INHALTSVERZEICHNIS.

1 Einleitung 3
1.1 Einführung . . . 3
1.2 Ziele der Arbeit . . . 3

2 Prädikat-Transitionsnetze im ConTract 5
2.1 Beschreibung von ConTract . . . 5
2.2 Prädikat-Transitionsnetze . . . 5
2.3 Prädikate . . . 7

3 Syntax und Semantik der Prädikate 9
3.1 Beschreibung der Metasprache . . . 9
3.2 einige Elemente der Prädikatsprache . . . 10
3.3 Typen . . . 10
3.4 Junktoren . . . 11
3.5 Vergleichsoperationen . . . 12
3.6 arithmetische Berechnungen . . . 13
3.7 Zeitfunktionen . . . 14
3.8 Typkompatibilität und Overloading . . . 14
3.9 implizite und explizite Typumwandlungen . . . 16
3.10 Vorrang . . . 17
3.11 Linksassoziativität oder Rechtsassoziativität . . . 18
3.12 Linksrekursion oder Rechtsrekursion . . . 18

4 Aufbau der Prädikatverwaltungskomponente 20
4.1 grobe Übersicht . . . 20
4.2 Übersetzung . . . 21
4.3 Zwischencode . . . 22
4.3.1 Syntaxbaum . . . 22
4.3.2 linearisierte Darstellung des Syntaxbaums . . . 24
4.4 Evaluierungsfunktion . . . 26

5 Detaillierte Darstellung der Prädikatverwaltungskomponente 28
5.1 lexikalische Analyse . . . 28
5.1.1 Ausführen eines FLEX-Programms . . . 28
5.1.2 Aufbau eines FLEX-Programms . . . 29
5.2 syntaktische Analyse . . . 31
5.2.1 Ausführen eines YACC-Programms . . . 32
5.2.2 Aufbau eines YACC-Programms . . . 32
5.2.3 Arbeitsweise eines mit YACC generierten Parsers . . . 34
5.3 semantische Analyse . . . 36

5.3.1 Fehlerbehandlung 36
5.4 Zwischencodeerzeugung 38
5.4.1 Aufbau des Syntaxbaums 38
5.5 Codeoptimierung 40
5.5.1 Algebraische Optimierungen 41
5.5.2 Optimierende short-circuit-Evaluation 42
5.6 Zwischencode 45
5.6.1 Schreiben des Zwischencodes durch den Compiler 45
5.6.2 Beschreibung des Zwischencodes 45
5.7 Evaluierungsfunktion 47

6 Ein ausführliches Beispiel **49**

7 Zusammenfassung und Ausblick **53**

A Grammatik der Prädikatsprache **54**

Literaturverzeichnis **56**

1 Einleitung

1.1 Einführung

Große Datenmengen werden heutzutage in Datenbanken gespeichert. Um auf diese Daten von mehreren parallelen Prozessen aus zugreifen zu können, wurde das klassische Transaktionskonzept entwickelt. Dies nimmt dem Programmierer die Arbeit ab, sich um die Verwaltung und Synchronisation der Datenobjekte kümmern zu müssen.

Da aber die zu verwaltenden Daten in der Zukunft immer größer werden, wie zum Beispiel bei Multimedia-Daten, gibt es immer mehr langlebige Transaktionen, die parallel ablaufen. Hier stößt das klassische Transaktionskonzept jedoch an Grenzen. Zwei Nachteile sind besonders deutlich:

1. lange Sperren: Eine Transaktion, die auf Daten zugreift, setzt Sperren auf diese Objekte für ihre gesamte Lebensdauer. Dies ergibt sich aus der Isoliertheits- und Konsistenzbedingung für Transaktionen. Aufgrund dieser Tatsache müssen nebenläufige Transaktionen, die auf die gleichen Daten zugreifen, warten bis die sperrende Transaktion endgültig fertig ist. Dadurch gibt es gerade bei langandauernden Transaktionen weniger Parallelität und die Effizienz sinkt gewaltig.
2. Fehlerbehandlung: Wenn bei einer Transaktion ein Fehler auftritt, so wird sie vollständig zurückgesetzt. Dies ergibt sich aus der Atomizitätsanforderung für Transaktionen. So nimmt bei langandauerndern Transaktionen das Zurücksetzen sehr viel Zeit in Anspruch, und auch dieses trägt zum Arbeitsverlust bei.

Wegen dieser Gründe wurde das klassische Transaktionskonzept erweitert. Im IPVR (Institut für Parallele und Verteilte Höchstleistungsrechner) an der Universität Stuttgart wurde dafür das **ConTract**-Modell entwickelt. Bei ConTracts wird ein langlebiger Datenbankzugriff in viele kleine klassische Transaktionszugriffe, den sogenannten Steps aufgeteilt. Verschiedene Steps können parallel ablaufen. Der Programmierer[1] kann nun selber den parallelen oder sequentiellen Ablauf und die Synchronisation der Steps effizient definieren, dies geschieht im sogenannten Skript.

Im Skript gibt es auch, wie bei jeder anderen Programmiersprache, auch Konstrukte für Verzweigungen und Schleifen. Diese Verzweigungen und Schleifen können auch bedingt sein, das heißt es wird aufgrund von Prädikaten entschieden, welcher Pfad gerade abgearbeitet werden soll.

1.2 Ziele der Arbeit

Am IPVR wird derzeit das APRICOTS-System weiterentwickelt. Es handelt sich dabei um einen Prototyp zur Implementierung des ConTract-Modells.

1. Wenn ich Programmierer sage, so meine ich natürlich auch die Programmiererin.

Im bisherigen Prototyp waren die Prädikate noch nicht vorhanden, die Steps wurden also nach einem starren Ablaufschema abgearbeitet.

Meine Aufgabe dieser Arbeit war es, das Modell um eine Prädikatverwaltungskomponente zu erweitern, damit in Zukunft auch bedingte Verzweigungen ausgeführt werden können. Schwerpunkte der Arbeit sind dabei:

1. Eine Sprache zu entwerfen, welche die Prädikate beschreibt.
2. Schnittstellen für eine Evaluierungsfunktion zur Verfügung zu stellen.

Im Rahmen dieser Studienarbeit soll ein Prototyp der Prädikatverwaltungskomponente entworfen werden. Einen Teil der Komponente habe ich in der Programmiersprache C implementiert. Dabei enthält die Implementierung folgendes:

- Einen Compiler, der das Prädikat in einen Zwischencode übersetzt.
- Eine Evaluierungsfunktion, die entscheidet, ob das Prädikat wahr oder falsch ist.

Ein wichtiger Aspekt bei der Implementierung war, daß die Evaluierungsfunktion beim Ablauf eines ConTracts möglichst effizient sein soll, das heißt sie soll möglichst wenig Zeit in Anspruch nehmen.

2 Prädikat-Transitionsnetze im ConTract

2.1 Beschreibung von ConTract

Ein ConTract ist eine konsistente und fehlertolerante Ausführung einer beliebigen Sequenz vordefinierter Aktionen - genannt Steps - nach einer explizit spezifizierten Kontrollstruktur - genannt Skript. Es wurde definiert, um langlebige Transaktionen zu spezifizieren.

Eine Einführung in ConTract enthält [WäRe91].

Steps sind die elementaren Verarbeitungseinheiten im ConTract. Sie enthalten die Algorithmen und sind eingebettet in eine traditionelle ACID-Transaktion - haben somit alle ACID-Eigenschaften und enthalten keine Parallelverarbeitung. Sie entsprechen den elementaren Befehlen einer beliebigen Programmiersprache.

Das Skript beschreibt den Kontrollfluß und andere Ausführungsaspekte, wie sequentielle oder parallele Verarbeitung und Synchronisation der Steps.

Der ConTract-Manager ist die oberste Kontrollinstanz bei der ConTract-Ausführung. Er startet die Ausführung, überwacht und steuert den Ablauf. Dabei orientiert er sich an den Regeln, die im ConTract-Skript stehen. Eine genaue Beschreibung des ConTract-Managers befindet sich in [Wah91].

Innerhalb eines ConTracts wird auf lokale Variablen eines ConTracts zugegriffen, den sogenannten Kontextvariablen. Diese sind in einer Datenbank - dem Kontext - gespeichert. Die Kontextdatenbank wird genauer in [BuRa90] beschrieben.

2.2 Prädikat-Transitionsnetze

Die Struktur eines Skripts wird durch eine eigene Programmiersprache beschrieben. Die Definition der Sprache findet sich in [Mas90] [Sei95].

Grafisch kann die Sprache durch Prädikat-Transitionsnetze dargestellt werden. Prädikat-Transitionsnetze sind eine Erweiterung von Petri-Netzen. Eine Einführung in Petri-Netze bietet [Ri85]. Im weiteren werden Petri-Netze als bekannt vorausgesetzt. Die Unterschiede der Prädikat-Transitionsnetze zu Petri-Netzen sind folgende:

- Die Steps entsprechen den Stellen im Petri-Netz.
- Die Transitionen sind die sogenannten Events im ConTractsystem.
- Auf einen Step können mehrere Events folgen und einem Event können mehrere Steps folgen. Steps und Events treten immer abwechselnd auf.

- Ein Übergang von einem Step zu einem Event enthält eine Bedingung, ein sogenanntes Prädikat. Das dem Prädikat nachfolgende Event darf nur feuern, wenn das Prädikat wahr ist, das heißt wenn die Bedingung erfüllt ist. Sonst ist dieser Weg gesperrt. Die Bedingungen beziehen sich auf die Kontextvariablen. So wird der Kontrollfluß gesteuert

Die nachfolgenden Abbildungen zeigen einige Beispiele für Prädikat-Tranisitionsnetze.

Dabei werden folgende Darstellungsmittel verwendet:

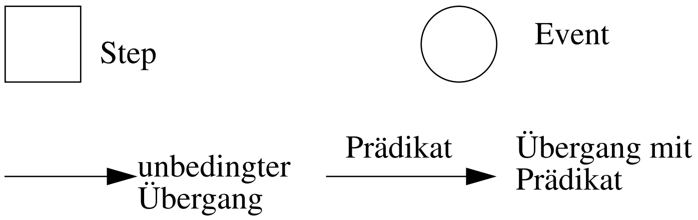


ABBILDUNG 1. Elemente des Prädikat-Transitionsnetz

Eine Sequenz:

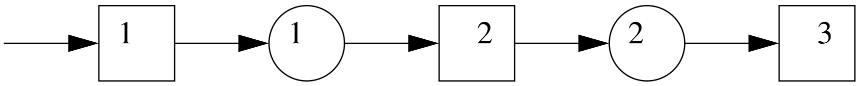


ABBILDUNG 2. Eine Sequenz im Prädikat-Transitionsnetz

Nacheinander wird Step 1, Step 2 und zum Schluß Step 3 ausgeführt.

Eine Verzweigung:

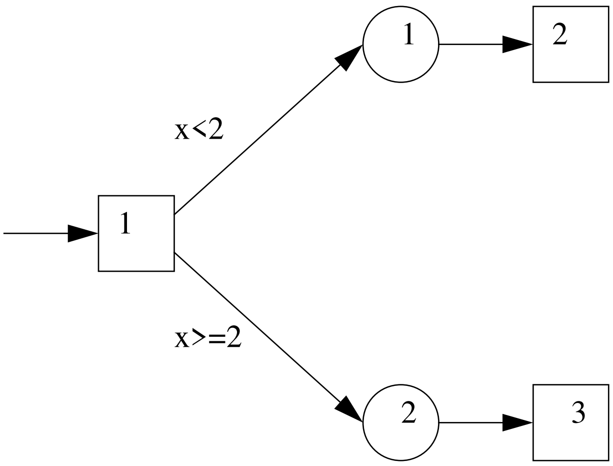


ABBILDUNG 3. Eine Verzweigung im Prädikat-Transitionsnetz

Zuerst wird Step 1 ausgeführt. Dann kommt eine Verzweigung - welcher Weg gegangen wird, entscheidet das Prädikat. Wenn x kleiner als 2 ist, so feuert Event 1 und Step 2 wird ausgeführt. Sonst, das heißt x ist größer oder gleich 2, feuert Event 2 und Step 3 darf zur Ausführung kommen

Eine Schleife:

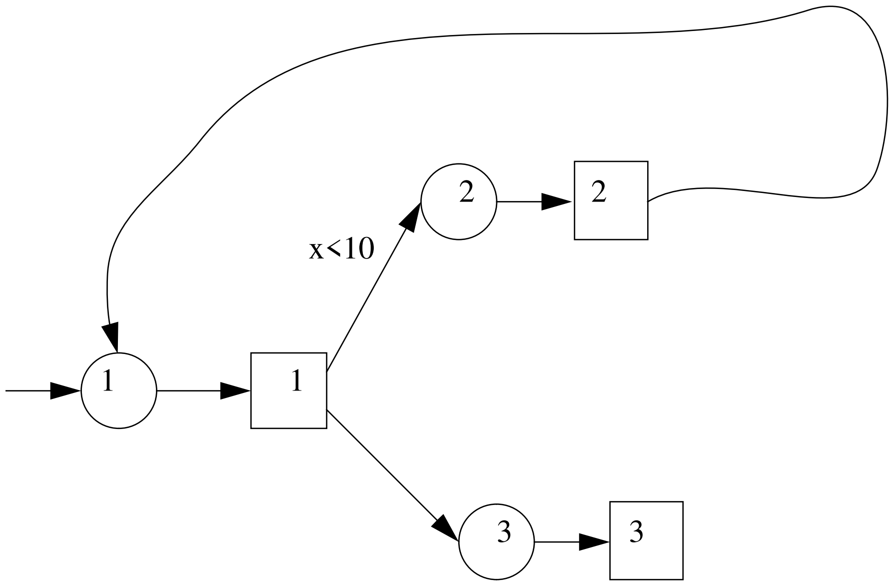


ABBILDUNG 4. Eine Schleife im Prädikat-Transitionsnetz

Die Schleife Event 1, Step 1, Event 2, Step 2 wird solange ausgeführt, solange der Wert von x nach dem Step 1 kleiner als 10 ist.

Ist der Wert von x nach dem Step 1 nicht kleiner als 10, so feuert Event 3 und danach wird Step 3 ausgeführt.

2.3 Prädikate

In diesem Unterkapitel werden der Begriff ‘Prädikat‘ und noch einige weitere Begriffe theoretisch definiert. Dies ist notwendig, um Begriffsklarheit zu gewährleisten. Es handelt sich hier um eine modifizierte Einführung in die Prädikatenlogik. Eine genaue Einführung findet sich in [Schö87].

Zur syntaktischen Beschreibung der Prädikate wird eine formale Sprache definiert. Im Weiteren wird sie Prädikatsprache genannt.

Die Bestandteile, das heißt die Terminale, dieser Sprache sind:

- Variablen
- Funktionssymbole
- Prädikatsymbole

Variablen sind Platzhalter für Elemente aus einer Wertemenge. Funktionssymbole sind Zeichen, die Funktionen oder Relationen repräsentieren, Prädikatsymbole erscheinen dementsprechend für Prädikate.

Nun kann definiert werden, was ein Term ist:

- Jede Variable ist ein Term.
- Falls f ein Funktionssymbol ist und falls $t_1,...,t_k$ Terme sind, so ist auch $f(t_1,..,t_k)$ ein Term.
- Nullstellige Funktionssymbole, das heißt Funktionssymbole ohne Argumente, heißen Konstanten und ebenfalls Terme.

Daraus kann nun wieder definiert werden, was eine Formel ist:

- Falls P ein Prädikatsymbol ist, und falls $t_1,...,t_k$ Terme sind, dann ist $P(t_1,...,t_k)$ eine Formel.
- Für jede Formel F ist auch !F eine Formel. Für alle Formeln F und G sind auch F||G und F&&G Formeln. Dies ist die Verknüpfung von Teilformeln durch Junktoren.

