**Diplomarbeit**

Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme

# Syntaktische Erweiterbarkeit von Programmiersprachen unter Benennungs-, Bindungs- und Typisierungsinvarianzen

vorgelegt von:
**Gerald Schröder**
Am Schützenplatz 14
21218 Seevetal

Betreuer:
Prof. Dr. J. W. Schmidt
Dr. M. Lehmann

Hamburg, den 17. Februar 1994

*Some are born to move the world —*
*To live their phantasies.*
*But most of us just dream about*
*The things we'd like to be.*
Rush, „Losing it", 1982

Für meine Eltern, die mir die Erfüllung meiner Träume ermöglicht haben.

Bei der Erfüllung dieses Traums standen mir noch viele andere Menschen zur Seite, denen ich hier danken möchte. Insbesondere gilt mein Dank Dr. Florian Matthes für die mündlichen und schriftlichen Diskussionen, in denen sich die Kernpunkte dieser Arbeit herauskristallisierten. Prof. Dr. Schmidt möchte ich für die Unterstützung und das besondere Interesse am Thema dieser Arbeit danken. Dr. Lehmann danke ich dafür, daß er mich auf einige Unstimmigkeiten der schriftlichen Version aufmerksam machte. An softwaretechnischen Grundlagen danke ich Dr. Florian Matthes für das Tycoon-System, Bernd Mathiske für die 'Tycoon-Machine', Luca Cardelli für das Quest-System und Rainer Müller für die 'Persistent Quest-Machine'. Andreas Rudloff half mir bei allen Problemen mit der vorhandenen Software. Andreas Gawecki beantwortete geduldig meine Fragen über Parser-Generatoren. Dominic Juhasz war der erste Anwender der in dieser Arbeit beschriebenen Syntaxerweiterungen und gab mir interessante Verbesserungsvorschläge. Bei allen Angehörigen des Arbeitsbereichs DBIS möchte ich mich für den zeitweisen Stillstand der Rechner während meiner Testläufe entschuldigen.
Hamburg, den 17. Februar 1994 *Gerald Schröder*

# Inhaltsverzeichnis


**1 Erweiterbare Sprachen und Datenbanksysteme** **1**

1.1 Programmiersprachen und Datenbanksysteme . . . . . 1

1.2 Historie erweiterbarer Programmiersprachen . . . . . 9

1.3 Ziel dieser Arbeit . . . . . 12

1.4 Aufbau dieser Arbeit . . . . . 13

**2 Syntaxerweiterungen im Compilermodell** **14**

2.1 Compilermodell . . . . . 14

2.2 Beschreibungstechniken und Generatoren . . . . . 17

2.3 Integration der syntaktischen Erweiterbarkeit in das Compilermodell . . . . 23

2.4 Beschreibung der syntaktischen Erweiterung . . . . . 25

2.5 Beschreibung der semantischen Interpretation . . . . . 27

2.6 Bindungsprobleme in Syntaxerweiterungen . . . . . 35

2.7 Implementation mit Compilerbau-Werkzeugen . . . . . 35

**3 Basissprache TLMin und Erweiterungssprache TLExt** **37**

3.1 Basissprache TLMin . . . . . 37

3.2 Erweiterungssprache TLExt . . . . . 43

3.3 Auswahl einer Sprachversion zur Übersetzung . . . . . 53

**4 Anwendung von Syntaxerweiterungen** **55**

4.1 Standardkonstrukte . . . . . 55

4.2 Normalisierungen . . . . . 63

4.3 Datenbankspezifische Konstrukte . . . . . 67

**5 Bindungen in Syntaxerweiterungen** **74**

5.1 Ansätze zur Lösung von Bindungsproblemen . . . . . 75

5.2 Ein neuer Ansatz zur Lösung von Bindungsproblemen . . . . . 76

5.3 Klassifikation von Bindungen in Syntaxerweiterungen . . . . . 82

5.4 Globale Bindungen in der Definition . . . . . 85

5.5 Lokale Bindungen in der Definition . . . . . 86

5.6 Globale Bindungen in Platzhaltern . . . . . 88

5.7 Implementation . . . . . 89

**6 Zusammenfassung und Ausblick** **91**

6.1 Zusammenfassung . . . . . 91

6.2 Ausblick . . . . . 92

**A Abstrakte und konkrete Syntax von TLMin und TLExt** **95**

A.1 Konkrete Syntax . . . . . 95

A.2 Abstrakte Syntax . . . . . 99

**B Statische Semantik** **105**

B.1 Notation . . . . . 105

B.2 Bindung und Sortenkonformität von TLExt . . . . . 106

B.3 Bindungsphase von TLMin . . . . . 114

# Kapitel 1

# Erweiterbare Sprachen und Datenbanksysteme

In diesem Kapitel werden in Abschnitt 1.1 verschiedene Ansätze zur Integration von Programmiersprachen und Datenbanksystemen vorgestellt. Aus der Diskussion der Vor- und Nachteile der Ansätze leitet sich die Forderung nach syntaktischer Erweiterbarkeit von Programmiersprachen als ein Hilfsmittel zur Integration mit Datenbanksystemen ab. Diese Forderung bildet den Ausgangspunkt dieser Arbeit über die syntaktische Erweiterbarkeit von Programmiersprachen.

„Erweiterbare Programmiersprachen" waren in den 60er Jahren und Anfang der 70er Jahre Forschungsthema der Informatik. Die Anzahl der Veröffentlichungen zu diesem Thema hat seitdem rapide abgenommen, so daß die Forschung abgeschlossen scheint. Was für ein Ziel verfolgt eine Arbeit über erweiterbare Programmiersprachen heutzutage? Um diese Frage zu beantworten, wird in Abschnitt 1.2 eine historische Einordnung dieser Arbeit vorgenommen, um damit die „Wiederentdeckung" des Themas „Erweiterbare Programmiersprachen" zu begründen. Ein Überblick über die Arbeit schließt das Kapitel ab.

## 1.1 Programmiersprachen und Datenbanksysteme

Die Programmierung datenintensiver Anwendungen ist ohne Unterstützung durch die Datenbankkonzepte Transaktionsverwaltung, Massendatentypen mit Anfrage- und Manipulationsoperationen sowie persistente Datenspeicherung aufwendig und fehlerträchtig [Schlageter, Stucky 83, Kap. 1.2 und 9.3]. Typischerweise unterstützen Anwendungsprogrammiersprachen diese Konzepte nicht, während Datenbanksysteme selten über eine algorithmisch vollständige Programmiersprache verfügen. Zur Verknüpfung von Anwendungsprogrammiersprachen und Datenbanksystemen existieren folgende Ansätze (vgl. [Atkinson, Bunemann 87, Kap. 2] und [Blaser et al. 87, Kap. 6.3]):

**Datenbanksprachen:** Spezielle Sprachen zur Beschreibung datenbankspezifischer Operationen, die nicht algorithmisch vollständig sind. Beispiele für Datenbanksprachen sind SQL [SQL 87], POSTQUEL [Stonebraker, Rowe 86], QUEL [Stonebraker et al. 76].

**Datenbankprogrammiersprachen:** Algorithmisch vollständige Sprachen mit zusätzlichen datenbankspezifischen Operationen. Beispiele für Datenbankprogrammiersprachen sind DBPL [Schmidt, Matthes 92], Galileo [Albano et al. 85], RIGEL [Rowe, Shoens 79], PS-Algol [Atkinson et al. 81], PLAIN [Wasserman et al. 81].

**Bibliotheken:** Datenbankspezifische Module (z. B. [CODASYL 78]), die an Programmiersprachen angebunden werden. Beispiele für Sprachen, die sich für die Anbindung von Bibliotheken eignen, sind Tycoon [Matthes 93], Modula-3 [Nelson 91], Napier-88 [Dearle et al. 89], Eiffel [Meyer 88].

Im folgenden werden die Ansätze exemplarisch anhand von SQL, DBPL und Tycoon diskutiert. Die Programmbeispiele zeigen die Lösung folgender Aufgabe:

> Zeige die Namen aller Personen aus der Gesamtmenge von Personen, deren Alter einen bestimmten Wert überschreitet.

### 1.1.1 SQL — eine Datenbanksprache

SQL [Date 89] ist eine Datenbanksprache, d. h. sie unterstützt nur datenbankspezifische Konzepte. Syntax und Semantik der Sprache sowie die Anbindung an einige Programmiersprachen sind in einem Standard [SQL 87] festgelegt.

SQL ist nicht algorithmisch vollständig und somit nicht zur Anwendungsprogrammierung geeignet. Die Verbindung mit einer Anwendungsprogrammiersprache erfolgt durch Einbettung von SQL-Anweisungen als Zeichenketten in Anwendungsprogramme (*embedded SQL*). Die SQL-Anweisungen können besonders gekennzeichnete Variable aus der Anwendungssprache enthalten.

In Beispiel 1.1.1 wird die Lösung der oben genannten Aufgabe in C mit eingebetteten SQL-Anweisungen gezeigt. Es handelt sich um eine C-Funktion, die das Alter *age* als Parameter erhält. In der Funktion wird eine lokale Variable *p* vom Typ *Person* angelegt. Außerdem wird zur Iteration über die Menge der Personen *persons* ein Datenbankzeiger *personCursor* erzeugt. Innerhalb der Deklaration des Datenbankzeigers wird die Variable *age* aus der Domäne der C-Variablen benutzt, die mit einem Doppelpunkt gekennzeichnet ist, um sie von SQL-Variablen zu unterscheiden. Nach der Öffnung des Datenbankzeigers wird der erste Datensatz in die C-Variable *p* transferiert. Solange ein bestimmter Fehlerkode *sqlca.sqlcode* keinen Fehler anzeigt, wird der Name der Person ausgegeben und der nächste Datensatz geladen. Als Abschluß der Funktion wird der Datenbankzeiger wieder geschlossen.

**Beispiel 1.1.1:** Lösung der Aufgabe in C mit eingebetteten SQL-Anweisungen

```
void printNames(int age)
{
  Person p;
  EXEC SQL DECLARE personCursor CURSOR FOR
    SELECT p FROM persons p WHERE p.age > :age;
  EXEC SQL OPEN personCursor;
  EXEC SQL FETCH personCursor INTO :p;
  while (sqlca.sqlcode == 0) {
    printf("%s\n", p.name);
    EXEC SQL FETCH personCursor INTO :p;
  };
  EXEC SQL CLOSE personCursor;
};
```

Die eingebetteten SQL-Anweisungen werden von einem speziellen Compiler (*pre-compiler*) in Anweisungen der Anwendungssprache, typischerweise Aufrufe eines Datenbanklaufzeitsystems, umgesetzt.

Folgende Vorteile und Probleme kennzeichnen diesen Ansatz:

- Die Einbettung von SQL in Anwendungsprogrammiersprachen ist einfach und führt zu einer hohen Verfügbarkeit.
- SQL enthält Konzepte zur Benennung, Bindung und Typisierung, die unabhängig von den entsprechenden Konzepten der Anwendungsprogrammiersprache sind. SQL-Anweisungen und Anweisungen aus der Anwendungsprogrammiersprache lassen sich nur eingeschränkt mischen. Das führt zu einem konzeptuellen „Bruch“ (*mismatch*), der die Verwendungsmöglichkeiten einschränkt und Fehler begünstigt [Schmidt, Matthes 93].
  Ein Beispiel für diesen Konzeptbruch sind die getrennten Namensräume von C und SQL: SQL-Objekte können innerhalb von C-Anweisungen nicht benutzt werden, da ihre Namen nicht bekannt sind. Innerhalb von SQL-Anweisungen können C-Objekte verwendet werden, aber ihre Namen müssen durch einen Doppelpunkt gekennzeichnet sein. So bezeichnet im folgenden Beispiel der Name *p* das durch **FROM** *persons p* eingeführte Objekt aus der SQL-Domäne, während *:p* für das durch die Deklaration *Person p;* eingeführte Objekt aus der C-Domäne steht:

  ```
  Person p;
  EXEC SQL ... SELECT p FROM persons p WHERE p.age > :p.age;
  ```

- SQL bietet einige ausdrucksstarke, datenbankspezifische Anweisungen, der Sprach- und Funktionsumfang ist aber bei weitem nicht befriedigend [Date 89]. Neue Anforderungen führen zu fortlaufenden, langwierigen Redefinitionen der Sprache durch

eine Standardisierungskommission. Dagegen gibt es für Anwendungsprogrammierer keine Möglichkeit, die Sprache oder Funktionalität zu erweitern.

- Über die Sprachdefinition hinaus werden keine Aussagen über das Datenbanksystem und seine Anbindung getroffen. Somit gibt es keine einheitlichen Programmierschnittstellen für SQL-Datenbanksysteme außerhalb der Sprache, was die Verwendungsmöglichkeiten von SQL-Systemen verringert [Brodie, Schmidt 82].

### 1.1.2 DBPL — eine Datenbankprogrammiersprache

DBPL [Schmidt, Matthes 92] ist eine Erweiterung von Modula-2 [ModISO 91], wobei die Erweiterungen fest in die Sprache und das Laufzeitsystem eingebaut sind (*built-in*). Die Erweiterungen umfassen zwei Bereiche:

**Laufzeitsystem:** Das Laufzeitsystem stellt in einer mehrschichtigen Architektur ein erweitertes relationales Datenmodell mit entsprechenden Operationen zur Verfügung. Die Schnittstelle zum Laufzeitsystem ist typunsicher. Um eine fehlerhafte Benutzung durch Anwendungsprogrammierer zu vermeiden, werden Aufrufe des Laufzeitsystems vom Compiler erzeugt.

**Compiler:** Der Sprachumfang von Modula-2 wird um einen Typkonstruktor für Relationen, Operationen auf Relationen, Transaktionen und ein Persistenzkonzept für Programmvariablen orthogonal erweitert. Diese neuen Konstrukte werden vom Compiler auf Typkonformität geprüft und (transparent für den Programmierer) in Aufrufe des Laufzeitsystems umgesetzt. Die Konzepte zur Benennung, Bindung und Typisierung werden von Modula-2 übernommen.

Die Lösung der oben angeführten Aufgabe in DBPL wird in Beispiel 1.1.2 gezeigt.

**Beispiel 1.1.2:** Lösung der Aufgabe in DBPL

```
PROCEDURE printNames(age: INTEGER);
VAR p: Person;
BEGIN
  FOR EACH p IN persons : p.age > age DO
    InOut.WriteString(p.name); InOut.WriteLn;
  END;
END printNames;
```

Es handelt sich um eine Prozedur, die das Alter *age* als Parameter erhält. Eine lokale Variable *p* vom Typ *Person* dient zur Iteration über die Menge von Personen *persons*. Im Gegensatz zum Beispiel in C mit eingebetteten SQL-Anweisungen muß der Anwendungsprogrammierer die Iteration nicht elementweise ausprogrammieren, sondern es steht ihm eine höhersprachliche Abstraktion in Form der **FOR EACH**-Schleife zur Verfügung, die

sich als Erweiterung der Syntax und Semantik der Basissprache präsentiert. Es besteht keine Trennung verschiedener Namensräume und somit kein konzeptueller Bruch.

Die bei der Erweiterung eines Modula-2–Compilers zu einem DBPL-Compiler und der weiteren Arbeit mit dem DBPL-System erworbenen Erfahrungen lassen sich folgendermaßen zusammenfassen [Matthes 88; Schröder 91; Matthes, Schmidt 91]:

- Das Laufzeitsystem ist ein komplexes Softwaresystem, das nur mit Systemkenntnissen gewartet und verändert werden kann. Sich ständig ändernde Anforderungen führen zu einem kontinuierlichen Wachstum des Systems. Das Laufzeitsystem oder auch Teile davon lassen sich nicht in eine gängige Bibliothek von Modulen, die auch Anwendungsprogrammierern zur Verfügung stehen, integrieren, da sie typunsicher sind.
- Neue Anforderungen bedingen Erweiterungen der Sprache. Jede Änderung des Sprachumfangs führt zu Änderungen am Compiler und evtl. am Laufzeitsystem, die nur Systemprogrammierer vornehmen können. Die Änderungen am Compiler lassen sich nicht in wenigen Modulen lokalisieren, sondern wirken sich an vielen Stellen aus, was die Fehlerträchtigkeit erhöht und die Übertragbarkeit verringert.
- Die datenbankspezifischen Sprachkonstrukte erleichtern die Programmierung von Datenbankanwendungen, da sie problemangemessen sind und sich durch die vollständige Integration in die Sprache nicht von den Konstrukten der Anwendungsprogrammiersprache abheben. Die Fehlermöglichkeiten werden durch Typprüfungen und automatisch generierte Aufrufe des Laufzeitsystems vermindert.

### 1.1.3 Tycoon — ein Bibliotheksansatz

Im Tycoon-Projekt [Matthes, Schmidt 91; Matthes 93; Niederée 92; Schmidt, Matthes 93] werden die Minimalanforderungen an eine Datenbankprogrammiersprache untersucht. Dabei haben sich folgende Konzepte als zentrale Anforderungen erwiesen (vgl. auch [Atkinson, Bunemann 87]):

**Persistenz:** Alle Zugriffe auf den Speicher erfolgen über eine schmale Schnittstelle (TSP, *Tycoon Store Protocol*). Die über diese Schnittstelle abgelegten Daten werden persistent gespeichert.

**Typisierung:** Ein polymorphes Typsystem gestattet die typsichere Beschreibung und Manipulation von Massendaten über generische Bibliotheken. Die Bibliotheken stellen z. B. Transaktionsverwaltungen und Operationen auf Massendaten zur Verfügung.

**Benennung und Bindung:** Über ein einheitliches Benennungs- und Bindungskonzept können externe Server, z. B. Datenbankserver, eingebunden und angesprochen werden (*add-on*).

Als prototypische Programmiersprache des Tycoon-Systems wird TL [Matthes 93] verwendet. Ein Beispiel für ein generisches Bibliotheksmodul ist *iter*, das Operationen auf Massendaten zur Verfügung stellt. Es enthält u. a. eine generische Funktion mit folgender Signatur:

*forEach(:E <*:**Ok** *bulk :T(:E) pred :***Fun***(? :E) :Bool act :***Fun***(? :E) :***Ok***) :***Ok**

Diese Funktion iteriert über eine Kollektion *bulk*, die Elemente des Typs *E* enthält, wobei dieser Typ erst beim Aufruf der Funktion festgelegt wird. Für jedes Element der Kollektion wird eine als Parameter zu übergebende Funktion *pred* aufgerufen. Wenn diese Funktion für ein Element den Wahrheitswert *true* liefert, wird die ebenfalls als Parameter zu übergebende Funktion *act* für dieses Element aufgerufen. Der benutzerdefinierte Kollektionstyp *T(:E)* abstrahiert von der Realisierung des Massendatentyps. Über den Typparameter *E* wird zugesichert, daß nur solche aktuellen Parameter für *bulk*, *pred* und *act* akzeptiert werden, die nicht zu einem Typfehler führen.

In Beispiel 1.1.3 wird die Lösung der gestellten Aufgabe unter Verwendung dieser Funktion gezeigt. Die Funktion *printNames* erhält einen Parameter *age*. Sie besteht aus einem einzigen Aufruf der generischen Bibliotheksfunktion *iter.forEach*, wobei als Typparameter *T* der Typ *Person* übergeben wird. Der Parameter *bulk* erhält als Wert eine Kollektion von Personen *persons*. Das Prädikat *pred* wird mit einer Funktion belegt, die für eine Person *p* prüft, ob das Alter der Person größer als der Parameter *age* der Funktion *printNames* ist. Die Funktion *act* gibt für eine Person *p* deren Namen aus.

**Beispiel 1.1.3:** Lösung der Aufgabe in TL

**let** *printNames* = **fun***(age :Int)*
  *iter.forEach(***Let** *E = Person*
    **let** *bulk = persons*
    **let** *pred* = **fun***(p :Person) p.age > age*
    **let** *act* = **fun***(p :Person)* **begin** *print.string(p.name) print.ln()* **end***)*

Die im Tycoon-Projekt gesammelten Erfahrungen lassen sich folgendermaßen zusammenfassen:

- Über die Speicherschnittstelle können mit geringem Aufwand verschiedene Speichersysteme angebunden werden. Die Bibliotheken können von Anwendungsprogrammierern erstellt, geändert und erweitert werden. Die Anbindung vorhandener Server kann bei entsprechend gut dokumentierten bzw. standardisierten Programmierschnittstellen ein Anwendungsprogrammierer ohne Eingriff in das System vornehmen.

- Die Formulierung generischer Bibliotheken und die Anbindung externer Server erfolgt in einer minimalen Sprache, die einen *einheitlichen* Rahmen für Benennung, Bindung und Typisierung zur Verfügung stellt.

- Dem Anwendungsprogrammierer präsentieren sich die Schnittstellen der Bibliotheken als Sammlungen von Typ- und Wertsignaturen, beispielsweise die oben angeführte

Signatur der Funktion *forEach*. Die Benutzung dieser Bibliotheken führt zu geschachtelten oder sequentiellen Funktionsaufrufen. Diese Aufrufe geben die Semantik nur unzureichend wieder und führen zu schwer wartbaren Programmen. Wünschenswert ist deshalb die Unterstützung anderer Programmierstile.

- Die Signaturen der Bibliotheksmodule sind nicht ausdrucksfähig genug. Es lassen sich keine Funktionen mit beliebig vielen Parametern unterschiedlichen Typs beschreiben, wie es z. B. bei einer generischen Iterationsfunktion über $n$ Kollektionen nötig wäre, die ungefähr so formuliert werden müßte:

  *forEachN(:E1, E2, ... <:***Ok** *bulk1 :T(:E1) bulk2 :T(:E2) ...*
  *pred :***Fun***(? :E1 ? :E2 ...) :Bool*
  *act :***Fun***(? :E1 ? :E2 ...) :***Ok***) :***Ok**

  Im Gegensatz zur Funktion *forEach* erhält die Funktion *forEachN* eine Menge von Kollektionen $bulk_i$ als Parameter, deren Elemente die Typen $E_i$ besitzen. Die beiden Funktionen *pred* und *act* erhalten dementsprechend eine Menge von Elementen unterschiedlichen Typs $E_i$ als Parameter. Dies läßt sich mit den Typsystemen polymorpher Programmiersprachen wie TL nicht ausdrücken. Stattdessen können nur Iterationsfunktionen mit einer *festen* Anzahl von Argumenten definiert werden und es liegt in der Verantwortung des Programmierers, eine Iteration über $n$ Kollektionen in Aufrufe dieser Funktionen umzusetzen.

### 1.1.4 Zusammenfassung und Schlußfolgerungen

Die mit den drei Ansätzen — Datenbanksprachen, Datenbankprogrammiersprachen und Bibliotheken — gesammelten Erfahrungen lassen sich folgendermaßen zusammenfassen:

- Datenbanksprachen wie SQL und Datenbankprogrammiersprachen wie DBPL verfügen über problemangemessene Sprachkonstrukte, die die Semantik eines Programms leicht verständlich beschreiben.
- Die Umsetzung von (eingebetteten) Datenbankanweisungen in Aufrufe eines Datenbanklaufzeitsystems ist relativ einfach.
- Die Verknüpfung einer Datenbanksprache mit einer Programmiersprache über eine Einbettung (eingebettetes SQL) führt zu konzeptuellen Problemen und beschränkt die Erweiterbarkeit.
- Die Erweiterung einer Programmiersprache zu einer Datenbankprogrammiersprache ist komplex und führt zu Problemen bei Änderungen der Ausgangssprache oder der Funktionalität der Datenbankerweiterungen.

- Der Bibliotheksansatz erlaubt skalierbare Funktionalität und zeigt sich flexibel gegenüber Änderungen. Ein Anwendungsprogrammierer kann ohne weitreichende Systemkenntnisse das System nach seinen Bedürfnissen gestalten.
- Der Bibliotheksansatz verfügt wie die Datenbankprogrammiersprachen über einen einheitlichen sprachlichen Rahmen mit gut verstandenen Konzepten zur Benennung, Bindung und Typisierung.
- Die in einer minimalen Sprache unter der massiven Benutzung von Bibliotheken formulierten Programme geben die Semantik nicht ausreichend wieder und sind daher schwer wartbar.

Wünschenswert ist eine Verknüpfung der Vorteile der drei Ansätze, die zu folgenden Forderungen an das in dieser Arbeit zu entwickelnde System führt:

- Skalierbare und erweiterbare Funktionalität (Bibliotheksansatz).
- Einheitlicher sprachlicher Rahmen bzgl. Benennung, Bindung und Typisierung (Bibliotheksansatz und Datenbankprogrammiersprachen).
- Semantisch ausdrucksfähige Sprachkonstrukte (Datenbanksprachen und Datenbankprogrammiersprachen).

Die semantisch aussagekräftigen, datenbankspezifischen Sprachkonstrukte werden letztendlich in Funktionsaufrufe von Bibliotheken abgebildet, die sich in der Minimalsprache formulieren lassen. Diese Abbildung muß geeignet beschrieben werden. Die Abbildung kann sich ändern, wenn die Funktionalität des Systems sich ändert. Zum Beispiel könnte ein Iterationskonstrukt wie **for each** in einer einfachen Version eine Schleife über ein Feld (*array*) der Minimalsprache beschreiben, in einem erweiterten System aber an einen Datenbankserver abgegeben werden, der über eine ggf. sehr große Tabelle iteriert. Außerdem sollen auch neue Sprachkonstrukte eingeführt werden können, die eine erweiterte Funktionalität, eine höhere Abstraktion oder einen anderen Programmierstil unterstützen. Um dies zu erreichen, muß die Syntax der Sprache analog zu der Funktionalität skalierbar bzw. erweiterbar sein.

Beispiel 1.1.4 zeigt die Lösung der oben angeführten Aufgabe in einer syntaktisch um ein Iterationskonstrukt im Stile von DBPL (siehe Abschnitt 1.1.2) erweiterten Version der in Abschnitt 1.1.3 vorgestellten minimalen Sprache TL.

**Beispiel 1.1.4:** Lösung der Aufgabe in syntaktisch erweitertem TL

```
let printNames = fun(age :Int)
  for each p in persons : p.age > age do
    print.string(p.name) print.ln()
  end
```

Die neue Syntax wird folgendermaßen definiert (im Vorgriff auf Kapitel 3 wird die in dieser Arbeit entwickelte Form der Syntaxerweiterung gewählt):

"for" "each" *id* = *ideG* "in" *bulk* = *valG* ":" *pred* = *valG* "do" *act* = *bndsG* "end"
=> **VAL**
  *iter.forEach(***Let** *E = ?*
    **let** *bulk* = **VAL** *bulk*
    **let** *pred* = **fun***(***IDE** *id :?)* **VAL** *pred*
    **let** *act* = **fun***(***IDE** *id :?)* **begin BNDS** *act* **end***)*
**END**

Die Syntaxerweiterung definiert eine Umsetzung des Sprachkonstrukts **for each** in einen Aufruf der in Abschnitt 1.1.3 vorgestellten Bibliotheksfunktion *forEach*. Das oben angeführte Beispiel führt zu folgender Expansion der Syntaxerweiterung:[1]

**let** *printNames* = **fun***(age :Int)*
  *iter.forEach(***Let** *E = ?*
    **let** *bulk* = *persons*
    **let** *pred* = **fun***(p :?) p.age > age*
    **let** *act* = **fun***(p :?)* **begin** *print.string(p.name) print.ln()* **end***)*

## 1.2 Historie erweiterbarer Programmiersprachen

Einen Einblick in die vielen Arbeiten über erweiterbare Programmiersprachen, die vor allem in den 60er und Anfang der 70er Jahre stattfanden, geben z. B. [Wegbreit 70; Solntseff, Yezerski 71; Layzell 85; Kohlbecker 86] sowie die beiden Konferenzen zu diesem Thema [Christensen, Shaw 69; Schuman 71].

„Erweiterbarkeit“ wird nach [Rowe 81] in folgenden Teilgebieten untersucht:

**Kontrollstrukturen:** Deklaration und Aufruf von Funktionen und Prozeduren sowie Ausführungskonzepte und Nebenläufigkeit.

**Datenstrukturen:** Typkonzepte und benutzerdefinierte Datentypen.

**Operationen:** Abstrakte Datentypen und Operationen sowie überladene und benutzerdefinierte Infix-Operatoren.

**Syntax:** Veränderung der Syntax einer Sprache.

Die Forderung nach „Erweiterbarkeit“ wird damit begründet, daß keine Sprache für jede Anforderung bzw. jeden Anwendungsbereich gleich gut geeignet ist und somit Anpassungen

[1] Für den TL-kundigen Leser sind die Fragezeichen "?" unbekannt. Sie stellen Unifikationsvariablen [Cardelli 91] dar, die es momentan in TL nicht gibt. Nähere Erläuterungen finden sich in Kapitel 4.

wünschenswert sind, um eine problemadäquate Formulierung der (Programmier-)Lösung zu ermöglichen [Layzell 85; Ipser 92].

Die beiden folgenden Zitate zeigen, daß sich die Intentionen zur Entwicklung erweiterbarer Programmiersprachen seit zwei Jahrzehnten nicht gewandelt haben:

> Language extensibility refers to the ability of the programmer to modify the language being used, with the intent of extending the power of the language. A relatively small 'base' language is defined, along with capabilities to add features such as new data types, new operators, new syntax, and new control structures in order to enable the program to more closely correspond to the problem domain. The derived language or 'task language' which thereby results can allow programs to be written in such a way that they are comprehensible to almost anyone familiar with the application area of the program. Persons working on the development of extensible language features foresee the establishment of a higher level language which could evolve gracefully via packages of definitions. The availability of such packages for particular task areas could then greatly increase program productivity.
> aus: [Wasserman 75]

> It could even be possible that the concept of a programming 'language' would gradually vanish in such development, in favour of a programming 'system' consisting of of relatively loosely coupled replacable parts. In the author's view, this kind of development would be profitable to software engineering in general: various kinds of differently oriented programming systems could be rapidly developed, and the programmers would be easily adapted to using these systems because they consist of familiar, generally known components.
> aus: [Koskimies 88]

Erweiterbare Konzepte aus den Bereichen Kontrollstrukturen (z. B. Module, generische Funktionen, parallele Anweisungen), Datenstrukturen (z. B. Polymorphismus, Vererbung) und Operationen (z. B. abstrakte Datentypen, benutzerdefinierte Infixoperatoren) besitzen inzwischen einen festen Platz in der Informatik, wie jedes Lehrbuch über Programmiersprachen zeigt [Horowitz 84; Sebesta 89].

Dagegen wird die *syntaktische* Erweiterbarkeit nicht mehr als lohnendes Forschungsthema gesehen:

> Though many attempts at designing extensible languages have been made, none have proven to be entirely successful.
> [Horowitz 84, S. 40]

[Gries 76] kritisiert, daß syntaktisch erweiterbare Programmiersprachen zu nicht mehr wartbaren Programmen führen, weil jeder Programmierer die Syntax mit einer neuen Semantik ausstatten kann, die nicht mehr offensichtlich ist, während bei einer Sprache mit festem Sprachumfang die Bedeutung eines Programms mit Hilfe der Sprachdefinition hergeleitet werden kann.

Dagegen lassen sich zwei Einwände formulieren:

1. Dieses Argument trifft auf einer anderen Ebene auch auf Programmiersprachen zu, die nicht syntaktisch erweiterbar sind, da alle Sprachen dem Programmierer die Freiheit zu semantikferner Programmierung erlauben. Beispielsweise läßt sich durch die Wahl sinnloser Namen für Funktionen, Variablen usw., die exzessive Benutzung globaler Namen, Sprunganweisungen, eingebettete oder externe Assembler-Routinen, *aliasing* (zwei Zugriffspfade für ein Objekt), Zeigermanipulationen usw. jedes Programm unwartbar gestalten. Beweis dafür sind viele unwartbare bzw. fehlerhafte Programme, die in der (relativ simplen) Programmiersprache C geschrieben wurden. Da syntaktische Erweiterungen einen zusätzlichen Freiheitsgrad für die Programmierer darstellen, ist auch hier eine Disziplin gefragt, die außerhalb des Verantwortungsbereichs der Sprache liegt.

2. Alle Syntaxerweiterungen sind auch dokumentiert und können somit neben der Sprachdefinition der Basissprache bei der Herleitung der Semantik eines Programms herangezogen werden. Im Extremfall kann auch das Programm in der reinen Basissprache *nach* der Expansion aller Syntaxerweiterungen betrachtet werden. Da die Basissprache per Definition nicht sehr umfangreich ist und somit die Semantik der wenigen Konstrukte der Basissprache noch einfacher als bei den meisten vergleichbaren Programmiersprachen ohne syntaktische Erweiterung definiert werden kann, steht dem Verstehen eines Programms auf dieser Ebene nichts im Wege.

[Gries 76] führt weiterhin aus, daß eine Spezialisierung von Sprachen für bestimmte Anwendungen durchaus wünschenswert ist, um die Semantik des Anwendungsbereichs besser zu erfassen, solange wohlverstandene Basissprachen den Ausgangspunkt bilden. Dabei setzt er neben Makros, deren Probleme Thema des folgenden Kapitels sind, auf eine Anpassung der Compiler wie im Falle der in Abschnitt 1.1.2 angeführten Datenbankprogrammiersprachen, was zu den geschilderten Problemen führt. Damit liefert er indirekt doch wieder ein Argument *für* syntaktische Erweiterbarkeit, eventuell angereichert um eine Programmierdisziplin ähnlich der strukturierten Programmierung, den Richtlinien für Modulentwurf und anderen Konzepten (vgl. [Balzert 82]). Die Frage, wann und in welchem Umfang syntaktische Erweiterungen sinnvoll sind und von wer sie vornimmt sind, ist nicht Thema dieser Arbeit. Es bleibt nur festzuhalten, daß im Kontext von Datenbanken die Erweiterung von Programmiersprachen um datenbankspezifische Anweisungen wünschenswert ist, um die Semantik dieser Anweisungen besser darzustellen und Fehler von Programmierern zu verringern, und daß diese Erweiterungen auch von Anwendungsprogrammierern und nicht nur von (hochspezialisierten) Systemprogrammierern vorgenommen werden sollen.

[Layzell 85] führt den Mißerfolg (syntaktisch) erweiterbarer Programmiersprachen u. a. auf folgende Ursachen zurück:

1. Die Basissprachen sind *zu* einfach und damit schwierig zu erweitern.
2. Die Sprachen zur Beschreibung der syntaktischen Erweiterungen sind zu komplex. Einfache Erweiterungsmöglichkeiten wie textuelle Ersetzungen (Makros) sind einfacher zu erlernen.
3. Die sprachlichen Erweiterungen sind nicht effizient.
4. Erweiterbare Programmiersprachen bieten nicht genügend Anreize, um von etablierten auf erweiterbare Sprachen umzusteigen.

Daraus lassen sich folgende Forderungen an erweiterbare Programmiersprachen ableiten:

- Erweiterungen müssen einfach beschreibbar sein und effizient umgesetzt werden.
- Basissprache und Erweiterungssprache müssen zusammen eine Mächtigkeit ergeben, die über existierende Programmiersprachen hinausgeht. Als Pflichtübung für jede erweiterbare Programmiersprache muß es gelten, die Syntax zumindest einer anerkannten Programmiersprache nachzubilden, so daß sie sofort, auch ohne weitere Erweiterungen, eingesetzt werden kann.

Als Negativbeispiel eines nicht gelungenen Mechanismus zur syntaktischen Erweiterbarkeit wird meist der Makro-Mechanismus angeführt, wie er z. B. in den Programmiersprachen C bzw. C++ und LISP zu finden ist. Die Makromechanismen beruhen auf Textersetzung, sind einfach zu verstehen, anzuwenden und zu implementieren. Makros erlauben es jedoch nicht, neue syntaktische Konstrukte zu definieren, und ihre Anwendung führt oft zu unerwarteten Resultaten. So warnt [Stroustrup 92] ausdrücklich vor der Verwendung von Makros in C++ und [Kohlbecker 86] ersetzt den Makro-Mechanismus von LISP komplett durch einen Mechanismus zur syntaktischen Erweiterbarkeit.

Dagegen bieten Compiler-Compiler die volle Mächtigkeit bei der Definition von Programmiersprachen. Das Ziel von Compiler-Compilern ist die vollständige Beschreibung eines Compilers in verschiedenen problemspezifischen Sprachen [Rechenberg, Mössenböck 88; Grosch, Emmelmann 90]. Allerdings bieten die wenigsten Compiler-Compiler eine Unterstützung von *inkrementellen* Änderungen [Heering et al. 89]. Die Fülle verschiedener Sprachen und Mechanismen und die Komplexität der resultierenden Beschreibungen widerspricht der Forderung nach einfacher Erlern- und Handhabbarkeit eines Mechanismus zur syntaktischen Erweiterung. Ein realistischer Ansatz zur syntaktischen Erweiterung ist also auf einer Ebene zwischen den Makros und den Compiler-Compilern anzusiedeln, was in Kapitel 2 weiter ausgeführt wird.

Aus den vorstehenden Ausführungen ergibt sich, daß — historisch gesehen — Ansätze zur syntaktischen Erweiterbarkeit verfolgt wurden, die nicht den Anforderungen aus der Praxis der Anwendungsprogrammierung gerecht werden. Damit wurde gleichzeitig das Forschungsfeld der syntaktisch erweiterbaren Programmiersprachen diskreditiert. Andere Ansätze wie [Leavenworth 66] sind dagegen nicht oder erst spät [Kohlbecker 86] aufgegriffen worden, obwohl sie die Erfüllung dieser Anforderungen versprechen.

## 1.3 Ziel dieser Arbeit

Ziel dieser Arbeit ist der Nachweis, daß es möglich ist, einen Mechanismus zur syntaktischen Erweiterung von Programmiersprachen zu entwickeln, der

- einfach aufgebaut ist,
- keine unerwarteten Resultate produziert,
- die Effizienz des Übersetzungsvorgangs nicht einschränkt und
- zusammen mit einer adäquaten Basissprache die Definition einer mächtigen Programmiersprache erlaubt.

Besondere Beachtung wird der Wahrung von Benennungs-, Bindungs- und Typisierungsinvarianzen der Basissprache gewidmet, indem Bindungsprobleme in syntaktischen Erweiterungen intensiv diskutiert werden. Der Mechanismus zur syntaktischen Erweiterung soll sich auf vorhandene Technologien aus dem Bereich Compilerbau stützen. Die Integration dieses Mechanismus in vorhandene Systeme wird am Beispiel des Tycoon-Systems untersucht.

Es wird davon ausgegangen, daß Basissprachen zum Aufbau von Bibliotheken existieren oder entwickelt werden. Dafür geeignete Sprachen sollten z. B. ein Modulkonzept und ein polymorphes Typsystem für generische Schnittstellen enthalten [Atkinson, Bunemann 87; Matthes 93; Schmidt, Matthes 93]. Am Beispiel der Programmiersprache TL wird gezeigt, daß der Sprachumfang vorhandener Programmiersprachen durch Einsatz eines Mechanismus' zur syntaktischen Erweiterung reduziert werden kann, ohne ihre Ausdrucksfähigkeit einzuschränken.

## 1.4 Aufbau dieser Arbeit

In diesem Kapitel wurde gezeigt, welche besonderen Vorteile eine syntaktisch erweiterbare Programmiersprache als *Datenbank*programmiersprache verspricht und warum das Gebiet

der syntaktisch erweiterbaren Programmiersprachen auch nach 30 Jahren Forschung immer noch nicht voll erschlossen ist.

Die Einordnung der syntaktischen Erweiterbarkeit in ein klassisches Compilermodell leistet Kapitel 2. Dabei wird auch die Verwendung vorhandener Basistechnologien aus dem Compilerbau, insbesondere Parser-Generatoren, untersucht.

Die im Rahmen dieser Arbeit entwickelte Sprache zur syntaktischen Erweiterung TLExt und die verwendete Basissprache TLMin werden in Kapitel 3 vorgestellt.

In Kapitel 4 wird die Anwendung von Syntaxerweiterungen zur Definition verschiedener Sprachkonstrukte gezeigt, wobei ein Schwerpunkt auf Anwendungen aus dem Datenbankbereich liegt.

Ein besonderes Problem von Syntaxerweiterungen stellen Bindungen dar, wie schon [Leavenworth 66] bemerkt. Deshalb ist Kapitel 5 der Lösung dieses Problems gewidmet.

