Wozu Browser?

Wie im täglichen Leben auch, helfen uns Werkzeuge bei der Bewältigung unserer Aufgaben. Auch die Arbeit mit Rechnersystemen erfordert eine große Anzahl solcher Werkzeuge oder Hilfsprogramme. Sie ermöglichen es uns zum Beispiel, Systeminformationen auszugeben oder auch Dateien zu suchen, anzusehen und zu verändern. Sie helfen uns unter anderem, Strukturen vorgegebener Daten aufzudecken und konkrete Informationen zu extrahieren, abstrahierende Übersichten zu erzeugen, Informationen zu suchen und zu lokalisieren, Daten zu vergleichen oder zu sortieren. Aus dieser Fülle von Möglichkeiten zeichnet sich eine spezielle Klasse von Werkzeugen aus - die sogenannten Browser. Es ist zwar nicht notwendig, diesen Begriff exakt zu definieren. Allerdings ist es hilfreich, die Aufgaben von Browsern im Bereich der Programmentwicklung zu skizzieren.

Zu den wichtigsten Methoden der Programmentwicklung zählen Abstraktion und Strukturierung, denn sie tragen dazu bei, die Komplexität einer gestellten Aufgabe zu bewältigen. Dieser Prozeß soll durch Browser unterstützt werden. Mit Hilfe eines Browser wird die flache und sequenzielle Anordnung von Quelldateien aufgebrochen und die dahinterliegenden Strukturen aufgedeckt. Die Möglichkeit, Quellcode aus den unterschiedlichsten Blickwinkeln und Abstraktionsebenen zu betrachten, gehört also zu den wichtigsten Aufgaben eines Browsers. Die Notwendigkeit hierfür wird durch die Tatsache erhärtet, daß der Quellcode im Allgemeinen öfter gelesen als geschrieben wird.

Es gibt viele Möglichkeiten, mit der Browser ihre Aufgaben verrichten. Die Palette reicht von text-orientierten Browsern bis hin zu solchen, die eine graphische Benutzeroberfläche nutzen. Gerade die

Nutzung letzterer ermöglicht die Realisierung hervorragender Browser. In diesem Zusammenhang wird auch oft der Begriff „Visual Programming“ verwendet. Es gibt aber auch prominente Kandidaten, die den Begriff des Browsers geprägt haben. Beispielsweise wäre eine Entwicklung mit Smalltalk[1] ohne Browser undenkbar. So verfügt die Smalltalk-Entwicklungsumgebung über einen Class-Hierarchy-Browser, verschiedene Messages-Browser und zahlreiche Inspectors. Der Class-Hierarchy-Browser zum Beispiel stellt die Klassenhierarchie dar und ermöglicht die Einsicht in einzelne Klassen und deren Methoden. An anderes Beispiel ist der Inspector, mit desen Hilfe die Struktur eines Objektes aufgedeckt werden kann. Darüber hinaus gibt es weitere Browser, die andere Einsichten des Codes und seiner Struktur gestatten. Browser stellen also ein mächtiges und unerläßliches Werkzeug in der Programmentwicklung dar.

1.2 Zielsetzung

Ziel dieses Fortgeschrittenen-Praktikums ist die Implementierung eines Browsers für GOS. Mit seiner Hilfe sollen Auswertungen über GOS-Quellcode durchgeführt und somit Informationen zum Beispiel über Klassen oder Methoden extrahiert werden können. Als Programmiersprache wird GOS selbst verwendet. Damit können die Entwicklungswerkzeuge Glex und GBison verwendet werden, da diese GOS-Code generieren. Zur Realisierung des GOS-Browsers müssen die folgenden Schwerpunkte berücksichtigt werden:

- **Parser**
  Für die Auswertungen ist es notwendig, den GOS-Quellcode zu parsen. Die Erstellung des Parsers einschließlich des Scanners für die lexikalische Analyse wird mit den Werkzeugen GBison und GLex vorgenommen. Zur Darstellung des abstrakten Syntaxbaumes müssen geeignete GOS- und Gofer-Datenstrukturen gefunden werden.
- **Fehlerbehandlung**
  Eine wichtige Prämisse ist, daß der zu analysierende GOS-Code fehlerfrei ist. Das bedeutet, daß der Parser keine Fehlerbehandlung beinhaltet. Eventuelle syntaktische Fehler im Code werden somit den Parse-Vorgang ohne besondere Fehlermeldung abbrechen.
- **Weiterverarbeitung**
  Eine weitere Analyse des abstrakten Syntaxbaumes soll jedoch (wegen der oben erwähnten Prämisse) nicht vorgenommen werden, so daß zum Beispiel nicht geprüft wird, ob verwendete Klassennamen oder Methoden auch tatsächlich deklariert sind. Zudem werden auch keine weiterführenden Verarbeitungen des abstrakten Syntaxbaumes durchgeführt, so daß zum Beispiel Gofer-Ausdrücke nicht gemäß ihren Prioritäten strukturiert werden, sondern die Struktur des gelesenen Codes wiedergeben.
- **Auswertungen**
  Die Auswertungen werden mit Hilfe des abstrakten Syntaxbaumes durchführt. Es müssen geeignete Algorithmen zu dessen Analyse entwickelt werden.
- **Layout-Rule**
  In Bezug auf die Gofer-Grammatik ist die Anwendung der sogenannten Layout-Rule zu erwähnen.

[1]Siehe A. Goldberg, Smalltalk-80, Addison-Wesley

Diese Regel erlaubt es, Strukturierungszeichen (Semikola und geschweifte Klammern) im Code wegzulassen, falls gewisse Strukturierungsregeln eingehalten werden. Für die Realisierung der Layout-Rule müssen geeignete Algorithmen und Methoden implementiert werden.

- **Textbasiert**
  Der GOS-Browser soll vollständig textbasiert sein, d.h. Quelldateien und Ergebnisse werden in ASCII-Form verarbeitet bzw. ausgegeben.

Unabhängig von dieser Zielsetzung lassen sich weitere Aspekte dieser Arbeit finden:

- Der Parser des GOS-Browsers stellt eine Teilkomponente für einen zukünftigen GOS-Compiler dar. Da Parser- und Scanner-Komponenten sowie die Datenstrukturen für den abstrakten Syntaxbaum auch für einen Compiler benötigt werden, können diese als Basis verwendet und erweitert werden.
- Die Implementierung stellt einen weiteren Test für den GOS-Interpreter dar. Das Programm des GOS-Browsers ist hinreichend groß und komplex, um den Interpreter mit diesem Programm gut austesten zu können.
- Der Parser stellt einen weiteren Test für die Werkzeuge GBison und GLex dar, da sowohl die GOS-Grammatik und die Regeln für die lexikalische Analyse als auch die zusätzliche Funktionalität (beispielsweise die Realisierung der Layout-Rule) sehr umfangreich ist.
- Die Verwendung von GOS als Programmiersprache und als Gegenstand des Parsers unterstützt eine kritische Diskussion der Programmiersprachen GOS und Gofer.

1.3 Aufbau der Ausarbeitung

In diesem Abschnitt wird der Aufbau der vorliegenden Arbeit kurz erläutert.

♦ **Programmiersprache GOS**
Im Kapitel 2 werden zuerst die Grundlagen der funktionalen Programmierung und die Programmiersprache Gofer in groben Zügen umrissen. Anschließend werden die Grundlagen der objektorientierten Programmierung und die Sprache GOS dargestellt. Hierbei wird nur insoweit auf die Konzepte eingegangen, als daß sie für das Verständnis der Programmiersprache GOS notwendig sind.

♦ **GOS-Browser**
Das Kapitel 3 wendet sich an die Anwender des GOS-Browsers. Zuerst wird die allgemeine Funktionsweise des Browsers beschrieben. Anschließend wird gezeigt, wie das Programm aufzurufen ist. Dannach folgt eine detailierte Beschreibung aller Kommandos und Auswertungsmöglichkeiten. Abgerundet wird dieses Kapitel durch ein Anwendungsbeispiel, anhand dessen einige Auswertungen mit Ergebnissen studiert werden können.

♦ **GOS-Grammatik**
Die GOS-Grammatik wird in Kapitel 4 vorgestellt. Nach einem kurzen Überblick über den Gofer-Teil und den OO-Teil wird die vom Parser erkannte Grammatik bis auf den Gofer-Teil beschrieben. Dabei

wird auf Unterschiede zwischen der ursprünglichen Fassung der Gofer-Grammatik und der hier verwendeten Variante eingegangen. Zuletzt wird auf das Layout von GOS-Programmen eingegangen. Hierzu gehört die Verwendung von Kommentaren und die Anwendung der sogenannten Layout-Rule, welche dann auch detailiert beschrochen wird.

♦ **Implementierung**
Die Implementierung des GOS-Browsers wird im Kapitel 5 besprochen. Es richtet sich daher primär an Entwickler, ist aber auch für Anwender durchaus interessant. Zu Beginn werden die Komponenten des GOS-Browsers und deren Funktionsweise beschrieben. Anschließend wird die Aufteilung des Programms auf mehrere Dateien, sowie deren Bedeutung erklärt. Im Anschluß daran wird das Parse-Framework ausführlich vorgestellt. Hierzu werden die verwendeten Gofer-Datenstrukturen und Klassen sowie deren Bedeutung und Verwendung beschrieben.
Als nächstes wird die Verarbeitung von Kommentaren und die Realisierung der Layout-Rule detailiert dargestellt. Die hierzu entwickelten Methoden der Scanner- und der Parser-Komponente werden erklärt und die Zusammenarbeit dieser beiden Komponenten beschrieben. Als letztes wird auf die Implementierung der Auswertungen eingegangen. Anhand von Beispielen werden die entwickelten Algorithmen vorgestellt.

Eine Zusammenfassung und ein kurzer Ausblick sowie mehrere Anhänge und das Literaturverzeichnis schließen die vorliegende Arbeit ab.

1.4 Notation und Glosar

Notation
Der Text enthält zahlreiche Code-Beispiele, welche mit einem speziellen Zeichensatz dargestellt werden. Die Größe des Zeichensatzes variiert an manchen Stellen.

```
CLASS CodeBeispiel SUBCLASSOF Beispiel
```

Auch die Produktionen der vorgestellten Grammatik werden in diesem Zeichensatz dargestellt.

```
GOSModule          ::=   "{" GOSTopDecls "}"                      ...Kommentar...
```

Glosar
Einige spezielle Begriffe und Abkürzungen sollen hier beschrieben werden, um eventuellen Mißverständnissen vorzubeugen.

♦ *AST*
Abkürzung von abstrakter Syntaxbaum (Abstract Syntax Tree)

♦ *Top-Level-Ebene bzw. Top-Level-Definitionen*
Definitionen auf der obersten Ebene eines Gofer- oder GOS-Programms werden als Top-Level-Definitionen bezeichnet - sie befinden sich auf der Top-Level-Ebene. Hingegen befinden sich Attribute- und Methoden-Definitionen oder auch lokale Definitionen in einer **where**-Klausel nicht auf der Top-Level-Ebene.

♦ *Klasse bzw. Class und Instance*
Der OO-Teil der Sprache GOS führt den Begriff der Klasse ein. Aber auch die Sprache Gofer kennt einen Klassenbegriff. Hierzu gehört auch der Begriff Instance. Um Verwechlungen zu vermeiden, werden diese beiden immer als *Typ-Klassen* bzw. *Typ-Instanzen* oder *Class-Definition* bzw. *Instance-Definition* bezeichnet. Die OO-Klasse hingegen wird immer mit dem Ausdruck *Klasse* schlechthin bezeichnet werden.

♦ *Parser*
Die Parser-Komponente des GOS-Browsers ist ein Objekt der Klasse **`YY_GBisonParser`**, deren Code vom Programm GBison generiert wird. Die Aufgabe dieses Objektes ist die grammatikalische Analyse und Erzeugung des abstrakten Syntaxbaumes.

♦ *Scanner*
Die Scanner-Komponente des GOS-Browsers ist ein Objekt der Klasse **`YY_GLexLexer`**, deren Code vom Programm GLex generiert wird. Die Aufgabe dieses Objektes ist die lexikalische Analyse der zu verarbeitenden Dateien.

Kapitel 2

Programmiersprache GOS

Die Programmiersprache GOS (Gofer Objekt System) ist eine Erweiterung der Programmiersprache Gofer. Sie ergänzt die funktionale Sprache um Elemente einer objektorientierten Sprache. Durch diese Kombination werden beide Programmierstile und somit auch deren Vorteile in einer einzigen Programmiersprache zugänglich, siehe [12]. In diesem Kapitel wird die Programmiersprache GOS und die Grundprinzipien der funktionalen und objektorientierten Programmierung in ihren Grundzügen dargestellt. Ansonsten sei auf [2], [7], [9] und [11] verwiesen.

2.1 Die funktionale Programmiersprache Gofer

Der Programmierstil imperativer Sprachen besteht darin, Sequenzen von Anweisungen mit Hilfe geeigneter Kontrollstrukturen für die Ausführung und Wiederholung zu arrangieren. Die Essenz der funktionalen Programmierung hingegen liegt darin, zumeist einfache Funktionen zu komplexeren zu kombinieren.

2.1.1 Mathematische Funktionen und funktionale Sprache

Eine wesentliche Eigenschaft funktionaler Sprachen ist, daß sie die referenzielle Transparenz der Mathematik erhalten. Dies bedeutet, daß Funktionen beliebig hierarchisch kombiniert werden können. Dies gilt im Allgemeinen nicht für Programmiersprachen, die über das Konzept der modifizierbaren Variable verfügen. Hierzu gehören auch und gerade die imperativen Programmiersprachen. Das nachfolgende kleine Beispiel verdeutlicht dieses Problem, nämlich das Problem des Seiteneffekts:

Gegeben seien die Definitionen der Variable x und der Prozedur p in einer Pascal-ähnlichen Sprache:

```
VAR x: Int

PROCEDURE p: Int
BEGIN
   x := x + 1;
   RETURN x;
END
```

Dann ist der Ausdruck **p - p** ungleich 0, was gleichbedeutend ist mit dem Ausdruck **p ≠ p**! In diesem Beispiel verhindert also der auftretende Seiteneffekt, daß zwei Aufrufe der Funktion *f* nicht den gleichen Wert liefern. In einer funktionalen Programmiersprache ist dies nicht möglich, da modifizierbare Variablen nicht existieren und das Funktionsergebnis tatsächlich nur von den Eingabewerten abhängt.

Viele mathematische Funktionen sind rekursiv definiert, d.h. die Definition der Funktion enthält eine Anwendung der Funktion selbst. So ist die mathematische Standarddefinition der Fakultät zumeist wie folgt gegeben. Die folgenden Beispiele werden dabei in Gofer-Syntax angegeben:

```
fac n = if n == 0 then 1 else n * (fac n - 1)
```

In einer Funktionsdefinition sind beide Seiten gleichwertig. Die referenzielle Transparenz garantiert, daß jede Seite durch die andere ersetzt werden darf. Ein anderes Beispiel ist die Definition der Fibonacci-Zahlen:

```
fib n = if n == 0 || n == 1 then 1 else (fib n - 1) + (fib n - 2)
```

Die Rekursion ist eine sehr mächtige Problemlösungstechnik. Bei der funktionalen Programmierung ist sie eine natürliche und häufig eingesetzte Strategie.

2.1.2 Datentypen

Die Datenobjekte stellen die Quell- und Zielmengen der Funktionen dar. Neben den einfachen Mengen für Zahlen, Boolesche Werte oder Zeichen besteht die Möglichkeit zur Bildung von Tupel- und Listen-Datenobjekten sowie der Konstruktion eigener Datentypen. Gofer bietet die folgenden grundlegenden primitiven Datentypen:

- Bool (Wahrheitswerte)
- Int (ganze Zahlen)
- Float (Gleitkommazahlen)
- Char (Zeichen)
- String (Zeichenketten)

In Gofer werden Tupel durch die Aufzahlung der Komponententypen gebildet, wobei diese durch Komma getrennt und in Klammern gesetzt werden. Listen werden durch den Typ der Listenelemente gebildet, wobei der Typ in eckigen Klammern eingeschlossen wird. Gofer erlaubt auch die Einführung von Abkürzungen für Datentypen. So könnten zum Beispiel die Typen **Point** und **Lines** wie folgt definiert werden:

```
type Point  =  (Float,Float)

type Lines  =  [String]
```

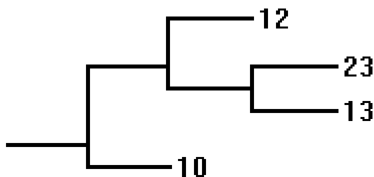
Neue Datentypen werden durch die Data-Definition eingeführt. Die Definition eines binären Baums (und somit einer rekursiven Datenstruktur) würde etwa so aussehen:

```
data Tree a  =  Lf a | Tree a :^: Tree a
```

Der Operator **:^:** ist ein sogenannter Konstruktor, mit welchem Werte dieses Typs erzeugt werden können. Eine Anwendung dieses Datentyps ist zum Beispiel der folgende Ausdruck:

```
( Lf 12 :^: ( Lf 23 :^: Lf 13 ) ) :^: Lf 10
```

Der obige Ausdruck repräsentiert somit folgende Datenstruktur:



2.1.3 Funktionsvereinbarung und -anwendung

Funktionsdefinitionen führen neue Funktionen ein. Die Argumente werden mit Hilfe von sogenannten Pattern angegeben, wobei zur Abdeckung aller Möglichkeiten der Argumente auch mehrere Varianten zu ein und der selben Funktion angegeben sein können. Bei Anwendung einer Funktion wird eine passende Variante gesucht, indem die tatsächlichen Argumente mit den Pattern verglichen werden. Die passende Variante kommt dann zur Ausführung. Zudem kann die Signatur einer Funktionen zusätzlich festgelegt werden.

Als Beispiel sei hier die Funktion gegeben, welche aus einem gegebenen Baum die Höhe errechnet:

```
height :: Tree a -> Int
height Lf a        =  1
height (a :^: b)   =  1 + max (height a) (height b)
```

Folgendes Beispiel zeigt die Anwendung dieser Funktion. Das Ergebnis ist 4:

```
height ( ( Lf 12 :^: ( Lf 23 :^: Lf 13 ) ) :^: Lf 10 )
```

Die primitiven Funktionen bilden den Kern der funktionalen Sprache. Alle neu definierten Funktionen werden letztendlich als Kombination dieser Funktionen aufgebaut. Die Menge der primitiven Funktionen ist durch die Sprache zumeist fest vorgegeben. Die Implementierung wird durch das zugrundeliegende System zur Verfügung gestellt und durch geeignete Konstrukte in die funktionale Ebene eingebettet. Zum Beispiel könnte die Funktion **==** zum Testen auf Gleichheit zweier Zahlen als primitive Funktion vorhanden sein. Eine solche Einbettung wird in Gofer mit dem Primitive-Konstrukt durchgeführt. Im folgenden Beispiel wird der Funktion **primEqInt** die primitive Funktion mit dem Namen "**primEqInt**" zugewiesen und die Signatur definiert.

```
primitive primEqInt "primEqInt" :: Int -> Int -> Bool
```

Die Anwendung oder Applikation einer Funktion auf ihre Argumente und die Produktion eines Wertes ist ein eingebauter Mechanismus. Dieser ersetzt die Parameter durch Argumente auf der rechten Seite und setzt diesen Mechanismus bei den dort gegebenenfalls vorhandenen Funktionen fort. In diesem Zusammenhang tritt die Schwierigkeit auf, diesen Mechanismus letztendlich durch das zugrundeliegende System durchführen zu müssen. Hierzu gibt es Auswertungsstrategien, die durch die Sprache festgelegt sind. Einerseits existiert die Möglichkeit, alle Argumente sofort zu berechnen und dann die Funktionen auszuwerten. Dies wird auch als „*eager evaluation*“ oder aber auch als „*call by value*“ bezeichnet. Seien die Funktion *f* und *g* folgendermaßen definiert:

```
f x ≡ f x

g y ≡ 0
```

Somit terminiert die Applikation

```
g(f(17))
```

nicht. Andererseits gibt es auch die Möglichkeit, Argumente nur dann auszuwerten, wenn sie für die Ermittlung eines Funktionswertes benötigt werden. Diese Strategie wird als „*lazy evaluation*“ oder „*call by name*“ bezeichnet. In diesem Fall terminiert die obige Applikation sehr wohl und liefert das Ergebnis 0. Dabei wird der Ausdruck **`f(17)`** mit dem Parameter **`y`** identifiziert, welcher für die Berechnung des Ergebnisses irrelevant ist. Die Auswertung von **`y`** und damit die Applikation von **`f 17`** ist damit nicht notwendig und wird somit auch nicht ausgeführt.

Neben diesen Elementen besitzt die funktionale Sprachen Gofer die Eigenschaft der Typinferenz. Dies bedeutet, daß sich der Typ eines Ausdrucks Aufgrund seiner Struktur und der zugrundeliegenden Funktionen herleiten läßt. Auf diese Weise können Typinkonsistenzen bei der Anwendung von Funktionen vom zugrunde liegenden System erkannt werden.

2.1.4 Funktionen höherer Ordnung

Neue Funktionen können durch Kombination bestehender Funktionen einschließlich der zu definierenden Funktion (Rekursion) definiert werden. Die bekannteste Form der Kombination von Funktionen ist die Funktionskomposition. Definiert ist die Komposition f°g der Funktionen f und g dadurch, daß zuerst die Funktion g auf die Argumente angewandt wird und dann anschließend die Funktion f auf das Resultat der ersten Anwendung angewandt wird. Zum Beispiel könnte die Funktion *zur_vierten* mit Hilfe der Funktion *quadrat* und der Komposition folgendermaßen definiert werden (der Operator **°** wird in Gofer als Punkt geschrieben):

```
zur_vierten  =  quadrat.quadrat
```

Die Funktionskomposition ist ein Beispiel für eine Funktion höherer Ordnung, auch funktionale Form genannt. Eine Funktion höherer Ordnung ist eine Funktion, die andere Funktionen als Parameter hat und eine Funktion als Ergebnis liefert. Sie ist eine Methode zur Kombination von Funktionen.

Auch das *if-then-else*-Konstrukt stellt eine funktionale Form dar. Mit Hilfe dieses Konstrukts wird aus drei Funktionen eine neue Funktion erzeugt. Es werden lediglich Einschränkungen über die Zielmengen der eingehenden Funktionen verlangt. So muß die Zielmenge der ersten Funktion die der Booleschen Werte sein, und die Zielmengen der letzten beiden Funktionen muß identisch sein.

2.1.5 Zusammenfassung

Imperative Sprachen sind effizienter im Hinblick auf die Ausführungszeit, da sie die Struktur und die Operationen der zugrunde liegenden Maschine widerspiegeln. Der Programmierer muß sich allerdings mit Details auf der Maschinenebene befassen, was sich im Programmierstil niederschlägt. Dieser basiert auf der Benennung und Zuweisung von elementaren Zellen und der Wiederholung von elementaren Aktionen.

Der funktionale Programmierstil hingegen erlaubt die Verwendung von Datenobjekten ohne Rücksicht auf Speicherzellen. Werte werden nicht zugewiesen, sondern durch Anwendung von Funktionen produziert und an andere Funktionen weitergegeben. Insgesamt scheint sich die funktionale Programmierung auf einer höheren Ebene abzuspielen, als die imperative Programmierung.

Durch die Forderung nach referenzieller Transparenz und das Fehlen von modifizierbaren Variablen leidet natürlich die Effizienz darunter, da Objekte ständig dynamisch erzeugt und beseitigt werden müssen.

Ein weiteres Problem bei funktionalen Sprachen ist I/O, da dies ein Seiteneffekt ist. Daher ist die Ein- und Ausgabe oft umständlich zu realisieren.

Trotzdem zeichnen sich funktionale Sprachen durch sehr angenehme Eigenschaften aus. Dazu gehören unter anderem ein klares Typsystem und die Nutzung der mathematischen Eigenschaften von Funktionen. Zudem läßt sich, anders als bei imperativen Programmiersprachen, die Semantik funktionaler Programmiersprachen mathematisch definieren. Damit sind Programme mathematischen Methoden zur Analyse, zum Beispiel die der Programmverifikation, zugänglich.

2.2 Objektorientierte Programmierung mit GOS

Dieser Abschnitt beschäftigt sich mit den Grundprinzipien der objektorientierten Programmierung. Die Philosophie dahinter betrachtet die Welt abstrakt als aus gleichberechtigt und einheitlich erscheinenden Objekten. Diese Sichtweise, welche natürlich stark an die gegebenen Anforderungen angepaßt ist, wird von vielen objektorientierten Sprachen mehr oder weniger verwirklicht. Obwohl die Palette solcher Sprachen weit gefächert ist, haben sie dennoch die gleichen Grundprinzipien als Grundlage. Dieser gemeinsame Kern läßt sich in seiner reinen Form durch wenige Konzepte beschreiben, wenngleich diese in den jeweiligen Sprachen unterschiedlich ausgeprägt sind.

Der Blick ist in der folgenden Darstellung auf die Sprache GOS gerichtet. Aus diesem Grund werden die Konzepte nur insoweit vorgestellt, als daß sie für das Verständnis der Sprache GOS notwendig und hilfreich sind. Für eine zusammenfassende Darstellung der Objektorientierung sei auf [3] und auf eine ausführliche Diskussion auf [2] und [11] verwiesen.

2.2.1 Objekte

Einer der zentralen Begriffe der objektorientierten Programmierung ist, wie der Name bereits andeutet, der des Objekts. Diese werden dazu verwendet, „Dinge“ aus der wirklichen Welt zu repräsentieren. Zu

einer gegebenen Problemstellung werden die relevanten Eigenschaften abstrahiert und mit Objekten modelliert.

Dabei gilt als wichtigstes Prinzip, daß Objekte quasi nur von Außen betrachtet werden und ihr Innenleben verborgen bleibt. Dies erfordert sehr viel Vertrauen. Man kann Objekten nicht seine eigenen Algorithmen aufzwingen, sondern man muß sich darauf verlassen, daß Objekte korrekt arbeiten. Diese Denkweise kommt der menschlichen Denkweise entgegen. Das beschriebene Prinzip ist unter dem Begriff Kapselung oder auch Information Hiding bekannt: es wird versucht, Darstellungsunabhängigkeit zu erreichen, so daß zum Verstehen und zur Nutzung eines Objekts seine konkrete Implementierung nicht bekannt sein muß.