Im weiteren Verlauf der Studienarbeit wird für den Begriff Formel auch der Begriff Prädikatstring verwendet. Diese Begriffe sind also äquivalent.

Diese zunächst sinnlose Folge von Symbolen einer Formel geht erst durch eine Interpretation in ein sogenanntes Prädikat über. Eine Interpretation ist eine Belegung aller Variablen mit Werten aus der Wertemenge, eine Festsetzung aller Konstante mit festen Werten und eine Zuordnung der Funktionssymbole zu festen Relationen oder Funktionen. Durch eine Interpretation kommt also Semantik hinzu, während eine Formel ein rein syntaktisches Gebilde ist. Ein Prädikat hat also eine Semantik.

Allgemein wird ein n-stelliges Prädikat P als Abbildung

$$P: M_1 * M_2 *...*M_n \text{ -> Wahrheitswert}$$

definiert.

Die Mengen $M_1,..,M_n$ sind Wertemengen der im Prädikat enthaltenden Variablen und Konstanten.

Ein Wahrheitswert ist ein Element aus der Menge {TRUE, FALSE}, der das Zutreffen des Prädikats charakterisiert. Er hängt von der Belegung der Variablen ab.

3 Syntax und Semantik der Prädikate

Das Ziel beim Entwurf der Prädikatsprache ist, für den ConTract-Programmierer verständliche Prädikate zu entwerfen, die aber trotzdem die Bandbreite der möglichen Anwendungen abdecken. Deshalb wurde sich an den allgemein bekannten imperativen Programmiersprachen, wie C [KeRi90] oder SQL [X/O91], orientiert, damit es für den ConTract-Programmierer nicht zu viel Umstellungsschwierigkeiten gibt.

Eine vollständige Grammatik der Prädikatsprache ist im Anhang enthalten. In diesem Kapitel werden die in der Sprache enthaltenden Elemente in Syntax und Semantik kurz vorgestellt, außerdem werden ihre Beziehungen untereinander erklärt.

Folgende Elemente sind in einem Prädikat enthalten:

- Variablen
- Konstanten
- Junktoren
- Funktionen oder Relationen

In den ConTract-Prädikaten existieren folgende Funktionen und Relationen:

- Vergleichsoperationen
- arithmetische Operationen
- Zeitfunktionen
- Typumwandlungsfunktionen

Diese genannten Elemente werden im Laufe des Kapitels genauer beschrieben.

Die Grammatik ist für den Prototypen entwickelt. Sie kann im weiteren ergänzt werden.

3.1 Beschreibung der Metasprache

In dieser Studienarbeit wird für den Syntax der Prädikatsprache folgende Metasprache verwendet:

- :
  Die linke Seite wird durch rechte Seite definiert.
- |
  Die links und rechts stehenden Sprachelemente sind Alternativen. Z.B. heißt a | b , daß an dieser Stelle entweder a oder b steht.
- []
  Das innerhalb der eckigen Klammern stehende Sprachelement ist optional, d.h. es kann existieren oder auch nicht.

- *
  Das links von diesem Zeichen stehende Element kann 0 bis unendlich oft mal wiederholt werden.
- { }
  Die geschweiften Klammern grenzen den Teil der Syntax ab, für den vertikale Striche oder Wiederholungen gelten.
- Terminale sind kleingeschrieben und müssen immer aufgeführt werden, wenn sie erscheinen.
- Nichtterminale sind großgeschrieben und werden noch weiter spezifiziert.
- Alle anderen Symbole, wie z.B. die runde Klammer, sind Sprachelemente, und müssen immer aufgeführt werden, wenn sie erscheinen.
- Symbole, die gequotet sind, wie '||', sind Sprachelemente, und müssen immer aufgeführt werden, wenn sie erscheinen.

3.2 Einige Elemente der Prädikatsprache

Da einige Elemente immer wieder benötigt werden, werden sie hier jetzt spezifiziert.

- LETTER
  LETTER ist ein Buchstabe.

  LETTER: A|B|C|..|Z|a|b|c|..|z

- DIGIT
  DIGIT ist eine Ziffer.

  DIGIT:1|2|3|4|5|6|7|8|9|0

- NAME
  Name ist ein Name einer Kontextvariablen. Es ist ein String, der mit einem Buchstaben anfängt und dann aus Buchstaben, Ziffern oder dem underscore ('_') besteht. Dabei ist die Länge eines Namens zur Zeit auf 60 Zeichen beschränkt.

  NAME: LETTER{LETTER|DIGIT|_}

3.3 Typen

Die Werte der Variablen werden in der Kontextdatenbank gespeichert, diese ist eine SQL Datenbank. Deshalb wurde sich bei der Auswahl der Typen an den Typen laut ANSI/ISO-SQL-Standard orientiert, um eine direkte Abbildung der Werte auf die in der Kontextdatenbank gespeicherten Werte zu ermöglichen. Folgende Typen gibt es:

- ganzzahlige Zahlen.

  LONG: DIGIT[DIGIT]*

- Gleitkommazahlen, die optional einen Bruchteil und optional einen Exponententeil haben.
  FLOAT: DIGIT[DIGIT]*[.DIGIT[DIGIT]*][{E|e}[+|-]DIGIT[DIGIT]*]
- String oder Zeichenkette
  STRING: “{ASCII-Zeichen*}“
  Dabei besteht ein String aus mehreren ASCII-Zeichen, die in Anführungszeichen eingeschlossen sind. Es ist zu bemerken, daß der Typ String kein Array of character ist, sondern ein eigener Typ ist.
- ARRAY
  Ein ARRAY ist eine geordnete Sequenz eines Basistyps (long, float oder string). Auf ein einzelnes ARRAY-Element kann man über einen Index zugreifen, dieser Index muß vom Typ long sein.
- RECORD
  Ein RECORD ist eine Struktur, die mehrere Komponenten verschiedenen Typs zu einer zusammengehörigen Einheit zusammenfaßt. Auf eine Komponente eines RECORDS greift man zu, indem man Recordname.Komponentenname angibt.

Zusätzlich zu diesen Typen gibt es noch den Typ einer Zeitkonstante. Diese ist zur Zeit nur als Konstante definiert, kann also noch nicht als Variable verwendet werden.

Syntax der Zeitkonstante:

TIME: HDIGIT : MSDIGIT DIGIT : MSDIGIT DIGIT
HDIGIT: {0DIGIT|1DIGIT|2{0|1|2|3}}
MSDIGIT: 0|1|2|3|4|5

3.4 Junktoren

Junktoren sind Operationen, die Teilprädikate miteinander verknüpfen und einen Wahrheitswert zurückliefern. Folgende werden verwendet:

- die logische Oder-Verknüpfung
  Syntax: ‘||‘
  Es handelt sich hier um kein exklusives, sondern um ein inklusives Oder.

TABELLE 1. Wahrheitswerttabelle für Oder-Verknüpfung

Prädikat 1	**Prädikat 2**	**Ergebnis**
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

- die logische Und-Verknüpfung
  Syntax: &&

TABELLE 2. Wahrheitswerttabelle für Und-Verknüpfung

Prädikat 1	**Prädikat 2**	**Ergebnis**
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

- die logische Nicht-Verknüpfung
  Syntax: !

TABELLE 3. Wahrheitswerttabelle für Nicht-Verknüpfung

Prädikat	**Ergebnis**
TRUE	FALSE
FALSE	TRUE

3.5 Vergleichsoperationen

Vergleichsoperationen sind Operationen, die als Operanden Variablen oder Konstanten bekommen und als Ergebnis einen Wahrheitswert zurückliefern. Es wurde sich in Syntax und Semantik weitgehend an der Programmiersprache C orientiert.

- kleiner
  Syntax: <
  Semantik: für numerische Typen: die übliche kleiner-Operation.
  für Strings: Der erste String ist kleiner als der zweite String, wenn er am ersten Buchstaben, wo ein Unterschied feststellbar ist, er einen kleineren Buchstaben (nach ASCII) hat als der entsprechende Buchstabe des zweiten Strings. Besteht bis zum Ende eines Strings kein Unterschied zum anderen String, so ist derjenige kleiner, der kürzer ist.
  für arrays: jedes ARRAY-Element muß kleiner sein als das entsprechende ARRAY-Element des zweiten Arrays.
  für records: jede Record-Komponente muß kleiner sein als die entsprechende Record-Komponente des zweiten Records.
- kleiner gleich
  Syntax: <=
  Semantik: kleiner oder gleich
- gleich
  Syntax: ==
  Semantik: übliche Wertegleichheit,

bei Strings müssen die Buchstaben und die Länge des Strings übereinstimmen.
bei Arrays muß jedes Element gleich sein.
bei Records muß jede einzelne Komponente gleich sein.

- ungleich
  Syntax: !=
  Semantik: nicht gleich

- größer gleich
  Syntax: >=
  Semantik: größer oder gleich

- größer
  Syntax: >
  Semantik: für numerische Typen: die übliche größer-Operation
  für Strings: Der erste String ist größer als der zweite String, wenn am ersten Buchstaben, wo ein Unterschied feststellbar ist, er einen größeren Buchstaben (nach ASCII) hat als der entsprechende Buchstabe des zweiten Strings. Besteht bis zum Ende eines Strings kein Unterschied zum anderen String, so ist derjenige größer, der länger ist.
  für arrays: jedes ARRAY-Element muß größer sein als das entsprechende ARRAY-Element des zweiten Arrays.
  für records: jede Record-Komponente muß größer sein als die entsprechende Record-Komponente des zweiten Records.

3.6 Arithmetische Berechnungen

Prädikate enthalten außer den oben genannten Operatoren auch oft arithmetische Berechnungen. Deshalb umfassen diese Prädikate auch arithmetische Berechnungen. Auch hier wird sich im Syntax und Semantik weitgehend an der Programmiersprache C orientiert. Folgende Berechnungen gibt es (siehe Kapitel 3.8):

- Addition
  Syntax: +
- Subtraktion
  Syntax: -
- Multiplikation
  Syntax: *
- Gleitpunktdivision
  Syntax: /
- Modulooperation
  Syntax: %
  Diese Operation dividiert den Wert1 durch den Wert2 und gibt den Restwert zurück.

- unäres Minus
  Syntax: -
  Diese Operation wandelt positive Zahlen in negative Zahlen um und umgekehrt. Im Gegensatz zur Subtraktion hat sie nur einen Operanden.

3.7 Zeitfunktionen

Dadurch daß APRICOTS ein transaktionsorientiertes und damit interaktives System ist, kann es Anwendungen geben, die von der Zeit abhängen. Zu diesem Zweck wurden Zeitfunktionen eingeführt. Folgende gibt es:

- before (Zeitkonstante)
  Wenn die aktuelle Ortszeit früher ist als die Zeitkonstante, so ist es TRUE
  Bsp: **before (12:00:00) und es ist zur Zeit gerade 11.45.34 Uhr = TRUE**
  before (12:00:00) und es ist zur Zeit gerade 12.00.00 Uhr = FALSE
  before (12:00:00) und es ist zur Zeit gerade 12.13.09 Uhr = FALSE
- at (Zeitkonstante)
  Wenn die aktuelle Ortszeit gleich der Zeitkonstante ist, so ist es TRUE
  Bsp: **at (12:00:00) und es ist zur Zeit gerade 12.00.00 Uhr = TRUE**
  at (12:00:00) und es ist zur Zeit gerade 12.00.01 Uhr = FALSE
- after (Zeitkonstante)
  Wenn die aktuelle Ortszeit später ist als die Zeitkonstante, so ist es TRUE
  Bsp: **after (12:00:00) und es ist zur Zeit gerade 12:00:01 = TRUE**
  after (12:00:00) und es ist zur Zeit gerade 12:00:00 = FALSE

3.8 Typkompatibilität und Overloading

Nicht jeder Operator kann auf eine Variable oder Konstante jedes Typs angewendet werden. So kann zum Beispiel es kein Prädikat ‘String / String‘ geben, da die Gleitpunktdivision nur bei Operanden vom Typ ‘float‘ funktioniert. Somit wird geprüft, ob die benutzten Konstanten und Variablen den in der Grammatik vorgegebenen Typen entsprechen. Welche Operatoren auf welche Typen angewendet werden können, kann man der folgenden Typkompatibilitätstabelle entnehmen, dabei bedeutet numerisch float oder long. Zeilen, die nicht in der Tabelle auftauchen, sind nicht erlaubt und bedeuten einen Fehler.

TABELLE 4. Typkompatibilitätstabelle

Operator	Operand 1	Operand 2	Resultat
||, &&	boolean	boolean	boolean
!	boolean	---	boolean
<, <=, ==, !=, >=, >	numerisch	numerisch	boolean

TABELLE 4. Typkompatibilitätstabelle

Operator	Operand 1	Operand 2	Resultat
<, <=, ==, !=, >=,>	string	string	boolean
<, <=, ==, !=, >=,>	array	array	boolean
<, <=, ==, !=, >=,>	record	record	boolean
+	long	long	long
+	long	float	float
+	float	long	float
+	float	float	float
+	string	string	string
-, *	long	long	long
-, *	long	float	float
-, *	float	long	float
-, *	float	float	float
/	numerisch	numerisch	float
%	long	long	long
unäres -	numerisch	numerisch	numerisch
int	float	---	long
ord	string	---	long
chr	long	---	string

Andererseits kann es Operatoren geben, die für mehrere Typen definiert sind und für verschiedene Typen eine verschiedene Semantik haben. Dieser Effekt heißt Overloading. In dieser Grammatik tritt er an verschiedenen Stellen auf. Ein gutes Beispiel dafür ist der Operator '+'. Er hat je nach Typ seiner Operanden eine andere Semantik:

- Wenn beide Operanden vom Typ long sind, ist es eine Ganzzahladdition.
  Bsp: **1+2 = 3**
- Wenn ein Operanden vom long und der andere vom Typ float ist, so gibt zuerst eine implizite Typumwandlung von long nach float und danach eine Gleitpunktaddition.
  Bsp: **1+2.7 = float(1) + 2.7 = 3.7**
- Wenn beide Operanden vom Typ float sind, ist es eine Gleitpunktaddition.
  Bsp: **8.5+6.2 = 14.7**
- Wenn beide Operanden vom Typ string sind, so gibt es eine Konkatenation, das heißt der rechte String wird hinten an den linken String angehängt.
  Bsp: **"Hunde" + "kuchen" = "Hundekuchen"**

Somit wird beim Evaluieren eines Prädikats je nach Operator und Typ der Operanden entschieden, welche Operation angewendet wird.

3.9 Implizite und explizite Typumwandlungen

Da - wie oben beschrieben - nicht jeder Typ für jeden Operanden möglich ist, muß es Typumwandlungen geben. Diese kann implizit oder explizit geschehen. Implizite Typumumwandlungen geschehen automatisch, das heißt der Programmierer braucht sie nicht extra anzugeben. Eine implizite Typumwandlung in dieser Grammatik ist die Umwandlung von long nach float. Wenn man zum Beispiel eine Zahl vom Typ long mit einer Zahl vom Typ float vergleicht, so wird vorher die Zahl vom Typ long in eine Zahl vom Typ float umgewandelt, um beide Zahlen miteinander vergleichbar zu machen. Diese Art von Typkonversion wird in dieser Prädikatverwaltungskomponente implizit gemacht, da es auch in allgemein bekannten Programmiersprachen - wie zum Beispiel C oder Pascal - so gehandhabt wird und somit sich ConTract-Programmierer nicht total umgewöhnen muß.