In den Anhängen A und B werden die abstrakte Syntax und die statische Semantik der Erweiterungssprache und (teilweise) der Basissprache formalisiert, so daß eine Übertragung auf andere Basissprachen erleichtert wird.

# Kapitel 2

# Syntaxerweiterungen im Compilermodell

Ein Ziel dieser Arbeit ist die weitgehende Nutzung vorhandener Basistechnologien. Im Falle von Syntaxerweiterungen stammen die zugrundeliegenden Technologien aus dem Compilerbau. Deshalb werden in diesem Kapitel ein Compilermodell umrissen, Syntaxerweiterungen in dieses Modell eingeordnet und die zur Implementierung nutzbaren Compilerbau-Technologien vorgestellt.

## 2.1 Compilermodell

Dieser Abschnitt erläutert ein Compilermodell, mit dessen Hilfe verschiedene Ansätze zur Syntaxerweiterung klassifiziert werden können. Es handelt sich um ein klassisches Phasenmodell nach [Aho et al. 88].

### 2.1.1 Grobstruktur eines Compilers

Ein Compiler hat die Aufgabe, einen Quelltext (*source text*) in einen maschinenabhängigen Zielkode (*target code*) zu übersetzen. Diese Aufgabe läßt sich in verschiedene Teilaufgaben zerlegen, die hintereinander ausgeführt werden können. Jede Teilaufgabe transformiert eine Repräsentation des Quelltextes in eine andere, wobei die letzte Repräsentation der Zielkode ist. Die Abarbeitung einer Teilaufgabe wird als Phase bezeichnet.

Dieses Modell beschreibt den *logischen* Aufbau eines Compilers. In einer Implementation können aus Effizienzgründen durchaus mehrere Phasen in einem Lauf (*pass*) zusammengefaßt und verschiedene Repräsentationen in einer Datenstruktur dargestellt werden.

Abbildung 2.1 zeigt die aus zwei Phasen und drei Repräsentationen bestehende Grobstruktur eines Compilers.

Abbildung 2.1: Grobstruktur eines Compilers

1. Das Frontend analysiert den Quelltext und transformiert ihn in einen maschinenunabhängigen Zwischenkode.
2. Das Backend synthetisiert aus dem Zwischenkode einen maschinenabhängigen Zielkode.

Syntaxerweiterungen beziehen sich nur auf das Frontend: Nach einer Syntaxerweiterung bearbeitet das Frontend Quelltexte mit geänderten oder neuen syntaktischen Konstrukten. Da die Form des Zwischenkodes und das Backend durch syntaktische Erweiterungen nicht betroffen sind, wird im folgenden nur noch das Frontend betrachtet.

### 2.1.2 Feinstruktur des Frontends eines Compilers

Das Frontend teilt sich in verschiedene Phasen auf, die im folgenden als Funktionen definiert werden [Alber 71; Stemple et al. 91]. Abbildung 2.2 a) zeigt die Phasen und Repräsentationen des Frontends. Abbildung 2.2 b) enthält ein Beispiel für die Transformation der Datenstrukturen.

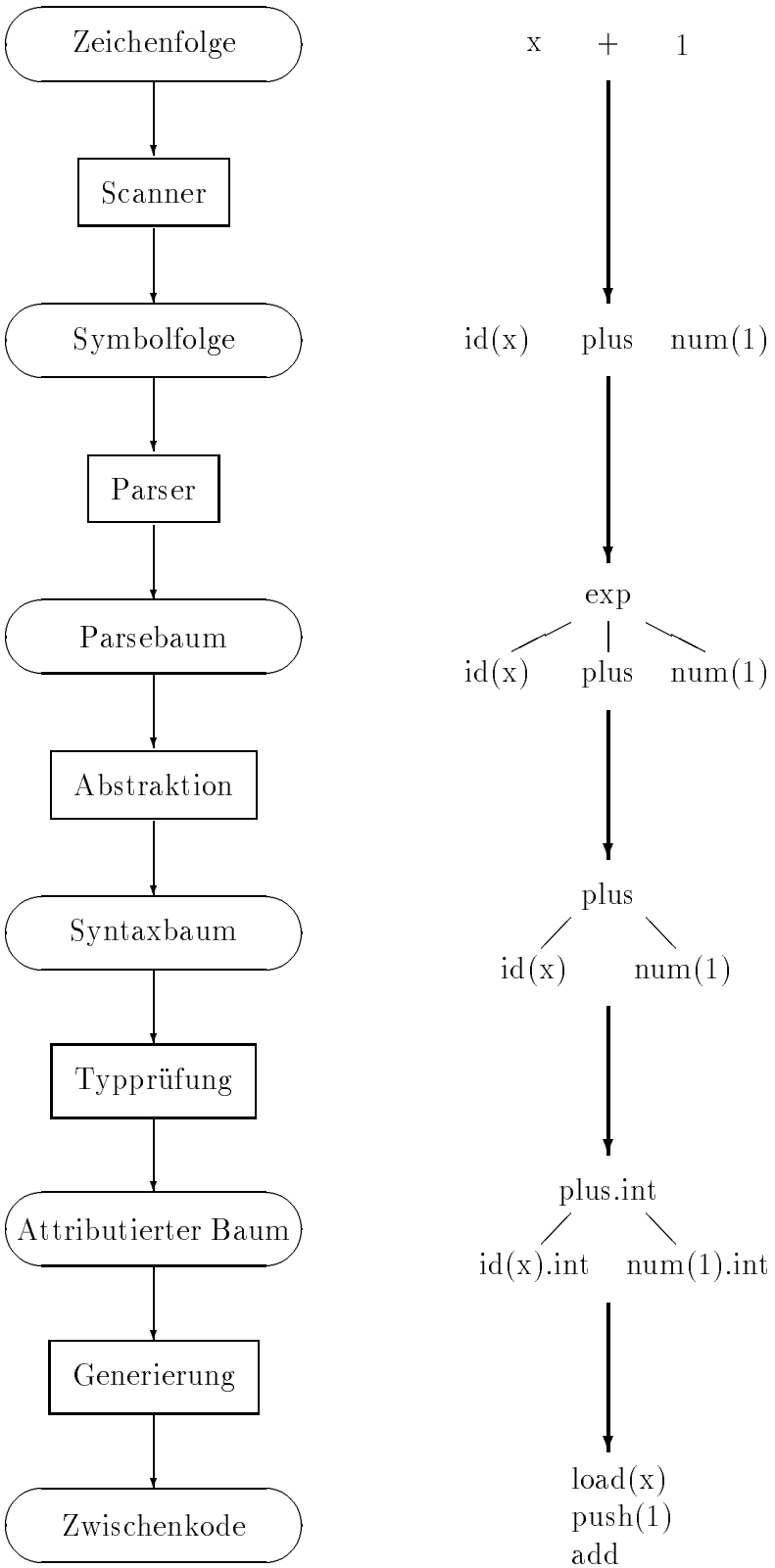
$scan : Characters \rightarrow Symbols$
: Die lexikalische Analyse (Scanner) zerlegt den Quelltext, der als Zeichenfolge betrachtet wird, in eine Symbolfolge.

$parse : Symbols \rightarrow ParseTree$
: Die syntaktische Analyse (Parser) analysiert die Symbolfolge auf syntaktische Konstrukte, die als Parsebaum repräsentiert werden.

$abstract : ParseTree \rightarrow AST, \quad AST = AbstractSyntaxTree$
: Durch die Abstraktion wird der Parsebaum in einen abstrakten Syntaxbaum transformiert.

$check : AST \rightarrow AST'$
: Die statische semantische Analyse (Typprüfung) überprüft, ob die Konstrukte semantisch korrekt verwendet werden, und fügt semantische Informationen als Attribute in den Syntaxbaum ein.

$generate : AST' \rightarrow IntermediateCode$
: Die Zwischenkode-Erzeugung (Generierung) erzeugt aus dem attributierten Syntaxbaum einen Zwischenkode, der vom Backend weiterverarbeitet wird.




a) Phasen und Repräsentationen b) Beispiel

Abbildung 2.2: Feinstruktur eines Compiler-Frontends

Mit den beschriebenen Funktionen und einem Quelltext (*source*) als Eingabe läßt sich der (logische) Aufbau des Compilers funktional folgendermaßen beschreiben:

$$backend(frontend(source)) = backend(generate(check(abstract(parse(scan(source))))))$$

Diese Beschreibungsform betont, daß die einzelnen Phasen (Funktionen) nur vom Ergebnis der vorherigen Phase abhängen.

## 2.2 Beschreibungstechniken und Generatoren

Eine Programmiersprache kann durch passende Beschreibungstechniken formal spezifiziert werden, was sich als vorteilhaft für die Kommunikation zwischen Sprachdesignern und Compilerbauern sowie Anwendungsprogrammierern erweist. Generatoren können aus bestimmten Beschreibungen automatisch entsprechende Teile eines Compilers erzeugen. Wenn die Beschreibung eines Frontends vorliegt und entsprechende Generatoren zur Verfügung stehen, kann die Syntax einer Sprache erweitert werden, indem die Beschreibung geändert und ein neues Frontend generiert wird.

In diesem Abschnitt wird eine Zuordnung der wichtigsten Beschreibungstechniken und Generatoren zu den einzelnen Phasen des Frontends vorgenommen, um

- diese Möglichkeit zur syntaktischen Erweiterung bewerten zu können und
- die in dieser Arbeit verwendeteten Beschreibungstechniken und Generatoren vorzustellen.

### 2.2.1 Lexikalische Analyse

Die Abbildung von Zeichenfolgen auf Symbolfolgen wird durch reguläre Ausdrücke spezifiziert. Jedem Symbol der Sprache wird ein regulärer Ausdruck zugeordnet, der die Repräsentation des Symbols als Zeichenkette beschreibt. Umgekehrt muß jeder Zeichenkette, die in einem Quelltext auftauchen kann, ein Symbol der Sprache zugeordnet werden. Aus diesen Paaren (regulärer Ausdruck, Symbol) erzeugt ein Scanner-Generator einen endlichen Automaten, der die Abbildung realisiert.

Die folgende Tabelle zeigt einige Beispiele für reguläre Ausdrücke:

| | |
|---|---|
| "begin" | Schlüsselwort: **begin** |
| ":" | Schlüsselwort: Doppelpunkt |
| [0–9]+ | ganze Zahl: mindestens eine Ziffer |
| [a–zA–Z][a–zA–Z0–9]* | Bezeichner: Buchstabe gefolgt von Buchstaben und Ziffern |

Ein Scanner übernimmt weitere Aufgaben, die sich teilweise nicht durch reguläre Ausdrücke beschreiben lassen, beispielsweise das Entfernen von Leerzeichen, Zeilenumbrüchen und

Zeilenvorschüben, Tabulatoren und (evtl. geschachtelten) Kommentaren, oder das Erkennen von speziellen Kommandos (*pragmas*), die den Compilerlauf beeinflußen, aber nicht zur Sprache gehören. Für diese Aufgaben müssen spezielle Beschreibungsformen eingeführt werden, worauf an dieser Stelle nicht näher eingegangen wird, da keine allgemein akzeptierte Beschreibungsform existiert.

### 2.2.2 Syntaktische Analyse

Die in einer Sprache erlaubten Symbolfolgen (die Sprachsyntax) werden durch kontextfreie Grammatiken beschrieben. Aus einer kontextfreien Grammatik erzeugt ein Parser-Generator einen Parser, der die eine Symbolfolge akzeptierenden Produktionen als Parsebaum darstellt.

Die grundlegende Form der kontextfreien Grammatiken beinhaltet folgende Elemente:

**Terminal:** Ein Symbol aus der Sprache, das vom Scanner erkannt wird, notiert als Zeichenkette oder Name eines im Scanner definierten regulären Ausdrucks.

Die folgende Tabelle zeigt einige Beispiele für Terminale:

| | |
|---|---|
| "for" | das Schlüsselwort **for** |
| ":" | ein Doppelpunkt, der auch als Schlüsselwort betrachtet wird |
| *int* | eine Zahl gemäß der Definition des Scanners |
| *id* | ein Bezeichner gemäß der Definition des Scanners |

**Nonterminal:** Name einer Produktion. Beispiele: *ideG, valG.*

**Produktion:** Benannte Folgen von Terminalen und Nonterminalen, wobei auch leere Folgen möglich sind, die durch das leere Wort $\epsilon$ notiert werden. Jede Folge wird **Alternative** genannt.

Das folgende Beispiel zeigt eine Produktion mit vier Alternativen:

*valG* ::= "nil" | *int* | *id* | *valG* "+" *valG*

Der Name (Nonterminal) der Produktion lautet *valG*. Die akzeptierten Symbolfolgen (Alternativen) sind das Schlüsselwort **nil**, eine Zahl *int* oder ein Bezeichner *id* gemäß der Definition des Scanners sowie die Folge [*valG*, +, *valG*], wobei das Nonterminal *valG* rekursiv eine von der Produktion *valG* akzeptierte Symbolfolge beschreibt.

Kontextfreie Grammatiken können mehrdeutig sein, d. h. zu einer Symbolfolge lassen sich evtl. mehrere Parsebäume konstruieren, die diese Symbolfolge akzeptieren. Mit den verschiedenen Parsebäumen sind normalerweise auch verschiedene Semantiken verbunden,

ein Programm sollte aber *eine* festgelegte Semantik besitzen. Deshalb schränken Parser-Generatoren die Klasse der erlaubten Grammatiken ein [Aho et al. 88, Kap. 4] und/oder benötigen zusätzliche Informationen zur Auflösung von Mehrdeutigkeiten [Aho et al. 88, Kap. 4.8]. Ein Generator kann also eine kontextfreie Grammatik als unzulässig zurückweisen. Verschiedene Generatoren können die gleiche Grammatik als zulässig oder unzulässig betrachten oder verschiedene Zusatzinformationen benötigen.

Zur Konstruktion von Parsebäumen sind zwei Verfahren möglich: Beim Top-Down-Verfahren wird der Parsebaum von der Wurzel (einer bestimmten Startproduktion) zu den Blättern (Symbolen) konstruiert. Bottom-Up-Parser konstruieren den Baum von den Blättern ausgehend zur Wurzel. Zu beiden Verfahren existieren Klassen von Grammatiken, die das Generieren eines deterministischen Parsers erlauben, der nach dem jeweiligen Verfahren den Parsebaum konstruiert. Bei der Entscheidung für eine Klasse von Grammatiken sind zwei sich widersprechende Aspekte zu bedenken: Die Menge der erlaubten Grammatiken ist bei Generatoren, die Bottom-Up-Parser z. B. aus LR-Grammatiken erzeugen, am größten [Knuth 65; Knuth 71]. Nach dem Top-Down-Verfahren arbeitende Parser, die z. B. aus LL-Grammatiken erzeugt werden, sind (meistens) effizienter [Grosch, Emmelmann 90]. Im Sinne der Allgemeinheit ist ein Generator für Bottom-Up-Parser zu empfehlen [Aho et al. 88, S. 234].

### 2.2.3 Abstraktion

Abstrakte Syntaxbäume werden durch Konstruktoren beschrieben, die als Parameter die Werte der Unterbäume bzw. Attribute des Knotens erhalten. Verschiedene Konstruktoren werden zu einer Sorte von Knoten zusammengefaßt.

Das folgende Beispiel zeigt Konstruktoren für abstrakte Syntaxbäume einer Sorte *val*:

```
val   ::=  nilNode()
       |   intNode(int)
       |   idNode(id)
       |   addNode(val val)
```

Ein Syntaxbaum der Sorte *val* ist entweder ein leerer Knoten *nilNode*, ein Knoten *intNode* mit einer ganzen Zahl *int* als Attribut, ein Knoten *idNode* mit einem Bezeichner *id* oder ein Knoten *addNode* mit zwei Unterbäumen der Sorte *val*.

Die Konstruktion eines abstrakten Syntaxbaums aus dem Parsebaum läßt sich durch semantische Aktionen in den Produktionen der kontextfreien Grammatik darstellen, wie das folgende Beispiel zeigt:

```
valG  ::=  "nil"              => nilNode()
       |   int                => intNode(int)
       |   id                 => idNode(id)
       |   valG1 "+" valG2    => addNode(valG1 valG2)
```

Durch das Schlüsselwort **nil** wird ein leerer Knoten *nilNode* erzeugt. Eine Integer-Zahl erzeugt einen Knoten *intNode*, der eine vom Scanner gelieferte Repräsentation *int* des Zahlenwertes als Attribut enthält. Ein Knoten *idNode* wird durch einen Bezeichner *id* erzeugt. Die Addition zweier Werte wird durch den Knoten *addNode* dargestellt, der als Unterbäume die von der Produktion *valG* erzeugten Syntaxbäume enthält.

## 2.2.4 Semantische Analyse

### 2.2.4.1 Attributierte Grammatiken

Die Analyse der statischen Semantik läßt sich mit attributierten Grammatiken beschreiben [Aho et al. 88, Kap. 5]. Jedem Knoten des Syntaxbaums werden weitere Attribute zugeordnet und Vorschriften zur Berechnung dieser Attribute angegeben.

Das folgende Beispiel zeigt eine attributierte Grammatik mit Typinformationen:

| | | | | |
|---|---|---|---|---|
| *val.env* | ::= | *nilNode()* | => | *val.type := undefined* |
| | \| | *intNode(int)* | => | *val.type := integer* |
| | \| | *idNode(id)* | => | *val.type := lookup(val.env id)* |
| | \| | *addNode($val_1$ $val_2$)* | => | *$val_1$.env := val.env;* |
| | | | | *$val_2$.env := val.env;* |
| | | | | **if** *$val_1$.type = integer* |
| | | | | **andif** *$val_2$.type = integer* |
| | | | | **then** *val.type := integer* |
| | | | | **else** *val.type := undefined* |
| | | | | **endif** |

Jeder Knoten des Syntaxbaums der Sorte *val* erhält zwei Attribute: *env* ist eine Umgebung, in der jedem deklarierten Bezeichner ein Typ zugeordnet wird; *type* ist der Typ eines Knotens mit den möglichen Werten *undefined* und *integer*. Ein Knoten *nilNode* hat den Typ *undefined*, ein Knoten *intNode* den Typ *integer*. Der Typ eines Knotens *idNode* wird durch eine Funktion *lookup()* bestimmt, die den einem Bezeichner *id* in der Umgebung zugeordneten Typ zurückliefert. Der Typ des Knotens *addNode* ist *integer*, wenn beide Unterbäume den Typ *integer* haben, ansonsten ist der Typ *undefined*. Außerdem könnten noch Fehlermeldungen erzeugt werden, was hier nicht gezeigt wird.

Die Attribute eines Knotens lassen sich in ererbte (*inherited*) und abgeleitete (*derived*) Attribute aufteilen. Ererbte Attribute eines Knotens erhalten ihren Wert durch einen Vorgängerknoten. Dieser Wert ist entweder der Wert eines ererbten Attributs des Vorgängerknotens oder der Wert eines abgeleiteten Attributs eines Knotens, der im Vorgängerknoten parallel zu diesem Knoten steht. Im Beispiel ist *val.env* ein ererbtes Attribut, dessen Wert im Knoten *addNode* an die Nachfolgerknoten weitergereicht wird. Die Werte der abgeleiteten Attribute werden durch die Nachfolgerknoten bestimmt. Im Beispiel ist *val.type* ein abgeleitetes Attribut.

Durch die Berechnungsvorschriften für die Attributwerte ergibt sich ein Abhängigkeitsgraph, der festlegt, wie oft bzw. in welcher Reihenfolge der Baum durchlaufen werden muß, um alle Attributwerte zu berechnen. Wünschenswert ist die Berechnung aller Attributwerte beim einmaligen Durchlauf durch den Baum, z. B. parallel zur syntaktischen Analyse. Dies wird durch durch links-attributierte Grammatiken sichergestellt [Aho et al. 88, Kap. 5.4]. Bei links-attributierten Grammatiken hängt der Wert eines ererbten Attributs nur von den ererbten Attributen der Vorgängerknoten und den abgeleiteten Attributen der im Vorgängerknoten links von diesem Knoten stehenden Knoten ab. Im obigen Beispiel dürfte im Knoten *addNode* das ererbte Attribut $val_2.env$ von dem abgeleiteten Attribut $val_1.type$ abhängig sein, aber nicht umgekehrt $val_1.env$ von $val_2.type$.

Links-attributierte Grammatiken korrespondieren sowohl mit LL-Grammatiken und Generatoren für Top-Down-Parser als auch mit LR-Grammatiken und Generatoren für Bottom-Up-Parser [Watt 77]. In dieser Arbeit werden links-attributierte Grammatiken zur *Definition* von Syntaxerweiterungen und entsprechende Parser-Generatoren zur *Implementierung* der Syntaxerweiterungen eingesetzt.

#### 2.2.4.2 Prädikate

Zur Analyse der statischen Semantik können Regeln formuliert werden, die festlegen, welche abstrakten Syntaxbäume wohlgeformt sind [Hennessy 90; Matthes 93]. Ein automatischer Beweiser kann mit Hilfe der Regeln die Wohlgeformtheit eines Syntaxbaums überprüfen.

Das folgende Beispiel zeigt Regeln für wohlgeformte Syntaxbäume:

*[IntNode]*

$$\frac{}{Env \vdash intNode(int) : Int}$$

*[IdNode]*

$$\frac{id \notin Env'}{(Env, id : Int, Env') \vdash idNode(id) : Int}$$

*[AddNode]*

$$\frac{Env \vdash val_1 : Int \quad Env \vdash val_2 : Int}{Env \vdash addNode(val_1\ val_2) : Int}$$

Für Syntaxbäume v der Sorte *val* wird ein Prädikat $v : Int$ definiert. Das Prädikat wird von jedem Knoten *intNode* erfüllt, wie die Regel [IntNode] angibt. Ein Knoten *idNode* erfüllt das Prädikat, wenn in der Umgebung dem Bezeichner *id* der Typ *Int* zugeordnet ist und der Bezeichner im Rest der Umgebung *Env'* nicht redefiniert wird. Gemäß der Regel [AddNode] erfüllt ein Knoten *addNode* das Prädikat, wenn beide Unterbäume in der gleichen Umgebung *Env* das Prädikat erfüllen.

Diese Art der Beschreibung der statischen Semantik wird in Anhang B für die in dieser Arbeit definierten Sprachen angewendet.

### 2.2.5 Zwischenkode-Erzeugung

Die Erzeugung des Zwischenkodes aus dem Syntaxbaum läßt sich wie die Typprüfung durch attributierte Grammatiken beschreiben [Aho et al. 88, Kap. 8], wie das folgende Beispiel zeigt:

| | | | |
|---|---|---|---|
| *val.env* | *::=* | *nilNode()* | *=> val.code := nop* |
| | \| | *intNode(int)* | *=> val.code := push(int)* |
| | \| | *idNode(id)* | *=> val.code := load(val.env id)* |
| | \| | *addNode($val_1$ $val_2$)* | *=> val.code := seq($val_1$.code $val_2$.code add)* |

Jeder Knoten des Syntaxbaums erhält ein weiteres Attribut *code*, das die Werte *nop* (keine Operation), *push(int)* (lege die Zahl *int* auf dem Keller ab), *load(env id)* (lege den durch den Bezeichner *id* in der Umgebung *env* benannten Wert auf dem Keller ab), *add* (addiere die beiden obersten Elemente des Kellers und lege das Ergebnis auf dem Keller ab) und *seq($x_i$)* (führe die Operationen $x_i$ nacheinander aus) annehmen kann. Der Knoten *nilNode* erzeugt den Befehl *nop* und der Knoten *intNode* den Befehl *push* mit dem Zahlenwert-Attribut als Argument. Der Knoten *idNode* führt dazu, daß ein durch den Bezeichner *id* benannter Wert auf dem Keller abgelegt wird. Der Knoten *addNode* erzeugt eine Befehlssequenz aus den Befehlen der beiden Unterbäume und dem Befehl *add*.

### 2.2.6 Zusammenfassung und Bewertung

Eine Syntaxerweiterung ist ein Eingriff in das Frontend. Wenn das Frontend handkodiert in einer Programmiersprache vorliegt, muß der Programmkode geändert werden, was aufgrund der Komplexität des Frontends sehr schwierig und fehleranfällig ist (siehe Kapitel 1.1.2).

Mit den in diesem Kapitel vorgestellten Beschreibungstechniken läßt sich ein Compiler-Frontend phasenspezifisch beschreiben und automatisch erzeugen. Die Beschreibung kann geändert und so die Syntax einer Sprache erweitert werden. Weitgehende Konsistenzprüfungen durch die Generatoren schränken die Fehlermöglichkeiten ein.

Die Beschreibung einer Sprache mit diesen Techniken ist sehr komplex, so daß eine Änderung schwierig ist (siehe z. B. [Rechenberg, Mössenböck 88; Reps, Teitelbaum 89; Tofte 90]). Außerdem fehlt ein Konzept zur modularen Erweiterung einer Sprache und zur Benutzung verschiedener Frontends.

Im folgenden wird ein Ansatz zur Integration von Syntaxerweiterungen in das Compilermodell unter Benutzung der vorgestellten Techniken entwickelt. Der Ansatz ist in seiner Komplexität eingeschränkt, damit die Syntaxerweiterungen auch für Anwendungsprogrammierer zugänglich sind und Fehler möglichst früh erkannt werden. Mit Hilfe dieses Ansatzes kann die modulare Erweiterung einer Sprache unter Beachtung von Benennungs-, Bindungs- und Typisierungsinvarianzen beschrieben werden, d. h. daß eine Sprache schrittweise erweitert werden kann und bei jeder Erweiterung die grundlegenden Konzepte der Sprache (bzgl. Benennung, Bindung und Typisierung) erhalten bleiben.

## 2.3 Integration der syntaktischen Erweiterbarkeit in das Compilermodell

In diesem Abschnitt werden die Möglichkeiten zur Integration syntaktischer Erweiterbarkeit in das entwickelte Compilermodell beschrieben und bewertet. Ansatzpunkte für Syntaxerweiterungen sind die in Abschnitt 2.1.2 und Abbildung 2.2 beschriebenen Phasen und Repräsentationen [Cheatham 66; Solntseff, Yezerski 71]. Es wird untersucht, wie die Erweiterung einer Phase beschrieben werden kann und welche Auswirkungen sich daraus ergeben.

Bei einer Syntaxerweiterung sind zwei Zeitpunkte bzw. Aktionen zu unterscheiden [Leavenworth 66; Sakharov 92]: Die *Definition* einer Syntaxerweiterung beschreibt ein neues syntaktisches Konstrukt und seine semantische Interpretation. Eine Benutzung des neuen syntaktischen Konstrukts führt zur *Expansion* des Konstrukts gemäß der semantischen Interpretation.

Ein System zur Syntaxerweiterung umfaßt zwei Sprachen [Solntseff, Yezerski 71]: Syntaxerweiterungen werden in einer *Erweiterungssprache* formuliert. Syntaxerweiterungen beziehen sich auf die Änderung der Syntax einer *Basissprache*.

### 2.3.1 Definition einer Syntaxerweiterung

Syntaxerweiterungen führen zu Änderungen einer oder mehrerer der in Abschnitt 2.1.2 beschriebenen Übersetzungsphasen des Frontends. Vereinfachend kann angenommen werden, daß eine Syntaxerweiterung ein neues Frontend (als Differenz zu einem vorherigen Frontend) beschreibt [Cheatham 66].

Für die Erweiterungssprache existiert wie für die Basissprache ein Compiler. Das Backend des Compilers der Erweiterungssprache erzeugt ein Frontend für den Compiler der Basissprache. Falls der Compiler der Erweiterungssprache auch sein eigenes Frontend ändern kann, wird dies *Reflektion* in der Erweiterungssprache genannt [Solntseff, Yezerski 71], in Analogie zur Reflektion in der Basissprache [Stemple et al. 91].

Wenn ein Compiler sowohl die Basis- als auch die Erweiterungssprache übersetzt und Konstrukte aus beiden Sprachen gemischt werden können, stellt sich die Frage nach der Semantik gemischter Konstrukte. Aus dem oben vorgestellten Compilermodell mit strikt sequentiellem Ablauf ergibt sich, daß eine Syntaxerweiterung keine Auswirkungen auf im gleichen Compilerlauf behandelte Konstrukte der Basissprache haben kann, da die Änderungen am Frontend durch das Backend vorgenommen werden und zum Zeitpunkt der Änderungen schon der gesamte Quelltext im Zwischenkode vorliegt. Soll eine andere Semantik implementiert werden, wie z. B. in den Ansätzen modifizierbarer Grammatiken [Burshteyn 90; Christiansen 90; Cabasino et al. 92], muß das Compilermodell geändert werden. Im Rahmen

dieser Arbeit wird davon abgesehen, da eine Änderung den Einsatz bekannter Technologien erschweren würde.

Im folgenden wird angenommen, daß die Definition einer Syntaxerweiterung ein neues Frontend erzeugt und bei der Übersetzung eines Quelltextes in der Basissprache angegeben werden kann, welches Frontend benutzt werden soll [Cheatham 66].

### 2.3.2 Expansion einer Syntaxerweiterung

Die konkrete Syntax einer Sprache wird durch die ersten beiden Phasen des Frontends (Scanner und Parser) erkannt, wie in Abbildung 2.2 gezeigt. Die Abstraktion transformiert die konkrete in die abstrakte Syntax, verbindet also beide Bereiche. Typprüfung und Zwischenkode-Erzeugung hängen allein von der abstrakten Syntax ab.

Änderungen der konkreten Syntax wirken sich auf die ersten drei Phasen aus, Änderungen der abstrakten Syntax auf die letzten drei. Im folgenden wird davon ausgegangen, daß aufgrund der in Abschnitt 2.2.6 geschilderten hohen Komplexität der Beschreibungen der abstrakten Syntax, der statischen Semantik und der Zwischenkode-Erzeugung die abstrakte Syntax unverändert bleibt. Vorausgesetzt wird, daß die Basissprache algorithmisch vollständig ist und über eine statische Semantik verfügt, die z. B. die Konzepte Polymorphismus, Datenabstraktion und Orthogonalität unterstützt [Atkinson, Bunemann 87; Matthes, Schmidt 91]. Gleichzeitig wird damit sichergestellt, daß die Sprache konzeptuell invariant bleibt, also in der erweiterten Sprache nicht unterschiedliche und evtl. unverträgliche Konzepte vermischt werden, was zu Verwirrung und Fehlern führen könnte.

Als Folge dieser Entscheidung kann kein neues syntaktisches Konstrukt einen neuen abstrakten Syntaxbaum erzeugen, sondern muß auf vorhandene abstrakte Syntaxbäume abgebildet werden. Dieser Abbildungsprozeß kann erst nach dem Erkennen eines syntaktischen Konstrukts, also nach der syntaktischen Analyse, stattfinden. Naheliegend ist die Erweiterung der Abstraktionsphase um die Abbildung der neuen syntaktischen Konstrukte in vorhandene abstrakte Syntaxbäume.

Wenn eine Änderung des strikt sequentiellen Ablaufs des Compilermodells in Betracht gezogen wird, kann die Expansion der neuen syntaktischen Konstrukte in Repräsentationen einer vorhergehenden Phase stattfinden. Abbildung 2.3 zeigt, wie Abbildung 2.2 prinzipiell modifiziert werden muß, um die Expansion einer Syntaxerweiterung in die Repräsentation einer vorhergehenden Phase zu erfassen. Dabei ist es unerheblich, in welcher Phase die Expansion stattfindet und welche der vorhergehenden Repräsentationen Ziel der Expansion ist. Der Übersetzungsprozeß enthält eine Schleife, die erst durch das Erkennen von Konstrukten der Basissprache beendet wird.

Diese Scheife ermöglicht schrittweise Transformationen (*term rewriting*), wie in den folgenden Abschnitten noch weiter ausgeführt wird. Die Terminierung dieser Schleife kann nicht ohne Einschränkungen gewährleistet werden, denn ein Konstrukt könnte in eine äquivalente Repräsentation von sich selbst abgebildet werden. Zum Zeitpunkt der Definition einer

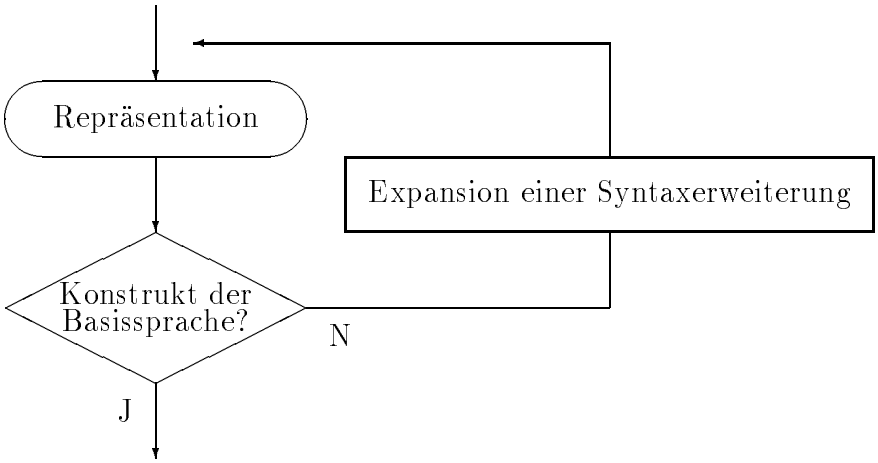



Abbildung 2.3: Syntaxerweiterungsschleife im Compilerlauf

Syntaxerweiterung sollte sichergestellt werden, daß die Expansion der Syntaxerweiterung terminiert. Da das Gebiet der Transformationen noch einige ungelöste Probleme umfaßt [Dershowitz et al. 91] und somit nicht als Basistechnologie zur Verfügung steht, wird dieser Ansatz hier nicht weiterverfolgt. Aus diesem und den in Abschnitt 2.5 genannten Gründen wird das strikt sequentielle Compilermodell beibehalten.

Zusammenfassend werden durch eine syntaktische Erweiterung in diesem eingeschränkten Modell die Phasen lexikalische und syntaktische Analyse sowie Abstraktion geändert. Die Beschreibung dieser Änderungen ist das Thema der nächsten beiden Abschnitte.

## 2.4 Beschreibung der syntaktischen Erweiterung

Eine syntaktische Erweiterung besteht nach [Cheatham 66; Leavenworth 66] aus

- einem syntaktischen Konstrukt und
- einer semantischen Interpretation des Konstrukts.

Die neuen Syntaxkonstrukte sollen gleichberechtigt neben vorhandenen Syntaxkonstrukten existieren, also sollte in der Beschreibungsform die gleiche Mächtigkeit wie bei der Beschreibung der Sprachsyntax vorliegen. Im Extremfall kann sogar die ganze Syntax der Sprache neu definiert werden. Zur Syntaxbeschreibung werden deshalb die in Abschnitt 2.2.2 beschriebenen kontextfreien Grammatiken eingesetzt.

Erkennungsalgorithmen für die Beschreibungen lassen sich mit Scanner- und Parser-Generatoren erzeugen. Der Generator sollte sicherstellen, daß zu jedem syntaktischen Konstrukt eindeutig hergeleitet werden kann, durch welche Produktion es erkannt wird, so daß

die entsprechende semantische Interpretation ausgewählt werden kann. Deshalb wird die Klasse der verarbeiteten Grammatiken, wie in Abschnitt 2.2.2 beschrieben, von dem verwendeten Parser-Generator eingeschränkt.

An dieser Stelle kann nicht entschieden werden, welche Grammatik- und Beschreibungsform gewählt werden sollte, da diese von den Anforderungen des verwendeteten Parser-Generators abhängen. Im folgenden wird keine spezielle Beschreibungsform benutzt und damit kein spezieller Generator unterstützt. An den Stellen, an denen ein spezieller Generator nötig ist, um die beschriebenen syntaktischen Konstrukte zu verarbeiten, wird dies angemerkt.

Die zur Definition eines syntaktischen Konstrukts eingesetzte Notation wird als Muster (*pattern*) bezeichnet. Muster entsprechen kontextfreien Produktionen (siehe Abschnitt 2.2.2), enthalten aber zusätzlich neben Terminalen und Nonterminalen noch Platzhalter zur Benennung von Konstrukten, die durch die Produktionen erzeugt werden:

"for" "each" *id* = *ideG* "in" *table* = *valG* ":" *pred* = *valG* "do" *act* = *bndsG* "end"

Dieses Muster enthält die in Anführungszeichen eingeschlossenen Terminale **for, each, in, :, do, end**, die Platzhalter *id, table, pred* und *act*, sowie die Nonterminale *ideG, valG* und *bndsG*.

Die Platzhalter benennen die Konstrukte, die von den durch Nonterminale benannten kontextfreien Produktionen erkannt bzw. von den semantischen Interpretationen der Produktionen erzeugt werden. Durch die Benennung können sie in die semantische Interpretation dieser Produktion einfließen. Platzhalter können auch als Benennung von abgeleiteten Attributen der in Abschnitt 2.2.4.1 eingeführten attributierten Grammatiken betrachtet werden, während die semantischen Interpretationen die Berechnungsvorschriften für die abgeleiteten Attribute darstellen. Obwohl im folgenden der Übersichtlichkeit halber nur ein abgeleitetes Attribut benutzt wird, sind mehrere abgeleitete Attribute und auch ererbte Attribute möglich; siehe dazu Kapitel 3.2.

Die semantische Interpretation folgt durch => getrennt von der Syntaxregel:

"for" "each" *id* = *ideG* "in" *table* = *valG* ":" *pred* = *valG* "do" *act* = *bndsG* "end"
=> ... *(* semantische Interpretation *)*

Die möglichen Formen der semantischen Interpretation werden im Abschnitt 2.5 diskutiert.

Durch die Zuordnung von Platzhaltern zu Produktionen erhalten die Platzhalter Sorten, die in [Cheatham 66] „syntaktische Typen“ genannt werden. Anhand der Sorten kann die korrekte Verwendung der Platzhalter in der semantischen Interpretation überprüft werden, falls die Form der semantischen Interpretation dies zuläßt. Die Definition der statischen Semantik der in dieser Arbeit definierten Erweiterungssprache in Anhang B.2 sieht eine Prüfung der semantischen Interpretation auf Sortenkonformität vor.

Die Beschreibung einer Syntaxerweiterung mit kontextfreien Grammatiken steht im Gegensatz zu den eingeschränkten Erweiterungsformen der Makro-Sprachen LISP [Steele 90]

und ANSI-C [Kernighan, Ritchie 90], die Syntaxerweiterungen nur in der Form und an Stelle von Funktionsaufrufen zulassen, z. B.:

*(foreach p in persons do printPerson)* — *LISP-Makro oder Funktionsaufruf*
*foreach(p, persons, printPerson)* — *ANSI-C–Makro oder Funktionsaufruf*

Vorteil der eingeschränkten Makro-Sprachen ist die einfache Realisierung durch Syntaxtabellen, in welche die Makros eingetragen werden [Clinger, Rees 91, S. 161]. Nachteilig wirkt sich aus, daß Überprüfungen der Makros z. B. auf Sortenkonformität allein der Verantwortung des Makro-Entwerfers unterliegen, falls sie überhaupt möglich sind, und neue syntaktische Konstrukte, die sich wie das **for each**-Konstrukt orthogonal in die Syntax der Sprache einfügen sollen, nicht darstellbar sind.

## 2.5 Beschreibung der semantischen Interpretation

In diesem Abschnitt wird diskutiert, in welcher Form semantische Interpretationen angegeben werden können, welche Phasen sie betreffen und in welche Repräsentationen sie expandieren. Die folgenden Abschnitte orientieren sich an den in Abschnitt 2.1.2 und in Abbildung 2.2 besprochenen Repräsentationen. Im Falle der in Abbildung 2.3 dargestellten Expansion einer Syntaxerweiterung in eine Repräsentation einer vorhergehenden Phase muß das Compilermodell gemäß Abschnitt 2.3.2 angepaßt werden. Es sind dann Transformationen möglich. Die daraus zusätzlich zu dem Termininierungsproblem entstehenden Nachteile werden dargestellt.

### 2.5.1 Zeichenfolge

Als semantische Interpretation werden Quelltexte und Platzhalter angegeben, die dann von der lexikalischen Analyse weiterverarbeitet werden.

Das folgende Beispiel zeigt einen Quelltext als semantische Interpretation:

"for" "each" *id* = *ideG* "in" *table* = *valG* ":" *pred* = *valG* "do" *act* = *bndsG* "end"
=> "iter.forEach(" *table* "fun(" *id* ":?)" *pred* "fun(" *id* ":?) begin" *act* "end)"

An Stelle der Platzhalter in der semantischen Interpretation werden Quelltexte (Zeichenfolgen) eingesetzt.