Ein Objekt kann im wesentlichen durch die drei folgenden Eigenschaften charakterisiert werden:

- Zustand
- Verhalten
- Identität

♦ Zustand

Der Zustand eines Objekts ergibt sich aus dem Inhalt seiner Attribute. Dies können elementare bzw. einfache Objekte wie zum Beispiel Zahlen oder Zeichenketten, aber auch komplexere Objekte sein. Der Zustand eines Objekts hängt somit auch vom Zustand seiner Komponenten ab.

Es gibt objektorientierte Sprache, die nur Objekte als Daten repräsentierende Einheiten kennen. Zu diesem Sprachen gehört zum Beispiel Smalltalk. Diese fast schon dogmatische Auffassung ist allerdings nicht nötig. Es ist auch möglich, Daten ohne diese Datenkappselung im Sinne „gewöhnlicher“ Datentypen einzusetzen. Beispielsweise könnten Zahlen und Boolesche Werte nicht als Objekte, sondern als Werte im herkömmlichen Sinne betrachtet werden. Die Palette ist dabei breit gefächert.

♦ Verhalten

Objekte halten nicht nur Daten, sondern sind auch aktive Elemente einer objektorientierten Programmiersprache. Deshalb verfügt ein Objekt auch über das Verhalten, in einer definierten Art und Weise zu agieren und zu reagieren. Jedes Objekt stellt Methoden bereit, die von anderen Objekten angestoßen werden können. Durch die Ausführung einer Methode ändert sich mitunter der Zustand des Objektes.

So könnte es zum Beispiel für ein Objekt, welches einen Fahrstuhl modelliert, eine Methode geben, die Fahrt in Richtung eines bestimmten Stockwerkes aufzunehmen. Da der Zustand dieses Fahrstuhl-Objekts die Fahrtrichtung miteinschließt, hätte sich nach Aufruf dieser Methode die Fahrtrichtung und somit auch der Zustand geändert. Das Objekt hat also in einer definierten Art und Weise reagiert. Auf der anderen Seite ist es gut möglich, daß das Objekt durch diesen Methodenaufruf auch aktiv wird. Es könnte zum Beispiel das Objekt, welches die Aufzusteuerung modelliert, damit beauftragen, die Fahrt durch Ansteuerung eines Motors aufzunehmen.

Das Verhalten eines Objekts ist im Allgemeinen von seinem Zustand abhängig. So sind im obigen Beispiel unterschiedliche Reaktionen auf den Fahrtrichtungswunsch denkbar. Falls der Aufzug bereits in diese Richtung fährt und das gewünschte Stockwerk noch nicht passiert hat, muß nichts weiter unternommen werden. Falls der Aufzug zu diesem Zeitpunkt „arbeitslos“ ist, wird er zweifelsohne die Arbeit aufnehmen, vorrausgesetzt, daß in diesem Modell Aufzüge nicht „außer Betrieb“ sein können.

Jedoch kann man nicht erwarten, daß der Aufzug halt macht und umkehrt, wenn er gerade in entgegengesetzter Richtung ein anderes Stockwert anfährt.

♦ **Identität**

Jedes Objekt hat unabhängig von seinem Zustand eine Identität, welche das Objekt von anderen Objekten eindeutig unterscheidet. Da auf Objekte gewöhnlich durch Variablenamen zugegriffen wird, besteht die Gefahr, Variablenamen mit den Objekten gleichzusetzen. Variablen verweisen auf Objekte, sind nur programmatische Konstrukte, um den Zugriff auf Objekte zu ermöglichen. Mehrere Variablen können auf ein und das selbe Objekt verweisen, und eine Variable kann im Laufe der Zeit auch auf mehrere Objekte verweisen. Ein Objekt kann auch existieren, ohne daß eine Variable direkt darauf verweist.

Die Unabhängigkeit von Objekten führt auch zur Tatsache, zwischen Gleichheit und Identität unterscheiden zu müssen. Zwei Objekte sind identisch, falls sie die gleiche Identität besitzen, wenn sie also im Grunde ein und das selbe Objekt darstellen. Hingegen können zwei unterschiedliche Objekt gleich sein ohne damit auch sofort identisch sein zu müssen. Die Gleichheit stützt sich dabei auf den Zustand der Objekte. Zum Beipiel können Punkte im zweidimensionalen Raum durch Objekte modelliert werden. Die Attribute x und y sollen dabei die x- und y-Koordinaten repräsentieren. Zwei Punkt-Objekte sind somit gleich, wenn die Werte der x- und y-Attribute gleich sind. Auch hier tritt wieder das Problem auf, zwischen Gleichheit und Identität unterscheiden zu müssen. Es wäre auch denkbar, das Verhalten von Punkt-Objekten so auszulegen, daß gleiche Objekte im obigen Sinne auch identisch sind. (Damit müßte man allerdings verbieten, daß die x- und y-Koordinaten direkt geändert werden können.)

Die Tatsache, daß es mehrere Objekte mit gleichem Verhalten gibt, sowie die Tatsache, daß Objekte in einer geeigneten Art und Weise programmtechnisch beschrieben werden müssen, führt uns zum Begriff der Klasse.

2.2.2 Klassen

Eine Klasse beschreibt Struktur und Verhalten von Objekten. So gesehen gehört jedes Objekt zu einer Klasse. Etwas pragmatischer betrachtet, kann man eine Klasse als Template betrachten. Aus diesem Template lassen sich dann Objekte dynamisch erzeugen, die Struktur und Verhalten dieser Klasse aufgeprägt bekommen. Objekte werden als Instanzen, Exemplare oder auch als Ausprägungen einer Klasse bezeichnet.

Klassen führen Definitionen von Begriffen ein, wohingegen Objekte konkrete Ausprägungen dieser Begriffe sind. Die Beziehung zwischen Klasse und Objekt ist etwa die, wie zwischen Menge und Element dieser Menge. Dennoch bereitet der Unterschied zwischen Objekt und Klasse oft Schwierigkeiten. Dies liegt nicht zuletzt daran, daß die natürliche Sprache diese Begriffe allzuoft vermischt oder sogar miteinander identifiziert. Objekte werden in der Regel mit Hilfe der Namen ihrer Klassen benannt, so zum Beispiel ein Fahrstuhl, ein Punkt usw.

Eine Klasse umschließt somit folgende Definitionen:

- Datentyp
- Methoden

- Attribute

♦ **Datentyp**
Die Definition einer Klasse führt in der Regel einen neuen Datentyp ein. Damit können zum Beispiel Attribute, Variablen oder Parameter mit dem Typ einer Klasse deklariert werden. Beim Erzeugen neuer Objekte erhält man als Ergebnis einen Objektidentifikator. Diese werden dann gewöhnlich wie Werte anderer Typen betrachtet, nur daß deren Typ genau dieser Klassen-Datentyp ist. Dies gilt natürlich nicht für ungetypte Sprachen wie Smalltalk.

♦ **Methoden**
Die Methoden bestimmen das Verhalten der Objekte. So gibt es Methoden, die den Zustand des Objekts nicht ändern, also nur einen Aspekt des Zustandes abfragen. Solche Methoden interessieren sich nur für bestimmte Eigenschaften und haben oft die Form von Adjektiven, wie zum Beispiel *isActive* oder *isEmpty*. Andere Methoden wiederum können und sollen den Zustand des Objekts ändern. Solche Methoden sind oft in Befehlsform gefasst wie zum Beispiel *push* oder *move*. Natürlich gibt es alle denkbare Zwischenformen und Kombinationen.

♦ **Attribute**
Attribute legen die innere Struktur eines Objektes fest, welche mehr oder weniger von der jeweiligen Implementierung abhängt. Zugleich bestimmt der Inhalt der Attribute den Zustand eines Objektes. Dagegen hat der Zustand, welcher im Sinne der Anforderungen interpretiert wird, abstrakteren Charakter, denn eine gewählte Struktur ist nur eine von vielen möglichen und dient ausschließlich als Mittel zum Zweck.

Zum Beispiel könnte die Fahrtrichtung eines Fahrstuhl-Objekts ein Teilaspekt des Zustandes sein. Dann wäre es denkbar, diesen Aspekt durch Zahlen zu repräsentieren, also zum Beispiel „*nach oben*“ mit 1, „*nach unten*“ mit -1 und „*in Ruhe*“ mit 0. Für ein Modell eines anspruchsvolleren Fahrstuhls, welcher zusätzlich die Geschwindigkeit kennt und steuert, könnte dieser Aspekt sogar aus der Geschwindigkeitsangabe abgeleitet werden. Die Richtung könnte also zum Beispiel durch das Vorzeichen der Geschwindigkeit errechnet werden.

Dieses Beispiel macht klar, daß Attribute eng im Zusammenhang mit einer konkreten Implementierung zu sehen sind, wohingegen Objekte ja gerade die Darstellungsunabhängigkeit sichern sollen. Trotzdem kann es sinnvoll sein, Attribute mit oder ohne Methoden direkt zugänglich zu machen, falls sie direkt mit Aspekten des Zustandes identifiziert werden können.

Die Fahrstuhl-Klasse könnte in GOS wie folgt definiert werden:

```
CLASS Elevator
PUBLIC
   -- Bewege den Fahrstuhl zum angegebenen Stockwert
   moveToFloor(destination :: Int) =
   BEGIN
      ...
   END;
```

```
    -- Schließe die Türen
    closeDoors() =
    BEGIN
        ...
    END;

    -- Öffne die Türen
    openDoors() =
    BEGIN
        ...
    END;

    -- Sind die Türen gerade offen?
    doorsOpen() Bool =
    BEGIN
        ...
    END;

PROTECTED
    direction :: Int; -- 0, 1, -1 (siehe oben)
    ...
ENDCLASS Fahrstuhl;
```

Darüberhinaus gibt es Beziehungen zwischen Klassen und somit den Objekten, die als isA-Beziehungen bezeichnet werden. Dies wird durch das Konzept der Vererbung in objektorientierte Programmiersprachen eingeführt, welches nun im nächsten Abschnitt vorgestellt wird.

2.2.3 Vererbung

Die isA-Beziehung ist eines der wichtigsten Ausdrucksmittel, das in der objektorientierten Programmierung zum Einsatz kommt. Wenn zwei Klassen K und K' in einer isA-Beziehung stehen, wenn also gilt „*K' isA K*", drückt man hierdurch die Tatsache aus, daß die Klasse K' eine Spzialisierung der Klasse K ist. Gleichzeitig bedeutet dies, daß die Klasse K gegenüber der Klasse K' eine Generalisierung darstellt. Ist nun das Objekt O von der Klasse K und das Objekt O' von der Klassen K', so ist das Objekt O' zum Objekt O kompatibel: alle Eigenschaften sowie Verhaltensweisen von O sind auch bei O' vorhanden. Dies wird oft dadurch ausgedrückt, daß man sagt, die Klasse K' erbt von der Klasse K. In diesem Zusammenhang wird K als Oberklasse und K' als Unterklasse bezeichnet.

Die Vererbung kommt in existierenden Sprachen in zwei Arten vor. Zum einen gibt es die Einfachvererbung, bei der eine Klasse höchstens von einer anderen Klasse erben kann. Zum anderen gibt es die Mehrfachvererbung, bei der eine Klasse mitunter auch von mehreren Klassen erben kann. Letztere Alternative bietet zwar erheblich mehr Freiheiten in der Modellierung des Problems, zieht aber leider auch Schwierigkeiten nach sich, die bei der Einfachvererbung nicht auftreten. Auf der anderen Seite ist es oft schwierig bis unmöglich, mit Einfachvererbung adäquat zu modellieren. Deshalb kann man bei Programmiersprachen mit Einfachvererbung allzu oft beobachten, daß allgemeineres Verhalten immer weiter nach oben, daß heißt zu Oberklassen rückt, um auch anderen Zweigen des Verbungsbaumes dieses

Verhalten nutzbar zu machen. Dies gilt dann natürlich auch für Klassen, für die es gar nicht gedacht war. Mit Mehrfachvererbung hingegen kann durch Bildung und Nutzung von mehreren Oberklassen, die durch die entsprechenden Unterklassen beerbt werden, sozusagen in die „Breite" modelliert werden. Auf die Problematik wird aber hier nicht weiter eingegangen, da sie für die Darstellung der Grundprinzipien nicht notwendig ist.

Die Beziehungen, die durch Vererbung entstehen, lassen sich durch einen gerichteten und zyklenfreien Graphen darstellen, in dem die Klassen die Knotenmenge bilden. Falls die Klasse K' von der Klasse K erbt oder abgeleitet ist, wie man auch sagt, so enthält der Graph eine gerichtete Kante von K nach K'. Im Falle der Einfachvererbung erhält man hierdurch einen Wald, also eine Menge von Bäumen. Deshalb wird dieser (Vererbungs-)Graph oft als Vererbungsbaum oder Vererbungshierarchie bezeichnet, obwohl es sich im allgemeinen Fall nicht mehr um Bäume handelt.

Betrachtet man einen Vererbungsgraph vom Standpunkt der isA-Beziehungen aus, so regelt dieser das Vererben der Eigenschaften von Oberklassen auf Unterklassen. Hierzu gehört ausschließlich das Verhalten von Objekten, da implementative Aspekte ausgeklammert sind. Programmtechnisch handelt es sich hierbei um die Schnittstelle eines Objekts bestehend aus Methoden und deren Signaturen. Die Implementation bleibt unberücksichtigt. Lediglich die semantischen Eigenschaften dieser Methoden werden vererbt. In der Praxis jedoch ist mit Vererbung viel mehr verbunden, als nur die Weitergabe von Schnittstellen und Semantik. Auch die Implementierungen von Methoden und die Struktur der Objekte, gebildet durch die Attribute, werden an die Unterklassen vererbt. Auf diese Weise kann die Implementation der Oberkasse wiederverwendet werden. Man spricht in diesem Zusammenhang auch von Code-Reuse.

Eine Gefahr dabei ist, daß dieser Mechanismus zum Mißbrauch führen kann, wenn nicht die isA-Beziehung im Vordergrund steht sondern der Code-Reuse. Das Ergbnis wird dann als Implementationshierarchie bezeichnet. Ein weiterer Nachteil ist folgender: eine Klasse definiert das Verhalten von Objekten seiner Klasse und entspricht im Grunde einem abstrakten Datentyp. Zugleich gibt die Klasse zumeist auch eine konkrete Implementierung vor. In Unterklassen wird das Verhalten der Oberklasse in der Regel spezialisiert und somit ein neuer abstrakter Datentyp eingeführt. Stellt sich nun heraus, daß dieser neue Datentyp durch eine geeignetere Implementierung realisiert werden könnte, bekommt man jedoch Schwierigkeiten, da auch die Implementierung der Oberklasse mitgeerbt wird. Dieses Problem wiegt um so schwerer, als das durch eine Typisierung in der Programmiersprache isA-Beziehungen gefordert, dadurch aber auch Implementierungen festgelegt werden. Aber genau dies erschwert die Erweiterbarkeit von Klassenhierarchien. Es ist daher notwendig, Klassenhierarchien im Hinblick auf diese Problematik sehr sorgfältig aufzubauen. In ungetypten Sprachen wie zum Beispiel Smalltalk, in denen der Zwang zur Bildung von isA-Beziehungen nicht existiert, sondern vielmehr zum guten Ton gehört, kann man oft die extensive Nutzung von Implementationshierarchien beobachten.

Durch die Vererbung der Implementation von Methoden kann es erforderlich sein, diese der Spezialisierung der Unterklasse anzupassen. In diesem Fall wird eine geerbte Methode überschrieben und mit einer passenden Implementierung versehen. Der Aufruf der überschriebenen Methode der Oberklasse ist in vielen objektorientierten Programmiersprachen vorgesehen.

Ein Beispiel soll diesen Vorgang verdeutlichen. Die Klasse *GraphicObject* beschreibt das allgemeine Verhalten von zweidimensionalen graphischen Objekten, natürlich im Hinblick auf die hier gewünschten Anforderungen eines Beispiels. Die Klasse *GraphicObject* definiert unter anderem eine Methode *move* zum Verschieben des Objekts in x- und y-Richtung. Da die Methode *move* den Zustand des Objektes verändert und dies damit die Kenntniss der Implementierung vorraussetzt, muß die Implementierung der Methode in der Klasse *GraphicObject* offen bleiben. In manchen Sprachen wie zum Beispiel Eiffel

könnte man solche Methoden als abstrakt deklarieren. Für die Unterklasse *Rectangle* wird die Struktur gewählt, zwei Attribute für oberen linken und unteren rechten Eckpunkt zu verwenden. Die Methode *move* der Klasse *Rectangle* überschreibt nun die eventuell undefinierte Implementation der Oberklasse, indem die Eckpunkte um den entsprechenden Betrag verschoben werden. Um nochmals auf die oben erwähnte Problematik der Vererbung einer Implementierung einzugehen, leiten wird die Klasse *Square* von der Klasse *Rectangle* ab. Denn ein Quadrat *ist ein* Rechteck. Alle Eigenschaften eines Rechtecks sind auch bei einem Quadrat vorhanden. Dies drückt man auch dadurch aus, daß ein Quadrat ein spezielles Rechteck ist. Das Quadrat verfügt aber über eine stärkere mathematische Struktur als ein Rechteck, so daß weniger Angaben zur eindeutigen Charakterisierung notwendig sind. Es würde zum Beispiel genügen, linke obere Ecke und Kantenlänge anzugeben. Diese Möglichkeit ist jedoch versperrt, da die geerbte Implementation der Klasse Rectangle bereits einen anderen Weg vorgibt.

```
CLASS GraphicObject
PUBLIC
   -- abstrakte Methode
   move(delta :: Point) =
   BEGIN
   END;
   ...
ENDCLASS GraphicObject;

CLASS Rectangle SUBCLASSOF GraphicObject;
PUBLIC
   -- überschreiben der geerbten Methode
   -- mit konkreter Implementierung
   move(delta :: Point) =
   BEGIN
      origin := origin + delta;
   END;

   -- Ist das Rechteck ein Quadrat?
   isSquare() Bool =
   BEGIN
      RETURN width == height;
   END;
   ...
PROTECTED
   origin :: Point;
   width, height :: Float;
ENDCLASS Rectangle;

CLASS Square SUBCLASSOF Rectangle
   -- überschreiben der geerbten Methode
   isSquare() Bool =
   BEGIN
      RETURN True;
   END;
   ...
```

```
ENDCLASS Square;
```

Die Vererbung führt zu einem weitern wesentlichen Punkt der objektorientierten Programmierung. Dies sind die Begriffe des Polymorphismus und der späten Bindung.

2.2.4 Polymorphismus, Overloading und späte Bindung

Unter Polymorphismus versteht man das Prinzip, daß Attributen, Variablen, Parametern usw. Objekte verschiedener Klassen zugewiesen werden können. In getypten Sprachen wird diese Freizügigkeit durch die isA-Hierarchie eingeschränkt, so daß nur Objekte der angegebenen Klasse sowie deren Unterklassen zugewiesen werden können. Wurde also zum Beispiel eine Variable mit der Klasse *Rectangle* deklariert (siehe Beispiel im vorausgegangenen Abschnitt), so können Objekte der Klassen *Rectangle* und *Square* zugewiesen werden. Hingegen können Objekte der Klasse *Cycle*, die eine direkte Unterklasse der Klasse *GraphicObject* sein soll, nicht zugewiesen werden. Der Grund ist einfach: durch die Typisierung sollen Eigenschaften der auftretenden Objekte zugesichert werden. Da diese Eigenschaften auf die Unterklassen vererbt werden, können deshalb auch Objekte der Unterklassen eingesetzt werden.

```
CLASS Cycle SUBCLASSOF GraphicObject;
...
ENDCLASS Cycle;

...

VAR
   r :: Rectangle;
BEGIN
   r := Rectangle!NEW(x,y,w,h); -- ok
   r := Square!NEW(x,y,a);      -- ok

   r := Cycle!NEW(x,y,r); -- fehlerhaft!
END;
```

Unter Overloading versteht man die Möglichkeit, eine Methode in verschiedenen Klassen mitunter auch unterschiedlich zu implementieren. Dieser einfache Mechanismus kommt auch in der natürlichen Sprache vor. Zum Beispiel kann man sagen, das

- der Mensch läuft,
- die Nase läuft oder
- der Motor läuft.

Diese drei Varianten unterscheiden sich erheblich in ihrer Semantik. Ein anderes Beispiel ist die Addition von ganzen und reelen Zahlen. Beide Varianten werden durch **`a + b`** ausgedrückt, wobei die Unterschiede in diesem Fall nicht mehr so klar zu erkennen sind. Das Prinzip des Overloading ist

allerdings nicht auf Klassen und objektorientiere Programmiersprachen beschränkt, sondern tritt auch, wie im Falle der Addition zu erkennen ist, auch in anderen Programmiersprachen auf.

Durch den Polymorphismus und der Möglichkeit, die Implementation geerbter Methoden zu überschreiben, tritt eine weitere Schwierigkeit auf. Folgendes Beispiel soll diese Problematik veranschaulichen: ein Attribute eines Objekts ist als Klasse *Rectangle* deklariert. Desweiteren nehmen wir an, daß die Klasse *Rectangle* die Methode *rotate* implementiert, um das Objekt bezüglich seines Mittelpunkts um 90° im Uhrzeigersinn zu drehen. Da die Rotation eines Quadrats um 90° die Repräsentation nicht ändert, sind also für ein Quadrat hierfür keine Aktivitäten notwendig. Daher wird in der Klasse *Square* die Methode *rotate* mit einem leeren Rumpf überschrieben. Nun wird die Methode *rotate* an das Objekt geschickt, welches durch das oben beschriebene Attribute referenziert wird. Was soll geschehen, wenn es sich um ein Objekt der Klasse Rectangle bzw. Square handelt? Antwort: das kommt darauf an, welche Strategie vorgegeben wurde, wobei zwischen zwei Möglichkeiten zu wählen ist. Die erste wird mit dem Begriff „*static bindung*“ oder „*frühe Bindung*“, die zweite mit „*late binding*“ oder „*späte Bindung*“ bezeichnet. Im ersten Fall wird die Tatsache ignoriert, daß auch Objekte von Unterklassen auftreten können. Somit wird und kann nur die Methode der deklarierten Klasse, in unserem Beispiel die Methode *rotate* der Klasse *Rectangle* aufgerufen werden. Daher der Name „*fühe Bindung*“ (nämlich der Methoden). Im anderen Fall ist die Klasse des Objekts zur Laufzeit entscheidend. Erst beim Aufruf der Methode wird entschieden, welche Implementation zum Einsatz kommt, d.h. es wird *spät gebunden*. In dem selben Beispiel wird also in diesem Fall die Methode der Klasse *Square* aufgerufen. Ob nun früh oder spät gebunden wird, hängt entweder von der Programmiersprache allein oder vom Programmierer ab, falls dies die Sprache zuläßt.

Im folgenden Beispiel würde jeder der beiden Aufrufe eine unterschiedliche Methoden ausführen, obwohl beide an die Variable **`r`** gesendet wurden. Dies liegt daran, daß im ersten Fall die Variable zu diesem Zeitpunkt mit einem Objekt der Klasse **`Rectangle`** belegt ist, wohingegen im zweiten Fall die Variable r mit einem Objekt der Klasse **`Square`** belegt ist.

```
VAR
   r :: Rectangle;
   b :: Bool;
BEGIN
   r := Rectangle!NEW(x,y,w,h); -- ok
   b := r!isSquare();

   r := Square!NEW(x,y,a);      -- ok
   b := r!isSquare();
END;
```

2.2.5 Zusammenfassung

Programmiersprachen sind dazu da, Probleme zu lösen, die in der sogenannten wirklichen Welt existieren. Je besser es die Sprachen erlauben, die Probleme dieser wirklichen Welt zu erfassen und zu formulieren, desto einfacher - so die Annahme - ist es auch, diese Probleme zu lösen. In diesem Sinn versucht auch die objektorientierte Programmierung die gegebenen Anforderungen zu bewältigen.

Konkret bedeutet objektorientierte Programmierung, Klassen von Objekten zu beschreiben, Objekte dynamisch zu erzeugen und ein Objekt eine Methode ausführen zu lassen, damit es die Aufgabe löst.

Dieses Objekt wird Teilaufgaben an andere Objekte deligieren, indem es Methoden aufruft und deren Lösungen verarbeitet.

Allerdings führt das objektorientierte Programmieren zu einer anderen Form der Programmierung: Probleme werden nicht dadurch gelöst, daß man die Lösung von Grund auf neu entwickelt, sondern versucht, aufbauend auf einem Grobkonzept, Klassen zu finden, deren Zusammenwirken möglicherweise die Fragestellung lösen kann. Zu diesen Klassen werden dann Unterklassen konstruiert, die speziell auf die Problemstellung zugeschnitten sind. Auf diese Weise erhält man frühzeitig ein erstes lauffähiges System (Prototyping), und das gewünschte Programmsystem entsteht durch schrittweises Annähern dieses Prototpyen an die Erfordernisse.

Kapitel 3

GOS-Browser

In diesem Kapitel wird die Anwendung und Arbeitsweise des GOS-Browsers beschrieben. Die hierzu verwendeten Programmbeispiele sind zu Anschauungszwecken nur auszugsweise und in gekürzter Form dargestellt. Im Anhang E sind die Programme vollständig aufgeführt. Eine Funktions-Referenz des GOS-Browsers ist im Anhang B enthalten.

3.1 Eingesetzte Programme

Der GOS-Browser wurde in GOS geschrieben und kommt somit mit Hilfe des GOS-Interpreters zur Ausführung. Um allerdings Unabhängigkeit von der Implementierung und den Versionen zu gewährleisten, wird zusätzlich ein Script zur Verfügung gestellt.

Im nächsten Abschnitt wird dieser Prozeß ausführlich beschrieben. Für eine detailiertere Darstellung insbesondere des GOS-Browsers und des Scripts sei auf das Kapitel „Implementierung“ verwiesen.

Die nun folgenden Angaben über die verwendeten Programme sollen nur als Anhaltspunkte dienen. Im Anhang A werden die zum Einsatz kommenden Programme mit ihren Versionen und Lokationen detailiert beschrieben. In diesem Anhang finden sich dann auch Informationen zur Installation und Anwendung des GOS-Browsers.

GOS-Interpreter
Für die vorliegende Version wurde der GOS-Interpreter mit der Version 1.22 vom 27.08.1996 verwendet. Während der Arbeit zum GOS-Browser tratt beim Kompilieren des GOS-Browser-Programms ein Problem auf. Der Quellcode des GOS-Interpreters mußte geändert und das Programm neu übersetzt werden. Die Modifikation betraf die Variable NUM_FIXUPS, die die maximale Anzahl von Sprung-Labels festlegt, die beim Kompilieren eines Gofer-Ausdrucks erzeugt werden können. Der Wert dieser Variable wurde von 100 auf 500 erhöht.