Außer dieser oben beschriebenen Umwandlung kann es aber auch andere Anwendungen geben, für die andere Typumwandlungen benötigt werden. Zum Beispiel möchte der ConTract-Programmierer mit einer Gleitpunktzahl eine Modulo-Operation machen. Dies ist aber nicht möglich, da diese Operation nur für Ganzzahlen definiert ist. Somit muß es ein Konstrukt geben, mit der eine solche Umwandlung möglich ist. Dafür gibt es die expliziten Typumwandlungen. Explizit bedeutet, daß der Benutzer diese Typumwandlungen explizit ausformulieren muß. In dieser Prädikatsgrammatik gibt es drei explizite Typumwandlungen:

- int (Float-Wert)
  Diese Funktion bekommt als Argument einen Gleitpunktwert und wandelt diesen in einen ganzzahligen Wert um, indem der Bruchteil einfach abgeschnitten wird.
  Bsp: **int (2.33) = 2**
  int (28.9999) = 28
- ord (String-Wert)
  Diese Funktion bekommt als Argument einen String. Sie nimmt das erste Zeichen des Strings und wandelt dieses in nach der allgemein bekannten ASCII-Tabelle in die zugehörige Dezimalzahl vom Typ long um.
  Bsp: **ord(“{“) = 123**
  ord(“{blablabla“) = 123
- chr (Long-Wert)
  Diese Funktion bekommt als Argument eine Ganzzahl. Diese wird in das zugehörige ASCII-Zeichen umgewandelt. Das Ergebnis ist vom Typ string.
  Bsp: **chr(123) = “{“**
  chr(ord(“a“)) = “a“

Warum werden diese Typumwandlungen nicht auch implizit gemacht? Dann bräuchte der Anwender doch weniger zu schreiben. Es gibt aber doch einige Gründe, die für explizite Typumwandlungen sprechen:

- Schutz von Variablen gegen sinnlose Operationen.
  Dadurch, daß es keine implizite Typumwandlung gibt, wird verhindert, daß offensichtlich sinnlose Operationen eine Bedeutung bekommen. Welche Bedeutung soll denn zum Beispiel **27 / “ABC“** haben?

- bessere Erkennung von Programmierfehlern vom Compiler.
  Das oben genannte Beispiel kann ein Programmierfehler sein. Bei impliziter Typumwandlungen würde die Typumwandlung implizit gemacht und der Fehler würde nicht erkannt werden.
- bessere Lesbarkeit des Programms.
  Dadurch, daß Typumwandlungen explizit ausformuliert werden müssen, sieht ein Leser des Programms ganz genau, wann eine Typumwandlung vom Programmierer beabsichtigt war. Dies erhöht die Lesbarkeit und damit auch die Wartbarkeit des Programms.
- Vermeidung zusätzlicher Nichtdeterminismen (Unbestimmtheiten)
  Bei impliziter Typumwandlung kann folgender Nichtdeterminismus auftreten:
  Es soll **3.6 + 4.6** berechnet werden, mit impliziter Umwandlung in Ganzzahlen. Nun kann dieses auf zwei Arten geschehen:

1. **int(3.6) + int(4.6) = 3 + 4 = 7**
2. **int(3.6 + 4.6) = int(8.2) = 8**

Bei expliziter Umwandlung kann der Programmierer selber angegeben, welche Umwandlung er haben will und dadurch tritt der oben genannte Nichtdeterminismus nicht auf.

Wegen der oben genannten Vorteile der expliziten Typumwandlungen wurden die impliziten Typumwandlungen auf ein Minimum beschränkt, nämlich auf die Umwandlung von long nach float.

3.10 Vorrang

Ein Operator hat vor einem anderen Operator den Vorrang, wenn er im gleichen Prädikat zuerst vor dem anderen ausgeführt wird.

Hier nun eine Liste des Vorrangs von Operatoren. Sie geht vom geringsten Vorrang zum höchsten Vorrang:

- logische Oder-Verknüpfung
- logische Und-Verknüpfung
- logische Nicht-Verknüpfung
- Vergleichsoperationen
- Zeitfunktionen
- Additive Operationen, das heißt Addition und Subtraktion
- Multiplikative Operationen, das heißt Multiplikation, Division und Modulo-Funktion
- unäres Minus, explizite Typumwandlungsfunktionen
- geklammerte Ausdrücke

Zur Erläuterung ein Beispiel:

2 * 3 + 4 == x || y < 2 && z > 7 wird nach diesen Vorrangregeln so berechnet:

(((2 * 3) + 4) == x) || ((y < 2) && (z > 7))

3.11 Linksassoziativität oder Rechtsassoziativität

Wie sieht es aber nun mit Operatoren aus, die den gleichen Vorrang haben?

Hier gibt es grundsätzlich 2 Möglichkeiten der Abarbeitung: Linksassoziativität oder Rechtsassoziativität. Bei Linksassoziativität wird das Prädikat von links nach rechts abgearbeitet, bei Rechtsassoziativität von rechts nach links.

Dazu ein Beispiel:

8 / 4 / 2 soll berechnet werden.

Bei Linksassoziativität erfolgt die Berechnung so:

(8 / 4) / 2 = 2 / 2 =1

Bei Rechtsassoziativität erfolgt die Berechnung so:

8 / (4 / 2) = 8 / 2 = 4

Es wurde sich für Linksassoziativität entschieden, da dies die Art der Abarbeitung ist, die allgemein vom Programmierer erwartet wird. Bei gleichen Vorrang wird also immer von links nach rechts abgearbeitet.

3.12 Linksrekursion oder Rechtsrekursion

Eine Grammatik kann linksrekursiv oder rechtsrekursiv formuliert werden.

Eine linksrekursive Formulierung ist zum Beispiel:

expr: expr + term
| term

Die entsprechende rechtsrekursive Alternative sieht dann so aus:

expr: term + expr
| term

Die Linksrekursion hat folgende Vorteile:

- Die Wahrscheinlichkeit des Stapel-Überlaufs wird reduziert.

- Das Prädikat wird automatisch linksassoziativ ausgewertet.

Die Rechtsrekursion besitzt dagegen folgenden Vorteil:

- Beim Top-Down-Parsen kommt man durch Linksrekursion in eine nichtabbrechende Rekursionsstruktur.

Da hier ein bottom-up-Parser genommen wurde (siehe später) und sich auch für Linksassoziativität entschieden wurde, wird die Grammatik linksrekursiv formuliert.

4 Aufbau der Prädikatverwaltungskomponente

In diesem Kapitel wird die Struktur der Prädikatverwaltungskomponente übersichtlich beschrieben. Es soll einen Überblick über die Strukturen und Aufgaben der Komponente verschaffen. Eine mehr ins Detail gehende Beschreibung findet sich im Kapitel 5.

4.1 Grobe Übersicht

Die Prädikatverwaltungskomponente hat die Aufgabe, die Bedingungen, die im Prädikat-Transitionsnetz stehen, zu überprüfen, ob diese wahr oder falsch sind. Sie bekommt eine Formel (ein syntaktisches Gebilde, welches die Elemente eines Prädikat enthält), interpretiert diese Formel zu einem Prädikat (fügt also Semantik hinzu) und bildet das Prädikat auf einen Wahrheitswert ab. Dieser Wahrheitswert wird dem ConTract-Manager zurückgeliefert. Außerdem entdeckt sie auch noch syntaktische und semantische Fehler und zeigt diese mit einer Fehlermeldung explizit an.

Die Prädikaverwaltungskomponente kann man grob in 2 Teile zerlegen:

1. Compiler oder Übersetzer
   Er übersetzt einen Prädikatstring in einen Zwischencode. Dies geschieht vor der Laufzeit eines ConTracts.
2. Evaluierungsfunktion
   Während der Laufzeit eines ConTracts wird die Evaluierungsfunktion jedesmal vom ConTract-Manager aufgerufen, wenn ein Prädikat im Prädikat - Transistions - Netz auf taucht. Sie liest den Zwischencode, evaluiert das Prädikat und liefert einen Wahrheitswert zurück.

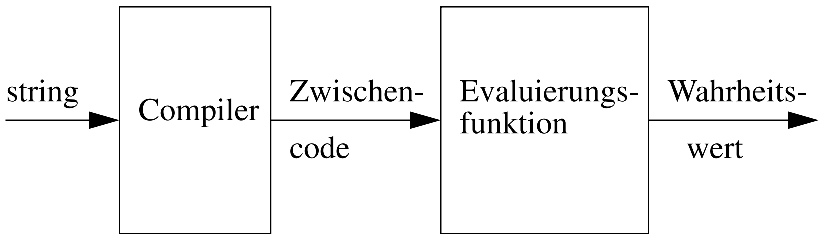


ABBILDUNG 5. Aufbau der Prädikatverwaltungskomponente

Die Übersetzung des Prädikatstrings in den Zwischencode erfolgt durch einen eigenen Compiler vor der Laufzeit eines ConTracts. Dieses Verfahren hat folgende Vorteile:

- Beim Übersetzen können schon mögliche Fehler durch den Compiler erkannt und dem Programmierer gemeldet werden. Dies ermöglicht dem Programmierer seine Fehler frühzeitig zu erkennen und so kann er sie schnell korrigieren.

- Die Übersetzung erfolgt schon vor der Laufzeit eines ConTracts, denn würde sie während der Laufzeit geschehen, so würde sie Zeit in Anspruch nehmen. Dies würde die Laufzeit des ConTracts sehr verlangsamen, und dieses sollte ja möglichst nicht passieren.
- Der Zwischencode ist durch den Compiler so modifiziert wurden, so daß Evaluierungsfunktion einige Berechnungen gar nicht mehr ausführen muß. Auch dieses bringt Zeitersparnis beim Evaluieren der Prädikate.
- Bei mehreren Ausführungen des gleichen ConTracts muß das Prädikat nicht jedesmal neu übersetzt werden.

4.2 Übersetzung

Die Übersetzung vom einem Prädikatstring zu einem Zwischencode erfolgt durch einen Compiler, der vom Editor aufgerufen wird. Das geschieht durch den Aufruf **ParseError parse_predicate(char *predicate, int cid, int pid)**. Der Parameter **predicate** enthält den Prädikatstring, **cid** enthält die Nummer des ConTracts und **pid** enthält die Nummer des Prädikats innerhalb eines ConTracts[1]. Die Funktion liefert einen Wert vom Typ **ParseError** zurück. **ParseError** ist ein Aufzählungstyp, der die Elemente **NOERR** und **ERROR** enthält. **ERROR** wird zurückgeliefert, falls der Compiler einen oder mehrere Fehler entdeckt hat, andernfalls wird **NOERR** zurückgegeben.

Zur Veranschaulichung eine grafische Darstellung:

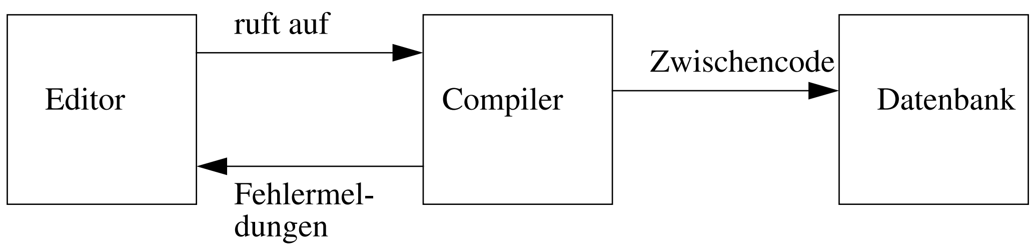


ABBILDUNG 6. Schnittstellen des Compilers

Der Compiler hat folgende Aufgaben:

- Vom Editor des ConTracts bekommt die Komponente einen String, der das Prädikat enthält. Dieser String wird lexikalisch und syntaktisch analysiert, ob er auch wirklich dem Syntax eines Prädikats entspricht. Falls nicht, werden Fehlermeldungen ausgegeben.

1. siehe [Wah91]

- Es werden Typkompatibilitätsprüfungen durchgeführt. Hier wird sich an der in Kapitel 3.8 dargestellten Tabelle orientiert. Wenn ein Fehler gefunden wird, so wird dieser gemeldet.
- Es wird überprüft, ob weitere Fehler vorhanden sind. Solche Fehler sind zum Beispiel ein zu langer Name eines Kontextelements oder wenn durch eine Konstante mit dem Wert 0 dividiert wird. Auch diese Fehler werden angezeigt.
- Wenn keine Fehler gefunden wurden, wird ein Zwischencode erstellt, der das Prädikat in handhabbare Einheiten zerlegt.
- Dieser Zwischencode wird optimiert. Diese Optimierung geschieht deshalb, damit die Evaluierungsfunktion möglichst keine überflüssigen Berechnungen machen muß.
- Der optimierte Zwischencode wird in einer SQL-Tabelle abgespeichert.

Die lexikalische Analyse erfolgt mit dem Scannergenerator flex, die syntaktische und semantische Analyse erfolgt mit dem Parsergenerator yacc. Eine genaue Beschreibung erfolgt im Kapitel 5.

4.3 Zwischencode

Der Zwischencode ist die Zerlegung des Prädikatstrings in handhabbare Einheiten. Er wird vom Compiler persistent abgespeichert und von der Evaluierungsfunktion gelesen. So stellt er die Schnittstelle zwischen der Übersetzung und der Evaluierung der Prädikate dar.

Eine Einheit des Zwischencodes ist ein Teil des gesamten Prädikatstrings. Dabei besteht eine Einheit aus einen Operator und ein oder zwei Operanden. Im Zwischencode wird eine solche Einheit Befehl genannt.

4.3.1 Syntaxbaum

Ein Syntaxbaum ist eine grafische Darstellung eines Zwischencodes. Dabei ist ein Syntaxbaum ein Binärbaum mit folgenden Eigenschaften:

- Operatoren sind mit den inneren Knoten assoziiert.
- Der linke Nachfolger ist der linke Operand.
- Der rechte Nachfolger ist entsprechend der rechte Operand.

Zur Veranschaulichung einige Beispiele für Syntaxbäume:

Beispiel 1:

3 + 4 == 7

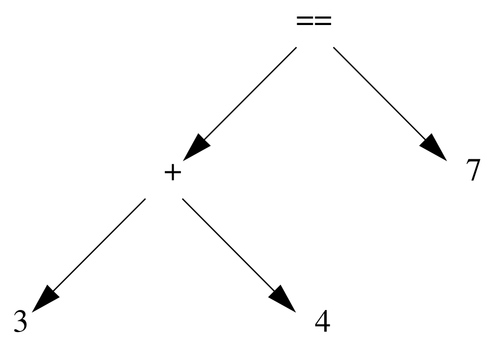


ABBILDUNG 7. Syntaxbaum für 3+4==7

Beispiel 2:

1 < 2 && ! 4 > 5

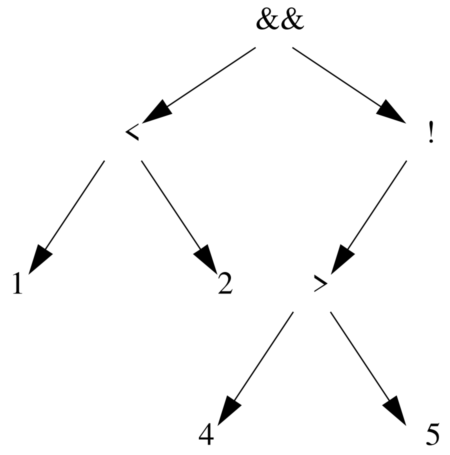


ABBILDUNG 8. Syntaxbaum für 1<2&&!4>5

Beispiel 3:

(1 + - 2) * 7 != ord (“a“)

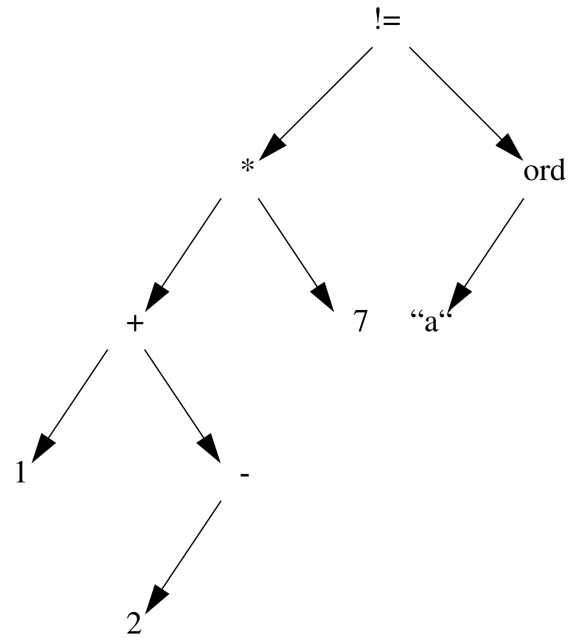


ABBILDUNG 9. Syntaxbaum für (1+-2)*7!=ord(“a“)

Warum wird als Zwischencode ein Syntaxbaum genommen?

