TUM

INSTITUT FÜR INFORMATIK

Frisco F - Eine funktionale, logische und algebraische Spezifikationssprache

Christian Lesny
Franz Huber
Bernhard Rumpe

TUM-I9901
Januar 99

TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM-INFO-1-I9901-2/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Technische Universität München

Institut für Informatik

Technischer Bericht

„Frisco F“

Eine funktionale, logische und algebraische Spezifikationssprache

Sprachbeschreibung und Parserimplementierung[1]

Christian Lesny
Franz Huber
Bernhard Rumpe

[1]Dieser technische Bericht entstand im Projekt SysLab, finanziert mit Unterstützung der DFG unter dem Leibnizpreis und durch die Firma Siemens.

Inhaltsverzeichnis

Kapitel 1

Einführung und Motivation

Die Entwicklung von Softwaresystemen ist eine komplexe Aufgabe, die es erfordert, ähnlich wie in anderen Ingenieurdisziplinen, zunänchst Konstruktionspläne des zu Produkts zu erstellen. Die immaterielle Natur und die große Komplexität heutiger Softwaresysteme erfordert die Verwendung unterschiedlicher Sichten und Beschreibungstechniken. Besonders graphische Beschreibungstechniken, wie sie die Unified Modeling Language (UML) [BRJ98] bietet, spielen dabei eine wesentliche Rolle. Dabei wird aber gerne übersehen, daß eine rein graphische Beschreibung entweder keine Beschreibungsvollständigkeit oder eine nicht mehr überblickbare Komplexität aufweist.

Deshalb wurde in dieser Arbeit der Versuch gemacht, eine textuelle Sprache zu definieren, die als Addendum zu einer graphischen Notation verwendet werden kann. Das Ergebnis ist eine Sprache, die die Vorteile funktionaler Sprachen, wie etwa deren Kompaktheit und Ausführbarkeit, mit einem prädikatenlogischen Teil erweitert, der es erlaubt abstrakte Eigenschaften zu spezifizieren. Ein in den funktionalen Teil integrierter algebraischer Teil erlaubt die Definition von Datentypen und darauf arbeitenden Funktionen, wie sie aus der algebraischen Spezifikation bekannt sind.

Die Geschichte der funktionalen Sprachen reicht zurück in die 30er Jahre mit den Arbeiten von Church und Kleene über den λ-Kalkül. Der Formalismus des λ-Kalküls kann als Kern jeder funktionalen Sprache angesehen werden. Auch in der Sprache LISP, die 1960 von McCarthy vorgestellt wurde, findet sich die λ-Notation wieder. Die Sprache Haskell [Bir89] und ihr Derivat Gofer [Jon94b] sind bisherige Höhepunkte in der Entwicklung von funktionalen Sprachen.

Die Object Constraint Language (OCL) [War98] ist als Teil der UML ebenfalls eine Ergänzung graphischer Darstellungen, bietet aber weder eine ausreichende Menge von Sprachkonzepten noch ist die Semantik genügend präzise formuliert. Darüberhinaus ist die OCL syntaktisch sehr unangenehm, während „Frisco F“ wesentlich kompakter ist und sich damit wesentlich besser für eine Integration in graphische Darstellungen eignet.

1.1 Zusammenfassung der Eigenschaften von „Frisco F“

Das Projekt „SysLab“ entwickelt und verbessert Techniken und Notationen zur Modellierung und Spezifikation von Softwaresystemen. In diesem Rahmen werden unter anderem Konzepte wie beispielsweise Objektmodelle, Interaktionsdiagramme, Automaten eingesetzt. Als Ergänzung ist hierfür eine funktionale Sprache hilfreich, die die Definition von Datentypen, Funktionen und prädikatenlogischen Ausdrücken, wie zum Beispiel Vor- und Nach-bedingungen, erlaubt. Die funktionale Programmiersprache Gofer erwies sich als geeignete Grundlage, da beispielsweise die komfortabel verwendbaren Pattern und die zur Verfügung stehenden Listen dazu genutzt werden können, Bedingungen in den von *Syslab* benutzten Beschreibungstechniken zu formulieren. Die Sprache Gofer, die von M. P. Jones entwickelt wurde und der Sprache Haskell syntaktisch und semantisch ähnlich ist, kann nicht unmittelbar eingesetzt werden. Stattdessen sind einige Modifikationen und Erweiterungen notwendig, um den Spezifikationscharakter stärker zu betonen. Der sich hieraus ergebenden Spezifikationssprache wurde der Name „Frisco F“ gegeben. Die Entwicklung der Sprache „Frisco F“ umfaßt folgende Modifikationen und Erweiterungen gegenüber der Sprache Gofer:

- Für die Sprache „Frisco F“ wird die sogenannte Layout-Rule von Gofer weggelassen, so daß die Sensitivität bzgl. Einrückung aufgegeben wird. Die Layout-Rule von Gofer erlaubt, die Struktur eines Programms durch Einrückungen auszudrücken. Auf diese Weise werden implizit geschweifte Klammern und Semikola nach vorgegebenen Regeln eingefügt, die ohne Verwendung von Einrückungen explizit im Quellcode angegeben werden müssen.

- Typklassen werden nicht übernommen. Dies ermöglicht eine Vereinfachung der Grammatik und des Typsystems, insbesondere des Typinferenz-Algorithmus. Es ist allerdings ein Minimum an Überladung für elementare Funktionen wie `==` oder für arithmetische Operationen wie `+` möglich. Daher wird ad-hoc-Polymorphie in einem eng begrenzten und nicht erweiterbaren Rahmen zur Verfügung stehen.

- Die Sprache wird um spezifikatorische Konzepte erweitert. In der Top-Level-Ebene werden Operator- und Axiom-Definitionen zur Formulierung von Axiomen eingeführt. Zugleich wird die Grammatik um prädikatenlogische Ausdrücke erweitert, insbesondere um Quantoren und Junktoren.

- Als weitere Eigenschaft wird eine strikte Semantik für „Frisco F“ festgelegt. Die in Gofer erlaubte Notation `[a,b..]` zur Erzeugung „unendlicher“ Listen und unendliche Datenstrukturen im allgemeinen sind in „Frisco F“ nicht möglich. Aus diesem Grund werden unendliche arithmetische Listen in der Grammatik nicht übernommen.

- Eine Reihe kleinere Modifikationen, wie zum Beispiel die partielle und damit erweiterbare Definition von Datentypen, sind in der Sprachbeschreibung angegeben.

1.2 Ergebnisse

In diesem Abschnitt werden die Ergebnisse des vorliegenden technischen Berichts zusammenfassend vorweggenommen. Als Ergebnis enthält dieser Berichts die Beschreibung der Sprache „Frisco F“, sowie des zugehörigen abstrakten Syntaxbaums und die Implementierung eines Parsers einschließlich der notwendigen Kontextchecks. Die Grammatik von „Frisco F“ basiert auf der Grammatik von Gofer, die entsprechend der oben erwähnten Änderungen modifiziert wird. Die Entwicklung des Parsers teilt sich in die lexikalische, syntaktische und kontextabhängige (auch “semantische”) Analyse. Die ersten beiden Schwerpunkte werden mit Hilfe von Scanner- und Parser-Generatoren realisiert. Hierzu wird eine kontextfreie Grammatik der Sprache „Frisco F“ angegeben. Für die semantische Analyse werden die notwendigen Kontextbedingungen aufgestellt und deren Implementierung beschrieben. Insbesondere wird auf die Typinferenz eingegangen, die bereits in Gofer erhebliche Komplexität besitzt. Als

Implementierungssprache wurde die objektorientierte Programmiersprache Java ausgewählt, da Java-Programme auf vielen Plattformen ausführbar sind.

1.3 Aufbau des Berichts

In diesem Abschnitt wird der Aufbau des vorliegenden Berichts kurz erläutert.

- **Grundlagen**
  In Kapitel 2 werden die wichtigsten Begriffe der funktionalen Programmierung kurz erläutert. Anschließend werden die in der Arbeit verwendeten Grundlagen des Übersetzerbaus und insbesondere die Übersetzung von funktionalen Sprachen kurz umrissen.

- **Sprachbeschreibung**
  Das Kapitel 3 bietet eine einführende, informelle Beschreibung aller Schwerpunkte von „Frisco F“. Ein anschließender Vergleich zeigt Änderungen und Ergänzugen gegenüber der Programmiersprache Gofer auf.

- **Grammatik**
  Nach einer kurzen Beschreibung der verwendeten Notation wird die kontextfreie Grammatik von „Frisco F“ in Kapitel 4 vorgestellt. Die gewählte Notation ist an die Notation von Gofer angelehnt. Das Kapitel schließt mit einer Beschreibung der Besonderheiten, die in der Grammatik von „Frisco F“ auftreten.

- **Lexikalische und syntaktische Analyse**
  In Kapitel 5 werden lexikalische und syntaktische Analyse kurz erläutert. Insbesondere wird auf die bei der Implementierung verwendete Grammatik eingegangen. Diese ist gegenüber der dokumentierten Grammatik modifiziert, um Probleme bei der Erzeugung des Parsers zu vermeiden (LALR(1) und ähnliche Probleme).

- **Semantische Analyse**
  Die Kontextprüfungen der semantische Analyse sind in Kapitel 6 beschrieben. Dabei wird zuerst auf die Prüfungen eingegangen, die schon während dem Parsen durchgeführt werden. Anschließend werden die restlichen Regeln beschrieben, die nach dem Parsen zu prüfen sind.

- **Typsystem und Typinferenz**
  In Kapitel 7 wird zuerst das Typsystem von „Frisco F“ vorgestellt. Anschließend werden die Grundlagen der Typinferenz dargestellt. Dies beinhaltet unter anderem den Unifikations-Algorithmus und den Typinferenz-Algorithmus.

- **Implementierung**
  Nach einem kurzen Überblick wird in Kapitel 8 die Implementierung anhand ausgewählter Schwerpunkte diskutiert. Dies betrifft im einzelnen den Aufbau und die Funktionsweise des „Frisco F“-Parsers, den generierten Scanner und Parser einschließlich des Tidy-Infix-Algorithmus, die Symboltabelle, den Algorithmus zur Berechnung stark zusammenhängender Komponenten gerichteter Graphen und die Implementierung der Typ- und Kindinferenz.

Eine Zusammenfassung und ein kurzer Ausblick sowie mehrere Anhänge und das Literaturverzeichnis schließen den vorliegenden Bericht ab.

Kapitel 2

Grundlagen

2.1 Funktionale Programmierung mit Gofer

2.1.1 Funktionale Programmierung

In diesem Abschnitt werden die wichtigsten Begriffe, die im Zusammenhang mit funktionaler Programmierung auftreten, kurz beschrieben. Für eine ausführliche Einführung sei auf [Bir89] und [Pau96] verwiesen.

- **funktional $\Leftrightarrow$ imperativ**

  Der Programmierstil imperativer Sprachen besteht darin, Sequenzen von Anweisungen mit Hilfe geeigneter Kontrollstrukturen für die Ausführung und Wiederholung zu arrangieren. Die Essenz der funktionalen (oder applikativen) Programmierung hingegen liegt darin, zumeist einfache Funktionen zu komplexeren zu kombinieren (vgl. [Ghe89], S. 381).

- **Referenzielle Transparenz**

  Eine wesentliche Eigenschaft funktionaler Sprachen ist, daß sie die referenzielle Transparenz der Mathematik erhalten. Dies bedeutet, daß Funktionen beliebig hierarchisch kombiniert werden können. Darin liegt der Unterschied zwischen funktionalen und imperativen Sprachen. Nach [Sto77] kann das Prinzip der referenziellen Transparenz („referential transparency“ nach Russell[1]) wie folgt zusammengefasst werden:

  > „Ein Ausdruck wird nur verwendet, um einen Wert zu benennen. In einem gegebenen Zusammenhang bezeichnet ein Ausdruck immer den gleichen Wert. Aus diesem Grund können Teilausdrücke durch andere mit dem gleichen Wert ersetzt werden (Substitutionsprinzip).“

  Das Substitutionsprinzip stellt ein wichtiges Merkmal der mathematischen Beweisführung dar. Im Kontext funktionaler Sprachen erleichtert dieses Merkmal u. a. den Nachweis von Programmeigenschaften oder die Transformation von Programmen (vgl. [Ghe89], S. 380 ff. und [Hin92], S. 3).

[1] B. Russell, Logiker und Philosoph. Leitete die erste Krise der Logik ein mit dem Russell-Paradox der Menge $R := \{M | M \notin M\}$ (1901): ist $R \in R$?

- **Funktionen höherer Ordnung**

  Funktionen sind Werte wie Zahlen und Listen (Objekte 1. Ordnung), so daß sie auch als Argumente an Funktionen übergeben und als Ergebnis zurückgegeben werden können. Funktionen, die funktionale Argumente erwarten oder „berechnen“, heißen Funktionen höherer Ordnung (vgl. [Hin92], S. 5).

- **lazy ⇔ eager**

  Die strikte Auswertung („eager evaluation“) von Werten nennt man „call by value“. Ausdrücke werden bei der Übergabe an Funktionen immer sofort berechnet. Die nicht-strikte Auswertung („lazy evaluation“, wörtlich: „faule Auswertung“) von Werten nennt man „call by need“. Teilausdrücke werden nur dann berechnet, wenn sie zur Berechnung notwendig sind (vgl. [Hin92], S. 5). Nicht-strikte Sprachen bieten die Möglichkeit, auch unendliche Datenstrukturen elegant zu bearbeiten (vgl. [Pau96], S. 191).

- **monomorph ⇔ polymorph**

  In einem monomorphen Typsystem besitzt jedes Objekt der Sprache genau einen Typ. Dies ist zum Beispiel bei listenverarbeitenden Funktionen hinderlich, da der Typ der Listenelemente in den Argument- und Ergebnistypen festgelegt werden muß. Somit muß für jeden verwendeten Elementtyp eine eigene Variante programmiert werden (vgl. [Hin92], S. 5, Fußnote 1).
  Einen möglichen Ausweg bieten generische Funktionen oder generische Klassen in objektorientierten Sprachen. In C++ werden generische Funktionen und Klassen Templates genannt (vgl. [Mey90], S. 105 ff. und [Str92], S. 275 ff.).
  In einem polymorphen Typsystem kann ein Objekt mehrere oder unendlich viele Typen besitzen. Zum Beispiel kann der Typ einer Liste definiert werden, deren Elementtyp nicht näher spezifiziert ist. So muß etwa die Funktion $length$ zur Berechnung der Länge einer Liste nicht auf die Elemente selbst zugreifen. Daher kann als Argument eine Liste mit beliebigem Elementtyp übergeben werden, d.h der Argumenttyp ist polymorph. Ein weiteres Beispiel ist die Identität id definiert durch $id\ x = x$, die angewandt auf ein beliebiges Objekt dieses unverändert zurückgibt. Die Signatur der Funktion id lautet $a \rightarrow\ a$. Der Argumenttyp ist beliebig und somit polymorph, und der Ergebnistyp stimmt mit dem Argumenttyp überein.

- **Ad-hoc-Polymorphie**

  Als Ad-hoc-Polymorphie wird die Möglichkeit bezeichnet, Funktionsnamen zu überladen. So kann zum Beispiel die Funktion + überladen sein, so daß sie für Ganzzahlen (Typ `Int`) oder Gleitkommazahlen (Typ `Float`) anwendbar ist.

- **Typinferenz**

  Typinferenz ist die automatische Herleitung des Typs eines Ausdrucks oder einer Funktion durch das System. Somit ist eine Programmierung ohne explizite Typisierung möglich, wobei das System mittels Typinferenz bereits zur Kompilierzeit die mitunter polymorphen Typen aller Größen kennt.

2.1.2 Die Programmiersprache Gofer

Gofer ist eine funktionale oder applikative Programmiersprache und wurde von Mark Philip Jones entwickelt. Für eine Einführung in Gofer sei auf [Jon94b] verwiesen. Gofer ist eine Weiterentwicklung von Haskell (vgl. [Hud91] und [Thi94], S. 158).
Die folgenden Punkte geben eine knappe Charakterisierung der Programmiersprache Gofer (vgl. [Jon94b], S. 6):

- Es ist möglich, Funktionen höherer Ordnung zu definieren.
- Als Auswertungsstrategie wird Lazy-Evaluation verwendet, im Gegensatz zur strikten Auswertung (eager evaluation).
- Gofer verfügt über ein polymorphes Typsystem.
- In Gofer wird Ad-hoc-Polymorphie mit Hilfe von Typklassen realisiert.
- Typklassen können auch mehrere Parameter haben.
- Es können Instanzen für beliebige Typen definiert werden.
- Gofer verfügt über Typinferenz.
- Gofer unterstützt benutzerdefinierte Datentypen und Typsynonyme.
- In Gofer können Funktionen und Ausdrücke mit Pattern-Matching definiert werden.
- Gofer unterstützt List-Comprehensions.
- Gofer stellt Konzepte zur Ein- und Ausgabe zur Verfügung.

2.1.3 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 (vgl. [Ghe89], S. 412 f.)
Ein weiteres Problem bei funktionalen Sprachen sind Ein- und Ausgabe, da dies Seiteneffekte sind. 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 der Programmverifikation,

zugänglich.
Zwar ist es relativ unwahrscheinlich, daß in absehbarer Zeit funktionale Programmiersprachen in industriellem Rahmen eingesetzt werden. Allerdings sind sie für Spezifikationssprachen sehr geeignet, mit denen ein Problem knapp, präzise und mit angemessenem Abstraktionsgrad formuliert werden kann. Aber auch im Bereich des „rapid prototyping“ können funktionale Sprachen sinnvoll eingesetzt werden (vgl. [Hin92], S. 2).

2.2 Grundlagen des Übersetzerbaus

Der folgende Abschnitt gibt einen knappen Überblick, wobei sich die Darstellung stark an [Poe94] anlehnt. Ansonsten sei auf [Aho88] und [Wil96] verwiesen.

Das Grundproblem des Übersetzerbaus ist das Auffinden einer Funktion (bzw. seiner Implementierung), die ein Quellprogramm einer höheren Programmiersprache (QS) in ein Zielprogramm einer maschinennahen Sprache (ZS) abbildet:

$$comp : QS \to ZS, P_{QS} \mapsto comp(P_{QS}) = P_{ZS}$$

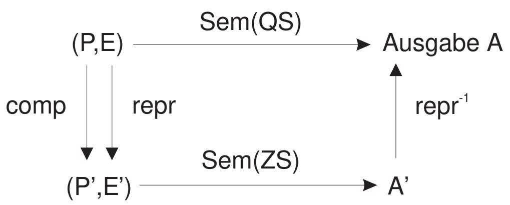
Die Semantik des übersetzten Programms muß dabei erhalten bleiben. Definieren die Funktionen sem_{QS} und sem_{ZS} die Semantik der Sprachen QS und ZS, so muß zusätzlich für jede Eingabe E gelten:

$$sem_{QS}(P_{QS}, E) = sem_{ZS}(comp(P_{QS}), E)$$

Berücksichtigt man zusätzlich, daß die Darstellung der Daten von den Sprachen abhängig ist, so muß außerdem eine bijektive Funktion $repr$ eingeführt werden:

$$repr : Data_{QS} \to Data_{ZS}, D_{QS} \mapsto repr(D_{QS}) = D_{ZS}$$

Somit gilt für alle Programme $P_{QS} \in QS$ und alle zulässigen Eingaben $E \in Data_{QS}$ für P_{QS}:



Zudem gilt es weitere Anforderungen zu berücksichtigen:

- Die Zielprogramme sollen möglichst effizient sein.
- Die Übersetzungsfunktion muß für Fehlerbehandlung zur Übersetzungs- und Laufzeit sorgen.
- Es sollte eine Wahlmöglichkeit zwischen schnellem Übersetzten mit ggf. wenig effizientem Zielcode und langsamer Übersetzung mit stark optimiertem Zielcode angeboten werden.

2.2.1 Der Übersetzungsprozeß

Konzeptionell kann der Übersetzungsprozeß grob in zwei Teile zerlegt werden. Der erste Teil ist die Analyse-Phase und der zweite Teil ist die Synthese-Phase. Beide Teile zergliedern sich wiederum in mehrere Teilaufgaben. Abbildung 2.1 zeigt die Phasen des Übersetzungsprozesses.

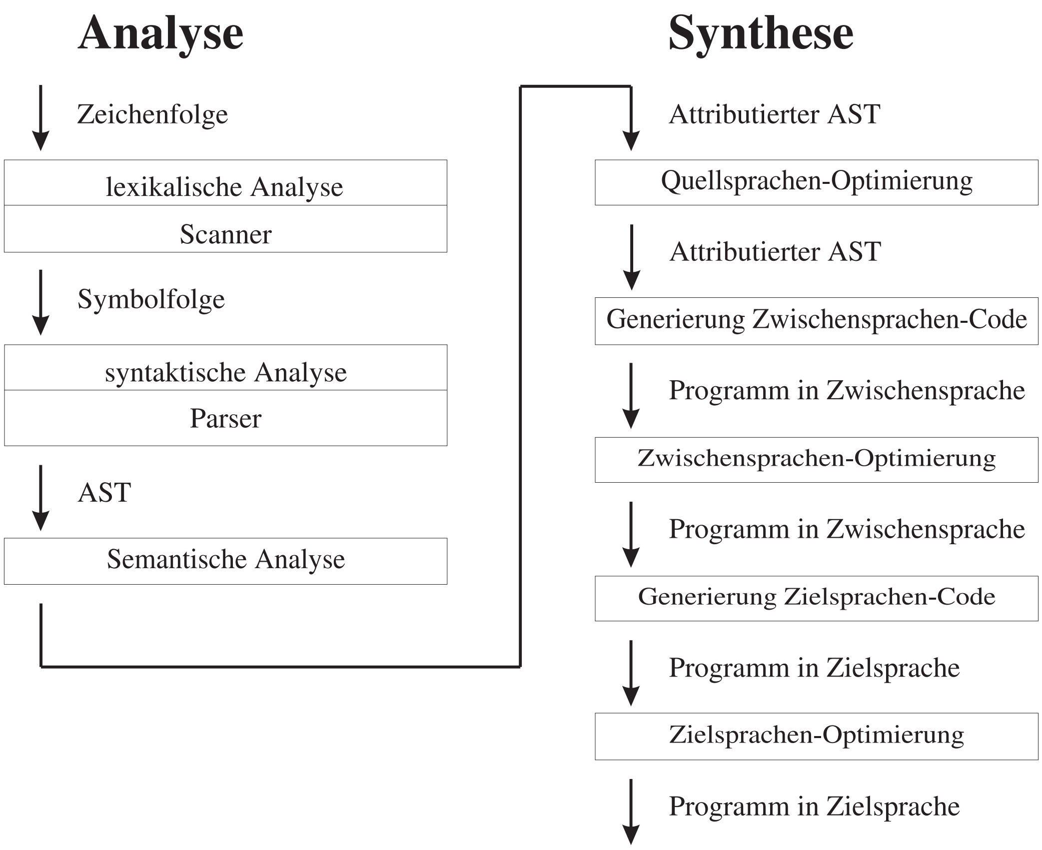


Abbildung 2.1: Die Phasen des Übersetzungsprozesses

Analyse-Phase

- *Lexikalische Analyse*
  Die lexikalische Analyse wird von einem Modul ausgeführt, welches gewöhnlich als Scanner bezeichnet wird. Seine Aufgabe ist es, den Eingabestrom zu zerhacken und in einen Symbolstrom umzuwandeln. Typische Symbole sind zum Beispiel Literale, Bezeichner oder Operatoren wie := oder <=.
  Kommentare und Zwischenzeichen, die für die nachfolgende Verarbeitung irrelevant sind, werden ignoriert (vgl. [Wil96], S. 224).

- *Syntaktische Analyse*
  Durch die syntaktische Analyse wird die Struktur eines Quellprogramms aufgedeckt. Das zugehörige Modul wird als Parser bezeichnet. Der Parser kennt die Grammatik der Quellsprache und somit die Struktur von Ausdrücken, Deklarationen, Listen usw. Die Eingabe des Parsers ist der Symbolstrom des Scanners, und die Ausgabe ist ein abstrakter Syntaxbaum. Der Parser muß zudem syntaktische Fehler im Quellprogramm

entdecken, analysieren und dem Anwender mitteilen (vgl. [Wil96], S. 227).

- *Semantische Analyse*
  Die semantische Analyse beschäftigt sich mit der Berechnung und Prüfung von Programmeigenschaften, die allein durch das Quellprogramm ermittelt werden können. Diese Eigenschaften werden als statisch bezeichnet. Beispielsweise kann zur Übersetzungszeit festgestellt werden, ob ein verwendeter Bezeichner aufgrund einer entsprechenden Deklaration sichtbar ist. Im Gegensatz dazu gibt es Eigenschaften, die erst zur Laufzeit des Programms ermittelt werden können. Solche Eigenschaften, wie zum Beispiel die maximale Rekursionstiefe eines Programms, werden als dynamisch bezeichnet. Die zwei wichtigsten Aufgaben der semantische Analyse sind:

  - *Bindungsanalyse*
    Welche Deklaration bzw. welches Programmelement ist mit einem angewandten Bezeichner gemeint?
    Sind alle angewandten Bezeichner deklariert?
    Binden eines Bezeichners an seine Deklaration.
  - *Typanalyse*
    Welchen Typ hat eine Ausdruck, eine Funktion usw.
    Passen die Typen „benachbarter“ Programmteile zusammen, sind z.B. die Typen der linken und rechten Seite einer Zuweisung zuweisungskompatibel?

  Durch die semantische Analyse wird der Syntaxbaum mit semantischen Informationen angereichert und ggf. modifiziert. Als Ergebnis erhält man einen attributierten Baum.

Synthese-Phase

Da sich der vorliegende Bericht ausschließlich mit Problemstellungen der Analyse-Phase beschäftigt, werden die Teilphasen der Synthese-Phase hier nur kurz erläutert.

- *Quellsprachen-Optimierung*
  In dieser Teilphase wird in der Regel eine Datenflußanalyse durchgeführt. Unter die vielfältigen Methoden der Optimierung fallen zum Beispiel die Elimination von „totem Code“ (*dead code elimination*), d.h. das Entfernen von Programmteilen, die zur Ausführung nicht benötigt werden, das „Herausziehen“ von schleifeninvarianten Ausdrücken aus Schleifen oder die Berechnung von konstanten Ausdrücken zur Übersetzungszeit (*constant folding*).

- *Übersetzung in ZS-nahe Zwischenform*
  Das ggf. optimierte Quellprogramm wird in eine Zwischensprache übersetzt, die leicht zu erzeugen und leicht ins Zielprogramm zu übersetzen sein sollte.

- *Zwischensprachen-Optimierung*
  In dieser Teilphase wird versucht, den Zwischencode zu verbessern. Optimierungsmöglichkeiten ergeben sich beispielsweise durch das Übersetzen von Sprachkonstrukten mit Hilfe von Codeschablonen. Viele der in der Quellsprachen-Optimierung verwendeten Methoden können auch hier eingesetzt werden.

- *Code-Generierung*
  Die Code-Generierung übersetzt den Zwischencode in ein Programm der Zielsprache (in der Regel Maschinencode).

- *Zielsprachen-Optimierung*
  Die Zielsprachen- und damit maschinenabhängige Optimierung versucht, durch lokale Betrachtungen des Zielprogramms Verbesserungen herbeizuführen. Dies wird auch als Peephole-Optimierung bezeichnet, da man sich vorstellen kann, ein kleines Fenster über den Zielcode zu bewegen und die Befehle im betrachteten Abschnitt ggf. durch andere, bessere Befehle zu ersetzen. Speziell handelt es sich um die Elimination überflüssiger Befehle und die Ersetzung von Befehlen durch effizientere Befehle.

2.2.2 Übersetzung funktionaler Sprachen

Auch in diesem Abschnitt werden wir uns auf die Analyse-Phase beschränken und somit insbesondere Code-Erzeugung und Laufzeitsystem übergehen. Aber auch in der Analyse-Phase beschränken wir uns auf die semantische Analyse, da lexikalische und syntaktische Analyse auch für funktionale Sprachen keine besonderen Schwierigkeiten (bezüglich der zugrundeliegenden Theorie) mit sich bringen.
Zu den wichtigsten Aufgaben der semantische Analyse gehören Bindungs- und Typanalyse. Die Bindungsanalyse soll unter anderem die korrekte Anwendung von Funktionsdefinition und Funktionsapplikation sicherstellen und die auftretenden Sichtbarkeitsbereiche von verschachtelten Definitionen korrekt behandeln. Jedem Bezeichner muß seine den Sichtbarkeitsregeln zugehörige Definition zugeordnet werden. Soweit möglich, müssen die dadurch entstehenden Einschränkungen überprüft werden. Die Typanalyse muß sicherstellen, daß alle Konstrukte in einem Programm typisiert werden können und insbesondere die auftretenden Teile zueinander typkompatibel sind. Wird ein Typinferenzalgorithmus verwendet, so übernimmt dieser den größten Teil der Typanalyse.
Viele Konzepte einer Sprache werden in dieser Phase nicht berührt. So ist beispielsweise die Wahl zwischen eager evaluation und lazy evaluation oder die Technik der Speicherverwaltung und -bereinigung in der Analyse-Phase irrelevant.

Kapitel 3

Sprachbeschreibung

Die Sprache „Frisco F“ ist eine Adaption der Sprache Gofer, wobei diese größtenteils übernommen wurde. Diese Teile werden in Abschnitt 3.1 dargestellt. Für eine ausführlichere Beschreibung der Sprache Gofer sei auf [Jon94b] verwiesen. Die Erweiterungen und Änderungen werden im Abschnitt 3.2 ausführlich erläutert. Die Beschreibung bezieht sich auf die Version 1.0 von „Frisco F“.

3.1 Informelle Beschreibung der Sprache

Im folgenden wird die Sprache „Frisco F“ beschrieben. Dieser Abschnitt lehnt sich stark an [Jon94b] an, da weite Teile der Sprache Gofer in die Sprache „Frisco F“ übernommen wurden. Die Beschreibung der Unterschiede erfolgt dann im anschließenden Abschnitt.

3.1.1 Bezeichner und Operatoren

Bezeichner

Ein Bezeichner ist ein Buchstabe, gefolgt von einer mitunter leeren Folge von Zeichen, von denen jedes entweder ein Buchstabe, eine Ziffer, ein Apostroph (`'`) oder ein Underscore (`_`) ist. Bezeichner, die mit einem kleinen Buchstaben beginnen, werden als „varid“ bezeichnet. Sie stehen für Funktions- und Variablennamen. Bezeichner, die mit einem großen Buchstaben beginnen, werden als „conid“ bezeichnet. Dieses stehen für sogenannte „constructor functions“ und werden auch für Datentyp- und Typsynonymnamen verwendet.
Beispiele für varids:

```
sum   f   f''   a_b
```

Beispiele für conids:

```
Tree   Leaf   String
```

Tabelle 3.1 zeigt eine Auflistung aller reservierten Wörter. Unter den reservierten Wörtern finden sich auch die ASCII-Ersatzzeichen, die in Tabelle 5.1 auf Seite 41 aufgelistet sind.

Operatoren

Ein Operator kann aus Zeichen zusammengesetzt werden, die in Tabelle 3.2 aufgelistet sind. Ähnlich wie bei Bezeichnern wird auch bei Operatoren zwischen zwei Arten unterschieden. Operatoren, die nicht mit einem Doppelpunkt beginnen, werden als „varop“ bezeichnet und

stehen für Funktionsnamen. Hingegen werden Operatoren, die mit einem Doppelpunkt beginnen, als „constructor functions“ verwendet und als „conop“ bezeichnet.
Beispiele für varops:

`+` `>-->` `!!` `-*-` `<+>`

Beispiele für conops:

`:^:` `:+` `:->` `:-:` `::>`

Die Tabelle 3.3 gibt eine Auflistung aller reservierten Operatoren.

ax	case	data	else	if
in	infix	infixl	infixr	let
of	op	then	type	where
ALL	ALLB	ALLP	AND	BOT
DEF	EX	EXB	EXP	FF
NOT	OR	TT		

Tabelle 3.1: Reservierte Wörter von „Frisco F“

<table>
<tr><td>:</td><td>!</td><td>#</td><td>$</td><td>%</td></tr>
<tr><td>&</td><td>*</td><td>+</td><td>.</td><td>/</td></tr>
<tr><td><</td><td>=</td><td>></td><td>?</td><td>@</td></tr>
<tr><td>\</td><td>|</td><td>-</td><td>^</td><td></td></tr>
</table>

Tabelle 3.2: Operatorzeichen

<table>
<tr><td>::</td><td>=</td><td>..</td><td>@</td><td>\</td><td>|</td></tr>
<tr><td><-</td><td>-></td><td>=></td><td><=></td><td>!</td><td>.</td></tr>
</table>

Tabelle 3.3: Reservierte Operatoren von „Frisco F“

Verwendung von Operatoren als Bezeichner und umgekehrt

Jeder Operator kann syntaktisch als Bezeichner verwendet werden, sofern dieser geklammert wird. So wird z.B. `+` im Ausdruck `2 + 3` als Operator verwendet. Hingegen wird `(+)` wie ein Funktionsname behandelt, so daß der obige Ausdruck äquivalent ist zu `(+) 2 3`.
Analog können Funktionsnamen als Operatoren verwendet werden, wenn sie in Akzentzeichen (‘) eingeschlossen werden. So kann z.B. der Ausdruck `div 2 3` auch als `2 ‘div‘ 3` geschrieben werden.