GOS-Browser
Die in dieser Arbeit beschriebene Version des GOS-Browser ist die Version 1.0 vom 08.01.1997.

Da der GOS-Browser in GOS implementiert wurde, konnten die Entwicklungswerkzeuge GLex und GBision eingesetzt werden. Mit deren Hilfe wurden Teile des Programms generiert. Diese Werkzeuge, die nicht zur Laufzeit, sondern nur zur Erzeugung des Browsers benötigt werden, müssen jedoch mit der verwendeten GOS-Version kompatibel sein. Aus diesem Grund seien hier auch die Versionen von GLex

und GBison genannt. GLex wurde in der Version vom 25.11.1995 eingesetzt, und GBision kam in der Version vom 04.12.1995 zum Einsatz.

Script
Das Script ist direkt von der Version des GOS-Interpreters und des GOS-Browsers abhängig, da spezielle Aufrufparameter des GOS-Interpreters verwendet werden. Das hier beschriebene Version des Scripts ist die Version 1.0 vom 08.01.1997.

3.2 Allgemeine Beschreibung

In diesem Abschnitt wird die Arbeitsweise des GOS-Browser beschrieben. Für ein detailierte Beschreibung der Abläufe sei auf das Kapitel „Implementierung“ verwiesen.

Der Ablauf beginnt mit dem Aufruf des Scripts und endet mit der Ausgabe zu den angegebenen Auswertungen. Den groben Ablauf zeigt die nachfolgende Abbildung 3.1.

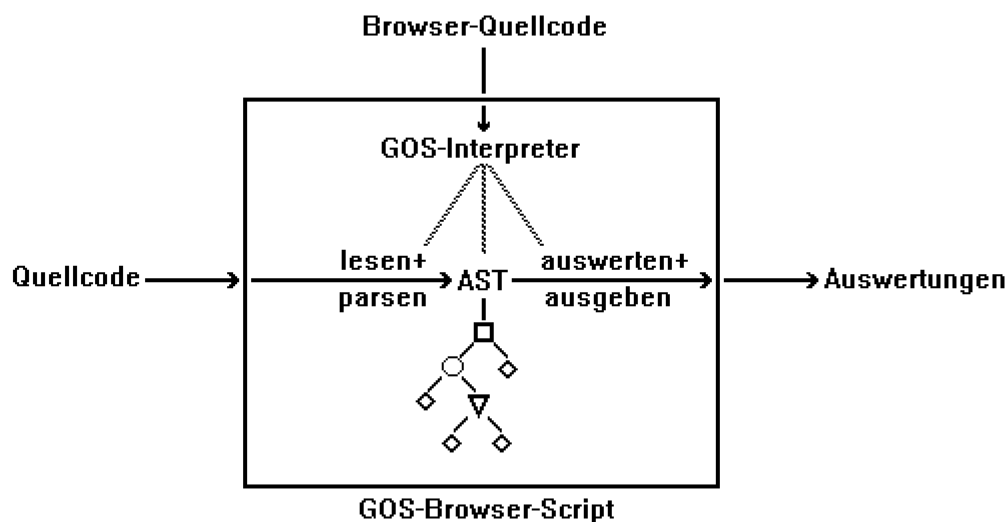


Abbildung 3.1: Ablauf

Schritt 1 - Script startet GOS-Browser
Nach dem Aufruf des Scripts mit den entsprechenden Argumenten (GOS-Programme und Auswertungsbefehle) wird der GOS-Interpreter mit dem GOS-Browser-Programm *gosbrowser.gos* aufgerufen. Die hierzu notwendigen Aufrufparameter für den GOS-Interpreter werden gleichfalls mitübergeben. Die dem Script übergebenen Argumente werden an den GOS-Browser weitergereicht.

Schritt 2 - Verarbeiten der Argumente
Nun beginnt der GOS-Browser mit seiner Arbeit, indem die übergebenen Argumente der Reihe nach gelesen und ausgewertet werden. Nun gibt es zwei Möglichkeiten: entweder wird das Argument als

Befehl oder als zu verarbeitende Datei erkannt. (Auswertungsbefehle beginnen mit einem Bindestrich, also z.B. -ao).

Handelt es sich um eine Dateiangabe, so wird die Datei als GOS-Programm interpretiert und mit Hilfe des Parsers verarbeitet. Falls mehrere Dateien angegeben wurden, so werden diese der Reihe nach zu den bereits verarbeiteten Dateien hinzugefügt. Mehrere Programme werden also so behandelt, als wäre deren Inhalt in einer einzigen Datei enthalten. Lediglich für Auswertungen macht diese Anordnung einen Unterschied, denn die Auswertungen beruhen immer auf den derzeit vorhandenen Vorrat an verarbeiteten Dateien. Dieser Effekt kommt also nur dann zum Tragen, wenn Auswertungsbefehle zwischen Dateiangaben angeordnet sind. Sollte der Parser bei der Verarbeitung einer Datei einen Fehler entdecken, so wird der gesamte Verarbeitungsvorgang abgebrochen und das Programm mit einem Fehler beendet. Dies liegt daran, daß der Parser in dieser vorliegenden Version über keinen geeigneten Fehlerbehandlungsmechanismus verfügt, denn der zu verarbeitenden Quellcode muß nach Vorraussetzung zumindest syntaktisch korrekt sein. Es ist nicht Aufgabe des Browsers, Programme auf Korrektheit zu prüfen, sondern syntaktisch einwandfreie Programme zu analysieren.

Wurde das Argument als Auswertungsbefehl erkannt, so ruft der GOS-Browser das entsprechende Programmstück für die Auswertung aus. Die Auswertung stützt sich dabei, wie bereits oben erwähnt, auf die bis dahin verarbeiteten GOS-Programme. Die geforderten Informationen werden aus dem abstrakten Syntaxbaum entnommen, welcher durch den Parser erzeugt wurde. Dieser Vorgang wird im Schritt 2b beschrieben. Die Ergebnisse werden dann anschließend ausgegeben. Unbekannte Befehle werden ignoriert und mit einer entsprechenden Warnung quittiert.

Schritt 2a - Parsen der GOS-Programme

Soll eine Datei mit Hilfe des Parsers verarbeitet werden, so wird zuerst ein Objekt erzeugt, welches den Eingabestrom über die angegebene Datei repräsentiert. Damit wird der Lexer versorgt, welcher, wie sein Name schon andeutet, die lexikalische Analyse des Eingabestroms übernimmt. Der damit erzeugte Strom von Terminalen, bestehend aus Bezeichner, Zahlen usw., wird dem Parser zugeführt. Dieser wiederum versucht nun, anhand der gegebenen GOS-Grammatik einen abstrakten Syntaxbaum aufzubauen. Dieser Versuch schlägt fehl, wenn das zu verarbeitende GOS-Programm einen Syntaxfehler enthält. Konnte das Programm erfolgreich verarbeitet werden, wird der so erzeugte Teilbaum zum bereits bestehenden Syntaxbaum hinzugefügt. Dieser wird natürlich beim Programmstart mit einem leeren Baum initialisiert.

Schritt 2b - Auswertungen des AST

Um die gewünschten Auswertungen zu erhalten, muß der erzeugte abstrakte Syntaxbaum analysiert werden. Hierzu werden Algorithmen eingesetzt, die Aufgrund der Struktur eines Baumes auch rekursiv sein können. Die Auswertungen beziehen sich zumeist auf Klassen-, Objekt-Definitionen und Gofer-Definitionen. Aber auch zur Extrahierung struktureller Informationen sind Auswertungen vorhanden, wie zum Beispiel Auswertungen über die Vererbungsstruktur der vorhandenen Klassen.

Im folgenden Abschnitt wird nun die Anwendung des GOS-Browser erklärt. Ebenso werden die vorhandenen Auswertungsbefehle beschrieben.

3.3 Anwendung und Funktionsbeschreibung

In diesem Abschnitt wird der Aufruf des GOS-Browser sowie seine Auswertungsbefehle beschrieben. Ein Anwendungsbeispiel mit Quellcode und Auswertungen inklusive Auswertungsergebnisse wird in einem sparaten Abschnitt diskutiert.

3.3.1 Programmaufruf

Der Aufruf des GOS-Browsers erfolgt mit Hilfe des Scripts *gosbrowser*. Die allgemeine Form ist wie folgt:

```
gosbrowser <arg1> [ <arg2> [ <arg3> ... ] ]
```

Die Argumente arg1, arg2, arg3, ... stellen entweder Dateinamen oder Befehle dar. Letztere werden mit einem vorangestellten Bindestrich eingeleitet, also zum Beispiel **`-ao`** oder **`-db`**. Die Liste aller möglichen Befehle wird im folgenden beschrieben. Beispiele für denkbare Aufrufe sind:

```
gosbrowser prg.gos -ac -ao

gosbrowser prg1.gos prg2.gos -db MyClass -dmad MyClass
```

Im zweiten Beispiel handelt es sich bei **`MyClass`** nicht um einen Dateinamen, sondern um ergänzende Angaben zu den Befehlen **`-db`** und **`-dmad`**. Diese erwarten nämlich zusätzlich einen Klassennamen, um konkrete Auswertungen für eine bestimmte Klassen durchführen zu können.

3.3.2 Auswertungsbefehle

Wie oben bereits erwähnt, werden Befehle mit einem Bindestrich eingeleitet, um sie von Dateiangaben zu unterscheiden. Je nach Befehl können auch weitere Angaben wie zum Beispiel Klassennamen notwendig sein. Eine Kurzreferenz ist im Anhang B zu finden.

Die folgende kleine Grammatik gibt einen Überblick über alle Auswertungensbefehle. Es ist zu beachten, daß die Buchstaben eines Befehls unmittelbar hintereinander geschrieben werden müssen. Die Verwendung von Trennzeichen wird in der Grammatik explizit angegeben.

```
Befehl              ::=  "-" Befehlsgruppen

Befehlsgruppen      ::=  "a" BefehlsgruppeA
                       | "d" BefehlsgruppeD Trennzeichen Identifier
                       | "s" BefehlsgruppeS Trennzeichen Identifier

Trennzeichen        ::=  " " [ Trennzeichen ]


BefehlsgruppeA      ::=  "c" | "o" | GoferGruppe

BefehlsgruppeD      ::=  "b" | "s" | "m" Filter
```

```
BefehlsgruppeS     ::=  "c" Filter′ | "o" | GoferGruppe

GoferGruppe        ::=  "gd" | "gt" | "gc" | "gf" | "gi" | "gp" | "gx"


Filter             ::=  Filter′ [ "d" | "i" ]

Filter′            ::=  [ "m" | "a" ] [ "pr" | "pu" ]
```

Bezeichner
Das Nicht-Terminal **`Identifier`** wird nicht explizit erklärt. Es steht für Bezeichner von Definitionen, also zum Beispiel **`MyClass`** oder **`myObject`** und entspricht den Terminalen ***`varid`*** und ***`conid`*** der GOS-Grammatik. Siehe hierzu auch Abschnitt 4.3.

Befehlsgruppen
Es gibt die Befehlsgruppen A, D und S, wobei die Gruppen A und S in ihrem Aufbau sehr ähnlich sind. Befehle der Gruppe A werden zum Auflisten von Definitionen verwendet. Um Einzelheiten einer Klassendefinition zu erhalten, werden Befehle der Gruppe D eingesetzt. Für die Ausgabe von speziellen Definitionen sind die Befehle der Gruppe S zuständig. Eine genaue Beschreibung folgt weiter unten. Befehle der Gruppen D und S benötigen zusätzlich einen Definitionsnamen. Diese werden von den eigentlichen Befehlen durch ein oder mehrere Leerzeichen getrennt.

Filter
Auswertungen im Zusammenhang mit Klassen erlauben zumeist die Angabe von Einschränkungen für die definierten Attribute und Methoden. Es gibt die zwei Arten **`Kat`** und **`Kat′`**.

Bedeutung der optionalen Filter

`m|a`

- ∅ Sowohl Attribute als auch Methoden werden angezeigt
- **`m`** Es werden nur Methoden angezeigt
- **`a`** Es werden nur Attribute angezeigt

`pr|pu`

- ∅ Es werden bzgl. public, protected und private keine Einschränkungen gemacht
- **`pr`** Es werden nur public und protected Deklarationen angezeigt
- **`pu`** Es werden nur public Deklarationen angezeigt

`d|i`

- ∅ Es werden alle, auch die geerbten Deklarationen der angegebenen Klasse betrachtet
- **`d`** Es werden nur die Deklarationen der angegebenen Klasse betrachtet
- **`i`** Es werden nur die Deklarationen der Oberklassen der angegebenen Klasse betrachtet

Befehlsbeschreibung

♦ **Befehlsgruppe A „list all definitions..."**
Befehle der Kategorie A listen Namen aller Definitionen einer bestimmten Art auf. So gibt z.B. **`-ac`** die Namen aller definierten Klassen und **`-agd`** die Namen aller definierten Gofer-Datentypen aus. Keiner der Befehle in dieser Gruppe benötigt weitere Argumente.

Folgende Definitionsarten werden unterstützt:

`c`	Klassendefinitionen (**`CLASS`**)
`o`	Definitionen global bekannter Objekte (**`OBJECT`**)
`gc`	Definitionen von Gofer-Typ-Klassen (**`class`**)
`gd`	Definitionen von Gofer-Datentypen (**`data`**)
`gf`	Definitionen von Gofer-Funktionen
`gi`	Definitionen von Gofer-Typ-Instanzen (**`instance`**)
`gp`	Definitionen von Gofer-Primitiven (**`primitive`**)
`gt`	Definitionen von Gofer-Typsynonymen (**`type`**)
`gx`	Gofer-Infix-Definitionen (**`infix`**)

♦ **Befehlsgruppe D „extract details"**
Um bestimmte Einzelheiten einer Klassendefinition zu erhalten, werden Befehle der Gruppe D eingesetzt. Beispielsweise gibt der Befehl **`-ds MyClass`** alle Unterklassen der Klasse **`MyClass`** aus. Falls die angegebene Klasse nicht definiert worden ist, wird eine entsprechende Warnung ausgegeben. Für die Variante **`-dm`** sind zusätzlich noch Filter möglich, um die Ausgabe auf bestimmte Elemente zu beschränken. Diese wurden weiter oben beschrieben. Folgende Befehle werden unterstützt:

`-db` *`Identifier`*
Dieser Befehl gibt alle Oberklassen der angegebenen Klasse aus. Es werden nicht nur die direkten Oberklassen ermittelt, sondern alle Klassen bis hin zu den Wurzeln des Vererbungsgraphen. Die direkten Oberklassen werden zusätzlich gesondert ausgegeben.

`-ds` *`Identifier`*
Dieser Befehl gibt alle Unterklassen der angegebenen Klasse aus. Es werden nicht nur die direkten Unterklassen ermittelt, sondern auch alle indirekten Unterklassen. Die direkten Unterklassen werden zusätzlich gesondert ausgegeben.

`-dm["d"|"i"]["m"|"a"]["pr"|"pu"]` *`Identifier`*
Attribute und Methoden einer Klasse werden mit diesem Befehl ausgegeben. Die Filter sind bereits weiter oben beschrieben worden.

♦ **Befehlsgruppe S „show definition“**
Befehle der Gruppe S werden zum Ausgeben einzelner Definitionen verwendet, so daß z.B. mit dem Befehl **`-so myObject`** die Definition des Objektes **`myObject`** ausgegeben werden kann. Es werden ebenfalls die Definitionsarten der Befehlsgruppe A unterstützt. Für die Variante **`-sc`** sind zusätzlich noch Filter möglich, um die Ausgabe auf bestimmte Elemente zu beschränken. Diese wurden weiter oben beschrieben. Die Syntax lautet wie folgt:

`-sc["m"|"a"]["pr"|"pu"]` ***`Identifier`***

3.4 Anwendungsbeispiel

In diesem Abschnitt werden mehrere Anwendungsbeispiel des GOS-Browsers aufgeführt. Die zugrundeliegenden GOS-Programme *example1.gos* und *example2.gos* sind im Anhang E zu finden.

Beispiel 1

Aufruf

```
example1.gos -ac example2.gos -ac -db Square -db Cycle
```

Ergenis

```
  All Classes
     GraphicObject,PolygonObject,Rectangle,RectangleObject,
     Square,Triangle

  All Classes
     Cycle,Ellipse,EllipseObject,GraphicObject,
     PolygonObject,Rectangle,RectangleObject,Square,
     Triangle

  Direct Base Classes of Square
     RectangleObject
  All Base Classes of Square
     GraphicObject,PolygonObject,RectangleObject

  Direct Base Classes of Cycle
     EllipseObject
  All Base Classes of Cycle
     EllipseObject,GraphicObject
```

Beispiel 2

Aufruf

```
example2.gos -db Cycle -dma Cycle -dmmpui Ellipse -dmd Ellipse
```

Ergenis

```
Direct Base Classes of Cycle
   EllipseObject
All Base Classes of Cycle
   EllipseObject

All attributes of Cycle
   center,radius

Inherited Public Methods of Ellipse
   area,isCycle,move,xRadius,#
   yRadius

All Defined Methods and Attributes of Rectangle
                                  w,h
   NEW,height,width
```

Kapitel 4

GOS-Grammatik

In diesem Kapitel wird die vom Parser verwendete GOS-Grammatik vorgestellt. Da sich die Sprache GOS in einen funktionalen Teil und in einen OO-Teil gliedern läßt, werden diese Teile getrennt vorgestellt. Der funktionale Teil ist weitgehend mit Gofer identisch. Darauf aufbauend wurde der OO-Teil von GOS definiert. Das Zusammenwirken dieser beiden Teile wird ebenfalls in diesem Kapitel beschrieben.

Ein weiterer Schwerpunkt ist die sogenannte Layout-Rule von Gofer. Diese Regel erlaubt es, Strukturierungszeichen (Semikola und geschweifte Klammern) im Code wegzulassen, falls gewisse Strukturierungsregeln eingehalten werden. Im Zuge der Integration von Gofer wurde diese Regel auf GOS erweitert. Die Layout-Rule wird in einem eigenen Punkt diskutiert.

Für die ursprüngliche Definition der Gofer-Grammatik bzw. der Layout-Rule sei auf [9] bzw. auf die Anhänge C und D verwiesen.

4.1 Beschreibung der Grammatik

Im folgenden wird nun das Zusammenspiel des funktionenen Teils und des OO-Teils der GOS-Grammatik beschrieben. Zu diesem Zweck ist es sinnvoll, zuerst die hier verwendete Gofer-Grammatik zu erläutern, welche nur unwesentlich von der ursprünglichen Definition abweicht.

4.1.1 Gofer-Grammatik

Ein Gofer-Programm besteht im wesentlichen aus Datentyp- und Funktions-Definitionen, welche beim Auswerten von Ausdrücken zur Anwendung kommen. Es gibt folgende Arten von Definitionen, welche im folgenden als Top-Level-Definitionen bezeichnet werden:

- Datatype-Definition
- Type-Definition
- Funktions-Definition
- Class- und Instance-Definition
- Infix-Definition

- Primitive-Definition

Für eine ausführliche Einführung in Gofer sein auf [7] bzw. [9] verwiesen.

♦ Datatype-Definition
Die Datatype-Definition erlaubt die Einführung eines neuen Datentyps. Ein Beispiel wurde bereits im Abschnitt 2.1.2 gegeben.

♦ Typ-Definition (type synonym)
Typ-Definitionen ermöglichen die Einführung von Abkürzungen für beliebige, i.A. komplexere Datentypen. Zum Beispiel könnte für binäre Bäume mit Elementen vom Typ Int der folgende (abkürzende) Typ eingeführt werden:

```
type IntTree  =  Tree Int
```

♦ Funktions-Definition
Funktions-Definitionen führen neue Funktionen ein. Siehe hierzu auch Abschnitt 2.1.3 und Abschnitt 2.1.4.

♦ Typ-Klassen und Typ-Instanzen
Der Begriff der Klasse in Gofer ist verschieden von dem Klassenbegriff, welcher durch den OO-Teil eingeführt wird. Eine Klassendefinition in Gofer ist die Konstruktion einer Typ-Klasse bestehend aus zusammengehörigen Funktionen, welche mit einer Instanz-Definition für ausgewählte Typen instanziiert werden kann. Die Funktionen werden somit überladen. Als Beispiel soll hier die Typ-Klasse Eq dienen. Diese ist deklariert als Zusammenfassung der Funktionen **==** und **/=** zum Vergleichen von zwei Werten des selben Typs auf Gleichheit bzw. Ungleichheit. Die Funktion **/=** ist mit Hilfe der not-Funktion auf die Funktion **==** abgestützt.

```
class Eq a where
   (==), (/=)  ::  a -> a -> Bool
   x /= y       =  not (x == y)
```

Nun kann die Typ-Klasse Eq z.B für den Typ Char instanziiert werden. Hierzu muß die Funktion **==** definiert werden, da die Implementierung in der Typ-Klassendefinition nicht angegeben ist, wohingegen die Funktion **/=** bereits in der Typ-Klassendefinition definiert wurde und somit automatisch verfügbar ist, d.h. geerbt wird.

```
instance Eq Char where
   c == d  =  ord c == ord d
```

In diesem Beispiel wird also auf die Typ-Instanz Eq Int (Ausdruck **`ord c == ord d`**) aufgebaut. Für die oben angegebene Datenstruktur **`Tree a`** kann ebenfalls eine Typ-Instanz Eq definiert werden:

```
instance Eq a => Eq Tree a where
   Lf a       == Lf b        =  a == b
   a1 :^: b1 == a2 :^: b2   =  (a1 == a2) && (b1 == b2)
   _          == _           =  False
```

♦ **Infix-Definition**

Mit der Infix-Definition kann die Assoziativität und Priorität einzelner Operatoren und Funktionen festgelegt werden. Hierzu stehen die **`infix`**-, **`infixl`**- und **`infixr`**-Definitionen zur Verfügung.

Die Priorität regelt dabei die Bindungsstärke der einzelnen Operatoren und Funktionen. Da beispielsweise der Operator „**`*`**“ eine höhere Priorität besitzt als der Operator „**`+`**“, wird der Ausdruck **`7 + 3 * 2`** so behandelt, als wäre dieser wie **`7 + (3 * 2)`** explizit geklammert. Die Prioritäten der üblichen Operatoren und Funktionen werdem im standard prelude definiert.

Die Assoziativität regelt die Reihenfolge der Abarbeitung gleicher Operatoren. Zum Beispiel ist der Ausdruck **`9 - 5 - 1`** dem Ausdruck **`(9 - 5) - 1`** äquivalent und ergibt somit den Wert **`3`**, da der Operator „**`-`**“ im standard prelude als links-assoziativ definiert ist. Wäre dieser rechts-assoziativ, so würde dies dem Ausdruck **`9 - (5 - 1)`** entsprechen und die Auswertung ergäbe den Wert **`5`**. Es gibt auch die Möglichkeit, einen Operator oder eine Funktion als „nicht-assoziativ“ zu definieren. In diesem Fall könnte der Audruck **`9 - 5 - 1`** auf Grund der Mehrdeutigkeit nicht ausgewertet werden und würde als Syntaxfehler betrachtet werden.

```
...
infixr 8 ^
infixl 7 *
infix  7 /, 'div', 'rem', 'mod'
infixl 6 +, -
...
```

♦ **Primitive-Definition**

Die tatsächliche Verarbeitung eines Gofer-Programms wird in sogenannten Primitiven erledigt. So gibt es Primitiven zum Addieren, Subtrahieren und Vergleichen von Zahlen. Mit der Primitive-Definition können solche Primitives an Operatoren und Funkionen gebunden werden, so daß die Primitiven über den Aufruf dieser erreichbar sind.

```
primitive ord "primCharToInt" :: Char -> Int
primitive chr "primIntToChar" :: Int -> Char
```

Sind nun Gofer-Definitionen gegeben, so können diese mit Ausdrücken angewendet werden. Der Gofer-Interpreter bietet die Möglichkeit, solche Audrücke interaktiv einzugeben und auswerten zu lassen. Zwar werden Audrücke auch in Funktions-Definitionen verwendet, jedoch werden diese Ausdrücke sozusagen

nur passiv, d.h. erst bei Anwendung der Funktion ausgewertet. Es gibt dabei aber keine Möglichkeit, Ergebnisse zwischenzuspeichern.

Diese Art der interaktive Anwendung kommt bei GOS nicht zur Ausführung. Stattdessen werden diese Ausdrücke in GOS-Anweisungen wie zum Beispiel Zuweisungen oder FOR-Schleifen verwendet.

4.1.2 OO-Teil der GOS-Grammatik

Die Definition einer Klasse ist eine Top-Level-Definition, da die Klassendefinition auf einer zur Top-Level-Definition von Gofer gleichwertigen Ebene steht. Die Definition einer Klasse könnte also zum Beispiel zwischen zwei Datatype-Definitionen stehen.

Die Objekt-Definition, welche durch den OO-Teil eingeführt wird, stellt ebenfalls eine Top-Level-Definition dar. Diese Definition von global bekannten Objekten ist die einzige Möglichkeit, Größen mit globaler Sichtbarkeit einzuführen.

Die Klassendefinition selbst besteht aus

- einem Klassennamen,
- der Angabe von keiner, einer oder mehreren Oberklassen,
- der Definition von Attributen und
- der Definition von Methoden.

♦ **Klassenname**
Durch den Klassennamen wird ein neuer Datentyp eingeführt. Da sich GOS auf das Gofer-Typsystem abstützt, wird hierdurch auch ein Gofer-Typ für die Identifikatoren auf die Objekte definiert.

♦ **Oberklassen**
Die Klassendefinition erlaubt auch die Angabe mehrerer Oberklassen. Somit ist Multiple-Inheritance möglich.

♦ **Attribut**
Eine Attribut-Definition führt eine Objekt-lokale Variable ein. Alle Attribute zusammen legen den Zustand eines Objektes fest. Die Attribute sind getypt, wobei hierfür Gofer-Typen verwendet werden. Da durch eine Klasse auch ein entsprechender Gofer-Typ eingeführt wird, kann ein Attribut auch einen Klassen-Typ besitzen.

♦ **Methoden**
Die Methoden einer Klasse definieren das Verhalten der Objekte dieser Klasse. Jede Methode kann Parameter und lokale Variablen besitzen sowie ein Ergebnis liefern, wobei dann sowohl Parameter und lokale Variablen als auch das Ergebnis getypt sein müssen.

