4. Übungsblatt „Programmiersprachen“ SS 99

Leif Kornstaedt, Prof. Gert Smolka

Ausgabe: 7. Mai 1999 **Abgabe:** 19. Mai 1999

Diesmal sollen Sie etwas komplexere Programmieraufgaben in SML lösen. Dabei bekommen Sie ein Gefühl für die Sprache und üben den Umgang mit Konstruktortypen und Ausnahmen.

Da die Vorlesung am 14. Mai ausfällt, ist die Bearbeitungszeit für dieses Übungsblatt bis zum 19. Mai ausgedehnt worden. Desweiteren gibt es diesmal 49 Punkte statt 33; es zählt also als anderthalb Blätter.

Aufgabe IV.1: Binäre Bäume (5 Punkte)

Gegeben sei folgende Darstellung für Bäume:

```
datatype 'a tree =
    Node of 'a * 'a tree * 'a tree
  | Empty
```

Schreiben Sie eine Prozedur `isBinary: 'a tree -> bool`, die testet, ob ihr Argument ein binärer Baum ist.

Aufgabe IV.2: Freie Variablen für F1 (10 Punkte)

Folgende SML-Deklarationen definieren einen Datentyp zur Repräsentation der abstrakten Syntax von F1. Dabei werden Bezeichner durch ganze Zahlen dargestellt. (Auf der Web-Seite zur Vorlesung sind diese Deklarationen als SML-Datei verfügbar.)

```
type id = string

datatype dec =                    (* d ::= *)
    ValDec of id * exp              (* val x = a *)
  | FunDec of id * id * exp         (* fun x1 x2 = a *)
and exp =                         (* a ::= *)
    ConstExp of const               (* c *)
  | VarExp of id                    (* x *)
  | IfExp of exp * exp * exp        (* if a1 then a2 else a3 *)
  | AppExp of exp * exp             (* a1 a2 *)
  | FnExp of id * exp               (* fn x => a *)
  | LetExp of dec * exp             (* let d in a end *)
and const =                       (* c ::= *)
    IntConst of int                 (* 0 | 1 | ~1 | 2 | ~2 | ... *)
  | BoolConst of bool               (* true | false *)

type program = dec list
```

Schreiben Sie eine Prozedur `freeVars: program -> id list`, die die freien Variablen eines Programms berechnet. Die Ergebnisliste soll keine Variable doppelt enthalten.

Aufgabe IV.3: Statische Semantik von explizit getyptem F1 ($5 + 5 + 12$ Punkte)

a. Definieren Sie einen Datentyp `ty` zur Repräsentation von F1-Typen.

b. Erweitern Sie den in Aufgabe IV.2 gegebenen Datentyp so, dass er Programme in explizit getypem F1 darstellt. Zur Erinnerung: In explizit getyptem F1 muß der Typ von Prozedurdeklarationen und Abstraktionen wie folgt annotiert werden.

   fun x_1 (x_2: τ_1): τ_2 = a

   fn x: τ => a

c. Schreiben Sie eine Prozedur `typecheck: program * ty env -> bool`, die testet, ob ein explizit getyptes F1-Programm in bezug auf eine Typumgebung typkorrekt ist. Umgebungen können Sie beispielsweise durch folgenden Typ darstellen:

```
type 'a env = (id * 'a) list
```

Aufgabe IV.4: Parser für Infix-Grammatiken (12 Punkte)

Die Aufgabe besteht darin, einen Parser für Infixgrammatiken zu schreiben. Dabei sollen Operatoren Zeichenreihen und Atome ganze Zahlen sein. Zunächst muss man sich geeignete Datenstrukturen für die Grammatik, das Eingabewort und den Ableitungsbaum überlegen. Danach braucht man einen Algorithmus für das Parsen. Dieser soll lineare Laufzeit haben.

Die Aufgabe ist nicht einfach, obwohl sie mit einem Programm von weniger als 40 Zeilen gelöst werden kann. Wir geben Ihnen einige Hilfestellungen, die Sie aber erst lesen sollten, nachdem Sie versucht haben, die Aufgabe ohne Hilfe zu lösen.