Präzedenz und Assoziativität

Jeder Operator besitzt eine Präzedenz zwischen 0 und 9. Da `*` gegenüber `+` in der Regel eine höhere Präzedenz besitzt, wird z.B der Ausdruck `2 + 4 * 6` als `2 + (4 * 6)` und nicht als `(2 + 4) * 6` interpretiert.
Zudem ist für jeden Operator die Assoziativität festgelegt. Ist ein Operator $\oplus$ links-assoziativ, so wird z.B. der Ausdruck $\texttt{x} \oplus \texttt{y} \oplus \texttt{z}$ als $\texttt{(x} \oplus \texttt{y)} \oplus \texttt{z}$ interpretiert. Ist ein Operator $\oplus$ rechts-assoziativ, so ist der obige Ausdruck gleichwertig zu $\texttt{x} \oplus \texttt{(y} \oplus \texttt{z)}$. Es ist auch möglich, daß einem Operator keine bestimmte Assoziativität zugewiesen wird. In einem solchen Fall ist der obige Ausdruck mehrdeutig und wird als Syntaxfehler zurückgewiesen.
Die Präzedenz und Assoziativität kann mit den Top-Level-Definitionen `infix`, `infixl` und `infixr` explizit angegeben werden. Andernfalls wird die höchste Präzedenz (9) und keine Assoziativität angenommen.
Auch Funktionen mit Bezeichnern besitzen eine Präzedenz und eine Assoziativität, da sie ebenfalls als Operatoren verwendet werden können.
Die Funktionsanwendung hat immer eine höhere Präzedenz als ein Infix-Operator. So wird z.B. der Ausdruck `f x + 1` als `(f x) + 1` und nicht als `f (x + 1)` interpretiert.

3.1.2 Standard-Datentypen

Funktionen

Sind t_1 und t_2 Typen, dann ist $t_1 \rightarrow\ t_2$ eine Funktion von t_1 nach t_2. Ist f vom Typ $t_1 \rightarrow\ t_2$ und x vom Typ t_1, so ist fx die Anwendung von x auf f und ergibt einen Wert vom Typ t_2. Sind $t_1, t_2, ..., t_n$ Typen, so ist $t_1 \rightarrow\ t_2 \rightarrow\ ... \rightarrow\ t_n$ eine Abkürzung für $t_1 \rightarrow\ (t_2 \rightarrow\ (... \rightarrow t_n)...)$. Der Ausdruck $fx_1x_2...x_n$ ist eine Abkürzung für $(...((fx_1)x_2)...x_n)$.
Zum Beispiel ist `(+)` eine Funktion vom Typ `Int -> Int -> Int`, und `(+) 3` ist eine Funktion vom Typ `Int -> Int`, die eine Zahl i auf $3\ +\ i$ abbildet. Der Ausdruck `(+) 3 5` ist vom Typ `Int` und ergibt den Wert 8.

Typ `Bool`

Der Typ `Bool` repräsentiert die Wahrheitswerte `True` und `False`. Es existieren unter anderem die folgenden Standard-Funktionen:

```
(&&), (||)  ::  Bool -> Bool -> Bool
not         ::  Bool -> Bool
```

Die bedingte Anweisung

`if` b `then` t `else` e

ist unmittelbar an den Typ `Bool` geknüpft, da der Ausdruck b eines `if`-Ausdrucks vom Typ `Bool` sein muß.

Typ `Int` und Typ `Float`

Der Typ `Int` repräsentiert Ganzzahlen, deren Wertebereich beschränkt ist. Der Typ `Float` repräsentiert Gleitkommazahlen, deren Wertebereich und Genauigkeit ebenfalls beschränkt sind. Gleitkommazahlen werden am Dezimalpunkt erkannt (z.B. `3.141`) und können auch in wissenschaftlicher Notation (z.B. `4.7e-3`) angegeben werden.

Typ `Char`

Der Typ `Char` repräsentiert einzelne Zeichen. Die einzelnen Zeichen werden in Hochkommata gesetzt, also z.B. `'a'` oder `'!'`. Die verfügbaren Escape-Codes (mit angegebenem ASCII-Code) sind in Tabelle 3.4 aufgelistet. Einzelne Zeichen können auch durch ihre ASCII-Werte in Oktal-, Dezimal- oder Hexadezimalzahlen angegeben werden. So kann anstatt dem Zeichen `'W'` mit ASCII-Wert 87 auch `'\o127'`, `'\87'` oder `'\x57'` verwendet werden.

\a	7	\b	8	\f	12	\n	10
\r	13	\t	9	\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				
\^@	0	\^A	1	\^B	2	\^C	3
\^D	4	\^E	5	\^F	6	\^G	7
\^H	8	\^I	9	\^J	10	\^K	11
\^L	12	\^M	13	\^N	14	\^O	15
\^P	16	\^Q	17	\^R	18	\^S	19
\^T	20	\^U	21	\^V	22	\^W	23
\^X	24	\^Y	25	\^Z	26	\^[	27
\^\	28	\^]	29	\^^	30	\^_	31

Tabelle 3.4: Escape-Codes

Listen

Ist t ein Typ, so ist $[t]$ ebenfalls ein Typ und repräsentiert endliche Listen, deren Elemente alle vom Typ t sind. Eine Liste kann durch Aufzählung der Elemente erzeugt werden. So ist z.B. `[1,2,3]` eine Liste mit drei Elementen vom Typ `Int`. Die leere Liste wird als `[]` geschrieben. Der Operator `:` erzeugt eine Liste aus einem Element und einer weiteren Liste. So ergibt der Ausdruck `17:[0,33]` die Liste `[17,0,33]`. Zudem gibt es zwei weitere Möglichkeiten, Listen zu erzeugen (Sequenzen und List Comprehensions).

- *Arithmetische Sequenzen*
  - Sind n und m vom Typ `Int`, so ist $[n..m]$ die Liste mit Elementen von n bis einschließlich m, falls $m \geq n$. Ist $m < n$, so ergibt dies die leere Liste. Zum Beispiel ist `[-4..3]` äquivalent zu `[-4,-3,-2,-1,0,1,2,3]` und `[-5..-6]` äquivalent zu `[]`.

- Seien n, n' und m vom Typ `Int`. Für $n' > n$ und $m \geq n$ ist $[n, n'..m]$ die Liste mit Elementen von n bis höchstens m mit Schrittweite $n' - n$. Für $n' < n$ und $m \leq n$ ist $[n, n'..m]$ die Liste mit Elementen von n bis mindestens m mit Schrittweite $n' - n$. Ist im ersten Fall $m < n$ oder im zweiten Fall $m > n$, so ist $[n, n'..m] = []$. Für $n' = n$ ist $[n, n'..m] = \perp$, da keine unendlichen Listen erlaubt sind und die Schrittweite 0 dies zur Folge hätte.
  Beispiele:

  ```
  [1,3..12]    =  [1,3,5,7,9,11]
  [2,100..1]   =  []
  [1,-2..-6]   =  [1,-2,-5]
  [1,1..5]     =  ⊥
  ```

- *List Comprehensions*
  List Comprehensions sind von der Form `[expr |` *qualifiers* `]`, wobei *qualifiers* eine durch Kommata getrennte Liste von drei möglichen Ausdrucksformen (Generator, Filter und lokale Definition) ist:

  - *Generatoren*
    Ein Generator hat die Form *pat* `<-` *exp*, wobei *exp* eine Liste bezeichnet. Die Liste `[`*e* `|` *pat* `<-` *exp*`]` entspricht der Teilliste von *exp*, deren Elemente auf *pat* matchen. So ist zum Beispiel

    ```
    [y | (3,y) <- [(1,0),(3,4),(7,3)] ] = [4]
    ```

    Formal ist `[`*e* `|` *pat* `<-` *exp*`]` definiert als `loop` *exp*, wobei `loop` definiert ist als:

    ```
    loop []          =  []
    loop (pat:xs)    =  e :  loop xs
    loop (_:xs)      =  loop xs
    ```

  - *Filter*
    Ein Filter ist ein Ausdruck vom Typ `Bool`. Formal entspricht `[`*e* `|` *exp*`]` dem Ausdruck

    ```
    if exp then [e] else []
    ```

  - *Lokale Definitionen*
    Lokale Definitionen sind von der Form *pat* `=` *exp*. Die Liste `[`*e* `|` *pat* `=` *exp*`]` ist formal definiert durch

    ```
    [let pat = exp in e]
    ```

Formal ist eine Liste mit mehreren Formen definiert durch:

```
[ e | q1, ..., qn-1, qn ] =
      concat (... (concat [...[[ e | qn ] | qn-1 ] ... | q1 ])...)
```

Die Funktion `concat` liefert die Konkatenation zweier Listen.

Typ `String`

Der Typ `String` ist ein Typsynonym für `[Char]` und repräsentiert Zeichenketten. String-Literale sind in Anführungszeichen eingeschlossene Zeichenketten, wobei die Escape-Codes (siehe Typ `Char`) verwendet werden können, z.B: `"hello, world\n"`. Innerhalb eines String-Literals kann eine Folge von Leer-, Tab- und New-Line-Zeichen zwischen zwei Backspace-Zeichen eingeschlossen werden, wobei diese Zeichen einschließlich der Backspace-Zeichen ignoriert werden. So ist zum Beispiel `"hello\␣␣␣␣␣\, world"` äquivalent zu `"hello, world"`. Da auch New-Line-Zeichen erlaubt sind, kann sich die Lücke auch über mehrere Zeilen erstrecken.
Da Zeichenketten letztendlich Listen sind, können alle Listen-Funktionen auch für Zeichenketten angewendet werden.

Tupel

Sind $t_1, t_2, ..., t_n$, $n \geq 2$ Typen, dann repräsentiert der Typ $(t_1, t_2, ..., t_n)$ alle n-Tupel der Form $(x_1, x_2, ..., x_n)$, wobei die x_i vom Typ t_i sind. Im Gegensatz zu Listen können die Elemente eines Tupel unterschiedliche Typen haben, wohingegen Listen in der Länge variieren können. Zum Beispiel ist der Ausdruck `(1,'x',[4.7,0.7])` vom Typ `(Int,Char,[Float])`.

3.1.3 Wertvereinbarungen

Mit einer Wertvereinbarung wird einem Namen ein Wert zugewiesen, also zum Beispiel eine Zahl oder eine Funktion. Der Name steht für den zugewiesenen Wert und kann zu keiner Zeit geändert werden. Die so eingeführten Bezeichner dürfen nicht mit Variablen einer imperativen Programmiersprache verwechselt werden, in denen der Wert einer Variablen, wie der Name bereits andeutet, veränderbar ist. In diesem Abschnitt werden die vielfältigen Formen einer Wertvereinbarung beschrieben.

Einfaches Pattern-Matching

Funktionen können mit Hilfe von einfachem Pattern-Matching definiert werden. Die Funktion besteht hierbei aus einer Liste von Alternativen der Form

$f\ pat_1\ pat_2\ ...\ pat_n$ = rhs;

Die Anzahl der Argumente heißt Stelligkeit der Funktion. Die Alternativen einer Funktion müssen alle die gleiche Stelligkeit haben.
Beispiele:

```
fact n      =  product [1..n];

not True    =  False;
not False   =  True;
```

Bei der Anwendung einer Funktion werden die Alternativen in der Reihenfolge der Aufschreibung geprüft, so daß immer die erste passende Alternative angewendet wird. Ist beispielsweise die Funktion `pred` definiert durch

```
pred x   =  x - 1;
pred 0   =  0;
```

so ergibt `pred 0` den Wert -1, da beide Alternativen passen und die erste angewendet wird. Die wichtigsten Pattern sind in der folgenden Tabelle zusammengestellt.

Name	Beispiel	Bedeutung
Wildcard	`_`	Jedes Argument, wobei sein Wert nicht angesprochen werden kann.
Tupel	`(x,y)`	n-Tupel, im Beispiel 2- und 3-Tupel, wobei die Komponenten durch die Variablen angesprochen werden können.
Listen	`[]`, `[x]`, `[x,y]`, `(x:xs)`	Listen, wobei die Elemente bzw. Head und Tail der Liste durch die Variablen angesprochen werden können.
As-Pattern	`p@(x,y)`	Der Wert des gesamten Patterns kann durch die Variable `p` angesprochen werden.
(n+k) Pattern	`(m+1)`	Jede Zahl größer oder gleich k passt, wobei n (im Beispiel m) der „Wert - k“ ist.

Fallunterscheidung

Auf der rechten Seite der Funktionsdefinition kann zusätzlich eine Fallunterscheidung verwendet werden. Die rechte Seite entspricht dann einer Menge von bewachten Ausdrücken. Die allgemeine Form ist:

$$
\begin{array}{lll}
f\ pat_1\ pat_2\ \ldots\ pat_n & \mid condition_1 & = e_1 \\
 & \mid condition_2 & = e_2 \\
 & \ldots & \\
 & \mid condition_m & = e_m
\end{array}
$$

Beispiel:

```
oddity n  | even n     =  "even"
          | otherwise  =  "odd";
```

Die Variable `otherwise` ist (zum Beispiel durch eine Prelude-Datei) mit `True` belegt.

Lokale Definitionen

Für jede Wertvereinbarung können lokale Definitionen angegeben werden, wobei die dort eingeführten Variablen nur im Ausdruck rhs bzw. nur in den Ausdrücken $condition_i$ und e_i sichtbar sind. Die allgemeine Form ist:

$$f\ pat_1\ pat_2\ \ldots\ pat_n = rhs\ \texttt{where}\ decls$$

bzw.

$$\begin{array}{lll} f\ pat_1\ pat_2\ \ldots\ pat_n & \mid condition_1 & = e_1 \\ & \mid condition_2 & = e_2 \\ & \ldots & \\ & \mid condition_m & = e_m \\ & \texttt{where}\ decls & \end{array}$$

Die folgende Funktion `numberOfRoots` erhält als Parameter die Koeffizienten a, b und c der quadratischen Gleichung $a \cdot x^2 + b \cdot x + c = 0$ und berechnet die Anzahl der Nullstellen dieser Gleichung. Die zur Entscheidung notwendige Diskriminante wird lokal berechnet und steht deshalb in einer lokalen Definition.

```
numberOfRoots a b c  | discr < 0  = 0
                     | discr == 0 = 1
                     | discr > 0  = 2
                     where discr = b*b - 4*a*c;
```

Polymorphie

Polymorphie und das Typsystem im allgemeinen wird in Kapitel 7 behandelt, so daß hier nur kurz auf diese Begriffe eingegangen wird.
Kommen in einem Typausdruck Typvariablen vor, so ist der Typ polymorph. Zum Beispiel stellt `[a]` einen Listentypen dar, dessen Elemente nicht näher spezifiziert sind. Die Funktion `length` berechnet die Länge einer Liste, und zwar unabhängig davon, welchen Typ die Elemente tatsächlich besitzen. Der Typ der Funktion `length` ist `[a] -> Int`. So ergibt `length [1..10]` den Wert `10` und `length "Hello world"` den Wert `11`.
Die Funktion `map` hat den (Funktions-)Typ `(a -> b) -> [a] -> [b]` und ist ebenfalls polymorph. Somit kann die Funktion `map` folgendermaßen verwendet werden:

```
map (*2) [1..10]               =   [2,4,6,8,10,12,14,16,18,20]
map length ["Hello","world"]   =   [5,5]
```

Funktionen höherer Ordnung

Funktionen, die andere Funktionen als Argumente erwarten oder Funktionen als Ergebnis zurückgeben, werden Funktionen höherer Ordnung oder funktionale Formen genannt. Zum Beispiel ist die Funktionskomposition eine Funktion höherer Ordnung, da sie zwei Funktionen als Argumente erwartet und eine weitere Funktion als Ergebnis zurückgibt:

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c);
(f.g) x = f (g x);
```

Mit `sum :: [Int] -> Int` und `sqrt :: Int -> Int` ergibt sich für die Funktion `sqrt.sum` der Typ `[Int] -> Int`.
Wird für die Funktion `map :: (a -> b) -> [a] -> [b]` nur das erste Argument angegeben (currying), so ergibt dies wiederum eine Funktion. Durch den Typ des Arguments wird der Typ der resultierenden Funktion festgelegt:

`map (*2)`	ergibt eine Funktion vom Typ `[Int] -> [Int]`.
	`a` und `b` wurden durch `Int` ersetzt, da `(*2) :: Int -> Int`.
`map length`	ergibt eine Funktion vom Typ `[[a]] -> [Int]`.
	`a` wurde durch `[a]` und `b` durch `Int` ersetzt, da `length :: [a] -> Int`.

Variablendeklaration

Werden in einer Wertvereinbarung keine Parameter angegeben, so liegt eine Variablendeklaration vor. Die allgemeine Form ist:

var = *rhs*

Der zugewiesene Wert kann ein beliebiger Ausdruck sein, also zum Beispiel eine Zahl, ein String oder eine Funktion.
Beispiele:

```
fact5   =  prod [1..5];
message =  "Hello, world";
odd     =  not.even;
```

Pattern-Deklaration

Eine weitere Variante der Wertvereinbarung ist die Pattern-Deklaration. Sie ist im Prinzip mit der Variablendeklaration vergleichbar, nur daß in einer Pattern-Deklaration auch mehrere Variablen deklariert werden können. Der Wert der neu eingeführten Variablen wird ermittelt, indem das Pattern auf der linken Seite mit dem Wert auf der rechten Seite verglichen wird. Die Pattern-Deklaration hat die allgemeine Form:

pat = *rhs*

Beispiel:

```
(x:xs) = "Hello";
```

wobei sich für die Variablen `x` und `xs` die Werte `'H'` und `"ello"` ergeben.
Im Zusammenhang mit Pattern-Deklarationen treten insbesondere die folgenden zwei Probleme auf:

1. Eine Variable kann auch als Pattern angesehen werden. Steht auf der linken Seite eine Variable, so wird dies allerdings als Variablendeklaration aufgefaßt.

2. Ein (n+k)-Pattern wird als Funktionsdefinition des Operators `(+)` interpretiert.

Typdeklaration

Obwohl die explizite Angabe von Typen nicht notwendig ist, kann für jede Variable und Funktion, die durch eine Wertvereinbarung eingeführt worden ist, die Signatur angegeben werden. Die allgemeine Form ist:

var_1 , var_2 , ... , var_n :: *type*

Explizite Typangaben werden beispielsweise zum Zwecke der Dokumentation oder zur Einschränkung des sonst zu allgemeinen Typs verwendet.
Zur Notation von polymorphen Typen sind Typvariablen notwendig. Hierfür werden im Prinzip „varid"-Bezeichner verwendet, wobei ein oder zwei Apostrophe am Anfang der Typvariable wie zum Beispiel in `'a` oder `''b` die Zugehörigkeit zu einer der anonymen Typklassen von „Frisco F" auszudrücken können. In einem solchen Fall darf ein Apostroph allerdings nicht wie für gewöhnliche „varid"-Bezeichner innerhalb der Typvariable verwendet werden, wie beispielsweise in `'t1'` oder `'a'b`. Für eine ausführlichere Beschreibung sei auf das Kapitel 7 verwiesen.

3.1.4 Ausdrücke

Neben den Literalen, Listen und Tupeln gibt es noch weitere Möglichkeiten, Ausdrücke zu bilden. Hierzu gehören `case`-, `if`-, Lambda- und `let`-Ausdrücke, sowie Operator-Sections und explizit getypte Ausdrücke. Diese Formen werden nun im folgenden beschrieben.

Case-Ausdrücke

Der `case`-Ausdruck besteht aus einem Ausdruck und einer Menge von Alternativen, die in Abhängigkeit vom Ausdruck zur Anwendung kommen. Seine allgemeine Form ist:

`case` *exp* `of` {*alts*}

Die Alternativen werden durch Kommas getrennt und sind entweder einfach

pat `->` *rhs*

oder verwenden eine Fallunterscheidung:

$$\begin{array}{ll} pat & \mid condition_1 \ \texttt{->}\ rhs_1 \\ & \mid condition_2 \ \texttt{->}\ rhs_2 \\ & \ldots \\ & \mid condition_n \ \texttt{->}\ rhs_n \end{array}$$

Die Alternativen werden wie bei Funktionen in der Reihenfolge der Aufschreibung geprüft. Falls keine der angegebenen Alternativen zutrifft, so wird (wie auch bei Funktionen) eine Fehlermeldung ausgegeben.

If-Ausdrücke

Ein `if`-Ausdruck ist von der Form

`if` *e* `then` *t* `else` *f*

und wertet je nach Bedingung e den `then`-Zweig t oder den `else`-Zweig f aus.
Jeder `if`-Ausdruck kann mit einem `case`-Ausdruck realisiert werden:

```
case e of
      True  -> t
      False -> f
```

Lambda-Ausdrücke

Lamba-Ausdrücke sind Funktionen, die jedoch im Gegensatz zu Funktionsdefinitionen keinen Namen haben. Sie werden wie gewöhnliche Werte ausschließlich in Ausdrücken verwendet. Die allgemeine Form eines Lambda-Ausdrucks ist:

`\` *apat...apat* `->` *exp*

Die Atomic-Pattern stellen die Argumente dar, und das Ergebnis wird durch den Ausdruck auf der rechten Seite berechnet. Zum Beispiel stellt der Lambda-Ausdruck `\x -> x*x` eine Funktion dar, die eine Zahl als Argument nimmt und als Ergebnis dessen Quadrat liefert. Ein anderes Beispiel ist der Lambda-Ausdruck `\x y -> x + y`, welcher äquivalent zum Operator `+` ist.

Let-Ausdrücke

Ein `let`-Ausdruck wird verwendet, falls für einen Ausdruck lokale Definitionen benötigt werden. Die allgemeine Form ist:

`let` *decls* `in` *exp*

Die eingeführten Definitionen *decls* sind nur im Ausdruck *exp* sichtbar. Der `let`-Ausdruck ist mit dem `where`-Teil einer Funktionsdefinition vergleichbar.

Operator Sections

Funktionen mit mehreren Parametern können auch mit einer geringeren Anzahl von Argumenten angewandt werden (currying). Das Ergebnis ist wiederum eine Funktion mit den verbleibenden Parametern. Dies gilt auch für Infix-Operatoren. Für diese partielle Parametrisierung wird durch die Infix-Schreibweise allerdings eine spezielle Syntax verwendet, die sogenannten Operator-Sections.
Ist $\oplus$ ein Infix-Operator und sind `x` und `y` Atomic-Ausdrücke, so sind die Operator-Sections wie folgt definiert:

$$\begin{array}{lcl} (\mathtt{x}\ \oplus)\ \mathtt{y} & \equiv & \mathtt{x} \oplus \mathtt{y} \\ (\oplus\ \mathtt{x})\ \mathtt{y} & \equiv & \mathtt{y} \oplus \mathtt{x} \end{array}$$

bzw. unter Verwendung von Lambda-Ausdrücken:

$$\begin{array}{lcl} (\mathtt{x}\ \oplus) & \equiv & \backslash\mathtt{y}\ \texttt{->}\ \mathtt{x} \oplus \mathtt{y} \\ (\oplus\ \mathtt{x}) & \equiv & \backslash\mathtt{y}\ \texttt{->}\ \mathtt{y} \oplus \mathtt{x} \end{array}$$

Beispiele:

`(1 +)`	ist die Funktion, die zum Argument die Zahl `1` addiert: `(1 +) 5 = 6`
`(1.0 /)`	ist die Funktion, die das Reziproke des Arguments berechnet: `(1.0 /) 8.0 = 0.125`
`(:[])`	ist die Funktion, die eine Liste mit dem Argument als Element zurückgibt: `(:[]) "X"= ["X"]`

Die zweite Variante kann auch mit der Funktion `flip`

```
flip f x y = f y x
```

definiert werden:

$(\oplus$ `x`$) \equiv$ `flip` $\oplus$ `x`

Somit ergibt sich z.B.:

```
(/ 2.0) 5.0 = flip (/) 2.0 5.0 = (/) 5.0 2.0 = 5.0 / 2.0 = 2.5
```

Explizit getypte Ausdrücke

Ausdrücke können explizit getypt werden, um z.B. den zu allgemeinen Typ, der vom System hergeleitet wird, einzuschränken. Zum Beispiel hat der Lambda-Ausdruck

```
\x -> [x]
```

den Typ `a -> [a]`. Sollen hingegen nur String-Argumente zugelassen werden, so muß der Ausdruck getypt werden:

```
(\x -> [x]) :: String -> [String]
```

Der deklarierte Typ muß allerdings mit dem vom System hergeleiteten Typ kompatibel sein. Dies bedeutet, daß die Typen entweder übereinstimmen oder der deklarierte Typ eine Instanz des hergeleiteten Typs ist. Im obigen Fall ist `String -> [String]` eine Instanz von `a -> [a]`. Hingegen ist die Typisierung

```
(\x -> [x]) :: String -> a
```

nicht zulässig, da `String -> a` keine Instanz von `a -> [a]` ist.

3.1.5 Datentypen und Typsynonyme

Benutzerdefinierte Datentypen

Zusätzlich zu den fest vorgegebenen Datentypen können in „Frisco F" neue Datentypen definiert werden. Die allgemeine Form ist:

`data` $Datatype\ a_1 ... a_n$ `=` $constr_1$ `|` ... `|` $constr_n$

Dabei ist $Datatype$ der Name des neuen Datentyps der Stelligkeit $n \geq 0$ mit den paarweise verschiedenen Typvariablen $a_1, ...a_n$, den Parametern des Datentyps. Der Name des neuen Datentyps darf zuvor noch nicht als Typkonstruktor oder Typsynonym benutzt worden sein und ist ein „conid"-Bezeichner. Die Konstruktorfunktionen $constr_1, ..., constr_n$ beschreiben, wie neue Elemente dieses Datentyps erzeugt werden können. Eine Konstruktorfunktion kann dabei eine der folgenden Formen annehmen:

- $Name\ type_1 ... type_m$. Der Name der Konstruktorfunktion $Name$ ist ein „conid"-Bezeichner und darf zuvor noch nicht als Konstruktorfunktion benutzt worden sein. Der Type der neuen Konstruktorfunktion $Name$ ist

$$type_1 \rightarrow\ ... \rightarrow\ type_m \rightarrow\ Datatype\ a_1 ... a_n$$

- $type_1 \oplus type_2$. Der Name der Konstruktorfunktion $\oplus$ ist ein „conop"-Operator und darf zuvor noch nicht als Konstruktorfunktion benutzt worden sein. Der Type der neuen Konstruktorfunktion $Name$ ist

$$type_1 \rightarrow\ type_2 \rightarrow\ Datatype\ a_1 ... a_n$$

- In letzter Position können zwei Punkte `..` anstatt einer Konstruktorfunktion verwendet werden, um den Datentyp als erweiterbar zu kennzeichnen. Somit können Konstruktorfunktionen durch eine weitere Definition eingeführt werden. Es ist auch möglich, nur zwei Punkte und sonst keine weiteren Konstruktorfunktionen anzugeben. In diesem Fall wird ebenfalls ein erweiterbarer Datentyp eingeführt, allerdings ohne an dieser Stelle Konstruktorfunktionen zu definieren.

- In erster Position können zwei Punkte `..` anstatt einer Konstruktorfunktion verwendet werden, um einen bereits als erweiterbar gekennzeichneten Datentyp mit weiteren Konstruktorfunktionen zu erweitern. In einer Erweiterung müssen die selben Typvariablen angegeben werden, wie die des zu erweiternden Datentyps.

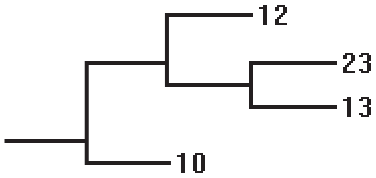
Werden in einer Konstruktorfunktion Typvariablen verwendet, so müssen diese in der Datentypdefinition vorkommen.
Durch das nachfolgende Beispiel wird ein Datentyp für Binärbäume mit Werten eines bestimmten Typs in den Blättern eingeführt:

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

Der Ausdruck

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

hat den Typ `Tree Int` und repräsentiert den Binärbaum



Eine Funktion zur Berechnung der Höhe eines solchen Binärbaumes könnte folgendermaßen definiert werden:

```
height (Lf _)              =  0;
height (left :^:  right)   =  max (height left) (height right);
```

Zum Abschluß noch ein Beispiel für einen erweiterbaren Datentyp und eine Erweiterung. An diesem Beispiel sieht man auch, daß der Name eines Typkonstruktors auch als Konstruktorfunktion verwendet werden kann.