♦ **Zugriffsregelung**
Attribut- und Methoden-Definitionen müssen entweder *public*, *protected* oder *private* deklariert werden. Dabei bedeutet *public*, daß diese Definitionen für „jedermann“ öffentlich sind, daß sie also an jeder

Stelle im Programm ohne Einschränkung angewendet werden können. Die Angabe von *private* hingegen erlaubt die Anwendung dieser Definitionen nur in Methoden der eigenen Klasse. Wenn also das Attribut A in der Klasse K als *private* deklariert wurde, so kann dieses Attribut in einer Methode der Klasse K ausgelesen werden, wohingegen in einer Methode der Klasse K' dieses Attribut nicht gelesen werden kann. Die Deklaration als *protected* ist eine erweiterte Form von *private*. Die Sichtbarkeit wird dabei zusätzlich auf alle Unterklassen ausgeweitet. Im obigen Beispiel heißt das, daß das Attribut A gelesen werden kann, falls K' eine Unterklasse von K ist. Trifft dies nicht zu, wird der Zugriff verweigert.

♦ **Instanziierung**
Neue Objekte werden durch den Konstruktor NEW erzeugt. Dieser wird im wesentlichen wie eine übliche Methode definiert, kann also auch Parameter besitzen. Jedoch ist die Angabe eines Ergebnistyps nicht möglich, da die einzige Aufgabe des Konstruktors darin besteht, das neu erzeugte Objekt in einen definierten Zustand zu versetzen. Jede Klasse muß einen Konstruktor definieren.

♦ **Methodenrümpfe**
Die Rümpfe der Methoden bilden den aktiven Teil von GOS. Jeder Rumpf besteht aus einer mitunter leeren Sequenz von Anweisungen, welche in folgende Gruppen eingeteilt werden können:

- Kontrollstrukturen
- Methodenaufruf
- Instanziierung
- Zuweisung

♦ **Kontrollstrukturen**
Wie von anderen imperativen Programmiersprachen bekannt, gibt es auch in GOS Kontrollstrukturen wie IF-THEN-ELSE-ENDIF und WHILE-DO-ENDDO.

```
IF (a < b) && (c == d) THEN
    ...
ELSE
    ...
ENDIF;

WHILE i > 0 DO
    ...
ENDDO;
```

Die Kontrollstrukturen können natürlich geschachtelt werden.
Die FOR-Schleife gibt es in zwei Varianten. Die erste ist die bekannte zählende Form mit einer Laufvariablen.

```
FOR i := 0 TO 10 DO
    ...
ENDDO;
```

Die zweite Form erlaubt das Iterieren über eine Liste, wobei die Laufvariable sukzessive mit den Elementen der Liste belegt wird. Unter Liste ist hier eine Liste im Sinne von Gofer gemeint, so daß in diesem Fall eine Verzahnung mit Gofer auftritt.

```
FOR element IN liste DO
    ...
ENDDO;
```

Die RETURN-Anweisung erlaubt das Verlassen einer Methode an jeder beliebigen Stelle im Rumpf. Falls die Methode ein Ergebnis liefert, muß die RETURN-Anweisung dieses Ergebnis zurückgeben.

```
IF a < b THEN
   RETURN c;
ENDIF;
```

Um Anweisungen von der Klasse eines Objekts abhängig machen zu können, wurde die TYPEIF-THEN-ELSE-ENDTYPEIF-Anweisung eingeführt. Im folgenden Beispiel wird also der THEN-Teil der Kontrollstruktur nur dann ausgeführt, wenn das Objekt von der Klasse *GraphicObject* oder einer seiner Unterklassen ist.

```
TYPEIF obj IS GraphicObject THEN
    ...
ENDIF;
```

- **Methodenaufruf**

Der OO-Teil von GOS führt den Methodenaufruf als Anweisung ein. Liegt ein Objekt als Attribut, Parameter oder lokale Variable vor, so kann von diesem Objekt eine Methoden ausgeführt werden. Damit auch Methoden vom Objekt selbst (für welches gerade die aktuelle Methode ausgeführt wird) aufgerufen werden können, ist das Objekt innerhalb der Methodenrümpfe durch die Pseudo-Variable **`self`** erreichbar.

```
obj!doSomething();

self!doSomethingElse();
```

- **Instanziierung**

Die Erzeugung neuer Objekte, die Instanziierung, wird durch den Aufruf des NEW-Konstruktors erreicht. Der NEW-Aufruf wird allerdings an die Klasse gesendet, wobei als Ergebnis das neu erzeugte Objekt zurückgegeben wird.

```
Cycle!NEW(mx,my,r);

Stack!NEW();
```

♦ **Zuweisung**
Die Zuweisung erlaubt die Belegung von Attributen und lokalen Variablen mit Ergebnissen aus Methodenaufrufen, Instanziierung und Ausdrücken. Die Ausdrücke sind Audrücke im Sinne Gofers, so daß hiermit die Brücke zwischen OO-Teil und Gofer geschlagen wird. Alle Möglichkeiten von Gofer, Ausdrücke zu bilden, können somit ausgeschöpft werden. Daß ein Ausdruck nicht als Anweisung erlaubt ist und immer mit einer Zuweisung verwendet werden muß liegt daran, daß Gofer-Ausdrücke frei von Seiteneffekten sind und deshalb die bloße Ausführung keine Auswirkungen hat. Erst wenn das Ergebnis eines Ausdrucks verwertet, also zum Beispiel in einem Attribute gesichert wird, ergibt die Ausführung auch einen Sinn.

```
graphicObject := Cycle!NEW(mx,my,r);

result := obj!doSomething();
```

4.1.3 Zusammenfassung

Betrachtet man die Struktur der GOS-Grammatik aus einem abstrakteren Blickwinkel, so läßt sich folgendes feststellen:

- Gofer bildet den Kern von GOS, wobei alle Möglichkeiten von Gofer voll ausgeschöpft werden können. Lediglich der interaktive Teil, welcher die Auswertung einzelner Ausdrücke zulässt, entfällt.
- Der OO-Teil führt den Klassenbegriff ein. Dabei werden insbesondere die Konzepte für (Mehrfach-) Vererbung, Kapselung, Polymorphismus und später Bindung eingeführt.
- Der Anweisungsteil wird durch Methodenrümpfe gebildet. Neben den üblichen und speziellen Kontrollstrukturen wird auch der Methodenaufruf eingeführt, welcher die Kommunikation mit anderen Objekten sichert. Zu den Anweisungen zählt auch die Instanziierung, d.h. die dynamische Erzeugung von Objekten.
- Die Zuweisung an Attribute und lokale Variablen ermöglicht die persistente Speicherung von Werten und somit auch die Änderung des Zustandes eines Objekts. Diese werden allerdings letztendlich durch Gofer-Ausdrücke gebildet. Selbst die Zuweisung des Ergebnisses eines Methodenaufrufs ist auf einen Gofer-Ausdruck zurückzuführen, da dieser durch eine RETURN-Anweisung zurückgegeben wird. Die einzige Ausnahme bildet hier die Instanziierung, da sie einen Wert liefert ohne einen Gofer-Ausdruck zu verwenden.

Zusammenfassend kann gesagt werden, daß der Gofer-Teil den funktionalen, berechnenden Anteil und der OO-Teil den strukturierenden, steuernden Anteil darstellt.

4.2 Notation

Die kontextfreie Grammatik ist in einer Variante der Backus-Naur-Form angegeben, welche sich an die Notation anlehnt, die zur Definition der Gofer-Grammatik gewählt wurde. Die folgende informelle Beschreibung verdeutlicht die angewandte Form.

♦ **Nicht-Terminale**
Nicht-Terminale werden wie gewöhnliche Bezeichner verwendet, also zum Beispiel **`Command`** oder **`IdentList`**.

♦ **Terminale**
Terminale werden in Anführungszeichen gesetzt, also zum Beispiel **`"OBJECT"`** oder **`";"`**.

♦ **Produktionen**
Produktionen sind von der Form

```
NichtTerminal ::= Produktionsdefinition
```

wobei die rechte Seite (Produktionsdefinition) aus Terminalen und Nicht-Terminalen Zeichen besteht, die mit den folgenden „Operatoren“ verknüpft sein können.

♦ **Konkatenation**
Die Konkatenation erfolgt durch Hintereinanderstellung (ohne zusätzliches Operatorzeichen).

`a b`	*a* und dann *b*
`a b c`	*a* und dann *b* und dann *c*

...

♦ **Alternative**
Die Alternative wird durch einen senkrechten Strich angezeigt. Es gilt die Regel, daß die Konkatenation stärker bindet als die Alternative.

**`a	b`**	*a* oder *b*
**`a	b	c`**	*a* oder *b* oder *c*

...

♦ **Gruppierung**
Zur expliziten Regelung der Bindung kann die Gruppierung eingesetzt werden, um z.B. für eine Alternative gegenüber einer Konkatenation eine höhere Bindung zu erreichen.

`( a )`	Regelung der Bindung
`a ( b | c ) d`	aber: a b	c d ≡ (a b)	(c d)

♦ **Option**
Die Option kann zwar durch eine Alternative mit ε (leere Produktion) realisiert werden, erleichtert jedoch die Darstellung und erhöht die Lesbarkeit der dargestellten Grammatik.

`[ a ]` *a* oder ε

♦ **Wiederholung**
Die Wiederholung gestattet die Bildung von Listen, wobei die Elemente beliebig oft auftreten können. Auch die leere Liste ist erlaubt. Zugunsten der Einfachheit wurde auf die Notation von Listen verzichtet, welche mindestens ein Element enthalten müssen.

`{ a }*` ε oder *a* oder *a a* oder *a a a* ...

4.3 Definition der GOS-Grammatik

Die GOS-Grammatik setzt sich aus dem Gofer-Teil und dem OO-Teil zusammen. Da sich beide Teile im Top-Level-Bereich vermengen, wird dieser Schwerpunkt in der Darstellung der Grammatik gesondert behandelt.

Der Parser erkennt die gesamte Gofer-Grammatik außer den Produktionen *module* und *interp*, welche für Gofer-Quellcode-Dateien und für den Kommandozeilen-Interpreter gedacht sind. Die erste wird durch die Struktur von GOS-Quellcode-Dateien ersetzt. Letztere ist für das GOS-System völlig unerheblich.

Zu bemerken ist die Tatsache, daß die GOS-Grammatik für Klassen- und Objektdefinitionen fordert, daß diese immer mit einem Semikolon abgeschlossen werden, wohingegen in der ursprünglichen Gofer-Grammatik das Semikolon für Definitionen der Top-Level-Ebene als Trennzeichen zwischen diesen Definitionen dient. Da in der vorliegenden Grammatik auch Klassen- und Objektdefinition zur Top-Level-Ebene gehören, wurde diese Regel auch auf die neuen Top-Level-Definitionen erweitert. Diese bedeutet nun, daß das Semikolon sowohl als Trennzeichen zwischen allen Top-Level-Definitionen (Klasse, Objekt, Gofer-Top-Level) dient, als auch die Klassen- und Objektdefinitionen abschließt. Dies betrifft letztendlich auch die Layout-Rule von Gofer, welche in einem eigenen Punkt weiter unten beschrieben wird. Diese sorgt nämlich in der Regel dafür, daß das trennende Semikolon automatisch eingefügt wird. In der Praxis führt dies also dazu, daß nur nach einer Klassen- oder Objektdefinition ein Semikolon gesetzt werden muß.

Da sich das GOS-Typsystem vollständig auf das Gofer-Typsystem abstützt, werden alle Größen mit Gofer-Typen deklariert. Somit ist das Nicht-Terminal **`GOSType`** mit dem Nicht-Terminal **`type`** (aus der Gofer-Grammatik) gleichzusetzen. Dies wird durch eine entsprechende Regel ausgedrückt.

Ein weiterer Berührungspunkt ist die Verwendung von Gofer-Ausdrücken. Alle GOS-Ausdrücke sind entweder Methodenaufrufe, Instanziierungen oder Gofer-Ausdrücke. Somit ist das Nicht-Terminal **`GOSExp`** mit dem Nicht-Terminal **`exp`** (aus der Gofer-Grammatik) gleichzusetzen. Dies wird ebenfalls durch eine entsprechende Regel ausgedrückt.

Zuletzt sei noch erwähnt, daß die in der GOS-Grammatik verwendeten Terminale ***varid*** und ***conid*** (für die Produktionen **`Ident`** und **`ClassIdent`**) nicht die entsprechenden Produktionen der Gofer-Grammatik bezeichnen, sondern die Terminale ***varid*** und ***conid***. Unglücklicherweise wurden diese Terminale und die Produktionen **`varid`** und **`conid`** in der Gofer-Grammatik gleich benannt, so daß eine Unterscheidung nur anhand der kursiven Schrift möglich ist. Die Terminale werden in [9] wie folgt beschrieben:

varid Bezeichner, welcher mit einem kleinen Buchstaben beginnt, z.B. **obj**.
conid Bezeichner, welcher mit einem großen Buchstaben beginnt, z.B. **Tree**.

Die Produktionen (aus der Gofer-Grammatik) für **varid** und **conid** hingegen lauten wie folgt:

```
varid              ::=  varid | "(" varop ")"

conid              ::=  conid | "(" conop ")"
```

Die verwendeten Terminale ***varop*** und ***conop*** werden folgendermaßen beschrieben:

varop Operator, welcher nicht mit einem Doppelpunkt beginnt, z.B: **+**.
conop Operator, welcher mit einem Doppelpunkt beginnt, z.B. **:^:**.

Der Unterschied zwischen dem Terminal ***varid*** und dem Nicht-Terminal **varid** besteht also darin, daß durch das Nicht-Terminal zusätzlich die Verwendung von ***varop***-Terminalen als varid' s ermöglicht wird. So kann z.B. **(+)** als varid behandelt werden. Ähnliches gilt für das Terminal ***conid*** und das Nicht-Terminal **conid**.

Die Liste aller verwendeten Terminale für die GOS-Grammaitk ist in Abschnitt 5.3.5 beschrieben.

Nach diesen Überlegungen läßt sich die Grammatik nun in folgende Schwerpunkte gliedern:

- Top-Level des Gofer- und OO-Teils
- Klassen- und Objekt-Definitionen
- Anweisungen (Commands)
- Zusammenführung von Typ, Ausdruck u.ä.
- leicht abgewandelte Gofer-Grammatik

Das Startsymbol ist das Nicht-Terminal **GOSModule** und wird im Bereich Top-Level aufgeführt.

♦ Top-Level

Der Top-Level-Bereich faßt die Klassen- und Objekt-Definitionen des OO-Teils und die Top-Level-Definitionen des Gofer-Teils, also die Datatype-, Type-, Funktions-, Class-, Instance-, Infix- und Primitive-Definitionen zusammen. Insbesondere wird die Verwendung von Semikola der Top-Level-Ebene geregelt.

GOS-Top-Level

```
GOSModule          ::=  "{" GOSTopDefns "}"            vgl. Regel module der
                                                        Gofer-Grammatik
```

GOSTopDefns	::=	GOSTopDefn	einzelne Definition oder Liste von durch Semikola getrennten Definitionen
	|	GOSTopDefn ";" GOSTopDefns	
GOSTopDefn	::=	GOSObjectDefn	Objekt-Definition
	|	GOSClassDefn	Klassen-Definition
	|	GoferTopDefn	Gofer-Definition

Gofer-Top-Level

GoferTopDefn	::=	"data" typeLhs "=" constrs	entspricht bis auf die Verkettung mit Semikola der Regel topdecls aus der Gofer-Grammatik
	|	"type" typeLhs "=" type	
	|	"infixl" [digit] op { "," op }*	
	|	"infixr" [digit] op { "," op }*	
	|	"infix" [digit] op { "," op }*	
	|	"primitive" prims "::" type	
	|	class	
	|	inst	
	|	decls	

♦ **Klassen- und Objektdefinition**

Dieser Teil beschreibt die Objekt- und Klassendefinition des OO-Teils. Letztere beinhaltet insbesondere die Attribut- und Methodendefinition.

Objekt-Definition

GOSObjectDefn	::=	"OBJECT" Ident ":=" NewCall ";"	Objekt-Definition

Klassen-Definition

GOSClassDefn	::=	"CLASS" ClassIdent [SubclassOf] ClassMemberDefns "ENDCLASS" ClassIdent ";"	Klassen-Definition
SubclassOf	::=	"SUBCLASSOF" ClassIdentList ";"	Liste der Oberklassen
ClassIdentList	::=	ClassIdent { "," ClassIdent }*	Liste von Klassennamen

Klassenrumpf, Zugriffskontrolle

ClassMemberDefns	::=	QualMemberDefns	Alle Members ohne Zugriffsregelung sind PUBLIC und können deshalb nur Methoden sein.
	|	MethodDefns QualMemberDefns	
QualMemberDefns	::=	{ QualMemberDefn }*	Gruppen von Members mit Zugriffsregelung.

```
QualMemberDefn     ::=  "PUBLIC" MethodDefns                    Nur Methoden können
                      | "PROTECTED" MemberDefns                 PUBLIC sein.
                      | "PRIVATE" MemberDefns

MethodDefns        ::=  { [ MethodDefn ] ";" }*                 Liste von Methoden

MemberDefns        ::=  { [ MemberDefn ] ";" }*                 Liste von Methoden und
                                                                Attributen

MemberDefn         ::=  MethodDefn                              Member ≡ Methode oder
                      | AttributeDefns                          Attribut
```

Attribut-Definition

```
AttributeDefns     ::=  IdentList "::" GOSType                  Attribut-Definitionen

IdentList          ::=  Ident { "," Ident }*                    Liste von Bezeichnern
```

Methoden-Definition

```
MethodDefn         ::=  MethodHeader                            Methoden-Definition
                        [ "VAR" LocalsList ]                    lokale Var., optional
                        "BEGIN"                                 Der Methodenrumpf
                           Commands                             besteht aus einer
                        "END"                                   mitunter leeren Menge
                                                                von Anweisungen

MethodHeader       ::=  Ident MethodParams ResultType "="       Kopf einer Methoden-
                      | "NEW" MethodParams "="                  bzw. Konstruktor-Def.

MethodParams       ::=  "(" [ ParamList ] ")"                   Parameter, optional

ParamList          ::=  Param { "," Param }*                    Liste von Parametern

Param              ::=  Ident "::" GOSType                      Parameter

ResultType         ::=  [ GOSType ]                             Ergebnistyp, optional

LocalsList         ::=  { IdentList "::" GOSType ";" }*         Liste der lokalen
                                                                Variablen
```

♦ **Commands**

Die diesem Teil werden die Kontrollstrukturen sowie der Methodenaufruf, die Instanziierung und die Zuweisung beschrieben.

Anweisungen

```
Commands           ::=  { [ Command ] ";" }*                    Liste von durch
                                                                Semikola getrennten
                                                                Anweisungen.
```

```
Command          ::=  Assignment                          Zuweisung,
                   |  BlockCommand                        Unterblock,
                   |  ForCommand                          For-Schleife,
                   |  IfCommand                           Bedingte Anweisungen,
                   |  MethodCall                          Methodenaufruf,
                   |  NewCall                             Instanziierung,
                   |  ReturnCommand                       Return-Anweisung,
                   |  TypeIfCommand                       Klassen-bedingte Anw.,
                   |  WhileCommand                        oder While-Schleife
```

Zuweisung

```
Assignment       ::=  Ident ":=" AssignedExp              Zuweisung an Attribut
                                                          oder Variable

AssignedExp      ::=  MethodCall                          Zuweisung von Methoden-
                   |  NewCall                             aufruf, Instanziierung
                   |  GOSExp                              oder Ausdruck
```

Unterblock

```
BlockCommand     ::=  "BEGIN"                             Anweisungen können zu
                         Commands                         Blöcken zusammengefasst
                      "END"                               werden.
```

Iteration

```
ForCommand       ::=  "FOR" ForHeader "DO"                Iteration
                         Commands
                      "ENDDO"

ForHeader        ::=  Ident ":=" GOSExp "TO" GOSExp       gezählte Iteration oder
                   |  Ident "IN" GOSExp                   Iteration über Liste
```

Bedingte Anweisungen

```
IfCommand        ::=  "IF" GOSExp "THEN"                  Bedingte Anweisungen
                         Commands
                      { "ELIF" GOSExp "THEN"              Liste von bedingten
                         Commands                         Alternativen
                      }*
                      [ "ELSE"                            Unbedingte Alternative
                         Commands                         optional
                      ]
                      "ENDIF"
```

Methodenaufruf, Instanziierung
Ein Methodenaufruf mit **`self`** als Empfänger ist durch das Nicht-Terminal **`MethodCall`** abgedeckt, da **`self`** nicht als Terminal behandelt wird, sondern eine implizit definierte lokale Variable ist. Somit ist **`self`** ein GOS-Ausdruck.

```
MethodCall          ::=  [ GOSExp ] "!" Ident Arguments          Methodenaufruf

NewCall             ::=  ClassIdent "!" "NEW" Arguments          Instanziierung

Arguments           ::=  "(" [ ArgumentList ] ")"                Argumente, optional

ArgumentList        ::=  GOSExp { "," GOSExp }*                  Liste von Ausdrücken
```

Return-Anweisung

```
ReturnCommand       ::=  "RETURN" [ GOSExp ]                     Return-Anweisung,
                                                                 Ergebnis ist optional
```

Klassenbedingte Anweisungen

```
TypeIfCommand       ::=  "TYPEIF" Ident "IS" ClassIdent "THEN"    Klassen-bedingte
                            Commands                              Anweisungen
                         [ "ELSE"                                 Alternative, optional
                            Commands
                         ]
                         "ENDTYPEIF"
```

While-Schleife

```
WhileCommand        ::=  "WHILE" GOSExp "DO"                     While-Schleife
                            Commands
                         "ENDDO"
```

♦ **Zusammenführung**
In diesem Teil wird die Zusammenführung von Typ, Ausdruck und Bezeichnern des Gofer-Teils und des OO-Teils beschrieben. Die Regeln **`type`** und **`exp`** werden in der Gofer-Grammatik beschrieben. Die Terminale ***`varid`*** und ***`conid`*** wurden bereits oben diskutiert.

GOSType ≡ Gofer-Typ

```
GOSType             ::=  type                                    aus Gofer-Grammatik
```

GOSExp ≡ Gofer-Ausdruck

```
GOSExp              ::=  exp                                     aus Gofer-Grammatik
```

Ident ≡ Terminal varid

Ident	::=	*varid*	Bezeichner mit kleinem Anfangsbuchstaben

ClassIdent ≡ Terminal conid

ClassIdent	::=	*conid*	Bezeichner mit großem Anfangsbuchstaben

⬩ Gofer-Grammatik
Die Gofer-Grammatik entspricht bis auf die Änderungen im Top-Level-Bereich der ursprünglichen Form. Deshalb sei hier nochmals auf die originale Fassung bzw. auf den Anhang C verwiesen. Die Änderungen betreffen die bereits oben beschrieben Nicht-Terminale **`module`** und **`interp`**. Die dritte und letzte Änderung betrifft das Nicht-Terminal **`topdecls`**, welches durch das Nicht-Terminal **`GoferTopDefn`** ersetzt wurde. Die listenbildende Alternative wurde entfernt, da dies durch das neue Nicht-Terminal **`GOSTopDefns`** erledigt wird. Werden also die Nicht-Terminale **`module`**, **`interp`** und **`topdelcs`** aus der originalen Fassung der Gofer-Grammatik entfernt und die obigen Produktionen hinzugenommen, so erhält man die vollständige GOS-Grammatik.

4.4 Layout

Dieser Abschnitt widmet sich der Gestalltung und Strukturierung von GOS-Quellcode. Hierfür gibt es die zwei Schwerpunkte Kommentare und die sogenannte Layout-Rule.

4.4.1 Kommentare

Kommentare dienen der informellen Beschreibung eines Programm hinsichtlich seines Zwecks und seiner Struktur. Aber auch Anmerkungen zum Entwicklungsprozeß können mit Hilfe von Kommentaren in den Quellcode mit einfließen.

Zu diesem Zweck gibt es zwei Möglichkeiten, Kommentare anzugeben. Dies sind Zeilenkommentare und geschachtelte Kommentare, welche sich auch über mehrere Zeilen hinweg erstrecken können.

⬩ Zeilenkommentar
Ein Zeilenkommentar wird mit den zwei Zeichen **`--`** eingeleitet und reicht bis zum Zeilenende. Innerhalb von Operator-Zeichen (zum Beispiel **`>-->`**) werden diese zwei Zeichen nicht als Kommentarbeginn interpretiert. So enthält die nachfolgende Zeile einen Zeilenkommentar

```
(xs ++ ys) -- xs
```

wohingegen die nächste Zeile keinen Zeilenkommentar enthält:

```
xs >--> ys >--> zs
```

♦ **Geschachtelter Kommentar**
Ein geschachtelter Kommentar beginnt mit den Zeichen `{-` und endet mit den Zeichen `-}`, wobei auch mehrere Zeilen dazwischen sein können. Der kürzeste geschachtelte Kommentar ist `{--}`. Wird ein Kommentar nicht beendet, so wird dies als syntaktischer Fehler betrachtet.

Geschachtelte Kommentare können, wie ihr Name bereits ausdrückt, auch geschachtelt werden. Deshalb wird

```
{- {- ... -} ... {- ... -} -}
```

als *ein* Kommentar betrachtet.

4.4.2 Layout-Rule

In diesem Abschnitt wird die gegenüber der ursprünglichen Version modifizierte Form der Layout-Rule beschrieben. Die Anwendung dieser Regeln wurde im Zuge der Integration von Gofer auf die Sprache GOS erweitert. Die Unterschiede zur Gofer-Version werden im Kapitel „Implementierung“ dargestellt.

Ein einleitendes Beispiel soll nun diese Layout-Rule erläutern. Hierzu wird die Funktion f definiert, welche lokale Definitionen beinhaltet. Dazu gehört insbesondere die Funktion g, welche selbst wiederum eine lokale Definition besitzt. Die Einrückungen drücken dabei die Struktur der Definition aus.

```
f x y = g (x + w) where
   g u = u + v where
      v = u * u
   w   = 2 + y
```

Die Definition der Funktion f enthält die zwei lokalen Definitionen

```
   g u = u + v where ...
   w = 2 + y
```

wobei die Definition

```
   v = u * u
```

zur Definition der Funktion g gehört. Die Grammatik von GOS erwartet jedoch, daß lokale Definitionen in geschweiften Klammern eingeschlossen und durch Semikola getrennt sind. Falls allerdings die öffnende geschweifte Klammer fehlt, kommt die Layout-Rule zum Einsatz. Diese sorgt dann dafür, daß abhängig von der Einrückung der nachfolgenden Zeilen automatisch geschweifte Klammern und Semikola in den Eingabestrom eingefügt werden. Die Anwendung der Layout-Rule endet nach dem Einfügen der schließenden geschweiften Klammer. Dabei ist zu beachten, daß die Layout-Rule, wie in diesem Beispiel, auch geschachtelt angewendet wird.