[Clinger, Rees 91] führen als Problem dieser „Text-Makros“ die Mißachtung der lexikalischen und syntaktischen Struktur der Sprache an, so daß je nach Kontext lexikalisch oder syntaktisch falscher Text erzeugt werden kann.

Das folgende Beispiel zeigt ein lexikalisch falsches Text-Makro (nach [Clinger, Rees 91, S. 155]):

"foo" => "'salad"
**foo** *bar'*
**foo foo** *bar'*
wird umgewandelt zu:

*'salad bar'*
*'salad 'salad bar'*

Während die erste Expansion von *foo* einen lexikalisch korrekten Ausdruck (eine Zeichenkette) ergibt, werden im zweiten Fall eine Zeichenkette *'salad '*, zwei Bezeichner *salad* und *bar* sowie ein lexikalischer Fehler (Zeichenkette ohne abschließendes Anführungszeichen) erkannt.

Die in Abbildung 2.3 gezeigte Schleife im Compilerlauf vorausgesetzt, erlaubt dieser Ansatz eine schrittweise Transformation einer Syntaxerweiterung, wie das folgende Beispiel zeigt:

"(" *head* = *valG* *rest* = *valListG* ")"
=> "(" *head* ")" "(" *rest* ")"
*(x y z)*
wird umgewandelt zu:

*(x)(y z)*
wird umgewandelt zu:

*(x)(y)(z)*

Problematisch ist die Überprüfung des Terminierens einer solchen Transformation. Wenn beispielsweise in der obigen Syntaxerweiterung die Produktion *valListG* auch eine leere Liste erlaubt, werden unendlich viele leere Klammerpaare erzeugt:

*(x)()()...(y z)*

### 2.5.2 Symbolfolge

Die semantische Interpretation besteht aus einer Symbolfolge mit Platzhaltern, welche von der syntaktischen Analyse weiterverarbeitet wird.

Das folgende Beispiel zeigt eine Symbolfolge als semantische Interpretation:

"for" "each" *id* = *ideG* "in" *table* = *valG* ":" *pred* = *valG* "do" *act* = *bndsG* "end"
=> *id*("iter") *kw*(".") *id*("forEach") *kw*("(") *ph(table)*
*kw*("fun") *kw*("(") *ph(id)* *kw*(":") *kw*("?") *kw*(")") *ph(pred)*
*kw*("fun") *kw*("(") *ph(id)* *kw*(":") *kw*("?") *kw*(")")
*kw*("begin") *ph(act)* *kw*("end") *kw*(")")

In dem Beispiel werden diese Symbolklassen verwendet:

| | |
|---|---|
| *kw* | Schlüsselwort |
| *id* | Bezeichner |
| *ph* | Platzhalter |

Die Platzhalter in der semantischen Interpretation werden durch Symbolfolgen ersetzt.

Dieser Ansatz respektiert die lexikalische Struktur, so daß die Symbolfolge der semantischen Interpretation lexikalisch korrekt ist. Nachteil ist nach [Clinger, Rees 91, S. 156] die Mißachtung der syntaktischen Struktur, so daß syntaktisch falsche Symbolfolgen möglich sind.

Das folgende Beispiel zeigt eine syntaktisch falsche Symbolfolge als semantische Interpretation:

"foo" => *kw*(")") *kw*("(")
*f*(**foo**)
**foo**
wird umgewandelt zu:

*f*()()
)(

Während jedes einzelne Symbol lexikalisch korrekt ist, ergibt die Expansion von *foo* nur im ersten Fall eine syntaktisch korrekte Symbolfolge.

Wie im vorigen Abschnitt sind Transformationen möglich, deren Termininierung gesondert zugesichert werden muß.

### 2.5.3 Parsebaum

Die semantische Interpretation besteht aus einem Parsebaum mit Platzhaltern, der zu einem abstrakten Syntaxbaum weiterverarbeitet wird.

Das folgende Beispiel zeigt Parsebäume als semantische Interpretation (Abbildung 2.4 zeigt den Parsebaum in einer anderen Darstellung):

"for" "each" *id* = *ideG* "in" *table* = *valG* ":" *pred* = *valG* "do" *act* = *bndsG* "end"
=> *valG*(*applyG*(
  *valG*(*dotG*(
    *valG*(*ideG*(*id*("iter"))) "."
    *valG*(*ideG*(*id*("forEach")))))
  "(" *valListG*(*valG*(*ph*("table"))
    *valListG*(*valG*(*funG*("fun" "("
      *ideG*(*ph*("id")) ":" *ideG*(*id*("?")) ")"
      *valG*(*ph*("pred"))))
    *valListG*(*valG*(*funG*("fun" "("
      *ideG*(*ph*("id")) ":" *ideG*(*id*("?")) ")"
      *valG*("begin" *bndsG*(*ph*("act")) "end")))
    *valListG*(*ϵ*))))
  ")"))

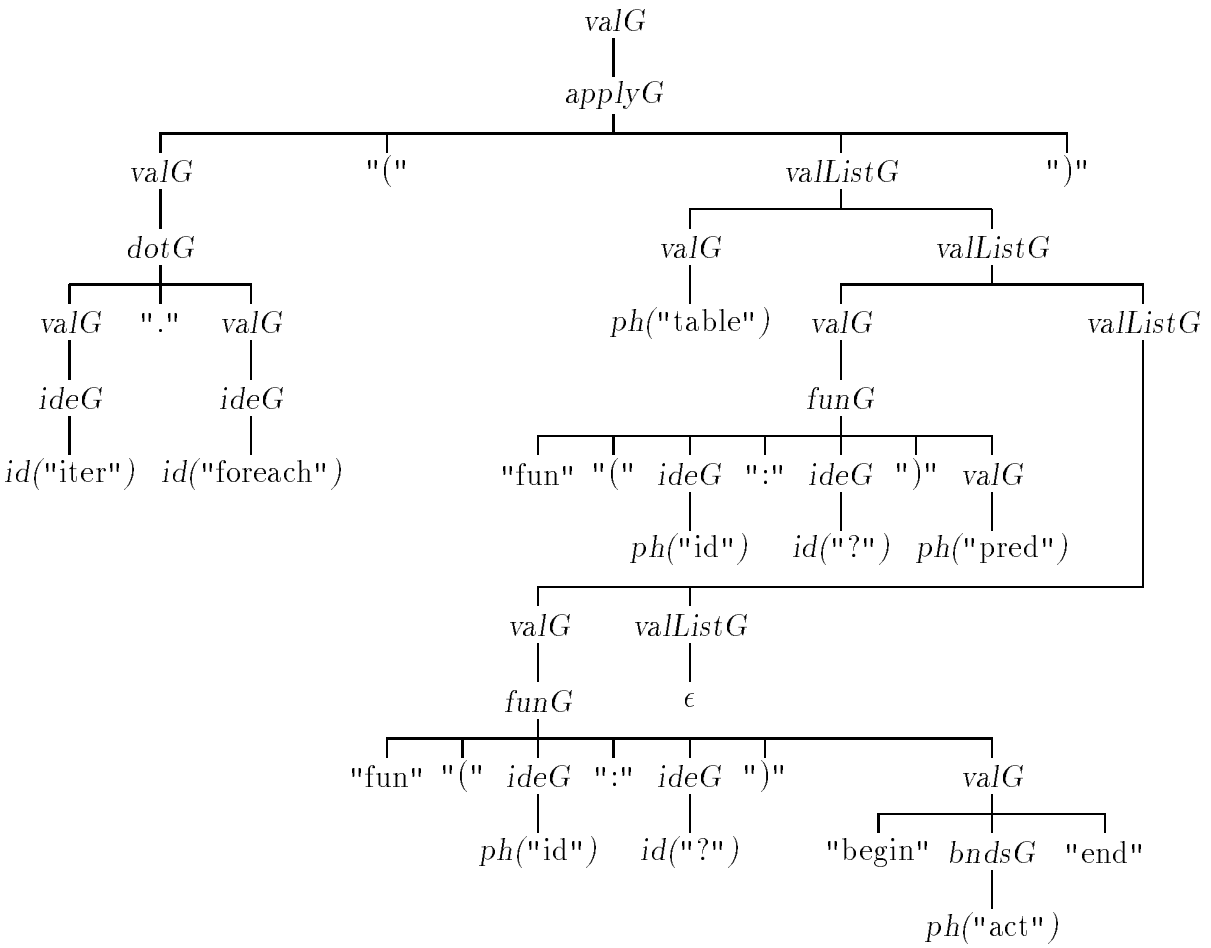



Abbildung 2.4: Parsebaum

Die Platzhalter in der semantischen Interpretation werden durch Parsebäume ersetzt.

In dem Beispiel werden diese kontextfreien Produktionen verwendet:

| | | |
|---|---|---|
| *valG* | ::= | *ideG* \| *applyG* \| *dotG* \| *funG* \| "begin" *bndsG* "end" \| *ph* |
| *ideG* | ::= | *id* \| *ph* |
| *applyG* | ::= | *valG* "(" *valListG* ")" |
| *valListG* | ::= | $\epsilon$ \| *valG valListG* |
| *dotG* | ::= | *valG* "." *valG* |
| *funG* | ::= | "fun" "(" *ideG* ":" *ideG* ")" *valG* |
| *bndsG* | ::= | ... \| *ph* |

Die textuelle Darstellung des Parsebaums ist nicht benutzergerecht und die in Abbildung 2.4 gezeigte Darstellung läßt sich beim textuellen Programmieren nicht einsetzen. Deshalb könnte zum Beispiel die folgende textuelle Darstellung eingesetzt werden, die kürzer und anschaulicher ist als die oben gezeigte:

"for" "each" *id* = *ideG* "in" *table* = *valG* ":" *pred* =*valG* "do" *act* = *bndsG* "end"
=> **valG** *iter.forEach(***ph** *table* **fun***(***ph** *id :?)* **ph** *pred* **fun***(***ph** *id :?)* **begin ph** *act* **end***)*

Hier benennt **valG** die Produktion, von der die semantische Interpretation akzeptiert wird, während **ph** Platzhalter kennzeichnet, die durch einen Parsebaum zu ersetzen sind.

Diese Darstellung berücksichtigt die syntaktische Struktur der Sprache, so daß es unmöglich ist, syntaktisch falsche semantische Interpretationen zu konstruieren, wenn die Sortenkonformität der Platzhalter überprüft wird. Trotzdem sind semantische Fehler möglich, wenn die Sprachsemantik des Bindens von Bezeichnern nicht beachtet wird [Clinger, Rees 91], wie das folgende Beispiel zeigt:

"swap" x = *ideG* "and" *y* = *ideG*
=> **valG begin let** *tmp* = **ph** *x* **ph** *x* := **ph** *y* **ph** *y* := *tmp* **end**
**swap** *foo* **and** *tmp*
wird umgewandelt zu:

**begin let** *tmp* = *foo* *foo* := *tmp* *tmp* := *tmp* **end**

In diesem Beispiel wird eine Syntaxerweiterung definiert, die einen lokalen Bezeichner *tmp* enthält. Bei der Expansion der Syntaxerweiterung kommt es zu einem Konflikt mit dem globalen Bezeichner *tmp*. Es gibt noch weitere Konfliktmöglichkeiten, die gelöst werden müssen. In Kapitel 5 wird näher darauf eingegangen.

Wenn in der semantischen Interpretation die durch das Muster der Syntaxerweiterung definierten Produktionen verwendet werden dürfen, sind wie in den vorherigen Abschnitten Transformationen möglich, deren Terminierung zugesichert werden sollte.

Die Syntaxerweiterungssprache TXL [Cordy et al. 91] verwendet die Transformation von Parsebäumen, ohne die Terminierung einer Syntaxerweiterung zu garantieren. Dadurch erreicht sie die algorithmische Vollständigkeit, so daß die Erweiterungssprache eine eigene Programmiersprache darstellt.

### 2.5.4 Syntaxbaum

Die semantische Interpretation besteht aus einem Syntaxbaum mit Platzhaltern, der von der semantischen Analyse weiterverarbeitet wird. Damit entspricht die semantische Interpretation der Definition der Abstraktion.

Das folgende Beispiel zeigt abstrakte Syntaxbäume als semantische Interpretation:

"for" "each" *id* = *ideG* "in" *table* = *valG* ":" *pred* = *valG* "do" *act* = *bndsG* "end"
=> *val(apply(apply(apply(*
*val(dot(val(ide(id(*"iter"*))) val(ide(id(*"forEach"*)))))*
*val(ph(*"table"*)))*
*val(fun(ide(ph(*"id"*)) ide(id(*"?"*)) val(ph(*"pred"*)))))*
*val(fun(ide(ph(*"id"*)) ide(id(*"?"*)) val(bnds(ph(*"act"*)))))))*

Die Platzhalter in der semantischen Interpretation werden durch Syntaxbäume ersetzt.

In dem Beispiel werden diese Konstruktoren für abstrakte Syntaxbäume benutzt:

| | | |
|---|---|---|
| *val* | ::= | *ide* \| *apply* \| *dot* \| *fun* \| *bnds* \| *ph* |
| *ide* | ::= | *id* \| *ph* |
| *apply* | ::= | *val val* |
| *dot* | ::= | *val val* |
| *fun* | ::= | *ide ide val* |
| *bnds* | ::= | ... \| *ph* |

Im Vergleich zum vorherigen Abschnitt anstelle der Produktionen hier die Konstruktoren der abstrakten Syntaxbäume. Eine nachzubildende Leistung der Abstraktion ist in diesem Beispiel die Auflösung der Parameterliste *valListG* eines Funktionsaufrufs *applyG* durch eine Folge von Funktionsaufrufen *apply* mit einem Parameter *val*. In [Matthes 93] wird dieser Schritt als Normalisierung bezeichnet.

Dieselbe vereinfachende Darstellung wie bei den Parsebäumen läßt sich auch hier anwenden. Ebenso können hier Bindungsprobleme auftreten. Es sind keine Transformationen möglich, da die semantische Interpretation nicht aus (erweiterbaren) Produktionen, sondern aus (festen) abstrakten Syntaxbäumen besteht. Somit ist eine Nicht-Terminierung bei der Expansion einer Syntaxerweiterung ausgeschlossen.

In der vereinfachten Darstellung der Syntaxbäume ist es möglich, vorherige Syntaxerweiterungen zu benutzen, da diese wiederum abstrakte Syntaxbäume erzeugen, wie im folgenden Beispiel gezeigt wird:

"foo1" => *f(1)*
"foo2" => **foo1** + *1*

Dieses Beispiel setzt voraus, daß die erste Syntaxerweiterung durchgeführt wurde, bevor die zweite Syntaxerweiterung definiert wird. Eine Rekursion der Produktionen ist somit ausgeschlossen. In der Praxis treten aber rekursive Produktionen auf, bei denen eine Syntaxerweiterung nur sinnvoll ist, wenn alle Produktionen gleichzeitig zur Syntaxerweiterung herangezogen werden, wie das folgende Beispiel zeigt (siehe auch die konkrete Syntax von TLMin in Anhang A):

*valG* ::= "begin" *bnds* = *bndsG* "end" ...
*bndsG* ::= "let" *id* = *ideG* "=" *val* = *valG* ...

Deshalb sollten die Produktionen einer Syntaxerweiterung zu einem Block oder Modul zusammengeschlossen werden, so daß sie gemeinsam zu einer Syntaxerweiterung der Basissprache führen, die in weiteren Syntaxerweiterungsblöcken bzw. -modulen benutzt werden kann.

Beim Parsen können rekursive Produktionen zu Endlosschleifen führen, z. B. durch folgende Grammatik:

*g1* ::= *g2*
*g2* ::= *g1*

Um dies auszuschließen, werden die Produktionen von einem Parser-Generator überprüft. Wie in Abschnitt 2.2.2 beschrieben, können unterschiedliche Parser-Generatoren unterschiedliche Anforderungen an die Grammatik stellen.

Dieses Verfahren zur Darstellung der semantischen Interpretation wird in der vorliegenden Arbeit weiterverfolgt. Siehe dazu Kapitel 3.

### 2.5.5 Meta-Operationen

Die bisher gezeigten semantischen Interpretationen bestanden aus statischen Objekten mit Platzhaltern. Die dynamische Konstruktion von Objekten erlauben Meta-Operationen. Eine Meta-Operation ist eine Operation, die von dem Syntaxerweiterungsalgorithmus, in diesem Fall meist Makro-Prozessor genannt [Layzell 85], zum Zeitpunkt der Expansion der Syntaxerweiterung ausgeführt wird. Zum Beispiel könnte eine Expansion abhängig von einer Bedingung sein oder eine Liste aufgelöst werden [Leavenworth 66; Kohlbecker 86]. Das folgende Beispiel zeigt, wie mit Hilfe einer Meta-Operation **FOR EACH** zur Listenbearbeitung eine Liste von Bezeichnern *ideList* und ein Wert *val* in eine Liste von Bezeichnern und Werten umgewandelt werden, wobei der in der semantischen Interpretation eingeführte Bezeichner *ide* wie ein Platzhalter behandelt wird:

*ideList* = *ideListG* "=" *val* = *valG*
=> **FOR EACH** *ide* **IN** *ideList* **DO**
**ph** *ide* = **ph** *val*

[Kohlbecker 86] verwendet eine implizite Umwandlung von Listen, die durch "..." dargestellt wird, wie das folgende Beispiel zeigt:

*ide* = *ideG* ... "=" *val* = *valG*
=> **ph** *ide* = **ph** *val* ...

Die Semantik dieser Listenoperationen ist insbesondere im Falle verschachtelter Listen nicht eindeutig, wie Kohlbecker [Kohlbecker 86, Kap. 5] zugibt.

Gegenüber den Transformationen besitzen Meta-Operationen den Vorteil, daß ihre Terminierung normalerweise garantiert werden kann. Meta-Operationen stellen eine Erweiterung des Expansionsalgorithmus dar und lassen sich ohne Schwierigkeiten in jede der besprochenen Alternativen zur Syntaxerweiterung einfügen. Es bleibt zu prüfen, welche zusätzlichen Möglichkeiten sich durch die Verwendung von Meta-Operationen ergeben bzw. welche Meta-Operationen sinnvoll sind. In der vorliegenden Arbeit werden keine Meta-Operationen verwendet, da diese Fragen noch nicht ausreichend untersucht wurden.

### 2.5.6 Semantik und Zwischenkode

Der mächtigste und dadurch auch komplexeste Weg zur Beschreibung einer Syntaxerweiterung ist die Definition eines neuen Knotens des abstrakten Syntaxbaums für das neue

Syntaxkonstrukt. Dazu gehören die Definition der Attribute des Knotens und Berechnungsvorschriften für die Attribute zur Prüfung der statischen Semantik und zur Übersetzung in Zwischenkode. Im Unterschied zu Abschnitt 2.2.6 könnte an dieser Stelle eine Integration in die Sprache und das Compilermodell erfolgen.

Dieser Ansatz verlegt die gesamte Verantwortung für die Semantik und somit die in Abschnitt 2.5.3 angesprochenen Bindungsprobleme auf den Anwendungsprogrammierer, der die Syntaxerweiterung vornimmt. Die Beschreibungsformen sind komplex und für einen Anwendungsprogrammierer schwer beherrschbar; Beispiele finden sich in [Rechenberg, Mössenböck 88; Reps, Teitelbaum 89; Tofte 90].

Dieser in den Bereich der Compiler-Compiler vorstoßende Ansatz erfüllt nicht die Voraussetzung dieser Arbeit, Syntaxerweiterungen durch einen Anwendungsprogrammierer zu erlauben, und wird deshalb nicht weiter ausgeführt.

### 2.5.7 Reflektion

Bei reflektiven Sprachen wie (erweitertem) Napier 88 [Kirby 92], LISP [Steele 90] oder TRPL [Sheard 90; TRPL 90; Stemple, Sheard 91; Stemple et al. 92a; Stemple et al. 91] kann die semantische Interpretation aus einer Funktion bestehen, die zur Übersetzungszeit ausgewertet wird. Eine Syntaxerweiterung kann als besondere Form der Funktionsabstraktion aufgefaßt werden, wobei die Platzhalter die Parameter der Funktion sind. Je nach Compilermodell kann die Funktion als Rückgabewert z. B. abstrakte Syntaxbäume, geprüfte Syntaxbäume oder Zwischenkode liefern. Im Gegensatz zu Meta-Operationen stehen in der semantischen Interpretation alle Operationen der algorithmisch vollständigen Basissprache zur Verfügung.

Das folgende Beispiel zeigt eine Funktion als semantische Interpretation (angelehnt an TRPL [Stemple et al. 92b]):

"for" "each" *id* = *ideG* "in" *table* = *valG* ":" *pred* = *valG* "do" *act* = *bndsG* "end"
=> **begin**
  **let** *predVal* = **VAL**(**fun**(*id :?*) *pred*)
  **let** *actVal* = **VAL**(**fun**(*id :?*) **begin** *act* **end**)
  **VAL**(*iter.forEach*(*table predVal actVal*))
**end**

In der semantischen Interpretation werden zwei Funktionen *predVal* und *actVal* konstruiert und in den Bibliotheksaufruf *iter.forEach* aufgenommen. Die Funktion **VAL** dient zur Kennzeichnung von Werten, die nur zur Übersetzungszeit existieren. Das konstruierte Ergebnis *iter.forEach...* wird anstelle des Musters in den zu übersetzenden Text (in der jeweiligen Repräsentation, z. B. als Syntaxbaum) eingesetzt und erst dann endgültig übersetzt.

Dieser Ansatz besitzt die algorithmische Vollständigkeit, die durch die Sprache gegeben ist. Alle Syntaxerweiterungen werden innerhalb der dem Anwendungsprogrammierer bekannten Programmiersprache ausformuliert, so daß er keine Erweiterungssprache zusätzlich lernen muß. Der Programmierer muß die Datenstrukturen und die Operationen auf der genutzten Repräsentation, hier dem Syntaxbaum, kennen, und sich über die Semantik der Funktion **VAL** oder ähnlicher Mechanismen (z. B. Quote und Backquote in LISP) im klaren sein. Nicht-terminierende Expansionen sind möglich, wenn die Sprache volle algorithmische Berechenbarkeit und damit auch die Möglichkeit zur Programmierung von Endlosschleifen zur Verfügung stellt.

Bindungsprobleme werden nicht automatisch vermieden, sondern müssen vom Anwendungsprogrammierer aufgelöst werden, wie Beispiele in TRPL [Stemple et al. 92a; Stemple et al. 92b] oder LISP [Kohlbecker 86] zeigen. Wenn ein geeignetes reflektives System zur Verfügung steht, können dem Anwendungsprogrammierer Routinen zur Lösung von Bindungsproblemen zur Verfügung gestellt oder in den Reflektionsmechanismus eingebunden werden, wie dies [Kohlbecker 86; Kohlbecker et al. 86; Bawden, Rees 88; Clinger, Rees 91] für LISP darstellen.

## 2.6 Bindungsprobleme in Syntaxerweiterungen

Wie in den vorherigen Abschnitten gesehen, führen alle Arten der semantischen Interpretation einer Syntaxerweiterung zu Bindungsproblemen. Die Lösung dieser Probleme im Rahmen des beschriebenen Compilermodells ist ein zentraler Aspekt dieser Arbeit und wird deshalb getrennt in Kapitel 5 beschrieben. An dieser Stelle sei nur angemerkt, daß der in dieser Arbeit gewählte Ansatz zur Lösung der Bindungsprobleme folgende Teile des Compilers betrifft:

- Die abstrakten Syntaxbäume werden in gebundene und ungebundene Syntaxbäume aufgeteilt. Gebundene Syntaxbäume (Syntaxbäume mit Bindungsinformationen) entstehen bei der Expansion von Syntaxerweiterungen und können wiederum durch Instanziierung von Platzhaltern eingefügte, ungebundene Syntaxbäume enthalten.
- Der Bindungsalgorithmus der semantischen Analyse (Typprüfung) wird erweitert, so daß die in gebundenen Syntaxbäumen vorhandenen Bindungen erhalten bleiben.

## 2.7 Implementation mit Compilerbau-Werkzeugen

Die verwendbare Technologie wird dadurch eingeschränkt, daß die meisten Compilerbau-Werkzeuge auf bestimmte Sprachen zugeschnitten sind und nicht als generische Bibliotheken vorliegen. Sie können somit nicht für neue Sprachen angepaßt und in eigene Programme

eingebunden werden. [Sakharov 92], der einen ähnlichen Ansatz für die Sprache C auf der Basis von Yacc [Johnson 75] und Lex [Lesk, Schmidt 75] verfolgt, zeigt die durch diese Werkzeuge auferlegten Einschränkungen.

Daher wurden die im Rahmen des Tycoon-Projekts in Form des Tycoon-Compiler-Toolkits [Schröder, Matthes 92] zur Verfügung stehenden generischen Werkzeuge genutzt. Ausgangsbasis bildete die erste Version des TL-Compilers [Matthes 93].

Eine syntaktische Erweiterung hat in dem hier benutzten eingeschränkten Modell Auswirkungen auf die ersten drei Phasen des Frontends: Scanner, Parser und Abstraktion. Diese drei Teile werden bei einer Syntaxerweiterung neu erzeugt.

Der Scanner könnte ebenfalls durch einen Scanner-Generator erzeugt werden. Im TL-Compiler wurde ein ausprogrammierter Scanner verwendet, so daß hier die Definition eines Scanners nötig gewesen wäre, was im Rahmen dieser Arbeit nicht geleistet wurde. Stattdessen sind eine feste Menge von Symbolen und ihre Repräsentation vorgegeben. Die Menge der Schlüsselworte ist veränderbar, so daß neue Schlüsselworte definiert und nicht benutzte entfernt werden können.

Parser und Abstraktion werden in einem Schritt von einem Parser-Generator aus kontextfreien Produktionen mit semantischen Aktionen erzeugt. Im Tycoon-Compiler-Toolkit lag ein LL(1)-Parser-Generator [Waite, Goos 85] vor. Da die Benutzung dieses Parser-Generators schwerwiegende Effizienzverluste mit sich gebracht hätte, wurde ein *inkrementeller* LL(1)-Parser-Generator [Gyimóthy et al. 88; Horspool 88] implementiert. Vorteil dieses Ansatzes ist es, daß bei einer Änderung der Beschreibung des Parsers nicht der ganze Parser, sondern nur Teile neu erzeugt werden. Der Effizienzvorteil verschwindet, sobald bei umfangreichen Änderungen der Sprache große Teile des Parsers oder sogar der ganze Parser neu erzeugt werden müssen.

Die semantische Interpretation wird durch abstrakte Syntaxbäume (siehe Abschnitt 2.5.4) vorgenomen. Durch die Verwendung des Parsers zur Abstraktion wird eine 1:1-Abbildung eines syntaktischen Konstrukts auf einen abstrakten Syntaxbaum realisiert. Normalisierungen [Matthes 93, Kap. 6.3] sind nicht möglich, können aber weitgehend durch syntaktische Erweiterungen dargestellt werden; siehe dazu Kapitel 3.

# Kapitel 3

# Basissprache TLMin und Erweiterungssprache TLExt

Ein System zur Syntaxerweiterung umfaßt zwei Sprachen: die Basis- und die Erweiterungssprache [Solntseff, Yezerski 71]. Die im Rahmen dieser Arbeit entwickelte Erweiterungssprache TLExt und die benutzte Basissprache TLMin werden in diesem Kapitel informal beschrieben.

Die *Basissprache* TLMin stellt die grundlegenden Konzepte einer algorithmisch vollständigen Sprache in Form von abstrakten Syntaxbäumen mit einer festgelegten statischen und dynamischen Semantik zur Verfügung. Wie in Abschnitt 2.3.2 gefordert, unterstützt die Sprache die Konzepte Polymorphismus, Datenabstraktion und Orthogonalität [Atkinson, Bunemann 87; Matthes, Schmidt 91]. Die Beschreibung der abstrakten Syntaxbäume erfolgt in Form einer konkreten Syntax, die durch Syntaxerweiterungen geändert werden kann.

In der *Erweiterungssprache* TLExt werden Syntaxerweiterungen formuliert. Den Kern der Erweiterungssprache bilden Muster, die (neue) syntaktische Konstrukte beschreiben, und semantische Interpretationen, die eine Umsetzung der Konstrukte festlegen (siehe auch Kapitel 2.4). Dabei wird der in Abschnitt 2.5.4 beschriebene Ansatz der semantischen Interpretation durch abstrakte Syntaxbäume verfolgt. Die abstrakten Syntaxbäume werden in der um Platzhalter ergänzten und eventuell durch Syntaxerweiterungen geänderten Syntax der Basissprache beschrieben.

Als Abschluß dieses Kapitels wird diskutiert, wie die zur Übersetzung eines Quelltextes der Basis- oder der Erweiterungssprache zu verwendende Syntax ausgewählt werden kann.

## 3.1 Basissprache TLMin

Die Basissprache umfaßt die Konstrukte, auf die alle syntaktischen Erweiterungen abgebildet werden. Die konkrete Syntax dieser Sprache kann beliebig geändert werden, während die abstrakte Syntax einen festen Umfang besitzt, so daß die Typprüfung und Zwischenkode-Erzeugung durch eine Syntaxerweiterung nicht betroffen sind.

Die in diesem Kapitel beschriebene Basissprache TLMin ist eine Untermenge von TL [Matthes 93]. Es handelt sich um ein typisiertes Lambda-Kalkül höherer Ordnung mit imperativen Erweiterungen. Im Vergleich zu TL fehlen rekursive und parallele Bindungen, da diese Konstrukte für Syntaxerweiterungen keine Vorteile bieten, aber die Bindung der Konstrukte aus der Basissprache unnötig komplizieren. Außerdem fehlen Vereinfachungen und redundante Konstrukte, die sich durch Syntaxerweiterungen darstellen lassen, wie in Kapitel 4 gezeigt wird.

Die abstrakte und die grundlegende konkrete Syntax werden in Anhang A formal beschrieben. Die abstrakte Syntax enthält Konstrukte, die zur Definition der semantischen Interpretation von Syntaxerweiterungen (siehe Abschnitt 3.2.2) und zur korrekten Behandlung von Bindungsproblemen bei der Expansion von Syntaxerweiterungen (siehe Kapitel 5) benötigt werden.

Im folgenden werden die Konzepte von TLMin informal erläutert. Zuerst werden zwei einführende Beispiele vorgestellt, dann die in TLMin vertretenden Sorten und ihre Verbindungen beschrieben und im letzten Abschnitt die Bindungsregeln diskutiert. Eine ausführliche Beschreibung der Ausgangssprache TL findet sich in [Matthes 93].

### 3.1.1 Beispiele: Maximumberechnung und Paarbildung

Die beiden folgenden Beispiele zeigen die grundsätzlichen Eigenschaften von TLMin. Aus Gründen der Übersichtlichkeit wird in den Beispielen nicht die in Anhang A beschriebene minimale Syntax von TLMin benutzt, sondern eine erweiterte Syntax, die sich durch Anwendung der in Kapitel 4 beschriebenen Syntaxerweiterungen ergibt.

**Beispiel 3.1.1:** Funktionen zur Berechnung eines Maximums in TLMin

```
let maxInt1 = fun(x :Int  y :Int)
  if x > y then x else y end
let z = maxInt1(1 2)
let genMax = fun(:T <:Ok  grt :Fun(? :T  ? :T) :Bool)
  fun(x :T  y :T)
    if grt(x y) then x else y end
let maxInt2 = genMax(:Int >)
let z = maxInt2(1 2)
```

In Beispiel 3.1.1 werden zwei Funktionen definiert. Die Funktion *maxInt1* berechnet das Maximum zweier ganzer Zahlen unter Benutzung einer als vordefiniert angenommenen Funktion > mit dem Typ **Fun***(? :Int ? :Int) :Bool*, die den Wahrheitswert *true* liefert, wenn die erste Zahl größer als die zweite ist. Die Fragezeichen *?* bedeuten, daß ein Wert dieses Typs an dieser Stelle einen beliebigen Bezeichner enthalten kann, z. B. könnte der Wert von > folgendermaßen definiert werden: **let** > = **fun***(a :Int b :Int)* ... An Stelle der Fragezeichen aus der Typdefinition stehen hier die Bezeichner *a* und *b*.

Die zweite Funktion *genMax* ist eine generische Funktion, die für einen beliebigen Typ *T* und eine Funktion *grt* auf diesem Typ eine Funktion zur Bestimmung des Maximums von zwei Werten dieses Typs generiert. Die Funktion *genMax* wird zur Generierung einer Funktion *maxInt2* genutzt, die wie *maxInt1* das Maximum zweier ganzer Zahlen berechnet.

**Beispiel 3.1.2:** Typoperator für Paare mit Operationen

```
let pair = tuple
  Let T = Oper(:T1 <:Ok  :T2 <:Ok)
    Tuple t1 :T1  t2 :T2 end
  let new = fun(:T1 <:Ok  :T2 <:Ok  t1 :T1  t2 :T2)
    tuple let t1 = t1  let t2 = t2 end
  let first = fun(:T1 <:Ok  :T2 <:Ok  p :T(:T1 :T2))
    p.t1
  let swap = fun(:T1 <:Ok  :T2 <:Ok  p :T(:T1 :T2))
    tuple let t1 = p.t2  let t2 = p.t1 end
end
let intStrPair = pair.new(:Int :String 1 "Hallo")
pair.first(:Int :String intStrPair) (* yields 1 *)
let strIntPair = pair.swap(:Int :String intStrPair)
pair.first(:String :Int intStrPair) (* yields "Hallo" *)
```

In Beispiel 3.1.2 wird in einem Tupelwert *pair* ein Typoperator *T* definiert, der zwei Typparameter *T1,T2* auf ein Tupel mit zwei Feldern abbildet. Dazu werden die Funktionen *new* zur Erzeugung eines Paarwertes, *first* zum Zugriff auf das erste Feld eines Paares und *swap* zum Tausch der Felder eines Paares definiert. Die Aufrufe der Funktion *pair.first* zeigen die Extraktion von Werten aus Paaren, wobei die extrahierten Werte als Kommentar angegeben sind.

### 3.1.2 Sorten: Bezeichner, Werte, Typen, Bindungen und Signaturen

Die Objekte der Basissprache werden zu Sorten gruppiert: Bezeichner, Werte, Typen, Bindungen und Signaturen. Jedes Objekt besitzt eine Sorte, analog zu den Typen von Werten:

- Ein Bezeichner (*identifier*) dient zur Bezeichnung von Objekten.

- Ein Wert (*value*) ist ein Objekt, das zur Laufzeit des Programms existiert bzw. existieren wird.
- Ein Typ (*type*) beschreibt einen Wert und dient zur Prüfung der statischen Korrektheit eines Programms.
- Eine Bindung (*binding*) definiert einen Namen für einen Wert oder einen Typ, mit dem dieser Wert oder Typ referenziert werden kann; zu den Bindungsregeln siehe auch den nächsten Abschnitt.
- Eine Signatur (*signature*) beschreibt eine Bindung, d. h. sie ordnet einem Namen im Falle einer Wertbindung den Typ (oder einen Supertyp) des Wertes und im Falle einer Typbindung den Supertyp des Typs zu.

Die folgende Tabelle zeigt exemplarisch die wechselseitigen Zusammenhänge der Sorten.

| Sorte | Beispiele | Kommentar |
|---|---|---|
| **IDE** | *valueIdentifier*, *TypeIdentifier* | Bezeichner |
| **VAL** | **ok**, *true*, 1, IDE.IDE<br>**array** BNDS **end**<br>**tuple** BNDS **end**<br>**fun**(SIGS) VAL | Werte |
| **TYP** | **Ok**, *Bool*, *Int*, *String*<br>*Array*(BNDS)<br>**Tuple** SIGS **end**<br>**Fun**(SIGS) :TYP | Typen |
| **BNDS** | **let** IDE = VAL<br>**Let** IDE = TYP | Bindungen |
| **SIGS** | IDE :TYP<br>:IDE <:TYP | Signaturen |

Werte werden durch Typen beschrieben:

| Wert | Typ | Kommentar |
|---|---|---|
| **ok** | *:***Ok** | Wert des kanonischen Supertyps **Ok** |
| *true* | *:Bool* | ein Wahrheitswert |
| 1 | *:Int* | eine ganze Zahl |
| **array** 1 **end** | *:Array(:Int)* | ein Feld von ganzen Zahlen |
| **tuple let** *x* = *1* **end** | *:***Tuple** *x :Int* **end** | ein Tupel mit einem ganzzahligen Feld |
| *x* | *:?* | ein Wert, dessen Typ von der Definition von *x* abhängt |

Typen können in einer Subtypbeziehung stehen, wobei diese Beziehung bei strukturierten Typen durch strukturelle Äquivalenz hergestellt wird:

| Subtyp | Supertyp | Kommentar |
|---|---|---|
| *T* | <:**Ok** | jeder Typ ist Subtyp des Supertyps **Ok** |
| *List(:T)* | <:*List(:***Ok***)* | Subtypen bei Typoperatorapplikationen |
| **Tuple** *x :T y :T* **end** | <:**Tuple** *? :T* **end** | ein spezieller Typ ist Subtyp eines allgemeineren Typs |

Eine Signatur beschreibt eine Bindung:

| Bindung | Signatur |
|---|---|
| **let** *x = 1* | *x :Int* |
| **Let** *T = Int* | *T* <:**Ok** |

Bindungen und Signaturen führen definierende Bezeichner ein, die referenzierende Bezeichner binden (zu den Bindungsregeln siehe auch den folgenden Abschnitt):

| Konstrukt | Kommentar |
|---|---|
| **let** *x = 1* | der Wert *x* wird definiert |
| **let** *y = x* | der Wert *x* wird referenziert |
| **fun***(:T* <:**Ok***)* | der Typ *T* wird definiert |
| **Let** *A = List(:T)* | der Typ *T* wird referenziert |

Bezeichner können an selektierender Position auftreten, wobei ihre Bindung von dem Typ des Präfixes abhängt:

| Konstrukt | Kommentar |
|---|---|
| **let** *t =* **tuple let** *x = 1* **end** | ein Tupel mit einer Komponente *x* wird definiert |
| *t.x* | die Tupelkomponente *x* wird selektiert |

### 3.1.3 Bezeichner: Benennung und Bindung

Ein Bezeichner ist der Name eines Objekts einer bestimmten Sorte, wobei im Fall von TLMin Namen nur für Werte und Typen eingeführt werden können.

Bezeichner können in drei Positionen auftreten:

- Ein in **definierender** Position auftretender Bezeichner definiert einen Namen für ein Objekt. Durch die folgenden Definitionen werden die Namen *valId* für einen Wert und *typId* für einen Typ vereinbart:

  **let** *valId* = ...
  **Let** *typId* = ...

- Ein in **referenzierender** Position auftretender Bezeichner referenziert ein Objekt, dessen Name durch die Definition eines Bezeichners definiert wird. Jedes referenzierende Auftreten eines Bezeichners wird gemäß der Bindungsregeln, die im Anschluß

erläutert werden, an ein definierendes Auftreten gebunden. Die folgenden Referenzen referenzieren einen Wert (*valId*) und einen Typ (*typId*):

**let** ... = *valId*
**Let** ... = *typId*

- Ein in **selektierender** Position auftretender Bezeichner selektiert ein Teil eines Objekts. Im Gegensatz zum referenzierenden Auftreten eines Bezeichners hängt die Bindung des selektierenden Bezeichners von dem Objekt ab, dessen Teil es bezeichnet. Erst wenn das Objekt (bzw. dessen Struktur oder Typ) bekannt ist, kann die Bindung vollzogen werden. Im folgenden Beispiel wird ein Objekt mit einem Teilobjekt definiert. Durch den Zugriff über den referenzierenden Bezeichner *objId* und den selektierenden Bezeichner *subObjId* wird das Teilobjekt selektiert:

  **let** *objId* = **tuple let** *subObjId* = ... **end**
  *objId.subObjId*

Jedes referenzierende Auftreten eines Bezeichners wird gemäß der Bindungsregeln (*binding rules*) bzw. Sichtbarkeitsregeln (*scoping rules*) an ein definierendes Auftreten dieses Bezeichners gebunden. Die Bindungsregeln legen fest, welches definierende Auftreten eines Bezeichners ein referenzierendes Auftreten bindet.