Die Grammatik repräsentieren wir wie folgt:

```
type precedence = int
datatype associativity = L | R
exception Error
type grammar = string -> precedence * associativity   (* Error *)
```

Hier ist ein Beispiel für eine Grammatik:

```
fun grammar "+" = (5, L)
  | grammar "-" = (5, L)
  | grammar "*" = (6, L)
  | grammar "<" = (4, L)
  | grammar ">" = (4, R)
  | grammar _   = raise Error
```

Das Eingabewort repräsentieren wir als `token list` mit

```
datatype token =
    A of int      (* Atom     *)
  | O of string   (* Operator *)
```

Ableitungsbäume sollen die Form

```
datatype tree =
    At of int
  | Op of string * tree * tree
```

haben. Der Parser soll durch eine Prozedur

```
parse: grammar -> token list -> tree option
```

realisiert werden.

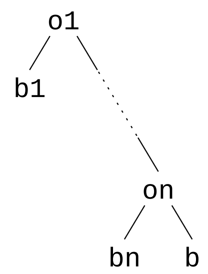
Der Parser soll mit Zuständen der Form

```
type state = (tree * string) list * tree * token list
```

arbeiten. Die dritte Komponente enthält die bisher noch nicht verarbeiteten Token der Eingabeliste. Die erste Komponente wird als Keller benutzt, wobei das oberste Element das erste Listenelement ist. Ein Kellereintrag (b, o) repräsentiert einen Teilbaum mit dem Operator o als Wurzelmarkierung, b als linkem Unterbaum und einem noch fehlenden rechten Unterbaum. Insgesamt sollte man sich die erste und zweite Komponente eines Zustands

```
((bn, on)::···::(b1, o1)::nil, b, ts)
```

als den noch erweiterbaren Baum



vorstellen.

Wenn `A n::ts` das Eingabewort ist, beginnt der Parser mit dem Zustand

```
(nil, At n, ts)
```

Die Zustandsübergänge des Parsers erfolgen mit zwei Regeln Push und Pop. (Für Kenner: Der Parser arbeitet ähnlich wie ein LR-Parser, wobei Push der Operation Shift und Pop der Operation Reduce entspricht.) Wenn das Eingabewort zulässig ist, endet der Parser schließlich mit einem Zustand

```
(nil, b, nil)
```

dessen Komponente `b` der Ableitungsbaum des Eingabeworts ist.

Die Regel sind durch

```
fun push (es, b, O o::A a:: tr) = ((b, o)::es, At a, tr)
  | push _ = raise Error

fun pop ((b, o)::er, b', ts) = (er, Op (o, b, b'), ts)
  | pop _  = raise Error
```

gegeben, wobei die Regeln nur angewendet werden dürfen, wenn sie die Zustandsinvariante

```
invariant: state -> bool
```

erhalten. Diese ist wie folgt definiert:

```
fun invariant (es, b, ts) =
    admissible (fold (es, b)) andalso invariant' (b, ts)

fun invariant' (Op (o, _, _), O o'::tr) = ok (o', o, L)
  | invariant' _ = true

fun fold (nil, b) = b
  | fold ((b, o)::es, b') = fold (es, Op (o, b, b'))
```

Die Prädikate

```
admissible: tree -> bool

ok: string * string * associativity -> bool
```

sind wie folgt definiert:

- `admissible b = true` genau dann, wenn `b` ein Ableitungsbaum gemäß der Grammatik ist.
- `ok (o, o', a) = true` genau dann, wenn die Grammatik erlaubt, dass der Operator `o'` unter dem Operator `o` auf der Seite `a` vorkommt.

Testen Sie ihren Parser mit der obigen Grammatik und den Eingabewörtern

```
[A 1, O "+", A 2, O "*", A 3, O "*", A 4, O "+", A 5]
[A 1, O ">", A 2, O "<", A 3]
```