- Ein Syntaxbaum spiegelt die natürliche, hierarische Struktur der Konstrukte in der Prädikatsprache wieder.
- Aus einem Syntaxbaum kann man sehr gut die Reihenfolge der Abarbeitung erkennen.
- Ein Syntaxbaum ist einfach und übersichtlich und damit ist er als Eingabe für die Evaluierungsfunktion gut geeignet, die ja nicht zu kompliziert sein soll.
- An einem Syntaxbaum kann man sehr leicht Optimierungen durchführen, man braucht dafür nur einige Zeiger umzusetzen.

4.3.2 Linearisierte Darstellung des Syntaxbaums

Ein Syntaxbaum ist eine Grafik und eine solche ist nicht leicht abspeicherbar. Aus diesem Grund wählt man eine linearisierte Darstellung der Daten, die ein solcher Syntaxbaum enthält.

Jeder Knoten des Baums, egal ob er nun ein Blatt oder ein innerer Knoten ist, entspricht einem Record und hat eine eindeutige Adresse. Der gesamte Baum ist dann ein ARRAY dieser Records.

Eine raffinierte und dennoch einfache Abbildung der Baumstruktur auf diese linearisierte Darstellung erfolgt so:

- Die Wurzel des Baumes hat die Adresse 1.
- Der linke Nachfolger des Knotens mit der Adresse i hat die Adresse 2*i.
- Der rechte Nachfolger des Knotens mit der Adresse i hat die Adresse 2*i+1.

Diese Art der Darstellung hat den Vorteil, daß man sehr schnell von einem Knoten zu seinen Nachfolgern kommen kann. Außerdem braucht man so in einem Knoten nicht die Adresse des linken und rechten Nachfolgers abzuspeichern, da sie ja nach der obigen Regel einfach berechnet werden kann.

Die im vorherigen Unterkapitel gezeichneten Unterbäume sehen linear nun so aus:

Beispiel 1:

3 + 4 == 7

Adresse	**Wert**
1	==
2	+
3	7
4	3
5	4

Beispiel 2:

1 < 2 && ! 4 > 5

Adresse	**Wert**
1	&&
2	<
3	!
4	1
5	2
6	>
12	4
13	5

Beispiel 3:

(1+ 2) *7 != ord (“a“)

Adresse	**Wert**
1	!=
2	*
3	ord
4	+

Adresse	**Wert**
5	7
6	“a“
8	1
9	-
18	2

Dieser Art von linearen Syntaxbaum kann nun sehr gut gespeichert und wieder gelesen werden.

Eine genaue Darstellung, wie dieser Zwischencode aufgebaut wird und wie er als SQL-Tabelle abgespeichert wird findet sich in Kapitel 5. Dieses Kapitel sollte lediglicht als Einführung und Übersicht dienen.

4.4 Evaluierungsfunktion

Die Evaluierungsfunktion ist nun endlich die Komponente, die das Prädikat auf einen Wahrheitswert abbildet. Der ConTract-Manager ruft die Funktion **Wahrheitswert eval_predicate(int cid, int pid, int ciid, int step)** auf, wenn er ein Prädikat im Prädikat-Tranisitionsnetz sieht. **cid** ist die Nummer des ConTracts, **pid** ist die Nummer des Prädikats, **ciid** ist die Nummer der ConTract-Instanz und **step** ist die Nummer derjenigen Stepinstanz, der das Prädikat im Prädikat-Transitionsnetz folgt[1]. Die Evaluierungsfunktion liefert ein Element vom Typ **Wahrheitswert** zurück. Dies kann entweder **TRUE** sein, falls das Prädikat wahr ist, oder **FALSE**, falls es falsch ist, oder **ERROR**, falls beim Evaluieren des Prädikat Fehler aufgetreten sind. Der ConTract-Manager kann aufgrund dieses Wahrheitswertes seinen Weg im Prädikat-Transitionsnetz wählen.

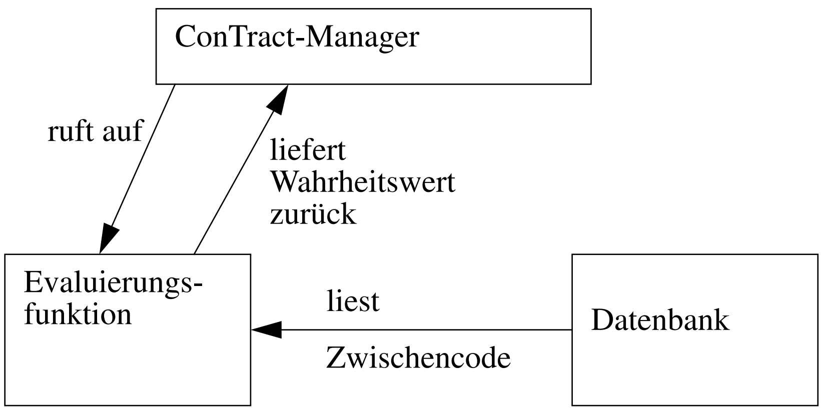


ABBILDUNG 10. Schnittstellen der Evaluierungsfunktion

1. siehe [Wah91]

Wie die Evaluierungsfunktion die Abbildung von einem Prädikat, welches als Syntaxbaum dargestellt ist, zu dem Wahrheitswert durchführt, wird in Kapitel 5 beschrieben.

5 Detaillierte Darstellung der Prädikatverwaltungskomponente

Dieses Kapitel erklärt die Prädikatverwaltungskomponente, also den Compiler, den Zwischencode und die Evaluierungsfunktion genauer. Es geht dabei in seinem Aufbau nach dem zeitlichen Ablauf der Teilkomponenten vor; Zuerst die lexikalische Analyse, dann die syntaktische Analyse, die semantische Analyse, der Zwischencodeerzeugung und -Optimierung und zum Schluß die Evaluierungsfunktion. Ein Teil der Prädikatverwaltungskomponente wurde implementiert.

5.1 Lexikalische Analyse

Bei der lexikalischen Analyse wird der Prädikatstring gelesen und in eine Strom von Tokens konvertiert, die wiederum die Eingabe der syntaktischen Analyse bilden. Ein Token stellt eine Folge von Zeichen dar, die zusammen eine bestimmte Bedeutung haben. Ein Token wird durch einen regulären Ausdruck beschrieben [ASU89].

Die Komponente, welche die lexikalische Analyse durchführt, heißt Scanner.

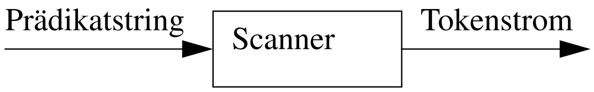


ABBILDUNG 11. Arbeitsweise eines Scanners

Der Scanner wurde mit Hilfe des Scanner-Generators **FLEX** erzeugt.

FLEX ist eine Weiterentwicklung des Scanner-Generators **LEX**. **FLEX** erhält als Eingabe eine Spezifikation der in der Sprache verwendeten regulären Ausdrücke. Daraus erzeugt **FLEX** ein C-Programm mit den Namen **lex.yy.c**, welches die Funktion **yylex()** enthält, die der Scanner ist.

5.1.1 Ausführen eines FLEX-Programms

Das Ausführen eines **FLEX**-Programms gliedert sich in 2 Schritte. Zuerst wird aus der Spezifikation der regulären Ausdrücke, dem flex-Quellcode, das C-file **lex.yy.c** erzeugt. Dies erfolgt durch den Aufruf:

flex flex-Quellcode

Im zweiten Schritt wird aus dem C-file ein ausführbares Programm gemacht. Dies geschieht durch das Kommando:

cc lex.yy.c -lfl

In dieser Studienarbeit ist der flex-Quellcode, also die Spezifikation der tokens, im file **regausdru**.

5.1.2 Aufbau eines FLEX-Programms

Ein flex-Quellcode gliedert sich in drei Teile:

- Definitionsteil
- Regelteil
- Benutzerroutinen

Die einzelnen Teile sind durch ein Doppelprozentzeichen '%%' voneinander abgegrenzt.

Der Definitionsteil.

Der Definitionsteil gliedert sich wiederum in zwei Teile. Der erste Teil enthält C-Preprozessor-Anweisungen, C-Variablen und C-Konstanten. Er ist von den Zeichen '%{' und '%{' umschlossen. Im file **regausdru** wird dieser Teil verwendet, um die Eingabe des Compilers, also den Prädikatstring, umzuleiten. Normalerweise wird die Eingabe aus 'standardinput' gelesen. Hier bekommt aber der Compiler die Eingabe als string vom Editor übergeben. Deshalb mußte die Eingabefunktion umgeschrieben werden. Die Eingabefunktion hat den Namen **YY_INPUT**. Sie ist nun so aufgebaut: Sie ruft in einer Schleife solange die Funktion **getNextChar()** auf, die das nächste Zeichen des Prädikatstrings liefert, bis das Endezeichen des Strings '\0' erreicht ist. Das jeweilig eingelesene Zeichen wird dann vom Scanner weiter verarbeitet.

Der zweite Teil des Definitionsteils besteht aus der Festlegung von praktischen Abkürzungen, die dann im Regelteil weiter verwendet werden können.

```
%{
#undef YY_INPUT
#define YY_INPUT(buf,result,max_size) \
        { \
        int c = getNextChar(); \
        result = (c == '\0') ? YY_NULL : (buf[0] = c, 1); \
        }
%}

letter [a-zA-Z]
digit [0-9]
msdigit [0-5]
hdigits (((0|1)[0-9])|(2[0-3]))
bruch „."[0-9][0-9]*
exp (E|e)(\+|\-)?[0-9][0-9]*
```

ABBILDUNG 12. Ausschnitt aus dem Definitionsteil

Regelteil.

Der Regelteil enthält Regeln, bestehend aus regulären Ausdrücken und Aktionen, die bestimmen, wie ein erkanntes Token zu verarbeiten ist.

Er ist dabei so aufgebaut:

regulärer Ausdruck 1	Aktion 1
regulärer Ausdruck 2	Aktion 2
...	...
regulärer Ausdruck n	Aktion n

Mit einem regulären Ausdruck werden Zeichenketten definiert, deren Vorkommen im Eingabestrom erkannt werden soll. Mit Aktionen wird festgelegt, was mit der erkannten Zeichenkette geschehen soll. Die Aktionen bestehen aus C-Anweisungen oder flex-spezifischen Arbeitsmakros. Jede Aktion muß dabei mit einem Semikolon enden, soll keine Aktion ausgeführt werden, so steht nur das Semikolon dort. Eine besondere Aktion ist das **return(token)**. Hier wird dem Parser, der die syntaktische Analyse durchführt, das erkannte token übergeben. Der Scanner stellt außerdem noch zwei Variablen zur Verfügung, die vom Parser verwendet werden können:

- **yytext** enthält die erkannte Zeichenkette
- **yyleng** enthält die Länge der Zeichenkette.

```
“||“                                        return(OR);
&&                                          return(AND);
!                                           return(NOT);
before                                      return(BEFORE);
at                                          return(AT);
after                                       return(AFTER);
\<                                          return(L);
“<=“                                        return(LEQ);
“==“                                        return(EQ);
“!=“                                        return(NEQ);
“>=“                                        return(GEQ);
“>“                                         return(G);
“+“                                         return(PLUS);
“-“                                         return(MINUS);
“*“                                         return(MAL);
“/“                                         return(GETEILT);
“%“                                         return(MUDOLO);
{letter}({letter}|{digit}|_)*               return(NAME);
{digit}({digit})*                           return(LONG_VALUE);
{digit}({digit})*({bruch})?({exp})?         return(FLOAT_VALUE);
{hdigits}:{msdigit}{digit}:{msdigit}{digit} return(TIME_VALUE);
\“([^\“])*\“                                return(STRING_VALUE);
.                                           ;
\n                                          ;
```

ABBILDUNG 13. Ausschnitt aus Regelteil

Benutzerroutinen.

Die Benutzerroutinen sind C-Funktionen, die vom FLEX-Programm verwendet werden. In diesem Fall sind keine solche Funktionen vorhanden.

5.2 Syntaktische Analyse

Die Aufgabe der syntaktischen Analyse oder Parsen besteht darin, die Tokens des Quellprogramms in hierarchische Sätze zusammenzufassen [ASU89]. Eine Darstellung der hierarischen Struktur dieser Sätze ist die eines Parse-Baums, wie das untenstehende Beispiel zeigt:

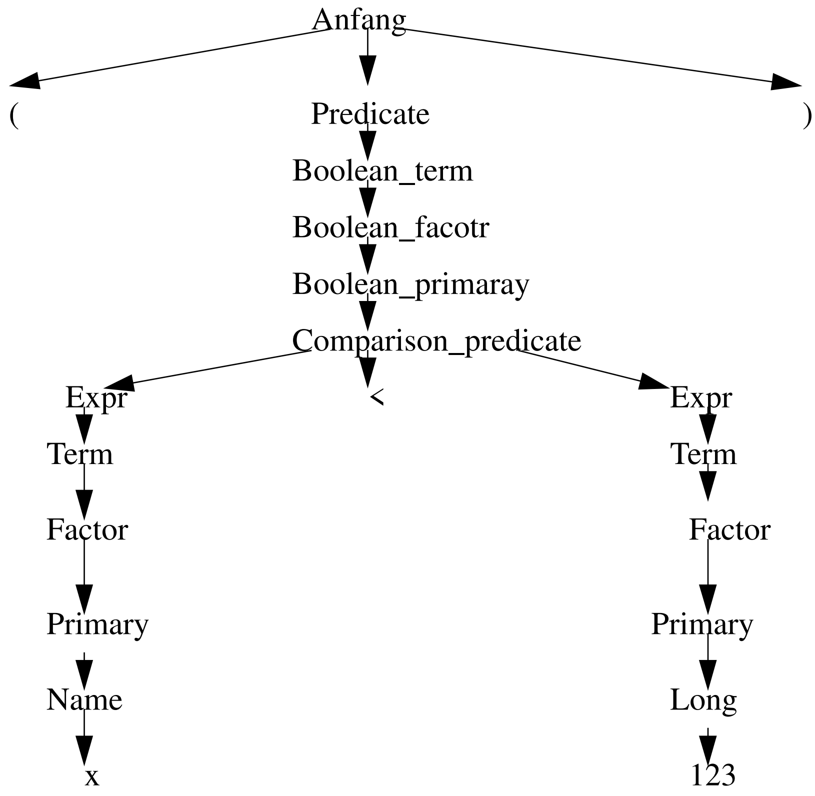


ABBILDUNG 14. Beispiel eines Parsebaums

Meistens wird ein Parse-baum nicht explizit erzeugt, sondern wird durch die Abarbeitungsreihenfolge beim Parsen vorgegeben.