In der vorliegenden Arbeit werden folgende Bindungs- bzw. Sichtbarkeitsregeln verwendet:

1. Jede Bindung eines referenzierenden Bezeichners an einen definierenden Bezeichner erfolgt statisch, d. h. zum Zeitpunkt der Übersetzung des Quelltextes und nicht zur Laufzeit des Zielkodes, im Unterschied zum dynamischen Binden von Bezeichnern in LISP.

2. Der Sichtbarkeitsbereich (*scope*) eines definierenden Bezeichners *ide* erstreckt textuell vom Auftreten seiner Definition im Quelltext (**let** ide = ...) bis zum Ende des umgebenden Blocks oder einer erneuten Definition des gleichen Bezeichners.

3. In seinem Sichtbarkeitsbereich bindet das definierende Auftreten des Bezeichners jedes referenzierende Auftreten des Bezeichners.

Die Definition eines Bezeichners kann vorübergehend verdeckt sein, wenn der Bezeichner in einem eingeschachtelten Block redefiniert wird:

```
let x = ... (* erste Definition von x *)
begin
  let x = ... (* die erste Definition x wird verdeckt *)
end (* die erste Definition von x wird wieder sichtbar *)
```

Der Bereich, in dem die erste Definition von $x$ nicht sichtbar ist, wird als Loch (*hole*) im Sichtbarkeitsbereich von $x$ bezeichnet.

Alternative Bindungs- und Sichtbarkeitsregeln erlauben z. B. ein Überladen (*overloading*) von Bezeichnern. Ein Bezeichner ist überladen, wenn zu einem Zeitpunkt mindestens zwei verschiedene Definitionen dieses Bezeichners sichtbar sind:

**let** $x = 1$
**let** $x =$ *"hallo"*

Nach den in dieser Arbeit verwendeten Regeln ist nur die zweite Definition von $x$ sichtbar. Beim Überladen sind beide Definitionen sichtbar und eine Referenz auf $x$ wird abhängig vom Typkontext, in dem sich die Referenz befindet, an eine der beiden Definitionen gebunden.

Eine Änderung der Bindungsregeln hat Auswirkungen auf den Mechanismus der Syntaxerweiterung, denn während der Definition einer Syntaxerweiterung werden bestimmte Bindungen vorgenommen. Siehe dazu die Diskussion der Bindung von Syntaxerweiterungen in Kapitel 5.

## 3.2 Erweiterungssprache TLExt

Die Erweiterungssprache beschreibt Erweiterungen bzw. Änderungen der Syntax der Basissprache. Wie in Kapitel 2.4 beschrieben, bestehen Syntaxerweiterungen aus syntaktischen Konstrukten und semantischen Interpretationen. Die syntaktischen Konstrukte werden durch kontextfreie Grammatiken (siehe Abschnitt 2.2.2) beschrieben. Als semantische Interpretationen werden abstrakte Syntaxbäume gemäß Abschnitt 2.5.4 verwendet. Zur Beschreibung der abstrakten Syntaxbäume dienen nicht Konstruktoren, sondern die um Platzhalter erweiterte Basissprache. Nach einer (erfolgreichen) Syntaxerweiterung kann die erweiterte Sprache zur Beschreibung der abstrakten Syntaxbäume genutzt werden. Wenn verschiedene Versionen der erweiterten Sprache existieren, sollte eine Auswahl zwischen den Versionen erlaubt sein.

Grundsätzlich kann auch die Syntax der Erweiterungssprache geändert werden (Reflektion, siehe Abschnitt 2.3.1). Aus Gründen der Übersichtlichkeit wird in dieser Arbeit davon abgesehen. Die in der Arbeit getroffenen Aussagen und Lösungen, z. B. zum Bindungsproblem, sind auf den reflektiven Fall übertragbar.

Nach einem einführenden Beispiel werden in den folgenden Abschnitten die einzelnen Sprachelemente der Erweiterungssprache TLExt vorgestellt, die Bindung und Sortenprüfung von Syntaxerweiterungen beschrieben und die Ausführung einer Syntaxerweiterung durch einen Parser-Generator diskutiert.

### 3.2.1 Beispiel: Iterationskonstrukt

Als einführendes Beispiel wird die Erweiterung der Syntax von Werten um ein Konstrukt zur Iteration über eine Tabelle beschrieben.

**Beispiel 3.2.1:** Erweiterung der Syntax um ein Iterationskonstrukt

```
grammar forEachSyntax
import iter
export
LET newValG ::=
  "for" "each" id = ideG "in" table = valG ":" pred = valG "do" act = bndsG "end"
  => VAL
    iter.forEach(VAL table
      fun(IDE id :?) VAL pred
      fun(IDE id :?) begin BNDS act end)
  END
ASSIGN valG ::= COPY valG | COPY newValG
end
```

Die in Beispiel 3.2.1 gezeigte Syntaxerweiterung *forEachSyntax* erweitert die syntaktischen Konstrukte der Sorte **VAL** um ein **for each**-Konstrukt. Die Produktion *newValG* besteht aus einer Alternative: Ein **for each**-Konstrukt wird in einen Aufruf einer Funktion *forEach* aus der (importierten) Bibliothek *iter* expandiert. Durch die Zuweisung wird die Produktion *valG* wird um die Alternativen der Produktion *newValG* erweitert.

Das folgende Beispiel zeigt ein durch diese Produktion beschriebenes syntaktisches Konstrukt und seine Expansion:

**for each** *p* **in** *persons : p.age > age* **do** *print.string(p.name) print.ln()* **end**
wird umgewandelt zu:

```
iter.forEach(persons
  fun(p :?) p.age > age
  fun(p :?) begin  print.string(p.name) print.ln() end)
```

Die in der Expansion stehenden Fragezeichen *?* zeigen eine fehlende Typangabe an. Ein Compiler, der Typinferenz [Damas, Milner 82] unterstützt, kann aus der Definition der Funktion *forEach* herleiten, welche Typen an Stelle der Fragezeichen einzusetzen sind. Der in dieser Arbeit verwendete TL-Compiler ist nicht so mächtig. Da Typinferenz kein grundsätzliches Problem darstellt, wird nicht näher darauf eingegangen, sondern die vereinfachende Schreibweise mit Fragezeichen als Platzhalter für fehlende Typen beibehalten. Es bleibt festzuhalten, daß der Compiler der Basissprache über Typinferenz verfügen sollte.

### 3.2.2 Sprachelemente

Die abstrakte und die konkrete Syntax von TLExt sind in Anhang A formal beschrieben. Im folgenden findet sich eine informale Erläuterung der Sprachelemente von TLExt.

Syntaxerweiterungen basieren auf links-attributierten Grammatiken, wie sie in Abschnitt 2.2.4.1 beschrieben sind. Die Produktionen verfügen also neben der Information über das syntaktische Konstrukts, das sie beschreiben, über ererbte und abgeleitete Attribute, wobei die Werte der abgeleiteten Attribute durch die semantische Interpretation festgelegt werden.

#### 3.2.2.1 Syntaxerweiterungsmodule

Die Beschreibung einer neuen Syntax findet in einem Syntaxerweiterungsmodul statt:

**grammar** *newSyntax*
**import** ...
**export**
*(* Deklarationen *)*
**end**

Es wird eine neue Syntax *newSyntax* definiert. Durch eine **import**-Klausel können Bezeichner von Bibliotheksmodulen in den Sichtbarkeitsbereich der Syntaxerweiterung importiert werden. Der Rumpf des Moduls besteht aus beliebig vielen Deklarationen.

#### 3.2.2.2 Deklarationen

Der Modulrumpf einer Syntaxerweiterung kann zwei Arten von Deklarationen enthalten:

- Einführung neuer Produktionen:

  **LET** *newG ::= (* Produktion *)*

- Änderung vorhandener Produktionen:

  **ASSIGN** *oldG ::= (* Produktion *)*

Die erste Deklaration definiert eine neue Produktion mit dem Nonterminal *newG*. Die zweite Deklaration redefiniert eine vorhandene Produktion mit dem Nonterminal *oldG*.

#### 3.2.2.3 Ererbte Attribute

Wenn eine Produktion über ererbte Attribute verfügt, müssen diese in der Deklaration mit den Sorten dieser Attribute aufgeführt werden:

**LET** *newG(val* ::**VAL** *bnds* ::**BNDS***) ::= (* Produktion *)*
**ASSIGN** *oldG(val* ::**VAL** *bnds* ::**BNDS***) ::= (* Produktion *)*

Die beiden Deklarationen verfügen über zwei ererbte Attribute. Das Attribut *val* besitzt die Sorte **VAL**, das Attribut *bnds* die Sorte **BNDS**. Die ererbten Attribute können wie Platzhalter (siehe weiter unten) als aktuelle Werte von ererbten Attributen beim Aufruf einer Produktion und in der semantischen Interpretation verwendet werden.

#### 3.2.2.4 Produktionen

Jede Produktion besteht aus mindestens einer Alternative:

**LET** *g ::= (* Alternative 1 *)* | *(* Alternative 2 *)* | ...

Jede Alternative beinhaltet entweder das Nonterminal einer Produktion, was einer Kopie dieser Produktion entspricht, oder ein Muster für ein syntaktisches Konstrukt und eine semantische Interpretation, welche die abgeleiteten Attribute der Produktion beschreibt:

**LET** *g1* ::= **COPY** *g2*
| *(* Muster *) => (* Interpretation *)*

#### 3.2.2.5 Muster

Das Muster (*pattern*) ist eine (evtl. leere) Folge von Schlüsselworten und Definitionen von Platzhaltern, die sich auf Produktionen beziehen:

**LET** *g* ::= "keyword" *p = nont* ...

Die durch das Nonterminal *nont* referenzierte Produktion beschreibt das an Stelle des Platzhalters zu parsende syntaktische Konstrukt. Der Platzhalter *p* erhält den aktuellen Wert des abgeleiteten Attributs der Produktion, der durch die semantische Interpretation der Produktion festgelegt wird. Falls die Produktion *nont* über mehrere abgeleitete Attribute verfügt, muß für jedes Attribut ein Platzhalter festgelegt werden:

**LET** *g ::= p1, p2 = nont* ...

Falls die Produktion *nont* auch ererbte Attribute erwartet, muß das Nonterminal mit aktuellen Werten für diese Attribute versehen werden:

**LET** *g(p1 ...) ::= p2 = ... p3 = nont(p1 p2)* ...

Die ererbten Attribute von *nont* erhalten die aktuellen Werte der Platzhalter *p1* und *p2*, wobei *p1* ein ererbtes Attribut der Produktion ist, während *p2* einen im Muster definierten Platzhalter, also ein abgeleitetes Attribut einer anderen Produktion, darstellt.

#### 3.2.2.6 Semantische Interpretation

Die semantische Interpretation ist die Berechnungsvorschrift für die abgeleiteten Attribute einer Produktion. Für jedes Attribut wird festgelegt, zu welcher Sorte der Basissprache es gehört, z. B. **TYP** für einen Typ. Der Wert des Attributs wird durch ein syntaktisches Konstrukt der entsprechenden Sorte beschrieben, z. B. ist **TYP** *List(:***TYP** *typ)* ein syntaktisches Konstrukt der Sorte **TYP**. Innerhalb des syntaktischen Konstrukts auftretende Platzhalter werden durch ihre Sorte gekennzeichnet (im Beispiel **TYP** *typ*). Der aktuelle Wert des Attributs während des Parsens ergibt sich, wenn die Platzhalter durch ihre aktuellen Werte ersetzt werden. Wenn z. B. der Platzhalter *typ* den Wert *Int* besitzt, erhält das oben angegebene abgeleitete Attribut den Wert *List(:Int)*.

Zur Erläuterung dient das oben angeführte Beispiel:

**LET** *forEachG ::=*
"for" "each" *id* = *ideG* "in" *table* = *valG* ":" *pred* = *valG* "do" *act* = *bndsG* "end"
=> **VAL**
  *iter.forEach(***VAL** *table*
    **fun***(***IDE** *id :?)* **VAL** *pred*
    **fun***(***IDE** *id :?)* **begin BNDS** *act* **end***)*
**END**

Die Platzhalter *id*, *table*, *pred* und *act* bezeichnen abgeleitete Attribute, deren syntaktische Form durch die Muster der Produktionen *ideG*, *valG* und *bndsG* beschrieben werden, und deren Werte sich durch die die semantischen Interpretationen dieser Produktionen berechnen.

In der semantischen Interpretation dieser Produktion wird durch das erste **VAL** die Sorte des (einzigen) abgeleiteten Attributs angegeben, was gleichzeitig bedeutet, daß ein syntaktisches Konstrukt gemäß der Produktion *valG* folgt. Innerhalb dieses Konstrukts sind die Platzhalter durch **VAL**, **IDE** und **BNDS** gekennzeichnet, um sie von Bezeichnern der Basissprache zu unterscheiden.

Um semantische Interpretationen mit Platzhaltern korrekt parsen zu können, müssen die konkrete und abstrakte Syntax der Basissprache um Platzhalter erweitert werden (siehe dazu Anhang A).

### 3.2.3 Bindung und Sortenprüfung

In diesem Abschnitt wird die statische Semantik der Erweiterungssprache informal beschrieben. Die formalen Regeln zur Prüfung der statischen Semantik von TLExt sind in Anhang B.2 angegeben.

#### 3.2.3.1 Vordefinierte Produktionen

Es wird davon ausgegangen, daß eine Grammatik für die Basissprache definiert ist, die folgende Produktionen zur Beschreibung der syntaktischen Konstrukte der verschiedenen Sorten der Basissprache enthält. Jede Produktion besitzt ein abgeleitetes Attribut der entsprechenden Sorte und keine ererbten Attribute:

| Nonterminal | Sorte | Bedeutung |
|---|---|---|
| *valG* | **VAL** | Wert |
| *typG* | **TYP** | Typ |
| *bndsG* | **BNDS** | Bindungen |
| *sigsG* | **SIGS** | Signaturen |

In der durch diese Produktionen gebildeten konkreten Syntax der Basissprache werden die semantischen Interpretationen beschrieben. Durch Redefinition dieser Produktionen kann die Syntax der Basissprache geändert werden. Außerdem stehen noch weitere Produktionen zur Verfügung, die bestimmte Objekte einer Sorte als abgeleitetes Attribut liefern. Sie können in Mustern benutzt, aber nicht verändert werden:

| Nonterminal | Sorte | Bedeutung |
|---|---|---|
| *ideG* | **IDE** | Bezeichner |
| *identifierG* | **IDE** | alphanumerischer Bezeichner |
| *infixG* | **IDE** | Infix-Bezeichner |
| *intG* | **VAL** | ganze Zahl |
| *realG* | **VAL** | Dezimalzahl |
| *longrealG* | **VAL** | lange Dezimalzahl |
| *charG* | **VAL** | einzelnes Zeichen |
| *stringG* | **VAL** | Zeichenkette |

#### 3.2.3.2 Bindung von Nonterminalen und Platzhaltern

Die in einem Muster auftretenden Referenzen von Nonterminalen werden statisch gebunden. Dabei gilt als Sichtbarkeitsregel, daß neben den vordefinierten Produktionen alle im Syntaxerweiterungsmodul definierten Produktionen sichtbar sind, so daß auch wechselseitig rekursive Produktionen definiert werden können:

**LET** *g1 ::= ... p2 = g2 ... => ...*
**LET** *g2 ::= ... p1 = g1 ... => ...*

Die in einer semantischen Interpretation auftretenden Platzhalter-Referenzen werden durch die Platzhalter-Definitionen im Muster der Produktion gebunden:

**LET** *g ::= ... p = valG ... => ...* **VAL** *p ...*
**LET** *gf ::= ... p = valG ... => ...* **VAL** *pf ... (* error *)*

In der ersten Produktion wird der in der semantischen Interpretation auftretende Platzhalter *p* von der im Muster erfolgten Definition gebunden. In der zweiten Produktion fehlt eine Definition von *pf*.

Die als aktuelle Werte von ererbten Attributen eines Produktionsaufrufs auftretenden Platzhalter müssen als ererbte Attribute der Produktion oder als abgeleitete Attribute eines links im Muster auftretenden Produktionsaufrufs definiert worden sein:

**LET** *g(p0 ::***VAL***) ::= p1 = g1 p2 = g2(p0 p1) ...*
**LET** *gf ::= p = g(p) ... (* error *)*
**LET** *gf ::= p = g(pf) pf = ... (* error *)*

Im ersten Beispiel werden beim Aufruf der Produktion *g2* das ererbte Attribut *p0* der Produktion *g* und der durch das abgeleitete Attribut der Produktion *g1* definierte Platzhalter *p1* als aktuelle Werte der ererbten Attribute weitergegeben. Wie das zweite Beispiel zeigt, ist es nicht möglich, einer Produktion *g* den Wert eines ihrer abgeleiteten Attribute *p* zu übergeben. Ein weiter rechts im Muster definierter Platzhalter *pf* im dritten Beispiel ist zum Zeitpunkt des Aufrufs von *g* noch nicht definiert.

#### 3.2.3.3 Sorten von Produktionen

Jede Produktion erhält die Sorten ihrer abgeleiteten Attribute, die sich aus der semantischen Interpretation ergeben. Gegenwärtig ist ein einfacher Algorithmus zur Sorteninferenz implementiert, der die Sorten allein aus der semantischen Interpretation ohne Kenntnis der Sorten anderer Produktionen ableitet. Bei der Definition einer Produktion müssen sich die Sorten eindeutig aus mindestens einer Alternative ableiten lassen:

**LET** *g1 ::= ... | ... =>* **VAL** *...* **END**
**LET** *gf ::=* **COPY** *g1 (* error *)*

Die erste Produktion erhält die Sorte **VAL**, die sich aus einer Alternative ableitet. Bei der zweiten Produktion existiert keine Alternative, aus der sich die Sorten ohne Kenntnis der Sorten einer anderen Produktion ableiten ließen.

Ein weiter entwickelter Algorithmus kann die inferierten oder vordefinierten Sorten anderer Produktionen mit einbeziehen. Problematisch bleiben rekursive Produktionen, für die sich keine (monomorphe) Sorte ableiten läßt, z. B.:

**LET** *g1 ::=* **COPY** *g2*
**LET** *g2 ::=* **COPY** *g1 (* error *)*

Alle Alternativen einer Produktion müssen dieselben Sorten besitzen:

**LET** *g ::=* **COPY** *valG | ... =>* **VAL** *...* **END**
**LET** *gf ::=* **COPY** *valG | ... =>* **TYP** *...* **END** *(* error *)*

Die beiden Alternativen der ersten Produktion besitzen die Sorte **VAL**, während in der zweiten Produktion eine Alternative die Sorte **VAL** und die andere die Sorte **TYP** besitzt.

#### 3.2.3.4 Sorten von Platzhaltern

Jedem Platzhalter wird eine Sorte zugewiesen, die sich aus den Sorten der abgeleiteten Attribute der Produktion ergibt:

**LET** *g ::= ... p = valG ... => ...*

Der Platzhalter *p* erhält die Sorte **VAL**.

In der semantischen Interpretation wird die Sortentreue überprüft, d. h. ein Platzhalter einer Sorte darf nur an einer Stelle in der semantischen Interpretation auftreten, an der auch ein syntaktisches Konstrukt dieser Sorte stehen könnte:

**LET** *g ::= ... id = ideG ... => ...* **fun**(**IDE** *id :?*) ...
**LET** *gf ::= ... v = valG ... => ...* **fun**(**VAL** *v :?*) ... *(* error *)*
**LET** *gf ::= ... v = valG ... => ...* **fun**(**IDE** *v :?*) ... *(* error *)*

Die erste Produktion ist korrekt, weil der Platzhalter der Sorte **IDE** an Stelle eines Bezeichners eingesetzt wird. In der zweiten und dritten Produktion wird versucht, an dieser Stelle einen Platzhalter Sorte **VAL** zu verwenden, was nicht möglich ist.

Die Kennzeichnung der Platzhalter-Referenzen in der semantischen Interpretation ist notwendig, um sie von Bezeichnern der Basissprache zu unterscheiden. Die Angabe der Sorte (**IDE**, **VAL** usw.) ist redundant, da die Sorte aus der Definition abgeleitet werden kann. Unklarheiten können entstehen, wenn an einer Stelle zwei Sorten erlaubt sind:

**LET** *g ::= ... p = ... => ...* **let** *x* = **PH** *p*

Der Platzhalter *p* könnte die Sorten **VAL** oder **IDE** (als Bezeichner eines Wertes) besitzen. Somit kann in der Abstraktionsphase für dieses Konstrukt nicht ohne die Sorteninformation des Platzhalters festgelegt werden, welcher Knoten des abstrakten Syntaxbaums zu erzeugen ist. Dies ließe sich während der Typprüfung korrigieren oder durch geschickte Konstruktion des Syntaxbaums umgehen.

#### 3.2.3.5 Ererbte und abgeleitete Attribute

Die ererbten Attribute erhalten ihre aktuellen Werte beim Aufruf der Produktion. Während der Übersetzung der Syntaxerweiterung wird geprüft, ob die Anzahl der definierten Platzhalter beim Aufruf einer Produktion der Anzahl der abgeleiteten Attribute entspricht. Die Platzhalter erhalten die Sorten der abgeleiteten Attribute. Es wird geprüft, ob die Anzahl und Sorten der aktuellen Werte für ererbte Attribute beim Aufruf einer Produktion mit der Definition der Produktion übereinstimmen:

**LET** *g1 ::= ... =>* **TYP** ... **END BNDS** ... **END**
**LET** *g2(p1 ::***VAL** *p2 ::***TYP** *p3 ::***BNDS**) *::= ... =>* **SIGS END**
**LET** *g3(p4 ::***VAL**) *::= p5, p6 = g1 p7 = g2(p4 p5 p6) ...*

Die Platzhalter *p5* und *p6* in der Produktion *g3* erhalten die Sorten **TYP** und **BNDS** der abgeleiteten Attribute von *g1*. Die Sorten der ererbten Attribute von *g2* (**VAL**, **TYP** und **BNDS**) werden mit den Sorten des ererbten Attributs *p4* und der Platzhalter *p5* und *p6* in der Produktion *g3* verglichen.

#### 3.2.3.6 Typkonformität der semantischen Interpretation

Die semantische Interpretation kann nicht auf Typkonformität geprüft werden, da die Sorten der Platzhalter nicht genügend Informationen enthalten:

**LET** *g ::= ... v = valG ... => ...* **VAL** *v + 1 ...* **VAL** *v [1] ... (* ?error? *)*

Die Expansion dieser Syntaxerweiterung wird mit Sicherheit einen Typfehler erzeugen, denn der aktuelle Wert des Platzhalters *v* kann nicht gleichzeitig eine ganze Zahl und ein Feld sein. Da die Sorte **VAL** über den Typ des Wertes nichts aussagt, ist die semantische Interpretation trotzdem sortenkonform und damit korrekt.

Es ist zu prüfen, inwieweit das Sortensystem parallel zum Typsystem erweitert werden kann, um solche Informationen zu enthalten:

**LET** *g ::= ... v :Int = valG ... => ...* **VAL** *v + 1 ...* **VAL** *v [1] ... (* error *)*

Hier wird explizit zugesichert, daß der Platzhalter *v* einen Wert des Typs *Int* beschreibt. Diese Zusicherung muß nun bei der Expansion geprüft werden, in Analogie zu der Prüfung von Parametern bei Funktionsaufrufen. Dies erfordert eine Erweiterung der Typprüfung der Basissprache.

Einige Fehler lassen sich auch durch ein erweitertes Sortensystem nicht verhindern:

**LET** *g* ::= "do" *ide1 = ideG ide2 = ideG*
**=> BNDS**
  **let IDE** *ide1* = **array** *1 2 3* **end**
  **let IDE** *ide2 = 1*
  **IDE** *ide1* [**IDE** *ide2*] := *5*
**END**

In dieser Syntaxerweiterung werden zwei Bezeichner in einer Folge von Bindungen benutzt. Die Bindung dieser Bezeichner erfolgt erst nach der Expansion der Syntaxerweiterung, was zu einer korrekten und zu einer falschen Expansion führen kann:

| **do** *x y* | **do** *x x* |
|---|---|
| wird umgewandelt zu: | wird umgewandelt zu: |
| **let** *x* = **array** *1 2 3* **end** | **let** *x* = **array** *1 2 3* **end** |
| **let** *y = 1* | **let** *x = 1* |
| *x[y] := 5* | *x[x] := 5 (* error *)* |

Während die Expansion von **do** *x y* korrekt ist (dem ersten Element des Feldes *x* wird die Zahl 5 zugewiesen), liegt bei der Expansion von **do** *x x* ein Typfehler vor, denn die

zweite Definition von x verdeckt die erste, so daß ein Feldzugriff auf x unmöglich ist. Diese Fehlersituation kann erst bei der Typprüfung des expandierten abstrakten Syntaxbaums erkannt werden.

Da also eine statische Typprüfung des *expandierten* Syntaxbaums auf jeden Fall notwendig ist, wird in dieser Arbeit darauf verzichtet, eine (unvollständige) Typprüfung während der Definition der Syntaxerweiterung durchzuführen.

Durch diese Entscheidung werden weitere Freiheitsgrade gewonnen, wie die folgende alternative Definition des **for each**-Konstrukts mit einer objektorientierten Implementation zeigt:

```
LET g ::= "for" "each" id = ideG "in" b = valG "do" act = bndsG "end"
 => VAL
  begin
    let b = VAL b
    b.init()
    loop
      if b.empty() then exit end
      let IDE id = b.first()
      BNDS act
      b.next()
    end
  end
END
```

Es wird davon ausgegangen, daß *b* ein Tupel (Objekt) mit *ungefähr* folgendem Typ ist:

```
 Let Bulk(:E <:Ok) <:Ok =
 Tuple
   init() :Ok              (* initialize for iteration *)
   empty() :Bool           (* true if no more elements *)
   first() :E              (* get first element if not(empty()) *)
   next() :Ok              (* next element *)
 end
```

Der neue Freiheitsgrad liegt darin, daß *b* nur *ungefähr* den Typ *Bulk* besitzen muß, d. h. es können noch beliebige andere Attribute oder auch eine andere Reihenfolge der Attribute vorliegen, ohne daß die Syntaxerweiterung geändert oder neu übersetzt werden müßte. Allerdings wird dieser Freiheitsgrad damit erkauft, daß eine Expansion dieser Syntaxerweiterung in der Typprüfung scheitern kann, wenn *b* diese Anforderungen nicht erfüllt.

#### 3.2.3.7 Fehler in expandierten Syntaxerweiterungen

Wie im vorigen Abschnitt gesehen, können Fehler in Syntaxerweiterungen auftreten, die erst nach der Expansion der Syntaxerweiterung von der Typprüfung der Basissprache erkannt werden. Dies führt zu zwei Problemen:

- Der Fehler wird in der expandierten Version des abstrakten Syntaxbaums gefunden. Wenn der expandierte abstrakte Syntaxbaum angezeigt wird (*unparsing*), muß der Anwendungsprogrammierer die erfolgte Expansion zurückverfolgen. Stattdessen sollte das System eine Rückverfolgung des Fehlers vornehmen, damit die Fehlermeldung für den Anwendungsprogrammierer sich auf seinen unexpandierten Quelltext beziehen kann.

- Eine Rückverfolgung des Fehlers ist schwer möglich, wenn der Fehler implizit durch die Definition der Syntaxerweiterung eingeführt wird (siehe obiges Beispiel mit den beiden Bezeichnern). Erst durch den Vergleich des Quelltextes mit der Definition der Syntaxerweiterung ergibt sich die konkrete Fehlersituation. In diesem Fall muß der Anwendungsprogrammierer auch auf die Definition der Syntaxerweiterung hingewiesen werden, die er an der fehlerhaften Stelle seines Quelltextes benutzt.

An dieser Stelle sind weitere Arbeiten in den Bereichen Typprüfung der Basissprache und Rückverfolgung von Fehlern nötig. Fehler sollten weitestgehend vermieden bzw. so früh wie möglich erkannt werden. Im Fehlerfall müssen aussagekräftige Fehlermeldungen generiert werden, damit der Anwendungsprogrammierer bei der Fehlersuche und -behebung unterstützt wird.

### 3.2.4 Ausführen einer Syntaxerweiterung

Die geänderte Grammatik mit den zu semantischen Aktionen umgewandelten semantischen Interpretationen wird einem Parser-Generator übergeben, um den Parser und die Abstraktionsphase des neuen Frontends zu erzeugen. Dabei können je nach Leistungsfähigkeit des Parser-Generators verschiedene Fehler gemeldet werden, wie in Abschnitt 2.2.2 erläutert.

Der in dieser Arbeit verwendete LL(1)-Parser-Generator erkennt Linksrekursionen und Mehrdeutigkeiten, die als Verletzung der LL(1)-Eigenschaft gelten. Ein LR- oder LALR-Parser-Generator würde dagegen eine größere Menge von Grammatiken akzeptieren, lag aber zum Zeitpunkt dieser Arbeit nicht vor.

Je nach Art des Parser-Generators könnte die Syntax und Semantik der Erweiterungssprache ergänzt werden, um die besonderen Eigenschaften des Parser-Generators auszunutzen. Beispielsweise könnten bei einem LL-Parser-Generator Beschreibungen für linksassoziative Konstrukte verwendet werden [Cardelli 91; Schröder, Matthes 92]. Ein LALR-Parser-Generator könnte über die Angabe von Präzedenzen die Assoziativität herstellen und Mehrdeutigkeiten auflösen [Aho et al. 88]. Es ist nicht Aufgabe dieser Arbeit, für jeden Parser-Generator eine passende Erweiterungssprache zu definieren.

## 3.3 Auswahl einer Sprachversion zur Übersetzung

Um einen Quelltext der (erweiterten) Basissprache oder der Erweiterungssprache zu übersetzen, muß die zu verwendende Syntax ausgewählt werden. Dazu gibt es zwei Möglichkeiten:

**extern:** Die zu benutzende Version wird z. B. als Parameter beim Aufruf des Compilers angegeben.

**intern:** Der Quelltext enthält als erste Anweisung die zu benutzende Version, z. B. in der Form **with grammar** *newGram* **do** ...

Bei der externen Version kann die Information über die verwendete Syntax verlorengehen. Die interne Version dagegen lokalisiert diese Information in dem Quelltext, so daß sie mit dem Quelltext verbunden bleibt. Allerdings muß in der internen Version beim Übersetzen des Quelltextes das streng sequentielle Compilermodell aufgegeben werden, denn erst nach der Übersetzung der einleitenden Anweisung **with grammar** ... ist die zum Übersetzen des restlichen Quelltextes verwendete Syntax bekannt. Außerdem kann die Syntax der Anweisung **with grammar** ... nicht geändert werden.

Um das sequentielle Compilermodell beibehalten und die vollständige Redefinition der Syntax ermöglichen zu können, sollte die externe Version favorisiert werden. Die Verbindung eines Quelltextes mit der in der Syntaxerweiterung verwendeten Syntax sollte extern durch die Ergänzung des Objektes „Quelltext“ um ein Attribut „Syntax“ erfolgen, wie dies rudimentär durch Dateikennungen (z. B. `*.c` für Quelltexte in C) geleistet wird. Diese Festlegung liegt außerhalb des Rahmens dieser Arbeit, sondern hängt allein von der verwendeten Umgebung (Betriebssystem, Kommandooberfläche, Steuerprogramme wie `make` usw.) ab.

# Kapitel 4

# Anwendung von Syntaxerweiterungen

In diesem Kapitel wird gezeigt, wie die Basissprache durch Einsatz der Erweiterungssprache um verschiedene Standardkonstrukte und datenbankspezifische Konstrukte erweitert werden kann. Außerdem werden Normalisierungen vorgestellt, die normalerweise in die Abstraktionsphase fallen.

Die Normalisierungen vereinfachen die Benutzung der Basissprache und werden deshalb teilweise in der Beschreibung der semantischen Interpretation einer Syntaxerweiterung vorausgesetzt. Da die Beschreibung der Normalisierungen komplexer ist als die Beschreibung von Standardkonstrukten, werden die Normalisierungen erst nach den Standardkonstrukten vorgestellt.

## 4.1 Standardkonstrukte

Als Beispiele für die Emulierung von Standardkonstrukten durch Syntaxerweiterungen werden hier einige Konstrukte herangezogen, die in TL [Matthes 93] definiert sind, in TLMin aber fehlen. Es zeigt sich, daß durch Syntaxerweiterungen der Sprachumfang einer Basissprache erheblich reduziert werden kann, was die Komplexität des Compilers der Basissprache verringert.

### 4.1.1 Schleifen

Ausgehend von der Annahme, daß die Basissprache nur ein Schleifenkonstrukt **loop-exit** anbietet, werden andere Schleifenkonstrukte als Syntaxerweiterung emuliert.

#### 4.1.1.1 While-Schleife

In einer While-Schleife wird ein Block von Anweisungen wiederholt, solange eine Bedingung erfüllt ist.

Beispiel:

**while** *not(eof(inFile))* **do** *writeChar(outFile, readChar(inFile))* **end**

Solange das Prädikat *not(eof(inFile))* erfüllt ist, wird die Anweisung *writeChar(...)* ausgeführt.

Syntaxerweiterung:

```
LET whileG ::= "while" pred = valG "do" act = bndsG "end"
  => VAL
    loop
      if VAL pred then BNDS act else exit end
    end
  END
ASSIGN valG ::= COPY valG | COPY whileG
```

Anwendung und Expansion:

**while** *not(eof(inFile))* **do** *writeChar(outFile, readChar(inFile))* **end**
wird umgewandelt zu:

```
loop
  if not(eof(inFile)) then writeChar(outFile, readChar(inFile)) else exit end
end
```

#### 4.1.1.2 Repeat-Schleife

In einer Repeat-Schleife wird ein Block von Anweisungen wiederholt, bis eine Bedingung erfüllt ist.

Beispiel:

**repeat** *writeChar(outFile, readChar(inFile))* **until** *eof(inFile)*

Die Anweisung *writeChar(...)* wird ausgeführt, bis das Prädikat *eof(inFile)* erfüllt ist.

Syntaxerweiterung:

```
LET repeatG ::= "repeat" act = bndsG "until" pred = valG
  => VAL
    loop
      BNDS act
      if VAL pred then exit end
    end
  END
ASSIGN valG ::= COPY valG | COPY repeatG
```

Anwendung und Expansion:

**repeat** *writeChar(outFile, readChar(inFile))* **until** *eof(inFile)*
wird umgewandelt zu:

```
loop
  writeChar(outFile, readChar(inFile))
  if eof(inFile) then exit end
end
```

#### 4.1.1.3 Zählschleife

In einer Zählschleife wird ein ganzzahliges Intervall durchlaufen und für jedes Element dieses Intervalls ein Block von Anweisungen ausgeführt.

Beispiel:

**for** *i = 1* **to** *10* **do** *print.int(i) print.ln()* **end**

Der Zähler *i* durchläuft das Intervall vom Anfangs- bis zum Endwert. Anfangs- und Endwert werden genau einmal in dieser Reihenfolge ausgewertet. Wenn der Anfangswert größer als der Endwert ist, wird rückwärts gezählt, sonst vorwärts. Für jeden Wert des Zählers *i* werden die Anweisungen *print.int(i) print.ln()* ausgeführt, wobei der Zähler *i* zwar sichtbar, aber nicht veränderbar ist. Wenn Anfangs- und Endwert gleich sind, werden die Aktionen einmal ausgeführt.

Folgende Funktionen werden benötigt:

```
+(? :Int  ? :Int) :Int          (* add two integer values *)
>(? :Int  ? :Int) :Bool         (* true if first integer value is greater than second *)
==(? :Int  ? :Int) :Bool        (* true if both integer values are equal *)
:=(? :Var(Int)  ? :Int) :Ok     (* assign second operand to first (mutable) *)
```

Syntaxerweiterung:

```
LET forG ::=
  "for" id = ideG "=" first = valG "to" last = valG "do" act = bndsG "end"
  => VAL
    begin
      let localFirst = VAL first
      let localLast = VAL last
      let inc = if localFirst > localLast then ~1 else 1 end
      letvar index = localFirst
      loop
        let IDE id = index
        BNDS act
        if index == localLast then exit else index := index + inc end
      end
```

```
  end
 END
ASSIGN valG ::= COPY valG | COPY forG
```

Anwendung und Expansion:

**for** *i = 1* **to** *10* **do** *print.int(i) print.ln()* **end**
wird umgewandelt zu:

```
begin
 let localFirst = 1
 let localLast = 10
 let inc = if localFirst > localLast then ~1 else 1 end
 letvar index = localFirst
 loop
  let i = index
  print.int(i)  print.ln()
  if index == localLast then exit else index := index + inc end
 end
end
```

Die Aussagen über die Häufigkeit und Reihenfolge der Auswertung sind wichtig, weil jede Auswertung zu Seiteneffekten führen kann, was oft unerwünscht ist. Wenn im vorstehenden Beispiel statt der lokalen Variablen *localFirst* und *localLast* die Platzhalter *first* und *last* direkt eingesetzt werden, wird der durch die Platzhalter repräsentierte Syntaxbaum repliziert, wobei jede Expansion die Seiteneffekte hervorruft. Eine Vertauschung der Auswertungsreigenfolge kann ähnliche Effekte hervorrufen.

Die folgende Syntaxerweiterung implementiert eine andere Semantik für die Zählschleife:

```
LET forG ::=
 "for" id = ideG "=" first = valG "to" last = valG "do" act = bndsG "end"
 => VAL
  begin
   let inc = if VAL first > VAL last then ~1 else 1 end
   letvar index = VAL first
   loop
    let IDE id = index
    BNDS act
    if index == VAL last then exit else index := index + inc end
   end
  end
 END
```

Die Anwendung dieser Syntaxerweiterung kann unerwünschte Seiteneffekte hervorrufen.

Zur Veranschaulichung wird eine Funktion *getCnt* definiert, die den Wert eines globalen Zählers als Seiteneffekt ändert:

```
letvar counter = 0
let getCnt = fun()
  begin
    counter := counter + 5
    counter
  end
```

Die folgende Anwendung und Expansion der Syntaxerweiterung nutzt die globale Variable *counter* und die Funktion *getCnt*:

**for** *i* = *counter* **to** *getCnt()* **do** ... **end**
wird umgewandelt zu:

```
begin
  let inc = if counter > getCnt() then ~1 else 1 end
  letvar index = counter
  loop
    let i = index
    ...
    if index == getCnt() then exit else index := index + inc end
  end
end
```

Durch die beim Aufruf von *getCnt* hervorgerufenen Seiteneffekte wird die Variable *counter* mehrmals unerwartet geändert.

### 4.1.2 Bedingungen und Verzweigungen

Ausgehend von der Annahme, daß die Basissprache nur über ein **if-then-else**-Konstrukt verfügt, werden verschiedene Bedingungskonstrukte emuliert.

#### 4.1.2.1 Kurzschluß-Und

Ein Kurzschluß-Und (*short-circuit and*) ist eine Verknüpfung zweier Prädikate mit dem Und-Operator, wobei das zweite Prädikat nur ausgewertet wird, wenn das erste Prädikat erfüllt ist.

Beispiel:

{*i* <= *size(arr)*} **andif** {*arr[i]* == ... }

Nur wenn beide Prädikate *true* sind, ist das gesamte Prädikat *true*. Das erste Prädikat *i* <= *size(arr)* wird genau einmal ausgewertet. Falls das erste Prädikat *false* ist, wird das zweite nicht ausgewertet, ansonsten einmal. Dies ist wichtig, wenn die Ausführung des zweiten Prädikats zu einer Ausnahme (*exception*) führen kann, wenn das erste Prädikat nicht erfüllt ist.