```
data Error a = NoError | Error a | ..;
...
data Error a = ..  | ScanError (Error a) | ParseError (Error a) | ..;
```

Typsynonyme

Typsynonyme werden dafür verwendet, bequeme Abkürzungen oder aussagekräftigere Namen für Typausdrücke verwenden zu können. Die allgemeine Form einer Typsynonym-Definition ist:

`type` $Synonym\ a_1 ... a_n$ `=` $type$

Dabei ist *Synonym* der Name des neuen Typsynonyms der Stelligkeit $n \geq 0$ mit den paarweise verschiedenen Typvariablen $a_1, ... a_n$, den Parametern des Typsynonyms. Der Name des neuen Typsynonyms darf zuvor noch nicht als Typkonstruktor oder Typsynonym benutzt worden sein und ist ein „conid“-Bezeichner. Der Typ auf der rechten Seite ist die Expansion des Typsynonyms. Werden in der Expansion Typvariablen verwendet, so müssen diese in der Typsynonym-Definition vorkommen.
Der Typausdruck

$Synonym\ type_1 ... type_n$

wird als Abkürzung für diejenige Expansion betrachtet, in der die Typvariablen a_i durch die Typen $type_i$ ersetzt wurden. Die Abkürzung und die Expansion sind gleichwertig, da durch die Definition eines Typsynonyms kein neuer Datentyp eingeführt wird.
Das prominenteste Beispiel für Typsynonyme ist `String`:

```
type String = [Char];
```

Zu beachten ist, daß das zu definierende Typsynonym nicht auf der rechten Seite der eigenen Definition verwendet werden darf. Somit ist das folgende Beispiel nicht zulässig:

```
type BadSynonym = [BadSynonym];
```

Dies würde bei dem Versuch der Expansion in eine nicht terminierende Rekursion münden:

```
BadSynonym ⇒ [BadSynonym] ⇒ [[BadSynonym]] ⇒ [[[BadSynonym]]] ...
```

3.1.6 Axiome

Die Formulierung von Axiomen ist in „Frisco F“ mit der `ax`-Definition möglich. Hierzu werden die Axiome in geschweifte Klammern eingeschlossen und durch Semikola voneinander getrennt. Zudem ist die Angabe mehrerer $\forall$- oder $\forall^\perp$-Quantoren möglich. Die Axiome selbst sind prädikatenlogische Ausdrücke, die optional mit einem Namen versehen werden können. Der Axiomname, welcher links vom Ausdruck steht und von einem Punkt gefolgt wird, ist ein „varid“- oder „conid“-Bezeichner.
Mit der `op`-Definition können zusätzlich Operatoren und Funktionen mit ihren Signaturen definiert werden, ohne eine Top-Level-Definition für diese angeben zu müssen. Diese Spezifikationsoperatoren und -funktionen können in prädikatenlogischen Ausdrücken verwendet werden.
In den prädikatenlogischen Ausdrücken können Quantoren und Junktoren verwendet werden. Neben den üblichen Quantoren $\forall$ und $\exists$ gibt es noch die Varianten $\forall^\perp$ und $\exists^\perp$, bei denen die eingeführten Variablen zusätzlich den Wert $\perp$ annehmen können. Schließlich erlauben es die Quantoren $\forall^P$ und $\exists^P$, über konkrete Pattern Aussagen treffen zu können. Zum Beispiel besagt der Ausdruck

$\exists^P$ `(x:xs) :: [a] .` e,

daß es ein Pattern `x:xs` vom Typ `[a]` gibt,so daß die Aussage e gilt. Zu den Junktoren zählen die Oder- ($\vee$) und Und-Verknüpfung ($\wedge$), die Implikation ($\Rightarrow$), die Äquivalenz ($\Leftrightarrow$), die Gleichheit ($=$), die Negation ($\neg$) und der Operator zum Test auf Definiertheit (δ).
Für prädikatenlogische Ausdrücke wird der eigene Datentyp $\mathcal{B}$ verwendet, für den die Konstruktorfunktionen `TT` und `FF` definiert sind.
Das folgende Beispiel zeigt die Formulierung einiger Axiome über den Datentyp `Stack`:

```
data Stack = ...;
...
op empty   :: Stack 'a;
op push    :: Stack 'a -> 'a -> Stack 'a;
op pop     :: Stack 'a -> Stack 'a;
op top     :: Stack 'a -> 'a;
op size    :: Stack 'a -> Int;
op isEmpty :: Stack 'a -> Bool;
ax ALL s :: Stack 'a, x :: 'a . {
   Size. size s >= 0;
   Leer. empty == s <=> isEmpty s;
   LIFO. pop (push s x) == s;
   Kon.  NOT (isEmpty s) => push (pop s) (top s) == s;
   Oben. top (push s x) == x;
   Anz0. size (empty) == 0;
   Anz.  size (push s x) == size s + 1;
}
```

3.1.7 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 liegen 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.

3.2 Vergleich mit Gofer

In diesem Abschnitt werden die Erweiterungen und Änderungen von „Frisco F“ gegenüber der Sprache Gofer erläutert.

- *Axiome und Spezifikations-Operatoren*
  „Frisco F“ wurde um die zwei Top-Level-Definitionen `ax` und `op` ergänzt. Mit der `ax`-Definition können Axiome formuliert werden. Die hierfür verwendeten Ausdrücke wurden um Elemente der Prädikatenlogik ergänzt. So sind beispielsweise Quantoren und Junktoren verfügbar. In einer `op`-Definition ist es möglich, Operatoren mit ihren Signaturen für die Spezifikation einzuführen. Diese sogenannten Spez-Operatoren können nur in den Axiomen verwendet werden.

- *Typklassen und ad-hoc-Polymorphie*
  „Frisco F“ wurde mit zwei anonymen Typklassen ausgestattet, die eine einfache Form von ad-hoc-Polymorphie erlauben. Diese Typklassen wurden eingeführt, um den sonst auftretenden Unbequemlichkeiten im Zusammenhang mit arithmetischen Operationen oder der Gleichheitsfunktion aus dem Wege gehen zu können. Siehe hierzu auch Kapitel 7 bzw. Anhang C.

- *Strikte Semantik*
  Für „Frisco F“ wurde strikte Semantik gewählt, so daß im Gegensatz zu Gofer nicht *lazy evaluation*, sondern *eager evaluation* verwendet wird. Eine Konsequenz davon ist, daß „Frisco F“ über keine „unendlichen Datenstrukturen“ und insbesondere Listen wie zum Beispiel `[1,2..]` verfügt.

- *Fehlende Layout-Rule und liberalere Grammatik*
  Die Layout-Rule von Gofer erlaubt, die Struktur eines Programms durch Einrückungen auszudrücken. Auf diese Weise werden automatisch geschweifte Klammern und Semikola nach vorgegebenen Regeln eingefügt, die ohne Verwendung von Einrückungen explizit im Quellcode angegeben werden müssen. Siehe hierzu Kapitel 13 in [Jon94b].
  Diese Layout-Rule wurde in „Frisco F“ nicht übernommen. Stattdessen wurde die Grammatik so modifiziert, so daß die Handhabung von diesen Strukturierungszeichen liberaler ausgelegt wird.
  Top-Level-Definitionen und lokale Definitionen (verwendet für `where`-Teil und `let`-Ausdruck) werden in Gofer durch Semikola getrennt. In „Frisco F“ können zwischen zwei Definitionen und nach der letzten Definition auch mehrere Semikola stehen.

Lokale Definitionen werden in geschweifte Klammern eingeschlossen. Wird allerdings nur eine einzige Definition angegeben, so können diese auch fehlen, wobei dann dieser einzelnen Definition keine Semikola folgen dürfen. Die gleichen Regeln gelten auch für die Top-Level-Definitionen der Axiome und Spez-Operatoren.
Eine Ausnahme ist der `case`-Ausdruck, da auch eine einzige Alternative immer in geschweiften Klammern eingeschlossen werden muß.
Die geschweiften Klammern um ein Quellcode-Modul sind aus der Grammatik von „Frisco F" entfernt worden.

- *Primitiven*
  Die Primitiven von „Frisco F" sind fest vorgegeben und können nicht erweitert werden. Die Liste der Primitiven ist im Anhang B enthalten.

Kapitel 4

Grammatik

In diesem Kapitel wird die kontextfreie Grammatik der Sprache „Frisco F" (Version 1.0) vorgestellt. Im Abschnitt 4.3 wird auf Besonderheiten der Grammatik (wie zum Beispiel das sogenannte „Dangling Else") eingegangen.

4.1 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 (vgl. [Jon94b]). Die folgende informelle Beschreibung verdeutlicht die angewandte Form.

Nicht-Terminale

Nicht-Terminale werden wie gewöhnliche Bezeichner verwendet, also zum Beispiel topLevel oder type.

Terminale

Terminale werden in Rahmen gesetzt, also zum Beispiel $\boxed{\text{data}}$ oder $\boxed{,}$.
Des weiteren werden für spezielle Terminale die folgenden Bezeichner verwendet:

- VARID, VAROP, CONID und CONOP - zu Bezeichnern und Operatoren siehe Abschnitt 3.1.1
- TYVAR - zu Typvariablen siehe Abschnitt 3.1.3 (Typdeklarationen)
- INTEGER, FLOAT, CHAR und STRING für Integer-, Float-, Character- und String-Literale

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.

$a \mid b$	a oder b
$a \mid b \mid c$	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
$a\ \{b \mid c\}\ d$	aber: $ab \mid cd \equiv \{ab\} \mid \{cd\}$

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. Im ersten Fall ist auch die 0-malige Anwendung erlaubt (leere Liste), wohingegen im zweiten Fall mindestens ein Element angegeben werden muß.

$\{a\}^*$	ϵ oder a oder aa oder aaa ...
$\{a\}^+$	$\equiv a\ \{a\}^*$, d.h. a oder aa oder aaa ...

Wiederholung mit Separator
Die Wiederholung mit Separator gestattet zusätzlich die Angabe eines Trennzeichens, das zwischen den Elementen angegeben werden muß.

$\{a \parallel s\}^*$	ϵ oder a oder asa oder $asasa$ ...
$\{a \parallel s\}^+$	$\equiv a\ \{sa\}^*$, d.h. a oder asa oder $asasa$ ...

4.2 Grammatik von „Frisco F“

Top-Level

topLevels	::=	$\{\ [\ \text{topLevel}\ ]\ \ \|\ \boxed{;}\ \}^*$
topLevel	::=	$\boxed{\text{data}}\ \text{typeLhs}\ \boxed{=}\ \text{constrs}$
	|	$\boxed{\text{type}}\ \text{typeLhs}\ \boxed{=}\ \text{type}$
	|	$\{\ \boxed{\text{infix}} \mid \boxed{\text{infixl}} \mid \boxed{\text{infixr}}\ \}\ \ [\ \text{DIGIT}\ ]\ \ \{\ \text{op}\ \|\ \boxed{,}\ \}^+$
	|	decl
	|	$\boxed{\text{op}}\ \text{signature}$
	|	$\boxed{\text{op}}\ \boxed{\{}\ \ \{\ \ [\ \text{signature}\ ]\ \ \|\ \boxed{;}\ \}^*\ \ \boxed{\}}$
	|	$\boxed{\text{ax}}\ \ \{\ \ \text{quantor}\ \ \}^*\ \ \boxed{\{}\ \ \{\ \ [\ \text{axiom}\ ]\ \ \|\ \boxed{;}\ \}^*\ \ \boxed{\}}$
typeLhs	::=	$\text{CONID}\ \ \{\ \text{VARID}\ \}^*$
constrs	::=	$[\ \boxed{..}\ \boxed{\mid}\ ]\ \ \{\ \ \text{constr}\ \ \|\ \boxed{\mid}\ \}^+\ \ [\ \boxed{\mid}\ \boxed{..}\ ]$
	|	$\boxed{..}$
constr	::=	type CONOP type
	|	$\text{CONID}\ \ \{\ \text{type}\ \}^*$
signature	::=	$\{\ \text{var}\ \|\ \boxed{,}\ \}^+\ \ \boxed{::}\ \text{type}$
quantor	::=	$\{\ \boxed{\forall} \mid \boxed{\forall^{\perp}}\ \}\ \ \{\ \text{pat}\ \boxed{::}\ \text{type}\ \|\ \boxed{,}\ \}^+\ \ \boxed{.}$
axiom	::=	$[\ \{\ \text{VARID} \mid \text{CONID}\ \}\ \boxed{.}\ ]\ \ \text{lExp}$

Type-Expressions

type	::=	$\{\ \text{ctype} \parallel \boxed{\texttt{->}}\ \}^{+}$
ctype	::=	atype
	|	$\text{CONID}\ \ \{\ \text{atype}\ \}^{*}$
atype	::=	TYVAR
	|	$\boxed{(}\ \ \{\ \text{type} \parallel \boxed{,}\ \}^{*}\ \ \boxed{)}$
	|	$\boxed{[}\ \text{type}\ \boxed{]}$

Value-Declarations

decl	::=	$\{\ \text{var} \parallel \boxed{,}\ \}^{+}\ \ \boxed{::}\ \text{type}$
	|	fun rhs [where]
	|	pat rhs [where]
fun	::=	var
	|	pat varop pat
	|	$\boxed{(}\ \text{pat varop}\ \boxed{)}$
	|	$\boxed{(}\ \text{varop pat}\ \boxed{)}$
	|	fun apat
	|	$\boxed{(}\ \text{fun}\ \boxed{)}$
rhs	::=	$\boxed{=}\ \text{exp}$
	|	$\{\ \boxed{\vert}\ \text{exp}\ \boxed{=}\ \text{exp}\ \}^{+}$
where	::=	$\boxed{\text{where}}\ \text{decls}$
decls	::=	decl
	|	$\boxed{\{}\ \ \{\ \ [\ \text{decl}\ ]\ \ \parallel \boxed{;}\ \}^{*}\ \ \boxed{\}}$

Expressions

lExp ::= $\{ \boxed{\forall} \mid \boxed{\exists} \mid \boxed{\forall^{\perp}} \mid \boxed{\exists^{\perp}} \} \ \{ \text{pat} \boxed{::} \text{type} \parallel \boxed{,} \}^{+} \ \boxed{.} \ \text{lExp}$

$\mid$ $\{ \boxed{\forall^{P}} \mid \boxed{\exists^{P}} \} \ \{ \text{pat} \boxed{=} \text{exp} \parallel \boxed{,} \}^{+} \ \boxed{.} \ \text{lExp}$

$\mid$ $\text{lExp} \ \{ \boxed{\vee} \mid \boxed{\wedge} \mid \boxed{\Leftrightarrow} \mid \boxed{\Rightarrow} \} \ \text{lExp}$

$\mid$ $\boxed{\neg} \ \text{lExp}$

$\mid$ $\boxed{\delta} \ \text{exp}$

$\mid$ $\text{exp} \boxed{=} \text{exp}$

$\mid$ exp

$\mid$ $\boxed{\text{TT}}$

$\mid$ $\boxed{\text{FF}}$

exp ::= $\boxed{\backslash} \ \{ \text{apat} \}^{+} \ \boxed{\text{->}} \ \text{exp}$

$\mid$ $\boxed{\text{let}} \ \text{decls} \ \boxed{\text{in}} \ \text{exp}$

$\mid$ $\boxed{\text{if}} \ \text{exp} \ \boxed{\text{then}} \ \text{exp} \ \boxed{\text{else}} \ \text{exp}$

$\mid$ $\boxed{\text{case}} \ \text{exp} \ \boxed{\text{of}} \ \boxed{\{} \ \text{alts} \ \boxed{\}}$

$\mid$ $\text{opExp} \ [\boxed{::} \ \text{type}]$

opExp ::= opExp op opExp

$\mid$ $[\boxed{\text{-}}] \ \{ \text{atomic} \}^{+}$

atomic ::= var

$\mid$ conid

$\mid$ INTEGER

$\mid$ FLOAT

$\mid$ CHAR

$\mid$ STRING

$\mid$ $\boxed{(} \ \{ \text{exp} \parallel \boxed{,} \}^{*} \ \boxed{)}$

$\mid$ $\boxed{(} \ \text{exp op} \ \boxed{)}$

$\mid$ $\boxed{(} \ \text{op exp} \ \boxed{)}$

$\mid$ $\boxed{[} \ \text{list} \ \boxed{]}$

$\mid$ $\boxed{\perp}$

$\mid$ $\boxed{!} \ \text{lExp}$

list ::= { exp ∥ $\boxed{,}$ }*
$\mid$ exp $\boxed{|}$ quals
$\mid$ exp $\boxed{..}$ exp
$\mid$ exp $\boxed{,}$ exp $\boxed{..}$ exp

quals ::= { qual ∥ $\boxed{,}$ }$^+$

qual ::= pat $\boxed{\texttt{<-}}$ exp
$\mid$ pat $\boxed{=}$ exp
$\mid$ exp

alts ::= { [pat altRhs [where]] ∥ $\boxed{;}$ }*

altRhs ::= $\boxed{\texttt{->}}$ exp
$\mid$ { $\boxed{|}$ exp $\boxed{\texttt{->}}$ exp }$^+$

Patterns

pat ::= pat conop pat
$\mid$ var $\boxed{+}$ INTEGER
$\mid$ { apat }$^+$

apat ::= var
$\mid$ $\boxed{\_}$
$\mid$ conid
$\mid$ INTEGER
$\mid$ CHAR
$\mid$ STRING
$\mid$ var $\boxed{@}$ pat
$\mid$ $\boxed{(}$ { pat ∥ $\boxed{,}$ }* $\boxed{)}$
$\mid$ $\boxed{(}$ pat conop $\boxed{)}$
$\mid$ $\boxed{(}$ conop pat $\boxed{)}$
$\mid$ $\boxed{[}$ { pat ∥ $\boxed{,}$ }* $\boxed{]}$

Variables and Operators

var	::=	varid
op	::=	varop
	|	conop
varid	::=	VARID
	|	$\boxed{(}$ VAROP $\boxed{)}$
varop	::=	VAROP
	|	$\boxed{'}$ VARID $\boxed{'}$
conid	::=	CONID
	|	$\boxed{(}$ CONOP $\boxed{)}$
conop	::=	CONOP
	|	$\boxed{'}$ CONID $\boxed{'}$

4.3 Besonderheiten der Grammatik

Die Grammatik der Sprache „Frisco F" weist einige Besonderheiten auf, die in diesem Abschnitt kurz beschrieben werden. Zum Teil wird in anderen Teilen dieses Berichts ebenfalls darauf eingegangen.

- *Dangling Else*
  Der Parser der „Frisco F"-Grammatik wird mit dem Parser-Generator „Bison" erzeugt. Dabei treten sogenannte Shift-Reduce-Konflikte auf, die darauf hinweisen, daß die Grammatik nicht ganz unproblematisch ist. Ein typisches Beispiel solcher Schwachstellen ist das sogenannte „Dangling Else". Es tritt immer dann auf, wenn der `if`-Ausdruck am Ende nicht durch ein Schlüsselwort abgeschlossen wird. So sind zum Beispiel die Ausdrücke

  ```
  if a < b then a else b + 1
  let x = 4 in x * x
  ```

  mehrdeutig, denn die Ausdrücke können auf die folgenden zwei Arten interpretiert werden:

  ```
  (if a < b then a else (b + 1))      (1)
  ((if a < b then a else b) + 1)      (2)

  (let x = 4 in (x * x))              (1)
  ((let x = 4 in x) * x)              (2)
  ```

  Der Parser-Generator löst das Problem auf, indem er die Regel „Shift vor Reduce" anwendet, so daß in diesem Fall die Alternativen 1 ausgewählt werden.

- *Variablendeklaration*
  Die Grammatik ist bei Variablendeklarationen nicht eindeutig. Wird auf der linken Seite einer Wertvereinbarung nur eine Variable angegeben, so können die zwei Alternativen $pat = rhs$ und $fun = rhs$ angewendet werden. Die tatsächlich verwendete Grammatik weicht von der hier dokumentierten Grammatik ab und betrifft auch das Problem der Variablendeklaration. Für Einzelheiten sei jedoch auf das Kapitel 6 verwiesen.

- *(n+k)-Pattern*
  Die Grammatik läßt für Pattern-Deklarationen die Verwendung von (n+k)-Pattern zu. Allerdings wird ein (n+k)-Pattern auf der linken Seite einer Wertvereinbarung als Definition des Operators `+` interpretiert und ist somit gleichwertig zu `(+) n k`.

Kapitel 5

Lexikalische und syntaktische Analyse

5.1 Lexikalische Analyse

In diesem Abschnitt wird der Scanner beschrieben, welcher ausgehend von einem Zeichenstrom Symbole identifiziert und überflüssige Zeichen und Kommentare ignoriert (Abbildung 5.1).

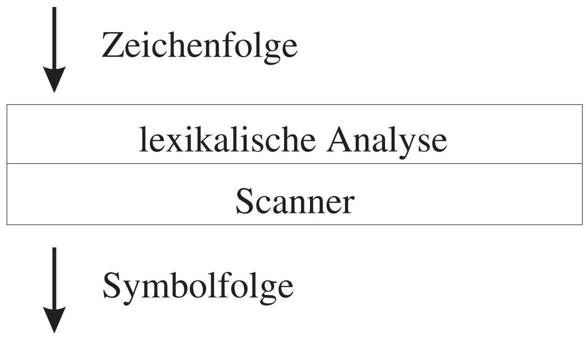


Abbildung 5.1: Der „Frisco F“-Scanner

Die Symbolfolge besteht aus den Terminal-Zeichen, die im folgenden beschrieben werden.

5.1.1 Terminale

Reservierte Wörter

Die Liste der reservierten Wörter der „Frisco F“-Grammatik wurde bereits in Tabelle 3.1 auf Seite 16 angegeben.

Spezielle Zeichen und Zeichenkombinationen

Die Liste der speziellen Zeichen und Zeichenkombinationen der „Frisco F“-Grammatik wurde bereits in Tabelle 3.3 auf Seite 16 angegeben.

Ersatzzeichen für graphische Symbole

Die Grammatik enthält graphische Symbole. Da diese in der Regel nicht darstellbar sind, werden die ASCII-Ersatzzeichen verwendet, die in Tabelle 5.1 angegeben sind. Die aktuelle Implementierung des Scanners erkennt nur die ASCII-Ersatzzeichen.

Bezeichner

Es wird zwischen Bezeichnern mit großen und kleinen Anfangsbuchstaben unterschieden. Die Bezeichner mit großen Anfangsbuchstaben werden als „conid" bezeichnet. Die Bezeichner mit kleinen Anfangsbuchstaben werden als „varid" bezeichnet.

Operatoren

Es wird zwischen „varop"- und „conop"-Operatoren unterschieden, wobei die Operatoren aus den Zeichen kombiniert werden können, die in Tabelle 3.2 auf Seite 16 aufgelistet sind. „Conop"-Operatoren müssen dabei mit einem Doppelpunkt beginnen, wohingegen alle anderen gebildeten Operatoren als „varop"-Operatoren gelten.

Graphisch	ASCII-Ersatzzeichen
$\forall$	`ALL`
$\forall^{\perp}$	`ALLB`
$\forall^{P}$	`ALLP`
$\exists$	`EX`
$\exists^{\perp}$	`EXB`
$\exists^{P}$	`EXP`
$\Rightarrow$	`=>`
$\Leftrightarrow$	`<=>`
$\wedge$	`AND`
$\vee$	`OR`
$\neg$	`NOT`
δ	`DEF`
$\perp$	`BOT`

Tabelle 5.1: ASCII-Ersatzzeichen für Symbole

Literale

Der Scanner erkennt Literale für Ganzzahlen (`Int`), Gleitpunktzahlen (`Float`), Zeichen (`Char`) und Zeichenketten (`String`). Die Literale und Escape-Codes für Zeichenliterale wurden bereits in Abschnitt 3.1.2 aufgeführt.

5.1.2 Kommentare

Es gibt zwei Arten von Kommentaren, nämlich durch `--` eingeleitete Zeilenkommentare und geschachtelte Kommentare, die von den Zeichen `{-` und `-}` eingeschlossen werden. Die Kommentare wurden bereits im Abschnitt 3.1.6 dargestellt.
Die Verarbeitung der Kommentare geschieht vollständig im Scanner. Zeilenkommentare können durch zwei einfache Regeln ignoriert werden, wohingegen geschachtelte Kommentare mit einem aufwendigeren Mechanismus behandelt werden müssen. Darauf wird in Abschnitt 8.3.1 näher eingegangen.

5.2 Syntaktische Analyse

In diesem Abschnitt wird der Parser beschrieben, welcher, ausgehend von einem Symbolstrom, einen abstrakten Syntaxbaum aufbaut (Abbildung 5.2).

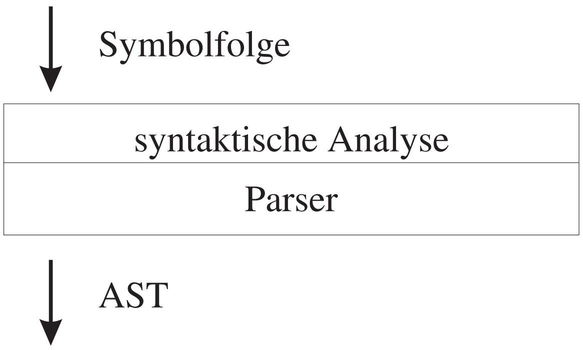


Abbildung 5.2: Der „Frisco F“-Parser

Der Parser wurde mit Bison generiert, einem LALR(1)-Parser-Generator. Die LALR-Technik wird oft in der Praxis benutzt, da sie gegenüber anderen Techniken Vorteile besitzt (vgl. [Aho88], S. 288 ff). Das L in LR(1) steht für „Abarbeiten der Eingabe von links nach rechts“ und das R steht für „Bildung einer Rechtsableitung in umgekehrter Reihenfolge“. Im Allgemeinen steht k in LR(k) für die Zahl der im voraus betrachteten Eingabesymbole, die bei Syntaxanalyse-Entscheidungen gebraucht werden. In diesem Fall wird also höchstens ein Eingabesymbol im voraus betrachtet. Das LA in LALR(1) steht für „lookahead“.

5.2.1 Grammatik

Die zu verarbeitende Grammatik muß eine LR(1)-Grammatik sein. Die zugrundeliegende Grammatik ist jedoch keine LR(1)-Grammatik, da Shift/Reduce-Konflikte auftreten. Bison löst das Problem, indem die Reihenfolge der Aufschreibung entscheidet (vgl. Abschnitt 4.3). Es sei noch erwähnt, daß ähnliche Shift/Reduce-Konflikte und sogar Reduce/Reduce-Konflikte auch in der Gofer-Grammatik auftreten (vgl. [Jon94a] Abschnitt 5.2, Gofer-Quelldatei parser.y).
Zur Vermeidung von Konflikten ist die zugrundeliegende Grammatik gegenüber der in Kapitel 4 dokumentierten Grammatik modifiziert. Die folgende Liste gibt einen Überblick über

die vorgenommenen Modifikationen.

- Patterns werden als Ausdrücke geparst, so daß in der semantischen Analyse spezielle Prüfungen notwendig sind. An jeder Stelle, an der in der (dokumentierten) Grammatik ein Pattern vorkommt, muß geprüft werden, ob der stattdessen geparste Ausdruck ein gültiges Pattern darstellt.
- Präfix-Konstruktorfunktionen werden als Typen geparst. In der semantischen Analyse wird überprüft, ob der geparste Typ eine korrekte Konstruktorfunktion ist.
- Die linke Seite einer Funktions- oder Pattern-Deklaration wird als Ausdruck (genauer opExp) geparst. In der semantischen Analyse wird entschieden, ob es sich um eine Funktions- bzw. Pattern-Deklaration oder überhaupt um eine gültige Deklaration handelt.
- Die Trennung zwischen „normalen“ Ausdrücken und prädikatenlogischen Ausdrücken wird in der zugrundeliegenden Grammatik nicht vorgenommen. Erst in der semantischen Analyse wird die Korrektheit der verwendeten Ausdrücke überprüft.

Die Behandlung dieser Modifikationen wird in Kapitel 6 ausführlich beschrieben.

5.2.2 Semantische Aktionen

Die semantischen Aktionen dienen dem Aufbau der AST-Strukturen. In den unteren Ebenen werden die entsprechenden Teilstrukturen zu größeren Strukturen zusammengesetzt. In der Top-Level-Ebene werden die erstellten Elemente in die Symboltabelle eingefügt. Für Infix-Ausdrücke ist zusätzlich ein Verarbeitungsschritt notwendig, da die Einhaltung der Präzedenz und Assoziativität der Operatoren in der Grammatik nicht verankert ist. Dieser Tidy-Infix-Algorithmus wird in Abschnitt 8.3.2 beschrieben.

Kapitel 6

Semantische Analyse

In diesem Kapitel wird die semantische Analyse beschrieben, welche, ausgehend von einem abstrakten Syntaxbaum, Kontextchecks durchführt und diesen mit semantischen Informationen anreichert. Die Analyse wird in drei Phasen ausgeführt. In der ersten Phase werden Initialisierungen vorgenommen, die bereits vor dem Parsen ausgeführt werden. Die zweite Phase betrifft Kontextprüfungen, die während dem Parsen durchgeführt werden, wohingegen die Prüfungen der dritten Phase erst nach dem Parsen ausgeführt werden. Aus diesem Grund ist die Anordnung der Übersetzer-Phasen, die in Kapitel 2 vorgestellt wurden, bezüglich der semantischen Analyse rein schematisch zu verstehen. Die Abbildung 6.1 zeigt, wie syntaktische und semantische Analyse tatsächlich miteinander verwoben sind.

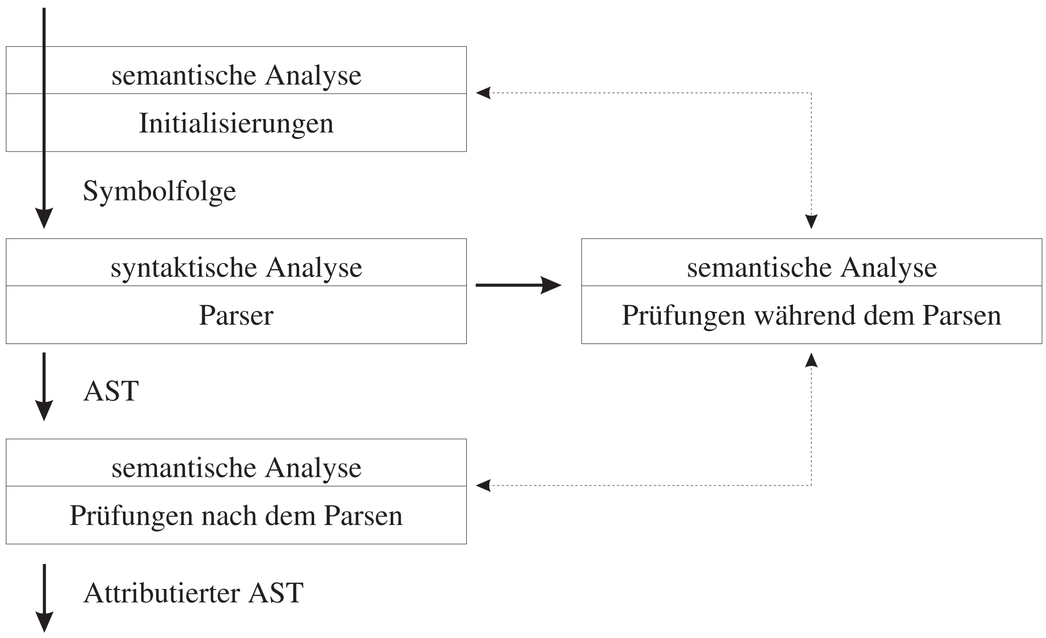


Abbildung 6.1: Semantische Analyse

6.1 Aktionen vor dem Parsen

Vor dem Parsen werden die benötigten Datenstrukturen wie Listen und Symboltabelle initialisiert. Zudem werden vordefinierte Objekte (wie zum Beispiel die Operatoren AND und OR) in die entsprechenden Strukturen eingefügt.

6.1.1 Initialisierungen

Regel Init

1. Folgende Listen werden mit einer leeren Liste initialisiert:
   - Liste für Data- und Type-Definitionen
   - Liste für Axiom-Definitionen
   - Liste für Gleichungen und Signatur-Deklarationen
   - Liste der Bindungen, siehe Eq-1Bnd bis Eq-4Bnd
   - Liste der Pattern-Variablen, siehe Pat-1Chk bis Pat-6Chk
2. Folgende „Dictionaries“ werden mit einem leeren Dictionary initialisiert:
   - Dictionary mit Infix-Operatoren
Schlüssel = Infix-Operatorname, Wert = Assoziativität und Präzedenz
   - Dictionary mit Spezifikations-Operatoren
Schlüssel = Spezifikations-Operatorname, Wert = Signatur
3. Die Symboltabelle wird als leere Symboltabelle initialisiert. Es wird ein Scope für die globale Ebene (Top-Level) erzeugt. Zur Implementierung siehe Abschnitt 8.3.3.

6.1.2 Vordefinierte Datentypen und Operatoren

Regel Init-Typen

Es werden folgende Datentypen in die Symboltabelle aufgenommen:

- Typkonstruktor `Bool` mit den Konstruktorfunktionen `True` und `False`
- Typkonstruktor $\mathcal{B}$ für prädikatenlogische Ausdrücke
- Typkonstruktor `Float`
- Typkonstruktor `Int`
- Typkonstruktor `Char`
- Typsynonym `String`
- Konstruktorfunktion `:` für Listen

Regel Init-Funktionen

Folgende Definitionen werden in die Symboltabelle aufgenommen:

- Primitiven (siehe Anhang B)
- Elementfunktionen der anonymen Typklassen (siehe Anhang C)
- Junktoren `AND`, `OR`, `=>` und `<=>` und Operator `=`

Regel Init-Infix

Für die folgenden Operatoren werden Infix-Definitionen angelegt:

```
infixl  7  *
infix   7  /
infixl  6  +, -
infix   4  ==, /=, <, <=, >, >=
infixl  0  AND, OR, =>, <=>, =
```

Für jeden Operator wird gemäß obiger Definition ein Eintrag im Dictionary für Infix-Operatoren eingetragen, siehe Infix-4P. Somit sind Assoziativität und Präzedenz dieser Operatoren in allen prädikatenlogischen Ausdrücken verfügbar.

6.2 Kontextchecks während dem Parsen

In der Top-Level-Ebene werden Data-, Type-, Infix-, Op- und Axiom-Definitionen bzw. Gleichungen (*equations*) und Signatur-Deklarationen verarbeitet. Für diese werden während dem Parsen zahlreiche Kontextbedingungen durchgeführt. Es ist aber nicht möglich, alle Bedingungen zu diesem Zeitpunkt zu prüfen. Eingeschlossen sind auch Prüfungen für Ausdrücke, insbesondere prädikatenlogische Ausdrücke und Typausdrücke.

6.2.1 Data-Definition

topLevel	::=	...
	|	$\boxed{\text{data}}$ typeLhs $\boxed{=}$ constrs
	|	...
typeLhs	::=	CONID { VARID }*
constrs	::=	[$\boxed{..}$ $\boxed{\vert}$] { constr ‖ $\boxed{\vert}$ }+ [$\boxed{\vert}$ $\boxed{..}$]
	|	$\boxed{..}$

dokumentiert:

constr	::=	type CONOP type
	|	CONOP { type }*

geparst:

constr	::=	type CONOP type
	|	type

Regel Data-1P

Alle in *typeLhs* durch $\{$ *varid* $\}^*$ eingeführten Typvariablen müssen paarweise verschieden sein.
Fehlerbehandlung:
Doppelt auftretende Typvariablen werden durch neue nicht-sichtbare Typvariablen ersetzt, so daß zumindest die Stelligkeit des Typkonstruktors erhalten bleibt. Zudem wird eine entsprechende Warnung ausgegeben.

Regel Data-2P

Es werden zwei Fälle unterschieden:

a) Enthält *constrs* nur die Pseudo-Konstruktorfunktion `..` oder beginnt *constrs* nicht mit `..`, dann darf der zu definierende Typkonstruktorname ($CONID$) in keiner vorausgegangenen Definition als Typkonstruktor oder Typsynonym verwendet worden sein. Somit werden folgende Fälle abgedeckt:

   - `..`
   - *constrs* | ... | *constrs*
   - *constrs* | ... | *constrs* | `..`

b) Beginnt *constrs* mit `..` und enthält *constrs* mindestens eine weitere Konstruktorfunktion, dann ist die Regel erfüllt, falls bereits mindestens eine Data-Definition dieses Typkonstruktors existiert und die letzte Konstruktorfunktion der zuletzt gegebenen Data-Definition dieses Typkonstruktors gleich `..` war. Zudem müssen auf der linken Seite dieselben Typvariablen stehen, wie die der zuletzt gegebenen Data-Definition dieses Typkonstruktors.

Fehlerbehandlung:
Die fehlerhafte Data-Definition wird ignoriert, und es wird eine entsprechende Warnung ausgegeben.

Regel Data-3P

Zur Vermeidung von Shift/Reduce- und Reduce/Reduce-Konflikten wird die zweite Alternative von *constr* als Typ geparst. Es muß somit sichergestellt werden, daß die Struktur des geparsten Typs einer Konstruktorfunktion entspricht. Anschließend wird der Typ in eine Konstruktorfunktion umgewandelt:

a) Ist t ein Funktionstyp, d.h. hat t die Form $type \rightarrow\ type \rightarrow\ ... \rightarrow\ type$, dann ist die Struktur ungültig.

b) Hat t die Form $atype'\ \ \{\ atype\ \}^*$ mit $atype' = VARID$, $()$, $(type)$, $(type, ..., type)$ oder $[type]$, dann ist die Struktur ungültig.

c) Hat t die Form $CONID\ \{\ atype\ \}^*$, dann ist die Struktur gültig. Für die Konstruktorfunktion ergibt sich: $CONID\ \{\ type'\ \}^*$, wobei jedes $atype$ strukturell zu einem Typ $type' = Type(CType(atype))$ transformiert wird. Die einzelnen $atype's$ werden durch weitere Regeln geprüft.

Fehlerbehandlung:
Die fehlerhafte Konstruktorfunktion wird ignoriert, und es wird eine entsprechende Warnung ausgegeben.

Regel Data-4P

Jede Konstruktorfunktion darf nicht bereits als Konstruktorfunktion in der gleichen oder in vorausgegangenen Definitionen verwendet worden sein.
Fehlerbehandlung:
Fehlerhafte Konstruktorfunktionen werden ignoriert, und es wird eine entsprechende Warnung ausgegeben.

Regel Data-5P

In jeder Konstruktorfunktion dürfen nur Typvariablen auftreten, die durch die linke Seite der Definition (*typeLhs*) eingeführt wurden (vgl. Regel Data-1P).
Fehlerbehandlung:
Die fehlerhafte Konstruktorfunktion wird ignoriert, und es wird eine entsprechende Warnung ausgegeben.

Regel Data-6P

Jede Data-Definition, die nicht durch die Regeln Data-1P bis Data-5P verworfen wurde, wird in eine für Data- und Type-Definitionen gemeinsame Liste eingefügt. Zudem werden die Namen aller Konstruktorfunktionen in die Symboltabelle für den globalen Namensraum mit den entsprechenden AST-Strukturen eingefügt.

a) Falls die Data-Definition keine Erweiterung war, wird der Typkonstruktorname in die Symboltabelle für den globalen Namensraum mit den entsprechenden AST-Strukturen eingefügt.

b) War die Data-Definition eine Erweiterung, so wird die entsprechende AST-Struktur, die bereits in der Symboltabelle eingefügt wurde, um die neuen Konstruktorfunktionen ergänzt. Die dazwischen liegenden Pseudo-Konstruktorfunktionen `..` werden entfernt.

Beispiel:

```
data D a b  =  X1 a | X2 b | ..;
data D a b  =  ..  | X3 a b | ..;
```

Liste:

[..., (`D a b = X1 a | X2 b | ..`), (`D a b = ..  | X3 a b | ..`), ...]

Symboltabelle:

```
[...,
    D = DT(D a b = X1 a | X2 b | X3 a b | ..),
    X1 = CF(X1 a),
    X2 = CF(X2 b),
    X3 = CF(X3 a b), ...
]
```

6.2.2 Type-Definition

topLevel	::=	...
	|	$\boxed{\text{type}}$ typeLhs $\boxed{=}$ type
	|	...
typeLhs	::=	CONID { VARID }*

Regel Type-1P

Alle in *typeLhs* durch { *varid* }* eingeführten Typvariablen müssen paarweise verschieden sein.
Fehlerbehandlung:
Doppelt auftretende Typvariablen werden durch neue nicht-sichtbare Typvariablen ersetzt, so daß zumindest die Stelligkeit des Typsynonyms erhalten bleibt. Zudem wird eine entsprechende Warnung ausgegeben.

Regel Type-2P

Das zu definierende Typsynonym ($CONID$) darf in keiner vorausgegangenen Definition als Konstruktorfunktion verwendet worden sein.
Fehlerbehandlung:
Die fehlerhafte Type-Definition wird ignoriert, und es wird eine entsprechende Warnung ausgegeben.

Regel Type-3P

Im Typausdruck dürfen nur Typvariablen auftreten, die durch die linke Seite der Definition (*typeLhs*) eingeführt wurden (vgl. Regel Type-1P).
Fehlerbehandlung:
Die Type-Definition wird ignoriert, und es wird eine entsprechende Warnung ausgegeben.

Regel Type-4P

Jede Type-Definition, die nicht durch die Regeln Type-1P bis Type-3P verworfen wurde, wird in eine für Data- und Type-Definitionen gemeinsame Liste eingefügt. Zudem wird der Typsynonymname in die Symboltabelle für den globalen Namensraum mit der entsprechenden AST-Struktur eingefügt.

6.2.3 Infix-Definition

dokumentiert:

topLevel	::=	...
	|	{ $\boxed{\text{infix}}$ | $\boxed{\text{infixl}}$ | $\boxed{\text{infixr}}$ } [DIGIT] { op ∥ $\boxed{,}$ }$^+$
	|	...

geparst:

topLevel	::=	...
	|	{ $\boxed{\text{infix}}$ | $\boxed{\text{infixl}}$ | $\boxed{\text{infixr}}$ } [INTEGER] { op ∥ $\boxed{,}$ }$^+$
	|	...

Regel Infix-1P

Da für $DIGIT$ ersatzweise eine Ganzzahl geparst wird, muß geprüft werden, ob diese zwischen 0 und 9 liegt. Fehlt $DIGIT$, so wird als Standard-Präzedenz 9 verwendet.
Fehlerbehandlung:
Liegt die Präzedenz nicht im Bereich 0 bis 9, so wird als Präzedenz 9 angenommen und eine Warnung ausgegeben.

Regel Infix-2P

Wurde ein Infix-Operator in einer Infix-Definition angegeben, so darf dieser in keiner weiteren Infix-Definition vorkommen, insbesondere nicht mehrmals in der gleichen Infix-Definition.
Fehlerbehandlung:
Alle weiteren Infix-Definitionen eines Infix-Operators werden ignoriert und jeweils eine Warnung ausgegeben.

Regel Infix-3P

Wurde ein Spez-Operator *op* in einer vorausgegangenen Op-Definition eingeführt, so darf dieser nicht in einer nachfolgenden Infix-Definition vorkommen (vgl. Regel Op-2P).
Fehlerbehandlung:
Alle weiteren Infix-Definitionen dieses Spez-Operators werden ignoriert und jeweils eine Warnung ausgegeben.
Bemerkung:
Zur Prüfung von Infix-3P muß im Dictionary für Infix-Operatoren nachgeschlagen werden (vgl. Regel Op-3P).

Regel Infix-4P

Jeder Infix-Operator einer Infix-Definition, der nicht durch die Regeln Infix-1P bis Infix-3P verworfen wurde, wird in einem Dictionary für Infix-Operatoren eingetragen. Der Operatorname, der wegen Regel Infix-2P eindeutig ist, wird als Schlüssel verwendet. Somit können jederzeit für einen Infix-Operator seine Präzedenz und seine Assoziativität nachgeschlagen werden.

6.2.4 Op-Definition

$$
\begin{array}{lll}
\text{topLevel} & ::= & \ldots \\
 & | & \boxed{\text{op}}\ \text{signture} \\
 & | & \boxed{\text{op}}\ \boxed{\{}\ \{\ [\ \text{signture}\ ]\ \|\ \boxed{;}\ \}^{*}\ \boxed{\}} \\
 & | & \ldots \\
 & & \\
\text{signature} & ::= & \{\ \text{var}\ \|\ \boxed{,}\ \}^{+}\ \boxed{::}\ \text{type}
\end{array}
$$

Regel Op-1P

Ein Spez-Operator darf höchstens in einer Op-Deklaration auftreten, insbesondere nicht mehrmals in ein und derselben.
Fehlerbehandlung:
Alle nachfolgenden Deklarationen eines Spez-Operators werden ignoriert, und es wird eine entsprechende Warnung ausgegeben.

Regel Op-2P

Wurde ein Infix-Operator *op* in einer vorausgegangenen Infix-Definition eingeführt, so darf dieser nicht in einer nachfolgenden Op-Definition vorkommen (vgl. Regel Infix-3P).
Fehlerbehandlung:
Alle weiteren Op-Definitionen dieses Infix-Operators werden ignoriert und jeweils eine Warnung ausgegeben.
Bemerkung:
Zur Prüfung von Op-2P muß im Dictionary für Infix-Operatoren nachgeschlagen werden (vgl. Regel Infix-4P).

Regel Op-3P

Jeder Spez-Operator einer Op-Definition, der nicht durch die Regeln Op-1P bis Op-2P verworfen wurde, wird in einem Dictionary für Spez-Operatoren eingetragen. Der Operatorname, der wegen Regel Op-1P eindeutig ist, wird als Schlüssel verwendet. Somit kann jederzeit für einen Spez-Operator seine Signatur nachgeschlagen werden.
Bemerkung:
Da die Spez-Operatoren nicht in die Symboltabelle eingefügt werden, können sie auch nicht in Ausdrücken verwendet werden.

6.2.5 Axiom-Definition

$$
\begin{array}{lll}
\text{topLevel} & ::= & \ldots \\
 & | & \boxed{\text{ax}}\ \{\ \text{quantor}\ \}^{*}\ \boxed{\{}\ \{\ [\ \text{axiom}\ ]\ \|\ \boxed{;}\ \}^{*}\ \boxed{\}} \\
 & | & \ldots \\
 & & \\
\text{quantor} & ::= & \{\ \boxed{\forall}\ |\ \boxed{\forall^{\perp}}\ \}\ \{\ \text{pat}\ \boxed{::}\ \text{type}\ \|\ \boxed{,}\ \}^{+}\ \boxed{.}
\end{array}
$$

axiom ::= [{ VARID | CONID } $\boxed{.}$] lExp

Regel Ax-1P

Trägt das Axiom einen Namen, so darf der Name noch nicht für ein anderes Axiom verwendet worden sein.
Fehlerbehandlung:
Der Name des Axioms wird ignoriert, und es wird eine entsprechende Fehlermeldung ausgegeben.

Regel Ax-2P

Jede Axiom-Definition wird in eine gesonderte Liste für Axiom-Definitionen eingefügt.

6.2.6 Gleichung

dokumentiert:

decl ::= ...
| fun rhs [where]
| pat rhs [where]
| ...

geparst:

decl ::= ...
| opExp rhs [where]
| ...

Regel Eq-1P

Jede Gleichung wird in eine für Gleichungen und Signatur-Deklarationen gemeinsame Liste eingefügt.

6.2.7 Signatur-Deklaration

decl ::= ...
| { var || $\boxed{,}$ }$^+$ $\boxed{::}$ type
| ...

Regel Sig-1P

Eine Variable darf höchstens in einer Signatur-Deklaration auftreten, insbesondere nicht mehrmals in ein und derselben.
Fehlerbehandlung:
Alle nachfolgenden Deklarationen einer Variable werden ignoriert, und es wird eine entsprechende Warnung ausgegeben.

Regel Sig-2P

Jede Signatur-Deklaration, die nicht durch die Regel Sig-1P verworfen wurde, wird in eine für Gleichungen und Signatur-Deklarationen gemeinsame Liste eingefügt.

6.2.8 Prädikatenlogische und gewöhnliche Ausdrücke

Während dem Parsen von Ausdrücken wird nur die Regel Exp-1P angewandt. Alle weiteren Prüfungen wie zum Beispiel Typprüfung und Variablenbindung werden erst nach dem Parsen durchgeführt.

Regel Exp-1P

Infix-Ausdrücke werden durch die Produktion *opExp*0 erkannt. Folgen mehrere Anwendungen von Infix-Operatoren hintereinander, wie zum Beispiel in e_0 op_1 e_1 ... op_n e_n, so liegt der Ausdruck als Liste in Form eines entarteten Baumes vor. Anhand der Präzedenz und Assoziativität der beteiligten Operatoren wird die lineare Struktur mit dem Tidy-Infix-Algorithmus in einen entsprechenden Baum umgewandelt. Dies wird in Abschnitt 8.3.2 ausführlich beschrieben.
Wird ein Operator *op* in einem solchen Ausdruck verwendet, ohne daß zuvor eine Infix-Definition angegeben wurde, so wird implizit die Definition

`infix` *op*

eingefügt. Die implizite Definition (keine Assoziativität und Präzedenz 9) ist wie eine explizite zu behandeln, d.h. die Infix-Regeln müssen geprüft werden.
Bemerkung:
Die Regel Exp-1P kommt insbesondere auch in allen Unterausdrücken eines Ausdrucks zum Tragen. Aus `[a+b*c,a-b-c]` wird z.B. `[a+(b*c),(a-b)-c]`.

6.2.9 Typausdrücke

Während dem Parsen werden keine Kontextbedingungen für Typausdrücke geprüft.

6.3 Kontextchecks nach dem Parsen

Nach dem Parsen sind folgende Daten-Strukturen mit den entsprechenden Eintragungen vorhanden:

- Liste mit Data- und Type-Definitionen
- Liste mit Gleichungen und Signatur-Deklarationen
- Liste mit Axiom-Definitionen
- Dictionary mit Infix-Operatoren (Infix-Definition)
- Dictionary mit Spez-Operatoren (Op-Definition)

In der Symboltabelle sind folgende Einträge vorhanden:

- Typkonstruktoren
- Typsynonyme
- Konstruktorfunktionen

Zudem sind Infix-Operatoren in Ausdrücken gemäß ihren Präzedenzen und Assoziativitäten geklammert.

6.3.1 Data- und Type-Definitionen

Regel Data-1Chk

In jeder Konstruktorfunktion einer Data-Definition müssen alle verwendeten Typkonstruktoren durch Data- oder Type-Definitionen definiert worden sein.

Regel Type-1Chk

Im Typ auf der rechten Seite der Type-Definition müssen alle verwendeten Typkonstruktoren durch Data- oder Type-Definitionen definiert worden sein.

Regel Type-2Chk

Typsynonyme dürfen nicht rekursiv bzw. verschränkt-rekursiv definiert werden. Folgende Definitionen sind z.B. unzulässig:

- *(rekursiv)* `type Tree = (Tree,Integer,Tree);`
- *(verschränkt-rekursiv)* `type T1 = T2 Integer; type T2 a = T1;`

6.3.2 Typsynonym-Expansion

Regel Type-1Exp

Für jedes Typsynonym werden die vollen Expansionen berechnet und zugewiesen. Beispielsweise werden für die Definitionen

```
data Tree a       =  Lf a | Tree a :^:  Tree a;
type Point a      =  (a,a);
type TreePoint a  =  Point (Tree a);
type Foo a b      =  [TreePoint a] -> (TreePoint b);
```

die folgenden Expansionen berechnet:

```
Point a      =  (a,a)
TreePoint a  =  (Tree a,Tree a)
Foo a b      =  [(Tree a,Tree a)] -> (Tree b,Tree b)
```

6.3.3 Verschränkt-rekursive Definitionen

Die Definitionen werden in Gruppen verschränkt rekursiver Definitionen aufgeteilt. Hierzu wird ein Superstrukturgraph benötigt.
Sei also G der Graph, in dem jedem Typkonstruktor und jedem Typsynonym bijektiv ein Knoten zugeordnet wird und eine gerichtete Kante von K_1 nach K_2 genau dann existiert, wenn sich K_1 auf K_2 abstützt. Die Data- oder Type-Definition K_1 stützt sich auf K_2 ab, wenn K_2 in der Definition von K_1 vorkommt. Diese Abhängigkeitsrelation partitioniert den Graphen G in Äquivalenzklassen, den starken Zusammenhangskomponenten.
Sei also G^* der (gerichtete und azyklische) Superstrukturgraph von G, dessen Knoten die stark zusammenhängenden Komponenten des Graphen G bilden.

Regel Data-1SCC

Aus G wird der Superstrukturgraph G^* berechnet. Die Knoten von G^* werden als die Gruppen von G^* bezeichnet. Hierzu müssen die Data- und Type-Definitionen analysiert werden, um die Abhängigkeiten zu ermitteln.

Regel Data-2SCC

In jeder Gruppe von G^* werden für alle Typkonstruktoren und Typsynonyme Kinds hergeleitet und zugewiesen (Kind-Inferenz).
Anschließend wird für jede Konstruktorfunktion ein Kind-Check durchgeführt, wobei alle Typen $well - kinded$ sein müssen. Ein Typ ist well-kinded, falls $*$ sein Kind ist.
Beispiel:

```
data Tree a     =  Lf a | Tree a :^:  Tree a
type IntTree    =  Tree Integer
data Tuple a b  =  T a b
type TupleS     =  Tuple Integer Float
type F          =  Tuple Char
```

Hergeleitete Kinds:

`Tree`	`::`	$* \to *$
`IntTree`	`::`	$*$
`Tuple`	`::`	$* \to * \to *$
`TupleS`	`::`	$*$
`F`	`::`	$* \to *$, `F` ist nicht well-kinded und wird zurückgewiesen

Regel Data-3SCC

Für jede Konstruktorfunktion wird der Typ berechnet.
Beispiel:

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

Hergeleitete Typen:

```
Lf    ::  a -> Tree a
:^:   ::  Tree a -> Tree a -> Tree a
```

6.3.4 Infix-Definitionen

Regel Infix-1Chk

Wurde ein Infix-Operator in einer Infix-Definition angegeben, so muß dieser auch durch eine Top-Level-Definition definiert worden sein. Die Definition des Infix-Operators kann vor oder nach der entsprechenden Infix-Definition stehen.
Fehlerbehandlung:
Infix-Operatoren in Infix-Definitionen ohne entsprechende Top-Level-Definition werden ignoriert, und es wird eine Warnung ausgegeben.

6.3.5 Op-Definitionen

Regel Op-1Chk

Wurde ein Spez-Operator in einer Op-Definition angegeben, so darf dieser nicht durch eine Top-Level-Definition definiert worden sein.
Fehlerbehandlung:
Spez-Operatoren in Op-Definitionen mit entsprechender Top-Level-Definition werden ignoriert, und es wird eine Warnung ausgegeben.

6.3.6 Umwandeln von Gleichungen in Bindungen

Regel Eq-1Bnd

Die Gleichungen werden zu Bindungen (*bindings*) zusammengefaßt. Hierzu wird die linke Seite einer jeden Gleichung überprüft. Da sie als *opExp* geparst wird, muß geprüft werden, ob sie gültig ist und ob eine Funktionsbindung oder eine Pattern-Bindung vorliegt. Wird der Ausdruck nicht als Funktionsbindung oder Pattern-Bindung erkannt, ist die linke Seite ungültig.
Test auf Funktionsbindung.
Folgende Fälle sind möglich:

a) Infix-Ausdruck: Es liegt eine Funktionsbindung vor, falls der Infix-Operator ein Variablen-Operator (varop) ist. In diesem Fall definiert der Operator die Funktion.

b) Funktionsapplikation: Es wird der Funktionsteil des Ausdrucks auf Funktionsbindung geprüft.

c) Atomic-Ausdruck: Es liegt eine Funktionsbindung vor, falls der Atomic-Ausdruck eine Variable oder eine Section mit Variablen-Operator (varop) ist. Im ersten Fall definiert die Variable, im zweiten Fall der Variablen-Operator die Funktion. Ist der Atomic-Ausdruck ein geklammerter Ausdruck, so wird der geklammerte Ausdruck auf Funktionsbindung geprüft.

d) Alle übrigen Formen stellen keine Funktionsbindungen dar.

Test auf Patternbindung.
Folgende Fälle sind möglich:

a) Infix-Ausdruck: Es liegt eine Pattern-Bindung vor, falls der Infix-Operator ein Konstruktor-Operator (conop) ist.

b) Funktionsapplikation: Es wird der Funktionsteil des Ausdrucks auf Patternbindung geprüft.

c) Atomic-Ausdruck: Es liegt eine Patternbindung vor, falls der Atomic-Ausdruck eine Konstruktor-Variable (conid), ein As-Pattern, eine aufzählende Liste, ein Tupel, das Unit-Element `()` oder eine Section mit Konstruktor-Operator (conop) ist. Ist der Atomic-Ausdruck ein geklammerter Ausdruck, so wird der geklammerte Ausdruck auf Patternbindung geprüft.

d) Alle übrigen Formen stellen keine Pattern-Bindungen dar.

Für eine Funktionsbindung wird die Regel Eq2-Bnd und für eine Pattern-Bindung die Regel Eq-3Bnd angewandt. Anschließend wird die Prüfung mit Regel Eq4-Bnd fortgesetzt.
Fehlerbehandlung:
Eine Gleichung mit einer unzulässigen linken Seite wird ignoriert, und es wird eine entsprechende Warnung ausgegeben.

Regel Eq-2Bnd (Funktionsbindung)

a) Alle Gleichungen einer Funktion müssen die gleiche Stelligkeit besitzen.

b) Der Funktionsname darf nicht durch eine vorausgegangene Gleichung definiert worden sein.

Die Regel Op-1Chk stellt bereits sicher, daß der Funktionsname nicht durch eine Op-Definition eingeführt worden ist.
Fehlerbehandlung:
Eine Gleichung mit einer differierenden Stelligkeit wird ignoriert, und es wird eine entsprechende Warnung ausgegeben.
Bemerkung:
In Gofer wird zusätzlich folgende Bedingung geprüft:
Gleichungen zu einer Funktion müssen hintereinander folgen, so daß zwischen den Alternativen einer Funktionsdefinition keine Pattern-Bindungen und keine Funktionsbindungen anderer Funktionen stehen dürfen.

Regel Eq-3Bnd (Pattern-Bindung)

a) Mit der Regel Pat-1Chk wird sichergestellt, daß das Pattern eine gültige Struktur besitzt.

b) Es wird sichergestellt, daß das Pattern mindestens eine Variable definiert. Damit werden Gleichungen wie `True = False` ausgeschlossen.

c) Keine Variable im Pattern darf durch eine vorausgegangene Gleichung definiert worden sein.

d) Die Regel Op-1Chk stellt bereits sicher, daß keine Variable im Pattern durch eine Op-Definition eingeführt worden ist.

Regel Eq-4Bnd

Zusammengehörige Gleichungen einer Funktion werden zu einer Bindung zusammengefaßt. Eine Gleichung mit Pattern-Bindung ist bereits eine einzelne Bindung.

6.3.7 Explizite Signatur-Deklarationen

Regel Sig-1Chk

Wurde eine Variable in einer Signatur-Deklaration angegeben, so muß diese auch durch eine Top-Level-Gleichung definiert worden sein. Die Definition der Variablen kann vor oder nach der entsprechenden Signatur-Deklaration stehen.
Fehlerbehandlung:
Variablen in Signatur-Deklarationen ohne entsprechende Top-Level-Gleichung werden ignoriert, und es wird eine Warnung ausgegeben.

Regel Sig-2Chk

Im Typ auf der rechten Seite der Signatur-Deklaration müssen alle verwendeten Typkonstruktoren durch Data- oder Type-Definitionen definiert worden sein.
Der Typ muß well-kinded sein, d.h. sein Kind muß * sein.

Regel Sig-3Chk

Signatur-Deklarationen werden den zugehörigen Variablen in den Bindungen zugeordnet.

6.3.8 Bindungen

Regel Bnd-1Chk

Für die folgenden Prüfungen wird ein neuer Sichtbarkeitsbereich betreten. Im Top-Level-Bereich stellt dieser den globalen Sichtbarkeitsbereich dar.

a) Die durch eine Funktionsbindung definierte Variable (Funktionsname) wird in die Symboltabelle für den aktuellen Namensraum mit der entsprechenden AST-Struktur der Bindung eingefügt.

b) Die durch eine Pattern-Bindung definierten Variablen werden in die Symboltabelle für den aktuellen Namensraum mit der entsprechenden (für alle Variablen gleichen) AST-Struktur der Bindung eingefügt.

Regel Bnd-2Chk

Die nachfolgenden Prüfungen erstellen für jede Bindung auch eine Liste der abhängigen Bindungen, die für die Konstruktion des Superstrukturgraphen benötigt werden.

Regel Bnd-3Chk

a) Für Funktionsbindungen werden alle Alternativen mit Regel Bnd4-Chk geprüft.

b) Für Pattern-Bindungen wird die rechte Seite der Gleichung mit Regel Bnd-5Chk geprüft.

Regel Bnd-4Chk (Alternative)

a) Die Argumente der Funktion auf der linken Seite müssen gültige Pattern sein. Sie werden mit Hilfe der Regel Pat-2Chk geprüft.

b) Für die Prüfung der rechten Seite der Gleichung (Alternative) wird ein neuer Sichtbarkeitsbereich mit den in a) ermittelten Pattern-Variablen erzeugt und nach Prüfung der Alternative wieder verlassen.

c) Die rechte Seite der Gleichung wird mit Regel Bnd-5Chk geprüft.

Fehlerbehandlung:
Eine fehlerhafte Alternative wird ignoriert, und es wird eine entsprechende Warnung ausgegeben.

Regel Bnd-5Chk (Rhs)

Enthält die rechte Seite lokale Definitionen (`where`), so werden zusätzlich folgende Schritte durchgeführt:

a) Die lokalen Definitionen werden analog zu Gleichungen und Signatur-Deklarationen im Top-Level-Bereich geprüft (Punkte 1 bis 4). Insbesondere wird ein neuer Sichtbarkeitsbereich eingeführt.

b) Der neue Sichtbarkeitsbereich wird nach der Prüfung Bnd-5Chk wieder verlassen.

Enthält die rechte Seite bewachte Ausdrücke, so werden Wächter und bewachte Ausdrücke mit Regel Bnd-6Chk geprüft. Andernfalls ist die rechte Seite ein einfacher Ausdrück, welcher ebenfalls mit Bnd-6Chk geprüft wird.

Regel Bnd-6Chk (Ausdruck)

a) Ist ein Ausdruck getypt, so muß der Typ well-kinded sein (Kind $*$), und im Typ müssen alle verwendeten Typkonstruktoren durch Data- oder Type-Definitionen definiert worden sein.

b) Ist der Ausdruck ein Infix-Ausdruck, so wird die linke und die rechte Seite mit Regel Bnd-6Chk b) geprüft. Zudem wird der Operator mit Regel Bnd-8Chk geprüft.

c) Ist der Ausdruck ein Präfix-Ausdruck, so wird der verbleibende Ausdruck mit Regel Bnd-6Chk c) geprüft. Für den Präfix-Operator kommt nur das unäre Minus in Frage.

d) Ist der Ausdruck ein `if`-Ausdruck, so werden die Bedingung, der Then- und der Else-Ausdruck mit Regel Bnd-6Chk geprüft.

e) Für Lambda-Ausdrücke wird Regel Bnd-9Chk geprüft.

f) Für `let`-Ausdrücke wird Regel Bnd-10Chk geprüft.

g) Für `case`-Ausdrücke wird Regel Bnd-11Chk geprüft.

h) Wird die Struktur des Ausdrucks durch keinen der Fälle a) bis g) abgedeckt, so stellt der Ausdruck eine Funktionsanwendung dar. Der Ausdruck ist somit eine nicht leere Liste von Atomic-Ausdrücken, wobei der erste Atomic-Ausdruck die Funktion und alle übrigen Atomic-Ausdrücke Argumente sind. Jeder Atomic-Ausdruck wird mit der Regel Bnd-7Chk geprüft.

Bemerkung:
Die Struktur von Ausdrücken weicht von der dokumentierten Grammatik ab.

Regel Bnd-7Chk (Atomic-Ausdruck)

Der Atomic-Ausdruck hat eine der folgenden Formen:

a) VARID, VAROP: Es wird Regel-Bnd8Chk geprüft.

b) CONID, CONOP: Es wird Regel-Bnd8Chk geprüft.

c) INTEGER, FLOAT, CHAR, STRING, (), BOT: Es werden keine weiteren Prüfungen durchgeführt.

d) Tupel: Jeder Ausdruck im Tupel wird mit Regel Bnd-6Chk geprüft.

e) Endliche und arithmetische Liste: Jeder in der Liste auftretende Ausdruck wird mit Regel Bnd-6Chk geprüft.

f) List-Comprehension: Es wird Regel Bnd-12Chk geprüft.

g) `!`-Ausdruck: Der Ausdruck wird mit Regel Lexp-1Chk geprüft.

h) Alle anderen Formen, insbesondere As-Pattern und Wildcard, sind in Ausdrücken nicht zulässig.

Regel Bnd-8Chk (VARID, VAROP, CONID und CONOP)

a) Der Operator ist varid oder varop: Der Operator wird gemäß den Sichtbarkeitsregeln sukzessive in den Sichtbarkeitsbereichen gesucht. Wurde keine Definition gefunden, so ist die Variable oder der Operator undefiniert. Wurde die Definition gefunden, so wird sie zur Dependents-Liste von dem Scope hinzugefügt, in dem die Definition eingetragen wurde. Dies wird für die Berechnung der stark zusammenhängenden Komponenten verschränkt rekursiver Bindungen benötigt (vgl. Abschnitt 6.3.9).

b) Der Operator ist conid oder conop: Der Operator muß als Konstruktorfunktion definiert sein.

Regel Bnd-9Chk (Lambda-Ausdruck)

a) Die Argumente der Lambda-Funktion müssen gültige Pattern sein. Sie werden mit Hilfe der Regel Pat-2Chk geprüft.

b) Für die Prüfung der rechten Seite der Ausdrucks wird ein neuer Sichtbarkeitsbereich mit den in a) ermittelten Pattern-Variablen erzeugt und nach Prüfung des Ausdrucks wieder verlassen.

c) Der Ausdruck auf der rechten Seite wird mit Regel Bnd-6Chk geprüft.

Bemerkung:
Vgl. Regel Bnd-4Chk.

Regel Bnd-10Chk (Let-Ausdruck)

a) Die lokalen Definitionen werden analog zu Gleichungen und Signatur-Deklarationen im Top-Level-Bereich geprüft (Punkte 1 bis 4). Insbesondere wird ein neuer Sichtbarkeitsbereich eingeführt.

b) Der neue Sichtbarkeitsbereich wird nach der Prüfung Bnd-10Chk wieder verlassen.

c) Der Ausdruck wird mit Regel Bnd-6Chk geprüft.

Bemerkung:
Vgl. Regel Bnd-5Chk.

Regel Bnd-11Chk (Case-Ausdruck)

a) Der Ausdruck wird mit Regel Bnd-6Chk geprüft.

b) Jede Case-Alternative wird wie eine Funktionsalternative geprüft (vgl. Regel Bnd-4Chk).

Regel Bnd-12Chk (List-Comprehension)

Der Ausdruck auf der linken Seite wird mit Regel Bnd-6Chk geprüft. Es wird ein neuer Sichtbarkeitsbereich betreten. Anschließend werden die Comprehensions von links nach rechts geprüft:

a) Wächter (*exp*): Der Ausdruck wird mit Regel Bnd-6Chk geprüft.

b) Lokale Definition (*pat* `=` *exp*): Die lokale Definition wird analog zu einer Gleichung im Top-Level-Bereich geprüft (Punkte 1 bis 4). Es wird allerdings kein neuer Sichtbarkeitsbereich angelegt, sondern der bereits angelegte Sichtbarkeitsbereich verwendet.

c) Generator (*pat* `<-` *exp*): Der Ausdruck wird mit Regel Bnd-6Chk geprüft. Das Pattern muß ein gültiges Pattern sein. Es wird mit Hilfe der Regel Pat-1Chk geprüft. Für die Prüfung aller weiteren Comprehensions werden die ermittelten Pattern-Variablen zum Sichtbarkeitsbereich hinzugefügt.

Nach Prüfung aller Comprehensions wird der aktuelle Sichtbarkeitsbereich wieder verlassen.

6.3.9 Verschränkt-rekursive Bindungen

Die Bindungen werden in Gruppen verschränkt rekursiver Definitionen aufgeteilt. Hierzu wird analog zu Abschnitt 6.3.3 ein Superstrukturgraph benötigt. Die Knoten entsprechen hier den Bindungen. Eine Bindung K_1 stützt sich auf K_2 ab, wenn eine Funktion der Bindung K_2 in der Definition von K_1 aufgerufen wird.

Regel Bnd-1SCC

Aus G wird der Superstrukturgraph G^* berechnet. Die Knoten von G^* werden als die Gruppen von G^* bezeichnet.

6.3.10 Axiom-Definition und prädikatenlogische Ausdrücke

Regel Ax-1Chk

Alle Ausdrücke einer Axiom-Definition werden mit Regel Lexp-1Chk geprüft.

Regel Lexp-1Chk

Für prädikatenlogische Ausdrücke gilt im wesentlichen die Regel Bnd-6Chk.

a) Für Infix-Operatoren sind zudem $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ und `=` bzw. ihre ASCII-Ersatzzeichen zulässig. Siehe Tabelle 5.1.

b) Für Präfix-Operatoren sind zudem $\neg$ und δ bzw. ihre ASCII-Ersatzzeichen zulässig. Siehe Tabelle 5.1.

c) Für Atomic sind zudem `TT` und `FF` zulässig.

d) Des weiteren sind die Quantoren-Ausdrücke zulässig, welche mit Hilfe der Regeln Lexp-2Chk und Lexp-3Chk geprüft werden.

Regel Lexp-2Chk (gewöhnliche Quantoren)

a) Zuerst werden die Pattern und Typen der Reihe nach geprüft. Hierzu werden die Pattern mit Pat-2Chk und die Typen analog zu Sig-2Chk geprüft.

b) Für den Ausdruck wird ein neuer Sichtbarkeitsbereich mit den aus a) ermittelten Pattern-Variablen erzeugt und nach Prüfung des Ausdrucks wieder verlassen.

c) Der Ausdruck wird gemäß Lexp-1Chk geprüft.

Regel Lexp-3Chk (Pattern-Quantoren)

a) Für den Ausdruck wird ein neuer Sichtbarkeitsbereich erzeugt.

b) Die Pattern und Ausdrücke werden der Reihe nach geprüft. Das Pattern wird gemäß Pat-2Chk geprüft und die ermittelten Pattern-Variablen zum Sichtbarkeitsbereich hinzugefügt. Jede Pattern-Variable darf noch nicht im Sichtbarkeitsbereich gebunden sein. Anschließend wird der Ausdruck mit Regel Bnd-6Chk geprüft.

c) Der Ausdruck wird gemäß Lexp-1Chk geprüft.

d) Der aktuelle Sichtbarkeitsbereich wird wieder verlassen.

6.3.11 Pattern

Regel Pat-1Chk (einzelnes Pattern)

Zuerst wird die Liste der bereits gefundenen Pattern-Variablen mit der leeren Liste initialisiert.
Anschließend wird das Pattern mit Regel Pat-3Chk geprüft.

Regel Pat-2Chk (Pattern-Liste)

Zuerst wird die Liste der bereits gefundenen Pattern-Variablen mit der leeren Liste initialisiert.
Anschließend werden die Pattern in der Liste der Reihe nach mit Regel Pat-3Chk geprüft.

Regel Pat-3Chk (opExp-Pattern)

a) Ist der Ausdruck ein Infix-Ausdruck, so wird mit Regel Pat-4Chk ggf. auf ein (n+k)-Pattern geprüft.

b) Ist der Ausdruck eine Funktionsanwendung, so ist der Ausdruck eine nicht leere Liste von Atomic-Ausdrücken. Der erste Atomic-Ausdruck muß eine definierte Konstruktorfunktion sein. Die Stelligkeit muß der Anzahl der restlichen Atomic-Ausdrücke entsprechen. Die restlichen Atomic-Ausdrücke werden mit Regel Pat-6Chk geprüft.

c) Ist der Ausdruck ein Atomic-Ausdruck, so wird der Atomic-Ausdruck mit der Regel Pat-6Chk geprüft.

d) Wird die Struktur des Ausdrucks durch keinen der Fälle a) bis c) abgedeckt, so ist die Struktur nicht zulässig.

Regel Pat-4Chk ((n+k)-Pattern)

a) Das Pattern hat die Struktur (var `+` x):

   - falls x ein Integer-Literal > 0 ist, dann wird ein (n+k)-Pattern erkannt und für var wird Regel Pat-6Chk geprüft.
   - falls x kein Integer-Literal ist oder x ein Integer-Literal ≤ 0 ist, wird ein Fehler erzeugt.

b) Hat das Pattern nicht die obige Struktur, so wird Regel Pat-5Chk geprüft.

Regel Pat-5Chk (Infix-Pattern)

a) Der Operator muß als Konstruktorfunktion definiert sein, und die Stelligkeit muß 2 sein.

b) Linke und rechte Seite werden mit Regel Pat-3Chk geprüft.

Andernfalls wird ein Fehler erzeugt.

Regel Pat-6Chk (Atomic-Pattern)

Der Atomic-Ausdruck hat eine der folgenden Formen:

a) VARID, VAROP: Es wird eine neue Pattern-Variable eingeführt.

b) CONID, CONOP: Der Operator muß als Konstruktorfunktion definiert sein und die Stelligkeit 0 haben.

c) INTEGER, CHAR, STRING, (), Wildcard: Es werden keine weiteren Prüfungen durchgeführt.

d) As-Pattern: Es wird eine neue Pattern-Variable eingeführt, und der Ausdruck wird mit Regel Pat-3Chk geprüft.

e) Tupel: Jeder Ausdruck im Tupel wird mit Regel Pat-3Chk geprüft.

f) Endliche Liste: Jeder in der Liste auftretende Ausdruck wird mit Regel Pat-3Chk geprüft.

g) Section: Der Ausdruck ist von der Form (*pat conop*) oder (*conop pat*). Das Atomic-Pattern pat wird mit Regel Pat-6Chk geprüft. Der Operator *conop* muß als Konstruktorfunktion definiert sein und die Stelligkeit 2 haben.

h) Alle übrigen Formen sind unzulässige Pattern.

Eine neue Pattern-Variable kann genau dann eingeführt werden, wenn die Variable nicht bereits als Pattern-Variable eingeführt wurde. Ansonsten wird ein Fehler erzeugt.

Kapitel 7

Typsystem und Typinferenz

7.1 Typsystem von Frisco F

7.1.1 Typausdrücke

In „Frisco F“ wird jedem Wert ein Typ zugeordnet. Um auszudrücken, daß der Ausdruck e vom Typ τ ist, schreibt man $e :: \tau$, also zum Beispiel `1 :: Int` oder `3.141 :: Float`. Typausdrücke werden aus Typkonstruktoren gebildet. Sind $\tau_1, ..., \tau_n$ Typausdrücke und ist C ein n-stelliger Typkonstruktor, so ist $C\tau_1...\tau_n$ ebenfalls ein Typausdruck. Für $n = 0$ spricht man von null-stelligen Typkonstruktoren. Zum Beispiel sind `Int` und `Float` null-stellige Typkonstruktoren. Wir werden im folgenden die Struktur von Typausdrücken schrittweise aufbauen.

Vordefinierte Typkonstruktoren

Die Datentypen `Bool`, `Int`, `Float` und `Char` sind bereits vordefiniert. Es handelt sich um null-stellige Typkonstruktoren, da keine weiteren Typausdrücke als Argumente im Typausdruck vorkommen. Zudem sind die Typkonstruktoren für Listen, Funktionen, Tupel und den speziellen Unit-Typ vordefiniert:

- Die Menge aller Listen vom Typ τ wird durch $[\tau]$ angegeben.

- Die Menge aller Funktionen mit Argumenten vom Typ τ_1 und Ergebnis vom Typ τ_2 wird durch $\tau_1 \rightarrow \ \tau_2$ angegeben. Der Typ $\tau_1 \rightarrow \ \tau_2 \rightarrow \ ... \rightarrow \ \tau_n$ entspricht einer Funktion mit Argumenten vom Typ $\tau_1, ..., \tau_{n-1}$ und Ergebnis vom Typ τ_n. Mit Hilfe des Prinzips des Currying[1] wird dieser Typ als $\tau_1 \rightarrow \ (\tau_2 \rightarrow \ (... \rightarrow \ \tau_n)...)$ aufgefaßt (vgl. Abschnitt 3.1.2).

- Für jedes $n \geq 2$ wird die Menge aller n-Tupel mit i-ter Komponente vom Typ τ_i angegeben durch $(\tau_1, \tau_2, ..., \tau_n)$.

- Ein spezieller (null-stelliger) Typkonstruktor ist das leere Tupel `()`, auch Unit-Typ genannt. Sein einziges Element ist das Unit-Element `()`.

[1] Oder auch Schönfinkeln; rührt von den Logikern H. B. Curry und M. Schönfinkel her, die die kombinatorische Logik entwickelt haben.

Beispiele:

- Das Typsynonym `String` ist definiert als `[Char]`, d.h. der Typ `String` entspricht der Menge aller Listen von Zeichen (Zeichenketten).
- Die Menge der komplexen Zahlen könnte durch ein 2-Tupel von Gleitpunktzahlen nachgebildet werden: `type Complex = (Float,Float)`.
- Eine Funktion von `Int` nach `Float` hat den Typ `Int -> Float`.
- Der Typ einer Funktion nach `Char`, die explizit keine Argumente nimmt, kann durch den Typ `() -> Char` beschrieben werden.

Polymorphie

Enthält ein Typausdruck keine Typvariablen, so wird er als monomorph bezeichnet. Zum Beispiel sind `Int`, `[Char]` und `(Int, Int) -> Float` monomorphe Typen.
Enthält ein Typausdruck hingegen mindestens eine Typvariable, so wird er als polymorph bezeichnet. Zum Beispiel sind $\alpha \rightarrow \beta$ und $[\alpha] \rightarrow Int$ polymorphe Typen. Diese Schreibweise stellt allerdings eine Abkürzung dar, denn die Typvariablen sind implizit durch einen $\forall$-Quantor gebunden. Korrekt werden die Typen als $\forall\alpha\forall\beta.\alpha \rightarrow \beta$ und $\forall\alpha.[\alpha] \rightarrow Int$ geschrieben. Hat also ein Wert einen polymorphen Typ, so ist damit gemeint, daß dieser Wert im Prinzip unendlich viele monomorphe Typen als Instanzierung annehmen kann. All diese Typen sind allerdings von der angegebenen Form, so daß der polymorphe Typ als Schablone aufgefaßt werden kann.
Beispiel:

> Die Identitätsfunktion *id* hat den Typ $\forall\alpha.\alpha \rightarrow \alpha$ oder kurz $\alpha \rightarrow \alpha$. Damit ist gemeint, daß die Funktion *id* jeden beliebigen Typ annehmen kann, sofern nur Argument- und Ergebnistyp übereinstimmen.

Die hier vorgestellten Typen werden als *shallow types* bezeichnet, in denen keine Quantoren in Teilausdrücken vorkommen. Ist also σ ein polymorpher Typ mit $\sigma = \forall\alpha_1 ... \forall\alpha_n.\tau, n \geq 0$, so enthält τ keine Quantoren. Diese Art von Polymorphie wird auch als flache Polymorphie (*shallow polymorphism*) bezeichnet (vgl. [Sok91]).
Typausdrücke werden also aus Typvariablen und Typkonstruktoren gebildet. Somit ist jede Typvariable ein Typausdruck, und sind $\tau_1, ..., \tau_n, n \geq 0$ Typausdrücke und ist C ein n-stelliger Typkonstruktor, so ist $C\tau_1 ... \tau_n$ ebenfalls ein Typausdruck. Die vordefinierten Typkonstruktoren für Listen, Funktionen und Tupel werden allerdings in Infix- und Mixfix-Schreibweise verwendet.
Beispiele:

- Die Funktion `length`, die die Länge einer Liste berechnet, hat den Typ $[\alpha] \rightarrow Int$.
- Die Funktion `fst`, die das erste Element eines 2-Tupels liefert, hat den Typ $(\alpha, \beta) \rightarrow \alpha$, und die Funktion `snd`, die das zweite Element eines 2-Tupels liefert, hat den Typ $(\alpha, \beta) \rightarrow \beta$.

- Die Funktion `map`, die auf jedes Element einer gegebenen Liste eine Funktion f anwendet und dadurch eine weitere Liste der Ergebnisse von f erzeugt, hat den Typ $(\alpha \to \beta) \to [\alpha] \to [\beta]$. Das erste Argument ist eine Funktion von α nach β, und das zweite Argument ist eine Liste von Werten vom Typ α. Das Ergebnis ist eine Liste von Werten vom Typ β.

Durch die Existenz von polymorphen Typen können auch Funktionen polymorph sein. Man spricht dann von einer polymorphen Funktion. Diese Art von Polymorphie wird auch als *parametrische Polymorphie* bezeichnet, bei welcher der gleiche Funktionsrumpf für unterschiedliche Werte verwendet werden kann. Beispielsweise wird beim Aufruf der Funktion `length` zur Berechnung der Länge einer Liste ein und derselbe Code ausgeführt, unabhängig davon, welchen Typ die Elemente dieser Liste haben.
Im Gegensatz hierzu spricht man von *ad-hoc-Polymorphie* oder *Überladen* (*overloading*), wenn für ein Funktionssymbol unterschiedliche Funktionsrümpfe in Abhängigkeit von den Argumenttypen definiert werden können. Beim Aufruf einer solchen Funktion wird in Abhängigkeit von den Argumenten entschieden, welcher spezielle Funktionsrumpf zur Ausführung kommt.

Konstruktorfunktionen

Zusammen mit den Typkonstruktoren sind auch diverse Konstruktorfunktionen definiert. Die Konstruktorfunktionen der vordefinierten Typkonstruktoren sind schon in der Grammatik verankert:

- Die Funktion `[]` $:: \alpha \to [\alpha]$ bildet aus einem Element eine Liste mit diesem Element, z.B. ist `[1]` die Liste mit dem Element `1`.
- Die Funktion `:` $:: \alpha \to [\alpha] \to [\alpha]$ bildet aus einem Element und einer Liste eine neue Liste, indem das Element am Anfang der Liste hinzugefügt wird. So ergibt der Ausdruck `1:[2]` die Liste `[1,2]`.
- Die Funktion `(,,...,)` $:: \alpha_1 \to \alpha_2 \to ... \to \alpha_n \to (\alpha_1, \alpha_2, ..., \alpha_n)$ bildet aus n Elementen ein n-Tupel. Der Typ des Tupels ergibt sich direkt aus den Typen der einzelnen Elemente. So ist `(1,1.5,'x')` ein 3-Tupel vom Typ `(Int,Float,Char)`.
- Genaugenommen sind auch die Literale der Typkonstruktoren `Int`, `Float`, `Char` und `String` Konstruktorfunktionen, so daß z.B. `17` eine Konstruktorfunktion des Typs `Int` ist.

Benutzerdefinierte Typkonstruktoren

Mit der `data`-Definition können weitere Typkonstruktoren eingeführt werden. So definiert zum Beispiel

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

den neuen einstelligen Typkonstruktor `Tree`. Für benutzerdefinierte Typkonstruktoren ist allerdings nur die Präfix-Schreibweise möglich, wohingegen zum Beispiel die Tupel-Typen in Mixfix-Schreibweise angegeben werden.
Durch die Definition eines Typkonstruktors werden auch Konstruktorfunktionen eingeführt, um Elemente dieses Typs konstruieren zu können. So sind zum Beispiel `Lf` und `:^:` Konstruktorfunktionen des Typkonstruktors `Tree`. Die Typen der Konstruktorfunktionen ergeben sich

direkt aus der Definition:

- `Lf :: a -> Tree a`, d.h. die Konstruktorfunktion `Lf` nimmt ein Argument und erzeugt daraus ein Tree-Objekt.
- `:^: :: Tree a -> Tree a -> Tree a`, d.h. die Konstruktorfunktion `:^:` nimmt zwei Tree-Objekte und erzeugt daraus ein neues Tree-Objekt.

Der vordefinierte Listen-Datentyp kann durch die folgende Definitionen nachgebildet werden:

```
data List a = Sigleton a | Cons a (List a)
```

Elemente werden dann wie folgt erzeugt:

`Singleton 1`	$\widehat{=}$	`[1]`
`Singleton (1.5,Lf 5)`	$\widehat{=}$	`[(1.5,Lf 5)]`
`Cons 1 (Singleton 2)`	$\widehat{=}$	`1:[2] = [1,2]`
`Cons (0.0,Lf 0) (Singleton (1.5,Lf 5))`	$\widehat{=}$	`(0.0,Lf 0):[(1.5,Lf 5)]`
		`= [(0.0,Lf 0),(1.5,Lf 5)]`

Prädikate

Die Verwendung von Typklassen (wie z.B. in Gofer oder Haskell) stellt eine Möglichkeit dar, ad-hoc-Polymorphie in die Programmiersprache einzuführen. Beispielsweise ist das Funktionssymbol `+` gewöhnlich überladen, um sowohl für Ganzzahlen als auch Gleitpunktzahlen anwendbar zu sein. Somit existieren die zwei unterschiedlichen Realisierungen

```
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
```

der Addition.

Eine weitere überladene Funktion ist die Gleichheitsfunktion `==`. Mit Hilfe dieser Funktion können Zahlen, Listen oder Tupel verglichen werden, indem strukturelle Übereinstimmung gelten soll. Beispielsweise gelten Listen als gleich, falls sie in der Länge übereinstimmen und die Elemente paarweise gleich sind. Die Gleichheit von Listen stützt sich also auf die Gleichheit der Elemente ab.

Obwohl Funktionen als Werte betrachtet werden, darf bei ihnen keine Gleichheit gefordert werden, denn gewöhnlich gelten zwei Funktionen f und g als gleich, wenn sie extensional gleich sind. Zwei Funktionen sind extensional gleich, wenn ihr Definitionsbereich D übereinstimmt und sie in ihren Bildern übereinstimmen, d.h. wenn gilt: $\forall x \in D. f(x) = g(x)$. Leider ist die extensionale Gleichheit nicht entscheidbar.

Stützt sich nun eine Funktion auf eine überladene Funktion ab, so gilt diese selbst als überladen, sofern der überladene Typ nicht durch eine explizite Signatur-Deklaration auf einen nicht-überladenen Typ eingeschränkt wird. Beispielsweise ist die Funktion `elem`, welche das Enthaltensein eines Elementes in einer Liste prüft, überladen, da sie sich auf die Gleichheit stützt. Der Versuch, der Funktion `elem` einen Typ zuzuweisen, ist jedoch zum Scheitern verurteilt. Einerseits ist der polymorphe Typ $\forall\alpha.\alpha \rightarrow [\alpha] \rightarrow \mathit{Bool}$ zu allgemein, denn die Funktion `elem` ist nicht auf Listen mit Funktionen anwendbar, da die Gleichheit von Funktionen nicht definiert ist. Andererseits ist die Aufzählung aller möglichen Typen nicht möglich, da zum Beispiel α durch die unendlich vielen Typen `(Int,Int)`, `(Int,Int,Int)`, ..., `(Int,Int, ..., Int)`, ... ersetzt werden kann. Einen Ausweg bietet die Verwendung von Prädikaten, indem man den Wertebereich einer Typvariablen durch Prädikate einschränkt. Der allgemeine polymorphe Typausdruck wird dann um Prädikate erweitert, so daß die Instanzierung nur mit solchen Typausdrücken möglich ist, die diese Prädikate erfüllen. Hierzu führt man die Typklasse Eq ein, die all diejenigen Typen enthält, für die die Gleichheit definiert ist. Unter Verwendung des Prädikats $\alpha \in Eq$ läßt sich nun der Typ der Funktion `elem` ausdrücken:

$$\texttt{elem} :: \forall \alpha . \alpha \in Eq \Rightarrow \alpha \rightarrow \ [\alpha] \rightarrow \ Bool$$

Da für Funktionen die Gleichheit nicht definiert ist, ist für Funktionstypen α auch das Prädikat $\alpha \in Eq$ nicht erfüllt. Somit kann die Funktion `elem` nicht für Funktionstypen instanziert werden, und das ist genau das, was wir ausdrücken wollten.

In „Frisco F" ist es nicht möglich, neue Typklassen einzuführen. Stattdessen existieren zwei (anonyme) Typklassen, die im folgenden mit Eq und Num bezeichnet werden. Die Typklasse Eq definiert die Funktionen `==` und `/=` zum Testen auf Gleich- und Ungleichheit, die Funktionen `<`, `<=`, `>` und `>=` zum Vergleichen, die Funktionen `min` und `max` zur Ermittlung des Minimums bzw. Maximums und die Funktion `hash` zum Berechnen eines Hashwertes. Diese Funktionen sind bis auf Funktionen selbst auf alle vordefinierten und benutzerdefinierten Datentypen anwendbar. Die Typklasse Num definiert die arithmetischen Operatoren `+`, `-`, `*`, `/`, die Funktion `fromInteger` zum Konvertieren einer Ganzzahl und die Funktion `negate` zum Negieren einer Zahl. Diese Funktionen sind auf Werte vom Typ `Int` und `Float` anwendbar. Für eine ausführliche Beschreibung sei auf Anhang C verwiesen.

Durch diese Einschränkung sind alle Prädikate von der Form $\alpha \in Eq$ oder $\alpha \in Num$, denn es gilt per Definition (vgl. Anhang C) für alle Typkonstruktoren C außer dem Funktionstypkonstruktor $\rightarrow$:

$$C\tau_1 ... \tau_n \in Eq \overset{\text{def}}{\Leftrightarrow} \tau_1, ..., \tau_n \in Eq$$

Eine Vereinfachung der Schreibweise kann nun wie folgt vorgenommen werden: Gibt es zu einer Typvariablen α das Prädikat $\alpha \in Eq$, so wird dieses weggelassen und jedes α im polymorphen Typausdruck durch $'\alpha$ ersetzt. Gibt es zu einer Typvariablen α das Prädikat $\alpha \in Num$, so wird dieses weggelassen und jedes α im polymorphen Typausdruck durch $''\alpha$ ersetzt (vgl. [Pau96]). Treffen beide Prädikate zu, so dominiert die Typklasse Num, da in diesem Fall das Prädikat $\alpha \in Num$ das Prädikat $\alpha \in Eq$ impliziert. Zudem wird der $\forall$-Quantor weggelassen. Für

$$\texttt{elem} :: \forall \alpha . \alpha \in Eq \Rightarrow \alpha \rightarrow \ [\alpha] \rightarrow \ Bool$$

schreiben wir dann

$$\texttt{elem} :: {'\alpha} \rightarrow \ ['\alpha] \rightarrow \ Bool$$

Damit erhält man die endgültige Form der Typausdrücke:

> Typausdrücke werden aus Typvariablen bzw. einfach und zweifach gestrichenen Typvariablen und Typkonstruktoren gebildet. Jede Typvariable bzw. einfach und zweifach gestrichene Typvariable ist ein Typausdruck. Sind $\tau_1, ..., \tau_n, n \geq 0$ Typausdrücke und ist C ein n-stelliger Typkonstruktor, so ist $C\tau_1 ... \tau_n$ ebenfalls ein Typausdruck. Die vordefinierten Typkonstruktoren für Listen, Funktionen und Tupel werden allerdings in Infix- und Mixfix-Schreibweise verwendet.

Die Funktionen der anonymen Typklassen Eq und Num haben somit folgende Signaturen:

```
==           ::  'a -> 'a -> Bool
/=           ::  'a -> 'a -> Bool
<            ::  'a -> 'a -> Bool
<=           ::  'a -> 'a -> Bool
>            ::  'a -> 'a -> Bool
>=           ::  'a -> 'a -> Bool
min          ::  'a -> 'a -> 'a
max          ::  'a -> 'a -> 'a
hash         ::  'a -> Int

+            ::  ''a -> ''a -> ''a
-            ::  ''a -> ''a -> ''a
*            ::  ''a -> ''a -> ''a
/            ::  ''a -> ''a -> ''a
fromInteger  ::  Int -> ''a
negate       ::  ''a -> ''a
```

7.1.2 Typisierung

Die Typisierung eines „Frisco F"-Programms ist Bestandteil der semantischen Analyse und erfolgt nach den Kontext-Checks (vgl. Kapitel 6) in drei Schritten.
Im ersten Schritt werden die Konstruktorfunktionen der `data`-Definitionen typisiert. So ergibt sich, wie bereits oben gezeigt, für die Konstruktorfunktion `:^:` der Typ `Tree a -> Tree a -> Tree a`.
Im zweiten Schritt werden alle Wertvereinbarungen in der Top-Level-Ebene typisiert. Grob gesprochen werden hierzu den auftretenden Namen monomorphe Typen zugewiesen, die dann abhängig von der Struktur der Deklarationen unifiziert werden. Tritt dabei ein Fehler auf, so wird ein Typfehler ausgegeben. Wurden hingegen alle Deklarationen typisiert, werden die ermittelten monomorphen Typen generalisiert, um die möglichst allgemeinen Typen zu erhalten. Die sich hieraus ergebenden Probleme werden im Anschluß kurz erläutert.
Im dritten Schritt werden die Axiome geprüft. Da die Wertvereinbarungen im Sichtbarkeitsbereich der Axiome liegen, müssen diese bereits typisiert worden sein.

Typisierung verschränkt rekursiver Wertvereinbarungen

Wertvereinbarungen werden immer in Gruppen typisiert. Einerseits müssen verschränkt rekursive Deklarationen, wie beispielsweise

```
even x  =  if x == 0 then True else odd (x - 1);
odd x   =  if x == 0 then False else even (x - 1);
```

immer zusammen typisiert werden. Auf der anderen Seite dürfen aber nicht zu viele Deklarationen zusammengenommen werden, da dadurch die gefundenen Typen im allgemeinen nicht am allgemeinsten sind. Werden zum Beispiel die folgenden Deklarationen zusammen getypt, so ergeben sich nicht die zu erwartenden Typen:

```
id x   =  x;
f _    =  id 1;
g x y  =  (id x, id y);
```

Die zu erwartenden Typisierungen sind `id :: a -> a`, `f :: a -> Int` und `g :: a -> b -> (a,b)`. Es ergibt sich jedoch durch die gemeinsame Typisierung `id :: Int -> Int`, `f :: a -> Int` und `g :: Int -> Int -> (Int,Int)`.
Um die korrekten Gruppen für die Typisierung zu finden, muß eine Analyse durchgeführt werden, welche Deklarationen sich auf welche Deklarationen abstützen. Dies ergibt einen gerichteten Graphen. Die stark zusammenhängenden Komponenten dieses Graphen bilden dann die Typisierungsgruppen, in denen sich verschränkt rekursive Deklarationen befinden. Es befindet sich allerdings keine Wertvereinbarung in einer Gruppe, die sich nicht auf eine andere Deklaration abstützt (außer sich selber).
Nach der Typisierung werden die gefundenen Typen generalisiert, um den möglichst allgemeinsten Typen zu erhalten. Für die Funktion `id` ergibt sich nach der Typisierung zuerst der monomorphe Typ $\alpha \to \alpha$ und nach der Generalisierung der polymorphe Type $\forall\alpha.\alpha \to \alpha$. Bei der Typisierung der Funktion `g` wird die Funktion `id` zweimal angewendet. In beiden Fällen wird der gefundene Typ der Funktion `id` nicht nur eingesetzt, sondern instanziert, da die Funktion nicht in dieser Gruppe typisiert wurde. Die Instanzierung des Typs $\forall\alpha.\alpha \to \alpha$ ergibt den monomorphen Typ $\alpha \to \alpha$, wobei für die Typvariablen noch nicht benutzte Typvariablen verwendet werden. Somit ergeben sich für die zwei Anwendungen der Funktion `id` die Typen $\tau_k \to \tau_k$ und $\tau_l \to \tau_l$ mit $k \neq l$. Im Falle der Anwendung `g 1 "Hello, world"` wird die Funktion `id` mit den monomorphen Typen $Int \to Int$ und $String \to String$ instanziert. Versucht man hingegen, die Funktion `id` in derselben Gruppe wie `g` zu typisieren, so tritt ein Typfehler auf, da dadurch versucht wird, die beiden oben genannten Typen in ein und dasselbe Typschema und somit `Int` und `String` zu unifizieren.

Explizite Signatur-Deklaration

Die Typisierung von Deklarationen mit expliziter Signatur-Deklaration bedarf einer besonderen Behandlung, da sichergestellt werden muß, daß die angegebene Signatur eine Instanz des hergeleiteten Typs ist. Beispielsweise ist die folgende Deklaration fehlerfrei

```
id' :: Int -> Int;
id' x = x;
```

und `id'` erhält den Typ `Int -> Int`. Hingegen ist die Deklaration