Das obige Beispiel würde also wie das nachfolgende Programmstück behandelt werden:

```
f x y = g (x + w) where
   {g u = u + v where
      {v = u * u
   };w    = 2 + y
}
```

Unter der expliziten Verwendung von geschweiften Klammern und Semikola könnte das selbe Programmstück auch in einer Zeile geschrieben werden:

```
f x y = g (x + w) where {g u = u + v where {v = u * u}; w   = 2 y}
```

Die Layout-Rule kommt auch im OO-Teil zur Anwendung. Dies bedeutet, daß zum Beispiel auch in einer Klassendefinition Semikola weggelassen werden können, weil sie durch die Layout-Rule eingefügt werden. Im wesentlichen betrifft dies die folgenden Punkte:

- Die Klassen- und Objektdefinitionen sind Top-Level-Definitionen. Aus diesem Grund wird in der Regel die Layout-Rule der „Top-Level-Ebene“ greifen. Dies wird an einem Beispiel weiter unten aufgezeigt. Die eingefügten Semikola werden jedoch nur in sehr seltenen Fällen einen Fehler verursachen, da die GOS-Grammatik in dieser Hinsicht sehr tolerant ist. Auftretende Fehler können allerdings durch einfache Quellcode-Umstrukturierungen beseitigt werden.
- Die Layout-Rule kann innerhalb von Gofer-Ausdrücken zur Anwendung kommen, zum Beispiel bei einer Zuweisung. Die Layout-Rule wird jedoch in diesem Fall bei korrekt formulierter Syntax auch sofort wieder beendet, so daß sich die Auswirkungen nicht über den Ausdruck hinaus fortsetzen können.

Definition der Layout-Rule

Die Layout-Rule läßt sich durch die folgenden sieben Regeln definieren:

1. Eine öffnende geschweifte Klammer "{" wird vor dem ersten Token am Anfang einer Datei oder nach den Schlüsselwörtern **where**, **let** bzw. **of** eingefügt, falls das folgende Token nicht selbst eine öffnende geschweifte Klammer ist. In diesem Fall wird die Layout-Rule aktiviert.
2. Ein Semikolon ";" wird vor dem ersten Zeichen nachfolgender Zeilen eingefügt, falls die Einrückung mit der Einrückung der Zeile übereinstimmt, in welcher die öffnende geschweifte Klammer eingefügt wurde. Dies gilt nur für nachfolgende Zeilen, zwischen denen sich keine Zeilen mit abweichender Einrückung befinden. Hierbei werden leere Zeilen oder Zeilen, die nur Kommentar enthalten, ignoriert. Siehe auch Punkt 5.
3. Die Layout-Rule endet bei einer Zeile, dessen Einrückung echt kleiner ist als die Einrückung der Zeile, in welcher die öffnende geschweifte Klammer eingefügt wurde. In diesem Fall wird eine schließende geschweifte Klammer eingefügt. Auch hierbei werden leere Zeilen oder Zeilen, die nur Kommentar enthalten, ignoriert. Siehe auch Punkt 5.
4. *Dieser Punkt wird zu Vergleichszwecken zur ursprünglichen Version ausgelassen.*

5. Wie bereits in den Punkten 2 und 3 erwähnt, beeinflussen leere Zeilen und Zeilen, welche nur Kommentar enthalten, nicht die Benutzung der Layout-Rule. Als leer wird eine Zeile betrachtet, wenn sie höchstens Leer- und Tabulatorzeichen enthält.
6. Für die Berechnung der Einrückung werden Tabulatorzeichen gesondert behandelt. Es wird dabei angenommen, daß ein Tabulator alle 8 Zeichen plaziert wird. Falls also ein Tabulator auftritt, werden bis zur nächsten Tabulator-Position Leerzeichen eingefügt.
7. Die Einrückung des Dateiende-Tokens ist 0. Somit wird automatisch eine schließende geschweifte Klammer am Dateiende angefügt, falls die Layout-Rule in der Top-Level-Ebene aktiv war, d.h. wenn am Dateianfang automatisch eine öffnende geschweifte Klammer eingefügt wurde.

Programmbeispiel
Das nachfolgende Programm demonstriert nochmals die Verwendung der Layout-Rule. Sein Inhalt ist unwesentlich. Auf der linken Seite steht die Version des Programms, die implizit die Layout-Rule ausnützt und so auch zur Anwendung kommen könnte. Die rechte Seite hingegen zeigt das Programm so, wie es nach Anwendung der Layout-Rule verarbeitet würde.

Das Beispielprogramm definiert die Klasse **`IntegerStack`** zur Repräsentation eines Stacks, der nur Integer-Werte aufnehmen kann. Zu seinen Methoden gehören **`push()`**, **`pop()`** und **`size()`** sowie der Konstruktor **`NEW()`**. Zur Implementierung verwendet diese Klasse den generischen Gofer-Datentyp **`Stack`**. Hierzu werden Funktionen definiert, die die funktionalen Pendants der oben genannten Methoden sind.

Das linke Programm nützt die Layout-Rule aus:

```
data Stack a = Empty
             | MkStack a (Stack a)

push :: a -> Stack a -> Stack a
push x s = MkStack x s

size :: Stack a -> Int
size s = length (stackToList s) where
   stackToList Empty         = []
   stackToList (MkStack x s) = x:xs where
      xs = StackToList s

pop :: Stack a -> (a, Stack a)
pop (MkStack x s) = (x, s)

CLASS IntegerStack
PROTECTED
   stack :: Stack Int;
PUBLIC
   NEW() =
   BEGIN
      stack := Empty;
   END;

   push(i :: Int) =
   BEGIN
      stack := push i stack;
   END;

   pop() Int =
   VAR popResult :: (Int, Stack Int);
   BEGIN
      IF (size stack) > 0 THEN
         popResult := pop stack;
         stack := snd popResult;
         RETURN fst popResult;
      ELSE
         error "Stack is empty"
      ENDIF;
   END;

   size() Int =
   BEGIN
      RETURN size stack;
   END;
ENDCLASS IntegerStack;
```

Das linke Programm würde wie das rechte Programm behandelt werden. In den mit ⇨ markierten Zeilen wurden Zeichen eingefügt:

```
⇨ {data Stack a = Empty
               | MkStack a (Stack a)

⇨ ;push :: a -> Stack a -> Stack a
⇨ ;push x s = MkStack x s

⇨ ;size :: Stack a -> Int
⇨ ;size s = length (stackToList s) where
⇨    {stackToList Empty = []
⇨    ;stackToList (MkStack x s) = x:xs where
⇨       {xs = StackToList s

⇨ }};pop :: Stack a -> (a, Stack a)
⇨ ;pop (MkStack x s) = (x, s)

⇨ ;CLASS IntegerStack
⇨ ;PROTECTED
     stack :: Stack Int;
⇨ ;PUBLIC
     NEW() =
     BEGIN
        stack := Empty;
     END;

     push(i :: Int) =
     BEGIN
        stack := push i stack;
     END;

     pop() Int =
     VAR popResult :: (Int, Stack Int);
     BEGIN
        IF (size stack) > 0 THEN
           popResult := pop stack;
           stach := snd popResult;
           RETURN fst popResult;
        ELSE
           error „Stack is empty“
        ENDIF;
     END;

     size() Int =
     BEGIN
        RETURN size stack;
     END;
⇨ ;ENDCLASS IntegerStack;
⇨ }
```

Kapitel 5

Implementierung

In diesem Kapitel werden die Klassen und Datenstrukturen für die Ablaufsteuerung, das Parsing und die Auswertungen beschrieben. Hierzu wird insbesondere der Zusammenhang zwischen der GOS-Grammatik und den verwendeten Strukturen des abstrakten Syntaxbaum aufgezeigt. Desweiteren wird in diesem Kapitel der Erzeugungsprozeß des GOS-Browser-Programms dargestellt, da dieser aus mehreren Schritten besteht. Aber auch die Funktionsweise der Auswertungen wird ausführlich erklärt.

5.1 Aufbau und Funktionsweise des GOS-Browsers

In diesem Abschnitt werden zuerst die Komponenten des GOS-Browsers vorgestellt. Jede Komponente wird durch seine Aufgaben und seiner Funktionsweise beschrieben. Eine detailiere Darstellung wird im Abschnitt 5.3 gegeben. Die Funktionsweise des GOS-Browsers und somit das Zusammenspiel dieser Komponenten wird sukzessive erklärt und am Schluß noch einmal zusammenfassend dargestellt.

Die Aufgaben des Programms lassen sich im wesentlichen in zwei große Schwerpunkte aufteilen. Der erste Teil beschäftigt sich mit dem Parsing und damit dem Aufbau des abstrakten Syntaxbaumes. Der zweite Teil ist für die Ablaufsteuerung und Verarbeitung der Kommandos und somit für die Auswertungen zuständig. Um diese Aufgaben zu lösen, werden die folgenden Komponenten eingesetzt:

- Ablaufsteuerung
- ParserManager
- FileReader
- Scanner
- Parser
- Abstrakter Syntaxbaum

Diese Komponenten werden nun im folgenden beschrieben. Das Zusammenwirken dieser Objekte wird dann im Abschnitt 5.3 erklärt.

5.1.1 Ablaufsteuerung

Die Ablaufsteuerung wird durch ein Objekt der Klasse *BrowserController* realisiert. Mit der Definition

```
OBJECT main := BrowserController!NEW();
```

wird beim Programmstart automatisch ein solches Objekt erzeugt. Hierdurch wird der Konstruktor des Objekts aufgerufen, welcher dann die Ablaufsteuerung übernimmt. Die Verarbeitung der Programmargumente ist im Abschnitt 3.2 beschrieben. Eine wesentliche Aufgabe der Ablaufsteuerung ist die Validierung der Kommandos. Unbekannte Befehle oder soche mit unzureichenden Argumenten werden mit einer Warnung zurückgewiesen. Wurde ein Dateiname erkannt, so wird die ParserManager-Komponente mit dem Parsing dieser Datei beauftragt. Wurde hingegen ein Befehl akzeptiert, so wird die entsprechende Auswertung mit Hilfe der ParserManager-Komponente ausgeführt und das Ergebnis ausgegeben. Dieser Vorgang wird durch die Abbildung 5.1 schematisch dargestellt.

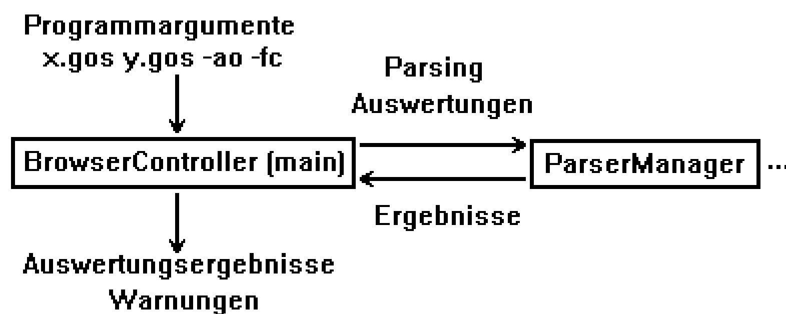


Abbildung 5.1: Schematische Darstellung der Ablaufsteuerung

5.1.2 ParserManager-Komponente

Aufgabe der ParserManager-Komponente ist es, den abstrakten Syntaxbaum zu verwalten. Dieser wird durch das Parsing von Dateien erzeugt bzw. erweitert. Die Durchführung der Auswertungen gehört ebenfalls dazu.

Das Parsing von Dateien wird an die Parser-Komponente deligiert. Der vom Parser erzeugte Teilbaum wird anschließend zum Gesamtbaum hinzugefügt. Die Auswertungen werden vom ParserManager selbst durchgeführt. Hierzu stehen mehrere Methoden zur Verfügung, um die gewünschten Informationen aus dem abstrakten Syntaxbaum zu erhalten. Dieser Vorgang wird von den Abbildungen 5.2 und 5.3 nochmals dargestellt. Für eine detailierte Darstellung sei auf Abschnitt 5.5 verwiesen.

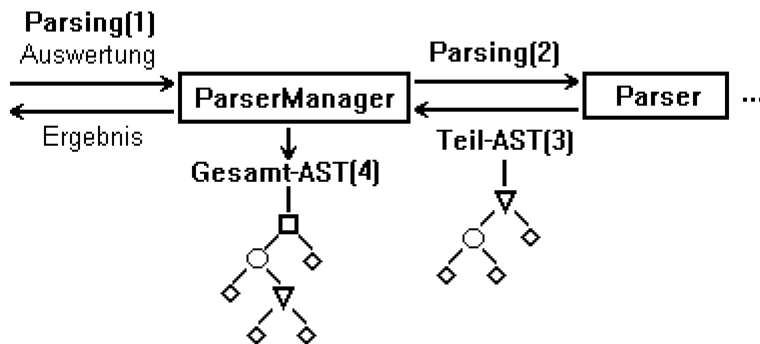


Abbildung 5.2: Der ParserManager erhält den Auftrag, eine Datei zu parsen (1). Dies wird an den Parser weitergeleitet (2). Als Ergebnis wird ein Teilbaum zurückgegeben (3). Dieser wird zum Gesamtbaum hinzugefügt (4).

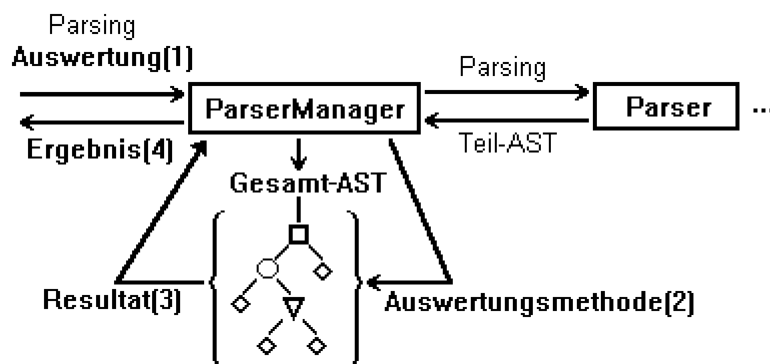


Abbildung 5.3: Der ParserManager erhält den Auftrag, eine Auswertung durchzuführen (1). Hierzu wird eine Auswertungsmethode aufgerufen, die den AST analysiert (2). Die Methode gibt das Resultat der Auswertung zurück (3). Dieses wird dann vom ParserManager als Antwort zurückgesendet (4).

5.1.3 Parser-Komponente

Die Parser-Komponente ist für das Parsing der Dateien zuständig. Das Ergebnis ist ein abstrakter Syntaxbaum, der die Struktur der Datei widerspiegelt. Im Sinne des gesamten Ablaufs handelt es sich hierbei um einen Teilbaum, da auch mehrere Dateien verarbeitet werden können.

Der Parser wird bis auf wenige zusätzliche definierte Hilfsmethoden vollständig generiert. Als Grundlage dient die Definition der GOS-Grammatik. Aus dieser wird dann mit Hilfe des Werkzeuges GBison eine Klasse erzeugt - die Parser-Komponente. Die Produktionen dieser Grammatik enthalten GOS-Anweisungen zum Aufbau des Baumes. Falls eine Produktion erkannt wurde, kommt die hinterlegte Anweisung zur Ausführung. Auf diese Weise wird ein abstrakter Syntaxbaum erzeugt, der die syntaktische Struktur des Dateiinhaltes beschreibt. Für eine exakte Beschreibung der Arbeitsweise dieses Parsers sei auf [4] verwiesen. Eine schematische Darstellung des Parse-Vorgangs zeigt Abbildung 5.4.

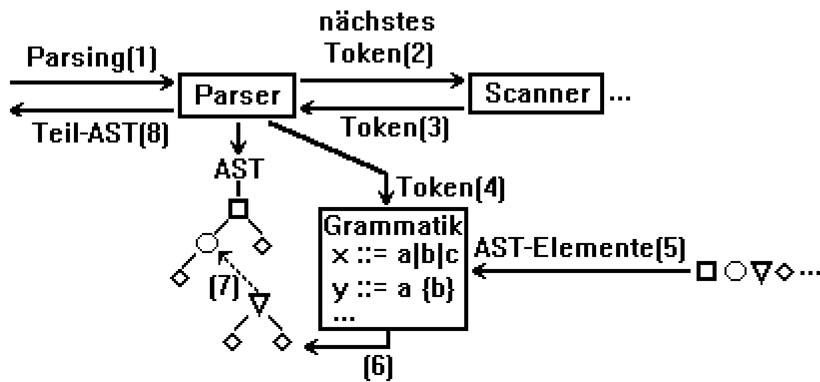


Abbildung 5.4: Nach dem Start des Parse-Vorganges (1) werden Token vom Scanner angefordert (2 und 3), der diese aus der Eingabedatei bildet. Mit Hilfe der Token werden passende Produktionen ausgewählt (4), wobei es auch sein kann, daß Token quasi wieder in den Eingabestrom zurückgegeben werden (Look ahead). Wurde eine Produktion erkannt, werden aus neuen AST-Elementen (5) Teilstrukturen erzeugt (6) und zum abstrakten Syntaxbaum hinzugefügt (7). Nach Abschluß dieses Prozesses wird der erzeugte Teilbaum als Antwort zurückgesendet (8).

5.1.4 Scanner-Komponente

Die Scanner-Komponente ist für die lexikalische Analyser der zu verarbeitenden Dateien zuständig. Anhand vorgegebener Regeln werden aus den Zeichen einer Dateien sogenannte Token gebildet, die dann als Token-Strom von der Parser-Komponente verarbeitet werden.

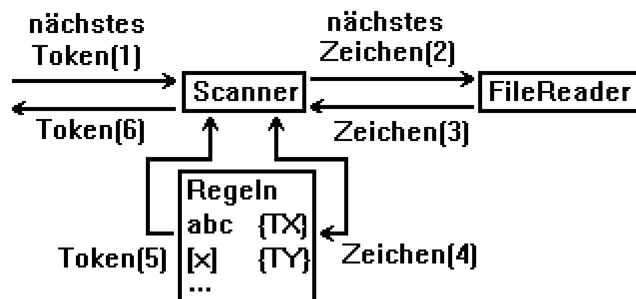


Abbildung 5.5: Ein Token wird vom Parser angefordert (1). Dazu werden Zeichen vom FileReader gelesen (2,3). Mit Hilfe der Zeichen wird eine passende Regel ausgewählt (4), wobei es auch sein kann, daß Zeichen quasi wieder in den Eingabestrom zurückgegeben werden. Wurde eine Regel erkannt, wird ein Token generiert (5) und als Antwort an den Parser zurückgesendet.

Der Scanner wird mit Hilfe des Werkzeuges GLex generiert. Als Grundlage dient die Definition der Regeln zum Erkennen der Token. Die so erzeugte Scanner-Klasse enthält zusätzlich Methoden, die zum Beispiel zur Verarbeitung von Kommentaren oder zur Realisierung der Layout-Rule benötigt werden. Für eine exakte Beschreibung der Arbeitsweise dieses Scanners sei auf [4] verwiesen. Eine schematische Darstellung des Scan-Vorgangs zeigt Abbildung 5.5.

5.1.5 FileReader-Komponente

Die FileReader-Komponente bildet die Schnittstelle zwischen Scanner-Komponente und zu verarbeitender Datei. Der Scanner fordert Zeichen von einem Objekt an, welches von der Klasse *YY_In* oder einer Unterklasse sein muß. Da die Klasse *FileReader* ist eine Unterklasse von *YY_In* ist, kann der Scanner auch vom FileReader-Objekt lesen. Der Scanner kümmert sich also nicht darum, wie die zu verarbeitenden Zeichen bereitgestellt werden, sondern deligiert diese Aufgabe an ein separates Objekt. Die FileReader-Komponente ließt diese Zeichen aus einer Datei. Die Komponente muß auch in der Lage sein, Zeichen vom Scanner wieder entgegen zu nehmen. Hierzu verfügt die FileReader-Komponente über einen Puffer, der zurückgegebene Zeichen zwischenspeichert. Es wird also versucht, ein vom Scanner angefordertes Zeichen zuerst aus dem Puffer zulesen. Der Puffer wird hierbei als Stack betrachtet, d.h. es wird das LIFO-Prinzip (Last in - First out) angewendet. Erst wenn dieser Puffer leer ist, werden weitere Zeichen aus der Datei geladen. Die nachfolgende Abbildung 5.6 stellt diesen Vorgang dar.

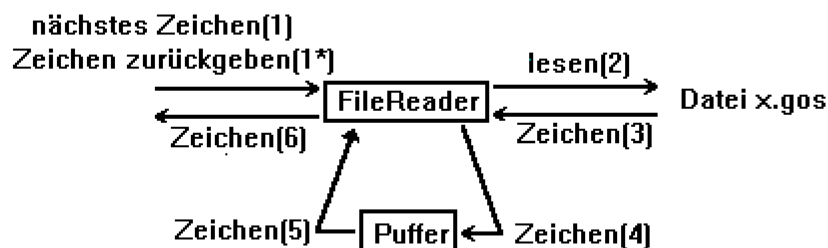


Abbildung 5.6: Wird ein Zeichen vom Scanner angefordert(1), wird zuerst geprüft, ob der Puffer bereits leer ist. Falls ja, werden Zeichen aus der Datei gelesen (2 und 3) und zum Puffer hinzugefügt (4). Anschließend wird ein Zeichen aus dem Puffer entnommen (5) und als Antwort an den Scanner zurückgesendet (6). Soll ein Zeichen wieder zurückgenommen werden (1*), so wird dieses Zeichen wieder zum Puffer hinzugefügt(4). Für den Puffer wird ein Stack verwendet wenn auch die Abbildung die Verwendung einer Warteschlange suggeriert.

5.1.6 Abstrakter Syntaxbaum

Die wichtigste Aufgabe des abstrakten Syntaxbaums ist die Repräsentation der syntaktischen Struktur eines GOS-Programms. Dieser ist aus mehreren Komponenten zusammengesetzt. Dazu gehören sowohl Objekte von GOS-Klassen als auch Strukturen von Gofer-Datentypen. Der abstrakte Syntaxbaum wird durch ein Objekt der Klasse *AbstractSyntaxTree* repräsentiert. Dieser enthält eine Liste von Objekten, die die Top-Level-Definitionen darstellen. Diese setzen sich wiederum aus Objekten und Datenstrukturen zusammen. Diese Grobstruktur ist in Abbildung 5.7 nochmals dargestellt.

Die Elemente des abstrakten Syntaxbaumes enthalten auch Methoden, die für die Implementierung der Auswertungen nützlich sind. Hierzu sei allerdings auf den Abschnitt 5.5 verwiesen.

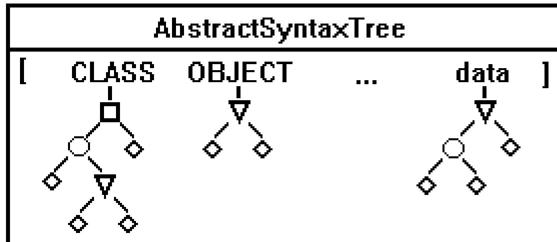


Abbildung 5.7: Das Objekt zur Repräsentation eines abstrakten Syntaxbaumes enthält eine Liste von Top-Level-Definitionen.

5.1.7 Zusammenfassung

Die nachfolgende Abbildung 5.8 zeigt zusammenfassend die Beziehungen zwischen den beschriebenen Komponenten.

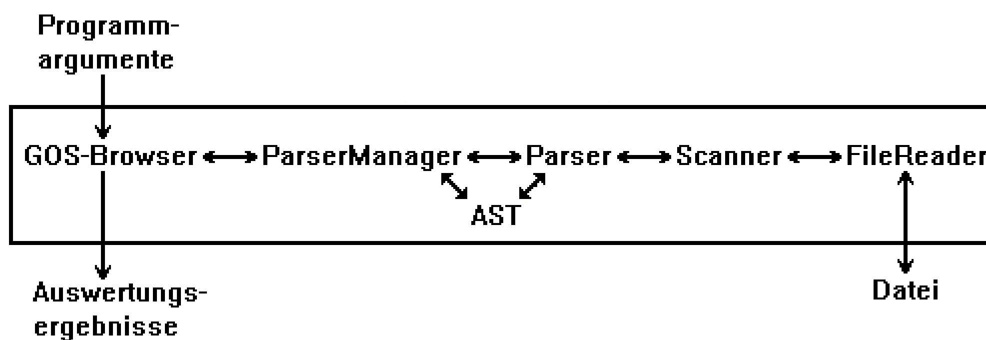


Abbildung 5.8: Diese Abbildung zeigt die bereits beschriebenen Beziehungen zwischen den Komponente, sowie die Schnittstellen des Programms zum Anwender und zu den zu verarbeitenden Dateien.

5.2 Programmstruktur

Dieser Abschnitt beschäftigt sich mit dem Quellcode des GOS-Browsers. Es wird die Grobstruktur, d.h. die Aufteilung und Bedeutung der Quelldateien erklärt. Da im Entwicklungsprozeß auch Code mit speziellen Werkzeugen generiert wird, enthält dieser Abschnitt auch eine Beschreibung des Erstellungsprozesses.

5.2.1 Quelldateien und Programmaufbau

Der Quellcode des GOS-Browsers verteilt sich auf mehrere Dateien. So sind die im vorausgegangenem Abschnitt beschriebenen Komponenten auf mehrere Quelldateien verteilt. Zum Beispiel sind die Ablaufsteuerung und der FileReader in separaten Dateien enthalten. Zusätzlich befinden sich darunter auch Dateien, die als Grundlage zur Generierung von GOS-Code dienen. Folgende Dateien bilden nun den Quellcode des GOS-Browsers:

- controller.gos
- parsermgr.gos
- filereader.gos
- ast.gos
- globals.gos
- parser.y
- scanner.x

Die Datei *controller.gos* enthält die Klassendefinition *BrowserController* für die Komponente der Ablaufsteuerung. Die Definition der ParserManager-Komponente ist in der Datei *parsermgr.gos* zu finden. Die FileReader-Komponente für den Scanner ist in der Datei *filereader.gos* untergebracht. Die gesamten Klassen und Datenstrukturen für den abstrakten Syntaxbaum sind in der Datei *ast.gos* definiert. Die Definitionen global bekannter Objekte wie zum Beispiel *io* oder *sys*, die mehr oder weniger von allen Komponenten benötigt werden, befinden sind in der Datei globals.gos.