Eine intuitive Form der Syntaxerweiterung ist folgende:

**LET** *andifG ::=* *pred1* = **COPY** *valG* "andif" *pred2* = *valG*
  => **VAL**
    **if VAL** *pred1* **then VAL** *pred2* **else** *false* **end**
  **END**
**ASSIGN** *valG ::=* **COPY** *valG* | **COPY** *andifG*

Diese Syntaxerweiterung ist bei LL(1)-Parser-Generatoren nicht erlaubt, weil eine Linksrekursion in *valG* vorliegt und beim Parsen nicht entschieden werden kann, wann die Produktion *andifG* anzuwenden ist. Deshalb sollte folgende Form der Syntaxerweiterung gewählt werden:

**LET** *andifG(pred1 ::***VAL***) ::=*
  "andif" *pred2* = *valG*
  => **VAL**
    **if VAL** *pred1* **then VAL** *pred2* **else** *false* **end**
  **END**
| *(* empty *)*
  => **VAL VAL** *pred1* **END**
**ASSIGN** *valG ::= val1* = **COPY** *valG newVal = andifG(val1)*
  => **VAL VAL** *newVal* **END**

Nach der Syntaxerweiterung soll ein Wert (*value*), beschrieben durch *valG*, entweder ein (normaler) Wert oder ein Wert gefolgt von **andif** und einem zweiten Wert sein. Um Linksrekursion und Zweideutigkeit, die ein LL(1)-Parser-Generator nicht behandeln kann, zu vermeiden, wird der erste erkannte Wert *val1* als vererbtes Attribut *pred1* an die Produktion *andifG* weitergegeben. Wenn das Schlüsselwort **andif** folgt, wird der zweite Wert *pred2* erkannt und beide Werte zu einer **if**-Anweisung umgewandelt. Ansonsten wird der erste (und einizige) Wert *pred1* zurückgegeben.

In der vorgestellten Syntaxerweiterung ist es nur über eine versteckte Rekursion in *andifG* über *pred2* = *valG* möglich, eine Folge von mehr als zwei **andif**s zu behandeln. Die folgende Syntaxerweiterung zeigt eine explizite Rekursion:

**LET** *andif2G(pred1 ::***VAL***) ::=**
  "andif" *pred2* = **COPY** *valG*
  => **VAL**
    **if VAL** *pred1* **then VAL** *pred2* **else** *false* **end**
  **END**
**LET** *andif1G(pred1 ::***VAL***) ::=**
  *val = andif2G(pred1) newVal = andif1G(val)*
  => **VAL VAL** *newVal* **END**
| *(* empty *)*
  => **VAL VAL** *pred1* **END**
**ASSIGN** *valG ::= val* = **COPY** *valG newVal = andif1G(val)*
  => **VAL VAL** *newVal* **END**

In dieser Syntaxerweiterung werden die Produktionen *andif1G* und *andif2G* durch eine Rekursion in *andif1G* solange wiederholt, bis kein **andif** mehr folgt.

Anwendung und Expansion:

{*i <= size(arr)*} **andif** {*arr[i] == ...*}
wird umgewandelt zu:

**if** {*i <= size(arr)*} **then** {*arr[i] == ...*} **else** *false* **end**

Analog kann ein Kurzschluß-Oder emuliert werden.

#### 4.1.2.2 Zusicherung

Eine Zusicherung (*assertion*) löst eine Ausnahme aus, wenn eine Bedingung nicht erfüllt ist.

Beispiel:

**assert** {*i <= size(arr)*}

Wenn das Prädikat *i <= size(arr) false* ist, wird eine Ausnahme ausgelöst.

In TLMin sind im Gegensatz zu TL keine Ausnahmen (*exceptions*) vorhanden. Es wird angenommen, daß die Implementation *exception* folgende Schnittstelle erfüllt:

```
interface Exception
export
  :T <:Ok
  new() :T                          (* return a new exception handle *)
  raise(? :T) :Nok                  (* raise an exception, call last handler *)
  push(handler :Fun() :Ok) :Ok      (* push an exception handler *)
  pop() :Ok                         (* pop last handler *)
  inException() :T                  (* return handle if in exception handler *)
end
```

Mit Hilfe dieser Schnittstelle wird eine neue Ausnahme *assertError* angelegt:

**let** *assertError = exception.new()*

Syntaxerweiterung:

**LET** *assertG* ::= "assert" *pred* = *valG*
  => **VAL if VAL** *pred* **then ok else** *exception.raise(assertError)* **end END**
**ASSIGN** *valG* ::= **COPY** *valG* | **COPY** *assertG*

Anwendung und Expansion:

**assert** {*i <= size(arr)*}
wird umgewandelt zu:

**if** {*i <= size(arr)*} **then ok else** *exception.raise(assertError)* **end**

Da diese Anweisung oft nur für die Testphase gedacht ist, bietet es sich an, für die Übersetzung der endgültigen Version diese alternative Syntaxerweiterung zu verwenden:

**LET** *assertG* ::= **assert** *pred* = *valG* => **VAL ok END**

Dies entspricht der konditionalen Übersetzung, die der Präprozessor von ANSI-C anbietet [Kernighan, Ritchie 90]:

**#ifdef** *assert*
  *pred*
**#else**
**#end**

Der Präprozessor fügt das Prädikat *pred* nur in den Text ein, wenn der Parameter *assert* gesetzt ist.

#### 4.1.2.3 Umwandlung von Elsif-Zweigen und Ergänzung fehlender Else-Zweige

Elsif-Zweige werden in einfache Verzweigungen (**else-if-then**) umgeformt. Ein fehlender Else-Zweig in einer If-Verzweigung wird mit einer leeren Sequenz von Anweisungen ergänzt.

Beispiel:

**if** $pred_1$ **then** $act_1$ **elsif** $pred_2$ **then** $act_2$ **else** $act_{n+1}$ **end**
**if** *pred* **then** *act* **end**
wird umgewandelt zu:

**if** $pred_1$ **then** $act_1$ **else**
  **if** $pred_2$ **then** $act_2$ **else**
    $act_{n+1}$
  **end**
**end**
**if** *pred* **then** *act* **else end**

Die Aktion $act_i$ des ersten Prädikats $pred_i$ ($i = 1..n$), das *true* ist, wird ausgeführt. Falls kein Prädikat *true* ist, wird die Aktion $act_{n+1}$ ausgeführt. Falls kein Else-Zweig mit der Aktion $act_{n+1}$ angegeben ist, wird eine leere Aktion erzeugt.

Syntaxerweiterung:

**ASSIGN** *ifG* ::= "if" *pred* = *valG* "then" *act* = *bndsG* *tail* = *tailG* "end"
  => **VAL**
    **if VAL** *pred* **then BNDS** *act* **else BNDS** *tail* **end**
  **END**
**LET** *tailG* ::=
  (* *empty* *) => **BNDS** (* *empty* *) **END**
| "else" *act* = *bndsG* => **BNDS BNDS** *act* **END**

```
| "elsif" pred = valG "then" act = bndsG  tail = tailG
   => BNDS
    if VAL pred then BNDS act else BNDS tail end
   END
```

## 4.2 Normalisierungen

In TL existieren verschiedene Normalisierungen bzw. abkürzende Schreibweisen, die redundant sind [Matthes 93, Kap. 6.3]. Die Normalisierung der konkreten Syntax in äquivalente abstrakte Syntaxbäume wird von der Abstraktionsphase vorgenommen. In TLMin wird diese Normalisierung nicht unterstützt. In diesem Abschnitt wird gezeigt, wie sich Normalisierungen mit Syntaxerweiterungen emulieren lassen.

### 4.2.1 Anonyme Wert- und Typbindungen

Anstatt einer Wert-/Typbindung kann ein Wert/Typ auftreten, der in eine anonyme Bindung umgewandelt wird.

Beispiel:

**if** ... **then** *true* **else** *false* **end**
*f(:Int 1)*
wird umgewandelt zu:

**if** ... **then let** *?* = *true* **else let** *?* = *false* **end**
**let** *?* = *f*(**Let** *?* = *Int  let ?* = *1)*

Syntaxerweiterung:

```
ASSIGN bndG ::=
  "Let" id = ideG "=" typ = typG
  => BNDS Let IDE id = TYP typ END
| "let" id = ideG "=" val = valG
  => BNDS let IDE id = VAL val END
| ":" typ = typG
  => BNDS Let ? = TYP typ END
| val = valG
  => BNDS let ? = VAL val END
```

### 4.2.2 Bezeichnerlisten

In Signaturen können Bezeichnerlisten mit einem Typ bzw. Supertyp auftreten, der für jeden Bezeichner gilt.

Beispiel:

**fun**(*:X, Y <:***Ok** *x1, x2 :X*) ...
wird umgewandelt zu:

**fun**(*:X <:***Ok** *:Y <:***Ok** *x1 :X x2 :X*) ...

Syntaxerweiterung:

```
ASSIGN sigG ::=
  ":" ide = ideG  sigs, suptyp = newSigTypG
  => SIGS : IDE ide <: TYP suptyp  SIGS sigs END
| ide = ideG  sigs, typ = newSigValG
  => SIGS IDE ide : TYP typ  SIGS sigs END
LET newSigTypG ::=
  "<:" suptyp = typG
  => SIGS END
     TYP TYP suptyp END
| "," ide = ideG  sigs, suptyp = newSigTypG
  => SIGS : IDE ide <: TYP suptyp  SIGS sigs END
     TYP suptyp END
LET newSigValG ::=
  ":" typ = typG
  => SIGS END
     TYP TYP typ END
| "," ide = ideG  sigs, typ = newSigValG
  => SIGS IDE ide : TYP typ  SIGS sigs END
     TYP TYP typ END
```

Die Schwierigkeit bei dieser Syntaxerweiterung besteht darin, daß der Typ bzw. Supertyp für jeden Bezeichner der Liste einmal kopiert werden muß. Deshalb werden von den (rekursiven) Produktionen *newSigTypG* und *newSigValG* sowohl die schon konstruierten Signaturen als auch der Typ bzw. Supertyp als abgeleitete Attribute zurückgegeben. Beim Aufstieg aus der Rekursion werden die Signaturen sukzessive konstruiert.

### 4.2.3 Inferenz

In einigen Programmiersprachen wird zu jeder Wert- oder Typbindung eine einschränkende Angabe des Typs bzw. Supertyps verlangt, z. B.:

**let** *x :Int = 1*
**Let** *X <:***Ok** *= Int*

In diesem Fall bietet sich eine Syntaxerweiterung an, die eine Typinferenz simuliert, so daß eine abkürzende Schreibweise ohne Typ bzw. Supertyp möglich ist.[1]

[1] In TLMin wird diese Inferenz von vornherein vorausgesetzt, um die Sprache übersichtlich zu halten. D. h. es ist in TLMin nicht möglich, einschränkende Typen anzugeben.

Zur Realisierung muß eine Meta-Funktion (siehe Abschnitt 2.5.5) *TypeOfVal* vorhanden sein, die zu einem Wertausdruck den zugehörigen Typ inferiert. Diese Meta-Funktion wird zur Übersetzungszeit ausgewertet, hat aber keine Auswirkungen auf den erzeugten Zwischen- und Zielkode. In TL ist eine solche Funktion nicht vorhanden. Stattdessen wird an Stelle des Typs eine interne Typkonstante *wrongType* eingesetzt, die der Typprüfung signalisiert, daß der Typ inferiert werden muß. Wie in Abschnitt 4.3 gezeigt wird, wäre diese Meta-Funktion auch an anderer Stelle von Nutzen. In der vorliegenden Implementation von TLMin ist sie nicht vorhanden, da dies weitgehende Eingriffe in den abstrakten Syntaxbaum und alle darauf arbeitenden Operationen bedeutet hätte.

Beispiel:

**let** *x = 1*
**Let** *X = Int*
wird umgewandelt zu:

**let** *x :TypeOfVal(1) = 1*
**Let** *X <:Int = Int*

Syntaxerweiterung:

**ASSIGN** *bndG* =
  "let" *id = ideG* ":" *typ = typG* "=" *val = valG*
  => **BNDS let IDE** *id :* **TYP** *typ* = **VAL** *val* **END**
| "Let" *id = ideG* "<:" *supTyp = typG* "=" *typ = typG*
  => **BNDS Let IDE** *id :* **TYP** *supTyp* = **TYP** *typ* **END**
| "let" *id = ideG* "=" *val = valG*
  => **BNDS let IDE** *id : TypeOfVal(***VAL** *val)* = **VAL** *val* **END**
| "Let" *id = ideG* "=" *typ = typG*
  => **BNDS Let IDE** *id* <: **TYP** *typ* = **TYP** *typ* **END**

### 4.2.4 Infix-Operatoren

Infix-Operatoren sind zweistellige Funktionen mit einer besonderen Syntax, bei der der Funktionsname (der Infixoperator) zwischen den Parametern steht. Sie werden in Funktionsaufrufe umgewandelt.

Beispiel:

$val_1$ + $val_2$
wird umgewandelt zu:

$+(val_1\ val_2)$

Nach [Aho et al. 88, S. 39/40] könnte eine Syntaxerweiterung für (linksassoziative) Infix-Operatoren folgendermaßen aussehen:

**ASSIGN** *valG ::=*
  *val* = **COPY** *valG* => **VAL VAL** *val* **END**
| *val1* = *valG* *infix* = *infixG* *val2* = *valG*
  => **VAL IDE** *infix*(**VAL** *val1* **VAL** *val2*) **END**

Diese intuitive Form hat den Nachteil, linksrekursiv zu sein, so daß ein LL-Parser-Generator sie nicht akzeptiert. Deshalb wird die Linksrekursion gemäß [Aho et al. 88, S. 58/59] aufgelöst:

**ASSIGN** *valG ::=*
  *val* = **COPY** *valG* *newVal* = *newVal1G*(*val*) => **VAL VAL** *newVal* **END**
**LET** *newVal1G*(*val1* ::**VAL**) *::=*
  (* *empty* *) => **VAL VAL** *val1* **END**
| *val12* = *newVal2G*(*val1*) *newVal* = *newVal1G*(*val12*) => **VAL VAL** *newVal* **END**
**LET** *newVal2G*(*val1* ::**VAL**) *::=*
  *infix* = *infixG* *val2* = **COPY** *valG* => **VAL IDE** *infix*(**VAL** *val1* **VAL** *val2*) **END**

Das Durchreichen der schon geparsten Werte über das ererbte Attribut *val1* ermöglicht die linksassoziative Konstruktion des abstrakten Syntaxbaums, so daß der Wert *v1–v2+v3* als {*v1–v2*}*+v3* und nicht als *v1–*{*v2+v3*} übersetzt wird.

Alle Infix-Operatoren besitzen in dieser Syntaxerweiterung die gleiche Priorität. Um unterschiedliche Prioritäten zu erreichen, müßten die Operatoren in Klassen eingeteilt und auf unterschiedliche Produktionen verteilt werden, z. B. für Ausdrücke, Terme und Faktoren [Aho et al. 88, S. 39]. Ein anderer Weg ist die Angabe von Prioritäten für Infix-Operatoren, die vom Parser-Generator verarbeitet werden [Aho et al. 88, Kap. 4.6].

Eine analoge Syntaxerweiterung bietet sich auch für Typen an, um Infix-Typoperatoren zu unterstützen.

### 4.2.5 Funktionsdeklaration

Zur Deklaration von Funktionen ist eine abkürzende Schreibweise möglich.

Beispiel:

**let** *f(p1 :P2)(p2 :P2) = val*
wird umgewandelt zu:

**let** *f* = **fun**(*p1 :P1*) **fun**(*p2 :P2*) *val*

Die Syntaxerweiterung setzt bei den Bindungen an. Im folgenden Beispiel wird nur der neue Teil der Bindungsproduktion für Wertbindungen gezeigt:

**ASSIGN** *bndG ::=*
  "let" *id* = *ideG* *val* = *funTailG*
  => **BNDS let IDE** *id* = **VAL** *val* **END**

```
LET funTailG ::=
  "=" val = valG
  => VAL VAL val END
| "(" sigs = sigsG ")" val = funTailG
  => VAL fun(SIGS sigs) VAL val END
```

Eine analoge Syntaxerweiterung bietet sich für die Deklaration von Typoperatoren an.

## 4.3 Datenbankspezifische Konstrukte

In diesem Abschnitt werden Syntaxerweiterungen für datenbankspezifische Konstrukte vorgestellt. Dabei werden Bibliotheken vorausgesetzt, die die entsprechende Funktionalität implementieren. Zum Zeitpunkt der Anfertigung dieser Arbeit standen solche Bibliotheken nur in TL, jedoch nicht in TLMin zur Verfügung.

### 4.3.1 Iteration

Eine Iteration über eine Kollektion eines Massendatentyps führt für jedes Element der Kollektion, das eine Bedingung erfüllt, eine Anzahl von Anweisungen aus.

Beispiel:

**for each** *p* **in** *persons : p.age > 17* **do** *printPerson(p)* **end**

Es wird angenommen, daß der Wert *persons ungefähr* folgendem Typ genügt (vgl. Abschnitt 3.2.3.6):

```
Let Bulk(:E <:Ok) =
Tuple
  init() :Ok          (* initialize for iteration *)
  empty() :Bool       (* true if no more elements *)
  first() :E          (* get first element if not(empty()) *)
  next() :Ok          (* next element *)
end
```

Syntaxerweiterung:

```
LET foreachG ::=
  "for" "each" id = ideG "in" table = valG ":" pred = valG "do" act = bndsG "end"
=> VAL
  begin
    let b = VAL table
    b.init()
    loop
      if b.empty() then exit end
```

```
    let IDE id = b.first()
    if VAL pred then BNDS act end
    b.next()
  end
 end
END
ASSIGN valG ::= COPY valG | COPY foreachG
```

Das folgende Beispiel zeigt eine Expansion dieser Syntaxerweiterung:

**for each** *p* **in** *persons : p.age > 17* **do** *printPerson(p)* **end**
wird umgewandelt zu:

```
begin
 let b = persons
 b.init()
 loop
  if b.empty() then exit end
  let p = b.first()
  if p.age > 17 then printPerson(p) end
  b.next()
 end
end
```

### 4.3.2 Selektion, Projektion und Kombination

Die drei grundlegenden Operationen zur Abfrage von relationalen Datenbanken, Selektion, Projektion und Kombination [Schmidt 87, Kap. 1.4.3.2], werden durch die SQL-Anweisung **select-from-where** erfaßt [SQL 87]. Die Syntax dieser Anweisung wird im folgenden in Aufrufe einer generischen Bibliothek abgebildet.

Beispiel:

**select** *p.name c.name* **from** *persons p, cities c* **where** *p.city == c.zip*

Die Abfrage ergibt die Namen aller Personen zusammen mit den Namen der Städte, in denen die Personen wohnen. Die Selektion wird durch die Bedingung *p.city == c.zip* erreicht. Die Projektion erfolgt auf die beiden Attribute *p.name* und *c.name*. Die beiden Tabellen *persons* und *cities* werden kombiniert.

Es wird eine Bibliothek *iter* für Tabellen benötigt, die folgende Schnittstelle implementiert:

```
interface Iter
export
 :T(:E <:Ok) <:Ok
 singleton(:E <:Ok  element :E) :T(:E)
```

```
  select(:E,R <:Ok
    from :T(:E)
    where :Fun(? :E) :Bool
    project :Fun(? :E) :R) :T(:R)
  flatten(:E <:Ok  iter2 :T(:T(:E))) :T(:E)
end
```

Der Typoperator $T$ beschreibt eine Tabelle mit dem Elementtyp $E$. Die Funktion *singleton* erzeugt einen Iterator mit einem Element. Die Funktion *select* selektiert aus einer Tabelle *from* alle Elemente, die das Prädikat *where* erfüllen, projeziert sie durch die Funktion *project* auf einen neuen Elementtyp $R$ und gibt eine Tabelle mit diesen Elementen zurück. Die Funktion *flatten* erzeugt aus einem Iterator von Iteratoren einen Iterator, der alle Elemente der Iteratoren enthält.

Syntaxerweiterung:

```
LET selectG ::=
  "select" prj = bndsG "from" sel = select1G(prj)
  => VAL VAL sel END
LET select1G(prj ::BNDS) ::=
  table = valG  ide = ideG  newPrj, pred = select2G(prj)
  => VAL
    iter.flatten(Let E = ?
      let iter2 =
        iter.select(Let E = ?  Let R = ?
          let from = VAL table
          let where = fun(IDE ide :?) VAL pred
          let project = fun(IDE ide :?) VAL newProj))
  END
LET select2G(prj ::BNDS) ::=
  "," sel = select1G(prj)
  => VAL VAL sel END
    VAL true END
| "where" pred = valG
  => VAL iter.singleton(Let E = ?  let element = tuple BNDS prj end) END
    VAL VAL pred END
ASSIGN valG ::= COPY valG | COPY selectG
```

Die Syntaxerweiterung besteht aus drei Produktionen. Die Produktion *selectG* erkennt die Projektion und reicht die Liste der Bindungen weiter. Der Aufruf der Bibliothek wird in der Produktion *select1G* konstruiert. Die Produktion *select2G* erzeugt bei einer Kombination einen rekursiven Aufruf der Produktion *select1G*; ansonsten erkennt sie das Selektionsprädikat *pred* und erzeugt aus der durchgereichten Projektionsliste eine Projektion.

Die Schwierigkeit bei dieser Syntaxerweiterung ist, daß die temporären Namen *ide* für Elemente aller Tabellen *table* gleichzeitig sichtbar sein müssen, um das Selektionsprädikat

und die Projektion richtig zu binden. Es ist nicht möglich, eine Selektionsfunktion *iter.select* für *beliebig* viele Tabellen zu konstruieren, sondern nur für eine feste Anzahl von Tabellen, z. B. für zwei Tabellen:

*select2(:E1,E2,R <*:**Ok**
*from1 :T(:E1) from2 :T(:E2)*
*where :***Fun***(? :E1 ? :E2) :Bool*
*project :***Fun***(? :E1 ? :E2) :R) :T(:R)*

Die Syntaxerweiterung muß also geschachtelte Aufrufe der Funktion *iter.select* erzeugen, wobei erst in der innersten Schachtelung alle Bezeichner sichtbar sind und somit erst und nur dort das Selektionsprädikat und die Projektionsliste einen Sinn ergeben.

Das folgende Beispiel zeigt eine Expansion dieser Syntaxerweiterung:

**select** *p.name c.name* **from** *persons p, cities c* **where** *p.city == c.zip*
wird umgewandelt zu:

```
iter.flatten(Let E = ?
  let iter2 =
    iter.select(Let E = ?  Let R = ?
      let from = persons
      let where = fun(p :?) true
      let project = fun(p :?)
        iter.flatten(Let E = ?
          let iter2 =
            iter.select(Let E = ?  Let R = ?
              let from = cities
              let where = fun(c :?) p.city == c.zip
              let project = fun(c :?)
                iter.singleton(Let E = ?  let element = tuple p.name c.name end)))))
```

Wie auf Seite 44 erwähnt, sollte der Compiler der Basissprache über Typinferenz verfügen, um die fehlenden Typangaben *?* zu inferieren. Wenn dies nicht der Fall ist, läßt sich die Inferenz teilweise mit der in Abschnitt 4.2.3 erwähnten Meta-Operation *TypeOfVal* simulieren. Zum Beispiel kann der Typ $R$ im obigen Beispiel folgendermaßen inferiert werden:

**Let** $R$ = *TypeOfVal(***tuple** *p.name c.name* **end***)*

Der Elementtyp einer Tabelle *table* kann so nicht ermittelt werden, da sich nur der Tabellentyp *iter.T(:E)* inferieren läßt. Es könnte eine weitere Meta-Operation *match* zur Ermittlung eines Parametertyps eines Typoperators eingesetzt werden, ähnlich der *case*-Operation in TRPL [Stemple et al. 92b] oder der *typecase*-Operation in TL [Matthes 93]:

**Let** $E$ = **match** $X$ **in** *Iter.T(:X)* **with** *TypeOfVal(persons)*

Die Semantik von **match** *id* **in** *typ1* **with** *typ2* ist: Die Typausdrücke *typ1* und *typ2* werden miteinander verglichen. In *typ1* ist der Platzhalter *id* enthalten. Der Teilausdruck, der in *typ2* an der Stelle steht, die in *typ1* durch *id* bezeichnet ist, wird als Ergebnis geliefert. Im

obigen Beispiel werden die Typausdrücke *Iter.T(:X)* und *Iter.T(:Person)* verglichen. Der Platzhalter *X* korrespondiert mit dem Typ *Person*.

### 4.3.3 Transaktion

Eine Transaktion ist eine geschützte Operation, die beim Scheitern der Ausführung alle Änderungen zurücksetzt.

Beispiel:

```
let insertPerson = transaction(p :Person)
begin
  noOfPersons := noOfPersons + 1
  persons.insert(p)
end
```

Diese Transaktion zählt einen Zähler *noOfPersons* der Anzahl der Personen hoch und fügt eine Person *p* in eine Menge von Personen *persons* ein. Wie die folgende Implementation der Einfüge-Operation *insert* zeigt, wird die Operation abgebrochen, falls die Person sich schon in der Menge befindet:

```
let insert = transaction(:E <:Ok  e :E)
  if member(e) then exception.raise(error) else ... end
```

Es wird neben den auf Seite 61 vorgestellten Ausnahmen eine Bibliothek *transaction* zur Transaktionsverwaltung benötigt, die folgende Schnittstelle implementiert (angelehnt an [Niederée 92]):

```
interface Transaction
export
  Let T = Tuple
    commit :Fun() :Ok     (* commit transaction *)
  end
  new() :T                (* start new transaction *)
end
```

Die Funktion *new* erzeugt ein Transaktionsobjekt des Typs *T*; gleichzeitig wird die (einzige) kontextverändernde Funktion, die Zuweisungsfunktion :=, umdefiniert, so daß eine Zuweisung zu einem Eintrag in das Undo-Log dieser Transaktion führt. Dies setzt voraus, daß die Funktion := global definiert und veränderlich ist. Der erfolgreiche Abschluß der Transaktion wird durch *commit* gemeldet. Wenn während der Transaktion eine Ausnahme auftritt, wird eine Routine zur Ausnahme-Behandlung aufgerufen, das Undo-Log abarbeitet. Sowohl *commit* als auch die Behandlung der Ausnahme restaurieren die ursprüngliche Bedeutung der Funktion :=.

Syntaxerweiterung:

```
LET transactionG ::=
  "transaction" "(" sigs = sigsG ")" body = valG
=> VAL
  fun(SIGS sigs)
  begin
    let t = transaction.new()
    let result = VAL body
    t.commit()
    result
  end
END
ASSIGN valG ::= COPY valG | COPY transactionG
```

Die Produktion *transactionG* erzeugt eine Funktion mit einem Funktionskörper, der ein Undo-Log aufbaut, alle Ausnahmen abfängt und damit die Abarbeitung eines Undo-Log ermöglicht.

Das folgende Beispiel zeigt eine Anwendung der Syntaxerweiterung:

```
let insertPerson = transaction(p :Person)
begin
  noOfPersons := noOfPersons + 1
  persons.insert(p)
end
```

wird umgewandelt zu:

```
let insertPerson = transaction(p :Person)
  fun(p :Person)
  begin
    let t = transaction.new()
    let result =
      begin
        noOfPersons := noOfPersons + 1 (* 1 *)
        persons.insert(p)
      end
    t.commit()
    result
  end
```

An der durch 1 markierten Stelle wird die umdefinierte Funktion := benutzt, die gleichzeitig einen Eintrag ins Undo-Log erzeugt. Falls die Ausführung von *persons.insert* eine Ausnahme auslöst, wird das Undo-Log abgearbeitet und die Änderung von *noOfPersons* rückgängig gemacht.

Dieses Beispiel zeigt eine Abstraktion durch Syntaxerweiterungen, die durch funktionale

Abstraktionen nicht geleistet werden kann. Es kann keine Funktion zur Transaktionsgenerierung geschrieben werden, die für beliebige Parameterfunktionen eine Transaktion erzeugt. Stattdessen wird für eine bestimmte Anzahl von Parametern eine Generatorfunktion geschrieben [Niederée 92], ohne daß die Funktionen je vollständig sein könnten:

**let** *g0* = **fun**(*:R <:***Ok** *body :***Fun**() *:R*) ...
**let** *g1* = **fun**(*:P1,R <:***Ok** *body :***Fun**(*? :P1*) *:R*) ...
**let** *g2* = **fun**(*:P1,P2,R <:***Ok** *body :***Fun**(*? :P1 ? :P2*) *:R*) ...
...
**let** *insertPerson* = *g1*(**fun**(*p :Person*) ...)

# Kapitel 5

# Bindungen in Syntaxerweiterungen

Die naive Expansion von Syntaxerweiterungen kann Bindungsprobleme (*variable captures*) verursachen. Beispielsweise sei folgende (sinnlose) Syntaxerweiterung definiert, in der lokale Bindungen auftreten und globale Bindungen durch einen Platzhalter eingeführt werden:

"expand" *bnds* = *bndsG* "end"
=> **BNDS**
  **let** *x* = *1*
  **BNDS** *bnds*
  **let** *y* = *x*
**END**

Die Syntaxerweiterung wird in einem bestimmten Kontext angewendet; in dem Beispiel und allen folgenden Beispielen sind expandierte Syntaxerweiterungen mit > und instantiierte Platzhalter mit >> gekennzeichnet:

**let** *x* = *"John"*
**expand**
  **let** *y* = *"Mary"*
  **let** *x* = *concat(x y)*
**end**
**let** *z* = *y*

wird naiv umgewandelt zu:

**let** *x* = *"John"*
> **let** *x* = *1*
>> **let** *y* = *"Mary"*
>> **let** *x* = *concat(x y)*
> **let** *y* = *x*
**let** *z* = *y*

Durch die naive Expansion der Syntaxerweiterung werden lokale Bindungen aus der Definition der Syntaxerweiterung global sichtbar und verdecken globale Bindungen (vgl. Bindungsregeln in Abschnitt 3.1.3). Durch die naive Expansion des Platzhalters werden (lokale)

Bindungen in der Syntaxerweiterung eingeführt, die lokale Bindungen aus der Definition verdecken.

In diesem Kapitel werden verschiedenen Lösungsansätze skizziert und ein auf de Bruijn-Indices aufbauender Ansatz [Abadi et al. 90; Cardelli 91] weiterentwickelt. Dieser Ansatz wird zur Lösung der verschiedenen Dimensionen des Problems herangezogen und abschließend diskutiert.

## 5.1 Ansätze zur Lösung von Bindungsproblemen

Leavenworth [Leavenworth 66] spricht Bindungsprobleme im Kontext von „Syntax-Makros“ kurz an und schlägt als Lösung eine Erweiterung der Bezeichner um „Blocknummern“ vor, ohne dieses Verfahren weiter auszuführen.

Kohlbecker [Kohlbecker 86; Kohlbecker et al. 86] formuliert Bedingungen für „hygienische Makros“ (*hygienic macros*) und entwickelt einen Algorithmus, der durch Umbenennungen (*colouring*) die unerwünschten Bindungen vermeidet.

Bawden und Rees [Bawden, Rees 88] lösen die Probleme durch den Abschluß (*closure*) der Ausdrücke in unterschiedlichen syntaktischen Umgebungen (*syntactic environments*).

Clinger und Rees [Clinger, Rees 91] kritisieren, daß Kohlbeckers Lösung zu ineffizient und Bawden/Rees' Lösung auf hochsprachlicher Ebene nicht anwendbar sei. Sie verbinden beide Lösungen, indem Sie sowohl Umgebungen als auch Umbenennungen benutzen. Das Ergebnis ist effizienter als Kohlbeckers Algorithmus und außerdem in eine hochsprachliche Umgebung eingebettet.

Cardelli [Cardelli 91] nutzt de Bruijn-Indices, um Bindungen darzustellen, und Indexmanipulationen, um die Bindungen bei einer Syntaxerweiterung zu erhalten. Dadurch werden Umbenennungen mit dem Problem, „frische“ Bezeichnernamen (*fresh identifiers*) erzeugen zu müssen, vermieden. Cardelli verzichtet darauf, globale Bindungen innerhalb der Syntaxerweiterung zu erhalten. Die folgende Syntaxerweiterung zeigt eine globale Bindung (des Bezeichners x):

**let** $x = \ldots$
"expand" $\Rightarrow x$

Die Anwendung dieser Syntaxerweiterung kann zu folgender unerwünschter Bindung von $x$ führen:

**let** $x = \ldots$
**let** $\ldots =$ **expand**
wird naiv umgewandelt zu:

**let** $x = \ldots$
**let** $\ldots = x$

Auf dem Ansatz von Cardelli baut das im folgenden beschriebene Lösungsverfahren auf. Das Verfahren nutzt zur effizienten Erhaltung der globalen Bindungen einen neuen Indextyp, Cardelli-Indices genannt.

## 5.2 Ein neuer Ansatz zur Lösung von Bindungsproblemen

In diesem Abschnitt wird als äquivalentes Problem zu dem Bindungsproblem bei Syntaxerweiterungen das Substitutionsproblem des Lambda-Kalküls vorgestellt. Mögliche Lösungen dieses Problems basieren auf der Manipulation von de Bruijn-Indices bzw. Cardelli-Indices. Um unnötige Indexmanipulationen zu vermeiden, werden beide Indextypen zusammen eingesetzt. Zwar sind weiterhin Indexmanipulationen nötig, aber weitaus weniger als beim Einsatz nur eines Indextyps.

### 5.2.1 Das Substitutionsproblem im Lambda-Kalkül

Das Problem, bei einer Umstellung der Programmstruktur bestehende Bindungen zu erhalten bzw. keine unerwünschten Bindungen zu erzeugen, läßt sich auf das Substitutionsproblem im Lambda-Kalkül zurückführen [Plotkin 81]: Bei der $\beta$-Reduktion werden gebundene Bezeichner-Referenzen durch den aktuellen Wert des definierenden Bezeichners (einen Lambda-Ausdruck) ersetzt. Dabei entstehende Namenskonflikte (*name clashes*), die zu unerwünschten Bindungen (*variable capture*) führen, müssen durch $\alpha$-Konversion, eine Umbenennung von Bezeichnern, aufgelöst werden.

Es läßt sich zeigen, daß die beiden Operationen $\beta$-Reduktion und $\alpha$-Konversion nicht unabhängig voneinander sind, sondern abwechselnd ausgeführt werden müssen, um ein korrektes (zum ursprünglichen Lambda-Ausdruck äquivalentes) Ergebnis zu erhalten [Meyer 90].

Im Rahmen des Lambda-Kalküls stellt die Substitution nur ein Hilfsmittel dar, um Beweise zu führen. Durch die Einführung von Makros in das Lambda-Kalkül (siehe Abschnitt 5.2.5) wird die Substitution explizit für den Anwender verfügbar und muß automatisch behandelt werden.

Abadi et al. beschreiben in [Abadi et al. 90] eine Lösung des Substitutionsproblems auf der Grundlage von de Bruijn-Indices [de Bruijn 72], um Typprüfungen zu ermöglichen. Alle Bindungen werden durch de Bruijn-Indices ersetzt, so daß die Bezeichner entfernt werden können. Für Substitutionen werden Regeln zur Manipulation der Indices angegeben, so daß die Bindungen erhalten bleiben.

### 5.2.2 de Bruijn-Indices

Ein de Bruijn-Index ist eine positive ganze Zahl, notiert durch *@i*, die einen referenzierenden Bezeichner durch einen Verweis auf das definierende Auftreten des Bezeichners im lexikalischen Kontext ersetzt:

*let a = ...*
*let f = fun(x) a + x*
*let a = ...*
*f(a)*
wird umgewandelt zu:

*let . = ...*
*let . = fun(.) @2 + @1*
*let . = ...*
*@2(@1)*

Die Namen der definierenden Bezeichner besitzen keine Bedeutung mehr. Die referenzierenden Bezeichner werden durch ganze Zahlen ersetzt, die angeben, an welchen Bezeichner im lexikalischen Kontext (rückwärts gezählt) sie gebunden sind. Im Beispiel werden zuerst *a* und anschließend *x* in den Kontext eingefügt. Beim Eintritt in die Funktionsdeklaration von *f* besteht der Kontext aus *(a,x)*. Das referenzierende Auftreten von *a* in der Funktion wird durch den Index @2 ersetzt, die Referenz *x* durch @1. Beim Aufruf der Funktion befindet sich *x* nicht mehr im Kontext, dafür sind *f* und *a* hinzugekommen, so daß der Kontext nun folgende Form hat: *(a,f,a)*. Die zweite Definition von *a* verdeckt die erste, so daß der Funktionsaufruf *f(a)* in @2(@1) umgewandelt wird.

Eine Auflösung des Funktionsaufrufs durch Substitution erfordert eine Umbenennung, weil bei einer naiven Substitution neue (falsche) Bindungen entstehen:

*f(a)*
wird naiv umgewandelt zu:

*a + a*

Hier sollen die beiden Bezeichner *a* unterschiedliche Objekte bezeichnen, was nur durch konsistentes Umbenennen eines Bezeichners, z. B. in *a'*, erreicht werden kann.

Bei Benutzung von de Bruijn-Indices reduziert sich die Umbenennung auf Indexmanipulationen, also die Addition und Subtraktion ganzer Zahlen. Die Indices aller freien Bezeichner in der Funktion *f*, in diesem Fall a, müssen angepaßt werden. Der de Bruijn-Index von *a* muß um die Anzahl der substituierten Parameter (hier einer: *x*) dekrementiert und um die Anzahl der zwischen der Definition der Funktion *f* und der Referenz von *f* eingeführten Bezeichner (hier zwei: *f* und *a*) inkrementiert werden:

*@2(@1)*
wird umgewandelt zu:

*@(2 − 1 + 2) + @1*

Diese Indexmanipulationen werden in [Abadi et al. 90; Matthes 93] formal beschrieben.

### 5.2.3 Cardelli-Indices

Bei jeder Substitution müssen die beschriebenen Indexmanipulationen vorgenommen werden. Sie lassen sich durch die Einführung eines neuen Indextyps, des Cardelli-Indexes,[1] verringern. Im Gegensatz zu einem de Bruijn-Index bezeichnet ein Cardelli-Index die Position des entsprechenden definierenden Bezeichners *vom Anfang* des Kontextes aus. Notiert werden Cardelli-Indices mit *@@i*, wobei *i* eine positive ganze Zahl ist:[2]

*let a = ...*
*let f = fun(x) a + x*
*let a = ...*
*f(a)*
wird umgewandelt zu:

*let . = ...*
*let . = fun(.) @@1 + @@2*
*let . = ...*
*@@2(@@3)*

Allerdings müssen auch Cardelli-Indices bei einer Substitution angepaßt werden, um die Bindungen zu erhalten:

*let f = fun(x) fun(y) x + y*
*let a = ...*
*f(a)*
wird umgewandelt zu:

*let . = fun(.) fun(.) @@1 + @@2*
*let . = ...*
*@@1(@@2)*

Die Substitution von *x* durch *a* in *f(a)* erfordert, daß der Index von *y* angepaßt wird. Die Anzahl der entfernten Definitionen muß subtrahiert (hier eine: x), die Anzahl der neu eingeführten Definitionen (hier zwei: *f* und *a*) addiert werden:

*fun(.) @@2 + @@*$(2-1+2)$

Es ist also nicht sinnvoll, de Bruijn-Indices generell durch Cardelli-Indices zu ersetzten, sondern die Eignung der beiden Indextypen für unterschiedliche Bindungen muß herausgearbeitet werden, was im folgenden Abschnitt geschieht.

[1] Die Verbindung dieser beiden Indextypen wurden meines Wissens zum ersten Mal von Luca Cardelli vorgeschlagen, ohne daß dazu eine schriftliche Referenz existiert [persönliche Kommunikation mit Florian Matthes].

[2] Implementationstechnisch bietet es sich an, de Bruijn- und Cardelli-Indices als negative und positive ganze Zahlen darzustellen, wobei die 0 als *wrongIndex* zur Kennzeichnung von Fehlern genutzt wird.

### 5.2.4 Lokale und globale Bindungen

In diesem Abschnitt werden die Vor- und Nachteile von de Bruijn- und Cardelli-Indices zusammengefaßt. Dazu werden anhand des folgenden Beispiels die Begriffe „lokale“ und „globale“ Bindung eingeführt:

```
let g = ...
let a = fun(l) g + l    (* globale (g) und lokale (l) Bindung *)
...                     (* n weitere Bindungen *)
a
```

Ein Ausdruck $a$ wird definiert und alle referenzierenden Bezeichner innerhalb dieses Ausdrucks werden gebunden. Der referenzierende Bezeichner $g$ ist global gebunden, denn der Bezeichner ist innerhalb des Ausdrucks $a$ frei, wird also außerhalb des Ausdrucks definiert. Dagegen wird der referenzierende Bezeichner $l$ lokal gebunden, denn die Definition des Bezeichners $l$ ist Teil des Ausdrucks $a$.