Die Komponente, die die syntaktische Analyse durchführt, heißt Parser.

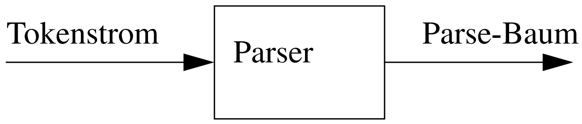


ABBILDUNG 15. Arbeitsweise eines Parsers

Der Parser wurde mit Hilfe des Werkzeugs **YACC** (**Yet another compiler compiler**) generiert. Er erzeugt aus einer kontextfreien Grammatikbeschreibung automatisch ein C-File mit den Namen **y.tab.c**, welches die Funktion **yyparse()** enthält. Diese ist für die syntaktische Analyse verantwortlich.

5.2.1 Ausführen eines YACC-Programms

Analog dem Ausführen eines **FLEX**-Programms besteht auch das Ausführen eines **YACC**-Programms aus zwei Teilen.

Im ersten Teil wird aus der Grammatikbeschreibung das C-file **y.tab.c** erzeugt. Dies geschieht durch den Befehl:

yacc grammatik

Im zweiten Teil muß aus dem C-file ein ausführbares Programm erzeugt werden. Dazu wird erstmal ein C-file gebraucht, welches ein Hauptprogramm enthält. Dort wird die Funktion **yyparse()** aufgerufen. Dieses C-file wird dann mit folgenden Kommando übersetzt:

cc Hauptprogramm -ly

Die Grammatikbeschreibung des Prototypen der Prädikatverwaltungskomponente befindet sich im File **ktxtfr**.

Das zugehörige Hauptprogramm hat den Namen **hu.c**.

5.2.2 Aufbau eines YACC-Programms

Eine **YACC**-Grammatikbeschreibung gliedert sich in vier Teile:

- optionaler Teil 1
- Deklarationsteil
- Regelteil
- optionaler Teil 2

Optionaler Teil 1.

Dieser Teil enthält C-Programmkonstrukte, Präprozessoranweisungen und globale Deklarationen von C-Variablen und C-Konstanten. Er ist von den Symbolen '%{' und '%}' umschlossen.

```
%{
#include"stdio.h"
#include"ctype.h"
#include"string.h"
#define Namelen 60        /*Laenge eines Namens*/
#define Typelen 60        /*Laenge eines Typs*/
```

```
#define Stringlen 60      /*Lange eines Strings*/
struct teilpr{
char operator[Operatorlen];      /*Nummer des Operator*/
int liOperand;                  /*Adresse des linken Operand*/
int reOperand;                  /*Adresse des rechten Operand*/
int hoehe;                      /*Hoehe des Baumes*/
char contextel_name[Namelen];    /*Name eines Kontextelements*/
char type[Typelen];             /*Ergebnistyp*/
};
%}
```

ABBILDUNG 16. Ausschnitt aus dem 1.optionalen Teil von yacc

Deklarationsteil.

Der Deklarationsteil enthält eine Liste der terminalen Symbole, die vom Scanner erkannt und dem Parser übergeben wurden. Dies geschieht mit der Syntax:

%token symbol1 symbol2....

Dadurch wird dem YACC die verschiedenen Tokens bekanntgemacht. Der Syntax eines Tokens hier muß mit dem Syntax des Return-Werts in der FLEX-Spezifikation übereinstimmen.

```
%token BEFORE AT AFTER
%token PLUS MINUS MAL GETEILT MUDOLO INT ORD CHR
%token L LEQ EQ NEQ GEQ G
%token OR AND NOT
%token NAME LONG_VALUE FLOAT_VALUE TIME_VALUE STRING_VALUE
```

ABBILDUNG 17. Ausschnitt aus dem YACC-Deklarationsteil

Der Deklarationsteil wird mit dem Doppelprozentzeichen ‘%%‘ beendet.

Regelteil.

Der Regelteil ist der wichtigste Teil eines YACC-Programms. Er enthält die Grammatikbeschreibung. Folgender Syntax wird dafür verwendet:

```
Nichtterminal 1  :  Regel 1
                 |  Regel 2
                 |  ...
                 ;

Nichtterminal 2  :  Regel 1
                 |  Regel 2
                 |  ...
                 ;
```

Dabei sind Regel 1, Regel 2,... Alternativen, von denen eine zur Anwendung kommt und das linksstehende Nichtterminal ersetzt.

Jede Regel ist eine Folge von nichtterminalen und terminalen Symbolen oder einzelnen Zeichen.

Nichtterminale Symbole, die in einer Regel auftauchen, müssen irgendwo nochmal auf der linken Seite als Nichtterminal auftauchen und werden dort dann weiter spezifiziert.

Terminale müssen vorher im Deklarationsteil als Token spezifiziert worden sein.

Einzelne Zeichen sind als terminale Symbole zu bewerten. Es wird für sie kein Symbolname vergeben und sie benötigen keine explizite Deklaration im Deklarationsteil.

Wichtig ist, daß jede Produktionsregel durch ein ';' abgeschlossen wird.

Nach jeder Regelalternative können eine oder mehrere semantische Aktionen stehen. Eine semantische Aktion ist ein C-Befehl, der ausgeführt wird, wenn die entsprechende Regelalternative angewendet wird. Mehrere semantische Aktionen, die zu einer Regelalternative gehören und hintereinander ausgeführt werden sollen, stehen hintereinander da und werden von geschweiften Klammern umschlossen. Die semantischen Aktionen dienen der semantischen Analyse, die in einem späteren Kapitel erklärt wird.

Der Regelteil wird wieder durch ein Doppelprozentzeichen '%%' beendet.

Optionaler Teil 2.

Der 2. optionale Teil enthält Präprozessoranweisungen und C-Funktionen.

Die wichtigste Anweisung, ist den Scanner einzubinden, da der Parser ja den Scanner benötigt. Also muß die Anweisung: #include "lex.yy.c" enthalten sein.

Die C-Funktionen sind Funktionen, die für die semantische Analyse benötigt werden. Im file **ktxtfr** sind das einige Hilfsfunktionen, die kleinere Aufgaben erledigen, Funktionen, die helfen, den Syntaxbaum aufzubauen, eine Fehlerbehandlungsfunktion, die aussagt, was im Falle eines Syntaxfehlers geschehen soll und eine Funktion, um den Zwischencode abzuspeichern.

5.2.3 Arbeitsweise eines mit YACC generierten Parsers

Ein Parser, der mit **YACC** erzeugt wurde, ist ein sogenannter LR(1)-Parser. Bei einem solchem Parser erfolgt die Syntaxanalyse bottom-up. Die bottom-up-Syntaxanalyse versucht, für einen Eingabestring einen Parse-Baum zu konstruieren, wobei an den Blättern (bottom) begonnen wird und sich zur Wurzel hochgearbeitet wird. Es ist also ein Prozeß, wo man einen Eingabestring, der nur aus Terminalen besteht, auf das Startsymbol der Grammatik - hier also das Symbol 'Anfang' - reduziert. Bei jedem Reduktionsschritt wird eine Zeichenfolge, die mit einer Zeichenfolge auf der rechten Seite einer Produktionsregel übereinstimmt, durch das zugehörige Nichtterminal auf der linken Seite ersetzt. Dies ist die Bildung einer umgekehrten Rechtsableitung.

Der Name LR(1) bedeutet, daß der Eingabestring von links nach rechts abgearbeitet wird und dabei eine umgekehrte Rechtsableitung gebildet wird. Außerdem wird immer 1 Zeichen vorrausgeschaut, um zu entscheiden, welche Regel angewendet wird.

Der Parser ist ein deterministischer Kellerautomat. Er kann folgende vier Operationen ausführen:

1. shift i
   Das terminale Eingabesymbol wird eingelesen und der Zustand mit der Nummer i wird auf dem Stack abgelegt. Danach geht der Parser in den neuen Zustand mit der Nummer i über.
2. reduce j
   Die reduce-Operation läßt sich in zwei Teiloperationen zerlegen: eine reine reduce-Operation und eine shift-Operation. Hat sich im Stack eine Sequenz gebildet, die reduzierbar ist, so wird diese Sequenz aus dem Stack genommen und gemäß Regel j in ein nichtterminales Symbol reduziert. Der Zustand des Kellerautomaten wird auf den Zustand zurückgesetzt, der vor Erkennung des ersten Teils der reduzierten Sequenz bestand (reine reduce-Operation). Das entstehende nichtterminale Symbol kann man sich nun als vor die Eingabe gesetzt vorstellen. Anschließend muß eine goto-Operation goto k ausgeführt werden, die analog einer shift-Operation für das entstandene nichtterminale Symbol einen Zustand mit der Nummer k auf dem Stack ablegt. Dabei wird ein neuer Zustand eingenommen, der nicht von der reduce-Operation, sondern von dem entstandenen nichtterminalen Symbol und dem Zustand vor der goto-Operation abhängt.
3. accept
   Der Eingabestring konnte vollständig abgearbeitet werden, das heißt es wurde bis zum Startsymbol der Grammatik reduziert und es sind dabei keine Fehler aufgetreten.
4. error
   Der Parser hat in der Liste der Eingabesymbole einen Fehler erkannt. Es handelt sich hier um einen syntaktischen Fehler, einen sogenannten Syntaxerror. Beim Erkennen eines solchen Eingabefehlers wird in den Fehlerbehandlungsmodus geschaltet. Es wird die Funktion **yyerror()** aufgerufen. In dieser Funktion wird angezeigt, daß ein Syntax error aufgetreten ist. Damit der Programmierer auch weiß, wo der Fehler passiert ist, wird auch noch das aktuelle Symbol angezeigt.
   Außer dem Aufruf der Funktion **yyerror()** wird noch der Stack freigegeben und **yyparse()** liefert den Return-Wert 1 für eine nicht erfolgreiche Bearbeitung; sonst wird 0 zurückgeliefert.

Die weitere Arbeitsweise[1] dieses Parsers kann man dem file **y.output** entnehmen. Es wird mit dem Befehl **yacc -v ktxtfr** erzeugt.

1. Weiteres über flex und yacc kann der angegebenen Literatur über lex und yacc [BDW87] [LMB92] entnommen werden.

5.3 Semantische Analyse

Die Phase der semantischen Analyse überprüft das Quellprogramm auf semantische Fehler und sammelt Typinformationen. Sie nutzt dabei die hierarische Struktur, also den Parse-Baum, die während der Syntaxanalyse erzeugt wurde. Das wichtigste Element der semantischen Analyse sind Typüberprüfungen. Hierbei überprüft der Compiler, ob die Operanden eines Operators von der Spezifikation der Quellsprache zugelassen sind. Dabei wird sich an der in Kapitel 3.8 vorgestellten Typkompatibilitätstabelle orientiert.

Die semantische Analyse befindet sich nicht in einer Extra-Datei, sondern ist in den semantischen Aktionen des Parsers implementiert.

5.3.1 Fehlerbehandlung

Nicht jedes eingegebene Prädikat ist korrekt. Die semantische Analyse ist dafür zuständig, die semantischen Fehler aufzuspüren und anzuzeigen. Die Behandlung der Syntaxerror erfolgt ja schon bei der syntaktischen Analyse.

Folgende semantische Fehler entdeckt der Compiler:

- type mismatch error
  Hier handelt es sich um Fehler, die der Typüberprüfung nicht stand halten, wo also Operanden eines nicht zugelassenen Typs auftreten.
- Name too long error
  Es wurde ein Kontextelementname angegeben, der zu lang ist, das heißt er hat mehr als 60 Zeichen.
- String too long error
  Analog dazu handelt es sich hier um eine Stringkonstante, die mehr als 60 Zeichen besitzt.
- Division by 0 error
  Dieser Fehler kann bei der Gleitpunktdivision oder bei der Modulo-Funktion auftreten. Er passiert, wenn der zweite Operand eine long-Konstante oder eine float-Konstante ist und diese den Wert 0 hat.
- Name not found error
  Wenn der Compiler einen Namen eines Kontextelements liest, so schaut er in der Kontextdatenbank [BuRa90] nach, welchen Typ dieses Element besitzt. Dies übernimmt die Funktion **get_type(int contract, char contextel_name[Namelen], char wert[Typelen])**, die sich in der Datei **find_type.sc** befindet. Dabei ist **contract** die Nummer des ConTracts, **contextel_name** der Name des Kontextelements und **wert** der Rückgabeparameter, der den Typ des Kontextelements enthält. Diese Funktion schaut in der Relation **context_variables** nach, ob die Kontextvariable vorhanden ist. Falls nicht, so wird die Fehlermeldung ausgegeben, sonst wird der Typ der Variablen zurückgegeben.

Die Ziele der Fehlerbehandlung sind folgende:

- Sie sollte verständlich, knapp, präzise und deutlich den Fehler melden.
- Der Programmierer sollte wissen, wo der Fehler aufgetreten ist.

- Die Fehlerbehandlung sollte sich möglichst schnell vom Fehler erholen, um Folgefehler entdecken zu können.

Ein gutes Beispiel für die Fehlerbehandlung in diesem Prädikatcompiler ist die Modulo-Funktion.

```
term MUDOLO factor   {
                         ... andere semantische Aktionen...
                   /*Typpruefung:*/
                  /*beide Typen muessen long sein, Ergebnis ist long */
                   if (strcmp(baum[$1].type,“long“)==0  &&
                      strcmp(baum[$3].type,“long“)==0)
                      {/*alles o.k.*/}

                  else /*Fehler*/
                  {Anzahl_Fehler++;
                  printf(„%s\n“,“ERROR“);
                  printf(„%s“,“type mismatch error at modulo \n“);
                  printf(„%s“,“Type must be long \n“);}

                 /*wenn der 2.Operand 0 ist, Fehler*/
                 if (strcmp(baum[$3].operator,“read long“)==0
                    && baum[$3].long_konstante==0)
                 { Anzahl_fehler++;
                   printf(„%s\n“,“ERROR“);
                   printf(„%s\n“,“division by 0 Error“);
                 }
                ...andere semantische Aktionen...
                }
```

ABBILDUNG 18. semantische Aktionen der Fehlerbehandlung für die Modulo-Funktion

Es wird zuerst überprüft, ob beide Operanden vom Typ long sind, da die Modulo-Funktion nur Operanden vom Typ long zuläßt. Wenn nun einer der Operanden nicht diesem Typ entspricht, so wird ein type mismatch error ausgegeben. Damit der Programmierer weiß, wo der Fehler aufgetreten ist, wird angezeigt, daß er bei modulo gefunden wurde. Zur weiteren Unterstützung des Programmierers kommt noch zusätzlich die Erklärung, daß der Typ long sein muß.

Als zweites wird überprüft, ob ein Divisionsfehler durch 0 vorhanden ist. Dieser tritt auf, wenn der zweite Operand den Wert 0 hat. Wird nun dieser Fehler festgestellt, so wird eine Fehlermeldung ausgegeben, daß ein Division durch 0 Fehler festgestellt wurde.

Bei beiden Fehlerarten wird eine Variable hochgezählt, die aussagt, wieviel Fehler gefunden wurden. Dieses wird am Schluß des Compilierens dem Programmierer auch noch angezeigt. Wenn am Schluß die Variable größer als 0 ist, also mindestens ein semantischer Fehler entdeckt wurde, so gibt **yyparse()** den Return-Wert 1 für eine nicht erfolgreiche Bearbeitung zurück, andernfalls wird der Return-Wert 0 zurückgeliefert.

Damit beim Compilieren nicht nur der Fehler angezeigt wird, der gerade gefunden wurde, sondern auch weitere Fehler entdeckt werden können, wird beim Auftreten eines Fehlers nicht angehalten. Im Gegenteil, es wird für die weitere Compilierung so getan, als wäre der Fehler gar nicht vorhanden. So können dann weitere Fehler gefunden werden.