Die beiden letzten Dateien spielen eine besondere Rolle. Sie dienen als Grundlage zur Generierung der Dateien *parser.gos* und *scanner.gos*. Die Datei *parser.y* enthält die Definition der GOS-Grammatik und einige Methoden, die zum Aufbau des abstrakten Syntaxbaumes notwendig sind. Die Datei *scanner.x* enthält die Definitionen von Regeln zur lexikalischen Analyse des Eingabestromes und zusätzlich Methoden zum Beispiel zur Realisierung der Layout-Rule. Die nachfolgenden Dateien werden also aus den Dateien *parser.y* und *scanner.x* generiert:

- parser.gos
- scanner.gos

Zusätzlich zu diesen Quelldateien werden zur Generierung die folgende Dateien benötigt, die aber in der Regel nicht verändert werden müssen. Auch für den GOS-Browser wurden diese Dateien nicht modifiziert.

- glex118.skl
- gbison118.simple

Die Beziehungen zwischen diesen Dateien lassen sich durch die folgende Abbildung 5.9 schematisch darstellen. Betrachtet man die Dateien als geschlossene Module, so können die Verbindungen als „benützt“-Relation aufgefasst werden. Die gestrichelten Pfeile verdeutlichen den Generierungsprozeß.

5.2.2 Erstellungsprozeß

Der Erstellungsprozeß gliedert sich in drei Schritte:

1. Generierung der Datei scanner.gos
2. Generierung der Datei parser.gos
3. Zusammenführen der Quelldateien

Für die ersten beiden Schritte werden die Werkzeuge GLex und GBison benötigt. Die Angabe der verwendeten Versionen ist im Anhang A enthalten. Diese beiden Programme entsprechen den Unix-Werkzeugen Lex und Yacc. Der wesentliche Unterschied besteht darin, daß die Programme GLex und GBison anstatt C-Code GOS-Code erzeugen.

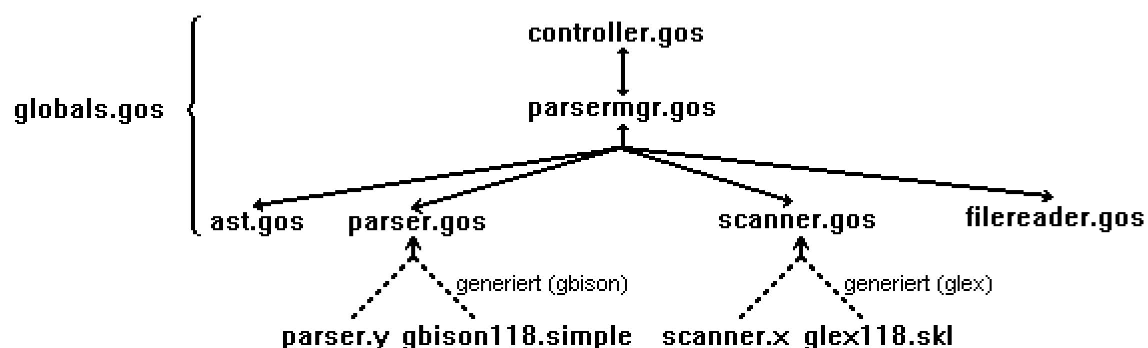


Abbildung 5.9: Die vollen Pfeile stellen eine Art „benützt“-Beziehung zwischen den „Modulen“ dar. Die gestrichelten Pfeile beschreiben die Generierung von Dateien. Die geschweifte Klammer deutet an, daß die Datei *globals.gos* von allen anderen Modulen genutzt wird.

Der letzte Schritt führt alle einzelnen Quelldateien sowie die generierten Dateien zu einer einzigen Datei *gosbrowser.gos* zusammen. Dies wird durch einfaches Hintereinanderkopieren mit dem Unix-Befehl *cat* durchgeführt. Die Reihenfolge der Dateien ist dabei nicht von Bedeutung, da der GOS-Interpreter hierzu keine Einschränkungen macht. Dieser Vorgang wird durch die Abbildung 5.10 dargestellt.

Für die Durchführung des Erstellungsprozesses existiert eine Make-Datei. Die Datei gosbrowser.gos wird also mit dem Unix-Kommando *make* erstellt.

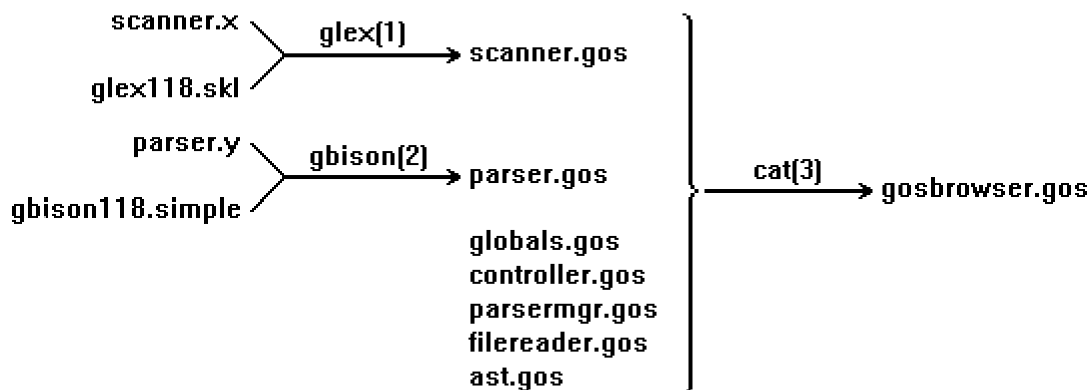


Abbildung 5.10: Der Erstellungsprozeß umfaßt die Generierung der Datei *scanner.gos* (1), die Generierung der Datei *parser.gos* (2) und das Zusammenführen in die Datei *gosbrowser.gos* (3).

5.3 Parse-Framework

Das Parse-Framework besteht aus den bereits beschriebenen Komponenten einschließlich des abstrakten Syntaxbaumes. In diesem Abschnitt werden die Klassen und Datenstrukturen hierzu ausführlich beschrieben.

5.3.1 Allgemeines

Die Scanner- und Parser-Komponenten lesen die mit Hilfe der Kommandos übergebenen Dateien und erzeugen einen abstrakten Syntaxbaum. Die Datenstrukturen dafür werden in der Datei *ast.gos* bereitgestellt. Zur Repräsentation dieses abstrakten Syntaxbaumes kommen sowohl GOS- als auch Gofer-Datenstrukturen zum Einsatz.

Für den OO-Teil der Grammatik werden GOS-Klassen eingesetzt. So gibt es zum Beispiel Klassen für Klassen-, Methoden- und Attribute-Definitionen. Für den Gofer-Teil wird die Top-Level-Ebene, also zum Beispiel Datatype- oder Infix-Definition mit entsprechenden Klassen repräsentiert. Alle anderen Teile der Grammatik wie zum Beispiel Typen oder Ausdrücke werden durch Gofer-Datenstrukturen ausgedrückt.

5.3.2 Gofer-Datenstrukturen für den abstrakten Syntaxbaum

Zur Repräsentation großer Teile der Gofer-Grammatik werden Gofer-Datenstrukturen eingesetzt. Die verwendeten Datentypen sind stark von der vorliegenden Definition der Gofer-Grammatik geprägt. Beispielsweise existiert für viele Nicht-Terminale ein entsprechender Datentyp. Teile von Produktionen können dabei in weitere Datentypen untergliedert sein. Als Beispiel sei das Nicht-Terminal *constrs* erwähnt, welches in die Typdefinitionen *GoConstrs* und *GoConstr* zerlegt wurde. Für einige Produktionen existiert keine eigene Datenstruktur, da die Daten entweder in den Datenstrukturen der umschließenden Produktion enthalten sind oder durch die Tupel- und Listentypen hinreichend dargestellt werden können.

Die Beschreibung der Datentypen wird in logische Gruppen aufgeteilt. Diese orientiert sich an zusammengehörige Produktionen und der Aufteilung der Gofer-Grammatik. Zudem existieren für den OO-Teil der GOS-Grammatik einige hilfreiche Definitionen. Die sich daraus ergebenden Schwerpunkte sind die folgenden:

- Variablen, Operatoren und Literale
- Ausdrücke, Atome, Listen
- Patterns
- Wertdeklarationen
- Class- und -Instance-Definitionen
- Typen
- Top-Level
- Definitionen für OO-Teil

Diese werden nun im einzelnen beschrieben. Zum besseren Verständnis werden die Produktionen der Gofer-Grammatik in den Abschnitten über Ausdrücke und Typen vollständig aufgeführt. Somit wird auch das Verhältnis zwischen der Grammatik und der Wahl der Datenstrukturen aufgedeckt.

♦ Variablen, Operatoren und Literale

Die Grammatik stützt sich bei Variablen und Operatoren auf die Terminale ***varid***, ***conid***, ***varop*** und ***conop***. Die ersten beiden Terminale stellen Bezeicher mit kleinem bzw. großem Anfangsbuchstaben dar. Die letzen beiden bezeichnen Operatoren, die aus einer eingeschränkten Menge von speziellen Zeichen (**:**, **!**, **#**, **$** usw.) aufgebaut werden. Operatoren, die mit einem Doppelpunkt beginnen, werden als ***conop*** bezeichnet, die anderen als ***conid***. Alle vier Terminale werden als Zeichenketten (**String**) gespeichert. Deshalb werden die Typen **Tvarid**, **Tconid**, **Tvarop** und **Tconop** als **String** definiert.

In einem Gofer-Programm besteht an manchen Stellen die Möglichkeit, mittels geeigneter Notation sowohl ***varid*** anstelle von ***varop*** also auch ***conid*** anstelle von ***conop*** zu verwenden. Die Umkehrung gilt natürlich auch. Dies realisieren die Nicht-Terminale **varid**, **conid**, **varop** und **conop**. Hierfür werden die Typen **Varid**, **Conid**, **Varop** und **Conop** als **String** definiert. Und schließlich gibt es die Nicht-Terminale **var** und **op**. Diese werden durch die Typen **Var** und **Op** repräsentiert, welche ebenfalls als **String** definiert werden.

Literale werden vom Scanner erkannt und als Terminale vom Parser verarbeitet. Die entsprechenden Typen **IntegerLiteral**, **FloatLiteral**, **CharLiteral** und **StringLiteral** werden analog zu **Varid**, **Conid** usw. als **String** definiert.

Insgesamt werden also die folgenden Typen eingeführt:

```
Tvarid      Tvarop      Varid       Varop       Var
Tconid      Tconop      Conid       Conop       Op
IntLit      FloatLit    CharLit     StringLit
```

♦ Ausdrücke, Atome und Listen

Die Darstellung von Ausdrücken läßt sich in drei Teile gleidern. Der erste Teil definiert umfassendere Strukturen wie zum Beispiel if-Ausdrücke und Funktions-Applikationen. Der zweite Teil beschäftigt sich mit den Atomen, den strukturell einfachen Elementen von Ausdrücken. Und der dritte Teil widmet sich einer besonderen Art von Atomen, den Listen.

Ausdrücke

Die folgenden Produktionen der Gofer-Grammatik definieren die Struktur von Ausdrücken. Die erste davon ist die Produktion **exp**:

```
exp                 ::=  "\" apat {apat} "->" exp           Lambda-Ausdruck
                       |  "let" "{" decls "}" "in" exp      lokale Definitionen
                       |  "if" exp "then" exp "else" exp    bedingter Ausdruck
                       |  "case" exp of "{" alts "}"        Case-Ausdruck
                       |  opExp "::" sigType                getypter Ausdruck
                       |  opExp
```

Das Nicht-Terminal **exp** wird durch den Datentyp **GoferExpression** repräsentiert. Die Alternativen der Produktion bilden die Konstruktoren dieses Datentyps. Die Nicht-Terminale der Alternativen finden sich als Parameter der jeweiligen Konstruktoren wieder.

```
data GoferExpression =
          GoExpressionLambda [GoAppPat] GoferExpression
        | GoExpressionLet [GoDefinition] GoferExpression
        | GoExpressionIf GoferExpression GoferExpression GoferExpression
        | GoExpressionCase GoferExpression [GoAlt]
        | GoExpressionTyped GoOpExp (Opt GoSigType)
```

Der Datentyp **Opt** ermöglich die optionale Angabe von Werten und ist folgendermaßen gegeben:

```
data Opt a = Absent | Present a
```

Der Teil **apat {apat}** der Lambda-Alternative wird beispielsweise durch den Datentyp **[GoAppPat]** repräsentiert. Wie weiter unten noch gezeigt wird, steht der Typ **GoAppPat** für das Nicht-Terminal **apat**. Für die letzten beiden Produktionen besteht die folgende Alternative der Datentypdefinition

```
data GoferExpression =
        ...
        | GoExpressionOpExpTyped GoOpExp GoSigType
        | GoExpressionOpExp GoOpExp
```

Diese Variante würde der existierenden Definition eher entsprechen. Jedoch berücksichtigt sie nicht, daß die letzten beiden Alternativen der Produktion im Grunde die folgende Alternative beschreiben:

```
exp               ::=  "\" apat {apat} "->" exp          Lambda-Ausdruck
                       ...
                     | opExp [ "::" sigType ]            evtl. getypter Ausdruck
```

Daher wurde die vorliegende Definition (mit **`Opt`**) gewählt, da sie die Absicht der Alternativen besser wiedergibt. Die fehlenden Produktionen zu diesem Teil lauten wie folgt:

```
opExp             ::=  opExp op opExp                    operator application
                     | pfxExp

pfxExp            ::=  "-" appExp                        negation
                     | appExp

appExp            ::=  appExp atomic                     function application
                     | atomic
```

Die hierzu vewendeten Datentypen sind **`GoOpExp`**, **`GoPfxExp`** und **`GoAtomic`**. Die Definition von **`GoAtomic`** wird im nächsten Teil gegeben. Die der ersten beiden lauten wie folgt.

```
data GoOpExp =
         GoOpExpOp GoOpExp Op GoOpExp
       | GoOpExpPfx GoPfxExp

data GoPfxExp =
         GoNegatedAppExp [GoAtomic]
       | GoAppExp [GoAtomic]
```

Auch diese Produktionen zeigen deutlich die Abbildung von Alternativen auf Konstruktoren. Die Namensgebung folgt dabei einer einfachen Regel. Der Datentyp hat in der Regel das Präfix „Go“ (für Gofer) und wird zumeist wie das Nicht-Terminal genannt. Die Alternativen werden aus dem Namen des Datentyps gebildet und mit einer abkürzenden Beschreibung der jeweiligen Alternative versehen. Gelegentliche kommen natürlich Ausnahmen vor, wenn beispielsweise eine andere Bezeichnung besser ist. Für das Nicht-Terminal **`appExp`** wurde kein eigener Datentyp eingeführt, da die Produktion keine Alternativen besitzt. Stattdessen wird der Typ **`[GoAtomic]`** verwendet.

Atome

Dieser Teil beschreibt das in Ausdrücken vorkommende Nicht-Terminal **`atomic`**. Die Produktion lautet wie folgt:

```
atomic            ::=  var                               Variable
                     | conid                             Konstruktor
                     | integer                           Integer-Literal
                     | float                             Floating-Point-Literal
                     | char                              Character-Literal
                     | string                            String-Literal
                     | "()"                              Unit Element
                     | "(" exp ")"                       geklammerter Ausdruck
                     | "(" exp op ")"                    Sections
                     | "(" op exp ")"
                     | "[" list "]"                      Listenausdruck
```

```
                    |  "(" exp "," exp {"," exp} ")"        Tupel
```

Der zugehörige Datentype **GoAtomic** ist wie folgt definiert:

```
data GoAtomic =
        GoAtomicVar Var
      | GoAtomicConid Conid
      | GoAtomicInteger IntLit
      | GoAtomicFloat FloatLit
      | GoAtomicChar CharLit
      | GoAtomicString StringLit
      | GoAtomicUnit
      | GoAtomicExpression GoferExpression
      | GoAtomicSectionEO GoferExpression Op
      | GoAtomicSectionOE Op GoferExpression
      | GoAtomicList GoList
      | GoAtomicTuple [GoferExpression]
```

Der Datentyp GoList repräsentiert Listenausdrücke. Dieser wird im folgenden und zugleich letzten Teil beschrieben.

Listen

Listenausdrücke werden durch die folgenden Produktionen beschrieben. Die eckigen Klammern, die Listen umschließen, sind bereits in der Produktion des Nicht-Terminals **atomic** enthalten.

```
list            ::=  [ exp {"," exp} ]                  Aufzählung, evtl. leer
                  |  exp | quals                        beschriebene Liste
                  |  exp ".."                           arithmetische Sequenz
                  |  exp "," exp ".."
                  |  exp ".." exp
                  |  exp "," exp ".." exp

quals           ::=  quals "," quals                    mehrfache Angaben
                  |  pat "<-" exp                       Generator
                  |  pat "=" exp                        lokale Definition
                  |  exp                                boolescher Wächter
```

Die Repräsentation von Listen wird durch die folgenden Datentypen vorgenommen. Der Datentyp **GoPattern** wird im Abschnitt über Patterns definiert.

```
data GoList =
        GoListEnum [GoferExpression]
      | GoListCompreh GoferExpression [GoQual]
      | GoListArithEPP GoferExpression
      | GoListArithEEPP GoferExpression GoferExpression
      | GoListArithEPPE GoferExpression GoferExpression
      | GoListArithEEPPE GoferExpression GoferExpression GoferExpression

data GoQual =
        GoQualGenerator GoPattern GoferExpression
      | GoQualLocalDef GoPattern GoferExpression
      | GoQualBooleanGuard GoferExpression
```

Zu bemerken ist die Tatsache, daß die Produktion **quals** nicht direkt mit dem Datentyp **GoQual** korrespondiert. Die Möglichkeit, mehrere durch Komma getrennte Angaben machen zu können, wird durch den Type **[GoQual]** im Datentyp **GoList** ausgedrückt, wohingegen der Datentyp **GoQual** nur die „nicht-listenbildenden" Alternativen repräsentiert. Diese Art der Listenbildung wird in der Gofer-Grammatik häufig verwendet. Die hier gewählte Lösung wird im folgenden auch für alle anderen Fälle dieser Art angewendet werden.

Alternativen (case)

Der *case*-Ausdruck enthält eine Liste von Alternativen. Diese werden durch die folgenden Produktionen beschrieben. Das Nicht-Terminal **where** für lokale Definitionen sowie das Nicht-Terminal **pat** für Patterns wird weiter unten beschrieben.

```
alts           ::=  alts "," alts                   mehrfache Alternativen
                 |  pat altRhs [where]              Alternative

altRhs         ::=  "->" exp                        einzelne Alternative
                 |  gdAlt {gdAlt}                   bewachte Alternativen

gdAlt          ::=  "|" exp "->" exp                bewachte Alternative
```

Auch hier kommt für das Nicht-Terminal **alts** die Listenbildung vor. Wie bereits weiter oben gezeigt, wird dies dadurch gelöst, daß für die nicht-listenbildenden Alternativen ein Datentyp geschaffen wird. In diesem Fall bleibt nur eine Alternative übrig, so daß eine einfache Typdefinition für **GoAlt** ausreicht. Die Liste wird mit Hilfe des Typs **[GoAlt]** ausgedrückt. Im Konstruktor **GoExpressionCase** des oben definierten Datentyps **GoferExpression** wird dieser Listentyp zur Repräsentation des Nicht-Terminals **alts** verwendet, das in der Produktion von **exp** vorkommt. Wie weiter unten noch gezeigt wird, werden für Patterns und lokale Definitionen die Datentypen **GoPattern** und **GoLocalDefinitions** verwendet. Die Definitionen lauten also wie folgt:

```
type GoAlt = (GoPattern,GoAltRhs,GoLocalDefinitions)

data GoAltRhs =
        GoAltRhsSingle GoferExpression
      | GoAltRhsGuarded [GoGuardedAlternative]

type GoGuardedAlternative = (GoferExpression,GoferExpression)
```

♦ Typen

Für Typen gibt es die Möglichkeit, diese mit oder ohne Kontextbedingungen anzugeben. Ein solcher mit Kontext wird als qualifizierter Typ bezeichnet, einer ohne schlicht als Typ. Die Definition des ersteren baut auf die des letzten auf.

Typ

Die Produktionen für einen Typen gliedern sich in drei Teile. Diese Teile ermöglichen die Angabe von Funktionstypen, Datentypekonstruktoren, Typsynonymen und grundlegenden Typen.

```
type            ::=  ctype ["->" type]                        Funktionstyp

ctype           ::=  conid {atype}                            Datentyp und Typsynonym
                  |  atype

atype           ::=  varid                                    Typvariable
                  |  "()"                                     Unit-Typ
                  |  "(" type ")"                             geklammerter Typ
                  |  "(" type "," type {"," type} ")"         Tupeltyp
                  |  "[" type "]"                             Listentyp
```

Dier hierfür verwendeten Datentyp- und Typendefinitionen lauten wie folgt:

```
type GoferType = [GoCType]

data GoCType =
         GoCTypeConid Tconid [GoAType]
       | GoCTypeAType GoAType

data GoAType =
         GoATypeVar Tvarid
       | GoATypeUnit
       | GoATypeType GoferType
       | GoATypeTuple [GoferType]
       | GoATypeList GoferType
```

Qualifizierter Typ
Für qualifizierte Typen ist die Angabe einer Kontextbedingung möglich. Dies wird durch das Nicht-Terminal **sigType** ausgedrückt.

```
sigType         ::=  [context "=>"] type                      [qualifizierter] Typ

context         ::=  "(" [pred {"," pred} ] ")"               allgemeine Form
                  |  pred                                     einzelner Kontext

pred            ::=  conid type {type}                        Prädikat
```

Die hierfür verwendeten Definitionen sind die folgenden.

```
type GoSigType = (Opt GoContext, GoferType)

data GoContext =
         GoContextGeneral [GTPred]
       | GoContextSingleton GTPred

type GoPred = (Tconid, [GoferType])
```

Die Diskussion der nachfolgenden Schwerpunkte ist im Gegensatz zu den vorausgegangenen Abschnitten kurz gefaßt. Für die Abbdilung der Produktionen auf die Datentypen wurde die gleiche Vorgehensweise angewendet.

♦ Patterns

Zur Definition von Pattern existieren die Nicht-Terminale **pat**, **appPat** und **apat**. Das Nicht-Terminal **apat** wird durch den Datentyp **GoAPat** repräsentiert, wobei das Nicht-Terminal **appPat** auf den Typen **[GoAPat]** abgebildet wurde. Dem Nicht-Terminal **pat** entspricht der Datentyp **GoPattern**.

```
data GoPattern =
         GoPatternOpAppl GoPattern Conop GoPattern
       | GoPatternNPlusK Var IntLiteral
       | GoPatternAppl [GoAPat]

data GoAPat =
         GoAPatVar Var
       | GoAPatAlias Var GoPattern
       | GoAPatIrrefutable GoPattern
       | GoAPatWildcard
       ...
       | GoAPatList [GoPattern]
       | GoAPatTuple [GoPattern]
```

♦ **Wertdeklarationen**

Wertdeklarationen sind in der Top-Level-Ebene enthalten und werden für lokale Definitionen verwendet. Sie beinhalten Type-Definitionen, Funktionsdefinitionen und Pattern-Bindungen. Das Nicht-Terminal hierfür lautet **decls**. Für diese Definitionen wurde der Datentyp **GoDefinition** eingeführt, wobei die Liste dem Typ **[GoDefinition]** entspricht. Lokale Definitionen werden durch das Nicht-Terminal **where** dargestellt. Hierfür wurde der Typ **GoWhere** eingeführt. Optionale lokale Definitionen werden allerdings nicht mit **Opt GoWhere** gebildet, sondern ebenfalls mit **GoWhere**, wobei fehlende Definitionen durch die leere Liste dargestellt werden. Für die Funktionsdefinition wurde ein eigener Type eingeführt (**GoFunDefn**), da er bei Class- und Instance-Definitionen ebenfalls benötigt wird.

```
data GoDefinition =
         GoDefinitionVar [Var] GoSigType
       | GoDefinitionFun GoFunDefn
       | GoDefinitionPat GoPattern GoRhs GoWhere

type GoFunDefn = (GoFun, GoRhs, GoWhere)

type GoWhere = [GoDefinition]
```

Das Nicht-Terminal **fun** wird auf den Datentyp **GoFun** abgebildet.

```
data GoFun =
         GoFunVar Var
       | GoFunInfix GoPattern Varop GoPattern
       | GoFunSectionPV GoPattern Varop
       | GoFunSectionVP Varop GoPattern
       | GoFunArgument GoFunction GoAPat
       | GoFunFunction GoFunction
```

Die rechte Seite von Funktionsdefinitionen und Pattern-Bindungen (Nicht-Terminale **rhs** und **gdRhs**) wird durch den Datentyp **GoRhs** und den Typ **GoGuardedExpression** abgebildet.

```
data GoRhs =
         GoRhsSimple GoferExpression
       | GoRhsGuarded [GoGuardedExpression]

type GoGuardedExpression = (GoferExpression, GoferExpression)
```

♦ Class- und Instance-Definitionen

Class- und Instance-Definitionen werden durch die Datentypen **`GoClassDefinition`** und **`GoInstanceDefinition`** vertreten. Der Kopf beider Definitionen (das sind die Nicht-Terminale **`class`** und **`inst`**) enthält einen optionalen Kontext und ein Prädikat. Der optionale Rumpf einer Instance-Definition besteht aus einer Liste von Funktionsdefinitionen (Nicht-Terminal **`idecls`**), wohingegen der optionale Rumpf einer Class-Definition zusätzlich auch Signaturen enthalten kann (Nicht-Terminal **`cdecls`**). Für die Darstellung der leeren Rümpfe (das sind die Nicht-Terminale **`cbody`** und **`ibody`**) werden keine Typsynonyme oder Datentypen eingeführt, sondern leere Listen verwendet.

```
type GoClassDefinition = (Opt GoContext, GoPred, [GoClassBodyDefinition])

type GoInstanceDefinition = (Opt GoContext, GoPred, [GoFunDefn])

data GoClassBodyDefinition =
         GoClassMemberFunctions ([Var], GoferType)
       | GoClassDefaultBinding GoFunDefn
```

♦ Top-Level

Da die Definitionen in der Top-Level-Ebene durch Klassen repräsentiert werden, bleiben nur noch Nicht-Terminale übrig, die als Teile dieser Produktionen auftreten.

```
type GoTypeLhs = (Tconid, [Tvarid])

type GoConstrs = [GoConstr]

data GoConstr =
         GoConstrConop GoferType Tconop GoferType
       | GoConstrConid Tconid [GoferType]

type prims = [(Var, StringLit)]
```

♦ Definitionen für OO-Teil

Dieser Abschnitt beschäftigt sich im Gegensatz zu den vorausgegangenen Abschnitten nicht mit der Gofer-Grammatik. Die hier vorgestellten Definitionen werden für die Repräsentation der Konstrukte des OO-Teils benötigt.