Beide Bindungsarten (lokal und global) lassen sich durch de Bruijn- und Cardelli-Indices darstellen. Bei einer Substitution des Ausdrucks $a$ in einen neuen Kontext sind die folgenden Manipulationen an den de Bruijn- bzw. Cardelli-Indices in dem Ausdruck nötig, wenn zwischen der Definition des Ausdrucks und der Stelle, an die er substituiert wird, $n$ weitere Bindungen eingeführt werden:

| Indextyp | Bindung | |
|---|---|---|
| | lokal | global |
| de Bruijn | — | $\mathrm{inc}(n+1)$ |
| Cardelli | $\mathrm{inc}(n+1)$ | — |

Daraus ergibt sich die Schlußfolgerung, daß bei einer Verwendung von de Bruijn-Indices für lokale Bindungen und Cardelli-Indices für globale Bindungen eines Ausdrucks keine Indexmanipulationen während einer Substitution des Ausdrucks in einen neuen Kontext nötig sind.

### 5.2.5 Makros im Lambda-Kalkül

Wie um obigen Beispiel gesehen, kann *jeder* Ausdruck in einen anderen Kontext substituiert werden. Die Bindung eines Ausdrucks wird zu einem Zeitpunkt vorgenommen, an dem noch nicht entschieden ist, ob er für eine Substitution herangezogen wird oder nicht. Bei der Bindung des Ausdrucks muß aber schon entschieden werden, welcher Indextyp (de Bruijn oder Cardelli) einzusetzen ist, also muß auch bekannt sein, ob eine Bindung global oder lokal ist. Dies ist im Lambda-Kalkül nicht vorgesehen:

```
let a = ...
let f = fun(x) {fun(y) y + x} (a)
...
let g = {fun(z) ...} (f)
```

Der referenzierende Bezeichner x in dem Teilausdruck *fun(y) y + x* ist zwar global zu dem Teilausdruck, aber lokal zu dem gesamten Ausdruck *f*. Es sind u. a. folgende Substitutionen möglich: *a* für *y* oder *f* für *z*. Bei der ersten Substitution muß *x* global mit einem Cardelli-Index gebunden sein, damit keine Indexmanipulationen nötig sind. Dagegen muß *x* bei der zweiten Substitution lokal mit einem de Bruijn-Index gebunden sein. Dieser Widerspruch resultiert daraus, daß im Lambda-Kalkül die Grenze zwischen lokalen und globalen Bindungen von dem betrachteten Teilausdruck abhängig ist.

Um diese Grenze explizit festzulegen, ist ein neues Konstrukt *macro[]* nötig, das substituierbare Konstrukte beschreibt:

*let x = ...*
*let m = macro[] fun(y) x + y*
wird umgewandelt zu:

*let . = ...*
*let . = macro[] fun(.) @@1 + @1*

Durch das Konstrukt *macro[]* wird festgelegt, daß alle Bindungen innerhalb dieses Konstrukts lokal sind, also durch de Bruijn-Indices ersetzt werden, während freie Bezeichner durch globale Cardelli-Indices ersetzt werden. Im Beispiel ist *x* ein freier (globaler) Bezeichner, während *y* lokal gebunden ist.

Alle Makros werden expandiert bzw. substituiert, *bevor* eine Funktion evaluiert wird. Das bedeutet, daß Makros nicht als Parameter oder Ergebnis einer Funktion auftreten können:

*let f = fun(x) x[]* *(* unzulässig *)*
*let g = fun() macro[] ...* *(* unzulässig *)*

Makros können andere Makros (und nichts anderes) als Parameter erhalten:

*let m = macro[p] x + p[]*
*m[macro[] y + z]*
wird umgewandelt zu:

*x + y + z*

Für die Parameter-Makros gelten dieselben Bindungsregeln wie für Makros.

Lokale Makros innerhalb eines Makros zerstören die einfache Unterscheidung zwischen globalen und lokalen Bindungen:

*let m = macro[] fun(x) {macro[] x}[]*
wird umgewandelt zu:

*let m = macro[] fun(.) {macro[] @@5/@1}[]*
wird umgewandelt zu:

*let m = macro[] fun(.) @@5/@1*
wird umgewandelt zu:

*let m = macro[] fun(.) @1*

Für das lokale Makro ist $x$ ein freier Bezeichner, wird also global mit einem Cardelli-Index gebunden. Für das umgebende Makro hingegen stellt $x$ keinen freien Bezeichner dar, also wird $x$ (nach der Expansion des lokalen Makros) lokal mit einem de Bruijn-Index gebunden.

Im Fall von lokalen Makros existieren also wieder beide Indextypen parallel, so daß Indexmanipulationen oder Indexumwandlungen (von Cardelli- in de Bruijn-Indices) nötig werden:

*let m = macro[p] fun(x) p[macro[] x]*
*... (* n weitere Bindungen *)*
*m[macro[q] fun(y) y + q[]]*
wird umgewandelt zu:

*let m = macro[p] fun(.) p[macro[] @@5/@1]*
*... (* n weitere Bindungen *)*
*m[macro[q] fun(.) @1 + q[]]*
wird umgewandelt zu:

*{macro[p] fun(.) p[macro[] @@(5 + n)/@1]}[macro[q] fun(.) @1 + q[]]*
wird umgewandelt zu:

*fun(.) {macro[q] fun(.) @1 + q[]}[macro[] @@(5 + n)/@1]*
wird umgewandelt zu:

*fun(.) fun(.) @1 + @@(5 + n)/@(1 + 1)*

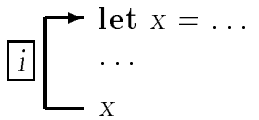
Das Makro $m$ erhält ein Makro $p$ als Parameter, das wiederum einen Parameter $q$ erwartet. Beim Aufruf des Makros $p$ wird ein lokales Makro *macro[] x* als Parameter übergeben. Der in diesem lokalen Makro auftauchende Bezeichner $x$ kann entweder global zu dem lokalen Makro oder lokal zu dem umgebenden Makro $m$ gebunden werden. Der Aufruf des Makros $m$ erfolgt mit dem Makro *macro[q] fun(y) y + q[]* als Parameter, das eine neue Bindung *fun(y)* einführt. Der Index des Bezeichners $x$ muß deshalb entweder beim Substituieren von $m$ (global) oder beim Substituieren des Parameters $q$ (lokal) angepaßt werden.

### 5.2.6 Zusammenfassung

In diesem Abschnitt wurde gezeigt, wie Bindungen bei Substitutionen mit Hilfe von Indexmanipulationen grundsätzlich angepaßt werden können. Mit der Einführung eines Makrokonstrukts werden Substitutionen explizit verfügbar, also umfangreiche Indexmanipulationen notwendig. Durch die Unterscheidung von lokalen und globalen Bindungen und der Darstellung globaler Bindungen durch Cardelli-Indices werden die Indexmanipulationen auf den Fall von lokalen Makros beschränkt. Im folgenden Abschnitt werden die möglichen Bindungsprobleme auf TLExt übertragen und ihre Lösung beschrieben.

## 5.3 Klassifikation von Bindungen in Syntaxerweiterungen

Abbildung 5.1 zeigt schematisch, welche Bindungen in Konstrukten der Basissprache in Syntaxerweiterungen auftreten; Bindungen von Konstrukten der Erweiterungssprache werden hier nicht betrachtet. In der Abbildung symbolisiert jeder Pfeil $\rightarrow$ die Bindung eines Bezeichners, wobei der Pfeil vom referenzierenden zum definierenden Auftreten eines Bezeichners zeigt und mit einer Markierung $\boxed{i}$ gekennzeichnet ist:




Die Punkte ⋮ bzw. ... symbolisieren beliebigen Quelltext. Im folgenden wird erläutert, wie die einzelnen Bindungen $\boxed{i}$ entstehen und bei der Expansion einer Syntaxerweiterung durch die Bindungen $\boxed{i'}$ ersetzt werden.

Folgende Zeitpunkte und Aktionen werden unterschieden:

**Definition einer Syntaxerweiterung:** Im folgenden Beispiel wird eine Syntaxerweiterung mit globalen Referenzen, lokalen definierenden und referenzierenden Bezeichnern sowie Platzhaltern definiert; die Markierungen $\boxed{i}$ verweisen auf die entsprechenden Teile in Abbildung 5.1.

```
grammar newGram
import g1                    (* Definition [1] *)
export
 LET g ::=
   "new" p1 = bndsG          (* Platzhalter-Definition p1 *)
   "and" p2 = bndsG          (* Platzhalter-Definition p2 *)
   "end"
   => BNDS
     let ... = g1            (* Referenz [1] *)
     let l1 = ...            (* Definition [2] *)
     BNDS p1                 (* Platzhalter-Referenz p1 *)
     let l2 = ...            (* Definition [3] *)
     BNDS p2                 (* Platzhalter-Referenz p2 *)
     let ... = l2            (* Referenz [3] *)
     let ... = l1            (* Referenz [2] *)
     let l3 = ...
   END
end
```

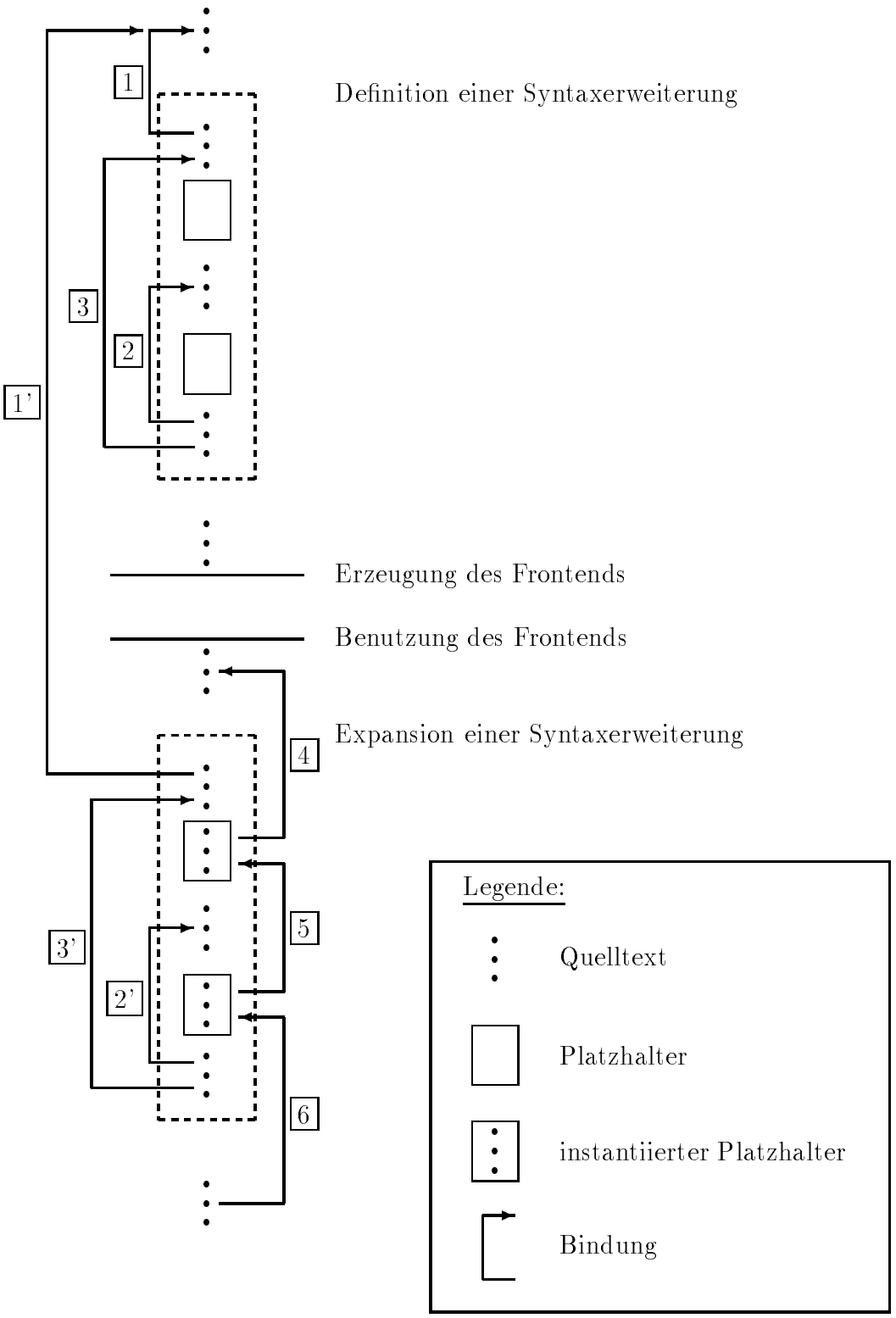



Abbildung 5.1: Schematische Darstellung von Bindungen in Syntaxerweiterungen

Die referenzierenden Bezeichner *g1*, *l1* und *l2* werden gebunden. Die Bindung [1] des Bezeichners *g1* wird *globale Bindung* genannt, weil *g1* einen global sichtbaren Bezeichner repräsentiert. Im Gegensatz dazu sind die Bindungen [2] und [3] der Bezeichner *l1* und *l2* *lokale Bindungen* und binden keinen Bezeichner außerhalb der Syntaxerweiterung oder in der Instantiierung eines Platzhalters.

**Erzeugung des Frontends:** Entsprechend der Definition der Syntaxerweiterung wird ein neues Frontend erzeugt.

**Benutzung des Frontends:** Ein durch eine Syntaxerweiterung erzeugtes Frontend wird zur Bearbeitung eines Quelltextes in der Basissprache benutzt:

**with grammar** *newGram*
*... (* Quelltext der Basissprache *)*

**Expansion einer Syntaxerweiterung:** Eine Syntaxerweiterung wird expandiert und die Platzhalter werden instantiiert; der Einfachheit halber werden die Expansionen als Quelltext dargestellt, auch wenn intern abstrakte Syntaxbäume genutzt werden:

```
let g2 = ...                (* Definition [4] *)
new
  let g3 = ...              (* Definition [5] *)
  let ... = g2              (* Referenz [4] *)
and
  let g4 = ...              (* Definition [6] *)
  let ... = g3              (* Referenz [5] *)
end
let ... = g4                (* Referenz [6] *)
 wird umgewandelt zu:
let g2 = ...                (* Definition [4] *)
> let ... = g1              (* Referenz [1'] *)
> let l1 = ...              (* Definition [2'] *)
>> let g3 = ...             (* Definition [5] *)
>> let ... = g2             (* Referenz [4] *)
> let l2 = ...              (* Definition [3'] *)
>> let g4 = ...             (* Definition [6] *)
>> let ... = g3             (* Referenz [5] *)
> let ... = l2              (* Referenz [3'] *)
> let ... = l1              (* Referenz [2'] *)
> let l3 = ...
let ... = g4                (* Referenz [6] *)
```

> Die globalen und lokalen Bindungen [1], [2] und [3] der Bezeichner *g1*, *l1* und *l2* aus der Definition der Syntaxerweiterung sollen erhalten bleiben. Neue globale Bindungen sind [4], [5] und [6] der Bezeichner *g2*, *g3* und *g4*.

Der Syntaxerweiterungsmechanismus muß gewährleisten, daß diese Bindungen vorgenommen werden und erhalten bleiben. Die dabei auftretenden Probleme und ihre Lösung werden im folgenden beschrieben. Die formale Beschreibung der Bindungsphase findet sich in Anhang B.3.

In dem Beispiel wird der Fall einer *lokalen* Bindung zwischen Platzhalter-Instantiierungen nicht gezeigt. Eine lokale Bindung könnte durch folgende Syntaxerweiterung entstehen:

"new" *p1* = *bndsG* "and" *p2* = *bndsG* "end"
=> **BNDS**
  **begin**
    **BNDS** *p1*
    ...
    **BNDS** *p2*
  **end**
**END**

Die folgende Anwendung der Syntaxerweiterung zeigt eine lokale Bindung:

**new**
  **let** *x* = ...
**and**
  **let** ... = *x*
**end**

wird umgewandelt zu:

> **begin**
>> **let** *x* = ...
> ...
>> **let** ... = *x*
> **end**

Durch Instantiierung des Platzhalters *p1* wird ein Bezeichner *x* definiert, der im durch Instantiierung des Platzhalters *p2* entstehenden Quelltext referenziert wird, aber nicht über die Blockgrenzen (**begin** ...**end**) hinaus sichtbar ist, also *lokal* zu diesem Block existiert. Diese Bindung entspricht der Bindung [5] und wird ebenso behandelt.

## 5.4 Globale Bindungen in der Definition

Fall [1] ist die globale Bindung des Bezeichners *g1* zum Zeitpunkt der Definition der Syntaxerweiterung. Während der Expansion der Syntaxerweiterung ist der Bezeichner *g1* evtl. nicht mehr sichtbar oder neu gebunden:

**let** *g1* = ...
**new** ... **and** ... **end**
wird naiv umgewandelt zu:

**let** *g1* = ...
> **let** ... = *g1* (* *Referenz* 1' *)

Die Lösung dieses Problems erfordert zwei Maßnahmen:

1. Bezeichner, die zum Definitionszeitpunkt einer Syntaxerweiterung global sichtbar sind, müssen auch zum Expansionszeitpunkt definiert sein. Dabei ist zu beachten, daß diese Bezeichner keine anderen Bezeichner verdecken dürfen und keine unerwünschten Bindungen erzeugen sollten, da der Benutzer der Syntaxerweiterung (normalerweise) nicht wissen soll, welche globalen Bezeichner während der Definition der Syntaxerweiterung sichtbar waren. Im Beispiel sollte *g1* aus *newGram* kein *g1* aus dem Quelltext der Basissprache verdecken und keine außerhalb einer Expansion auftretende Referenz auf *g1* binden.

2. Die in einer Expansion einer Syntaxerweiterung auftretenden globalen Referenzen sind an die entsprechenden definierenden Bezeichner aus der Definition der Syntaxerweiterung zu binden.

Im Kontext von de Bruijn- und Cardelli-Indices werden die globalen Bindungen durch Cardelli-Indices dargestellt. Bei der Verwendung eines Frontends ist darauf zu achten, daß entweder der gleiche Kontext wie bei der Definition der Syntaxerweiterung existiert oder die Indices (einmal) entsprechend angepaßt werden. Da die Indices global gültig sind, müssen sie bei der Expansion einer Syntaxerweiterung nicht mehr angepaßt werden.

Die einmalige Anpassung der Indices entspricht den Anpassungen, die beim Importieren von Modulen benötigt werden: Zwei Moduldefinitionen, die in verschiedenen Kontexten übersetzt wurden und in einen neuen Kontext importiert werden, müssen aufeinander abgestimmt werden. Also wird hier bei einer modularen Sprache kein neues Konzept eingeführt, sondern auf ein vorhandenes Konzept im neuen Kontext von Syntaxerweiterungen zurückgegriffen.

## 5.5 Lokale Bindungen in der Definition

Die Fälle 2 und 3 stellen lokale Bindungen der Bezeichner *l1* und *l2* zum Zeitpunkt der Definition der Syntaxerweiterung dar. Während der Expansion können durch Instantiierung der Platzhalter neue Definitionen eingeführt werden, die diese Bezeichner binden:

```
new
  let l1 = ...
and
  let l2 = ...
end
```

wird naiv umgewandelt zu:

```
> let ... = g1          (* Referenz [1] *)
> let l1 = ...          (* Definition [2] *)
>> let l1 = ...         (* Redefinition *)
> let l2 = ...          (* Definition [3] *)
>> let l2 = ...         (* Redefinition *)
> let ... = l2          (* Referenz [3] *)
> let ... = l1          (* Referenz [2] *)
```

Während der Definition der Syntaxerweiterung werden die Bezeichner *l1* und *l2* gebunden und durch (lokale) de Bruijn-Indices ersetzt. Der um de Bruijn-Indices erweiterte Programmtext sieht folgendermaßen aus:

```
let ... = g1          (* Referenz [1] *)
let l1 = ...          (* Definition [2] *)
BNDS p1               (* Platzhalter-Referenz p1 *)
let l2 = ...          (* Definition [3] *)
BNDS p2               (* Platzhalter-Referenz p2 *)
let ... = l2@1        (* Referenz [3] *)
let ... = l1@3        (* Referenz [2] *)
```

Durch die Instantiierung der Platzhalter *p1* und *p2* können neue Definitionen eingefügt werden, die sich auf die de Bruijn-Indices der Referenzen von *l1* und *l2* auswirken: Jeder de Bruijn-Index eines referenzierenden Bezeichners muß um die Anzahl der seit der Definition des Bezeichners durch Instantiierung von Platzhaltern neu eingeführten Definitionen inkrementiert werden:

```
> let ... = g1          (* Referenz [1] *)
> let l1 = ...          (* Definition [2] *)
>> let l1 = ...         (* Redefinition *)
> let l2 = ...          (* Definition [3] *)
>> let l2 = ...         (* Redefinition *)
> let ... = l2@(1 + 1)  (* Referenz [3] *)
> let ... = l1@(3 + 2)  (* Referenz [2] *)
```

Zwischen der Definition und der Referenz von *l1* sind zwei neue Definitionen (*l1* und *l2*) durch Instantiierung der Platzhalter *p1* und *p2* erfolgt. Zwischen Definition und Referenz von *l2* ist dagegen nur eine neue Definition (*l2*) durch Instantiierung des Platzhalters *p2* erfolgt.

In der vorliegenden Implementation erfolgt die Manipulation der de Bruijn-Indices während der Bindungsphase (*scoping*), erfordert also eine Änderung dieser Phase (vgl. Anhang B.3). Alle innerhalb einer expandierten Syntaxerweiterung durch Instantiierung von Platzhaltern eingeführten Bindungen werden über einen Zähler $count_L$ gezählt. Bei der Definition eines lokalen Bezeichners einer Syntaxerweiterung wird der aktuelle Zählerstand in der Umgebung $E_L$ vermerkt. Diese Umgebung wird nicht durch Definitionen innerhalb der Instantiierung eines Platzhalters beeinflußt. Die Referenz eines lokalen Bezeichners einer Syntaxerweiterung bewirkt einen Zugriff auf diese Umgebung über den de Bruijn-Index der Referenz. Aus der Differenz des aktuellen Zählerstandes von $count_L$ und des Zählerstandes zum Zeitpunkt der Definition des Bezeichners kann die nötige Korrektur des de Bruijn-Index berechnet werden: $index \Leftarrow index + (count_L - E_L[index])$.

## 5.6 Globale Bindungen in Instantiierungen von Platzhaltern

Die Fälle 4, 5 und 6 beschreiben Bindungen, die durch die Instantiierung von Platzhaltern bei der Expansion einer Syntaxerweiterung entstehen. Die drei möglichen Fälle sind:

**4** Binden einer durch eine Instantiierung entstehenden Referenz eines Bezeichners an eine Definition außerhalb der Expansion.

**5** Definition eines Bezeichners und Binden einer entsprechenden Referenz zwischen Instantiierungen von Platzhaltern.

**6** Binden einer außerhalb der Expansion befindlichen Referenz an eine durch eine Instantiierung eingeführte Definition.

Problematisch werden diese Fälle durch lokale Bezeichner, die durch die Expansion der Syntaxerweiterung definiert werden, im Beispiel *l1* und *l2*, und globale Bindungen beeinflussen:

```
let l1 = ...        (* Definition 4 *)
new
  let ... = l1      (* Referenz 4 *)
  let l2 = ...      (* Definition 5 *)
and
  let ... = l2      (* Referenz 5 *)
  let l3 = ...      (* Definition 6 *)
end
let ... = l3        (* Referenz 6 *)
```

wird naiv umgewandelt zu:

```
let l1 = ...                    (* Definition [4] *)
> let ... = g1
> let l1 = ...                  (* Redefinition *)
>> let ... = l1                 (* Referenz [4] *)
>> let l2 = ...                 (* Definition [5] *)
> let l2 = ...                  (* Redefinition *)
>> let ... = l2                 (* Referenz [5] *)
>> let l3 = ...                 (* Definition [6] *)
> let ... = l2
> let ... = l1
> let l3 = ...                  (* Redefinition *)
let ... = l3                    (* Referenz [6] *)
```

Die lokalen Redefinitionen von *l1*, *l2* und *l3* sollen nicht bindend auf referenzierende Bezeichner in der Umgebung der Expansion und in Instantiierungen von Platzhaltern wirken. Die in den Instantiierungen von Platzhaltern vorgenommenen Bindungen sollen auch außerhalb der Expansion bzw. zwischen verschiedenen Instantiierungen sichtbar sein.

Dies wird erreicht, indem die lokalen Redefinitionen der Bezeichner *l1*, *l2* und *l3* als anonyme Bezeichner ohne bindende Wirkung in den Kontext eingetragen werden. Dies ist möglich, weil die durch sie gebundenen lokalen Referenzen schon während der Definition der Syntaxerweiterung gebunden werden.

## 5.7 Implementation

Der geschilderte Ansatz stellt Anforderungen an die Basissprache und erfordert Erweiterungen des Compilers, die in diesem Abschnitt diskutiert werden.

Die Bindung von lokalen Bezeichnern in Syntaxerweiterungen wird während der Definition der Syntaxerweiterung vorgenommen. Die Bindung von Bezeichnern findet normalerweise zusammen mit der Typprüfung in der Typprüfungsphase des Übersetzungsvorgangs statt. Während der Definition der Syntaxerweiterung stehen aber nur sehr unvollständige Typinformationen zur Verfügung, da die Platzhalter (in der gegenwärtigen Version) nur Sorteninformationen, aber keine Typinformationen tragen. Somit ist keine vollständige Typprüfung möglich, so daß eine eigenständige Bindungsphase eingeführt werden muß. Diese Bindungsphase kann nicht auf Typinformationen zurückgreifen, was sich auf die Bindungsregeln auswirkt, so daß z. B. die Überladung von Bezeichnern nicht durch Typinformationen aufgelöst werden kann. Außerdem können selektierende Bezeichner nicht überprüft werden, da sie vom Typ des Objekts abhängen, auf das sie sich beziehen. Somit sind auch Anweisungen, die typabhängig neue Bezeichner definieren, z. B. **open** in TL, nicht erlaubt.

Die Expansion der Syntaxerweiterung erzeugt einen abstrakten Syntaxbaum mit gebundenen und ungebundenen Bezeichnern, wobei die gebundenen Bezeichner durch Indices dargestellt werden. Lokale definierende Bezeichner werden durch anonyme Bezeichner ersetzt. In der Typprüfungsphase (der Basissprache) müssen die durch Instantiierung von Platzhaltern eingeführten Bindungen gezählt und die Indices der lokalen Bindungen der Syntaxerweiterung angepaßt werden. Also ist eine Erweiterung der Typprüfungsphase nötig. Der abstrakte Syntaxbaum muß dementsprechend erweitert werden um

- gebundene referenzierende Bezeichner (Indices) in Expansionen von Syntaxerweiterungen,
- Kennzeichnung von (lokalen) definierenden Bezeichnern in Expansionen von Syntaxerweiterungen,
- Kennzeichnung definierender Bezeichner in Instantiierungen von Platzhaltern.

# Kapitel 6

# Zusammenfassung und Ausblick

## 6.1 Zusammenfassung

Ausgangspunkt dieser Arbeit bildet das Problem der Integration von Programmiersprachen und Datenbanksystemen. Aus den Stärken und Schwächen der drei Ansätze zur Lösung dieses Problems (Einbettung spezieller Datenbanksprachen, Erweiterung allgemeiner Programmiersprachen zu Datenbankprogrammiersprachen bzw. Anbindung von Bibliotheken) wird die Forderung nach syntaktischer Erweiterbarkeit als Hilfsmittel zur skalierbaren und programmierergerechten Anbindung von Datenbanksystemen an Programmiersprachen abgeleitet.

Syntaktisch erweiterbare Programmiersprachen sind seit ca. 30 Jahren bekannt, haben sich aber nicht durchgesetzt und sind nur noch selten Gegenstand der Forschung. Aus den Problemen bestehender Ansätze zur syntaktischen Erweiterung ergibt sich die Frage, ob mit bekannten und wohlverstandenen Technologien ein einfacher Mechanismus zur syntaktischen Erweiterung realisiert werden kann, der keine unerwarteten Resultate produziert und die Effizienz des Übersetzungsvorgangs nicht einschränkt. Dabei wird eine Basissprache vorausgesetzt, deren Regeln zur Benennung, Bindung und Typisierung invariant gegenüber Syntaxerweiterungen sein sollen.

Aufbauend auf ein Phasenmodell von Compilern werden verschiedene Möglichkeiten zur syntaktischen Erweiterung diskutiert. Gesichtspunkte dieser Diskussion sind einerseits die Flexibilität der Ansätze und andererseits die Einschränkung der Fehlermöglichkeiten, insbesondere die Zusicherung, daß eine Syntaxerweiterung terminiert. Als geeigneter Ansatzpunkt für die syntaktische Erweiterung erweisen sich die syntaktische Analyse, in der neue syntaktische Konstrukte erkannt werden, und die Abstraktionsphase, in der eine Menge von grundlegenden Syntaxbäumen erzeugt wird. Zur Beschreibung von Syntaxerweiterungen bieten sich attributierte kontextfreie Grammatiken an, die von gängigen Parser-Generatoren verarbeitet werden, so daß die Effizienz der Übersetzung der von Compilern entspricht, die mit solchen Generatoren erzeugt werden.

Mit Hilfe einer minimalen Basissprache TLMin und einer Sprache zur Beschreibung von Syntaxerweiterungen TLExt werden verschiedene Einsatzgebiete von Syntaxerweiterungen aufgezeigt: Einerseits können gängige (redundante) Standardkonstrukte von Programmiersprachen, z. B. verschiedene Schleifenkonstrukte, emuliert und Normalisierungen von komplexen in einfache Konstrukte vorgenommen werden. Andererseits lassen sich spezialisierte Konstrukte definieren, die sich auf vorhandene Bibliotheken stützen und deren Benutzung erleichtern. Dies wird am Beispiel von Datenbankkonstrukten gezeigt.

Außerdem zeigt sich, daß durch Syntaxerweiterungen eine Abstraktion erreicht werden kann, die durch Funktionen nicht oder nur schwer erreichbar ist: Es können Generatoren mit beliebigen Signaturen definiert werden, während funktionale Generatoren auf eine feste Menge von Signaturen eingeschränkt sind.

Ein besonderes Problem von Syntaxerweiterungen, das noch Gegenstand der Forschung ist, sind Bindungsprobleme. Dieses Problem wird auf das Substitutionsproblem im Lambda-Kalkül zurückgeführt. Durch den Einsatz der aus diesem Problembereich bekannten (Manipulation von) de Bruijn-Indices und eine Erweiterung um Cardelli-Indices wird eine effiziente Lösung des Bindungsproblems beschrieben.

Die in dieser Arbeit entwickelten Algorithmen zur statischen Prüfung von Syntaxerweiterungen und zur Lösung des Bindungsproblems sind formalisiert und somit einer Diskussion und Weiterentwicklung zugänglich.

## 6.2 Ausblick

In dieser Arbeit konnten nicht alle Aspekte erweiterbarer Programmiersprachen erschöpfend behandelt werden. In diesem Abschnitt werden deshalb offengebliebene Fragen gestellt und einige Hinweise für weitere Forschungen gegeben.

Im Bereich der *Basissprachen* ist zu klären, welche Konzepte und Konstrukte eine minimale Basissprache enthalten sollte und welche Konstrukte sich durch Syntaxerweiterungen nachbilden lassen. Einige Eigenschaften von Compilern, die bisher vor dem Programmierer verborgen werden, sollten explizit zugänglich sein, um Syntaxerweiterungen zu unterstützen, z. B. die Typinferenz (vgl. Seite 44 sowie die Abschnitte 4.2.3 und 4.3.2).

Weitreichende Auswirkungen auf die Behandlung von Syntaxerweiterungen haben das Typsystem und die Bindungsregeln der Basissprache. Durch polymorphe Typsysteme [Atkinson, Bunemann 87; Matthes, Schmidt 91] ist eine mächtige Grundlage gegeben, die sehr gut durch Syntaxerweiterungen ergänzt wird, denn die verschachtelten Aufrufe generischer Bibliotheken sind semantisch nicht leicht zu durchschauen und können durch Syntaxerweiterungen vereinfacht werden.

Im Kontext von Syntaxerweiterungen ergeben sich Bindungsprobleme, wie die Diskussion in Kapitel 5 zeigt. Durch neue Bindungsregeln, wie das Überladen von Bezeichnern, entstehen weitere Probleme, so daß dieser Schritt sorgfältig abgewogen werden muß. Außerdem

existieren in vielen Programmiersprachen Anweisungen wie **open** und **Repeat** in TL, die in Wechselwirkung mit dem Typsystem Bindungen beeinflussen und somit bei (untypisierten) Syntaxerweiterungen nicht tragbar sind. Evtl. läßt sich dieses Problem mit einer Erweiterung des Sortensystems lösen (vgl. Abschnitt 3.2.3.6).

Auf der Ebene des syntaktischen *Erweiterungsmechanismus'* sind Grenzen und mögliche Ergänzungen zu diskutieren. Falls für Transformationen ein sinnvolles Einsatzgebiet aufgezeigt werden sollte, muß der Ansatzpunkt für Syntaxerweiterungen im Compilermodell überdacht werden; Kapitel 2 enthält die dafür nötigen Grundlagen. In diesem Fall ist eine nähere Beschäftigung mit dem Gebiet des *term rewriting* nötig, um nicht-terminierende Syntaxerweiterungen auszuschließen.

Eine weitere Einschränkung der Mächtigkeit des Syntaxerweiterungsmechanismus erscheint dagegen nicht sinnvoll, denn ohne rekursive Produktionen sowie die Unterstützung mehrerer vererbter und abgeleiteter Attribute wären einige der in Kapitel 4 gezeigten Anwendungen nicht realisierbar (vgl. Abschnitte 4.2.2 und 4.3.2). Da die verfügbaren Parser-Generatoren diese Eigenschaften unterstützen, liegt darin auch kein Problem.

Es ist festzulegen, welche Klasse(n) von Parser-Generatoren in welcher Form unterstützt werden sollten, z. B. durch Ergänzung der Erweiterungssprache für LL-Parser-Generatoren wie in [Cardelli 91]. Außerdem müssen Anforderungen an Parser-Generatoren formuliert werden, denn die bekannten Generatoren wie YACC mit textuellen Ein- und Ausgaben sowie den Einschränkungen bzgl. der Basissprache sind unbrauchbar. Stattdessen sollten generische Bibliotheken eingesetzt werden. In diesem Zusammenhang ist auch zu klären, ob inkrementelle Parser-Generatoren im praktischen Einsatz Vorteile gegenüber den klassischen Parser-Generatoren zeigen.

Durch die Integration von Scanner-Generatoren können weitere Freiheitsgrade gewonnen werden, z. B. um neue Datentypen zu definieren und zu verarbeiten. Allerdings ist zu beachten, daß einem Scanner normalerweise weitere Aufgaben zufallen, die sich nicht einfach beschreiben lassen, z. B. die Entfernung von Kommentaren.

Zur Unterstützung von Syntaxerweiterungen können Meta-Operationen (vgl. Abschnitt 2.5.5) bzw. reflektive Funktionen (vgl. Abschnitt 2.5.7) eingesetzt werden, z. B. die in Abschnitt 4.2.3 angesprochene Meta-Operation *typeOfVal*. Welche Operationen sinnvoll und nützlich sind, wird erst die Praxis zeigen. Evtl. läßt sich hier eine Verbindung zu den oben angesprochenen problematischen Anweisungen wie **open** und **Repeat** aus TL finden, denn diese Anweisungen beeinflussen auch nicht den Kontrollfluß, sondern stellen als zur Übersetzungszeit auszuführende Anweisungen eine Variante von Meta-Operationen dar.

Ein besonderes und hier weitgehend ausgeklammertes Problem stellen Fehlermeldungen dar [Cheatham 66; Sakharov 92]. Bei der *Definition* einer Syntaxerweiterung auftretende Fehler, z. B. unerlaubte Linksrekursionen, müssen dem Benutzer mit möglichst genauen Hinweisen auf die Ursache gemeldet werden. Dies wird durch die explizite Namensgebung von Produktionen und Platzhaltern in der Erweiterungssprache unterstützt, hängt aber

von dem verwendeten Parser-Generator ab. Während bei der *Expansion* einer Syntaxerweiterung keine Fehler auftreten können, sind (semantische) Fehler im expandierten Syntaxbaum möglich und wahrscheinlich (vgl. Abschnitt 3.2.3.6). Ein einfaches *unparsing* des fehlerhaften Syntaxbaums reicht als Fehlermeldung nicht aus, sondern es müssen explizite Informationen über die Herkunft des Syntaxbaums, z. B. aus welcher Syntaxerweiterung er stammt, mitgeführt werden. Der in dieser Arbeit eingesetzte TL-Compiler verfügt über eine solche Möglichkeit der Rückverfolgung, auf die hier nicht näher eingegangen werden kann (vgl. [Matthes 93]).

Auf der Ebene des *Sprachdesigns* ergeben sich durch syntaktisch erweiterbare Programmiersprachen besondere Probleme, wie [Gries 76] richtig bemerkt (wenn in dieser Arbeit auch seiner Schlußfolgerung, syntaktisch erweiterbare Sprachen abzulehnen, nicht gefolgt wird). In Kapitel 4 werden neben den allgemein nutzbaren Standardkonstrukten bzw. Normalisierungen spezielle Datenbankkonstrukte vorgestellt, deren Einsatz nur für bestimmte (datenintensive) Anwendungen, und selbst da nur für Teile der Anwendung, sinnvoll ist. Für andere (Teile von) Anwendungen werden dagegen Konstrukte für die grafische Darstellung o. ä. benötigt. Hier besteht die Gefahr, wie bei PL/1 oder den 4GL-Sprachen eine monoton wachsende und damit unübersichtliche Sprache zu erhalten. Die Alternative sind Sprachversionen für spezialisierte Anwendungen, wie dies sich z. B. bei der Trennung der Benutzeroberfläche von der Anwendung mit der Entwicklung von "*interface description languages*", der Entwicklung spezialisierter Datenbanksprachen oder Beschreibungen von externen Datenformaten [Abiteboul et al. 93; Linnemann 93] andeutet.

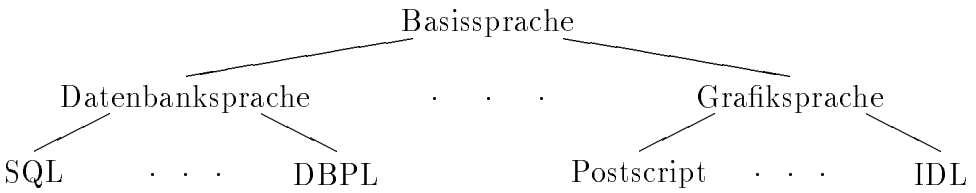



Abbildung 6.1: Baum von Sprachversionen

Resultat dieser Tendenz ist ein Baum von Sprachversionen, wie in Abbildung 6.1 angedeutet. Gegenwärtig sieht der hier vorgestellte Mechanismus zur Syntaxerweiterung nicht vor, zwei (oder mehr) Sprachversionen wieder zu einer Sprache zusammenzuführen, um z. B. in einem Modul Datenbankkonstrukte und Grafikkonstrukte gemeinsam zu benutzen. Die Zusammenführung der Sprachversionen könnte Probleme bereiten, wenn z. B. ein syntaktisches Konstrukt von beiden Sprachversionen mit unterschiedlicher Semantik belegt wird oder ein Teil der Basissprache durch eine Sprachversion nicht erweitert, sondern komplett ersetzt wird. Deshalb sollte eine Disziplin der Trennung von unterschiedlichen Teilen der Anwendung, wie sie durch Module schon ansatzweise geleistet wird, entwickelt werden.

æ

# Anhang A

# Abstrakte und konkrete Syntax von TLMin und TLExt

## A.1 Konkrete Syntax

### A.1.1 Notation

Die Notationen der konkreten Syntax und der größte Teil der Definition von TLMin sind [Matthes 93] entnommen.

