13. Übungsblatt zu Programmierung

Prof. Gert Smolka, Thorsten Brunklaus
www.ps.uni-sb.de/courses/prog-ws00/

Abgabe: 16. Februar 2001 vor der Vorlesung (per Email)

Allgemeine Hinweise: Die Übungsblätter sollen in Zweiergruppen bearbeitet werden. Die Lösungen der Aufgaben schicken Sie bitte bis Freitag vor der Vorlesung per Email an Ihren Übungsgruppenleiter. Jede Gruppe soll nur eine Lösung einreichen, versehen mit den Namen und den Matrikelnummern der Gruppenmitglieder.

Aufgabe 13.1: Grammatik und Parser für Typen mit Pfeil und Stern (3+9) Sie sollen eine Grammatik und einen Parser für Typen schreiben, die mit `int`, `->`, `*` und Klammern gebildet werden können. Die konkrete Syntax der Typen soll der von Standard ML entsprechen. Dabei steht `->` auf einer höheren Rangstufe als `*` und wird rechtsassoziativ gruppiert. Beispielsweise soll

```
int * int * int -> int * int
```

denselben Typ darstellen wie

```
(int * int * int) -> (int * int)
```

Die mit `*` dargestellten Produkte können zwei-, drei- oder höherstellig sein. Beispielsweise sollen

```
int * int * int
(int * int) * int
int * (int * int)
```

drei verschiedene Typen darstellen.

(a) Geben Sie eine eindeutige kontextfreie Grammatik für die Typen an. Halten Sie sich dabei an die obigen Vorgaben. Behandeln sie `*` als rechtsassoziativen Infixoperator. Zeichen Sie den Dependenzgraphen der Grammatik.

(b) Schreiben Sie einen Parser wie folgt:

```
parse : token list -> ty  (* Error *)


datatype token = INT | ARROW | STAR | LPAR | RPAR


datatype ty = Int
            | Arrow of ty * ty
            | Star of ty list
```

Die mit `*` dargestellten Produkte sollen so wie in den folgenden Beispielen interpretiert werden:

```
int * int * int       ⟹    Star[Int, Int, Int]
(int * int) * int     ⟹    Star[Star[Int, Int], Int]
int * (int * int)     ⟹    Star[Int, Star[Int, Int]]
```

Geben Sie zuerst eine um ein Hilfsnonterminal erweiterte Grammatik an, aus der die Parsingprozeduren abgeleitet werden können. Die Parsingprozedur für das Hilfsnonterminal soll den Typ

```
token list -> ty list * token list
```

haben. Zeichen Sie den Dependenzgraphen der erweiterten Grammatik.

Aufgabe 13.2: Generatoren (2+2+2+3) Ein Generator für eine Folge $x_1, x_2, \ldots$ von Werten eines Typs t ist eine Prozedur `unit` $\rightarrow t$, die beim n-ten Aufruf x_n liefert.

(a) Schreiben Sie einen Generator `nextSquare` für die Folge $0, 1, 4, 9, \ldots$ der Quadratzahlen.

(b) Schreiben Sie eine Prozedur

```
newNextSquare : unit -> unit -> int
```

die bei jedem Aufruf einen neuen Generator für die Folge der Quadratzahlen liefert.

(c) Schreiben Sie eine Prozedur

```
newGenerator : (int -> 'a) -> unit -> 'a
```

die zu einer Prozedur f einen Generator für die Folge

$$f(0),\ f(1),\ f(2),\ \ldots$$

liefert.

(d) Schreiben Sie eine Prozedur

```
newNextPrime : unit -> unit -> int
```

die bei jedem Aufruf einen neuen Generator für die Folge $2, 3, 5, 7, \ldots$ der Primzahlen liefert.

Aufgabe 13.3: Imperative Schlangen (7) Schreiben Sie eine Struktur, die imperative Schlangen gemäß der folgenden Spezifikation realisiert:

```
type 'a queue
val queue  : unit -> 'a queue
val insert : 'a * 'a queue -> unit
val head   : 'a queue -> 'a         (* Empty *)
val remove : 'a queue -> unit       (* Empty *)
val empty  : 'a queue -> bool
```

Die Operation `queue` liefert eine neue Schlange, die noch keine Einträge enthält. Die Operation `insert` trägt einen Wert in eine Schlange ein. Die Operation `head` liefert den ältesten Eintrag in einer Schlange. Die Operation `remove` entfernt den ältesten Eintrag aus einer Schlange. Die Operation `empty` testet, ob eine Schlange Einträge enthält.

Aufgabe 13.4: Rotieren von Reihungen (4+4) Sie sollen eine Prozedur

```
rotate : 'a array -> unit
```

schreiben, die die Komponenten einer nichtleeren Reihung um eine Position nach rechts schiebt. Dabei soll die letzte Komponente an die Stelle der ersten Komponente rücken.

(a) Schreiben Sie `rotate` mithilfe einer iterativ rekursiven Hilfsprozedur `rotate'`.

(b) Schreiben Sie `rotate` mithilfe einer Schleife.

Aufgabe 13.5: Binäre Suche (8) Eine Reihung des Types `int array` heißt sortiert, wenn ihre Komponenten aufsteigend sortiert sind. Schreiben Sie eine Prozedur

```
member : int array -> int -> bool
```

die zu einer sortierten Reihung a und einer Zahl x entscheidet, ob eine Komponente von a den Wert x hat. Für Reihungen der Länge n soll `member` die Laufzeit $O(\log n)$ haben.
Die logarithmische Laufzeit läßt sich mithilfe von binäre Suche erreichen, eine Technik, die wir bereits im Zusammenhang mit Rot-Schwarz-Bäumen kennengelernt haben. Dazu stellen wir uns die sortierte Reihung als eine Folge vor, die von links nach rechts läuft und gehen wie folgt vor:

(a) Bestimme einen Index m, der etwa in der Mitte der Reihung liegt.

(b) Wenn x der Wert der m-ten Komponente ist, liefere `true`.

(c) Wenn x kleiner als der Wert der m-ten Komponente ist, suche in der linken Teilreihung weiter.

(d) Wenn x größer als der Wert der m-ten Komponente ist, suche in der rechten Teilreihung weiter.

(e) Wenn die zu durchsuchende Teilreihung leer ist, liefere `false`.

Eine Teilreihung können wir durch das Paar aus ihrem kleinsten und größten Index darstellen.

Aufgabe 13.6: Statische Semantik von F mit Referenzen (6) Sei F, wie in Abschnitt 13.8 gezeigt, um Referenzen erweitert. In Abschnitt 11.2 wird die statische Semantik von F mithilfe von Inferenzregeln definiert. Geben Sie die für Referenzen zusätzlich erforderlichen Regeln an.