Den Nicht-Terminalen **`Ident`**, **`ClassIdent`**, **`GOSType`** und **`GOSExp`** entsprechen die Typen **`Identifier`**, **`ClassName`**, **`GOSType`** und **`GOSExp`**:

```
type ClassName  = Tconid

type Identifier = Tvarid
```

```
type GOSType = GoferType

type GOSExp  = GoferExpression
```

Der Typ **`TypedIdent`** wird für lokale Variablen, Attribute und Parameter verwendet, da diese immer getypt sind.

```
type TypedIdent = (Identifier,GOSType)
```

Der Typ **`CondBlock`** wird zur Vereinfachung definiert und wird für bedingte Blöcke verwendet. Diese treten beispielsweise bei IF-Anweisungen auf.

```
type CondBlock  = (Expression,CommandList)
```

Der Datentyp **`AccessType`** wird für Attribute und Methoden benötigt, um diese als *public*, *protected* oder *private* deklarieren zu können. Die Definition gleicht einem Aufzählungstypen:

```
data AccessType = PublicAccess | PrivateAccess | ProtectedAccess
```

5.3.3 Klassenhierarchie zum abstrakten Syntaxbaum

Wohingegen weite Teile der Gofer-Grammatik mit Gofer-Datenstrukturen selbst repräsentiert werden, kommen für den OO-Teil und die Top-Level-Ebene Klassen zum Einsatz. Die hieraus entstandene Klassenhierarchie wird in Abbildung 5.11 dargestellt. Die Definitionen sind in der Datei *ast.gos* enthalten. Wichtig ist die Tatsache, das die Wahl der Konstrukturen stark von der vorliegenden Grammatik abhängt. Da dort der abstrakte Syntaxbaum aufgebaut wird, sind die Parameter den Produktionen angepaßt.

- **AbstractSyntaxTreeElement**

Die Klasse **`AbstractSyntaxTreeElement`** ist die gemeinsame Oberklasse aller Elemente des abstrakten Syntaxbaumes. Die Unterklassen stellen im Grunde die Repräsentation der wichtigsten Nicht-Terminale der Grammatik dar. In der hier beschriebenen Version verfügt die Oberklasse über keine wesentliche Methoden, die implementierten Auswertungen keine allgemeinen Eigenschaften vorraussetzen. Es wäre allerdings denkbar, in der gemeinsamen Oberklassen die Position im Quellcode abzulegen. Die Klasse ist abstrakt, da von ihr keine Objekte erzeugt werden.

- **TopLevelDefinition**

Die Klasse **`TopLevelDefinition`** ist gemeinsame Oberklasse aller Top-Level-Definitionen. Wie auch die Klasse **`AbstractSyntaxTreeElement`** definiert diese Klasse keine weiteren Methoden. Sie dient daher im Grund nur als Definition eines Klassentyps. Die Klasse ist ebenfalls abstrakt, da von ihr keine Objekte erzeugt werden.

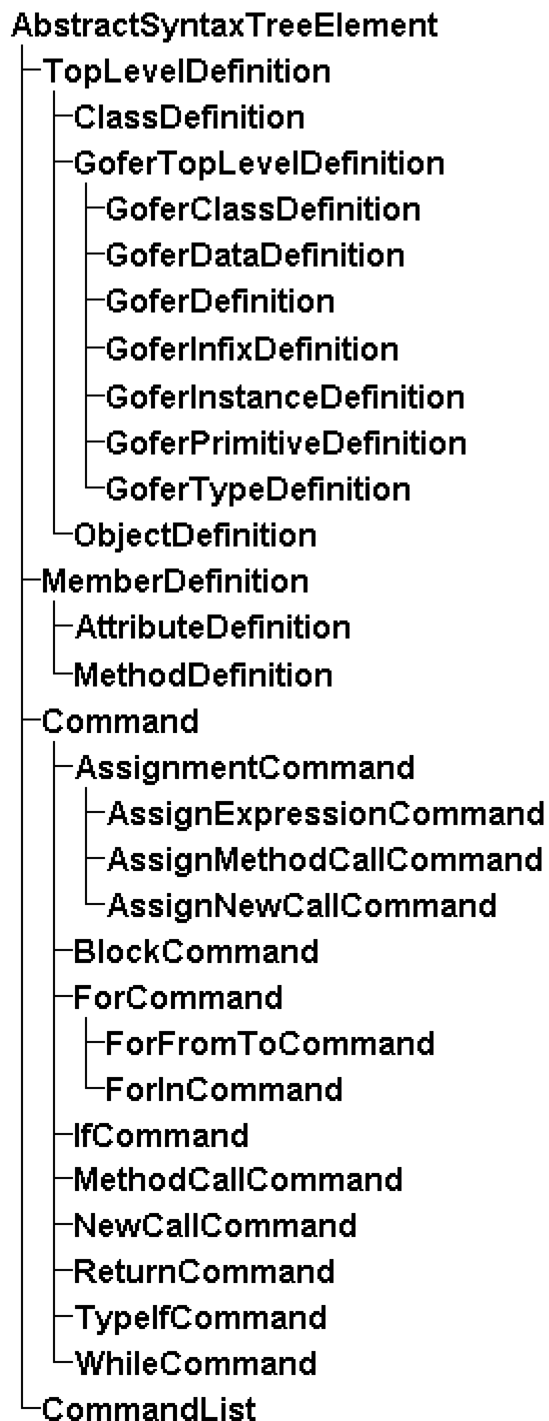


Abbildung 5.11: Klassenhierarchie für die Elemente des abstrakten Syntaxbaumes.

♦ **ClassDefinition**
Jede Klassendefinition wird durch ein Objekt der Klasse **`ClassDefinition`** repräsentiert. Die Definition entspricht dem Nicht-Terminal **`GOSClassDefn`** aus der GOS-Grammatik in Kapitel 4. Das Objekt verwaltet die Liste der Oberklassen, die Attribute- und Methodendefinitionen. Für die letzten beiden werden wiederum Objekte verwendet. Die zugehörigen Klassen werden weiter unten beschrieben. Die Klasse enthält außer den üblichen Zugriffs- und Bearbeitungsmethoden keine speziellen Methoden.

Die Klasse definiert folgende Attribute:

```
name     :: ClassName;
inherits :: [ClassName];
members  :: [MemberDefinition];
```

Das Attribute **`name`** bezeichnet die Klassennamen, das Attribute **`inherits`** enthält eine Liste der Oberklassen und das Attribute **`members`** ist eine Liste aller Attribute- und Methodendefinitionen dieser Klasse.

Der Konstruktor erlaubt die Erzeugung eines Objekte einschließlich der Angabe des Klassennamens, der Oberklassen und den Elementdefinitionen.

```
NEW(n :: ClassName, inheritedClasses :: [ClassName], memberDefns :: [MemberDefinition]
```

Die Grammtatik enthält für die Klassendefinition die folgende Produktion für das Nicht-Terminal **`GOSClassDefn`**:

```
GOSClassDefn  ::=  "CLASS" ClassIdent [ SubclassOf ] ClassMemberDefns "ENDCLASS" ...
```

Dies legt die oben gewählte Variante des Konstruktors nahe. Damit kann auf die Definition von Methoden verzichtet werden, die Werte von Attributen ändern. Entsprechendes gilt für die Konstruktoren der ürigen Klassen.

Drei Zugriffsmethoden erlauben das Auslesen der Attribute. Zudem gibt es zwei zusätzliche Methoden, um die Attibute- und Methodendefinitionen separat auslesen zu können.

```
className() ClassName

superclasses() [ClassName]

classMembers() [MemberDefinition]

attributeMembers() [AttributeDefinition]

methodMembers() [MethodDefinition]
```

Die Klasse definiert auch zwei weitere Methoden, um neue Elemente zu den Listen für Oberklassen und Elementdefinitionen aufnehmen zu können.

```
addSuperclass(n :: ClassName)

addMember(m :: MemberDefinition)
```

Exemplarisch für alle weiteren Klassen woll an dieser Stelle der Quellcode der Klasse **ClassDefinition** gezeigt werden.

```
CLASS ClassDefinition
   SUBCLASSOF TopLevelDefinition;

PROTECTED
   name     :: ClassName;                 -- Klassenname
   inherits :: [ClassName];               -- Liste der Oberklassen
   members  :: [MemberDefinition];        -- Liste der Elementdefinitionen

PUBLIC
   --
   -- Konstruktor - Initialisierung des Objekts
   --
   NEW(n                :: ClassName,
       inheritedClasses :: [ClassName],
       memberDefns      :: [MemberDefinition]) =
   BEGIN
      name := n;                          -- Die Attribute werden
      inherits := inheritedClasses;       -- mit den Argumenten
      members := memberDefns;             -- initialisiert.
   END;

   --
   -- Die Methode addSuperclass() fuegt eine Oberklasse hinzu
   --
   addSuperclass(n :: ClassName) =
   BEGIN
      inherits := n:inherits;
   END;

   --
   -- Die Methode addMember() fuegt eine Elementdefinition hinzu
   --
   addMember(m :: MemberDefinition) =
   BEGIN
      members := m:members;
   END;

   --
   -- Die Zugriffsmethode className() gibt den Klassennamen zurueck
   --
   className() ClassName =
   BEGIN
      RETURN name;
   END;

   --
   -- Die Zugriffsmethode superclasses() gibt die Liste der
   -- Oberklassen zurueck
   --
   superclasses() [ClassName] =
   BEGIN
      RETURN inherits;
   END;

   --
   -- Die Zugriffsmethode classMembers() gibt die Liste der
   -- Elementdefinitionen zurueck
```

```
   --
   classMembers() [MemberDefinition] =
   BEGIN
      RETURN members;
   END;

   --
   -- Die Zugriffsmethode attributeMembers() gibt die Liste der
   -- Attributedefinitionen zurueck
   --
   attributeMembers() [AttributeDefinition] =
   VAR
      member :: MemberDefinition;
      list :: [AttributeDefinition];
   BEGIN
      list := []::[AttributeDefinition];
      FOR member IN members DO
         TYPEIF member IS AttributeDefinition THEN
            list := (coerceAttributeDefinition member):list;
         ENDTYPEIF;
      ENDDO;

      RETURN list;
   END;

   --
   -- Die Zugriffsmethode methodMembers() gibt die Liste der
   -- Methodendefinitionen zurueck
   --
   methodMembers() [MethodDefinition] =
   VAR
      member :: MemberDefinition;
      list :: [MethodDefinition];
   BEGIN
      list := []::[MethodDefinition];
      FOR member IN members DO
         TYPEIF member IS MethodDefinition THEN
            list := (coerceMethodDefinition member):list;
         ENDTYPEIF;
      ENDDO;

      RETURN list;
   END;

ENDCLASS ClassDefinition;
```

⬩ ObjectDefinition

Jede Definition eines global bekannten Objekts (Nicht-Terminal **`GOSObjectDefn`** der GOS-Grammatik) wird durch ein Objekt der Klasse **`ObjectDefinition`** dargestellt. Es enthält den Namen des zu definierenden globalen Objekts und den Konstruktor-Aufruf. Hierfür werden die folgenden Attribute verwendet.

```
name  :: Identifier;
value :: NewCallCommand;
```

Die Klasse definiert einen Konstruktor

```
NEW(v :: NewCallCommand)
```

und zwei Zugriffsmethoden

```
objectName() Identifier

assignedValue() NewCallCommand
```

♦ **GoferTopLevelDefinition**

Die Klasse **`GoferTopLevelDefinition`** ist die Oberklasse aller Top-Level-Definitionen der Gofer-Grammatik, wobei für jede eine entsprechende Unterklasse existiert. Die Klasse selbst ist abstrakt. Wie auch die Klasse **`TopLevelDefinition`** definiert diese Klasse keine weiteren Methoden. Sie dient daher im Grund nur als Definition eines Klassentyps.

♦ **Unterklassen von GoferTopLevelDefinition**

Jede Unterklasse von **`GoferTopLevelDefinition`** entspricht einer Top-Level-Definition von Gofer. Daher gibt es folgende Unterklassen:

- **`GoferClassDefinition`**
- **`GoferDataDefinition`**
- **`GoferDefinition`**
- **`GoferInfixDefinition`**
- **`GoferInstanceDefinition`**
- **`GoferPrimitiveDefinition`**
- **`GoferTypeDefinition`**

Jede dieser Unterklassen enthält Attribute zur Repräsentation der entsprechenden Definition. Hierfür werden ausschließlich Gofer-Datenstrukturen verwendet. Als Beispiel soll die Klasse **`GoferDataDefinition`** vorgestellt werden. Die folgenden Attribute gehen aus der Produktion zu den Nicht-Terminalen **`GoferTopDefn`** und **`typeLhs`** hervor:

```
name      :: Tconid;
variables :: [Tvarid];
constrs   :: [GoConstr];
```

Der Konstruktor lautet dementsprechend

```
NEW(n :: Tconid, vars :: [Tvarid], c :: [GoConstr]) =
```

Zudem existieren die folgenden Zugriffsmethoden:

```
datatypeName() Tconid

typeVariables() [Tvarid]

constructors() [GoConstr]
```

♦ MemberDefinition

Die Klasse **`MemberDefinition`** ist die abstrakte Oberklasse der Attribut- und Methoden-Definitionen. Sie definiert das Attribute **`name`** für einen Attribut- bzw. Methodennamen und das Attribute **`access`** für die Zugriffsregelung.

```
name   :: Identifier;
access :: AccessType;
```

Zudem sind geeignete Zugriffsmethoden vorhanden.

```
getAccess() AccessType

setAccess(newAccess :: AccessType)

getName() Identifier

setName(newAccess :: Identifier)
```

♦ AttributeDfinition

Die Klasse AttributeDefinition wird für alle Attribut-Definitionen verwendet. Zusätzlich wird in dieser Klasse das Attribut **`attributeType`** definiert, um den Typ eines Attributs aufnehmen zu können.

```
attributeType :: GOSType;
```

Zusätzlich sind Zugriffsmethoden implementiert.

```
getType() GOSType

setType(newType :: GOSType)
```

♦ MethodDfinition

Jede Methoden-Definition wird duch ein Objekt der Klasse **`MethodDefinition`** repräsentiert. Es enthält zusätzlich Attribute für die Methodenparameter, den Ergebnistyp, die lokalen Variablen und den Methodenrumpf:

```
params  :: [TypedIdent];
mthType :: Opt GOSType;
locals  :: [TypedIdent];
block   :: BlockCommand;
```

Zusätzlich sind Zugriffsmethoden implementiert.

```
getParams() [TypedIdent]

setParams (newParams :: [TypedIdent])

addParameter(param :: TypeIdent)

getType() Opt GOSType

setType (newType :: Opt GOSType)

getLocals() [TypedIdent]

setLocals (newLocals :: [TypedIdent])

addLocal(var :: TypeIdent)

getBlock() BlockCommand

setBlock (newBlock :: BlockCommand)
```

♦ **Command**

Die Klasse **`Command`** ist die abstrakte Oberklasse für GOS-Anweisungen. Es existieren folgende *direkte* Unterklassen.

- **`AssignmentCommand`**
- **`BlockCommand`**
- **`ForCommand`**
- **`IfCommand`**
- **`MethodCallCommand`**
- **`NewCallCommand`**
- **`ReturnCommand`**
- **`TypeIfCommand`**
- **`WhileCommand`**

Die Klassen **`AssignmentCommand`** und **`ForCommand`** sind selbst wiederum abstrakt. Für alle drei Möglichkeiten, einen Wert an eine Variable zuzuweisen (GOS-Ausdruck, Methodenaufruf und Instanziierung), gibt es die entsprechende Unterklassen **`AssignExpressionCommand`**, **`AssignMethodCallCommand`** und **`AssignNewCallCommand`**. Die Unterklassen **`ForFromToCommand`** und **`ForInCommand`** der Klasse **`ForCommand`** entsprechen den zwei Varianten einer FOR-Schleife.

♦ **CommandList**
Objeke der Klasse **`CommandList`** werden für solche Stellen der Grammatik verwendet, an denen durch Semikola getrennte Anweisungen auftreten können. Dies sind beispielsweise Rümpfe von FOR- und WHILE-Schleifen oder THEN- und ELSE-Zweige von IF-Anweisungen.

5.3.4 Die Klasse FileReader

Die Klasse **`FileReader`** ist eine Unterklasse von **`YY_In`**, welche in dem von GLex generierten Code definiert ist. Aufgabe dieser Klasse ist es, den Scanner mit einem Eingabestrom zu versorgen, wobei nun die Unterklasse **`FileReader`** den Scanner mit dem Inhalt einer Datei versorgt. Die Definition ist in der Datei *filereader.gos* enthalten. Zu den wichtigsten Methoden dieser Klasse zählen **`getchar()`**, **`ungetchar()`**, **`ungetstring()`** und **`eof()`** und natürlich der Konstruktor

```
NEW(fileName :: String)
```

zum Erzeugen eines Objekts mit Zugriff auf die Datei **`fileName`**.

Die Klasse definiert eine weitere Methode, die vom Scanner benutzt wird um Kommentare zu verarbeiten. Ein Aufruf der Methode **`ignoreGoferComments`** liest solange aus der Datei, bis das Ende eines eventuell geschachtelten Kommentars erreicht ist.

5.3.5 Die Scanner-Klasse

Der Code der Scanner-Komponente wird von dem Programmierwerkzeug GLex generiert. Die zugehörige Klasse heißt **`YY_GLexLexer`**. Zusätzlich hierzu werden Attribute und Methoden definiert, die zur Realisierung der Layout-Rule notwendig sind. Dieser Punkt wird allerdings in einem eigenen Abschnitt weiter unten behandelt. Zudem enthält der Scanner die Verarbeitung von Kommentaren. Diese können allerdings geschachtelt sein, so daß eine gesonderte Behandlung erforderlich ist. Die Definitionen der Regeln zur lexikalischen Analyse und der zusätzlichen Attribute und Methoden sind in der Datei *scanner.x* enthalten. Die Anwendung des Scanners wird im Abschnitt über die ParserManager-Komponente vorgeführt.

<u>*Terminale*</u>
Der Scanner erkennt Terminale sowohl aus der GOS- als auch aus der Gofer-Grammatik. Diese werden jeweils durch einen geeigneten Konstruktor identifiziert, welche mit dem Präfix **Tres** beginnen.

♦ **Reservierte Wörter für Gofer**
Folgende Liste zeigt die reservierten Wörter der Gofer-Grammatik.

```
case        class       data        else
if          in          infix       infixl
infixr      instance    let         of
primitive   then        type        where
```

♦ **Reservierte Wörter für GOS**
Folgende Liste zeigt die reservierten Wörter der GOS-Grammatik.

```
BEGIN        CLASS        DO           ELIF
ELSE         END          ENDCLASS     ENDDO
ENDIF        ENDTYPEIF    FOR          IF
IN           IS           NEW          PRIVATE
PROTECTED    PUBLIC       SUBCLASSOF   TO
THEN         TYPEIF       VAR          WHILE
```

♦ **Bezeichner**
Bezeichner werden von der GOS- und der Gofer-Grammatik verwendet. Dabei wird zwischen Bezeichner mit großen und kleinen Anfangsbuchstaben unterschieden. Die Bezeichner mit großen Anfangsbuchstaben werden als **conid** bezeichnet und bilden das Pendant zu ***conid*** der Gofer-Grammatik. Zudem werden die **conid**-Terminale als Klassennamen verwendet. Die Bezeichner mit kleinen Anfangsbuchstaben werden als **varid** bezeichnet und bilden das Pendant zu ***varid*** der Gofer-Grammatik. Zudem werden die **varid**-Terminale als Variablen- und Attributenamen verwendet.

♦ **Spezielle Zeichenkombinationen für Gofer**
:: verwendet in getypten Ausdrücken
= zum Beispiel verwendet für Typ und Data-Definitionen
.. verwendet in Listenausdrücken
@ Aliase in Pattern
\ Lambda-Ausdruck
| zum Beispiel verwendet für Alternativen und Data-Constructors
<- Generator in Listenausdrücken
-> zum Beispiel verwendet für Funktionstypen und Lambda-Expressions
~ Irrefutable Patterns
=> verwendet für Contexte
() Unit-Elemente
(-) geklammertes Minus

♦ **Spezielle Zeichenkombinationen für GOS**
:= Zuweisungsoperator
! Methodenaufruf

♦ **Operatoren**
Operatoren werden von der Gofer-Grammatik verwendet. Dabei wird zwischen **varop**- und **conop**-Operatoren unterschieden. Operatoren können aus einer festgelegten Menge von Zeichen kombiniert werden. **Conop**-Operatoren müssen dabei mit einem Doppelpunkt beginnen, wohingeben alle anderen gebildeten Operatoren als **varop**-Operatoren gelten.

♦ **Literale**
Literale werden durch die Gofer-Grammatik vorgegeben. Es existieren Literale für Ganzzahlen (Int), Gleitpunktzahlen (Float), Zeichen (Char) und Zeichenketten (String).

Folgende Escape-Codes für Zeichenliterale werden erkannt:

\a 7	\b 8	\f 12	\n 10
\r 13	\t 09	\v 11	
\\ \	\" "	\' '	
\NUL 0	\SOH 1	\STX 2	\ETX 3
\EOT 4	\ENQ 5	\ACK 6	\BEL 7
\BS 8	\HT 9	\LF 10	\VT 11
\FF 12	\CR 13	\SO 14	\SI 15
\DLE 16	\DC1 17	\DC2 18	\DC3 19
\DC4 20	\NAK 21	\SYN 22	\ETB 23
\CAN 24	\EM 25	\SUB 26	\ESC 27
\FS 28	\GS 29	\RS 30	\US 31
\SP 32	\DEL 127		

5.3.6 Die Parser-Klasse

Der Code der Parser-Klasse wird von dem Programmierwerkzeug GBison generiert. Die zugehörige Klasse heißt **`YY_GBisonParser`**. Zusätzlich werden Hilfsmethoden und -Attribute definiert, die zum Aufbau des abstrakten Syntaxbaumes nützlich sind. Das wichtigste davon ist das Attribut **`parseTree`**. Es enthält den aufzubauenden abstrakten Syntaxbaum. Die Anwendung des Parsers wird im Abschnitt über die ParserManager-Komponente vorgeführt.

Zur Realisierung der Layout-Rule wurden in der Grammatik und im Code einiger Produktionen spezielle Maßnahmen ergriffen. Dies wird in einem eigenen Abschnitt weiter unten erläutert.

5.3.7 Die Klasse ParserManager

Die Klasse ParserManager entkoppelt die Hauptprogrammlogik von den eigentlichen Arbeitsprozessen und verbirgt die Implementierungsdetails des Parsens und den Auswertungen. Um nun eine Datei parsen zu können, müssen zwei Schritte unternommen werden. Zuerst wird die gewünschte Datei mit der Methode **`setFileName()`** vorbereitet und dann anschließend mit **`parse()`** vearbeitet. Dieser Vorgang kann beliebig oft durchgeführt werden, wobei dann die Teilbäume zu einem einzigen abstrakten Syntaxbaum verschmolzen werden. Für die Auswertungen stehen zahlreiche Methoden zur Verfügung. Auf diese wird im Abschnitt über die Auswertungen näher eingegangen werden.

Um die Anwendung der Scanner- und Parser-Komponenten zu demonstrieren, ist im folgenden eine gekürzte Version der Klasse **`ParserManager`** dargestellt.

```
CLASS ParserManager

PROTECTED
   fileName :: String;
   scanner  :: YY_GLexLexer;
   parser   :: YY_GBisonParser;
   yyin     :: FileReader;


PUBLIC
   --
   -- Konstruktor - Initialisierung des Objekts mit
   -- neuen Scanner- und Parser-Objekten und
   -- leerem abstrakten Syntaxbaum.
   --
   NEW() =
   BEGIN
      scanner := YY_GLexLexer!NEW();
      parser := YY_GBisonParser!NEW();
      parser!resetParseTree();
      fileName := "";
   END;

   --
   -- Zu verarbeitende Datei vorbereiten.
   --
   setFileName(fName :: String) =
   BEGIN
      fileName := fName;                        -- Dateiname merken
      yyin := FileReader!NEW(fName);            -- FileReader-Objekt erzeugen
      scanner!set_yyin(coerceYY_In yyin);       -- und dem Scanner mitteilen
   END;

   --
   -- vorbereitete Datei parsen und
   -- Fehlercode zurueckgeben
   --
   parse() Int =
   VAR
      result :: Int;
      ready :: Bool;
   BEGIN
      IF fileName == "" THEN         -- wurde setFileName() aufgerufen?
         result := -1;
      ELSE
         ready := yyin!isReady();    -- Ist FileReader bereit zum Lesen?
         IF ready THEN
            result := parser!yyparse(scanner);    -- Parser arbeitet auf Scanner
         ELSE
            result := -2;
         ENDIF;
      ENDIF;

      RETURN result;
   END;

...
Auswertungsmethoden
...

ENDCLASS ParserManager;
```

5.4 Layout

Das Layout von GOS-Programmen betrifft die Verwendung von Kommentaren und die Anwendung der Layout-Rule. Die implementativen Aspekte hierzu werden in den beiden folgenden Abschnitten erläutert.

5.4.1 Kommentare

GOS verwendet die selben Kommentare wir Gofer. Zum einen gibt es durch `--` eingeleitete Zeilenkommentare und zum anderen geschachtelte Kommentare, die von den Zeichen `{-` und `-}` eingeschlossen werden. Die Kommentare wurden im Abschnitt 4.1.1 bereits dargestellt.
Die Verarbeitung der Kommentare geschieht vollständig im Scanner. Zur Erkennung von Zeilenkommentare existiert eine Regel, die dafür sorgt, daß der Kommentar ignoriert wird. Zur Verarbeitung von geschachtelten Kommentaren ist eine Regel zur Erkennung der Zeichenfolge `{-` vorhanden. Um allerdings einen solchen Kommentar im allgemeinen ignorieren zu können, muß eine Methode verwendet werden. Die Mächtigkeit der Scanner-Regeln reicht hierfür nicht aus. Wurde also ein Kommentaranfang erkannt, wird die Methode **`ignoreGoferComments()`** des Scanners aufgerufen. Diese ruft wiederum die Methode **`ignoreGoferComments()`** der FileReader-Komponente auf. Anschließend ist das nächste Zeichen im Eingabestrom das Zeichen direkt nach dem Kommentarende. Wenn also die Regel verarbeitet wurde, ist somit auch der geschachtelte Kommentar ignoriert worden.

5.4.2 Layout-Rule

Die Realisierung der Layout-Rule betrifft sowohl den Scanner als auch den Parser. Die meiste Arbeit wird allerdings vom Scanner erledigt.