Die folgenden Notationen werden zur Definition der lexikalischen und syntaktischen Elemente benutzt, wobei *Id* ein Nonterminal (eine Meta-Variable) bezeichnet und *A* und *B* für beliebige syntaktische Konstrukte stehen:

| | |
|---|---|
| $Id_1, \ldots, Id_n ::= A$ | die Nonterminale $Id_i$ sind als $A$ definiert ($n \geq 1$) |
| *Id* | ein Nonterminal |
| **if** | ein Terminal |
| "x" | das Zeichen *x* ("" ist die leere Zeichenkette, """ ein Anführungszeichen) |
| ( A ) | = *A* |
| *A B* | = *A* gefolgt von *B* (bindet am stärksten) |
| *A* \| *B* | = *A* oder *B* |
| [ A ] | = ( "" \| A ) |
| { A } | = ( "" \| A { A } ) |

### A.1.2 Lexikalische Analyse und Symbole

Dem Compilermodell aus Kapitel 2.1 folgend, wird der Quelltext als Zeichenstrom betrachtet. Er wird durch die lexikalische Analyse in einen Symbolstrom zerlegt, der aus Symbolen

der folgenden Kategorien besteht: *int, real, longreal, char, string, identifier, infix, colonInfix, delimiter.*

Formatierende Zeichen (*whitespaces*) trennen Symbole. Folgende Zeichen werden als formatierende Zeichen angesehen: Leerzeichen, horizontaler und vertikaler Tabulator, Zeilenvorschub, Zeilenrücklauf. Kommentare können aus beliebigen Zeichen bestehen, die durch *(* *)* begrenzt werden. Kommentare können geschachtelt werden. Beim Erkennen eines Symbols werden führende formatierende Zeichen überlesen und die längste Zeichenfolge erkannt, die ein Symbol ergibt. Deshalb sind formatierende Zeichen nur zwischen zwei aufeinanderfolgenden Bezeichnern oder Infixsymbolen nötig.

Die folgenden Symbole werden vom Scanner erkannt:

*int* ::= ["~"] *digit* { *digit* }
*real* ::= *int* "." *digit* { *digit* } |
*int* [ "." *digit* { *digit* } ] "E" *int*
*longreal* ::= *int* [ "." *digit* { *digit* } ] "D" *int*
*char* ::= "'" ( *digit* | *alpha* | *special* | *escape* | *delimiter* | *reserved* ) "'"
*string* ::= "" { *digit* | *alpha* | *special* | *escape* | *delimiter* | *reserved* } ""
*infix* ::= *special* { *special* }
*colonInfix* ::= ":" { *special* }
*identifier* ::= *alpha* { *digit* | *alpha* }
*delimiter* ::= "(" | ")" | "{" | "}" | "[" | "]" | "." | "," | ";"
*digit* ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
*alpha* ::= "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"
*reserved* ::= "~" | " "
*special* ::= "@" | "#" | "$" | "%" | "&" | "*" | "_" | "+" | "=" |
"-" | "|" | "\" | "`" | ":" | "<" | ">" | "/" | "^" |
"?" | "!"
*escape* ::= "\" ("n" | "t" | "r" | "f" | "\" | "'" | "" | *digit digit digit*)

Die folgenden Spezialzeichen in Zeichen oder Zeichenketten werden vom Scanner interpretiert:

| | |
|---|---|
| \n | Zeilenvorschub |
| \t | Tabulator |
| \r | Zeilenrücklauf |
| \f | Seitenvorschub |
| \' | ' |
| \" | " |
| \\ | \ |
| \nnn | Zeichen mit Ordinalwert *nnn* |
| | drei Dezimalziffern, im Bereich [0...255] |
| \f ... f\ | mehrere Formatierzeichen, *f* wird überlesen |

Mit der letzten Regel können über eine Zeile hinausgehende Zeichenketten konstruiert werden.

### A.1.3 TLMin

Platzhalter der Form **IDE/VAL/TYP/BND/BNDS/SIG/SIGS** *Placeholder* sind nur in semantischen Interpretationen von TLExt-Konstrukten erlaubt, werden aber der Einfachheit halber in die Beschreibung der Basissprache aufgenommen.

#### A.1.3.1 Reservierte Schlüsselworte

Dies ist die (initiale) Menge von Schlüsselworten in TLMin; durch Syntaxerweiterungen kann diese Menge geändert werden:

**array begin else end exit fun if let letvar loop ok then tuple**
**Fun Let Ok Oper Tuple Var**
**BND BNDS SIG SIGS VAL TYP IDE**
= <: :

#### A.1.3.2 Bindungen

*Bindings*::=
    { *TypeBinding* | *ValueBinding* | *MutableBinding* |
    **BND** *Placeholder* | **BNDS** *Placeholder* }
*TypeBinding*::=
    **Let** *TypeIde* "=" *Type*
*ValueBinding*::=
    **let** *ValueIde* "=" *Value*
*MutableBinding*::=
    **letvar** *ValueIde* "=" *Value*

#### A.1.3.3 Werte

*Value*::=
    $Value_1$ { "(" *Bindings* ")" | "." *LabelIde* | "[" *Value* "]" }
$Value_1$::=
    "{" *Value* "}" |
    *ValueIde* |
    **ok** |
    *int* | *char* | *string* | *real* | *longreal* |
    **fun** "(" *Signatures* ")" *Value* |
    **tuple** *Bindings* **end** |
    **array** *Bindings* **end** |
    **begin** *Bindings* **end** |

**if** *Value* **then** *Bindings* **else** *Bindings* **end** |
**loop** *Bindings* **end** |
**exit** |
**VAL** *Placeholder*

#### A.1.3.4 Signaturen

*Signatures::=*
{ *TypeSignature* | *ValueSignature* | **SIG** *Placeholder* | **SIGS** *Placeholder* }
*TypeSignature::=*
":" *TypeIde* "<:" *Type*
*ValueSignature::=*
*ValueIde* ":" *Type*

#### A.1.3.5 Typen

*Type::=*
$Type_1$ { "(" *Bindings* ")" }
$Type_1$*::=*
"{" *Type* "}" |
{ *ValueIde* "." } *TypeIde* |
**Ok** |
**Fun** "(" *Signatures* ")" ":" *Type* |
**Tuple** *Signatures* **end** |
**Oper** "(" *Signatures* ")" *Type* |
**Var** "(" *Type* ")" |
**TYP** *Placeholder*

#### A.1.3.6 Bezeichner

*Ide, ValueIde, TypeIde, LabelIde, Placeholder::=*
*identifier* | *infix* | *colonInfix* | **IDE** *Placeholder*

### A.1.4 TLExt

#### A.1.4.1 Reservierte Schlüsselworte

Dies ist die Menge von reservierten Schlüsselworten in TLExt:

**end grammar**
**ASSIGN BNDS COPY END LET SIGS TYP VAL**
= => | ::=

#### A.1.4.2 Syntaxerweiterungen

*SyntaxExtension::=*
    **grammar** { *Grammar* } **end**
*Grammar::=*
    ( *SimpleGrammar* | *AssignGrammar* )
*SimpleGrammar::=*
    **LET** *GramIde* [ "(" { *SortSig* } ")" ] "::=" *Productions*
*AssignGrammar::=*
    **ASSIGN** *GramIde* [ "(" { *SortSig* } ")" ] "::=" *Productions*
*Productions::=*
    *Production* { "|" *Production* }
*Production ::=*
    **COPY** *GramRef* | { *Pattern* } "=>" *Base* { *Base* }
*Pattern::=*
    ( *Keyword* | *PlaceholderDefinition* )
*Keyword::=*
    *String*
*PlaceholderDefinition::=*
    *Placeholder* { "," *Placeholder* } "=" [ **COPY** ] *GramRef*
*GramRef::=*
    *GramIde* [ "(" { *Placeholder* } ")" ]
*Base::=*
    **VAL** *Value* **END** | **TYP** *Type* **END** |
    **SIGS** *Signatures* **END** | **BNDS** *Bindings* **END**
*SortSig::=*
    *Placeholder* "::" *Sort*
*Sort::=*
    **IDE** | **VAL** | **TYP** | **SIGS** | **BNDS**
*GramIde, Placeholder::=*
    *identifier*

## A.2 Abstrakte Syntax

### A.2.1 Notation

Die Notationen entsprechen weitgehend denen aus [Matthes 93], die wiederum auf [Plotkin 81] aufbauen.

In der Definition der abstrakten Syntax werden Basismengen und abgeleitete Mengen unterschieden. In den Basismengen finden sich die Terminale der konkreten Syntax wieder,

| | | | |
|---|---|---|---|
| $Def$ | ::= | **Placeholder**$(ide)$ | Platzhalter für eine Definition |
| | \| | **Exp**$(Def)$ | expandierter definierender Bezeichner |
| | \| | **Def**$(ide)$ | definierender Bezeichner |
| $Label$ | ::= | **Placeholder**$(ide)$ | Platzhalter für eine Selektion |
| | \| | **Exp**$(Label)$ | expandierter selektierender Bezeichner |
| | \| | **Label**$(ide)$ | selektierender Bezeichner |
| $Ref$ | ::= | **Placeholder**$(ide)$ | Platzhalter für eine Referenz |
| | \| | **Exp**$(Ref)$ | expandierter referenzierender Bezeichner |
| | \| | **Ref**$(ide)$ | referenzierender Bezeichner |
| | \| | $index$ | gebundener referenzierender Bezeichner |

Tabelle A.1: Abstrakte Syntax für Bezeichner

soweit sie semantische Bedeutung besitzen. Aus den Nonterminalen der konkreten Syntax ergeben sich die abgeleiteten Mengen.

Im Gegensatz zu [Plotkin 81; Matthes 93] erhalten Elemente aus den Mengen keine ausgezeichneten Namen, sondern werden mit dem Namen der Menge, deren Element sie sind, bezeichnet. Diese Namenswahl ist eindeutig, denn die Mengen tauchen nicht als Bestandteil der abstrakten Syntax auf. Verschiedene Elemente einer Menge werden durch Qualifizierung $(Val, Val', Val'')$ unterschieden. Mit dieser Notation wird dem Leser die Zuordnung eines Elements zu seiner Menge erleichtert. Elemente einer Basismenge beginnen mit einem Kleinbuchstaben $(ide, int)$, Elemente einer abgeleiteten Menge mit einem Großbuchstaben $(Def, Typ$ usw.).

### A.2.2 TLMin

In Tabelle A.2 wird die abstrakte Syntax der Basissprache TLMin definiert. Die Bezeichner in ihren verschiedenen Rollen werden in Tabelle A.1 gezeigt. In den Tabellen werden die folgenden syntaktischen Objekte benutzt:

| Basismengen: | |
|---|---|
| $int, real \ldots$ | ein Literal |
| $ide$ | ein Bezeichnername |
| $index$ | ein lokaler oder globaler Index |

| | | | |
|---|---|---|---|
| $Typ$ | ::= | **Placeholder**$(ide)$ | Platzhalter für einen Typ |
| | \| | **Exp**$(Typ)$ | expandierter Typ |
| | \| | **Nok** | Subtyp aller Typen |
| | \| | **Ok** | Supertyp aller Typen |
| | \| | **Bool** \| **Int** \| ... | Basistypen |
| | \| | $Ref$ | Typbezeichnerreferenz |
| | \| | $Val.Label$ | Selektion eines Typbezeichners |
| | \| | **Fun**$(Sigs) : Typ$ | Funktionstyp |
| | \| | **Arr**$(Typ)$ | Feldtyp |
| | \| | **Tup**$(Sigs)$ | Tupeltyp |
| | \| | **Var**$(Typ)$ | Typ veränderlicher Bindungen |
| | \| | **Oper**$(Sigs)Typ$ | Typoperatorabstraktion |
| | \| | $Typ(Bnds)$ | Typoperatorapplikation |
| $Val$ | ::= | **Placeholder**$(ide)$ | Platzhalter für einen Wert |
| | \| | **Exp**$(Val)$ | expandierter Wert |
| | \| | **ok** | der kanonische Wert des Typs **Ok** |
| | \| | $int\|real\ldots$ | Literal |
| | \| | $Ref$ | Wertbezeichner-Referenz |
| | \| | $Val.Label$ | Selektion eines Wertbezeichners |
| | \| | **fun**$(Sigs)Val$ | Funktionskonstruktion |
| | \| | $Val(Bnds)$ | Funktionsapplikation |
| | \| | **arr**$(Bnds)$ | Feldkonstruktion |
| | \| | $Val[Val]$ | Feldelementselektion |
| | \| | **tup**$(Bnds)$ | Tupelkonstruktion |
| | \| | **seq**$(Bnds)$ | sequentielle Auswertung |
| | \| | **if**$(Val, Bnds, Bnds)$ | bedingte Auswertung |
| | \| | **loop**$(Bnds)$ | Schleife |
| | \| | **exit** | Schleifenterminierung |
| $Sig$ | ::= | **Placeholder**$(ide)$ | Platzhalter für Signaturen |
| | \| | **Exp**$(Sigs)$ | expandierte Signaturen |
| | \| | $Def : Typ$ | Wertsignatur |
| | \| | $Def <: Typ$ | Typsignatur |
| $Sigs$ | ::= | $\oslash$ | leere Signatur |
| | \| | $(Sigs,\ Sig)$ | Signatursequenz |
| $Bnd$ | ::= | **Placeholder**$(ide)$ | Platzhalter für Bindungen |
| | \| | **Exp**$(Bnds)$ | expandierte Bindungen |
| | \| | $Def = Typ$ | Typbindung |
| | \| | $Def = Val$ | Wertbindung |
| | \| | **var** $Def = Val$ | variable Wertbindung |
| $Bnds$ | ::= | $\oslash$ | leere Bindung |
| | \| | $(Bnds,\ Bnd)$ | Bindungssequenz |

Tabelle A.2: Abstrakte Syntax für TLMin

| Abgeleitete Mengen: | |
|---|---|
| $Typ$ | ein Typ |
| $Val$ | ein Wert |
| $Sig$ | eine Signatur |
| $Sigs$ | eine Sequenz von Signaturen |
| $Bnd$ | eine Bindung |
| $Bnds$ | eine Sequenz von Bindungen |
| $Def$ | ein Bezeichner in definierender Position |
| $Ref$ | ein Bezeichner in referenzierender Position |
| $Label$ | ein Bezeichner in selektierender Position |

In Tabelle A.1 werden die verschiedenen Sorten von Bezeichnern zusammengefaßt: definierend, referenzierend und selektierend (vgl. Abschnitt 3.1.3). Innerhalb der semantischen Interpretation einer in TLExt beschriebenen Syntaxerweiterung kann ein Bezeichner durch einen Platzhalter **Placeholder**(*ide*) ersetzt werden, wobei *ide* der Name des Platzhalters ist. Bei der Expansion einer Syntaxerweiterung (in einem TLMin-Programm oder in der semantischen Interpretation einer TLExt-Syntaxerweiterung) werden diese Platzhalter durch **Exp**(...) ersetzt; diese Kennzeichnung ist erforderlich, um die Bindungen, wie in Kapitel 5 beschrieben, korrekt auszuführen. Eine Sonderstellung nimmt ein referenzierender Bezeichner in einer Syntaxerweiterung ein: Er wird durch einen lokalen oder globalen Index *index* ersetzt, der ggf. nach der Expansion einer Syntaxerweiterung angepaßt werden muß. Die formale Beschreibung dieser Bindungsmanipulationen findet sich in Anhang B.3.

Die Tabelle A.2 zeigt die abstrakte Syntax für Typen, Werte, Signaturen und Bindungen aus TLMin, das eine Untermenge von TL bildet. Innerhalb der semantischen Interpretation einer in TLExt beschriebenen Syntaxerweiterung können ein Typ, ein Wert, eine oder mehrere Signaturen bzw. eine oder mehrere Bindungen durch einen Platzhalter **Placeholder**(*ide*) ersetzt werden, wobei *ide* der Name des Platzhalters ist. Bei der Expansion einer Syntaxerweiterung (in einem TLMin-Programm oder in der semantischen Interpretation einer TLExt-Syntaxerweiterung) werden die Platzhalter durch **Exp**(...) ersetzt und das gesamte expandierte Konstrukt durch **Exp**(...) markiert; diese Kennzeichnungen sind erforderlich, um die Bindungen, wie in Kapitel 5 beschrieben, korrekt auszuführen (vgl. Anhang B.3).

### A.2.3 TLExt

Die abstrakte Syntax von TLExt wird in Tabelle A.3 definiert. Die folgenden syntaktischen Objekte sind in TLExt definiert:

| Basismengen: | |
|---|---|
| $kw$ | ein Schlüsselwort |
| $ide$ | Bezeichner |

| | | | |
|---|---|---|---|
| $Grammars$ | ::= | $\oslash$ | leere Grammatik |
| | \| | $(Grammars, Grammar)$ | Sequenz von Grammatiken |
| $Grammar$ | ::= | $\mathbf{Def}(ide)(SortSigs) = Prods$ | Definition eines Nonterminals |
| | \| | $\mathbf{Ref}(ide)(SortSigs) ::= Prods$ | Redefinition eines Nonterminals |
| $Prods$ | ::= | $ProdRef$ | Referenz/Kopie von Prod. |
| | \| | $Pats \Rightarrow Bases$ | Muster und Expansionen |
| | \| | $Prods \mid Prods$ | Produktionsalternativen |
| $Pats$ | ::= | $\oslash$ | leeres Muster |
| | \| | $(Pats, Pat)$ | Mustersequenz |
| $Pat$ | ::= | $kw$ | Schlüsselwort |
| | \| | $PDefs : ProdRef$ | Definition von Platzhaltern |
| $ProdRef$ | ::= | $\mathbf{Ref}(ide)(\downarrow PRefs)$ | Referenz von Produktionen |
| | \| | $\mathbf{COPY}(\mathbf{Ref}(ide)(\downarrow PRefs))$ | Kopie von Produktionen |
| $PDefs$ | ::= | $\mathbf{Def}(ide)$ | eine Platzhalterdefinition |
| | \| | $(PDefs, \mathbf{Def}(ide))$ | Platzhalterdefinitionensequenz |
| $PRefs$ | ::= | $\oslash$ | leere Referenzensequenz |
| | \| | $(PRefs, \mathbf{Ref}(ide))$ | Platzhalterreferenzensequenz |
| $Bases$ | ::= | $Base$ | ein Objekt aus der Basissprache |
| | \| | $(Bases, Base)$ | Objektsequenz |
| $Base$ | ::= | $\mathbf{Typ}(Typ)$ | Typ aus der Basissprache |
| | \| | $\mathbf{Val}(Val)$ | Wert aus der Basissprache |
| | \| | $\mathbf{Sigs}(Sigs)$ | Signaturen aus der Basissprache |
| | \| | $\mathbf{Bnds}(Bnds)$ | Bindungen aus der Basissprache |
| $Sorts$ | ::= | $Sort$ | eine Sorte |
| | \| | $(Sorts, Sort)$ | Sortensequenz |
| $Sort$ | ::= | $\mathbf{SortIde} \mid \mathbf{SortVal} \mid \mathbf{SortTyp}$ | Basissorten |
| | \| | $\mathbf{SortBnds} \mid \mathbf{SortSigs}$ | Basissorten |
| | \| | $\mathbf{SortGram}(\downarrow SortSigs)(\uparrow Sorts)$ | Sorte eines Nonterminals |
| | \| | $\mathbf{SortPlaceholder}(Sort)$ | Sorte eines Platzhalters |
| $SortSigs$ | ::= | $\oslash$ | leere Sortensignatur |
| | \| | $(SortSigs, SortSig)$ | Sortensignatursequenz |
| $SortSig$ | ::= | $\mathbf{Def}(ide) : Sort$ | Platzhaltersignatur |

Tabelle A.3: Abstrakte Syntax für TLExt

| Abgeleitete Mengen: | |
|---|---|
| *Grammar* | eine Grammatik |
| *Grammars* | eine Sequenz von Grammatiken |
| *Prods* | Produktionen |
| *ProdRef* | Referenz oder Kopie von Produktionen |
| *Pat* | ein Muster |
| *Pats* | eine Sequenz von Mustern |
| *PDefs* | eine Sequenz von Platzhalterdefinitionen |
| *PRefs* | eine Sequenz von Platzhalterreferenzen |
| *Base* | ein Objekt aus der Basissprache |
| *Bases* | eine Sequenz von Objekten aus der Basissprache |
| *SortSig* | eine Platzhaltersignatur |
| *SortSigs* | eine Sequenz aus Platzhaltersignaturen |
| *Sort* | eine Sorte |
| *Sorts* | eine Sequenz von Sorten |

Die Tabelle A.3 zeigt die abstrakte Syntax von TLExt. Im folgenden wird die Tabelle kurz erklärt. Eine ausführliche Beschreibung der Erweiterungssprache findet sich in Kapitel 3.2.

Eine Grammatik *Grammars* besteht aus Definitionen oder Redefinitionen von Nonterminalen. Jedem Nonterminal können vererbte Attribute zugeordnet werden, die durch eine Sortensignatur *SortSigs* beschrieben werden. Produktionen sind entweder Referenzen/Kopien von den Produktionen eines Nonterminals *ProdRef* oder ein Muster *Pats* und die Berechnungsvorschriften für abgeleitete Attribute *Bases*. Ein Muster besteht aus Schlüsselworten *kw* und Platzhalter-Definitionen *Defs*. Die Platzhalter erhalten die aktuellen Werte der abgeleiteten Attribute der angegebenen Produktionen *ProdRef*. Die Referenz oder Kopie von Produktionen enthält Platzhalter-Referenzen *Refs*, die angeben, welche aktuellen Werte an die vererbten Attribute des Nonterminals zu übergeben sind. **COPY**(...) bedeutet, daß nicht eine Referenz auf das Nonterminal vermerkt wird, sondern die Produktionen des Nonterminals kopiert werden, so daß Redefinitionen des Nonterminals ohne Auswirkungen auf diese Produktion bleiben. Die Berechnungsvorschriften für die Werte der abgeleiteten Attribute *Bases* bestehen aus syntaktischen Konstrukten der Basissprache *Base*, die Platzhalter-Referenzen enthalten können. Eine Sorte ist entweder eine Basissorte (für die syntaktischen Konstrukte der Basissprache) oder ein Sortenkonstruktur (für Konstrukte der Erweiterungssprache). Sortenkonstrukturen beschreiben die Sorte eines Nonterminals **SortGram**, wobei die Signatur der vererbten Attribute und die Sorten der abgeleiteten Attribute angegeben werden, oder die Sorte eines Platzhalters **SortPlaceholder**, der ein syntaktisches Konstrukt einer Basissorte repräsentiert. Eine Sortensignatur *SortSigs* ordnet Bezeichnern (Platzhaltern) Sorten zu und dient damit zur Beschreibung vererbter Attribute.

# Anhang B

# Statische Semantik

In diesem Kapitel werden die Bedingungen für wohlgeformte Konstrukte der Erweiterungssprache TLExt und die geänderte Bindungsphase der Basissprache TLMin beschrieben. Die Typprüfungsphase der Basissprache ist nicht Thema dieser Arbeit.

## B.1 Notation

Die Bedingungen für wohlgeformte Konstrukte werden als Klauseln notiert, die der operationalen Semantikbeschreibung von [Plotkin 81; Hennessy 90] entlehnt sind.

Eine Klausel [Clause] der Form $\frac{premises}{conclusion}\,[Clause]$ sagt aus, daß die Konklusion *conclusion* gilt, wenn die Prämissen *premises* erfüllt sind. Falls keine Prämisse angegeben wird, handelt es sich um ein Axiom und die Konklusion gilt ohne Einschränkung.

Eine Konklusion oder eine Prämisse der Form $Ctxt \vdash formula$ bedeutet, daß im Kontext $Ctxt$ die Formel $formula$ bewiesen werden kann.

Der Kontext $Ctxt$ besteht aus einem oder mehreren Attributen, wobei jedes Attribut einen eindeutigen Namen $attr$ besitzt. Falls der Kontext nur aus einem Attribut besteht, wird dieses direkt angegeben. Da die Angabe des gesamten Kontextes bei mehreren Attributen sehr unübersichtlich sein würde, ist als abkürzende Schreibweise $Ctxt(attr = value)$ bzw. $Ctxt(c)$ mit $c = (attr = value)$ definiert, mit der Semantik, daß das Attribut $attr$ den Wert $value$ besitzt und die Werte der anderen Attribute beliebig sind.

Ein Attribut kann entweder eine Umgebung $Env$, ein Modus $Mode$ oder ein Zähler $Count$ sein.

Eine Umgebung $env \in Env$ ist eine geordnete Liste von (gleichartigen) Elementen $(element, \ldots)$, $element \in Element$. Auf Umgebungen sind folgende Operationen definiert:

- Erzeugung einer leeren Umgebung: $nil :\rightarrow Env$, notiert durch $\oslash$.
- Konstruktion einer Umgebung: $cons : Env \times Element \rightarrow Env$, notiert durch $(env, element)$, wobei $element \in Element$ ein Element ist.
- Als verkürzende Schreibweise wird auch $(env, element, env')$ benutzt, wobei die Funktion $append : Env \times Env \rightarrow Env$, notiert durch $(env, env')$, vorausgesetzt wird.
- Anzahl der in der Umgebung befindlichen Elemente: $size : Env \rightarrow Int$, notiert durch $|env|$.
- Prüfung, ob ein Element in einer Umgebung vorhanden ist: $in : Element \times Env \rightarrow Bool$, notiert durch $element \in env$.

Außerdem erweist sich die Operation $get : Env \times Int \rightarrow Element$ als adäquat, um eine konzise und implementationsnahe Formulierung zu ermöglichen. Sie wird notiert durch $env[int]$, mit der Semantik, das ein durch den ganzzahligen Index bezeichnetes Element aus der Umgebung extrahiert wird. Dabei wird zwischen lokalen $> 0$ und globalen $< 0$ Indices unterschieden. Die Semantik des Indexzugriffs wird durch folgende Regeln beschrieben:

$$\frac{+(|E'|+1) = index}{E, element, E' \vdash index :: element} \quad \textit{[Environment Access: Local]}$$

$$\frac{-(|E|+1) = index}{E, element, E' \vdash index :: element} \quad \textit{[Environment Access: Global]}$$

Es wird vorausgesetzt, daß auf den Indices (als ganzen Zahlen) die Operationen $+$, $-$ und $=$ definiert sind.

Ein Modus kann eine endliche Anzahl verschiedener Werte annehmen; beispielsweise besteht die Wertemenge des Modus $Bool$ aus den Konstanten $true$ und $false$. Auf den Werten eines Modus ist die Vergleichsoperation definiert.

Ein Zähler kann positive ganze Zahlen als Werte annehmen. Es sind die Operationen $+$, $-$, $>$, $<$ usw. definiert.

Eine Formel $formula$ besteht entweder aus einem Prädikat, einer Zuordnung oder einer Substitution. Eine Formel kann abstrakte Syntaxbäume $AST$ als Teile enthalten. Ein Prädikat $AST\ \ form$ sagt aus, daß der Syntaxbaum $AST$ die Form $form$ hat. Teilweise fehlt die Angabe der Form, wenn aufgrund der Struktur des Syntaxbaums keine Mehrdeutigkeiten möglich sind. Eine Zuordnung $AST :: result$ bedeutet, daß der Syntaxbaum $AST$ ein Resultat $result$ erzeugt.

## B.2 Bindung und Sortenkonformität von TLExt

Bindung und Sortenkonformität werden in einem Kontext untersucht, der als einziges Attribut die Umgebung $E$ enthält, die jedem definierenden Bezeichner **Def**$(ide)$ der Erweiterungssprache (Nonterminal oder Platzhalter) eine Sorte $Sort$ zuordnet: $E = (ide :$

$Sort, \ldots$). Im folgenden wird auf die Angabe des Kontextes verzichtet und das Attribut $E$ direkt synonym für den Kontext benutzt.

Die in diesem Anhang definierten Klauseln formalisieren die folgenden Bindungsregeln für die Erweiterungssprache TLExt:

- Die Bindung der Platzhalter erfolgt lexikalisch, d. h. ein definierender Bezeichner bindet vom Zeitpunkt seines Auftretens bis zum Ende des umgebenden Sichtbarkeitsbereichs oder bis zu einer erneuten Definition des Bezeichners alle referenzierenden Bezeichner mit demselben Namen.
- Die Nonterminale einer Grammatik folgen der Sichtbarkeitsregel von rekursiven Bindungen, d. h. ein definierender Bezeichner bindet alle referenzierenden Bezeichner mit demselben Namen innerhalb der Grammatik, also auch Referenzen vor der Definition des Bezeichers.
- Das Überladen von Bezeichnern ist nicht möglich.

Die Bindung der Bezeichner aus der Basissprache erfolgt getrennt und kann anderen Bindungsregeln gehorchen.

### B.2.1 Bezeichner

*[Scope Reference]*

$$\frac{ide \notin E'}{(E,\ ide : Sort,\ E') \vdash \mathbf{Ref}(ide) : Sort}$$

Ein referenzierender Bezeichner *ide* wird von einem definierendem Bezeichner gebunden, wenn kein Bezeichner mit diesem Namen später (in $E'$) definiert wurde und die Sorte $Sort$ von Definition und Referenz übereinstimmt.

### B.2.2 Sorten und Sortensignaturen

In diesem Abschnitt werden Äquivalenztests auf Sortensignaturen und die Umwandlungen von/in Sortensignaturen beschrieben.

*[Defs & Sorts: Env Empty]*

$$\frac{}{E \vdash \emptyset : \emptyset :: \emptyset}$$

*[Defs & Sorts: Env Cons]*

$$\frac{E \vdash Defs : Sorts :: E'}{E \vdash (Defs,\ \mathbf{Def}(ide)) : (Sorts,\ Sort) :: (E',\ ide : \mathbf{SortPlaceholder}(Sort))}$$

Zwei Listen von definierenden Bezeichnern $Defs$ und Sorten $Sorts$ erzeugen eine Umgebung, in der jeder Bezeichner *ide* mit der Sorte **SortPlaceholder**($Sort$) als Platzhalter gekennzeichnet ist.

*[SortSigs: Equiv Empty]*

$$\frac{}{E \vdash \oslash \Leftrightarrow \oslash}$$

*[SortSigs: Equiv Cons]*

$$\frac{E \vdash SortSigs \Leftrightarrow SortSigs'}{E \vdash (SortSigs, \mathbf{Def}(ide) : Sort) \Leftrightarrow (SortSigs', \mathbf{Def}(ide') : Sort)}$$

Zwei Sortensignaturen sind äquivalent, wenn die Sorten übereinstimmen. Die Bezeichner der Sortensignatur müssen nicht übereinstimmen.

*[Refs & SortSigs: Equiv Empty]*

$$\frac{}{E \vdash \oslash \Leftrightarrow \oslash}$$

*[Refs & SortSigs: Equiv Cons]*

$$\frac{E \vdash Refs \Leftrightarrow SortSigs \qquad E \vdash \mathbf{Ref}(ide) : \mathbf{SortPlaceholder}(Sort)}{E \vdash (Refs, \mathbf{Ref}(ide)) \Leftrightarrow (SortSigs, \mathbf{Def}(ide') : Sort)}$$

Ein referenzierender Bezeichner *ide* paßt zu einer Signatur **Def**(ide') : Sort, wenn der Bezeichner einen Platzhalter mit der Sorte **SortPlaceholder**($Sort$) repräsentiert.

*[SortSigs: Env Empty]*

$$\frac{}{E \vdash \oslash :: \oslash}$$

*[SortSigs: Env Cons]*

$$\frac{E \vdash SortSigs :: E'}{E \vdash (SortSigs, \mathbf{Def}(ide) : Sort) :: (E', ide : \mathbf{SortPlaceholder}(Sort))}$$

Eine Sortensignatur erzeugt eine Umgebung, in der jeder Bezeichner *ide* als Platzhalter mit der Sorte **SortPlaceholder**($Sort$) gekennzeichnet ist.

### B.2.3 Muster

In diesem Abschnitt werden wohlgeformte Muster beschrieben.

*[Pattern: Empty]*

$$\frac{}{E \vdash \oslash :: \oslash}$$

*[Pattern: Cons]*

$$\frac{E \vdash Pats :: E' \qquad (E, E') \vdash Pat :: E''}{E \vdash (Pats, Pat) :: (E', E'')}$$

*[Pattern: Keyword]*

$$\frac{}{E \vdash kw :: \oslash}$$

*[Pattern: Define Placeholders 1]*

$$\frac{E \vdash \mathbf{Ref}(ide) : \mathbf{SortGram}(\downarrow SortSigs)(\uparrow Sorts) \quad E \vdash Refs \Leftrightarrow SortSigs \quad E \vdash Defs : Sorts :: E'}{E \vdash Defs : \mathbf{COPY}(\mathbf{Ref}(ide)(\downarrow Refs)) :: E'}$$

*[Pattern: Define Placeholders 2]*

$$\frac{E \vdash \mathbf{Ref}(ide) : \mathbf{SortGram}(\downarrow SortSigs)(\uparrow Sorts) \quad E \vdash Refs \Leftrightarrow SortSigs \quad E \vdash Defs : Sorts :: E'}{E \vdash Defs : \mathbf{Ref}(ide)(\downarrow Refs) :: E'}$$

Die Platzhalter $Defs$ erzeugen eine Umgebung, deren Sorten sich aus den Sorten $Sorts$ der abgeleiteten Attribute des Nonterminals $ide$ ergeben. Es wird geprüft, ob die aktuellen Werte der vererbten Attribute $Refs$ mit der Signatur $SortSigs$ sortenkonform sind.

### B.2.4 Produktionen

In diesem Abschnitt werden wohlgeformte Produktionen beschrieben.

*[Productions: Nonterminal Reference]*

$$\frac{E \vdash \mathbf{Ref}(ide) : \mathbf{SortGram}(\downarrow SortSigs)(\uparrow Sorts) \quad E \vdash Refs \Leftrightarrow SortSigs}{E \vdash \mathbf{Ref}(ide)(\downarrow Refs) : Sorts}$$

*[Productions: Nonterminal Copy]*

$$\frac{E \vdash \mathbf{Ref}(ide) : \mathbf{SortGram}(\downarrow SortSigs)(\uparrow Sorts) \quad E \vdash Refs \Leftrightarrow SortSigs}{E \vdash \mathbf{COPY}(\mathbf{Ref}(ide)(\downarrow Refs)) : Sorts}$$

Die Sorten $Sorts$ der abgeleiteten Attribute einer Referenz/Kopie eines Nonterminals $ide$ ergeben sich der Sorte $\mathbf{SortGram}(\downarrow SortSigs)(\uparrow Sorts)$ der Definition des Nonterminals. Es wird geprüft, ob die aktuellen Werte $Refs$ für die vererbten Attribute mit der Signatur $SortSigs$ sortenkonform sind.

*[Productions: Pattern & Expansions]*

$$\frac{E \vdash Pats :: E' \quad (E, E') \vdash Bases : Sorts}{E \vdash Pats \Rightarrow Bases : Sorts}$$

Die abgeleiteten Attribute einer Produktion besitzen die Sorten $Sorts$, wenn die semantischen Interpretationen $Bases$ in der durch das Muster $Pats$ lokal erweiterten Umgebung diese Sorten besitzen.

*[Productions: Alternatives]*

$$\frac{E \vdash Prods : Sorts \quad E \vdash Prods' : Sorts}{E \vdash Prods \,|\, Prods' : Sorts}$$

Die abgeleiteten Attribute von alternativen Produktionen müssen die gleichen Sorten $Sorts$ besitzen.

### B.2.5 Grammatik

Dieser Abschnitt beschreibt wohlgeformte rekursive Grammatiken.

*[Grammars: Empty]*

$$\frac{}{E \vdash \oslash :: \oslash}$$

*[Grammars: Cons]*

$$\frac{(E, E', E'') \vdash \mathit{Grammars} :: E' \quad (E, E', E'') \vdash \mathit{Grammar} :: E''}{E \vdash (\mathit{Grammars}, \mathit{Grammar}) :: (E', E'')}$$

Die Erfüllbarkeit dieses Prädikats hängt von der Evaluationsstrategie ab, denn die beiden Prämissen sind wechselseitig voneinander abhängig (rekursive Grammatikregeln). In der gegenwärtigen Implementation wird folgender Algorithmus verwendet: Die Erweiterung des Kontextes durch die Produktion *Prod* muß sich ohne Kenntnis des Kontextes ermitteln lassen, also muß mindestens eine Alternative der Form [Productions: Pattern & Expansions] vorhanden sein.

*[Grammar: Define Nonterminal]*

$$\frac{E \vdash \mathit{SortSigs} :: E' \quad (E, E') \vdash \mathit{Prods} : \mathit{Sorts}}{E \vdash \mathbf{Def}(\mathit{ide})(\mathit{SortSigs}) = \mathit{Prods} :: \mathit{ide} : \mathbf{SortGram}(\downarrow \mathit{SortSigs})(\uparrow \mathit{Sorts})}$$

Die Definition eines Nonterminals *ide* durch Produktionen *Prods* erweitert die Umgebung um den Bezeichner *ide* mit der Sorte $\mathbf{SortGram}(\downarrow \mathit{SortSigs})(\uparrow \mathit{Sort})$, wobei sich die Sorten *Sorts* der abgeleiteten Attribute aus den Produktionen *Prods* ergeben. Es wird davon ausgegangen, daß die Erweiterung des Kontextes um diese Definition schon erfolgt ist (siehe [Grammars: Cons]). Die Produktionen können auf die vererbten Attribute *SortSigs* zurückgreifen.

*[Grammar: Redefine Nonterminal]*

$$\frac{E \vdash \mathbf{Ref}(\mathit{ide}) : \mathbf{SortGram}(\downarrow \mathit{SortSigs}')(\uparrow \mathit{Sorts}) \quad E \vdash \mathit{SortSigs} \Leftrightarrow \mathit{SortSigs}' \quad E \vdash \mathit{SortSigs} :: E' \quad (E, E') \vdash \mathit{Prods} : \mathit{Sorts}}{E \vdash \mathbf{Ref}(\mathit{ide})(\mathit{SortSigs}) ::= \mathit{Prods} :: \oslash}$$

Die Redefinition eines Nonterminals *ide* durch Produktionen *Prods* ist wohlgeformt, wenn die Produktionen die Sorten *Sorts* als abgeleitete Attribute ergeben, das Nonterminal eine Grammatik-Regel mit der Sorte $\mathbf{SortGram}(\downarrow \mathit{SortSigs}')(\uparrow \mathit{Sorts})$ referenziert und die Signatur der vererbten Attribute *SortSigs* äquivalent zu der Signatur *SortSigs'* ist. Die Umgebung wird nicht verändert.

### B.2.6 Basissprache

Die Objekte der Basissprache werden auf Sortenkonformität, aber nicht auf Typkonformität geprüft. Die als Platzhalter in den Objekten der Basissprache auftretenden referenzierenden Bezeichner werden daraufhin überprüft, ob die definierenden Bezeichner, an die sie

gebunden sind, die richtige Sorte besitzen. Die Typkonformität kann erst nach der Expansion einer Syntaxerweiterung (endgültig) festgestellt werden. Sie wird von der Basissprache festgelegt und ist damit nicht Bestandteil dieser Arbeit. Die Typkonformitätsregeln für die Sprache TL, von der die hier verwendete Basissprache TLMin abgeleitet ist, finden sich in [Matthes 93]. Es sei an dieser Stelle darauf hingewiesen, daß referenzierende und selektierende Bezeichner der Basissprache nicht darauf hin überprüft werden können, ob sie an Objekte der richtigen Sorte gebunden sind.

#### B.2.6.1 TLExt→TLMin

Dieser Abschnitt beschreibt die Einbettung von Konstrukten der Basissprache TLMin in die Syntaxerweiterungssprache TLExt. Jedem Konstrukt wird eine Sorte zugeordnet.

*[Base Language: Cons]*

$$\frac{E \vdash Bases : Sorts \quad E \vdash Base : Sort}{E \vdash (Bases, Base) : (Sorts, Sort)}$$

*[Base Language: Type]*

$$\frac{E \vdash Typ\ type}{E \vdash \mathbf{Typ}(Typ) : \mathbf{SortTyp}}$$

*[Base Language: Val]*

$$\frac{E \vdash Val\ value}{E \vdash \mathbf{Val}(Val) : \mathbf{SortVal}}$$

*[Base Language: Sigs]*

$$\frac{E \vdash Sigs\ sigs}{E \vdash \mathbf{Sigs}(Sigs) : \mathbf{SortSigs}}$$

*[Base Language: Bnds]*

$$\frac{E \vdash Bnds\ bnds}{E \vdash \mathbf{Bnds}(Bnds) : \mathbf{SortBnds}}$$

#### B.2.6.2 Bezeichner

Dieser Abschnitt beschreibt wohlgeformte definierende, referenzierende und selektierende Bezeichner. Jeder Bezeichner kann durch einen Platzhalter mit der Sorte **SortPlaceholder**(**SortIde**) ersetzt werden.