```
id' :: a -> Int;
id' x = x;
```

fehlerhaft. Der Typ `a -> Int` ist keine Instanz von `a -> a`, da bei einer Instanzierung die Typvariablen immer konsistent ersetzt werden müssen.

7.2 Typinferenz

Unter Typinferenz versteht man das Herleiten der Typen von Werten und Definitionen. Falls dies nicht möglich ist, wird ein Typfehler ausgegeben. Der hierzu verwendete Algorithmus wird in diesem Abschnitt beschrieben. Folgende Punkte müssen dabei berücksichtigt werden (vgl. [Fie88]):

1. Es muß ein syntaktisches Typschema definiert werden, mit welchem jedem Ausdruck in der Syntax ein eindeutiger (und allgemeinster) Typ zugewiesen werden kann. Solche Ausdrücke heißen *wohlgetypt* (*welltyped*) und die Typen heißen $wohlgeformte\ Typen$ $(well - typings)$. Kurz:
   Ausdruck wohlgetypt $\Leftrightarrow$ *dem Ausdruck kann ein Typ zugeordnet werden*
2. Das Typschema muß semantisch vollständig sein, d.h. jeder syntaktisch wohlgetypter Ausdruck ist auch semantisch frei von Typverletzungen. Daher ist bei der Auswertung eines wohlgetypten Ausdrucks sichergestellt, daß all seine primitiven Funktionen nur mit Objekten von passendem Typ angewandt werden. Kurz:
   syntaktisch wohlgetypt $\Leftrightarrow$ *semantisch wohlgetypt*
3. Der Algorithmus muß syntaktisch vollständig sein, d.h. findet der Algorithmus für einen Ausdruck einen Typ, so ist der Ausdruck auch tatsächlich wohlgetypt. Kurz:
   Algorithmus findet Typ $\Leftrightarrow$ *Ausdruck ist wohlgetypt*
4. Der Algorithmus soll in der Hinsicht vollständig sein, daß er den allgemeinsten Typ findet, falls der Ausdruck wohlgetypt ist. Kurz:
   Ausdruck wohlgetypt $\Leftrightarrow$ *Algorithmus findet allgemeinsten Typ*

Für eine ausführlichere Behandlung dieser Thematik sei auf [Jon92], [Fie88] und [Hin92] verwiesen.

7.2.1 Inferenzregeln

Grundlage des Typinferenz-Algorithmus ist die Formulierung von syntaxgesteuerten Inferenzregeln. Die hierzu notwendigen technischen Hilfsmittel werden im folgenden kurz eingeführt. Die Basis zur Verarbeitung von Typausdrücken ist die Substitution. Auf ihr bauen Instanzierung und Unifikation auf. Um mehrere Gruppen von Wertvereinbarungen und verschachtelte Deklarationen verarbeiten zu können, müssen die gesammelten Typinformationen in einer Menge von Typannahmen abgelegt werden. Generalisierung und Instanzierung sind notwendig, um polymorphe Typen handhaben zu können. Mit diesen Hilfsmitteln können dann die Herleitungsregeln formuliert werden.

Substitution

Die Substitution wird im allgemeinen auf Ausdrücke über einem Alphabet von Variablen und Funktionssymbolen angewendet (vgl. [Rob65]). In diesem speziellen Fall soll die Substitution für monomorphe Typausdrücke angewendet werden, wobei Typvariablen durch Typausdrücke ersetzt weden. Es sei hier nochmals erwähnt, daß auf den Unterschied zwischen monomorphen und polymorphen Typen unbedingt zu achten ist. Beispielsweise sind die Typen $[\alpha]$ und $\alpha \rightarrow \beta$ monomorph, wohingegen die Typen $\forall\alpha.[\alpha]$ und $\forall\alpha\forall\beta.\alpha \rightarrow \beta$ polymorph sind. Siehe hierzu auch [Kor95].

Definition 7.1 (Substitution) *Sei $\mathcal{T}$ die Menge der monomorphen Typterme über einer Menge von Typkonstruktoren $\mathcal{C}$ und Typvariablen $\mathcal{TV}$. Dann ist eine Substitution $\tilde{\mathcal{S}}$ eine totale Abbildung*

$$\tilde{\mathcal{S}} : \mathcal{TV} \rightarrow \mathcal{T}$$

Eine Ausweitung der Abbildung $\tilde{\mathcal{S}}$ auf die Abbildung $\mathcal{S}$ mit

$$\mathcal{S} : \mathcal{T} \rightarrow \mathcal{T}$$

kann auf folgende Weise vorgenommen werden.

$$\begin{aligned} \mathcal{S}(\alpha) &:= \tilde{\mathcal{S}}(\alpha), \alpha \in \mathcal{TV} \\ \mathcal{S}(c(\tau_1, \tau_2, ..., \tau_n)) &:= c(\mathcal{S}(\tau_1), \mathcal{S}(\tau_2), ...\mathcal{S}(\tau_n)), n \geq 0, c \in \mathcal{C} \end{aligned}$$

Eine Substitution besteht aus einer ggf. leeren Menge von Einzelsubstitutionen τ/α. Die Substitution selbst wird als $[\tau_1/\alpha_1, \tau_2/\alpha_2, ..., \tau_n/\alpha_n]$, $n \geq 0$ geschrieben, wobei die $\alpha_i, 0 < i \leq n$, paarweise verschieden sind. Diese Substitution bezeichnet eine simultane Ersetzung der Typvariablen α_i durch Terme τ_i. Für den Fall $n = 0$ erhalten wir die leere Substitution, also eine identische Abbildung $\mathcal{I}$.
Die Anwendung $\mathcal{S}(\sigma)$ einer Substitution $\mathcal{S} = [\tau_1/\alpha_1, \tau_2/\alpha_2, ..., \tau_n/\alpha_n]$, $n \geq 0$ auf den Typausdruck σ wird geschrieben als $\sigma[\tau_1/\alpha_1, \tau_2/\alpha_2, ..., \tau_n/\alpha_n]$.

Typannahmen

Werden Ausdrücke und Definitionen mit Hilfe des Typinferenz-Algorithmus getypt, müssen die ermittelten Typen gespeichert werden und abrufbar sein. Hierzu wird eine Menge von Typzuweisungen verwendet, so daß einer Variablen, also zum Beispiel einem Funktionsnamen ein Typ zugewiesen werden kann.

Definition 7.2 (Typannahmen, assumption set) *Unter einer Typannahme $\mathcal{A}$ versteht man eine ggf. leere Menge, die paarweise verschiedenen Variablen einen Typ zuweist, d.h.*

$$\mathcal{A} = \{x_1 : \tau_1, x_2 : \tau_2, ..., x_n : \tau_n\}, n \geq 0, x_i \neq x_j \text{ für } i \neq j$$

Um auszudrücken, daß in $\mathcal{A}$ für eine Variable x keine Typannahme enthalten ist, schreiben wir $\mathcal{A}_x$ und meinen damit

$$\begin{aligned} \mathcal{A}_x &:= \mathcal{A}, \text{ falls für } x \text{ keine Belegung in } \mathcal{A} \text{ existiert} \\ \mathcal{A}_x &:= \mathcal{A} \setminus \{x : \tau\}, \text{ falls } \mathcal{A} \text{ die Belegung } x : \tau \text{ für } x \text{ enthält} \end{aligned}$$

Das Anfügen einer Typannahme für die Variable x zu $\mathcal{A}$ entspricht der Vereinigung $\mathcal{A} \cup \{x : \tau\}$, wobei diese Operation nur definiert ist, falls $\mathcal{A}$ noch keine Typannahme für x enthält. Hierfür schreiben wir kurz $\mathcal{A}, x : \tau$.

Generalisierung und Instanzierung

Durch die Generalisierung wird einem Ausdruck in Abhängigkeit von Typannahmen ein möglichst allgemeiner Typ zugewiesen.

Definition 7.3 (Typ-Generalisierung) *Sei ν ein monomorpher Typausdruck und $\mathcal{A}$ eine Menge von Typannahmen. Unter FTV(ν) bzw. FTV($\mathcal{A}$) versteht man die Menge der freien, d.h. ungebundenen Typvariablen von ν bzw. $\mathcal{A}$. Dann ist die Generalisierung Gen(ν,$\mathcal{A}$) des monomorphen Typausdrucks ν unter Berücksichtigung der Typannahmen $\mathcal{A}$ definiert durch:*

$$Gen(\nu, \mathcal{A}) = \forall(FTV(\nu) \setminus FTV(\mathcal{A})).\nu$$

Durch die Generalisierung werden somit all diejenigen Typvariablen durch einen $\forall$-Quantor gebunden, für die nicht bereits eine Typannahme besteht.

Definition 7.4 (Typ-Instanzierung) *Sei $\forall\alpha_i.\nu$ ein polymorpher Typausdruck in den Typvariablen α_i und momomomophem Typaudruck ν. Dann heißt der monomorphe Typausdruck μ eine Instanz von $\forall\alpha_i.\nu$ genau dann, wenn es eine Substitution $\mathcal{S}$ gibt, so daß gilt:*

$$\mu = \mathcal{S}(\nu)$$

In diesem Fall gibt es Typen τ_i, so daß gilt:

$$\mu = \nu[\tau_i/\alpha_i]$$

Den Zusammenhang von Instanzierung und Generalsisierung zeigt Abbildung 7.1.

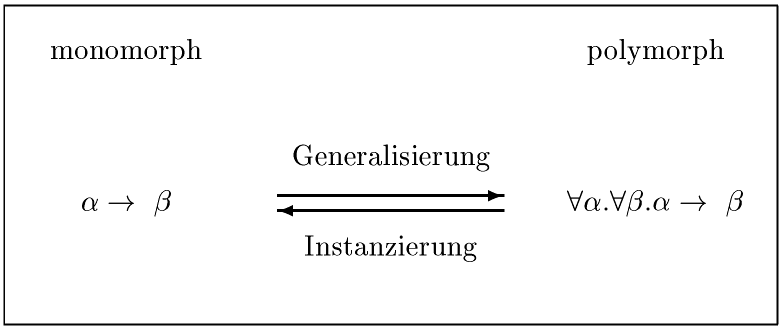


Abbildung 7.1: Generalisierung und Instanzierung

Herleitungsregeln

Die Herleitungsregeln geben an, unter welchen Vorraussetzungen Typannahmen hergeleitet werden können. Dabei bedienen wir uns der folgenden Schreibweise:

Definition 7.5 (Typ-Ableitung) *Kann aus der Menge $\mathcal{A}$ der Typannahmen abgeleitet werden, daß e den monomorphen Typ τ hat und die freien Variablen in e durch Typen in $\mathcal{A}$ spezifiziert sind, so schreiben wir $\mathcal{A} \vdash e : \tau$.*

Abbildung 7.2 zeigt die syntaxgesteuerten Inferenzregeln. Bei der Behandlung eines Let-Ausdrucks werden die lokalen Deklarationen generalisiert. Zur Vereinfachung wurden Regeln für Case-Ausdrücke weggelassen und nur Regeln für aufgezählte Listen angegeben.

7.2.2 Unifikation

Die Basis des Typinferenz-Algorithmus ist der Unifikations-Algorithmus von Robinson. Bei der Unifikation wird versucht, die Äquivalenz zweier Terme zu zeigen. Dies geschieht durch Finden einer passenden Substitution.
In folgenden werden kurz die grundlegenden Begriffe eingeführt, soweit sie für die weiteren Ausführungen benötigt werden. Grundlage hierfür ist im wesentlichen [Fie88], [Mil78] und [Kor95]. Für eine ausführliche Einführung und Beschreibung des Unifikations-Algorithmus sei auf [Rob65] verwiesen.

var	$\mathcal{A}_x, x : \tau \vdash x : \tau$
application	$\dfrac{\mathcal{A} \vdash f : \sigma \to \tau \quad \mathcal{A} \vdash x : \sigma}{\mathcal{A} \vdash (fx) : \tau}$
abstraction	$\dfrac{\mathcal{A}_x, x : \sigma \vdash e : \tau}{\mathcal{A} \vdash (\lambda x.e) : \sigma \to \tau}$
tuple	$\dfrac{\mathcal{A} \vdash e_i : \tau_i, 1 \leq i \leq n, n \geq 2}{\mathcal{A} \vdash (e_1, ..., e_n) : (\tau_1, ..., \tau_n)}$
simple list	$\dfrac{\mathcal{A} \vdash e_i : \tau, 1 \leq i \leq n, n \geq 1}{\mathcal{A} \vdash [e_1, ..., e_n] : [\tau]}$
explicit	$\dfrac{\mathcal{A} \vdash e : \tau}{\mathcal{A} \vdash (e :: \tau) : \tau}$
let	$\dfrac{\mathcal{A} \vdash e : \sigma \quad \mathcal{A}_x, x : Gen(\mathcal{A}, \sigma) \vdash f : \tau}{\mathcal{A} \vdash (\text{let } x = e \text{ in } f) : \tau}$
if	$\dfrac{\mathcal{A} \vdash x : Bool \quad \mathcal{A} \vdash t : \tau \quad \mathcal{A} \vdash e : \tau}{\mathcal{A} \vdash (\text{if } x \text{ then } t \text{ else } e) : \tau}$

Abbildung 7.2: syntaxgesteuerte Herleitungsregeln

Unifikator

Definition 7.6 (Unifikator) *Eine Substitution $\mathcal{U}$ wird Unifikator der Terme σ und τ genannt, falls $\mathcal{U}$ die beiden Terme in äquivalente Terme überführt, d.h. falls für $\sigma, \tau \in \mathcal{T}$ gilt:*

$$\mathcal{U}(\sigma) \equiv \mathcal{U}(\tau)$$

Die Terme σ und τ heißen unifizierbar, falls es einen Unifikator für σ und τ gibt.
Die Substitution $\mathcal{U}$ wird allgemeinster Unifikator der Terme σ und τ genannt, falls für jeden weiteren Unifikator $\mathcal{U}'$ der Terme σ und τ eine Substitution $\mathcal{S}$ exisitert, so daß gilt:

$$\mathcal{U}' = \mathcal{S}\mathcal{U}$$

Unifikations-Algorithmus

Robinson hat in [Rob65] gezeigt, daß es in einem Typsystem, wie „Frisco F“ es hat, einen solchen allgemeinsten Unifikator gibt, sobald überhaupt ein Unifikator existiert.

Satz 7.7 (Robinson, 1965) *Es gibt einen Algorithmus $\mathcal{V}$, dessen Eingabe ein Paar von Ausdrücken σ, τ (über einem Alphabet von Variablen) ist, so daß gilt:*

- *Terminiert $\mathcal{V}(\sigma,\tau)$ ohne Erfolg, so gibt es keinen Unifikator für σ und τ.*
- *Terminiert $\mathcal{V}(\sigma,\tau)$ mit Erfolg und liefert die Substitution U als Ergebnis. U hat dabei folgende Eigenschaften:*

1. *$U\sigma = U\tau$, d.h. U ist Unifikator von σ und τ.*
2. *Falls R ein weiterer Unifikator ist, dann gibt es eine Substitution S mit $R = SU$, d.h. U ist allgemeinster Unifikator von σ und τ.*
3. *In U sind nur solche Variablen betroffen, die in σ und τ vorkommen.*

Beispiele:

$$\begin{array}{lcl} \mathcal{V}(\alpha,\alpha) & = & \text{I (Identität)} \\ \mathcal{V}(\alpha \rightarrow \beta,\, Int \rightarrow Bool) & = & [Int/\alpha,\, Bool/\beta] \\ \mathcal{V}(\alpha \rightarrow Bool,\, Int \rightarrow Int) & = & \emptyset \text{ (}\mathcal{V}\text{ terminiert ohne Erfolg)} \end{array}$$

7.2.3 Typinferenz-Algorithmus

Auf der Grundlage des Kalküls von Abbildung 7.2 kann nun ein Typinferenz-Algorithmus entwickelt werden. Wir bezeichnen den Algorithmus im folgenden mit $\mathcal{W}$. Die Argumente von $\mathcal{W}$ sind eine Menge von Typannahmen und der zu typisierende Ausdruck. Als Ergebnis erhält man den Typ des Ausdrucks und eine Substitution, die aufgrund der rekursiven Arbeitsweise mitgeführt wird. Zudem ist der dargestellte Algorithmus dahingehend vereinfacht, daß er nicht mit Wertvereinbarungen, sondern nur mit Ausdrücken arbeitet.

Folgende Bezeichnungen werden verwendet:

- $\mathcal{A}$ ist eine Menge von Typannahmen.
- α, β usf. sind Typvariablen.
- τ, σ usf. stehen für Typen.
- S, T, U usf. bezeichnen Substitutionen.
- $\mathcal{V}(\sigma,\tau)$ ist der allgemeinste Unifikator von σ und τ.

Die Arbeitsweise des Algorithmus $\mathcal{W}$ benötigt an manchen Stellen „neue“ Typvariablen. Hier müssen Typvariablen eingesetzt werden, die vorher noch nicht verwendet worden sind. Zur Implementierung dieser Erzeugung von Typvariablen sei auf Kapitel 8 verwiesen.

$$
\begin{array}{lll}
\mathcal{W}((\mathcal{A}_x,\, x : \forall\alpha_i.\tau),\, x) & = & (\mathrm{I},\, \tau[\beta_i/\alpha_i]) \;\; \text{where} \\
 & & \qquad \beta_i \;\; = \;\; <\textit{neue Variable}> \\[1em]
\mathcal{W}(\mathcal{A},\, fx) & = & (UST,\, U\beta) \;\; \text{where} \\
 & & \qquad \begin{array}{lcl} (T,\tau) & = & \mathcal{W}(\mathcal{A},\, f) \\ (S,\sigma) & = & \mathcal{W}(T\mathcal{A},\, x) \\ \beta & = & <\textit{neue Variable}> \\ U & = & \mathcal{V}(S\tau,\, \sigma \to \beta) \end{array} \\[1em]
\mathcal{W}(\mathcal{A},\, \lambda x.e) & = & (T,\, T\beta \to \tau) \;\; \text{where} \\
 & & \qquad \begin{array}{lcl} (T,\tau) & = & \mathcal{W}(\mathcal{A}_x,\, x : \beta,\, e) \\ \beta & = & <\textit{neue Variable}> \end{array} \\[1em]
\mathcal{W}(\mathcal{A},\, (e_1,e_2,\, ...,\, e_n)) & = & (T_nT_{n-1}...T_1,\, (T_nT_{n-1}...T_2\tau_1,\, T_nT_{n-1}...T_3\tau_2,\, ...,\, T_n\tau_{n-1},\, \tau_n)) \;\; \text{where} \\
 & & \qquad \begin{array}{lcl} (T_1,\tau_1) & = & \mathcal{W}(\mathcal{A},\, e_1) \\ (T_2,\tau_2) & = & \mathcal{W}(T_1\mathcal{A},\, e_2) \\ ... & & \\ (T_n,\tau_n) & = & \mathcal{W}(T_{n-1}...T_1\mathcal{A},\, e_n) \end{array} \\[1em]
\mathcal{W}(\mathcal{A},\, [e_1,e_2,\, ...,\, e_n]) & = & (U_{12..n}T_n..U_{12}T_2T_1,\, [U_{12..n}\tau_n]) \;\; \text{where} \\
 & & \qquad \begin{array}{lcl} (T_1,\tau_1) & = & \mathcal{W}(\mathcal{A},\, e_1) \\ (T_2,\tau_2) & = & \mathcal{W}(T_1\mathcal{A},\, e_2) \\ U_{12} & = & \mathcal{V}(T_2\tau_1,\tau_2) \\ (T_3,\tau_3) & = & \mathcal{W}(U_{12}T_2T_1\mathcal{A},\, e_3) \\ U_{123} & = & \mathcal{V}(T_3U_{12}\tau_2,\tau_3) \\ ... & & \\ (T_n,\tau_n) & = & \mathcal{W}(U_{12..(n-1)}T_{n-1}..U_{12}T_2T_1\mathcal{A},\, e_n) \\ U_{12..n} & = & \mathcal{V}(T_nU_{12..(n-1)}\tau_{n-1},\tau_n) \end{array} \\[1em]
\mathcal{W}(\mathcal{A},\, e :: \tau) & = & (US,\, U\sigma) \;\; \text{where} \\
 & & \qquad \begin{array}{lcl} (S,\sigma) & = & \mathcal{W}(\mathcal{A},\, e) \\ U & = & \mathcal{V}(\sigma,\, \tau) \end{array} \\[1em]
\mathcal{W}(\mathcal{A},\, \text{let } x = e \text{ in } f) & = & (ST,\, \sigma) \;\; \text{where} \\
 & & \qquad \begin{array}{lcl} (T,\tau) & = & \mathcal{W}(\mathcal{A},\, e) \\ \tau' & = & \mathrm{Gen}(T\mathcal{A},\, \tau) \; (= \forall\alpha_i.\tau) \\ (S,\sigma) & = & \mathcal{W}(T\mathcal{A}_x,\, x : \tau',\, f) \end{array} \\[1em]
\mathcal{W}(\mathcal{A},\, \text{if } x \text{ then } t \text{ else } e) & = & (U'S'SUT,\, U'\sigma') \;\; \text{where} \\
 & & \qquad \begin{array}{lcl} (T,\tau) & = & \mathcal{W}(\mathcal{A},\, x) \\ U & = & \mathcal{V}(\tau,\, \mathrm{Bool}) \\ (S,\sigma) & = & \mathcal{W}(UT\mathcal{A},\, t) \\ (S',\sigma') & = & \mathcal{W}(SUT\mathcal{A},\, e) \\ U' & = & \mathcal{V}(S'\sigma,\, \sigma') \end{array}
\end{array}
$$

Abbildung 7.3: Typinferenzalgorithmus $\mathcal{W}$

Kapitel 8

Implementierung

8.1 Überblick

In diesem Abschnitt werden die verwendeten Entwicklungswerke und die eingesetzte Programmiersprache Java kurz beschrieben. Eine Auflistung der Packages gibt einen groben Überblick über die Implementierung. Das gesamte Kapitel bezieht sich auf die Version 1.0 der „Frisco F“-Implementierung.

8.1.1 Die Programmiersprache Java

Zur Implementierung des Parsers wurde die Programmiersprache Java ausgewählt. Java ist eine objektorientierte Programmiersprache, die im Gegensatz zu C++ keine Hybrid-Sprache ist. Java kennt außer Klassen und Interfaces keine weiteren globalen Sprachkonstrukte, so daß jeglicher Zugriff nur auf Klassen, Interfaces und lokal bekannte Elemente wie z.B. lokale Variablen und Instanzvariablen erfolgen kann. Mit Hilfe von Interfaces können Teilsignaturen definiert werden, die dann von Klassen implementiert werden. Auf Klassenebene wird Einfachvererbung und auf Interface-Ebene wird Mehrfachvererbung unterstützt.
Java wird ähnlich wie Smalltalk interpretiert, so daß der Quellcode in einen Zwischencode übersetzt wird. Existiert auf einer beliebigen Plattform ein geeigneter Interpreter, so können dort Java-Programme direkt ausgeführt werden. Hierzu werden Klassen und Interfaces als Dateien verwaltet, wobei auch mehrere Klassen und Interfaces in einer Datei enthalten sein können.
Java bietet nicht nur die Möglichkeit, Klassen auf Dateiebene zu verwalten. Es können auch sogenannte Packages gebildet werden, die eine Zusammenfassung von mehreren Dateien ermöglichen. Gewöhlich wird ein Package durch ein Verzeichnis realisiert, das diese Dateien enthält.

8.1.2 Scanner- und Parserunterstützung für Java

Um sowohl Scanner- als auch Parser-Generatoren verwenden zu können, wurde ein Programmpaket von verwendet, welches mit den Unix-Tools Flex und Bison zusammenarbeitet und im wesentlichen aus den Tools „jf“ und „jb“ besteht.

jf - Flex für Java

Der Scanner-Generator Flex ist eine verbesserte Variante des Unix-Tools Lex, welches zur Generierung von Scannern aus einer entsprechenden Definition eingesetzt wird. Da diese

Tools gewöhnlich dafür eingesetzt werden, Scanner in C zu generieren, sind sie für den Einsatz in Java nicht direkt geeignet. Mit Hilfe des Tools jf ist es nun möglich, den von Flex erzeugten Zwischencode in Java-Quellcode zu übersetzen. Dieser Quellcode kann dann anschließend mit einem Java-Compiler übersetzt werden.

jb - Bison für Java

Der Parser-Generator Bison ist eine verbesserte Variante des Unix-Tool Yacc. Ähnlich zu Flex wird Bison gewöhnlich zur Generierung von Parsern in C verwendet, so daß auch hier ein direkter Einsatz für Java nicht möglich ist. Das Tool jb ist nun in der Lage, den von Bison erzeugten Zwischencode in Java-Quellcode zu übersetzen, welcher dann ebenfalls anschließend mit einem Java-Compiler übersetzt werden kann.

8.1.3 Implementierte Java-Packages

Die Implementierung des „Frisco F“-Parsers stützt sich auf mehrere Packages, die im folgenden kurz beschrieben werden sollen. Die erste Auflistung gibt einen Überblick über allgemein verwendbare Packages, die für die bequeme Realisierung des Parsers implementiert wurden.

- Fehlerbehandlung (Exception Handling)
  Diese Gruppe enthält das Package *exceptions*, welches eng mit den Klassen aus den Logging-Packages zusammenarbeitet. Die wichtigste Klasse dieses Packages ist die Klasse *CascadedException*.

- Protokollierung (Logging)
  Diese Gruppe enthält die Packages *logBase*, *outputStreamLog*, *dialogLog* und *messagesDialog*. Sie stellen wichtige Varianten der Protokollierung bereit.

- Sonstiges
  Das Package *utility* stellt Klassen zur Verfügung, die oft benötigte Funktionen implementieren. Das Package *scc* ist eine Implementierung zur Berechnung der stark zusammenhängenden Komponenten (*strongly connected components*) eines Graphen. Das Package *testing* stellt ein kleines Test-Framework zur Verfügung.

Die folgende Auflistung zeigt die Packages der eigentlichen Parserimplementierung.

- Abstrakter Syntaxbaum
  Die Packages *astAtomic*, *astBase*, *astExp*, *astTopLevel*, *astType* und *bindings* enthalten zusammen die Klassen für den AST.

- Parsing
  Der Kern des Parsers wird aus den Packages *parser*, *parserEnvironment*, *symboltable* und *yy* gebildet.

- Typ- und Kind-Inferenz
  Die Packages *substitution* und *typeAndKindInference* sind für die Typ- und Kind-Inferenz zuständig.

8.2 Aufbau und Funktionsweise

In diesem Abschnitt wird zuerst der grobe Aufbau des „Frisco F“-Parsers beschrieben. Anschließend werden die Arbeitsschritte erläutert, die zum Parsen eines „Frisco F“-Programms notwendig sind. Auf die semantische Analyse wird dabei detaillierter eingegangen.

8.2.1 Die Struktur des „Frisco F“-Parsers

Der „Frisco F“-Parser wird durch die Klasse *FriscoFParser* realisiert. Mit ihr ist es möglich, „Frisco F“-Programme aus Java-Streams (also zum Beispiel aus Dateien oder direkt aus Strings) zu verarbeiten. Um den „Frisco F“-Parser über die Kommandozeile aufrufen zu können, wurde zusätzlich die Klasse *FriscoF* mit der Klassen-Methode `main()` implementiert. Die verfügbaren Kommandos werden im Anhang A beschrieben.

Des „Frisco F“-Parser besteht im wesentlichen aus drei Teilen. Zum einen enthält der „Frisco F“-Parser den von jb generierten Parser (Klasse *YYParse_FriscoF*) und den von jf generierten Scanner (Klasse *YYLex_FriscoF*). Zum anderen enthält der „Frisco F“-Parser ein Objekt der Klasse *ParserEnvironment*, welches für die Verwaltung des AST und der Symboltabelle sowie für die Durchführung der semantischen Analyse zuständig ist. Die Abbildung 8.1 zeigt die grobe Struktur des „Frisco F“-Parsers.

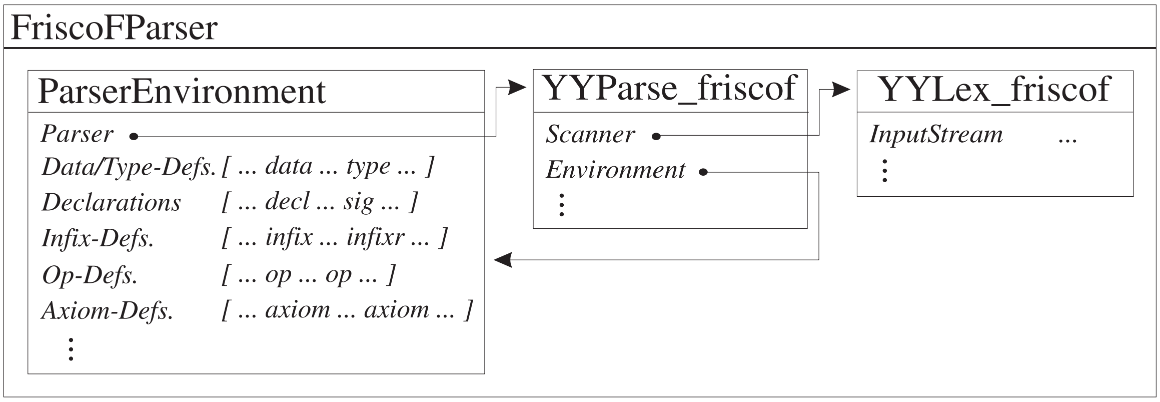


Abbildung 8.1: Grobstruktur des „Frisco F“-Parsers

Der AST setzt sich im wesentlichen aus den Komponenten zusammen, die in der Top-Level-Struktur des „Frisco F“-Programms auftreten. Die folgende Zusammenstellung gibt einen Überblick über diese Elemente.

- Data- und Type-Definitionen
- Funktions- und Pattern-Deklarationen inklusive expliziter Signatur-Deklarationen
- Infix-Definitionen
- Op-Definitionen (Spezifikations-Operatoren mit Signatur)
- Axiome (prädikatenlogische Ausdrücke)

Nach dem Parsen besteht der AST aus Listen dieser oben aufgeführten Elemente. Im Verlauf der semantischen Analyse wird der AST allerdings ergänzt und ggf. umstrukturiert. So werden beispielsweise die Funktions- und Pattern-Deklarationen, die in Form von Gleichungen vorliegen, in Bindungsgruppen umgewandelt. Der AST wird somit in der semantischen Analyse um eine Liste von Bindungsgruppen ergänzt, wobei die Liste der Gleichungen verworfen wird. Die semantische Analyse wird vollständig von der Klasse *ParserEnvironment* übernommen.

8.2.2 Die Phasen des „Frisco F"-Parsers

Bei der Verarbeitung eines „Frisco F"-Programms werden drei Phasen durchlaufen. Die Abbildung 8.2 gibt einen Überblick über die Phasen des „Frisco F"-Parsers.

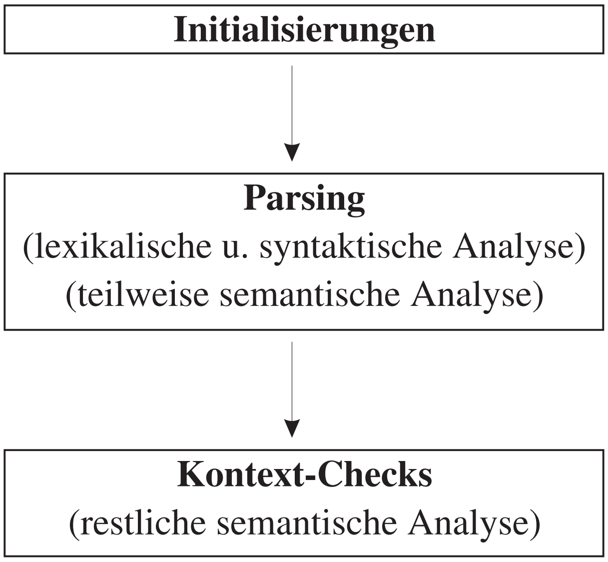


Abbildung 8.2: Hauptphasen des „Frisco F"-Parsers

Die semantische Analyse ist auf die letzten zwei Phasen verstreut. In der zweiten Phase werden diejenigen Checks durchgeführt, die sequentiell anwendbar sind. So kann beispielsweise schon während dem Parsen geprüft werden, ob ein Datentyp bereits definiert worden ist. In der dritten Phase werden die restlichen Checks durchgeführt. Diese benötigen im allgemeinen die Gesamtheit der Definitionen, um die Prüfungen durchführen zu können. So wird beispielsweise erst nach dem Parsen geprüft, ob die in einem Funktionsaufruf angesprochene Funktion auch tatsächlich definiert wurde.

8.2.3 Teilaufgaben der semantischen Analyse

Wie bereits erwähnt, wird ein Teil der semantischen Analyse schon während dem Parsen durchgeführt. Die Kontext-Checks werden in den semantischen Aktionen der Grammatik-Regeln aufgerufen und sind in Abschnitt 6.2 beschrieben. Die folgende Liste gibt eine kurze Zusammenfassung dieser Prüfungen:

- *Data- und Type-Definitionen.* Es wird überprüft, ob die Definitionen strukturell korrekt sind und die eingeführten Bezeichner nicht bereits definiert worden sind.
- *Infix- und Op-Definitionen.* Hier werden einfache strukturelle Prüfungen durchgeführt und sichergestellt, daß sich Infix- und Op-Definitionen gegenseitig ausschließen.
- *Signatur-Deklarationen.* Hier werden einfache strukturelle Prüfungen durchgeführt.

In der dritten Phase werden die restlichen Kontext-Checks durchgeführt. Die Methode `semanticAnalysis()` der Klasse *ParserEnvironment* übernimmt diese Aufgabe. Es werden folgende Schritte der Reihe nach ausgeführt.

1. `analyseTypes()`
   Hier werden die Kontext-Checks aus den Abschnitten 6.3.1, 6.3.2 und 6.3.3 durchgeführt. Sie stellen sicher, daß die Definitionen konsistent sind. Insbesondere wird eine Kind-Prüfung durchgeführt, und die Konstruktorfunktionen werden typisiert.

2. `Vector bindings = makeBindings(aDeclarations)`
   Die Kontext-Checks aus den Abschnitten 6.3.6 und 6.3.7 wandeln die Gleichungen in Bindungen um und überprüfen die expliziten Signatur-Deklarationen.

3. `infixOperatorsDefined()`
   Die Kontext-Checks aus Abschnitt 6.3.4 stellen sicher, daß die Infix-Operatoren definiert wurden.

4. `specificationOperatorsUndefined()`
   Die Kontext-Checks aus Abschnitt 6.3.5 stellen sicher, daß die Spezifikations-Operatoren nicht definiert wurden.

5. `aBindingSCCs = dependencyAnalysis(bindings)`
   Mit Hilfe der Kontext-Checks aus den Abschnitten 6.3.8 und 6.3.9 wird eine Abhängigkeitsanalyse durchgeführt, so daß die Bindungen insbesondere in Bindungsgruppen aufgeteilt werden können.

6. `checkAxioms()`
   Hier werden die Kontext-Checks aus Abschnitt 6.3.10 durchgeführt. Dadurch wird sichergestellt, daß die Axiome korrekt formuliert sind.

7. `typeDefinitionGroups(aBindingSCCs)`
   Hier werden die Definitionen typisiert. Implementierungsdetails sind in Abschnitt 8.4 zu finden.

8. `typeAxioms()`
   Hier werden die Axiome typisiert. Implementierungsdetails sind ebenfalls in Abschnitt 8.4 zu finden.

Die Regeln der semantischen Analyse sind gekennzeichnet (zum Beispiel Regel Data-5P). Im Quellcode sind die Methoden und Stellen, die zur Implementierung einer Regel notwendig sind, durch einen entsprechenden Kommentar gekennzeichnet. Die Stellen können durch Textsuche nach den Regelbezeichnern gefunden werden.

8.3 Implementierungsdetails

In diesem Abschnitt wird auf einige Details der Implementierung eingegangen. Dies betrifft im einzelnen den Scanner, den Parser (insbesondere den Tidy-Infix-Algorithmus), die Symboltabelle und den Algorithmus zur Berechnung der stark zusammenhängenden Komponenten eines gerichteten Graphen. Die Implementierung der Typ- und Kind-Inferenz wird in einem eigenen Abschnitt im Anschluß hieran ausführlich besprochen.

8.3.1 Scanner

Der Scanner wird durch die Klasse *YYLex_FriscoF* repräsentiert. Beim Erzeugen eines Scanner-Objekts können Eingabe- und Ausgabe-Strom übergeben werden. Als Default-Objekte werden `System.in` und `System.out` verwendet, falls eines der beiden Argumente weggelassen wird. Anschließend können durch den Aufruf der Methode `yylex()` die Terminale der Reihe nach abgeholt werden.
Die Verarbeitung von Kommentaren wird mit Hilfe von vier Zuständen und einem Zähler realisiert. Befindet sich der Scanner nicht in einem Kommentar (Zustand `INITIAL`), so wird beim Erkennen eines Kommentaranfangs in den Zustand `COMMENT` gewechselt und der Zähler mit 1 initialisiert. Befindet sich der Scanner im Zustand `COMMENT`, so wird beim Erkennen eines Kommentaranfanges der Zähler hochgezählt und beim Erkennen eines Kommentarendes der Zähler heruntergezählt. Erreicht der Zähler wieder den Wert 0, so wurde der äußerste Kommentar beendet und der Scanner wechselt in den Zustand `INITIAL`. Zum Erkennen von Kommentaranfang und -ende werden kurzzeitig die Hilfszustände `COMMENTK` und `COMMENTM` betreten. Zeilenkommentare werden durch zwei Scanner-Regeln verarbeitet.
Die Abbildung 8.3 zeigt die Schritte von der Scanner-Definition bis zur Erzeugung der Klasse *YYLex_FriscoF*.

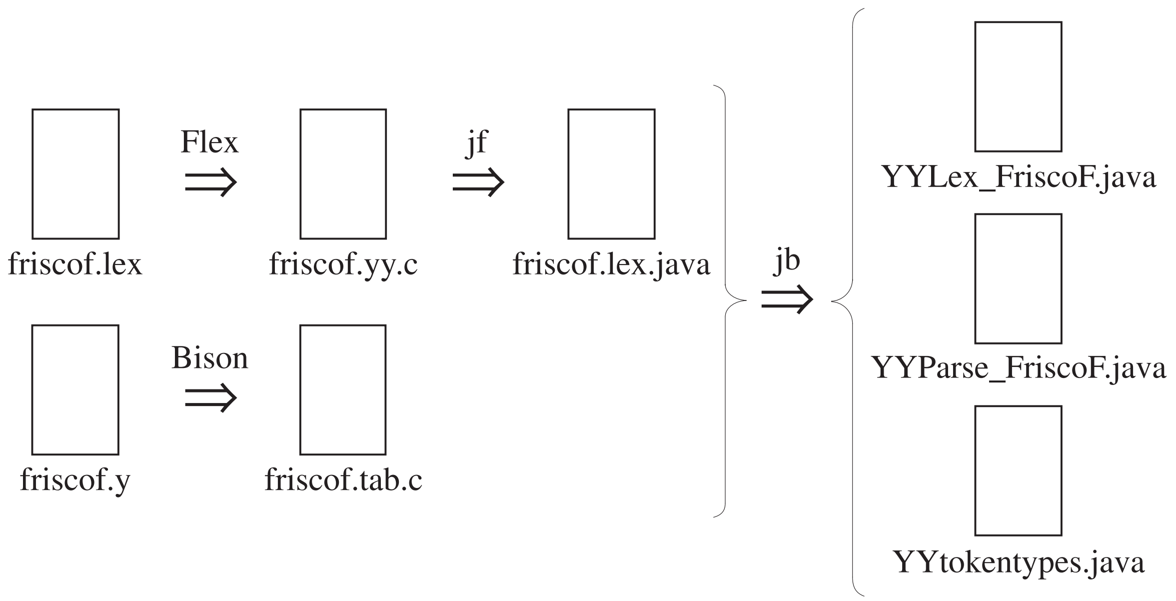


Abbildung 8.3: Die Erzeugung des Scanners und Parsers

Die Generierung wird durch ein Make-File geregelt. Siehe hierzu Anhang A.

8.3.2 Parser

Die Klasse *YYParse_FriscoF* repräsentiert den Parser. Zur Erzeugung eines Parser-Objektes muß ein Scanner-Objekt übergeben werden. Optional kann ein Ausgabe-Strom übergeben werden, wobei als Default `System.out` verwendet wird. Durch den Aufruf der Methode `yyparse()` wird ein Quellprogramm geparst, welches durch den Scanner zur Verfügung gestellt wird.
Der Parser hält eine Referenz auf das Parser-Environment, damit durch die semantischen Regeln der AST aufgebaut werden kann. Wie bereits erwähnt, werden hier einige Kontext-Checks durchgeführt. Insbesondere werden Infix-Ausdrücke mit dem sogenannten Tidy-Infix-Algorithmus bearbeitet. Dieser wird weiter unten beschrieben.
Die Abbildung 8.3 zeigt die Schritte von der Parser-Definition bis zur Erzeugung der Klasse *YYParse_FriscoF*. Die Klasse *YYtokentypes* wird von Scanner und Parser gemeinsam benutzt und wird ebenfalls automatisch generiert.

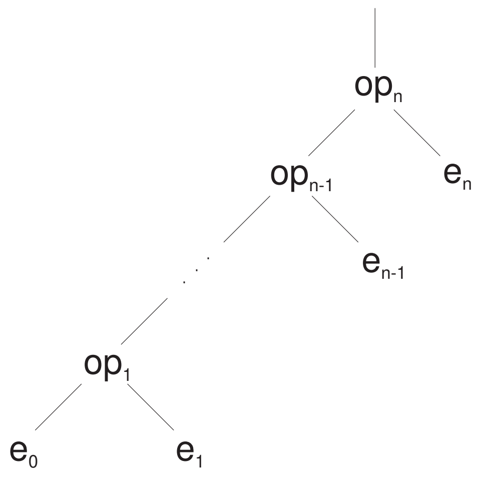
Tidy-Infix-Algorithmus

Die Realisierung von Infix-Operatoren mit fest vorgegebener Präzedenz und Assoziativität kann bereits in die Grammatik einfließen. So kann zum Beispiel die Grammatik von Ausdrücken mit den links-assoziativen Infix-Operatoren + und * mit „Punkt vor Strich“-Regel wie folgt definiert werden:

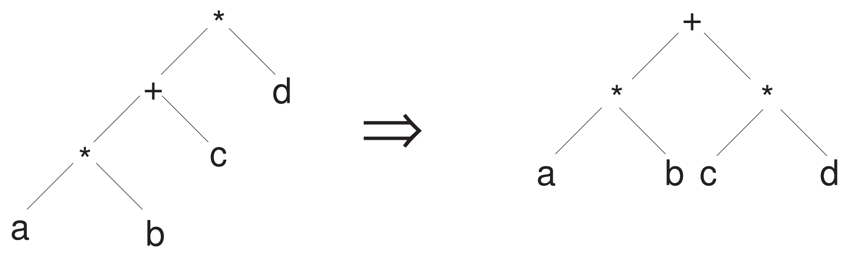
Expression	::=	Expression $\boxed{*}$ Term
	|	Term
Term	::=	Term $\boxed{+}$ Atomic
	|	Atomic
Atomic	::=	Variable
	|	Literal
	|	$\boxed{(}$ Expression $\boxed{)}$

Wenn allerdings die Präzedenz und Assoziativität von Infix-Operatoren nicht a priori definiert werden können oder sollen, so muß die korrekte Klammerung entweder explizit im Quellcode vorgenommen werden (wie zum Beispiel bei Smalltalk) oder der Ausdruck nach dem Parsen nochmals verarbeitet werden. Letzters wird bei „Frisco F“ angewandt, wobei Präzedenz und Assoziativität von Infix-Operatoren mit der Infix-Definition festgelegt werden.
Zu diesem Zweck werden Ausdrücke mit Infix-Operatoren „flach“ geparst. So liegt der Ausdruck
$e_0\ op_1\ e_1\ \ldots\ op_n\ e_n$

nach dem Parsen in folgender Form vor:



Dieser (entartete) Baum wird nun mit dem folgenden Algorithmus so umgestellt, daß die Präzedenzen und Assoziativitäten eingehalten werden. Beispielsweise wird der Ausdruck `a*b+c*d` folgendermaßen umgestellt:



Es ist auch möglich, daß dies nicht gelingt. In diesem Fall ist der Ausdruck mehrdeutig. Ist der Operator (/) nicht-assoziativ, so ist zum Beispiel der Ausdruck

$a/b/c$

mehrdeutig, da zwischen den Alternativen

$(a/b)/c$

und

$a/(b/c)$

nicht entschieden werden kann.
Der Algorithmus „Tidy-Infix" zum Umstellen von Infix-Ausdrücken ist ein einfacher Shift-Reduce-Parser und wurde größtenteils von der Gofer-Implementierung (Version 2.28 Januar 1993) übernommen. Der Algorithmus wird im folgenden in einer Pseudo-Sprache wiedergegeben. Der Stack s besteht aus verketteten Infix-Ausdrücken.