Zur Implementierung der Layout-Rule wird die Gofer-Grammatik in einer modifizierten Form eingesetzt. Dies betrifft die Teile der Gofer-Grammatik für lokale Definitionen (**where**), Let-Ausdrücken und Of-Klauseln des Case-Ausdrucks, welche geschachtelte Definitionsblöcke enthalten. Diese werden in geschweiften Klammern eingeschlossen. Da die Layout-Rule das Weglassen dieser Klammern ermöglicht, wurde zur Vereinfachung der Implementierung die Grammatik von den öffnenden Klammern befreit. Die Terminale **where**, **let** und **of** enthalten sozusagen implizit eine öffnende geschweifte Klammer. Dies wird dadurch erreicht, daß nach dem Erkennen eines der Schlüsselwörter **where**, **let** oder **of** die nachfolgenden Zeichen untersucht werden. Folgt diesen Symbolen ein '{' Token, so wird dieses einfach verschluckt und ein geschachtelter Definitionblock eröffnet, in dem die Layout-Rule allerdings nicht aktiviert wird. Fehlt das '{' Token, so wird ein geschachtelter Definitionsblock mit aktiver Layout-Rule eröffnet. In jedem Fall enthalten die Symbole **where**, **let** und **of** implizit die öffnende geschweifte Klammer, auch wenn diese im ersten Fall auch tatsächlich im Eingabestrom auftaucht. Dies bedeutet, daß nach diesen Schlüsselwörtern die öffnende geschweifte Klammer in der vorliegenden Grammtik nicht mehr vorkommt.

Um die Layout-Rule anwenden zu können, muß für jede Zeile die Einrückung bekannt sein. Für Einrückungen größer 0 kann hierfür eine Scanner-Regel angegeben werden. Die Erkennung der Einrückung 0 dagegen bereitet Probleme. Es kann nämlich keine Regel angegeben werden, die quasi eine leere Zeichenkette am Anfang einer Zeile erkennt.

Um dieses Problem zu umgehen, wird am Ende jeder Zeile die Einrückung der nächsten relevanten Zeile sozusagen vorrausberechnet. Dies geschieht innerhalb einer Regel, die das Zeilenende-Zeichen ' \n' erkennt. Ergibt sich ein Wert größer als 0, so wird nichts unternommen, da in diesem Fall die oben genannte Regel greift. Ergibt sich allerdings eine Einrückung von 0, so wird die Layout-Rule innerhalt der '\n'-Regel für die Einrückung 0 angewendet.

Da die Definitionsblöcke geschachtelt sein können, wird zur Verwaltung ein Stack eingesetzt. Jedes Element des Stacks stellt somit einen offenen geschachtelten Definitionsblock dar, wobei jedes Element ein Tupel mit folgenden Komponenten ist:

- Es enthält die Einrückung der Zeile, in welcher der Definitionsblock eröffnet wurde.
- Es enthält ein Flag, welches angibt, ob die Layout-Rule vor dem Öffnen des Definitionsblockes aktiv war.
- Zudem wird zum Definitionsblock ein Flag abgelegt, welches angibt, ob das gesamte Konstrukt durch das Schließen des Blockes beendet wird. Dies ist für **where** und **of** der Fall. Für **let** gilt dies nicht, da nach dem Definitionsblock das let-Konstrukt nicht beendet ist. Diese Information wird benötigt um entscheidenen zu können, ob nach dem Definitionsblock eventuell noch ein ' ;' Token eingefügt werden muß.

Zur Verarbeitung stehen beim Scanner im wesentlichen drei Methoden zur Verfügung. Die Methode **`countIndentation()`** berechnet zu einem gegebenen String die Einrückungszahl gemäß der Definition der Layout-Rule. Grob gesprochen zählt hierbei jedes Zeichen außer dem Tab-Zeichen einfach, und das Tab-Zeichen rückt die Position bis zum nächsten Tabstop vor, der nach jeweils 8 Zeichen plaziert wird. Die Methode **`checkNewIndentation()`** überprüft anhand des derzeitigen Zustandes der Layout-Rule, welche Maßnahmen eingeleitet werden müssen. Beispielsweise könnte sich ergeben, das ein Semikolon eingefügt werden muß oder ein Block mit lokalen Definitionen zu Ende geht. In diesem Fall muß eine schließende geschweifte Klammer eingefügt und der Stack entsprechend abgebaut werden. Die Methode **`checkCascadedBlock()`** wird nach dem Erkennen eines der Schlüsselwörter **where**, **let** und **of** aufgerufen. Wie oben bereits beschrieben wird damit nach einer folgenden öffnenden geschweiften Klammer Ausschau gehalten und entsprechend die Layout-Rule aktiviert oder deaktiviert. In jedem Falls wird ein Element auf den Stack gelegt, da es sich hierbei um einen geschachtelten Block handelt.

Die realisierte Layout-Rule unterscheidet sich von der ursprünglichen Version. Dies betrifft den Punkt 4 (vgl. Abschnitt 4.4.2 bzw. Anhang D) der Definition. In der Gofer-Version sorgt dieser Punkte dafür, daß eine schließende geschweifte Klammer eingefügt wird und die Layout-Rule endet, falls vor lokalen Definitionen (nach **where** und **let**) oder vor Alternativen (nach **of**) automatisch eine öffnende geschweifte Klammer eingefügt wurde und die schließende geschweifte Klammer bei genau diesen lokalen Definitionen oder Alternaiven fehlt. Somit ist es möglich, den Ausdruck

```
let a = fact 12 in a+a
```

ohne expliziter Angabe von geschweiften Klammern zu verwenden. Dies entspricht dann dem Ausdruck

```
let {a = fact 12} in a+a
```

Ohne diese Regelung wird vor dem Schlüßelwort **in** keine geschweifte schließende Klammer eingefügt, da diese von den übrigen Regeln nicht abgedeckt wird.

5.5 Auswertungen

Die ParserManager-Komponente übernimmt die Kontrolle über alle Auswertungen. Für jeden Auswertungsbefehl steht eine entsprechende Methode zur Verfügung. Beispielsweise existiert für jede Auswertung der Gruppe A jeweils eine Methode:

```
-ac       getAllClassNames() [String]
-ao       getAllObjectNames() [String]
-agc      getAllTypeClassNames() [String]
-agd      getAllDatatypeNames() [String]
...
```

Zur Durchführung der Auswertungen stehen mehrere Hilfsmethoden zur Verfügung. Zum Beispiel wird die Methode

```
findClass(className :: String) Opt ClassDeclaration
```

dazu verwendet, eine Klassendefinition im abstrakten Syntaxbaum anhand des Klassennames zu finden. Zudem verfügen die Klassen für die GOS-Definitionen über Methoden, die für die Realisierung der Auswertungen hilfreich oder sogar notwendig sind.

5.5.1 Arbeitsweise

Alle Auswertungen werden von der Ablaufsteuerung an die ParserManager-Komponente deligiert. Sobald ein Befehl erkannt wurde, wird die entsprechende Methode des ParserManagers aufgerufen und das Ergebnis formattiert ausgegeben. Den Methoden werden gegebenenfalls Bezeichner und Filter als weitere Argumente übergeben.

Für Befehle der Gruppe A werden lediglich die Top-Level-Definitionen der gewünschten Art ausgefiltert und deren Namen in einer Liste gesammelt. Diese werden dann als Ergebnis zurückgegeben.

Für einen Befehl der Gruppe S wird die Top-Level-Definition der gewünschten Art gesucht, die dem übergebenen Namen entspricht. Beispielsweise wird zur Verarbeitung des Befehls **-so myObject** die Methode **showObjectDefinition()** der ParserManager-Komponente aufgerufen. Als Argument wird der Name des Objektes übergeben, dessen Definition ausgegeben werden soll. Diese Methode durchsucht die Liste aller Top-Level-Definitionen nach Objekten der Klasse **ObjectDefinition** und vergleicht den Namen jeder gefundenen Definition mit dem übergebenen Objektnamen. Falls die richtige Definition gefunden wurde, wird die Methode **show()** an das Objekt gesendet, um das Ergebnis zu erhalten. Die Methode **show()** wird von jeder Klasse implementiert, die eine Top-Level-Definition repräsentiert. Dies sind alle Unterklassen der Klasse **TopLevelDefinition**.

Die Befehle der Gruppe D sind für die Analyse von Klassendefinitionen zuständig. Als erstes wird anhand des übergegenen Klassennamens das zugehörige Objekt der Klasse **ClassDefinition** mit Hilfe der Methode **findClass()** gesucht. Anhand der Klassendefinition lassen sich dann die gewünschten Informationen ermitteln. Der Befehl **-dm** geht einen Schritt weiter. Je nach Angabe der

Filter werden Attribut- und Methodendefinitionen der Klasse und gegebenenfalls auch der Oberklassen ermittelt. Als Ergebnis werden die Namen der Attribute und Methoden zurückgegeben.

5.5.2 Beispielauswertung

Exemplarisch soll in diesem Abschnitt der Arbeitsablauf einer konkreten Auswertung nachvollzogen werden. Die Beispielauswertung

```
example1.gos -scmpr Square
```

soll alle öffentlichen Methoden der Klasse Square ausgeben.

1. Die Ablaufsteuerung erkennt den Befehl und ruft Methode **`showClassDefinition()`** der ParserManager-Komponente auf. Als Argumente werden der Klassenname und die Filtereinstellungen übergeben.
2. In der Auswertungsmethode wird die Methode **`findClass()`** aufgerufen, um die Klassendefinition mit dem Namen **`Square`** zu suchen. Falls die Klasse nicht gefunden wurde, wird eine Warnung als Auswertungsergebnis zurückgegeben.
3. An die Definition, ein Objekt der Klasse **`ClassDefinition`**, wird die Methode **`show()`** gesendet und das Resultat als Auswertungsergebnis zurückgegeben. Als Argumente werden die Filtereinstellungen (**`m`** für Methoden und **`pr`** für *protected*) übergeben.
4. Die Methode show() ermittelt anhand der Filtereinstellungen die gewünschten Elemente. Es können sowohl grundsätzlich Attribute und Methoden, als auch Elemente einer bestimmten Zugriffsart ausgefiltert werden. In diesem Fall werden alle Methoden mit Zugriffsrecht *protected* oder *public* ermittelt.
5. An jede Elementdefinition, in diesem Fall alles Objekte der Klasse **`MethodDefinition`**, wird die Methode **`signature()`** gesendet, um die Signatur jedes Elements zu erhalten. Die Liste der Signaturen wird als Auswertungsergebnis zurückgegeben.
6. Die Auswertung ergibt folgende Ausgabe:

```
Show Public Methods of Class Square
   PUB NEW(iOrigin :: Point, iLength :: Float)
   PUB height() Float
   PUB isSquare() Bool
   PUB width() Float
```

Kapitel 6

Zusammenfassung und Ausblick

Im Rahmen dieser praktischen Semesterarbeit wurde ein GOS-Browser unter Benutzung der Werkzeuge GLex und GBison implementiert. Mit Hilfe des GOS-Browsers ist es nun möglich, GOS-Programme aus verschiedenen Abstraktionsebenen betrachten zu können. Insbesondere kann die Vererbungsstruktur von Klassenhierarchien aus unterschiedlichen Blickwinkeln beleutet werden. Die wichtigsten Schwerpunkte der Realisierung waren

- die Definition der Regeln für die lexikalische Analyse,
- die Definition der GOS- und somit auch der Gofer-Grammatik für die Parser-Komponente,
- die Entwicklung und Implementierung geeigneter Datenstrukturen in GOS bzw. Gofer, um den resultierenden abstrakten Syntaxbaum abbilden zu können,
- die Implementierung von Auswertungen, um Informationen aus dem verarbeiteten Quellcode extrahieren zu können und
- die Realisierung der sogenannten Layout-Rule von Gofer, welche die Implementierung des Scanners und die abgebildete Grammatik für den Parser entscheidend beeinflußt hat.

Ein wichtiges und nicht vollständig gelöstes Problem tratt während den Arbeiten zur Grammatik zu tage. Dieses Problem betrifft die sogenannten Shift/Reduce und Reduce/Reduce-Konflikte, die bei der Umsetzung der vorgelegten GOS-Grammatik in den entsprechenden GOS-Code auftreten. Es ist zwar noch möglich, einige dieser Konflikte zu beseitigen. Jedoch ist die Beseitigung aller Konflikte unmöglich, da sie inhärent mit der vorliegenden Gofer-Grammatik verknüpft sind. Eine genaue Analyse wurde im Rahmen dieser Arbeit allerdings nicht durchgeführt, so daß die Grammatik in dieser Hinsicht sicher noch verbessert werden kann.

Abgesehen von diesem Problem läßt sich dieser Browser noch um einiges erweitern. Zum Beispiel sind Weiterentwicklungen und Erweiterungen zu folgenden Themen naheliegend:

- Fehlerbehandlung beim Parsen von fehlerhaftem Code. Hierzu muß die Definition der Grammatik modifiziert werden, so daß aussagekräftige Fehlermeldungen ausgegeben werden. Es wäre auch denkbar, durch geeignete Methoden auftretende Fehler zu analysieren und die Kompilation gegebenenfalls fortsetzen zu können.

- Die Möglichkeiten des Browsers könnten beispielsweise durch Anreicherung des abstrakten Syntaxbaumes um semantische Informationen, den Einsatz von Symboltabellen oder der Berücksichtigung lokaler Sichtbarkeit erweitert werden.
- Der Browser kann und soll um weitere nützliche Auswertungen ergänzt werden. Zum Beispiel ist die Erweiterung um eine Vergleichs-Funktion denkbar. Diese würde es ermöglichen, verschiedene Versionen eines Quellcodes zu vergleichen und somit die Unterschiede auszugeben.

Alles in allem wurde mit diesem neuen Werkzeug eine solide Basis für weiterführende Anwendungen geschaffen. Auf dessen Grundlage können bessere Auswertungen und Analysen aufgesetzt werden. Aber auch für eine zukünftige Kompiler-Komponente kann diese Arbeit als Ausgangspunkt dienen.

Anhang A

Technische Daten und Bedienhinweise

Versionen, Lokationen und Dokumentation
Im folgenden werden die verwendeten Versionen, Lokationen und Dokumentationen der eingesetzten Programme und Komponenten angegeben.

- **GOS-Browser**

Dateien: gosbrowser, gosbrowser.gos
Version: 1.0 vom 08.01.1997
Pfad: hpbroy13, lesny/fopra/gosbrowser/

- **GOS-Interpreter**

Datei: gos
Version: 1.22 vom 27.08.1996
Pfad: hpbroy13, /usr/proj/gos/gos122/bin/
Dokumentation: Siehe [7], [10].

- **GLex**

Dateien: glex, glex118.skl
Version: vom 25.11.1995
Pfad: hpbroy13, /usr/proj/gos/glex/glex
Dokumentation: Siehe [4].

- **GBison**

Dateien: gbison, gbison118.simple
Version: vom 04.12.1995
Pfad: hpbroy13, /usr/proj/gos/gbison2/gbison
Dokumentation: Siehe [4].

- **Gofer-Grammtik**

Version: 2.20 vom 20.10.1995
Dokumentation: Siehe [9].

♦ **GOS-Grammtik**
Dokumentation: Siehe [7], [10], [12].

♦ **Beispielprogramme**
Dateien: example1.gos, example2.gos
Version: 1.0 vom 08.01.1997
Pfad: hpbroy13, lesny/fopra/gosbrowser/
Dokumentation: Siehe Anhang E sowie Abschnitt 3.4

Installation
Um den GOS-Browser verwenden zu können, muß der Pfad zum GOS-Interpreter angegeben sein. Falls Dieser noch nicht vorhanden ist, muß dieser in die entsprechende Set-Path-Anweisung aufgenommen werden, beispielsweise:

```
path = ...;/usr/proj/gos/gos122/bin;...lesny/fopra/gosbrowser
```

Kompilation des GOS-Browsers
Zur Kompilation des GOS-Browsers ist ein Makefile im oben beschriebenen Pfad vorhanden. Der Aufruf

```
make all
```

wird das Programm *gosbrowser.gos* neu generieren.

Anhang B

GOS-Browser Referenz

Dieser Anhang enthält eine Kurzreferenz der Kommandos des GOS-Browser. Für eine detailierte Darstellung sei auf den Abschnitt 3.3.2 verwiesen.

Programmaufruf
Die allgemeine Form des GOS-Browser-Aufrufs:

```
gosbrowser <arg1> [ <arg2> [ <arg3> ... ] ]
```

Die Argumente arg1, arg2, arg3, ... stellen entweder Dateinamen oder Befehle dar. Letztere werden mit einem vorangestellten Bindestrich eingeleitet, also zum Beispiel **-ao** oder **-db**. Eine Kurzreferenz der Befehle ist im folgenden angegeben.

Abkürzungen

Identifier Bezeichner

Filter Filter **[m|a][pr|pu][d|i]**

Filter' Filter **[m|a][pr|pu]**

Bedeutung der optionalen Filter:

m|a
- ∅ Sowohl Attribute als auch Methoden werden angezeigt
- **m** Es werden nur Methoden angezeigt
- **a** Es werden nur Attribute angezeigt

`pr|pu`

∅	Es werden bzgl. public, protected und private keine Einschränkungen gemacht
`pr`	Es werden nur public und protected Deklarationen angezeigt
`pu`	Es werden nur public Deklarationen angezeigt

`d|i`

∅	Es werden alle, auch die geerbten Deklarationen der angegebenen Klasse betrachtet
`d`	Es werden nur die Deklarationen der angegebenen Klasse betrachtet
`i`	Es werden nur die Deklarationen der Oberklassen der angegebenen Klasse betrachtet

Befehlsgruppe A

`-ac`	**list all classes**
`-ao`	**list all objects**
`-agd`	**list all gofer datatypes**
`-agt`	**list all gofer types**
`-agc`	**list all gofer classes**
`-agi`	**list all gofer instances**
`-agp`	**list all gofer primitives**
`-agf`	**list all gofer infixes**

Befehlsgruppe D

`-db` *`Identifier`*	**list all base classes (superclasses)**
`-ds` *`Identifier`*	**list all subclasses**
`-dm` *`Filter Identifier`*	**list members**

Befehlsgruppe S

`-sc` *`Filter' Identifier`*	**show class definition**
`-so` *`Identifier`*	**show object definition**
`-sgd` *`Identifier`*	**show gofer datatype definition**
`-sgt` *`Identifier`*	**show gofer type definition**
`-sgc` *`Identifier`*	**show gofer class definition**
`-sgi` *`Identifier`*	**show gofer instance definition**
`-sgp` *`Identifier`*	**show gofer primitive definition**
`-sgf` *`Identifier`*	**show gofer infix definition**

Anhang C

Gofer-Grammatik

Dieser Anhang enthält die ursprüngliche Fassung der Gofer-Grammatik. Sie ist dem Anhang A von [9] entnommen. Für eine Beschreibung der Modifikationen sei auf den Abschnitt 4.3 verwiesen.

Anhang D

Beschreibung Gofer-Layout

Dieser Anhang enthält die ursprüngliche Definition des Gofer-Layouts. Sie ist dem Kapitel 13 von [9] entnommen. Für eine Beschreibung des verwendeten Layouts sei auf den Abschnitt 4.4 verwiesen.

Anhang E

Anwendungsbeispiel

Dieser Anhang enthält zwei vollständige GOS-Programme. Es dient als Anwendungsbeispiel für den Abschnitt 3.4, in welchem Ergebnisse von einigen Auswertungen über diese Programme dargestellt werden. Die Programme sind in den Dateien example1.gos und example2.gos enthalten. Siehe hierzu Anhang A.

Beispiel-Quellcode "example1.gos"

```
type Point = (Float,Float)

instance Num Float => Num Point where
   (a1,b1) + (a2,b2)= (a1+a2,b1+b2)
   (a1,b1) - (a2,b2)= (a1-a2,b1-b2)
   negate (a,b)          = (-a,-b)


CLASS GraphicObject -- abstract
PUBLIC
   -- abstract constructor
   NEW() =
      BEGIN
      END;

   -- abstract
   area() Float =
      BEGIN
      END;

   -- abstract
   move(delta :: Point) =
      BEGIN
      END;
ENDCLASS GraphicObject;


CLASS RectangleObject SUBCLASSOF GraphicObject; -- abstract
PROTECTED
   origin :: Point;

PUBLIC
   -- abstract constructor
   NEW() =
      BEGIN
      END;

   -- abstract
   width() Float =
      BEGIN
      END;

   -- abstract
   height() Float =
      BEGIN
      END;

   area() Float =
      VAR
         w, h :: Float;
      BEGIN
         w := self!width();
         h := self!height();
         RETURN w*h;
      END;
```

```
   move(delta :: Point) =
      BEGIN
         origin := origin + delta;
      END;

   isSquare() Bool =
      VAR
         w, h :: Float;
      BEGIN
         w := self!width();
         h := self!height();
         RETURN w == h;
      END;
ENDCLASS RectangleObject;


CLASS Rectangle SUBCLASSOF RectangleObject;
PROTECTED
   w, h :: Float;

PUBLIC
   -- constructor
   NEW(iOrigin :: Point, iWidth :: Float, iHeight :: Float) =
      BEGIN
         origin := iOrigin;
         w := iWidth;
         h := iHeight;
      END;

   -- redefine
   width() Float =
      BEGIN
         RETURN w;
      END;

   -- redefine
   height() Float =
      BEGIN
         RETURN h;
      END;
ENDCLASS Rectangle;


CLASS Square SUBCLASSOF RectangleObject;
PROTECTED
   length :: Float;

PUBLIC
   -- constructor
   NEW(iOrigin :: Point, iLength :: Float) =
      BEGIN
         origin := iOrigin;
         length := iLength;
      END;

   -- redefine
```

```
   width() Float =
      BEGIN
         RETURN length;
      END;

   -- redefine
   height() Float =
      BEGIN
         RETURN length;
      END;

   -- redefine
   isSquare() Bool =
      BEGIN
         RETURN True;
      END;
ENDCLASS Square;


CLASS Triangle SUBCLASSOF GraphicObject;
PROTECTED
   base :: Point;
   p, q, h :: Float;

PUBLIC
   NEW(iBase :: Point, ip :: Float, iq :: Float, ih :: Float) =
      BEGIN
         base := iBase;
         p := ip;
         q := iq;
         h := ih;
      END;

   move(delta :: Point) =
      BEGIN
         base := base + delta;
      END;

   area() Float =
   BEGIN
      RETURN ((p+q)*h)/2.0;
   END;
ENDCLASS Triangle;
```

Beispiel-Quellcode "example2.gos"

```
CLASS EllipseObject SUBCLASSOF GraphicObject;
PROTECTED
   center :: Point;

PUBLIC
   -- abstract constructor
   NEW() =
      BEGIN
```

```
      END;

   -- abstract
   xRadius() Float =
      BEGIN
      END;

   -- abstract
   yRadius() Float =
      BEGIN
      END;

   area() Float =
      VAR
         xr, yr :: Float;
      BEGIN
         xr := self!xRadius();
         yr := self!yRadius();
         RETURN xr*yr*pi;
      END;

   move(delta :: Point) =
      BEGIN
         center := center + delta;
      END;

   isCycle() Bool =
      VAR
         xr, yr :: Float;
      BEGIN
         xr := self!xRadius();
         yr := self!yRadius();
         RETURN xr == yr;
      END;
ENDCLASS EllipseObject;


CLASS Ellipse SUBCLASSOF EllipseObject;
PROTECTED
   xr, yr :: Float;

PUBLIC
   -- constructor
   NEW(iCenter :: Point, ixr :: Float, iyr :: Float) =
      BEGIN
         center := iCenter;
         xr := ixr;
         yr := iyr;
      END;

   -- redefine
   xRadius() Float =
      BEGIN
         RETURN xr;
      END;

   -- redefine
```

```
   yRadius() Float =
      BEGIN
         RETURN yr;
      END;
ENDCLASS Ellipse;


CLASS Cycle SUBCLASSOF EllipseObject;
PROTECTED
   radius :: Float;

PUBLIC
   -- constructor
   NEW(iCenter :: Point, iRadius :: Float) =
      BEGIN
         center := iCenter;
         radius := iRadius;
      END;

   -- redefine
   xRadius() Float =
      BEGIN
         RETURN radius;
      END;

   -- redefine
   yRadius() Float =
      BEGIN
         RETURN radius;
      END;

   -- redefine
   isCycle() Bool =
      BEGIN
         RETURN True;
      END;
ENDCLASS Cycle;
```

Literaturverzeichnis

1. Bauer, F. L., Goos, G.: *Informatik, eine einführende Übersicht, Teil 2*, Springer-Verlag, 1984

2. Booch, G.: *Object Oriented Analysis and Design with Applications*, The Benjamin/Cummings Publishing Company, 1994

3. Broy, M.: *Informatik, Eine grundlegende Einführung, Teil IV*, Springer Verlag, 1995

4. Clasohm, C.: *Scanner und Parser-Generator für das Gofer Objekt System*, TUM-Diplomarbeit, 15. November 1995

5. Engel, R.: *Objektorientierte Programmierung, Eine Einführung*, Markt&Technik, 1990

6. Ghezzi, C., Jazayeri, M.: *Konzepte der Programmiersprachen, Begriffliche Grundlagen, Analyse und Bewertung*, R. Oldenbourg Verlag GmbH, München, 1989

7. Gruschke, B.: *Tutorial und Referenzhandbuch für GOS*, TUM-Diplomarbeit, 15. Mai 1996

8. Hopcroft, J. E., Ullman, J. D.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*, Addison-Wesley (Deutschlang) GmbH, 1993

9. Jones, M. P.: *An Introduction to Gofer*, draft version, May 22, 1994

10. Krüger, I., Sihling, M.: *Fortgeschrittenen-Praktikum, Implementierung der Laufzeitkomponente des GOS-Interpreters*, TUM-Fopra, 30. März 1995

11. Meyer, B.: *Objektorientierte Software-Entwicklung*, Hanser Verlag, 1990

12. Rumpe, B.: *Imperativ Objektorientierte und Funktionale Programmierung in einer Sprache vereint*, Kolloquium Programmiersprachen und Grundlagen der Programmierung, Adalbert Stifter Haus, Alt Reichenau, 11.10.1995, Tiziana Margaria (Hrsg.), MIP Bericht N. 9519, Dez. 1995, Universität Passau, Fakultät für Mathematik und Informatik

13. Schünemann U.; Bichler K.: *Praktische Semesterarbeit, Parser und operationelle Sematik für GOS*, TUM-Fopra, 1994

14. Wirth N.: *Compilerbau*, Teubner Studienbücher Informatik, B.G. Teubner, Stuttgart, 1986