5.4 Zwischencodeerzeugung

Auch die Zwischencodeerzeugung findet nicht in einer Extra-Datei statt, sondern ist in den semantischen Aktionen des Parsens eingebunden. Wie im Kapitel 4 bereits beschrieben, wird als Zwischencode ein Syntaxbaum aufgebaut.

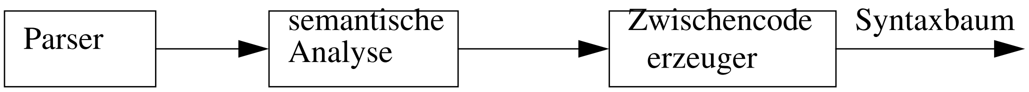


ABBILDUNG 19. Einbettung der Zwischencodeerzeugung

5.4.1 Aufbau des Syntaxbaums

Der Syntaxbaum wird durch die semantischen Aktionen während des Parsens aufgebaut. Dabei wird er zuerst im Arbeitsspeicher vollständig aufgebaut und später persistent abgespeichert. Diese Methode hat den Vorteil, daß der Syntaxbaum im Arbeitsspeicher einfach bearbeitet, zum Beispiel auch optimiert, werden kann.

Da später ja eine linearisierte Darstellung des Syntaxbaums herauskommen soll, wird der Baum nicht explizit mit Zeigern aufgebaut, sondern er ist ein Array of Record. Dabei entspricht jedes Element dieses Arrays einen Knoten im Syntaxbaum. Der Index eines Arrayelements entspricht dabei der Adresse des Knotens des Baums.

Der Aufbau des Baumes geschieht mit Hilfe von Attributen. Ein Attribut ist eine Angabe, die mit einem Grammatiksymbol assoziiert ist. Ein Attribut kann für jede mögliche Angabe stehen, zum Beispiel für einen Typ, einen Wert, einen Speicherplatz oder was auch immer sonst relevant ist. In diesem Prädikatcompiler wird pro Grammatiksymbol nur ein Attribut benötigt, dieses enthält die Adresse des Knotens im Syntaxbaum. Zu jeder Produktion gehören eine Menge von semantischen Aktionen. Mit Hilfe einiger dieser semantischen Aktionen werden die Attribute berechnet, die in der jeweiligen Produktion vorkommen. Es kann zwei Arten von Attributen geben:

- zusammengesetzte oder synthetisierte Attribute
  Ein Attribut heißt zusammengesetzt, wenn der zugehörige Wert für einen Knoten des Parse-Baums aus den Attributwerten der Nachfolger dieses Knotens bestimmt werden kann. Zusammengesetzte Attribute berechnet man, indem der Parse-Baum von unten nach oben, also bottom-up durchlaufen wird.
- ererbte Attribute
  Ein ererbtes Attribut ist ein Attribut, dessen Wert aus den Werten der Vorgänger und/oder aus den Werten der Nachbarknoten berechnet wird.

In dem vorliegenden Prädikatcompiler wird zusammengesetzte Attributierung verwendet. Diese hat gegenüber der ererbten Attributierung einen erheblichen Vorteil. Da die syntaktische Analyse bottom-up erfolgt und die Auswertung der zusammengesetzten Attribute ebenso, kann die syntaktische Analyse und die Berechnung der Attribute beim gleichen Parse-Baum-Durchlauf geschehen. So sind synthetisierte Attribute einfach und schnell zu berechnen.

Verwendet werden YACC-eigene Attribute. YACC stellt für jede Regel die Attribute $1,$2,$3,.... zur Verfügung. Jedes dieser Attribute ist mit einem Grammatiksymbol auf der rechten Regelseite verknüpft. $1 ist mit dem ersten Grammatiksymbol auf der rechten Seite verknüpft, $2 mit dem zweiten Grammatiksymbol und so weiter.... Außerdem gibt es noch das Zeichen $$. Dieses bezeichnet das dem nichtterminalen Symbol auf der linken Produktionsseite zugeordneten Wert, und wird in der semantischen Aktion berechnet. Zusammengefaßt kann man sagen, für jede Regel gilt: Das zusammengesetzte Attribut $$, welches dem nichtterminalen Grammatiksymbol auf der linken Regelseite zugeordnet ist, wird durch die der Regel zugehörigen semantischen Aktion berechnet, dabei können die Attribute $1,$2,... verwendet werden, die mit Grammatiksymbolen auf der rechten Regelseite assoziiert sind.

Im allgemeinen bleibt dem Benutzer verborgen, wie YACC die Berechnung der Attribute intern realisiert. YACC besitzt für die Attribute einen Wertestack, analog dem Stack der der Sprachverarbeitung dient. In diesem werden die Werte der Attribute gespeichert. Bei einer Reduktion, also wenn eine Regel zur Anwendung kommt und dadurch auch durch die semantischen Aktionen ein synthetisiertes Attribut berechnet werden soll, so wird einer globalen Variablen mit dem Namen **yyval** der passende Wert zugeordnet.

Wie wird nun der Syntaxbaum aufgebaut?

Der Syntaxbaum ist ein Array of Record. Ein Array-Element besteht aus folgenden Record-Komponenten:

```
struct teilpr{
char operator[Operatorlen];      /*Operator*/
int liOperand;                   /*Adresse des linken Operand*/
int reOperand;                   /*Adresse des rechten Operand*/
int hoehe;                       /*Hoehe des Baumes*/
char contextel_name[Namelen];    /*Name eines Kontextelements*/
long long_konstante;             /*Longkonstante*/
float float_konstante;           /*floatkonstante*/
char string_konstante[Stringlen];/*Stringkonstante*/
char time_konstante[Timelen];    /*Timekonstante*/
char type[Typelen];              /*Ergebnistyp*/
};
```

Der eigentliche Aufbau ist nun ganz einfach. Wird ein neues Element dem Syntaxbaum angefügt, welches kein Blatt ist, so wird zunächst ein Zähler hochgezählt, um die neue Adresse des Knotens zu generieren. Diese Adresse wird dem zusammengesetzten Attribut $$ zugewiesen. Dann werden die Adresse des linken und rechten Operanden eingetragen, diese sind die Attribute der zugehörigen Grammatiksymbole auf der rechten Seite. Dies geschieht durch die Funktion **einf**. Der Operator und der Ergebnistyp dieser Regel werden in die entsprechenden Komponenten kopiert.

```
expr : expr PLUS term    { /*Aufbau des Baums:*/
                         zaehler=zaehler+1;$$=zaehler;
                         einf($$,$1,$3);
                         strcpy(baum[$$].operator,“+“);
                          ... andere semantische Aktionen}
```

ABBILDUNG 20. Beispiel für Einfügen eines inneren Knotens in den Syntaxbaum

Das Einfügen eines Blatts erfolgt analog. Als Adresse des linken und rechten Nachfolgers wird 0 eingetragen. Dafür ist die Funktion blatt verantwortlich. Wenn hier abzuspeichernde Namen oder Konstanten auftauchen, so werden sie in die entsprechenden Komponenten kopiert.

```
primary : FLOAT_VALUE     {zaehler=zaehler+1;$$=zaehler;
                          blatt($$);
                          strcpy(baum[$$].operator,“read float“);
                          baum[$$].float_konstante=atof(yytext);
                          strcpy(baum[$$].type, „float“);
                           }
```

ABBILDUNG 21. Beispiel für Einfügen eines Blatts in den Syntaxbaum

5.5 Codeoptimierung

Der erzeugte Syntaxbaum kann noch so verbessert werden, so daß die Evaluierungsfunktion beschleunigt wird. Diese Verbesserung kann durch Transformationen erreicht werden, die man Optimierungen nennt. Die Komponente des Compilers, die solche Transformationen ausführt, wird Optimierer genannt; der verbesserte Code wird optimierter Code genannt. Ein Compiler, der einen Optimierer enthält, heißt optimierender Compiler.

ABBILDUNG 22. Einbettung des Optimierers

Die optimierenden Transformationen sollten einige Eigenschaften erfüllen:

- Eine Optimierung soll die durchschnittliche Geschwindigkeit deutlich erhöhen. In diesem Fall heißt das, die Evaluierungsfunktion soll im Durchschnitt mit dem optimierten Zwischencode schneller ablaufen, als ohne die Optimierung. Natürlich ruft nicht jede Transformation eine Verbesserung hervor, es kann passieren, daß eine Optimierung mal in dem einen oder anderen Fall keine Beschleunigung bringt, es aber im Durchschnitt schneller wird.

- Eine Optimierung sollte die Bedeutung des Codes nicht verändern. Dies bedeutet, daß mit Optimierung bei der gleichen Eingabe nicht eine andere Ausgabe erzeugt wird als ohne die Optimierung. Die Wirkung eines Programms darf also nicht verändert werden, im Zweifelsfall sollte auf eine Transformation eher verzichtet werden.
- Außerdem muß sich der Aufwand für eine Optimierung sich lohnen. Es ist nicht effektiv, sich viel Mühe mit code-verbessernden Transformationen zu geben, die sich zur Laufzeit nicht auszahlen. Die Beschleunigung muß den Aufwand rechtfertigen.
- Selten benutzte Teile im Code sollten nicht optimiert werden. Hier lohnt sich der Aufwand des Optimierens nicht, da die Teile ja selten gebraucht werden und somit der Beschleunigungseffekt fast nie eintritt. Hier rechtfertigt der Nutzen den Aufwand für die Transformationen also nicht.

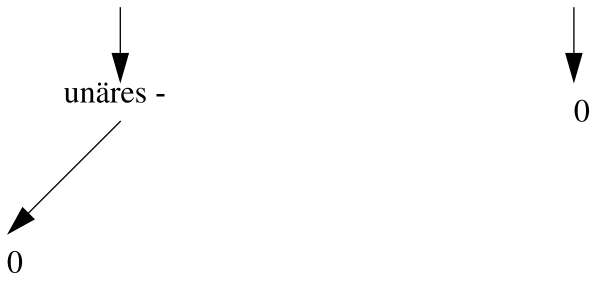
5.5.1 Algebraische Optimierungen

Eine wichtige Klasse der Optimierungen stellen algebraische Optimierungen dar. Sie sind relativ einfache und einsichtige Transformationen, und rechtfertigen deshalb auch den Aufwand. Darum werden sie auch in diesem Prädikatscompiler durchgeführt. Als Grundlage dieser Art von Optimierungen gelten algebraische Gleichungen.

Im folgenden wird dargestellt, welche algebraischen Optimierungen durchgeführt wurden. Dabei steht jeweils die zugrundeliegende algebraische Gleichung darüber, links der nicht-optimierte Syntaxbaum und rechts der optimierte Syntaxbaum.

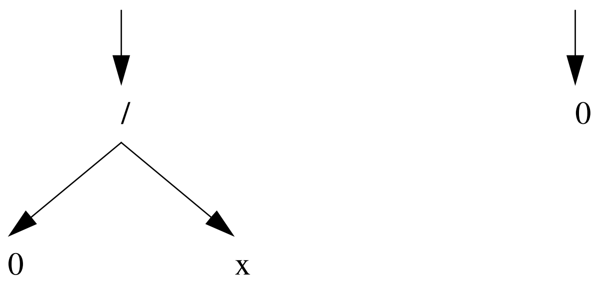
1. Optimierung:

$$-0 = 0$$



2. Optimierung:

$$0 / x = 0$$



Die weiteren Optimierung sind nicht mehr grafisch dargestellt, sie erfolgen analog.

3. Optimierung:

$$0 * x = 0$$

4. Optimierung:

$$x * 0 = 0$$

5. Optimierung:

$$0 - x = -x$$

6. Optimierung:

$$x - 0 = x$$

7. Optimierung:

$$0 + x = x$$

8. Optimierung:

$$x + 0 = 0$$

9. Optimierung:

$$0 \% x = 0$$

Diese Art von Transformationen verkürzen den Syntaxbaum um ein bis zwei Knoten. Dieses hat den Effekt, daß die Evaluierungsfunktion jene Knoten nicht mehr durchlaufen muß, dadurch wird sie schneller und eine Beschleunigung tritt ein.

5.5.2 Optimierende short-circuit-evaluation

In diesem Kapitel wird eine weitere Methode gezeigt, um die Laufzeit der Evaluierungsfunktion zu verbessern. Dabei werden die Besonderheiten bei UND- und ODER-Verknüpfungen ausgenutzt. Bei diesen Verknüpfungen brauchen in bestimmten Fällen nicht beide Teilprädikate ausgewertet werden, sondern es reicht, wenn nur das erste überprüft wird. Ob dieser günstige Fall eintritt hängt von dem Ergebnis des ersten Teilprädikats ab.

Im einzelnen sieht das so aus:

Verknüpfung	**Ergebnis Teilprädikat 1**	**was passiert**
UND	TRUE	Teilprädikat 2 muß evaluiert werden
UND	FALSE	Teilprädikat 2 muß nicht evaluiert werden, Gesamtergebnis = FALSE
ODER	TRUE	Teilprädikat 2 muß nicht evaluiert werden, Gesamtergebnis = TRUE
ODER	FALSE	Teilprädikat 2 muß evaluiert werden

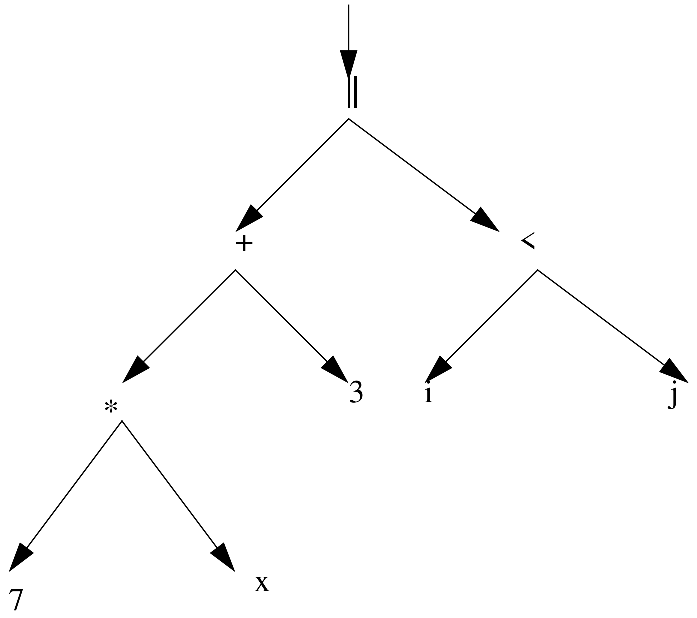
Wie es sehr gut zu sehen ist, braucht nur in der Hälfte der Fälle das 2. Teilprädikat evaluiert zu werden. Das bringt eine enorme Zeitersparnis, denn im Durchschnitt gesehen ist die Gesamtevaluierungszeit einer solchen Verknüpfung nur noch halb so lang. Diese Art der Evaluierung heißt **short-circuit-evaluation**. Da sie so effektiv ist, wird sie auch hier in der Prädikatverwaltungskomponente angewendet.

Der Zwischencode kann aber daraufhin optimiert werden, so daß die short-circuit-evaluation noch schneller vor sich geht. Bis jetzt ist das 1. Teilprädikat, welches ja immer ausgewertet werden muß, immer der linke Unterbaum im Syntaxbaum oder das zuerst eingelesene Prädikat gewesen. Doch die UND- oder ODER-Verknüpfungen sind kommutative Operationen, die Operanden können also vertauscht werden. Das bedeutet, die Prädikate können so getauscht werden, daß das kürzere Teilprädikat immer das erste ist. So wird nur das kürzere Teilprädikat immer evaluiert, während das längere nur in etwa der Hälfte der Fälle ausgewertet werden muß. Dies bringt wiederum Zeitersparnis.