*[Def/Ref/Label: Placeholder]*

$$\frac{E \vdash \mathbf{Ref}(ide) : \mathbf{SortPlaceholder}(\mathbf{SortIde})}{E \vdash \mathbf{Placeholder}(ide)}$$

Eine Platzhalter-Referenz *ide* ist ein definierender, referenzierender oder selektierender Bezeichner, wenn der Platzhalter mit der Sorte **SortIde** im statischen Kontext definiert ist.

*[Def: Expanded]*

$$\frac{E \vdash Def}{E \vdash \mathbf{Exp}(Def)}$$

*[Ref: Expanded]*

$$\frac{E \vdash Ref}{E \vdash \mathbf{Exp}(Ref)}$$

*[Label: Expanded]*

$$\frac{E \vdash Label}{E \vdash \mathbf{Exp}(Label)}$$

*[Def: ide]*

$$\frac{}{E \vdash \mathbf{Def}(ide)}$$

*[Ref: ide]*

$$\frac{}{E \vdash \mathbf{Ref}(ide)}$$

*[Ref: index]*

$$\frac{}{E \vdash index}$$

*[Label: ide]*

$$\frac{}{E \vdash \mathbf{Label}(ide)}$$

### B.2.6.3 Typen

Dieser Abschnitt beschreibt die Propagation der Umgebung $E$ durch Typausdrücke. Jeder Typausdruck kann durch einen Platzhalter mit der Sorte **SortPlaceholder**(**SortTyp**) ersetzt werden.

*[Type: Placeholder]*
$$\frac{E \vdash \mathbf{Ref}(ide) : \mathbf{SortPlaceholder}(\mathbf{SortTyp})}{E \vdash \mathbf{Placeholder}(ide)\ type}$$

*[Type: Expanded]*
$$\frac{E \vdash Typ\ type}{E \vdash \mathbf{Exp}(Typ)\ type}$$

*[Type: Nok]*
$$\frac{}{E \vdash \mathbf{Nok}\ type}$$

*[Type: Ok]*
$$\frac{}{E \vdash \mathbf{Ok}\ type}$$

*[Type: Basetype]*
$$\frac{}{E \vdash \mathbf{Bool}, \mathbf{Int}, \ldots\ type}$$

*[Type: Ref]*
$$\frac{E \vdash Ref}{E \vdash Ref\ type}$$

*[Type: Field Selection]*
$$\frac{E \vdash Val\ value \quad E \vdash Label}{E \vdash Val.Label\ type}$$

Bei referenzierenden und selektierenden Bezeichnern kann nicht geprüft werden, ob sie einen Typ bezeichnen.

*[Type: Fun]*
$$\frac{E \vdash Sigs\ sigs \quad E \vdash Typ\ type}{E \vdash \mathbf{Fun}(Sigs) : Typ\ type}$$

*[Type: Arr]*
$$\frac{E \vdash Typ\ type}{E \vdash \mathbf{Arr}(Typ)\ type}$$

*[Type: Tup]*
$$\frac{E \vdash Sigs\ sigs}{E \vdash \mathbf{Tup}(Sigs)\ type}$$

*[Type: Var]*
$$\frac{E \vdash Typ\ type}{E \vdash \mathbf{Var}(Typ)\ type}$$

*[Type: Oper]*
$$\frac{E \vdash Sigs\ sigs \quad E \vdash Typ\ type}{E \vdash \mathbf{Oper}(Sigs)Typ\ type}$$

*[Type: Apply]*
$$\frac{E \vdash Typ\ type \quad E \vdash Bnds\ bnds}{E \vdash Typ(Bnds)\ type}$$

### B.2.6.4 Werte

Dieser Abschnitt beschreibt die Propagation der Umgebung $E$ durch Wertausdrücke. Jeder Wertausdruck kann durch einen Platzhalter mit der Sorte **SortPlaceholder**(**SortVal**) ersetzt werden.

*[Value: Placeholder]*
$$\frac{E \vdash \mathbf{Ref}(ide) : \mathbf{SortPlaceholder}(\mathbf{SortVal})}{E \vdash \mathbf{Placeholder}(ide)\ value}$$

*[Value: Expanded]*
$$\frac{E \vdash Val\ value}{E \vdash \mathbf{Exp}(Val)\ value}$$

*[Value: ok]*
$$\frac{}{E \vdash \mathbf{ok}\ value}$$

*[Value: Literals]*
$$\frac{}{E \vdash int, real, \ldots\ value}$$

*[Value: Ref]*
$$\frac{E \vdash Ref}{E \vdash Ref\ value}$$

*[Val: Field Selection]*
$$\frac{E \vdash Val\ value \quad E \vdash Label}{E \vdash Val.Label\ value}$$

Bei referenzierenden und selektierenden Bezeichnern kann nicht geprüft werden, ob sie einen Wert bezeichnen.

*[Value: Abstraction]*
$$\frac{E \vdash Sigs\ sigs \quad E \vdash Val\ value}{E \vdash \mathbf{fun}(Sigs)Val\ value}$$

*[Value: Application]*
$$\frac{E \vdash Val\ value \quad E \vdash Bnds\ bnds}{E \vdash Val(Bnds)\ value}$$

*[Value: Array]*
$$\frac{E \vdash Bnds\ bnds}{E \vdash \mathbf{arr}(Bnds)\ value}$$

*[Value: Element Selection]*
$$\frac{E \vdash Val\ value \quad E \vdash Val'\ value}{E \vdash Val[Val'])\ value}$$

*[Value: Tuple]*
$$\frac{E \vdash Bnds\ bnds}{E \vdash \mathbf{tup}(Bnds)\ value}$$

*[Value: Sequence]*
$$\frac{E \vdash Bnds\ bnds}{E \vdash \mathbf{seq}(Bnds)\ value}$$

*[Value: Conditional]*
$$\frac{E \vdash Val\ value \quad E \vdash Bnds\ bnds \quad E \vdash Bnds'\ bnds}{E \vdash \mathbf{if}(Val, Bnds, Bnds')\ value}$$

*[Value: Loop]*
$$\frac{E \vdash Bnds\ bnds}{E \vdash \mathbf{loop}(Bnds)\ value}$$

*[Value: Exit]*
$$\frac{}{E \vdash \mathbf{exit}\ value}$$

### B.2.6.5 Signaturen

Dieser Abschnitt beschreibt die Propagation der Umgebung $E$ durch Signaturen. Jede Signatur kann durch einen Platzhalter mit der Sorte **SortPlaceholder**(**SortSigs**) ersetzt werden.

*[Sigs: Empty]*
$$\frac{}{E \vdash \oslash\ sigs}$$

*[Sigs: Cons]*
$$\frac{E \vdash Sigs\ sigs \quad E \vdash Sig\ sig}{E \vdash (Sigs, Sig)\ sigs}$$

*[Sig: Placeholder]*
$$\frac{E \vdash \mathbf{Ref}(ide) : \mathbf{SortPlaceholder}(\mathbf{SortSigs})}{E \vdash \mathbf{Placeholder}(ide)\ sig}$$

*[Sig: Expanded]*
$$\frac{E \vdash Sigs\ sigs}{E \vdash \mathbf{Exp}(Sigs)\ sig}$$

*[Sig: Type Signature]*
$$\frac{E \vdash Def \quad E \vdash Typ\ type}{E \vdash Def <: Typ\ sig}$$

*[Sig: Value Signature]*
$$\frac{E \vdash Def \quad E \vdash Typ\ type}{E \vdash Def : Typ\ sig}$$

#### B.2.6.6 Bindungen

Dieser Abschnitt beschreibt die Propagation der Umgebung $E$ durch Bindungen. Jede Bindung kann durch einen Platzhalter mit der Sorte **SortPlaceholder**(**SortBnds**) ersetzt werden.

*[Bnds: Empty]*

$$\frac{}{E \vdash \oslash\ bnds}$$

*[Bnds: Cons]*

$$\frac{E \vdash Bnds\ bnds \qquad E \vdash Bnd\ bnd}{E \vdash (Bnds, Bnd)\ bnds}$$

*[Bnd: Placeholder]*

$$\frac{E \vdash \mathbf{Ref}(ide) : \mathbf{SortPlaceholder}(\mathbf{SortBnds})}{E \vdash \mathbf{Placeholder}(ide)\ bnd}$$

*[Bnd: Expanded]*

$$\frac{E \vdash Bnds\ bnds}{E \vdash \mathbf{Exp}(Bnds)\ bnd}$$

*[Bnd: Type Binding]*

$$\frac{\begin{array}{c}E \vdash Def \\ E \vdash Typ\ type\end{array}}{E \vdash Def = Typ\ bnd}$$

*[Bnd: Value Binding]*

$$\frac{\begin{array}{c}E \vdash Def \\ E \vdash Val\ value\end{array}}{E \vdash Def = Val\ bnd}$$

*[Bnd: Variable Binding]*

$$\frac{\begin{array}{c}E \vdash Def \\ E \vdash Val\ value\end{array}}{E \vdash \mathbf{var}\ Def = Val\ bnd}$$

## B.3 Bindungsphase von TLMin

Die Bindung von referenzierenden Bezeichnern findet normalerweise gleichzeitig mit der Typprüfung in der Typprüfungsphase statt.[1] Wie in Kapitel 5 beschrieben, wird eine (eingeschränkte) Bindungsphase für die Definition von Syntaxerweiterungen benötigt. Außerdem sind Änderungen an der Bindungsphase der Basissprache nötig, um Bindungsprobleme bei der Expansion von Syntaxerweiterungen zu lösen. Deshalb wird hier die Bindungsphase von TLMin unabhängig von der Typprüfung beschrieben. Die Typprüfung entspricht der von TL [Matthes 93, Kap. 6 und Anhang A] und ist hier nicht wiedergegeben.

In der Bindungsphase wird zu jedem referenzierenden Auftreten eines Bezeichners gemäß der Sichtbarkeitsregeln geprüft, ob ein entsprechender definierender Bezeichner im statischen Kontext existiert. Jedes referenzierende Auftreten eines Bezeichners wird durch einen Index ersetzt, der die Position des entsprechenden definierenden Bezeichners in der Umgebung angibt. Schon gebundene Bezeichner, die durch lokale de Bruijn-Indices ersetzt sind, werden angepaßt. Selektierende Bezeichner können nicht gebunden werden, sondern werden erst von der Typprüfung behandelt.

Die Bindungsphase liefert (konzeptionell) eine Kopie des Syntaxbaums, in der alle referenzierenden Bezeichner durch Indices ersetzt sind. Im folgenden werden nur „echte“ Substitutionen $x \Leftarrow x'$ angegeben und die Kopieroperationen implizit angenommen.

D. h. die Klausel *[Example]*

$$\frac{Ctxt \vdash \alpha_1 \Leftarrow \alpha'_1 \quad \ldots \quad Ctxt \vdash \alpha_n \Leftarrow \alpha'_n}{Ctxt \vdash AST(\alpha_1, \ldots, \alpha_n) \Leftarrow AST(\alpha'_1, \ldots, \alpha'_n)}$$

[1] In Anhang B.2 werden Bindung und Sortenprüfung von TLExt auch gleichzeitig behandelt.

wird durch die Klausel $\dfrac{Ctxt \vdash \alpha_1 \quad \ldots \quad Ctxt \vdash \alpha_n}{Ctxt \vdash AST(\alpha_1, \ldots, \alpha_n)}$ *[Example Shortcut]* abgekürzt.

In der Implementation wird der Syntaxbaum nicht kopiert, sondern die Substitutionen werden durch Seiteneffekte auf dem Syntaxbaum realisiert.

Für die Bindungsphase wird ein Kontext $Ctxt$ mit folgenden Attributen benötigt :

$expMode \in \{expanded, normal\}$ : Wenn $expMode = expanded$, handelt es sich bei dem zu bindenden Programmstück um die Expansion einer Syntaxerweiterung, in der spezielle Bindungsregeln zur Verhinderung von Bindungsproblemen gelten. $expMode = normal$ gilt außerhalb von Syntaxerweiterungen oder in instantiierten Platzhaltern.

$E_G = (ide, \ldots)$ : In dieser Umgebung werden alle definierten Bezeichner verzeichnet.

$E_L = (int, \ldots)$ : In dieser Umgebung wird für jeden lokal in einer expandierten Syntaxerweiterung definierten Bezeichner der aktuelle Wert des Zählers $count_L$ vermerkt, der zur Korrektur von gebundenen Referenzen auf diesen Bezeichner benötigt wird.

$count_G \in Int$ : Dieser Zähler zählt alle Bindungen ab einem bestimmten Punkt. Alle bei der Bindung eines referenzierenden Bezeichners entstehenden de Bruijn-Indices $i > count_G$ werden durch Cardelli-Indices ersetzt. Dieser Zähler wird benutzt, um bei der Definition einer Syntaxerweiterung alle globalen Bindungen durch Cardelli-Indices darzustellen.

$count_L \in Int$ : Dieser Zähler zählt alle Bindungen, die *nicht* in einer expandierten Syntaxerweiterung auftreten, also normale und in Instantiierungen von Platzhaltern auftretende Bindungen. Dieser Zähler wird benötigt, um gebundene lokale Referenzen (de Bruijn-Indices) in expandierten Syntaxerweiterungen zu korrigieren.

### B.3.1 Bezeichner

Dieser Abschnitt beschreibt die durch definierende Bezeichner ausgelösten Kontextveränderungen und die Bindung bzw. Bindungsmanipulationen bei referenzierenden Bezeichnern, die sich durch Substitutionen auf dem Syntaxbaum ausdrücken. Je nachdem, ob der definierende bzw. referenzierende Bezeichner sich im normalen Programmtext ($Ctxt(expMode = normal)$), in einer expandiereten Syntaxerweiterung ($Ctxt(Ctxt = expMode)expanded$) oder in einem instantiierten Platzhalter einer Syntaxerweiterung ($Ctxt(expMode = normal)$) befindet, werden verschiedene Operationen ausgeführt. Selektierende Bezeichner können nicht gebunden werden.

*[Def: Expand Syntax Extension]*

$$\frac{Ctxt(expMode = expanded) \vdash Def :: c}{Ctxt(expMode = normal) \vdash \mathbf{Exp}(Def) :: c}$$

*[Def: Expand Placeholder]*

$$\frac{Ctxt(expMode = normal) \vdash Def :: c}{Ctxt(expMode = expanded) \vdash \mathbf{Exp}(Def) :: c}$$

*[Ref: Expand Syntax Extension]*

$$\frac{Ctxt(expMode = expanded) \vdash Ref}{Ctxt(expMode = normal) \vdash \mathbf{Exp}(Ref)}$$

*[Ref: Expand Placeholder]*

$$\frac{Ctxt(expMode = normal) \vdash Ref}{Ctxt(expMode = expanded) \vdash \mathbf{Exp}(Ref)}$$

*[Label: Expand Syntax Extension]*

$$\frac{Ctxt(expMode = expanded) \vdash Label}{Ctxt(expMode = normal) \vdash \mathbf{Exp}(Label)}$$

*[Label: Expand Placeholder]*

$$\frac{Ctxt(expMode = normal) \vdash Label}{Ctxt(expMode = expanded) \vdash \mathbf{Exp}(Label)}$$

*[Def: Placeholder]*

$$\frac{}{Ctxt \vdash \mathbf{Placeholder}(ide) :: Id}$$

*[Ref/Label: Placeholder]*

$$\frac{}{Ctxt \vdash \mathbf{Placeholder}(ide)}$$

*[Def: Scope Local in Expansion]*

$$\frac{c = (E_G = (E_G, ?), count_G = count_G + 1, E_L = (E_L, count_L))}{Ctxt(expMode = expanded) \vdash \mathbf{Def}(ide) :: c}$$

Ein definierender Bezeichner *ide* in einer Expansion einer Syntaxerweiterung erzeugt eine Änderung des Kontextes: Ein anonymer Bezeichner ? wird in die Umgebung aller definierten Bezeichner $E_G$ eingetragen. Der Zähler der in der Expansion eingeführten Bindungen $count_G$ wird erhöht. Der Zähler der globalen Bindungen $count_L$ wird (für diesen Bezeichner) in die Umgebung $E_L$ eingetragen.

*[Def: Scope Global]*

$$\frac{c = (E_G = (E_G, ide), count_G = count_G + 1, count_L = count_L + 1)}{Ctxt(expMode = normal) \vdash \mathbf{Def}(ide) :: c}$$

Ein definierender Bezeichner *ide* im normalen Quelltext oder einem instantiierten Platzhalter erzeugt eine Änderung des Kontextes: Der Bezeichner wird in die Umgebung aller definierten Bezeichner $E_G$ eingetragen. Der Zähler der in der Expansion eingeführten Bindungen $count_G$ wird erhöht. Der Zähler der globalen Bindungen $count_L$ wird erhöht.

*[Ref: Scope Local Index]*

$$\frac{index > 0}{Ctxt \vdash index \Leftarrow index + (count_L - E_L[index])}$$

Ein gebundener *lokaler* referenzierender Bezeichner in einer expandierten Syntaxerweiterung, dargestellt durch einen de Bruijn-Index $index > 0$, wird um die Anzahl der seit seiner Definition durch die Instantiierung von Platzhaltern neu eingeführten Bindungen inkrementiert.

*[Ref: Scope Global Index]*

$$\frac{index < 0}{Ctxt \vdash index \Leftarrow index}$$

Ein gebundener *globaler* referenzierender Bezeichner in einer expandierten Syntaxerweiterung, dargestellt durch einen Cardelli $index < 0$, wird unverändert übernommen.

*[Ref: Scope Local]*

$$\frac{ide \notin E' \quad |E'| <= count_G}{Ctxt(E_G = (E,\, ide,\, E')) \vdash \mathbf{Ref}(ide) \Leftarrow +(|E'| + 1)}$$

Ein referenzierender Bezeichner *ide* wird gebunden. Er wird durch einen de Bruijn-Index $i > 0$ ersetzt, der eine lokale Bindung darstellt, wenn er nicht die Schranke $count_G$ überschreitet.

*[Ref: Scope Global]*

$$\frac{ide \notin E' \quad |E'| > count_G}{Ctxt(E_G = (E,\, ide,\, E')) \vdash \mathbf{Ref}(ide) \Leftarrow -(|E| + 1)}$$

Ein referenzierender Bezeichner *ide* wird gebunden. Er wird durch einen Cardelli-Index $i < 0$ ersetzt, der eine globale Bindung darstellt, wenn sein de Bruijn-Index die Schranke $count_G$ überschreiten würde.

*[Label: Scope]*

$$\frac{}{Ctxt \vdash \mathbf{Label}(ide)}$$

Ein selektierender Bezeichner *ide* kann nicht gebunden werden.

### B.3.2 Typen

Dieser Abschnitt beschreibt, wie der Kontext $Ctxt$ durch Typausdrücke propagiert wird.

*[Type: Expand Syntax Extension]*

$$\frac{Ctxt(expMode = expanded) \vdash Typ}{Ctxt(expMode = normal) \vdash \mathbf{Exp}(Typ)}$$

*[Type: Expand Placeholder]*

$$\frac{Ctxt(expMode = normal) \vdash Typ}{Ctxt(expMode = expanded) \vdash \mathbf{Exp}(Typ)}$$

*[Type: Placeholder]* *[Type: Nok]* *[Type: Ok]*

$$\frac{}{Ctxt \vdash \mathbf{Placeholder}(ide)} \qquad \frac{}{Ctxt \vdash \mathbf{Nok}} \qquad \frac{}{Ctxt \vdash \mathbf{Ok}}$$

*[Type: Basetype]*

$$\frac{}{Ctxt \vdash \textbf{Bool}, \textbf{Int}, \ldots}$$

*[Type: Field Selection]*

$$\frac{Ctxt \vdash Val \quad Ctxt \vdash Label}{Ctxt \vdash Val.Label}$$

[Type: Ref] wird implizit in Abschnitt B.3.1 behandelt.

*[Type: Fun]*

$$\frac{Ctxt \vdash Sigs :: c \quad Ctxt(c) \vdash Typ}{Ctxt \vdash \textbf{Fun}(Sigs) : Typ}$$

*[Type: Arr]*

$$\frac{Ctxt \vdash Typ}{Ctxt \vdash \textbf{Arr}(Typ)}$$

*[Type: Tup]*

$$\frac{Ctxt \vdash Sigs :: c}{Ctxt \vdash \textbf{Tup}(Sigs)}$$

*[Type: Var]*

$$\frac{Ctxt \vdash Typ}{Ctxt \vdash \textbf{Var}(Typ)}$$

*[Type: Oper]*

$$\frac{Ctxt \vdash Sigs :: c \quad Ctxt(c) \vdash Typ}{Ctxt \vdash \textbf{Oper}(Sigs)Typ}$$

*[Type: Apply]*

$$\frac{Ctxt \vdash Typ \quad Ctxt \vdash Bnds :: c}{Ctxt \vdash Typ(Bnds)}$$

### B.3.3 Werte

Dieser Abschnitt beschreibt, wie der Kontext $Ctxt$ durch Wertausdrücke propagiert wird.

*[Value: Expand Syntax Extension]*

$$\frac{Ctxt(expMode = expanded) \vdash Val}{Ctxt(expMode = normal) \vdash \textbf{Exp}(Val)}$$

*[Value: Expand Placeholder]*

$$\frac{Ctxt(expMode = normal) \vdash Val}{Ctxt(expMode = expanded) \vdash \textbf{Exp}(Val)}$$

*[Value: Placeholder]*

$$\frac{}{Ctxt \vdash \textbf{Placeholder}(ide)}$$

*[Value: ok]*

$$\frac{}{Ctxt \vdash \textbf{ok}}$$

*[Value: Literals]*

$$\frac{}{Ctxt \vdash int, real, \ldots}$$

*[Val: Field Selection]*

$$\frac{Ctxt \vdash Val \quad Ctxt \vdash Label}{Ctxt \vdash Val.Label}$$

[Value: Ref] wird implizit in Abschnitt B.3.1 behandelt.

*[Value: Abstraction]*

$$\frac{Ctxt \vdash Sigs :: c \quad Ctxt(c) \vdash Val}{Ctxt \vdash \textbf{fun}(Sigs)Val}$$

*[Value: Application]*

$$\frac{Ctxt \vdash Val \quad Ctxt \vdash Bnds :: c}{Ctxt \vdash Val(Bnds)}$$

*[Value: Array]*

$$\frac{Ctxt \vdash Bnds :: c}{Ctxt \vdash \textbf{arr}(Bnds)}$$

*[Value: Element Selection]*

$$\frac{Ctxt \vdash Val \quad Ctxt \vdash Val'}{Ctxt \vdash Val[Val']}$$

*[Value: Tuple]*

$$\frac{Ctxt \vdash Bnds :: c}{Ctxt \vdash \textbf{tup}(Bnds)}$$

*[Value: Sequence]*

$$\frac{Ctxt \vdash Bnds :: c}{Ctxt \vdash \mathbf{seq}(Bnds)}$$

*[Value: Conditional]*

$$\frac{Ctxt \vdash Val \quad Ctxt \vdash Bnds :: c \quad Ctxt \vdash Bnds' :: c'}{Ctxt \vdash \mathbf{if}(Val, Bnds, Bnds')}$$

*[Value: Loop]*

$$\frac{Ctxt \vdash Bnds :: c}{Ctxt \vdash \mathbf{loop}(Bnds)}$$

*[Value: Exit]*

$$\frac{}{Ctxt \vdash \mathbf{exit}}$$

### B.3.4 Signaturen

Dieser Abschnitt beschreibt, wie der Kontext $Ctxt$ durch Signaturen propagiert und wie die durch Signaturen erzeugten Kontextveränderungen $c$ zusammengesetzt werden.

*[Sigs: Empty]*

$$\frac{}{Ctxt \vdash \oslash :: Id}$$

*[Sigs: Cons]*

$$\frac{Ctxt \vdash Sigs :: c \quad Ctxt(c) \vdash Sig :: c'}{Ctxt \vdash (Sigs, Sig) :: (c, c')}$$

*[Sigs: Expand Syntax Extension]*

$$\frac{Ctxt(expMode = expanded) \vdash Sigs :: c}{Ctxt(expMode = normal) \vdash \mathbf{Exp}(Sigs) :: c}$$

*[Sigs: Expand Placeholder]*

$$\frac{Ctxt(expMode = normal) \vdash Sigs :: c}{Ctxt(expMode = expanded) \vdash \mathbf{Exp}(Sigs) :: c}$$

*[Sig: Type Signature]*

$$\frac{Ctxt \vdash Def :: c \quad Ctxt \vdash Typ}{Ctxt \vdash Def <: Typ :: c}$$

*[Sig: Value Signature]*

$$\frac{Ctxt \vdash Def :: c \quad Ctxt \vdash Typ}{Ctxt \vdash Def : Typ :: c}$$

*[Sig: Placeholder]*

$$\frac{}{Ctxt \vdash \mathbf{Placeholder}(ide) :: Id}$$

### B.3.5 Bindungen

Dieser Abschnitt beschreibt, wie der Kontext $Ctxt$ durch Bindungen propagiert und wie die durch Bindungen erzeugten Kontextveränderungen $c$ zusammengesetzt werden.

*[Bnds: Empty]*

$$\frac{}{Ctxt \vdash \oslash :: Id}$$

*[Bnds: Cons]*

$$\frac{Ctxt \vdash Bnds :: c \quad Ctxt(c) \vdash Bnd :: c'}{Ctxt \vdash (Bnds, Bnd) :: (c, c')}$$

*[Bnds: Expand Syntax Extension]*

$$\frac{Ctxt(expMode = expanded) \vdash Bnds :: c}{Ctxt(expMode = normal) \vdash \mathbf{Exp}(Bnds) :: c}$$

*[Bnds: Expand Placeholder]*

$$\frac{Ctxt(expMode = normal) \vdash Bnds :: c}{Ctxt(expMode = expanded) \vdash \mathbf{Exp}(Bnds) :: c}$$

*[Bnd: Type Binding]*

$$\frac{Ctxt \vdash Def :: c \quad Ctxt \vdash Typ}{Ctxt \vdash Def = Typ :: c}$$

*[Bnd: Value Binding]*

$$\frac{Ctxt \vdash Def :: c \quad Ctxt \vdash Val}{Ctxt \vdash Def = Val :: c}$$

*[Bnd: Variable Binding]*

$$\frac{Ctxt \vdash Def :: c \quad Ctxt \vdash Val}{Ctxt \vdash \mathbf{var}\ Def = Val :: c}$$

*[Bnd: Placeholder]*

$$\frac{}{Ctxt \vdash \mathbf{Placeholder}(ide) :: Id}$$

# Literaturverzeichnis


*Abadi et al. 90:* M. Abadi, L. Cardelli, P.-L. Curien, und J.-J. Lévy. „Explicit Substitutions". In: *Proceedings of the Seventeenth ACM Symposium on Principles of Programming Languages*, Seite 31–46, 1990.

*Abiteboul et al. 93:* S. Abiteboul, S. Cluet, und T. Milo. „Querying and Updating the File". Technischer Bericht, I.N.R.I.A., 1993.

*Aho et al. 88:* A. V. Aho, R. Sethi, und J. D. Ullman. *Compilerbau.* Addison-Wesley, 1988.

*Albano et al. 85:* A. Albano, L. Cardelli, und Orsini R. „Galileo: A Strongly-Typed, Interactive Conceptual Language". *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

*Alber 71:* K. Alber. „Translator Functions for Compiler Description". In: *1. GI-Fachtagung Programmiersprachen*, Seite 60–78, 1971.

*Atkinson et al. 81:* M. P. Atkinson, K. J. Chisholm, und W. P. Cockshott. „PS-algol: An Algol with a Persistent Heap". *ACM SIGPLAN Notices*, 17(7), 1981.

*Atkinson, Bunemann 87:* M. P. Atkinson und P. Bunemann. „Types and Persistence in Database Programming Languages". *ACM Computing Surveys*, 19(2):105–190, 1987.

*Balzert 82:* H. Balzert. *Die Entwicklung von Software-Systemen.* Bibliographisches Institut, 1982.

*Bawden, Rees 88:* A. Bawden und J. Rees. „Syntactic Closures". Technischer Bericht, MIT, 1988.

*Blaser et al. 87:* A. Blaser, M. Jarke, H. Lehmann, und G. Müller. „Datenbanksprachen und Datenbankbenutzung". In: Lockemann und Schmidt [87], Kapitel 6.

*Brodie, Schmidt 82:* M. Brodie und J. W. Schmidt. „Final Report of the ANSI/X3/SPARC DBS-SG Relational Database Task Group". *ACM SIGMOD Record*, 12(4):i, 1982.

*Burshteyn 90:* B. Burshteyn. „On the Modification of the Formal Grammar at Parse Time". *ACM SIGPLAN Notices*, 25(5):117–123, 1990.

*Cabasino et al. 92:* S. Cabasino, P. S. Paolucci, und G. M. Todesco. „Dynamic Parsers and Evolving Grammars". *ACM SIGPLAN Notices*, 27(11):39–48, 1992.

*Cardelli 91:* L. Cardelli. „F-sub, the System". Technischer Bericht, DEC SRC, 1991.

*Cheatham 66:* T. E. Cheatham, Jr. „The Introduction of Definitional Facilities into Higher Level Programming Languages". In: *AFIPS Proceedings of the Joint Computer Conference*, Band 29, Seite 623–637, 1966.

*Christensen, Shaw 69:* C. Christensen und C. J. Shaw, Hrsg. *Proceedings of the Extensible Languages Symposium*, August 1969.

*Christiansen 90:* H. Christiansen. „A Survey of Adaptable Grammars". *ACM SIGPLAN Notices*, 25(11):25–44, 1990.

*Clinger, Rees 91:* W. Clinger und J. Rees. „Macros that Work". In: *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, Seite 155–162, 1991.

*CODASYL 78:* CODASYL. „Data Description Language Commitee Report". *Information Systems*, 3(4):247–320, 1978.

*Cordy et al. 91:* J. R. Cordy, C. D. Halpern, und E. Promislow. „TXL: A Rapid Prototype System for Programming Language Dialects". *Computer Languages*, 16(1):97–107, 1991.

*Damas, Milner 82:* L. Damas und R. Milner. „Principal Type-schemes for Functional Programs". In: *Proc. 9th ACM Symposium on Principles of Programming Languages*, Seite 207–212, 1982.

*Date 89:* C. J. Date. *A Guide to the SQL Standard.* Addison-Wesley, 2. Auflage, 1989.

*de Bruijn 72:* N. G. de Bruijn. „Lambda-calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem". *Indag. Math.*, 34(5):381–392, 1972.

*Dearle et al. 89:* A. Dearle, R. Connor, F. Brown, und R. Morrison. „Napier88 – A Database Programming Language?". In: *Proceedings of the Second International Workshop on Database Programming Languages, Salishan, Oregon*, 1989.

*Dershowitz et al. 91:* N. Dershowitz, J.-P. Jouannaud, und J. W. Klop. „Open Problems in Rewriting". In: *Rewriting Techniques and Applications*, Seite 445–456, 1991.

*Gries 76:* D. Gries. „Some Comments on Programming Language Design". In: *4. GI-Fachtagung Programmiersprachen*, Seite 235–252, 1976.

*Grosch, Emmelmann 90:* J. Grosch und H. Emmelmann. „A Tool Box for Compiler Construction". In: *Compiler-Compilers*, Seite 106–116, 1990.

*Gyimóthy et al. 88:* T. Gyimóthy, T. Horváth, F. Kocsis, und J. Toczki. „Incremental Algorithms in PROF-LP". In: *Compiler Compilers and High Speed Compilation.* Springer, 1988.

*Heering et al. 89:* J. Heering, P. Kint, und J. Rekers. „Incremental Generation of Parsers". *ACM SIGPLAN Notices*, 24(7):179–191, 1989.

*Hennessy 90:* M. Hennessy. *The Semantics of Programming Languages.* John Wiley & Sons, 1990.

*Horowitz 84:* E. Horowitz. *Fundamentals of Programming Languages.* Springer, 2. Auflage, 1984.

*Horspool 88:* R. N. Horspool. „ILALR: An Incremental Generator of LALR(1) Parsers". In: *Compiler Compilers and High Speed Compilation.* Springer, 1988.

*Ipser 92:* E. A. Ipser, Jr. „Exploratory Language Design". *ACM SIGPLAN Notices*, 27(4):41–50, 1992.

*Johnson 75:* S. C. Johnson. „Yacc: Yet Another Compiler-Compiler". Technischer Bericht, Bell Laboratories, 1975.

*Kernighan, Ritchie 90:* B. W. Kernighan und D. M. Ritchie. *Programmieren in C.* Hanser, 2. Auflage, 1990.

*Kirby 92:* G. N. C. Kirby. „Persistent Programming with Strongly Typed Linguistic Reflection". Technischer Bericht, University of St. Andrews, 1992.

*Knuth 65:* D. E. Knuth. „On the Translation of Languages from Left to Right". *Information and Control*, 8:607–639, 1965.

*Knuth 71:* D. E. Knuth. „Top-Down Syntax Analysis". *Acta Informatica*, 1:79–110, 1971.

*Kohlbecker et al. 86:* E. E. Kohlbecker, D. P. Friedman, M. Felleisen, und B. Duba. „Hygienic Macro Expansion". In: *Conference on Lisp and Functional Programming*, Seite 151–159, 1986.

*Kohlbecker 86:* E. E. Kohlbecker. *Syntactic Extensions in the Programming Language LISP.* Dissertation, Indiana University, 1986.

*Koskimies 88:* K. Koskimies. „Software Engineering Aspects in Language Implementation". In: *Compiler Compilers and High Speed Compilation.* Springer, 1988.

*Layzell 85:* P. J. Layzell. „The History of Macro Processors in Programming Language Extensibility". *The Computer Journal*, 28(1):29–33, 1985.

*Leavenworth 66:* B. M. Leavenworth. „Syntax Macros and Extended Translation". *Communications of the ACM*, 9(11):790–793, 1966.

*Lesk, Schmidt 75:* M. E. Lesk und E. Schmidt. „Lex – a Lexical Analyzer Generator". Technischer Bericht, Bell Laboratories, 1975.

*Linnemann 93:* V. Linnemann. „Grammatiken und Syntaxbäume in Datenbanken". In: *Datenbanksysteme in Büro, Technik und Wissenschaft*, 1993.

*Lockemann, Schmidt 87:* P. C. Lockemann und J. W. Schmidt, Hrsg. *Datenbank-Handbuch.* Springer, 1987.

*Matthes, Schmidt 91:* F. Matthes und J. W. Schmidt. „Bulk Types: Built-In or Add-On?". In: *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*, 1991.

*Matthes 88:* F. Matthes. „Typvollständigkeit in Datenbankprogrammiersprachen — DBPL Sprachentwurf und Implementation". Diplomarbeit, Johann Wolfgang Goethe-Universität, Frankfurt, 1988.

*Matthes 93:* F. Matthes. *Persistente Objektsysteme. Integrierte Datenbankentwicklung und Programmerstellung.* Springer, 1993.

*Meyer 88:* B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

*Meyer 90:* B. Meyer. *Introduction to the Theory of Programming Languages.* Prentice Hall, 1990.

*ModISO 91:* ISO/IEC JTC1/SC22/WG13. *Interim Version of the 4th Working Draft Modula-2 Standard*, 1991.

*Nelson 91:* G. Nelson, Hrsg. *Systems programming with Modula-3.* Prentice Hall, 1991.

*Niederée 92:* C. Niederée. „Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung". Diplomarbeit, Universität Hamburg, 1992.

*Plotkin 81:* G. D. Plotkin. „A Structural Appraoch to Operational Semantics". Technischer Bericht, Aarhus University, 1981.

*Rechenberg, Mössenböck 88:* P. Rechenberg und H. Mössenböck. *Ein Compiler-Generator für Mikrocomputer.* Hanser, 2. Auflage, 1988.

*Reps, Teitelbaum 89:* T. W. Reps und T. Teitelbaum. *The Synthesizer Generator.* Springer, 1989.

*Rowe, Shoens 79:* L. A. Rowe und K. A. Shoens. „Data Abstraction, Views and Updates in RIGEL". In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Seite 71–81, 1979.

*Rowe 81:* L. A. Rowe. „Data Abstraction from a Programming Language Viewpoint". *ACM SIGMOD Record*, 11(2):29–35, 1981.

*Sakharov 92:* A. Sakharov. „Macro Processing in High-Level Languages". *ACM SIGPLAN Notices*, 27(11):59–66, 1992.

*Schlageter, Stucky 83:* G. Schlageter und W. Stucky. *Datenbanksysteme: Konzepte und Modelle*. Teubner, 1983.

*Schmidt, Matthes 92:* J.W. Schmidt und F. Matthes. „The Database Programming Language DBPL: Rationale and Report". Technischer Bericht, Universität Hamburg, 1992.

*Schmidt, Matthes 93:* J. W. Schmidt und F. Matthes. „Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems". In: *Proceedings of the International IEEE/RIDE Workshop on Interoperability, Vienna*, 1993.

*Schmidt 87:* J. W. Schmidt. „Datenbankmodelle". In: Lockemann und Schmidt [87], Kapitel 1.

*Schröder, Matthes 92:* G. Schröder und F. Matthes. „Using the Tycoon Compiler Toolkit". Technischer Bericht, Universität Hamburg, 1992.

*Schröder 91:* G. Schröder. „Die Standardisierung von Modula-2". Technischer Bericht, Universität Hamburg, 1991.

*Schuman 71:* S. A. Schuman, Hrsg. *Proceedings of the International Symposium on Extensible Languages*, Dezember 1971.

*Sebesta 89:* R. W. Sebesta. *Concepts of Programming Languages*. Benjamin/Cummings, 1989.

*Sheard 90:* T. Sheard. „Compile-time Reflection: A New Approach to Polymorphism", 1990.

*Solntseff, Yezerski 71:* N. Solntseff und A. Yezerski. „A Survey of Extensible Programming Languages". Technischer Bericht, McMaster University, 1971.

*SQL 87:* ISO. *Standard ISO 9075, Information processing systems — Database language SQL*, 1987.

*Steele 90:* G. L. Steele, Jr. *Common Lisp: The Language*. Digital Press, 2. Auflage, 1990.

*Stemple et al. 91:* D. Stemple, R. Morrison, und Atkinson M. „Type-safe Linguistic Reflection". In: *Database Programming Languages: Bulk Types and Persistent Data*, Seite 357–362, 1991.

*Stemple et al. 92a:* D. Stemple, T. Sheard, und L. Fegaras. „Reflection: A Bridge from Programming to Database Languages". In: *Proceedings HICSS, Hawaii*, Seite 46–55, 1992.

*Stemple et al. 92b:* D. Stemple, R. B. Stanton, T. Sheard, P. Philbrow, R. Morrison, G. N. C. Kirby, L. Fegaras, R. L. Cooper, R. C. H. Connor, M. P. Atkinson, und S. Alagic. „Type-Safe Linguistic Reflection: A Generator Technology". Technischer Bericht, University of St. Andrews, 1992.

*Stemple, Sheard 91:* D. Stemple und T. Sheard. „A Recursive Base for Database Programming Primitives". In: *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, 1991.

*Stonebraker et al. 76:* M. Stonebraker, E. Wong, P. Kreps, und G. Held. „The Design and Implementation of INGRES". *ACM Transactions on Database Systems*, 1(3):189–222, 1976.

*Stonebraker, Rowe 86:* M. Stonebraker und L. A. Rowe. „The Design of POSTGRES". In: *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Washington, D. C.*, Seite 340–355, 1986.

*Stroustrup 92:* B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2. Auflage, 1992.

*Tofte 90:* M. Tofte. *Compiler Generators.* Springer, 1990.

*TRPL 90: A User's Guide to TRPL: A Compile-time Reflective Programming Language*, 1990.

*Waite, Goos 85:* W. M. Waite und G. Goos. *Compiler Construction.* Springer, 1985.

*Wasserman et al. 81:* A. L. Wasserman, D. D. Sheretz, und M. L. Kerstin. „Revised Report on the Programming Language PLAIN". *ACM SIGPLAN Notices*, 16(5):59–80, 1981.

*Wasserman 75:* A. I. Wasserman. „Issues in Programming Language Design — an Overview". *ACM SIGPLAN Notices*, 10(7):10–11, 1975.

*Watt 77:* D. A. Watt. „The Parsing Problem for Affix Grammars". *Acta Informatica*, 8:1–20, 1977.

*Wegbreit 70:* B. Wegbreit. *Studies in Extensible Programming Languages.* Dissertation, Harvard University, 1970.

# Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Hamburg, 17. Februar 1994_____

Gerald Schröder