```
FUNCTION TidyInfix(e :: Infix-Ausdruck) :: Infix-Ausdruck
BEGIN
  s := []; -- leerer Stack
  WHILE e ist Infix-Ausdruck ∧ rechter Teilausdruck von e ≠ NIL DO
    IF s = [] THEN
      -- SHIFT 1
      -- e hat die Form ( ... ) op_i e_i
      s := push(s, top(e)); -- oberstes Element (op_i e_i) auf Stack bringen.
      e := left(e); -- e mit linkem Teilausdruck ( ... ) belegen.
    ELSE
      (Pe,Ae) := PA(op(e)); -- ermittle Präzedenz und Assoz. von e
      (Ps,As) := PA(op(top(s))); -- ermittle Präzedenz und Assoz. von top(s)

      IF Pe = Ps ∧ (Ae ≠ As ∨ Ae = nicht-assoziativ) THEN
        error "ambiguous use of infix-operator"
      FI;
      IF Pe > Ps ∨ (Pe = Ps ∧ Ae = links-assoziativ) THEN
        -- SHIFT 2
        -- e hat die Form ( ... ) op_i e_i
        s := push(s, top(e)); -- oberstes Element (op_i e_i) auf Stack bringen.
        e := left(e); -- e mit linkem Teilbaum ( ... ) belegen.
      ELSE
        -- REDUCE 1
        -- s hat die Form (op_t e_t):ss
        -- e hat die Form ( ... ) op_i e_i
        e := ( ... ) op_i (e_i op_t e_t) -- r. Teilausdr. mit ob. Stack-Elem. klammern
        s := pop(s); -- oberstes Element vom Stack nehmen, d.h. s := ss.
      FI;
    FI;
  OD;

  WHILE s ≠ NIL DO
    -- REDUCE 2
    -- s hat die Form (op_t e_t):ss
    -- e hat die Form ( ... ) op_i e_i
    e := (( ... ) op_i e_i) op_t e_t -- linken Teilausdruck klammern
    s := pop(s); -- oberstes Element vom Stack nehmen, d.h. s := ss.
  OD;
END
```

Als Beispiel soll der Algorithmus mit den Ausdrückem $a * b + c * d$ und $a + b * c + d$ vorgeführt werden. Es wird angenommen, daß + die Präzedenz 8 bzw. $*$ die Präzedenz 9 hat und beide Operatoren links-assoziativ sind.
Verarbeitung des Ausdrucks a*b+c*d:

Operation	**e**	**s**	**(Pe,Ae)**	**(Ps,As)**
	((a*b)+c)*d	[]		
SHIFT 1				
	(a*b)+c	[*d]		
REDUCE 1			+ (8,LEFT)	* (9,LEFT)
	(a*b)+(c*d)	[]		
SHIFT 1				
	a*b	[+(c*d)]		
SHIFT 2			* (9,LEFT)	* (9,LEFT)
	a	[*b, +(c*d)]		
REDUCE 2				
	a*b	[+(c*d)]		
REDUCE 2				
	(a*b)+(c*d)	[]		

Verarbeitung des Ausdrucks a+b*c+d:

Operation	**e**	**s**	**(Pe,Ae)**	**(Ps,As)**
	((a+b)*c)+d	[]		
SHIFT 1				
	(a+b)*c	[+d]		
SHIFT 2			* (9,LEFT)	+ (8,LEFT)
	a+b	[*c, +d]		
REDUCE 1			+ (8,LEFT)	* (9,LEFT)
	a+(b*c)	[+d]		
SHIFT 2			+ (8,LEFT)	+ (8,LEFT)
	a	[+(b*c),+d]		
REDUCE 2				
	a+(b*c)	[+d]		
REDUCE 2				
	(a+(b*c))+d	[]		

8.3.3 Symboltabelle

Ein Objekt der Klasse *SymbolTable* repräsentiert eine Symboltabelle. Die wichtigsten Methoden betreffen das Einfügen und das Suchen von Funktions- und Pattern-Bindungen, Typkonstruktoren, Typsynonymen und Konstruktorfunktionen.
Während der Abhängigkeitsanalyse (*dependency analysis*) von Funktions- und Pattern-Bindungen müssen auch lokale Definitionen verarbeitet werden. Hierzu verwaltet die Symboltabelle einen Stack von Sichtbarkeitsbereichen, wobei der globale Sichtbarkeitsbereich immer vorhanden ist, in dem auch die Top-Level-Definitionen eingefügt werden.

Die Symboltabelle stellt Funktionen zur Verfügung, um Bezeicher injektiv auf Ganzzahlen abbilden zu können. Dies sind die Methoden `nameToID()` und `IDToName()`. Damit kann eine effizientere Verarbeitung von Bezeichnern durch Ganzzahlen eingesetzt werden. Die Nummern werden in einem Objekt der Klasse *Hashtable* verwaltet. Diese Klasse ist im Java-Standard-Package *java.util* enthalten und verwendet Hashing zur Speicherung der Bezeichner. Da Typkonstruktoren und Konstruktorfunktionen die gleichen Namen haben können, reicht der Name als Schlüssel in die Symboltabelle nicht aus. Daher wird ein kombinierter Schlüssel verwendet, in dem der Typ des Eintrags und die ID des Bezeichners eingeht.
Die Sichtbarkeitsbereiche sind durch Objekte der Klasse *Scope* realisiert. Diese verwenden ebenfalls eine Hashtabelle, um die Einträge effizient verwalten zu können. Zudem können in jedem Sichtbarkeitsbereich gebundene Variablen und eine Liste von abhängigen Definitionen verwaltet werden. Diese Elemente werden speziell für die Abhängigkeitsanalyse benötigt.

8.3.4 Berechnung stark zusammenhängender Komponenten

Data- und Type-Definitionen sowie Funktions- und Pattern-Bindingen können verschränkt-rekursiv sein. Für die Analyse dieser Definitionen ist es allerdings notwendig, so wenig Definitionen wie möglich als Gruppe zu verarbeiten, da zum Beispiel das Ergebnis der Typinferenz sonst zu speziell sein kann. Würden beispielsweise die Definitionen

```
id x   =   x;
v      =   id 1;
```

zusammen typisiert werden, so ergäbe sich für `id` der Typ `Int -> Int` und nicht `a -> a`!
Betrachtet man die Definitionen als Knoten in einem gerichteten Graphen, dann sind die adjazenten Knoten der Definiton d genau die Definitionen, auf die sich d abstützt. Auf diese Weise kann der Graph in stark zusammenhängende Komponenten partitioniert werden.
Für Data- und Type-Definitionen entsprechen den adjazenten Knoten die verwendeten Typkonstruktoren und Typsynonyme auf der rechten Seite der Definition. Für Funktions- und Pattern-Bindungen sind die adjazenten Knoten genau die verwendeten Operatoren, Variablen und insbesondere Funktionsnamen.
Der verwendete Algorithmus zur Ermittlung der stark zusammenhängenden Komponenten geht auf den in [Aho72] beschriebenen Algorithmus zurück.

```
FUNCTION SCC(nodes :: [Node]) :: [[Node]]

  FUNCTION LowLink(node :: Node) :: Int
  BEGIN
    low := count;
    dfn := count;
    IF number(node) > 0 THEN adj := [] ELSE adj := adjacentNodes(node) FI;
    number(node) := count;
    stack := push(stack, node);

    count := count + 1;

    FOR n IN adj DO
      IF number(n) > -1 THEN x := number(n) ELSE x := LowLink(n) FI;
```

```
      IF low > x ∧ x ≠ 0 THEN low := x FI;
    OD;

    IF low = dfn THEN
      SCC := [];
      REPEAT
        number(top(stack)) := 0;
        SCC := top(stack):SCC;
      UNTIL node = pop(stack);
      SCCs := SCC:SCCs;
    FI;

    RETURN low;
  END

BEGIN
  count := 1;
  stack := [];
  SCCs := [];
  FOR node IN nodes DO
    number(node) := -1;
  OD;

  FOR node IN nodes DO
    IF number(node) = -1 THEN LowLink(node); FI;
  OD;

  RETURN SCCs;
END
```

Beschreibung des Algorithmus

Für jeden Knoten n kann $number(n)$ die Werte -1, 0 und größer 0 annehmen.

- -1: Knoten n wurde noch nicht besucht.
- 0: Knoten n ist bereits Bestandteil einer Komponente.
- >0: Koten n ist im aktuellen Pfad enthalten.

In der Hauptschleife werden die Variablen count, stack und SCCs und number(n) aller Knoten initialisiert. Anschließend wird über alle Knoten iteriert und für alle bis dahin noch nicht besuchten Knoten LowLink aufgerufen.

Mittels LowLink wird der Graph im Depth-First-Prinzip durchwandert. Alle besuchten Knoten werden in einem Stack gesammelt und der Reihe nach numeriert. Enthält der Pfad keinen Zyklus, so gelangt man an einen Knoten, für den die Menge der adjazenten Knoten leer ist und daher die Bedingung low = dfn gilt. Dieser Knoten wird dann zu einer Komponente und bekommt die Nummer 0 zugewiesen. Trifft man hingegen auf einen adjazenten Knoten, dem bereits die Nummer 0 zugewiesen wurde, so wird dieser ignoriert. Solche Knoten wurden bereits in früheren Durchgängen in Komponenten zusammenfaßt und müssen übergangen werden. Anderfalls ermittelt man die kleinste Nummer, die durch den Aufruf von LowLink aller adjazenten Knoten zurückgegeben werden. Wurde kein Knoten mit einer kleineren Nummer gefunden als die des aktuellen Knotens, so bildet der aktuelle Knoten mit allen Knoten höherer Nummer eine Komponente. Da die Knoten in aufsteigender Reihenfolge (bezüglich der Nummer) auf dem Stack abgelegt wurden, muß der Stack bis zum aktuellen Knoten geleert werden, wobei diese Elemente zu einer Komponente zusammenfaßt werden. Zudem bekommen die ermittelten Knoten die Nummer 0 zugewiesen. In der Funktion LowLink wird also für jeden besuchten Knoten $number(n)$ auf einen Wert $\neq -1$ gesetzt. Da jeder Aufruf von LowLink an die Bedingung $number(n) = -1$ geknüpft ist, wird die Funktion für jeden Knoten nur einmal aufgerufen. Der Algorithmus terminiert, wenn jeder Knoten (aufgrund der Hauptschleife) mindestens einmal besucht wurde.

8.4 Implementierung der Typ- und Kind-Inferenz

Die Hauptaufgabe der Typprüfung ist es sicherzustellen, daß jeder Audruck und jede Definition einen Typ besitzt. Kann ein Ausdruck oder eine Definition nicht getypt werden, so wird dies als Fehler betrachtet. Zudem wird sichergestellt, daß ggf. explizit angegebene Signatur-Deklarationen mit den hergeleiteten Typen konsistent sind.
Die Implementierung der Typinferenz stützt sich auf die Implementierung von Gofer Version 2.28 Januar 1993. Aus diesem Grund ist auch dieser Abschnitt stark an [Jon94a] angelehnt (vgl. Kapitel 7).
Die Typinferenz wird durch die Klasse *TypeInference* realisiert. Ein Objekt dieser Klasse enthält die aktuelle Subsitution und die Menge der Typannahmen. Beides wird im folgenden beschrieben.

8.4.1 Aktuelle Substitution

Der Typinferenz-Algorithmus $\mathcal{W}$ berechnet neben dem Typ auch sukzessive die notwendige Substitution. Dies geschieht funktional. Die vorliegende Implementierung hingegen verwendet eine einzige Substitution, die sukzessiv modifiziert wird. Diese wird als die aktuelle Substitution bezeichnet.
Typausdrücke werden durch sogenannte Typskelette repräsentiert. Ist der Typ polymorph, so kommen in ihm Typvariablen vor. Diese werden konsistent durch Variablen-Offsets $O_0, O_1, ...$ ersetzt. Zum Beispiel hat die Funktion id den Typ $\forall\alpha.\alpha \rightarrow \alpha$. Das zugehörige Typskelett ist:

$$POLY_1\ S_{\rightarrow}(O_0, O_0)$$

$POLY_i$ besagt, daß das Typskelett polymorph mit i Typvariablen ist. Die Funktion map hat den Typ $\forall\alpha\forall\beta.(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$ mit dem zugehörigen Typskelett

$$POLY_2\ S_{\rightarrow}(S_{\rightarrow}(O_0, O_1), S_{\rightarrow}(S_{[]}(O_0), S_{[]}(O_1)))$$

Wird die Funktion id auf ein Argument vom Typ `Int` angewandt, so wird jedes O_0 durch das Typskelett S_{INT} von `Int` ersetzt. Dadurch entsteht das Typskelett

$$S_{\rightarrow}(S_{INT}, S_{INT}).$$

Es ist zu beachten, daß $S_{\rightarrow}$ und $S_{[]}$ analog zu Typkonstruktoren aus Typskeletten wieder Typskelette konstruieren. Anstatt bei jeder Instanzierung das Typskelett zu kopieren, wird hier jedoch ein anderer Weg beschritten: monomorphe Typen werden als Kombination von

Typskeletten und Offset-Werten beschrieben. Die erste Komponente, das Typskelett, repräsentiert die Struktur des Typs (Typschablone), ohne zu bestimmen, an welche Typen die Typvariablen im Skelett tatsächlich gebunden sind. So kann das Skelett

$$S_{\rightarrow}(O_0, O_0)$$

für alle Instanzierungen der Funktion id verwendet werden. (Hier fehlt das $POLY_1$, da eine Typ-Instanzierung monomorph ist.)
Die zweite Komponente ist ein Offset-Wert in ein Feld von Typskelettvariablen. Hier wird der Begriff Typskelettvariable verwendet, damit sie nicht mit den Typvariablen eines Typs verwechselt werden. Jede Typskelettvariable hat die Komponenten *bound* und $offset$. Diese Komponenten beschreiben ein anderes Paar von Typskelett und Offset-Wert. Ist die Komponente *bound* hingegen nicht belegt ($NULL$), so ist die Typskelettvariable ungebunden.
Das Feld der Typskelettvariablen stellt gerade die aktuelle Substitution dar, da es als Abbildung von Typskelettvariablen zu Typausdrücken angesehen werden kann.
Soll nun ein polymorphes Typskelett instanziert werden, so muß ein Paar, bestehend aus Typskelett und Offset-Wert, erzeugt werden. Das Typskelett stammt direkt aus dem polymorphen Typskelett. Die Typskelettvariablen müssen neu angelegt werden, so daß die aktuelle Substitution um die benötigte Anzahl von Typskelettvariablen erweitert wird. Die neuen Typskelettvariablen werden als ungebunden gekennzeichnet. Der Offset-Wert ist nun der Offset der ersten neu angelegten Typskelettvariable in der aktuellen Substitution. Es ist zu beachten, daß dadurch die Typskelettvariablen eines instanzierten Typs immer hintereinander liegen. Beispielsweise könnte die Instanzierung des Skeletts

$$POLY_2\ S_{\rightarrow}(S_{\rightarrow}(O_0, O_1), S_{\rightarrow}(S_{[]}(O_0), S_{[]}(O_1)))$$

das Paar

$$(S_{\rightarrow}(S_{\rightarrow}(O_0, O_1), S_{\rightarrow}(S_{[]}(O_0), S_{[]}(O_1))), 17)$$

ergeben, falls der Offset-Wert der ersten neu angelegten Typskelettvariable 17 ist. Der Zugriff auf Typskelettvariablen erfolgt über deren Offset in der aktuellen Substitution. Zum Beispiel entspricht der Zugriff auf die Typvariable b im Typ der Funktion *map* dem Zugriff auf den Variablen-Offset O_1. Bei gegebenem Offset-Wert W entspricht dies der Typskelettvariablen mit Offset $W + 1$.
Noch einmal zur Verdeutlichung: Der Offset-Wert ist abhängig von der Instanzierung eines Typskeletts und repräsentiert zusammen mit dem Typskelett einen monomorphen Typ. Ein Variablen-Offset ist nur von der Struktur des Typskeletts abhängig und regelt den Zugriff auf Typvariablen innerhalb einer gegebenen Instanzierung. Der Offset bestimmt die Position einer Typskelettvariable innerhalb der aktuellen Substitution.
Bei der Instanzierung eines monomorphen Typs müssen keine Typskelettvariablen angelegt werden. Somit besteht das Paar aus dem Typskelett selbst und einem beliebigen Offset-Wert. Dieser hat keine Bedeutung, da kein Zugriff auf Typvariablen (in einem monomorphen Typ) erfolgen kann.

8.4.2 Typannahmen

Die Typannahmen werden als Liste von Paaren repräsentiert, in denen einem Bezeichner ein Typ zugewiesen wird. Es ist jedoch notwendig, die Liste in zwei Teillisten zu zerlegen:

- *λ-gebundene Variablen*, wie zum Beispiel Funktionsparameter oder Parameter in λ-Ausdrücken, können nur einen monomorphen Typ besitzen. Beispielsweise wird die folgende Definition durch die Typprüfung zurückgewiesen:

  ```
  fkt i = (i True, i 'a');
  ```

  Die λ-gebundene Variable `i` auf der rechten Seite wird mit zwei Werten unterschiedlichen Typs angewandt. Für die linke Anwendung ergibt sich der Typ `Bool` $\to \alpha$ und für die rechte Anwendung der Typ `Char` $\to \alpha$, so daß die Unifikation fehlschlägt.

- *Let-gebundenen Variablen*, die zum Beispiel in einer lokalen Definition oder auf der Top-Level-Ebene definiert wurden, können polymorphe Typen zugewiesen werden. Zum Beispiel hat die Variable `id` in der Definition

  ```
  let id x = x in (id True, id 'a');
  ```

  den polymorphen Typ $\forall\alpha.\alpha \to \alpha$. Daher kann die Variable `id` im Rumpf für zwei Werte unterschiedlichen Typs angewandt werden, da bei jeder Anwendung von `id` der polymorphe Typ von `id` instanziert wird.

Die zwei Arten von Variablen können nebeneinander benutzt werden. So sind beispielsweise in der Definition

```
member x xs = any isx xs where isx y = x == y;
```

die Variablen `member` und `isx` let-gebunden und die Variablen `x`, `xs` und `y` λ-gebunden.
Aus diesem Grund werden die Typannahmen durch zwei Listen repräsentiert, eine für let- und eine für λ-gebundene Variablen. Um verschachtelte Definitionen handhaben zu können, werden anstatt der einfachen Listen zwei Stacks von Listen verwendet. Beim Eintritt in eine lokale Definition werden zwei neue Listen auf die Stacks gebracht und nach dem Verlassen wieder entfernt. Auf diese Weise lassen sich auch Verschattungen von Bezeichnern einfach realisieren. Die Stacks würden beispielsweise nach dem Typcheck des Ausdrucks `x == y` folgende Struktur aufweisen (mit gewissen Typen it, mt, yt, xst und xt):

$$\begin{array}{lcllll}
\text{let-gebunden} & = & [(\texttt{isx}, it)] & : & [(\texttt{member}, mt)] \\
\lambda\text{-gebunden} & = & [(\texttt{y}, yt] & : & [(\texttt{xs}, xst), (\texttt{x}, xt)]
\end{array}$$

8.4.3 Generalisierung

Die inverse Operation zur Instanzierung ist die Generalisierung, um aus einem monomorphen Typ den allgemeinsten Typ abzuleiten, der möglich ist. Die Generalisierung eines Typs τ mit einer Menge von Typannahmen $\mathcal{A}$ wird mit Gen(τ,$\mathcal{A}$) bezeichnet. Die Berechnung wird in zwei Schritten durchgeführt. Im ersten Schritt werden alle „festen“ Typskelettvariablen, also zum Beispiel alle diejenigen, die in der aktuellen Menge von Typannahmen vorkommen, als fest markiert. Alle anderen werden als unbenützte generische Variablen markiert. Im zweiten Schritt wird das zu generalisierende Typskelett kopiert, wobei Typskelettvariablen abhängig von ihrer Markierung ersetzt werden. Jede feste Typskelettvariable wird durch einen absoluten Variablen-Offset ersetzt, der dem Offset der Typskelettvariable entspricht. Solche absoluten Variablen-Offsets verweisen unabhängig von nachfolgenden Instanzierungen immer auf diese feste Typskelettvariable. Für unbenützt generisch markierte Typskelettvariablen wird ein neuer Offset (beginnend mit 0) erzeugt. Die Typskelettvariablen werden mit diesem Offset als generisch markiert. Anstatt der Typskelettvariablen wird ein Variablen-Offset mit diesem Offset verwendet. Für als generisch markierte Typskelettvariablen wird der Variablen-Offset mit der Markierung verwendet. Somit werden die nicht festen Typskelettvariablen der Reihe nach aufgespürt und durch die Variablen-Offsets O_0, O_1, ... ersetzt. Wurde keine generische Variable erzeugt, so ist die Generalisierung monomorph. Ansonsten wird das kopierte Typskelett mit $POLY_i$ gekennzeichnet, wobei i die Anzahl der gefundenen generischen Typvariablen ist.

8.4.4 Grundlegende Typprüfungen

Das Herzstück der Typinferenz ist die Methode zum Typisieren von Ausdrücken. Der hergeleitete Typ ist ein Paar, bestehend aus Typskelett und Offset-Wert, und wird in den Instanzvariablen aTypeIs und aTypeOffset abgelegt. Die eigentliche Typisierung eines Ausdrucks wird dabei an diesen deligiert. Das Objekt der Klasse *TypeInference* stellt lediglich die Methoden zur Verfügung, bestimmte Typen abzuleiten und die aktuelle Substitution bzw. die Menge der Typannahmen modifizieren zu können.
Der einfachste Fall ist die Typisierung eines Literals. Zum Beispiel wird ein Objekt der Klasse *AtomicInteger*, welches ein Integer-Literal repräsentiert, die Methode `inferInteger()` aufrufen. Damit wird das Typskelett S_{INT} mit dem (irrelevanten) Offset 0 in aTypeIs und aTypOffset abgelegt.
Die meisten anderen Fälle sind natürlich nicht so einfach. Zum Beispiel wird zur Typisierung eines `if`-Ausdrucks der folgende Code ausgeführt:

```
int beta = ti.newTyvar();
ti.check(aCondition,ti.boolSkeleton(),0);
ti.check(aThen,var,beta);
ti.check(aElse,var,beta);
ti.inferVariableType(beta);
```

Die Variable `ti` ist ein Objekt der Klasse *TypeInference* und wird an die Methoden zur Typisierung übergeben. Im Beispiel wird zuerst eine neue Typvariable belegt, um den Typ des `if`-Ausdrucks aufnehmen zu können, der mit den Typen des then- und else-Zweigs übereinstimmt. Die Methode `check()` berechnet den Typ des ersten Arguments und versucht diesen, mit dem übergebenen Typ im zweiten und dritten Argument zu unifizieren. Falls die Unifikation fehlschlägt, wird ein Fehler ausgegeben. Die drei Aufrufe von `check` stellen sicher, daß die Bedingung ein Boolescher Wert ist, und daß then- und else-Zweig den gleichen Typ haben, indem die beiden Typen in der Typskelettvariable `beta` unifiziert werden. Die Variable `var` ist das Typskelett O_0, damit das Paar `(var,beta)` den Typ repräsentiert, der an die Typskelettvariable `beta` gebunden ist. Der Aufruf von `inferVariableType(beta)` belegt nun aTypeIs und aTypeOffset mit dem Typ, der an die Typskelettvariable `beta` gebunden worden ist. Dies entspricht genau dem Typ des `if`-Ausdrucks.
Als letztes Beispiel sei noch die Typisierung eines Let-Ausdrucks beschrieben. Hierfür wird der folgende Code ausgefüht:

```
ti.typeBindingGroups(SCCs);
aExpression.inferType(ti);
ti.leaveBindings();
```

Durch den Aufruf von `ti.typeBindingGroups(SCCs)` wird ein neuer Sichtbarkeitsbereich betreten und die Bindungsgruppen `SCCs` typisiert (siehe nächster Abschnitt). Hierdurch werden die typisierten Definitionen in die Menge der Typannahmen aufgenommen. Die Bindungsgruppen wurden in der Abhängigkeitsanalyse erzeugt. Anschließend wird der Ausdruck typisiert, wobei der hergeleitete Typ nun in den Variablen aTypeIs und aTypeOffset steht. Zum Schluß wird der Sichtbarkeitsbereich durch den Aufruf von `ti.leaveBindings()` wieder verlassen.

8.4.5 Typisierung von Bindungsgruppen

Um eine Bindungsgruppe zu typisieren, muß zuerst untersucht werden, welche Methode zur Typisierung angewendet werden soll. Dabei sind drei Fälle zu unterscheiden:

- Die Gruppe enthält eine Pattern-Deklaration oder eine Variablendeklaration (d.h. eine Deklaration, die auf der linken Seite nur eine Variable enthält), aber keine expliziten Signatur-Deklarationen. In diesem Fall kommt die Monomorphie-Restriktion zum Einsatz (vgl. [Jon94b] Abschnitt 14.4.6, [Thi94] S. 57). Zur Typisierung wird die Methode `noOverloading()` verwendet.
- Enthält die Gruppe keine Pattern-Deklarationen, keine Variablendeklarationen und keine expliziten Signatur-Deklarationen, so wird mit der Methode `implicitTyping()` der allgemeinste Typ hergeleitet, der möglich ist.
- Im verbleibenden Fall enthält die Gruppe keine Pattern-Deklarationen, aber dafür explizite Signatur-Deklarationen und ggf. Variablendeklarationen. In diesem Fall wird die Methode `explicitTyping()` verwendet. Sie leitet die Typen der Funktionen her und stellt sicher, daß diese mit den expliziten Signaturen in Einklang stehen.

Im folgenden werden wir den zweiten Fall diskutieren. Der erste Fall ist etwas leichter, der dritte Fall etwas schwerer, da die benutzerdefinierten Signaturen überprüft werden müssen. Die zugrunde liegenden Prinzipien sind aber immer dieselben.
Betrachten wir also die folgende Bindungsgruppe mit verschränkt rekursiven Bindungen ohne explizite Signatur-Deklarationen.

$$\begin{array}{lcl} f_1\ args_1 & = & e_1\,; \\ \ldots & & \\ f_n\ args_n & = & e_n\,; \end{array}$$

Zur Berechnung der Typen werden die folgenden Schritte durchgeführt:

- Für jede Funktion f_i wird eine neue Typskelettvariable β_i angelegt, die den Funktions-Typ der Funktion repräsentiert. Die Typskelettvariablen werden zusammen mit den Funktionsnamen als Let-gebundene Variablen in die Typannahmen aufgenommen.
- Für jede Bindung der Gruppe wird ein Typcheck durchgeführt. Für eine Gleichung f_i $args_i = e_i$ wird der Typ auf der linken und rechten Seite berechnet und anschließend unifiziert, um sicherzustellen, daß sie denselben Typ darstellen.
- Die durch die Typskelettvariablen β_i gegebenen Typen werden generalisiert, um die korrekten Typen der Funktionen f_i zu erhalten.

8.4.6 Kind-Inferenz

So wie Typen dazu benutzt werden, Werte zu klassifizieren, werden Kinds zur Klassifikation von Typkonstruktoren eingesetzt. Ohne die Komplexität der Polymorphie und Vielfältigkeit der Ausdrücke kann die Kind-Inferenz als eine stark vereinfachte Variante der Typinferenz angesehen werden. Sie wird ausschließlich zur Berechnung passender Kinds für Typkonstruktoren eingesetzt, um sicherzustellen, daß nur wohlgeformte Typausdrücke verwendet werden. Zur Kind-Inferenz wird ebenfalls eine aktuelle Substitution benötigt, und die Unifikation bezieht sich in diesem Fall auf Kind-Ausdrücke. Kind-Ausdrücke haben eine relativ einfache Struktur. Sie bestehen entweder aus dem Basis-Kind $*$, aus Kind-Variablen (ähnlich den Typskelettvariablen) oder der Applikation $\kappa_1 \rightarrow\ \kappa_2$ zweier Kinds. Zum Beispiel haben alle 0-stelligen Typkonstruktoren so wie `Int` oder `Float` den Kind $*$. Der 1-stellige Typkonstruktor für Listen hat den Kind $* \rightarrow\ *$ und der 2-stellige Typkonstruktor für Funktionen hat den Kind $* \rightarrow\ * \rightarrow\ *$. Kind-Variablen treten nur während der Kind-Inferenz auf und können an beliebige Kind-Ausdrücke gebunden werden. Mit Hilfe der Kind-Inferenz kann zum Beispiel der Ausdruck `[Tree]` zurückgewiesen werden. Da der Typkonstruktor für Listen den Kind $* \rightarrow\ *$ hat, wird zwischen den Klammern ein Typ vom Kind $*$ erwartet. Die Kind-Unifikation schlägt jedoch fehl, da der Kind $*$ nicht mit dem Kind $* \rightarrow\ *$ von Tree unifiziert werden kann. Hingegen ist der Typausdruck `[Tree Int]` wohlgeformt, da der Typausdruck `Tree Int` den Kind $*$ hat.

8.4.7 Typisierung von prädikatenlogischen Ausdrücken

Die Typisierung von prädikatenlogischen Ausdrücken unterscheidet sich von der Typisierung gewöhnlicher Ausdrücke nur unwesentlich. Es wird allerdings ein eigener nicht sichtbarer Typkonstruktor $\mathcal{B}$ verwendet, um den Typ solcher Ausdrücke von Booleschen Ausdrücken unterscheiden zu können. Der Typ $\mathcal{B}$ besteht aus den Werten TT und FF wohingegen der Typ `Bool` aus den Werten $True$, $False$ und $\perp$ besteht. Formal existieren zwei Funktionen, um Werte von $\mathcal{B}$ in Werte von `Bool` abzubilden und umgekehrt. Diese Funktionen sind wie folgt definiert:

$$
\begin{array}{lcl}
\iota & :: & \mathcal{B} \rightarrow \texttt{Bool}; \\
\iota(TT) & := & \texttt{True}; \\
\iota(FF) & := & \texttt{False}; \\
 & & \\
[[\ ]] & :: & \texttt{Bool} \rightarrow \mathcal{B}; \\
[[\texttt{True}]] & := & TT; \\
[[\texttt{False}]] & := & FF; \\
[[\perp]] & := & FF;
\end{array}
$$

Prädikatenlogische Ausdrücke treten nur in Axiomen und in `!`-Ausdrücken auf. Die Typisierung des `!`-Ausdrucks ist relativ einfach:

```
ti.checkPLExpressionType(aExpression);
ti.inferBool();
```

Zuerst wird durch den Aufruf von `ti.checkPLExpressionType(aExpression)` überprüft, ob der Ausdruck ein korrekter prädikatenlogischer Ausdruck ist. Ist das der Fall, dann ist gemäß der Einbettung ι der Typ dieses Ausdrucks der Typ `Bool`. Dies geschieht durch den Aufruf

von `ti.inferBool()`.
Die Typisierung der Quantoren $\forall$, $\exists$, $\forall^{\perp}$ und $\exists^{\perp}$ wirft, technisch gesehen, keine neuen Probleme auf, da die quantisierten Variablen lediglich als λ-gebundene Variablen eingeführt werden müssen und die angegebenen Typen als explizite Signaturen behandelt werden. Die Typisierung der Quantoren $\forall^{P}$ und $\exists^{P}$ ist ein wenig komplizierter, da die Typen der zu quantisierenden Variablen durch Pattern-Matching mit einem gegebenen Ausdruck ermittelt werden müssen.
Da insbesondere auch alle gewöhnlichen Ausdrücke als prädikatenlogische Ausdrücke verwendet werden können, muß hier die Abbildung $[\![\ ]\!]$ vorgenommen werden. Aus diesem Grund ist es ausreichend, den Ausdruck als vom Typ `Bool` zu erkennen:

```
inferType(ti);
ti.shouldBe(ti.boolSkeleton(),0);
```

Durch den Aufruf von `inferType(ti)` wird der Ausdruck typisiert und anschließend durch den Aufruf von `ti.shouldBe(ti.boolSkeleton(),0)` mit dem Typ `Bool` unifiziert.

Kapitel 9

Zusammenfassung und Ausblick

Im Rahmen dieses technischen Berichts ist die Spezifikationssprache „Frisco F" definiert worden. Der erste Teil des Berichts definiert die Sprache „Frisco F", wobei die Programmiersprache Gofer als Vorlage gedient hat. Der zweite Teil beschreibt die Implementierung. Zur Realisierung der lexikalischen und syntaktischen Analyse wurden die Werkzeuge Flex und Bison eingesetzt. Dies schließt die kontextabhängige Analyse, insbesondere die Typinferenz, mit ein.

Die folgenden Ziele wurden mit der Arbeit erreicht und in diesem Bericht dokumentiert:

- Definition der Sprache „Frisco F" mit Gofer als Vorbild
- Vereinfachung des Typsystems durch Weglassen von Typklassen
- Strikte Semantik und keine unendlichen Datenstrukturen
- Keine Layout-Rule wie in Gofer
- Definition von Axiomen durch prädikatenlogische Ausdrücke
- Implementierung der Synthese-Phase einschließlich der Typinferenz

Als große Hilfe erwiesen sich die Arbeiten von Mark Philip Jones und der Quellcode der Gofer-Implementierung. Ohne diese Grundlage wäre die vorliegende Implementierung in diesem Umfang nicht zustande gekommen. Lobend sei auch das jb/jf-Programmpaket erwähnt, das den problemlosen Einsatz von Flex und Bison für Java ermöglicht hat.

Der Ausgangspunkt der Entwicklung der Sprache „Frisco F" war die mangelnde Verfügbarkeit einer geeigneten textuellen Sprache als Ergänzung zu graphischen Beschreibungstechniken. Die in diesem Bericht beschriebene Arbeit kann als Grundlage weiterer Entwicklungen verwendet werden. So ist es insbesondere denkbar, den Parser durch Analyse- und weiterverarbeitende Werkzeuge zu ergänzen und in eine graphische Entwicklungsumgebung zu integrieren.

Anhang A

Technische Daten und Bedienhinweise

A.1 Installation

Folgende Komponenten werden benötigt:

- Das jb/jf-Programmpaket ist zur Entwicklung, aber auch zur Laufzeit notwendig. In Abschnitt A.3 ist beschrieben, wie das Paket besorgt werden kann. Zur Installation sei auf die beliegenden Dokumente verwiesen.

- Ebenfalls zur Entwicklung und zur Laufzeit wird Java 1.0.2 benötigt. Es wird vorausgesetzt, daß eine entsprechende Java-Installation vorliegt.

- Zur Laufzeit sind die ebenfalls in Abschnitt A.3 angegebenen Java-Packages und Skripten notwendig. Letztere werden in Abschnitt A.2 beschrieben.

A.2 Kommandos

Voraussetzungen

Zur Laufzeit sind die kompilierten Java-Klassen der „Frisco F"-Implementierung und die kompilierten Java-Klassen des jb/jf-Programmpakets erforderlich. Die kompilierten Java-Klassen sind in den gleichen Verzeichnissen enthalten wie ihre Quelldateien.

Kommandozeilen-Parser

Um eine „Frisco F"-Datei parsen zu können, steht das Skript `friscof` zur Verfügung. Das Skript bringt die Klasse *FriscoF* zur Ausführung. Als Argumente können übergeben werden:

- Der Dateiname eines „Frisco F"-Quellprogramms.

- Mit `-l` *datei* wird die Datei *datei* als Liste interpretiert, in der jede Zeile ein Dateiname eines „Frisco F"-Quellprogramms bezeichnet.

Beispielsweise werden durch

```
friscof -l prelude_files special_defs.ff my_defs.ff
```

im folgenden zuerst die Dateien geparst, die in der Datei `prelude_files` angegeben sind. Anschließend werden die Dateien `special_defs.ff` und `my_defs.ff` geparst.

Parser-GUI

Die Parser-GUI ist eine einfache graphische Oberfläche, in der die Funktionalität des „Frisco F"-Parsers Verfügbar ist. Neben der Möglichkeit, Dateien parsen zu können, kann der Quellcode auch in einem Textfeld eingegeben werden. Zur Ausgabe der Meldungen wird nicht StdOut verwendet, sondern ein eigenes Meldungsfenster geöffnet. Die Parser-GUI kann mit dem Skript `ffgui` gestartet werden. Die Abbildungen A.1 und A.2 zeigen Parser-GUI und Meldungsfenster.

A.3 Technische Daten

Programmdateien

Alle Programmdateien liegen in einem einzigen Hauptverzeichnis bzw. in einem Unterverzeichnis davon. Im Hauptverzeichnis sind die folgenden Dateien enthalten:

`Copyright`	Copyright-Vermerk des jb/jb-Programmpakets.
`friscof`	Skript für den Kommandozeilen-Parser.
`ffgui`	Skript für die Parser-GUI.
`javaff`	Skript zum Ausführen des Java-Interpreters.
`make.ff`	Skript zum Kompilieren der „Frisco F"-Quelldateien.
`makefile.ff`	Make-Datei zum Kompilieren der „Frisco F"-Quelldateien.

Die folgenden Unterverzeichnisse enthalten den Quellcode und die kompilierten Java-Dateien des „Frisco F"-Parsers ohne die Parser-GUI.

/astAtomic	/astBase	/astExp	/astTopLevel
/astType	/bindings	/exceptions	/logBase
/outputStreamLog	/parser	/parserEnvironment	/scc
/substitution	/symboltable	/typeAndKindInference	/utility
/yy			

Die folgenden Unterverzeichnisse enthalten den Quellcode und die kompilierten Java-Dateien, die für die Parser-GUI zusätzlich benötigt werden.

/dialogLog /messagesDialog /parserGUI /testing

Das Unterverzeichnis `makeparser` enthält die Scanner-Definition `friscof.lex` und die Parser-Definitionen `friscof.y` sowie die Make-Datei `makefile.yy` und das Skript `make.yy` zum Erzeugen des Scanners und des Parsers.

Das Unterverzeichnis `source` enthält unter anderem die Datei `Prelude.ff` (Standard-Prelude für „Frisco F").

Die Unterverzeichnisse `jbf` und `regexp` sind Bestandteil des jb/jf-Programmpakets.

jb/jf-Programmpaket

Das jb/jf-Programmpaket Version 4.2 kann unter der ftp-Adresse

`ftp://ftp.cs.colorado.edu/pub/cs/distribs/arcadia/jb.tar`

abgeholt werden. Eine Informationsdatei ist unter der ftp-Adresse

`ftp://ftp.cs.colorado.edu/pub/cs/distribs/arcadia/jb.txt`

zu finden. Zur Installation und zur Kompilierung der Java-Klassen sei auf die entsprechenden Dokumente verwiesen.

Entwicklungswerzeuge

Als Entwicklungswerkzeuge wurden verwendet:

- Java-Compiler Version 1.0.2
- Java-Interpreter Version 1.0.2
- jb/jf-Programmpaket Version 4.2
- Bison Version 1.25
- Flex Version 2.5.3
- Unix-Tool make

A.4 Entwicklung

Scanner und Parser

Zur Erzeugung der Scanner- und Parser-Dateien steht im Unterverzeichnis `makeparser` das Skript `make.yy` zur Verfügung. Es verwendet die Make-Datei `makefile.yy`. Dieses Skript ist aufzurufen, falls eine der Dateien `friscof.lex` oder `friscof.y` geändert wurde.

Packages

Zur Kompilierung der Java-Klassen (beispielsweise nach der Erzeugung des Scanners oder Parsers) steht im Hauptverzeichnis das Skript `make.ff` zur Verfügung. Es verwendet die Make-Datei `makefile.ff`.

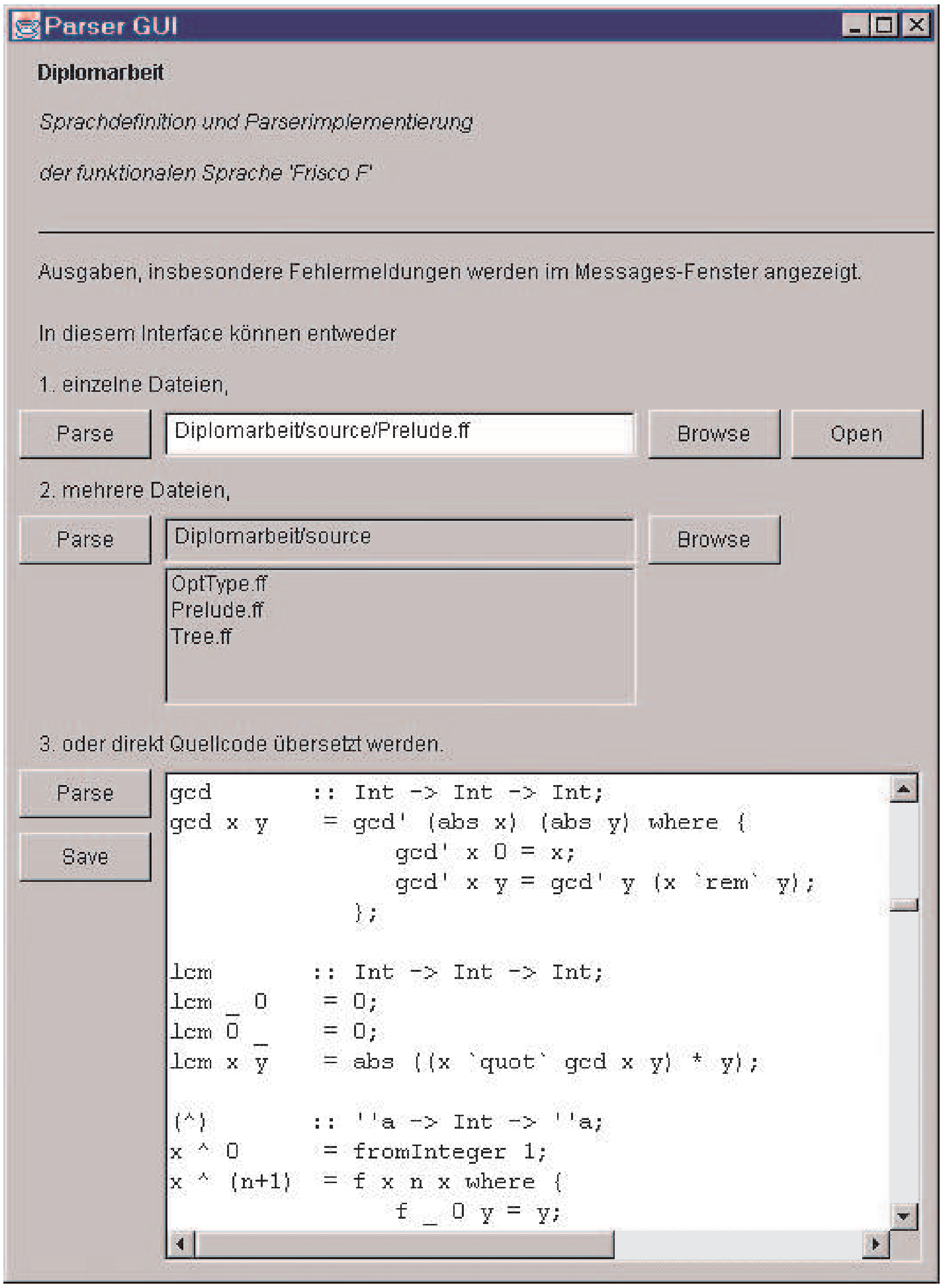


Abbildung A.1: Die Parser-GUI

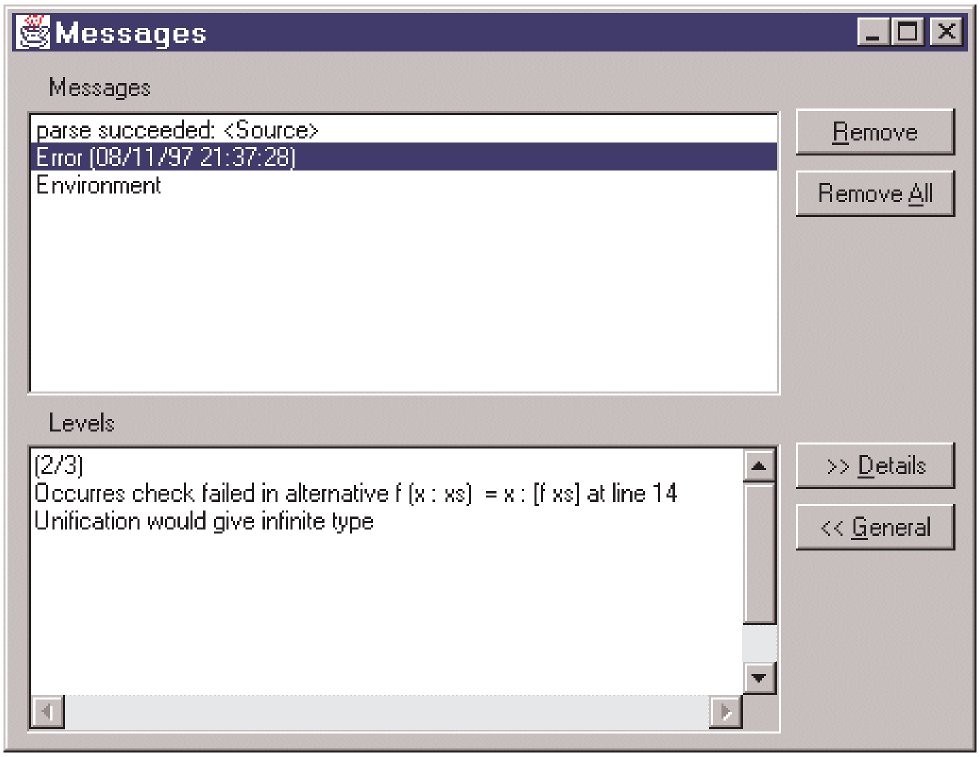


Abbildung A.2: Das Meldungsfenster

Anhang B

Primitiven von „Frisco F“

Folgende Primitiven von „Frisco F“ sind fest vorgegeben und können nicht erweitert werden.

`primEqInt`	`::`	`Int -> Int -> Bool`	prüft Gleichheit zweier Ganzzahlen
`primLeInt`	`::`	`Int -> Int -> Bool`	prüft „kleiner oder gleich“ für zwei Ganzzahlen
`primPlusInt`	`::`	`Int -> Int -> Int`	addiert zwei Ganzzahlen
`primMinusInt`	`::`	`Int -> Int -> Int`	subtrahiert zwei Ganzzahlen
`primDivInt`	`::`	`Int -> Int -> Int`	Ganzzahl-Division, gerundet gegen $-\infty$
`primQuotInt`	`::`	`Int -> Int -> Int`	Ganzzahl-Division, gerundet gegen 0
`primRemInt`	`::`	`Int -> Int -> Int`	Rest mit (x ‘div‘ y)*y + (x ‘rem‘ y) == x
`primModInt`	`::`	`Int -> Int -> Int`	Modulo mit gleichem Vorzeichen wie Divisor
`primMulInt`	`::`	`Int -> Int -> Int`	multipliziert zwei Ganzzahlen
`primNegInt`	`::`	`Int -> Int`	negiert eine Ganzzahl
`primEqFloat`	`::`	`Float-> Float-> Bool`	prüft Gleichheit zweier Gleitpunktzahlen
`primLeFloat`	`::`	`Float-> Float-> Bool`	prüft „kleiner oder gleich“ für zwei Gleitpunktzahlen
`primPlusFloat`	`::`	`Float-> Float-> Float`	addiert zwei Gleitpunktzahlen
`primMinusFloat`	`::`	`Float-> Float-> Float`	subtrahiert zwei Gleitpunktzahlen
`primDivFloat`	`::`	`Float-> Float-> Float`	dividiert zwei Gleitpunktzahlen
`primMulFloat`	`::`	`Float-> Float-> Float`	multipliziert zwei Gleitpunktzahlen
`primNegFloat`	`::`	`Float-> Float`	negiert eine Gleitpunktzahl
`primSinFloat`	`::`	`Float-> Float`	Sinus
`primAsinFloat`	`::`	`Float-> Float`	Arkussinus
`primCosFloat`	`::`	`Float-> Float`	Kosinus
`primAcosFloat`	`::`	`Float-> Float`	Arkuskosinus
`primTanFloat`	`::`	`Float-> Float`	Tangens
`primAtanFloat`	`::`	`Float-> Float`	Arkustangens
`primAtan2Float`	`::`	`Float-> Float -> Float`	Arkustangens mit Quandranteninformation
`primLogFloat`	`::`	`Float-> Float`	natürlicher Logarithmus
`primLog10Float`	`::`	`Float-> Float`	Logarithmus zur Basis 10
`primExpFloat`	`::`	`Float-> Float`	Exponent zur Basis e
`primSqrtFloat`	`::`	`Float-> Float`	Quadratwurzel
`primCharToInt`	`::`	`Char -> Int`	berechnet ASCII-Wert eines Zeichens
`primIntToChar`	`::`	`Int -> Char`	berechnet Zeichen aus ASCII-Wert
`primFloatToInt`	`::`	`Float -> Int`	runded eine Gleitpunktzahl zu einer Ganzzahl
`primIntToFloat`	`::`	`Int -> Float`	konvertiert eine Ganzzahl in eine Gleitpunktzahl
`primPrint`	`::`	`a -> String`	erzeugt String-Repräsentation des Arguments
`primHash`	`::`	`'a -> Int`	erzeugt Hashwert des Arguments

Anhang C

Spezifikation der anonymen Typklassen

„Frisco F“ stellt zwei (anonyme) Typklassen (im folgenden mit *Eq* und *Num* bezeichnet) implizit zur Verfügung. Instanzen der Typklasse *Eq* werden automatisch für Standard- und benutzerdefinierte Datentypen außer für Funktionen gebildet, so daß die explizite Definition von Typklassen-Instanzen nicht notwendig ist. Zur Typklasse *Num* existieren Instanzen für `Int` und `Float`.

Typklasse *Eq*
Die Typklasse *Eq* umfaßt folgende Elementfunktionen (member functions):

```
(==)  ::  'a -> 'a -> Bool;
(/=)  ::  'a -> 'a -> Bool;
(<)   ::  'a -> 'a -> Bool;
(>)   ::  'a -> 'a -> Bool;
(<=)  ::  'a -> 'a -> Bool;
(>=)  ::  'a -> 'a -> Bool;
min   ::  'a -> 'a -> 'a;
max   ::  'a -> 'a -> 'a;
hash  ::  'a -> Int;
```

Es folgen nun die Elementfunktionen mit Default-Definition.

```
x /= y  =  not (x == y);
x >= y  =  y <= x;
x < y   =  x <= y && x /= y;
x > y   =  y < x;
hash x  =  primHash x;
```

```
max x y  |  x >= y  =  x;
         |  y >= x  =  y;
min x y  |  x <= y  =  x;
         |  y <= x  =  y;
```

Für alle übrigen Elementfunktionen folgt nun die Definition nach Typen gegliedert.

Typ `Int`:

```
x == y  =  primEqInt x y;
x <= y  =  primLeInt x y;
```

Typ `Float`:

```
x == y  =  primEqFloat x y;
x <= y  =  primLeFloat x y;
```

Typ `Char`:

```
x == y  =  primCharToInt x == primCharToInt y;
x <= y  =  primCharToInt x <= primCharToInt y;
```

Typ `Bool`:

```
False  ==  False  =  True;
True   ==  True   =  True;
_      ==  _      =  False;

False  <=  False  =  True;
True   <=  True   =  True;
False  <=  True   =  True;
_      <=  _      =  False;
```

Typ ():

```
() == ()  =  True;
() <= ()  =  True;
```

Falls die Instanz $Eq\ a$ definiert ist, so ist auch die Instanz für Listen vom Typ $[a]$ definiert.
Listentyp $Eq\ a \Rightarrow Eq\ [a]$:

```
[]      ==  []      =  True;
(_:_)   ==  []      =  False;
[]      ==  (_:_)   =  False;
(x:xs)  ==  (y:ys)  =  x == y && xs == ys;

[]      <=  _       =  True;
(_:_)   <=  []      =  False;
(x:xs)  <=  (y:ys)  =  x < y || (x == y && xs <= ys);
```

Falls für $n \geq 2$ die Instanzen $Eq\ a_i, 1 \leq i \leq n$ definiert sind, so ist auch die Instanz für n-Tupel $(a_1, a_2, ..., a_n)$ definiert.
Tupeltyp $Eq\ a_1$, $Eq\ a_2$, ..., $Eq\ a_n \Rightarrow Eq\ (a_1, a_2, ..., a_n)$:

```
(x1,...,xn) == (y1,...,yn)  =  x1 == y1 && ...  && xn == yn;
(x1,...,xn) <= (y1,...,yn)  =  x1 <= y1 && ...  && xn <= yn;
```

Falls für $n \geq 0$ die Instanzen $Eq\ a_i, 1 \leq i \leq n$ definiert sind, so ist auch die Instanz für einen n-stelligen benutzerdefinierten Typkonstruktor C mit $m \geq 1$ Konstruktorfunktionen C_1, ..., C_m definiert.
Typkonstruktor $Eq\ a_1$, $Eq\ a_2$, ..., $Eq\ a_n \Rightarrow Eq\ C\ a_1\ a_2\ ...\ a_n$:

Für $n = 0$ gilt:

C_1 == C_1 = True;
C_2 == C_2 = True;
...
C_m == C_m = True;
_ == _ = False;

Für $n > 0$ gilt:

$C_1\ x_1\ \ldots\ x_n$ == $C_1\ y_1\ \ldots\ y_n$ = x_1 == y_1 && ... && x_n == y_n;
$C_2\ x_1\ \ldots\ x_n$ == $C_2\ y_1\ \ldots\ y_n$ = x_1 == y_1 && ... && x_n == y_n;
...
$C_m\ x_1\ \ldots\ x_n$ == $C_m\ y_1\ \ldots\ y_n$ = x_1 == y_1 && ... && x_n == y_n;
_ == _ = False;

Für gleiche Konstruktorfunktionen wird der Vergleich auf die Elemente erweitert. Bei ungleichen Konstruktorfunktionen entscheidet die Reihenfolge der Aufschreibung bei der Definition des Datentyps C. Für $1 \leq i < j \leq m$ gilt:

$C_i\ x_1\ \ldots\ x_n$ <= $C_i\ y_1\ \ldots\ y_n$ = (x_1, ..., x_n) <= (y_1, ..., y_n);
$C_i\ x_1\ \ldots\ x_n$ <= $C_j\ y_1\ \ldots\ y_n$ = True;

Für $1 \leq j < i \leq m$ gilt:

$C_i\ x_1\ \ldots\ x_n$ <= $C_j\ y_1\ \ldots\ y_n$ = False;

Die Instanz A `Bool` ist mit

```
Bool = False | True;
```

ein Sonderfall dieser Instanzen mit $n = 0$ und $m = 2$ ($C_1 =$ `False`, $C_2 =$ `True`).

Typklasse *Num*

Die Typklasse *Num* umfaßt folgende Elementfunktionen (member functions):

```
(+)          :: ''a -> ''a -> ''a;
(-)          :: ''a -> ''a -> ''a;
(*)          :: ''a -> ''a -> ''a;
(/)          :: ''a -> ''a -> ''a;
negate       :: ''a -> ''a;
fromInteger  :: Int -> ''a;
```

Für alle Elementfunktionen folgt nun die Definition nach Typen gegliedert.

Typ `Int`:

```
x + y          =  primPlusInt x y;
x - y          =  primMinusInt x y;
x * y          =  primMulInt x y;
x / y          =  primDivInt x y;
negate x       =  primNegInt x;
fromInteger x  =  x;
```

Typ `Float`:

```
x + y          =  primPlusFloat x y;
x - y          =  primMinusFloat x y;
x * y          =  primMulFloat x y;
x / y          =  primDivFloat x y;
negate x       =  primNegFloat x;
fromInteger x  =  primIntToFloat x;
```

Literaturverzeichnis

[Aho72] Aho, A. V., Hopcroft, J. E., Ullman, J. D., *Design and Analysis of Algorithms*, Addison Wesley, 1972

[Aho88] Aho, A. V., Sethi, R., Ullman, J. D., *Compilerbau, Teil 1, Teil 2*, Addison Wesley 1988

[Bir89] Bird, R., Wadler, P., *Introduction to functional programming*, Prentice Hall International, 1989

[BRJ98] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language – User Guide*, Addison Wesley, 1998

[Fie88] Field, A. J., Harrison, P. G., *Functional Programming*, Addison Wesley Publishing Company, 1988

[Fla96] Flanagan, D., *Java in a Nutshell*, O'Reilly & Associates, Inc, 1996

[Ghe89] Ghezzi, C., Jazayeri, M., *Konzepte der Programmiersprachen, Begriffliche Grundlagen, Analyse und Bewertung*, R. Oldenbourg Verlag GmbH, München, 1989

[Hop90] Hopcroft, J. E., Ullman, J. D., *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*, Addison-Wesley Publishing Company, 2. Auflage 1990

[Hin92] Hinze R., *Einführung in die funktionale Programmierung mit Miranda*, Teubner, 1992

[Hud91] Hudak, P., Wadler, P. et al, *Report on the programming language Haskell, a non-strict purely functional language (Version 1.1)*, Technical report, Yale University/Glasgow University, August 1991

[Job92] Jobst, F., *Compilerbau, Von der Quelle zum professionellen Assemblertext*, Hanser, 1992

[Jon92] Jones, M. P., *Qualified Types: Theory and Practice*, PhD dissertation, Oxford University Computing Laboratory, Programming Research Group, 11 Kelbe Road, Oxford OX1 3QD, England, July 1992

[Jon94a] Jones, M. P., *The implementation of the Gofer functional programming system*, Research Report YALEU/DCS/RR-1030, Yale University, Department of Computer Science, P.O. Box 208285, New Haven, CT 06520-8285, May 1994, http://www.cs.nott.ac.uk/Department/Staff/mpj/goferimp.html

[Jon94b] Jones, M. P., *An Introduction to Gofer*, draft version, May 22, 1994

[Kor95] Korger, M., *Ein um Subtyping erweitertes Typinferenz-Verfahren für das Gofer-Objekt-System*, Diplomarbeit, Technische Universität München, Fakultät für Informatik, Februar 1995

[Mey90] Meyer, B., *Object-oriented Software Construction*, Prentice Hall, 1988

[Mil78] Milner, R., *A Theory of Type Polymorphism in Programming* Journal of Computer and System Sciences, 17:348 - 375, 1978

[Pau96] Paulson, L. C., *ML for the Working Programmer*, Cambridge University Press, 1996, 2nd Edition

[Poe94] Poetzsch-Heffter, A., *Übersetzung von Programmiersprachen*, Vorlesungsskript WS 94/95, Version 1.0, Technische Universität München, Fakultät für Informatik

[Rob65] Robinson, J. A., *A Maschine-Oriented Logic Based on the Resolution Principle*, Journal of the ACM, 12(1):23/41, Januar 1965

[Sch85] Schreiner, A. T., Friedmann, G., *Compiler bauen mit UNIX - eine Einführung*, Hanser, 1985

[Sok91] Sokolowski, S., *Applicative Higher Order Programming: The standard ML perspective*, Capman & Hall Computing, 1991

[Sta89] Staubach, G., *UNIX-Werkzeuge zur Textmusterverarbeitung - Awk, Lex und Yacc*, Springer, 1989

[Sto77] Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, 1977

[Str92] Stroustrup, B., *Die C++ Programmiersprache*, Addison Wesley, 1992, 2. Auflage

[Thi94] Thiemann, P., *Grundlagen der funktionalen Programmierung*, Teubner, 1994

[War98] Warmer, J., Kleppe, A., *The Object Constraint Language*, Addison Wesley, 1998

[Wil96] Wilhelm, R., Maurer, D., *Compiler Design*, Addison Wesley, 1996