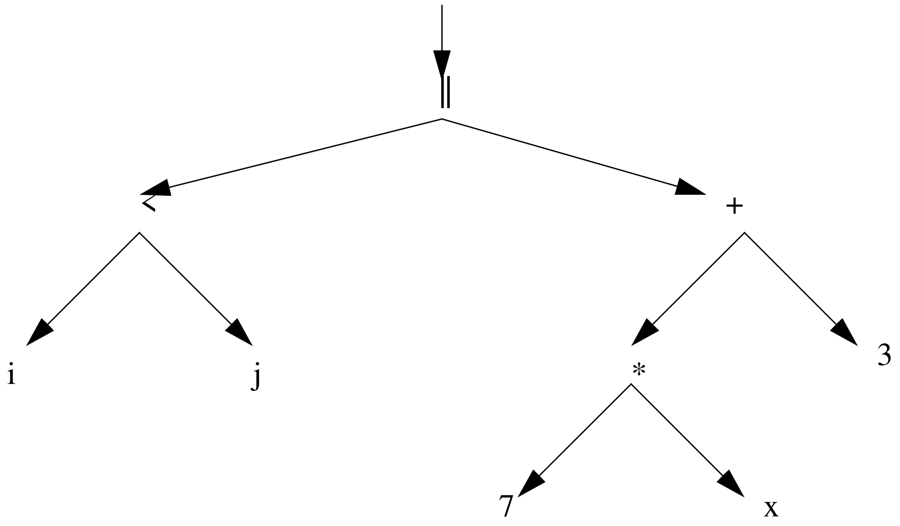
Die Optimierung, die im Zwischencode gemacht werden muß, um diese Beschleunigung zu erreichen, ist auch nicht besonders aufwendig. Im Syntaxbaum wird an der Stelle, wo eine UND- oder ODER-Verknüpfung auftaucht, der höhere Unterbaum - dieser entspricht dem längeren Teilprädikat - nach rechts getauscht. Der rechte Unterbaum ist derjenige, der als zweites ausgewertet wird, und durch diese Maßnahme ist damit sichergestellt, daß das längere Teilprädikat als zweites evaluiert wird.

Dazu ein Beispiel:

Das Prädikat **7 * x + 3 > y || i <j** soll evaluiert werden. Ohne die Optimierung würde der Syntaxbaum so aussehen:



Mit der Optimierung sieht der Syntaxbaum dann so aus :



Durch den optimierenden Compiler muß der Programmierer nun mehr aufpassen. Er kann sich nun nicht darauf verlassen, daß das Teilprädikat, welches er als erstes eingegeben hat, auch als erstes ausgewertet wird. Das ist ein Nachteil dieser Methode. Da aber die Zeitersparnis durch die Optimierung so offensichtlich ist, muß der negative Aspekt in Kauf genommen werden.

5.6 Zwischencode

Der Zwischencode stellt die Schnittstelle zwischen Compiler und Evaluierungsfunktion dar. Gespeichert wird er persistent in einer Datenbank.

ABBILDUNG 23. Rolle des Zwischencodes

5.6.1 Schreiben des Zwischencodes durch den Compiler

Der Prozeß des Abspeichern geht folgendermaßen vor sich:

- Zuerst wird geschaut, ob der Compiler keine Fehler entdeckt hat. Wenn Fehler gefunden wurden, ist es sinnlos den Zwischencode des fehlerhaften Prädikats persistent zu sichern.
- Nehmen wir an es sind keine Fehler aufgetreten, und das Prädikat könnte abgespeichert werden. Nun wird in der Datenbank nachgeschaut, ob ein Prädikat für das gleiche ConTract-Skript mit der gleichen Prädikatnummer vorhanden ist. Falls ja, so ist dieses offensichtlich veraltet, die neue Version ist das aktuelle Prädikat. Das veraltete Prädikat wird nicht mehr gebraucht und deswegen gelöscht. Danach erscheint eine Meldung, daß das frühere Prädikat gelöscht wurde.
- Die Funktion **abspeichern** wird aufgerufen. Diese produziert aus dem Syntaxbaum im Arbeitsspeicher einer linearisierte Darstellung des Syntaxbaums, wie im Kapitel 4.3.2 beschrieben.
- Die Funktion **abspeichern** ruft nun zum tatsächlichen Abspeichern in der Datenbank die Funktion **insert_predicate** auf. Diese befindet sich in einem Extra-file mit dem Namen **pre_insert.sc**. Nun wird das Prädikat wirklich in die Datenbank geschrieben. Sollten bei dem Vorgang Fehler auftauchen, so wird eine Fehlermeldung angezeigt.
- Nachdem der Zwischencode geschrieben wurde, ist die Aufgabe des Compilers beendet. Falls das Compilieren des Prädikates fehlerfrei war, wird der Return-Wert 0 für eine erfolgreiche Bearbeitung zurückgeliefert und der Lauf des Compilers beendet.

5.6.2 Beschreibung des Zwischencodes

Der Zwischencode wird in einer SQL-Relation abgespeichert. Dieses Verfahren wurde gewählt, da:

- Der andere Zwischencode des ConTract-Systems auch in SQL-Tabellen gespeichert ist. So ist der gesamte Zwischencode in dem gleichen Ort vorhanden, und er ist schnell auffindbar.
- Das Datenbankmanagementsystem garantiert die Integrität und Konsistenz der gespeicherten Daten.

So befindet er sich in der Relation **predicate** in der Datenbank **apricots**. Die Relation hat folgenden Aufbau:

TABELLE 5. Aufbau der Relation predicate

Name	Typ
Cid	integer
Predica-teno	integer
Adr	integer
Operator	char(60)
Type	char(60)
Litype	char(60)
Retype	char(60)
Contextel _name	char(60)
Long_kon stante	integer
Float_kon stante	float
String_ko nstante	char(60)
Time_kon stante	char(60)

Die Cid ist eine eindeutige Nummer des ConTracts, zu dem das Prädikat gehört. Predicateno ist eine Nummer des Prädikat innerhalb des ConTracts. Adr ist die Adresse eines Knotens im Syntaxbaum. Operator zeigt an, welcher Operator oder Funktion dieser Knoten besitzt. Type ist der Ergebnistyp der Operation, Litype und Retype sind die die Typen der linken und rechten Nachfolger. Contextel_name ist der Name eines zu lesenden Kontextelements; Long_konstante, Float_konstante, String_konstante und Time_konstante sind die Werte der entsprechenden Konstanten.

Die Werte für Litype, Retype, Contextel_name, Long_konstante, Float_konstante, String_konstante und Time_konstante können auch NULL sein, falls der entsprechende Wert nicht vorhanden ist.

Primärschlüssel der Relation ist die Kombination Cid, Predicateno und Adr - das Datenbanksystem achtet auf die Eindeutigkeit.

Zur Verdeutlichung ein Beispiel. Der ConTract habe die Nummer 82 und das Prädikat habe die Nummer 6. Prädikat ist **(7*6+3>11||5<6)**.

Folgender Zwischencode erscheint:

Cid	Predica-teno	Adr	Operator	Type	Litype	Retype	Long_konstante
82	6	1	‖	boolean	boolean	boolean	
82	6	2	<	boolean	long	long	
82	6	3	>	boolean	long	long	
82	6	4	read long	long			5
82	6	5	read long	long			6
82	6	6	+	long	long	long	
82	6	7	read long	long			11
82	6	12	*	long	long	long	
82	6	13	read long	long			3
82	6	24	read long	long			7
82	6	25	read long	long			6

Die weiteren Spalten enthalten bei diesem Beispiel keine Elemente und sind daher nicht weiter relevant.

Die Wurzel dieses Syntaxbaums hat die Adresse 1 und den Operator ‖. Sie liefert ein Ergebnis vom Typ boolean und hat einen linken und rechten Operanden, die beide vom Typ boolean sind.

An diesem Beispiel sieht man recht deutlich, daß am Knoten mit dem Operator ‖ der rechte und der linke Unterbaum getauscht wurden, so daß rechts der längere Unterbaum hängt (der rechte Unterbaum hat nun die Wurzel mit der Adresse 3).

5.7 Evaluierungsfunktion

Die Evaluierungsfunktion wird vom ConTract-Manager aufgerufen und liefert einen Wahrheitswert zurück. Dabei liest sie den Zwischencode aus der Datenbank.

Die Evaluierungsroutine besteht aus mehreren rekursiven Funktionen, die sich gegenseitig aufrufen. Der Aufrufsablauf entspricht einem postorder-Durchlauf des Syntaxbaums. Bei einem postorder-Durchlauf wird zuerst der linke Unterbaum abgearbeitet, dann der rechte Unterbaum und zum Schluß die Wurzel. In diesem Fall bedeutet das folgendes: Zuerst wird der linke Operand berechnet, dann - falls vorhanden - der rechte Operand und zum Schluß wird aufgrund der Werte der beiden Operanden und des Operators das Ergebnis für diesen Teilausdruck ausgerechnet. Dieses wird dann an die aufrufende Funktion zurückgeliefert. Die Berechnung des linken und rechten Operanden erfolgt durch weitere Aufrufe von Evaluierungsfunktionen.

Im großen und ganzen wird sich an diese Ausführung gehalten, doch es gibt es eine Ausnahme: die short-circuit-evaluation. An der Stelle wird der rechte Operand nur ausgewertet, wenn es erforderlich ist. So verkürzt sich die Laufzeit der Evaluierungsfunktion.

Am deutlichsten wird die Arbeit der Evaluierungsroutinen anhand eines Beispiel. Im folgenden wird die Verarbeitung anhand des Beispiels im Kapitel 5.6.2 gezeigt:

1. Der ConTract-Manager ruft die Evaluierungsfunktion auf. Dabei teilt er ihr mit, daß es sich um den ConTract mit der Nummer 82 und das Prädikat mit der Nummer 6 handelt.
2. Die Evaluierungsfunktion liest die Wurzel des Syntaxbaum. Das ist per Definition die Zeile mit der Adr 1. Diese Zeile wird nun per SQL-Abfrage eingelesen.
   Der linke Operand soll nun ausgewertet werden. Also ruft die Evaluierungsfunktion sich selbst rekursiv mit der Adresse der linken Operanden, in diesem Fall 2, auf.
3. Bei dieser Inkarnation wird die entsprechende Zeile eingelesen und es wird festgestellt, das es sich um einen kleiner-Vergleich handelt. Um den Wert des linken Operanden zu bekommen wird eine Evaluierungsfunktion mit der Adresse des linken Nachfolgers (also 4) aufgerufen, die ein Ergebnis vom Typ long zurückliefert.
4. Diese Funktion liest den Wert 5 ein und gibt ihn der aufrufenden Funktion zurück.
5. Nun ist man wieder auf der selben Ebene wie im Punkt 3. Da die kleiner-Operation zwei Operanden benötigt, sollte auch der zweite Operand geholt werden. Deshalb wird wieder die Evaluierungsfunktion aufgerufen, die ein Ergebnis vom Typ long liefert, diesmal mit der Adresse des rechten Nachfolgers, nämlich 4.
6. Diese Funktion liest den Wert dieser Konstante, nämlich 6. Da hier im Syntaxbaum keine weiteren Nachfolger existieren, werden keine weiteren Funktionen aufgerufen, sondern diese Inkarnation wird beendet und der Wert 6 wird zurückgeliefert.
7. Wieder auf der gleichen Rekursionsstufe wie bei Punkt 3 angekommen, kann nun der tatsächliche kleiner-Vergleich geschehen, da jetzt beide Operanden vorhanden sind. Es wird ausgewertet, daß 5 wirklich kleiner als 6 ist, und so wird der Wahrheitswert TRUE zurückgeliefert.
8. Jetzt befindet sich der Ablauf wieder auf der gleichen Rekursionsebene wie bei Punkt 2. Da es sich hier um eine ODER-Verknüpfung handelt, und der linke Operand den Wert TRUE hat, kommt hier die short-circuit-evaluation zum Zuge. Der rechte Operand braucht nicht mehr evaluiert zu werden, das Gesamtergebnis ist TRUE. Dieses Gesamtergebnis wird der aufrufenden Funktion zurückgegeben.
9. Nun ist man wieder auf der obersten Rekursionsebene. Das Ergebnis ist TRUE, dieses kann nun dem ConTract-Manager zurückgeliefert werden, und die Evaluierungsfunktion wird verlassen.

6 Ein ausführliches Beispiel

Zur Verdeutlichung hier noch ein ausführliches Beispiel, bei dem alle Verarbeitungsschritte ausführlich dargestellt werden.

Es soll das Prädikat **((x+0)*7 > 22.3 && before (12:00:00))** evaluiert werden. Dabei soll x ein Kontextelement vom Typ long mit dem Wert x=10 sein.

Der dazugehörige Parsebaum ist folgender:

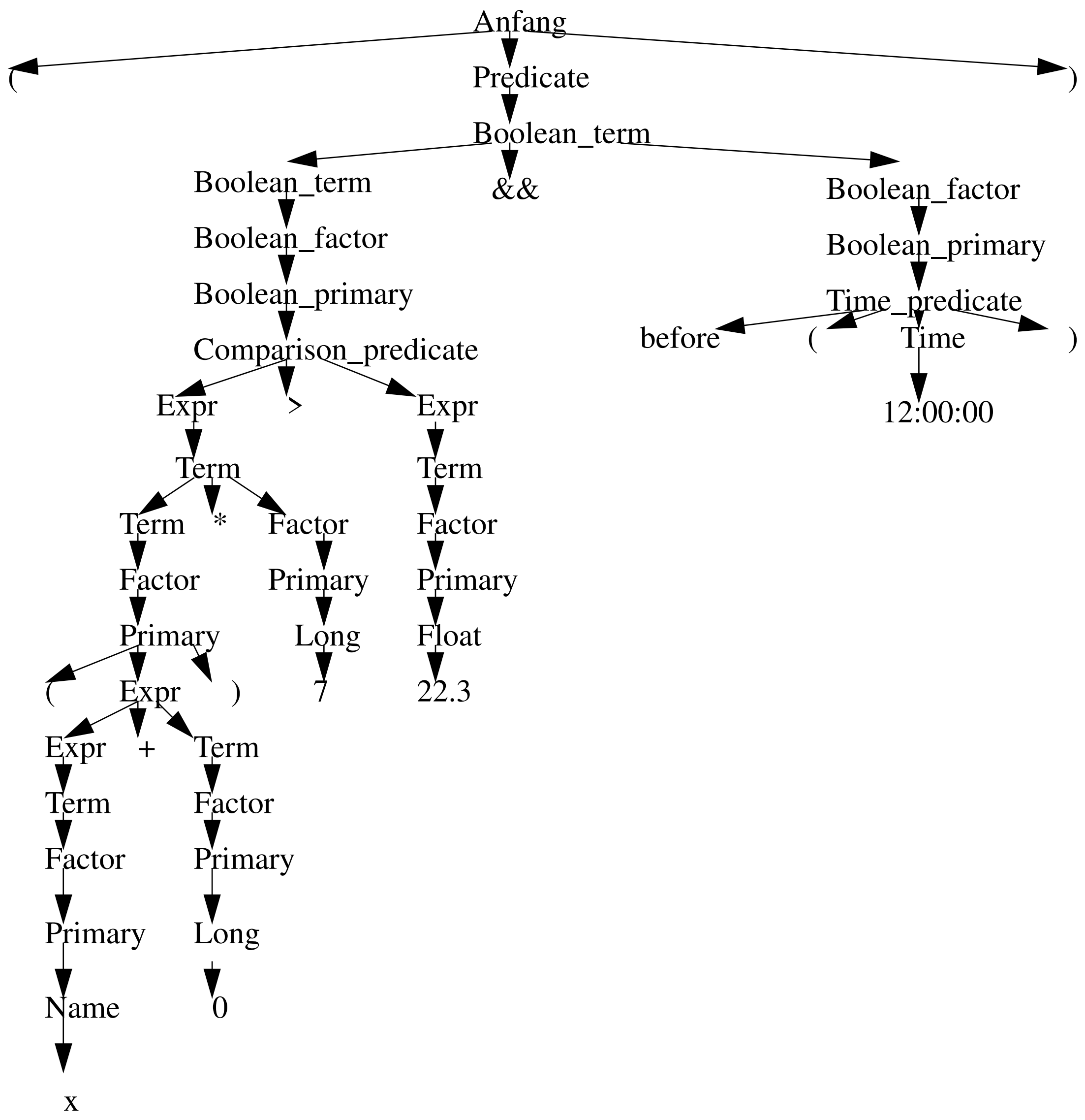


ABBILDUNG 24. Parsebaum für ((x+0)*7>22.3&&before(12:00:00))

Der Parser compiliert durch das Prädikat entlang der Struktur des Parse-Baums. Dabei sucht er nach Fehlern. In diesem Beispiel findet er keine,da

1. Syntaxfehler keine vorhanden sind.

2. Typkompatibilitätsfehler gibt es auch nicht,da

- Die Addition ist eine Ganzzahladdition, beide Operanden sind vom Typ long, das Ergebnis ist long
- Die Multiplikation ist eine Ganzzahlmultiplikation, beide Operanden sind vom Typ long, das Ergebnis ist long.
- Der größer-Vergleich stellt auch kein Problem dar. Es handelt sich um einen Vergleich von einem long-Wert mit einem float-Wert, dies ist problemlos durch implizite Typkonversion möglich.
- Die Und-Verknüpfung verknüpft zwei boolean-Werte.

3. weitere Fehler existieren nicht.

Außer der Fehleranalyse baut der Compiler auch noch einen Syntaxbaum aus dem Prädikat auf. Dieser sieht dann so aus:

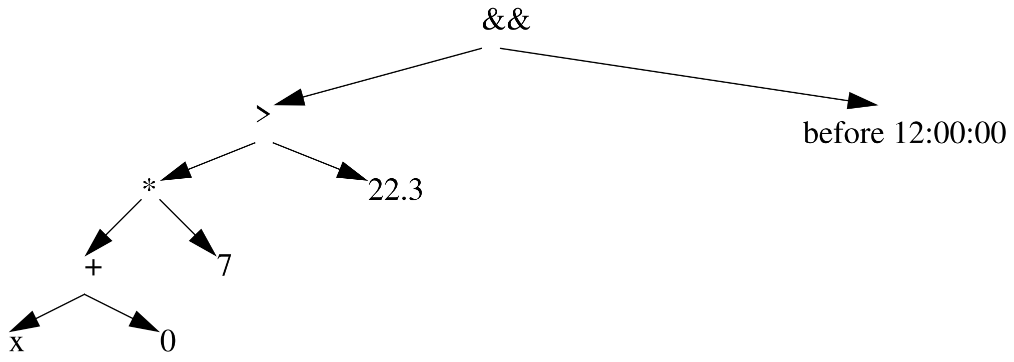


ABBILDUNG 25. Syntaxbaum für ((x+0)*7>22.3&&before(12:00:00))

Dazu eine Erläuterung: Klammern sind nur dazu da, die hierarische Struktur zu verdeutlichen, sie werden überlesen und im Syntaxbaum nicht mehr benötigt. Deshalb können sie einfach weggelassen werden.

Dieser oben gezeigte Syntaxbaum ist noch nicht optimiert. Als erste Optimierung kann die algebraische Gleichung **x+0=x** angewendet werden.

Dann gibt es folgenden Syntaxbaum:

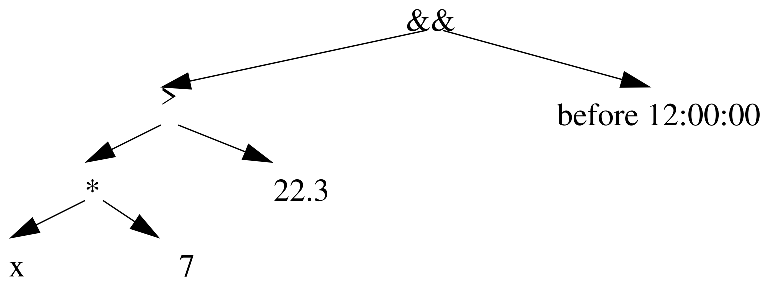


ABBILDUNG 26. teilweise optimierter Syntaxbaum für ((x+0)*7>22.3&&before(12:00:00))

Noch eine weitere Optimierung ist möglich. Um die short-circuit-evaluation zu verkürzen wird der höhere Unterbaum der &&-Verknüpfung nach rechts getauscht.

Hier nun vollständig optimierte Syntaxbaum:

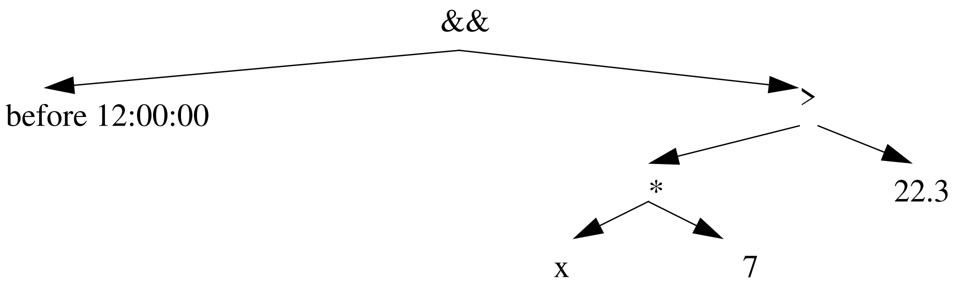


ABBILDUNG 27. optimierter Syntaxbaum für ((x+0)*7>22.3&&before(12:00:00))

Nun kann der Zwischencode als SQL-Tabelle abgespeichert werden. Dazu wird angenommen, der ConTract habe die Nummer 12 und das Prädikat die Nummer 5. Dann sieht der Zwischencode folgendermaßen aus:

TABELLE 6. Zwischencode für ((x+0)*7>22.3&&before(12:00:00))

Cid	Predica-teno	Adr	Operator	Type	Litype	Retype	Contextel_name
12	5	1	&&	boolean	boolean	boolean	
12	5	2	before	boolean			
12	5	3	>	boolean	long	float	
12	5	6	*	long	long	long	
12	5	7	read float	float			
12	5	12	read con-textele-ment	long			x
12	5	13	read long	long			

TABELLE 7. Zwischencode für ((x+0)*7>22.3&&before(12:00:00) Fortsetzung

Cid	Predica-teno	Adr	Long_konstante	Float_konstante	String_konstante	Time_konstante
12	5	1				
12	5	2				12:00:00
12	5	3				
12	5	6				
12	5	7		22.300		
12	5	12				
12	5	13	7			

Jetzt kann die Evaluierungsfunktion tätig werden. bei geht sie in postorder durch den Baum durch und berechnet die Werte. Dabei geht sie folgendes Ablaufschema entlang:

1. Sie evaluiert den Knoten mit der Adresse 2. Dabei holt sie sich die aktuelle Kalenderzeit vom Betriebssystem und vergleicht sie mit der Konstante 12:00:00. Wenn es zur Zeit früher als 12.00.00Uhr ist, so kommt TRUE heraus, sonst FALSE.
2. Sie bemerkt, daß es sich hier um eine UND-Verknüpfung handelt. War das vorherige Teilprädikat FALSE, so braucht die Evaluierungsfunktion nichts mehr zu tun. Sie liefert den Wert FALSE zurück (short circuit evaluation). War es dagegen TRUE, so muß das zweite Teilprädikat evaluiert werden.
3. Hierfür wird als erstes das Kontextelement x gelesen (Adr=12). Der Wert von x wird von der Kontextdatenbank geholt durch die Funktion **get_var_value (int ciid, int step, char contextel_name[Namelen],char typbezeichner[Typelen], union wertemenge *p_to_var)**. Dabei ist **ciid** die Instanznummer des ConTracts, **step** die Instanznummer des Steps, **contextel_name** der Name der Variablen, **typbezeichner** der Typ der Variablen und **p_to_var** der Rückgabeparameter, der auf den Wert der Kontextvariablen zeigt. In diesem Fall ist dieser Wert 10.
4. Als nächstes wird die Longkonstante mit den Wert 7 gelesen (Adr=13).
5. Dann erfolgt eine Multiplikation (Adr=6) von 10 und 7. Als Ergebnis kommt 70 heraus.
6. Als nächstes wird die Gleitkommakonstante mit dem Wert 22.3 gelesen (Adr=7).
7. Es folgt der größer-Vergleich (Adr=3). Hier wird festgestellt, daß 70 größer als 22.3 ist, und somit ist das Ergebnis TRUE.
8. Da das erste Teilprädikat der Und-Verknüpfung TRUE war, und das zweite Teilprädikat auch TRUE ist, ist die gesamte Und-Verknüpfung auch TRUE und damit auch das gesamte Prädikat. Also wird der Wahrheitswert TRUE zurückgeliefert.

7 Zusammenfassung und Ausblick

Es sollte eine prototypische Prädikatverwaltungskomponente für das ConTract-Verarbeitungssystem entwickelt werden.

Dafür wurde zuerst eine Sprache definiert, die den Syntax und die Semantik der Prädikate beschreibt. Hierbei wurde besonders auf die Benutzerfreundlichkeit geachtet.

Dann wurde ein Teil der Prädikatverwaltungskomponente prototypisch entwickelt. Die Komponente teilt sich in zwei Teile: den Compiler, der einen Prädikatstring einliest und diesen in einen Zwischencode übersetzt und die Evaluierungsfunktion, die aus dem Zwischencode einen Wahrheitswert berechnet. Dabei wurde vor allem auf die Effizienz der Evaluierungsfunktion geachtet. Diese Prädikatverwaltungskomponente wurde implementiert.

Die ganze Studienarbeit beschäftigt sich nur mit einem Prototypen. Dieser kann ausgebaut werden.

Möglichkeiten der Erweiterungen wären:

- Die im Prädikat verwendeten Variablen und Konstanten sollten auch noch andere Typen besitzen. Zum Beispiel: In Zeiten, wo die objektorientierte Programmierung immer wichtiger wird, sollte diese Erweiterung enthalten sein.
- Die Grammatik selbst kann um andere Operationen und Funktionen erweitert werden, da die hier entwickelte nur die Grundlegenden enthält. Ein Beispiel wären bitorientierte Vergleiche, damit der Programmierer auch hardwarenah programmieren kann. Ein anderes Beispiel sind Funktionen, die nicht nur auf die Zeit, sondern auch auf das Datum zugreifen.
- Es könnte noch viel mehr Optimierungsmaßnahmen beim Compilieren geben, zum Beispiel die Suche nach gemeinsamen Teilausdrücken.
- Der Compiler könnte Berechnungen mit Konstanten schon vor der Laufzeit auswerten. Das würde während der Laufzeit Zeitersparnis bringen.
- Eine weitere interessante Erweiterung ist, verschiedene Prädikate, die einem Step folgen, parallel zu evaluieren. Dies würde einen weiteren Effizienzgewinn verursachen.
- Eine andere Alternative ist, innerhalb der Prädikate parallel zu evaluieren. Dies bedeutet, der eine Prozeß evaluiert das erste Teilprädikat, der zweite Prozeß das zweite Teilprädikat usw... Für lange Prädikate wäre dies eine bedenkenswerte Alternative, denn dadurch könnte die Evaluierungszeit solcher Prädikate extrem verkürzt werden.

A Grammatik der Prädikatsprache

Zur Beschreibung der Grammatik wird die im Kapitel 3.1 vorgestellte Metasprache angewendet.

```
ANFANG : ( PREDICATE )

PREDICATE : PREDICATE ‘||‘ BOOLEAN_TERM
          | BOOLEAN_TERM

BOOLEAN_TERM : BOOLEAN_TERM && BOOLEAN_FACTOR
             | BOOLEAN_FACTOR

BOOLEAN_FACTOR : ! BOOLEAN_PRIMARY
               | BOOLEAN_PRIMARY

BOOLEAN_PRIMARY : COMPARISON_PREDICATE
                | TIME_PREDICATE
                | ( PREDICATE )

TIME_PREDICATE : before ( TIME )
               | at ( TIME )
               | after ( TIME )

TIME : TIME_VALUE

COMPARISON_PREDICATE : EXPR < EXPR
                     | EXPR <= EXPR
                     | EXPR == EXPR
                     | EXPR != EXPR
                     | EXPR >= EXPR
                     | EXPR > EXPR

EXPR : EXPR + TERM
     | EXPR - TERM
     | TERM

TERM : TERM * FACTOR
     | TERM / FACTOR
     | TERM % FACTOR
     | FACTOR

FACTOR : - PRIMARY
       | int ( PRIMARY )
       | ord ( PRIMARY )
       | chr ( PRIMARY )
       | PRIMARY

PRIMARY : NAME
```

| FLOAT
| LONG
| STRING
| (EXPR)
| ARRAY[EXPR]
| RECORD.KOMPONENTE

ARRAY : NAME

RECORD : NAME

KOMPONENTE : NAME

NAME : LETTER[LETTER|DIGIT|_]*

FLOAT : DIGIT[DIGIT]*[.DIGIT[DIGIT]*][{E|e}[+|-]DIGIT[DIGIT]*]

LONG : DIGIT[DIGIT]*

STRING : "{ASCII-Zeichen}*"

TIME_VALUE : HDIGITS : MSDIGIT DIGIT : MSDIGIT DIGIT

HDIGITS : 0DIGIT | 1DIGIT | 2{0|1|2|3}

MSDIGIT : 0|1|2|3|4|5

LETTER : a|b|...|z|A|B|...|Z

DIGIT : 0|1|2|3|4|5|6|7|8|9

Literaturverzeichnis

[ASU89] Aho, Sethi, Ullmann: *Compilerbau*, Addison-Wesley, 1989.

[BDW87] Bach, Domann, Weng-Beckmann: *UNIX Handbauch zur Programmentwicklung, Kapitel 11: Programmgeneratoren*, Hanser Verlag, 1987.

[BuRa90] Holger Burkert, Bernd Raichle: *Entwicklung eines Daten- und Verarbeitungsmodell für den Kontext*, Studienarbeit Nr. 859, 1990.

[KeRi90] Kernighan, Ritchie: *Programmieren in C, Zweite Ausgabe ANSI C*, Hanser Verlag, 1990.

[LMB92] John R. Levine, Tony Mason, Doug Brown: *lex & yacc*, O'Reilly & Associates INC., Second Edition, 1992.

[Mas90] Ioannis Masmanidis: *Syntax und Compiler für eine Sprache zur Spezifikation verteilter und paralleler Abläufe*, Studienarbeit Nr. 897, 1990.

[Ri85] Lutz Richter: *Betriebssysteme*, Teubner-Verlag, 1985.

[Schö87] Uwe Schöning: *Logik für Informatiker*, Band 56, BI-Wissenschaftverlag, Reihe Informatik, 1987.

[Sei95] Jens Seifert: *Ein Programmierwerkzeug für APRICOTS*, Studienarbeit Nr. 1422, 1995.

[WäRe91] Helmut Wächter, Andreas Reuter: *The ConTract Model*, technischer Bericht, 1991.

[Wah91] Peter Wahl: *Interpretative Abwicklung von Prädikat-Transitionsnetzen*, Studienarbeit Nr.1021, 1991.

[X/O91] X/OPEN Priliminary Specification: *Structured Query Language (SQL)*, 1991.