UNIVERSITÄT DES SAARLANDES
FR 6.2 – Informatik
Harald Ganzinger
Andreas Podelski
Uwe Waldmann

Klausur zur Vorlesung „Programmiersprachen“ (Sommersemester 2001)

Name, Vorname : ....

Matr.Nr. : ....

- Legen Sie Personal– und Studentenausweis offen vor sich hin.
- Schreiben Sie auf *jedes* zu benutzende Blatt (inklusive Schmierpapier) *zuerst* Ihren Namen und Ihre Matrikelnummer.

Bei der Abgabe:

- Kreuzen Sie jede bearbeitete Aufgabe in untenstehender Tabelle an.
- Heften Sie die Klausur und *alle* Ihre Lösungsblätter in der linken oberen Ecke zusammen. Bei der Aufsicht steht ein Klammerhefter zur Verfügung.
- Aus Versehen mitgenommene Lösungen können nicht gewertet werden.

Unterschreiben Sie hier: **Viel Erfolg!**

....

Aufgabe	1a	1b	1c	2	3	4	5	Σ
bearbeitet								
Punkte								

Aufgabe 1 *(SML, Interpreter)* (4 + 4 + 6 = 14 Punkte)

Wir stellen hier eine einfache Programmiersprache für loop-Programme vor. Die konkrete Syntax dieser Sprache ist durch die folgende Grammatik gegeben:

```
<stat> ::= <empty>           // leere Anweisung.
         | <stat> ; <stat>   // Sequenz.
         | <var> := <exp>    // Zuweisung.
         | ifpos             // bedingte Anweisung: falls <exp>
             <exp>           // positiv ist, wird der then-Teil
           then              // ausgeführt, sonst der else-Teil.
             <stat>
           else
             <stat>
           endif
         | loop              // Schleife: falls <exp> positiv ist,
             <exp>           // so wird <stat> <exp>-mal ausgeführt,
           do                // anderenfalls wird <stat> nicht
             <stat>          // ausgeführt.
           endloop

<exp>  ::= <intconst>        // ganzzahlige Konstante.
         | <var>             // Variable.
         | <exp> + <exp>     // Summe.
         | <exp> * <exp>     // Produkt.
         | <exp> - <exp>     // Differenz.
```

Beachten Sie folgende Unterschiede zu der in der Vorlesung beschriebenen Sprache für while-Programme:

- Es gibt nur einen einzigen Typ (ganze Zahlen).
- Es gibt binäre, aber keine unären Operatoren.
- Statt einer if-Anweisung gibt es eine ifpos-Anweisung; **ifpos** e **then** s_1 **else** s_2 **endif** entspricht **if** $e > 0$ **then** s_1 **else** s_2 **endif**.
- Statt einer while-Anweisung gibt es eine loop-Anweisung. Der Ausdruck <exp> wird bei einer loop-Anweisung loop <exp> do <stat> endloop nur *einmal* ausgewertet, die Anzahl der Schleifendurchläufe steht also stets zu Beginn der Schleife fest.
- Es gibt keine Variablendeklarationen. Variable sind automatisch deklariert, aber nicht automatisch initialisiert: Erst durch eine Zuweisung erhalten sie einen Wert.

Teil (a)

Geben Sie eine abstrakte Syntax für diese Sprache in Form von SML-Datentypen `stat` und `exp` an; codieren Sie dabei Variable durch Strings.

Teil (b)

Schreiben Sie einen Auswerter `evalExp` für die Ausdrücke der Sprache. Die Funktion `evalExp` hat den Typ `exp -> (string * int) list -> int`; ihr erstes Argument ist ein Ausdruck, ihr zweites Argument eine Umgebung, d. h., eine Liste von Paaren, bestehend jeweils aus einem Variablennamen und einem Wert. Falls allen Variablen, die in dem Ausdruck vorkommen, in der Umgebung ein Wert zugeordnet ist, dann liefert `evalExp` den Wert des Ausdrucks bezüglich der Umgebung; anderenfalls liefert `evalExp` eine Exception `non_initialized`.

Teil (c)

Schreiben Sie eine Interpreter `evalStat` für die Sprache. Die Funktion `evalStat` hat den Typ `stat -> (string * int) list -> (string * int) list`; ihr erstes Argument ist eine Anweisung s, ihr zweites Argument eine Umgebung u. Das Ergebnis von `evalStat` ist eine neue Umgebung, und zwar diejenige, die sich durch Ausführen von s in der Anfangsumgebung u ergibt. Sofern während der Ausführung von s auf eine Variable lesend zugegriffen wird, die in diesem Moment noch keinen Wert besitzt, liefert `evalStat` eine Exception `non_initialized`.

Aufgabe 2 *(MML, Typinferenz)* (6 Punkte)

Geben Sie einen Typ τ an, so daß die Aussage

$$5 \;:\; \texttt{int} \vdash \texttt{let}\ i = \lambda x.5\ \texttt{in}\ (i\ \ i)\ \ :\ \tau$$

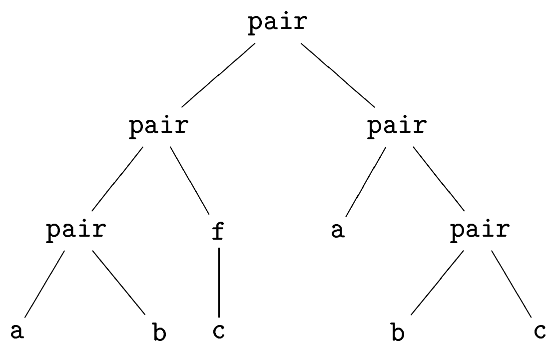
mittels des Damas/Milner-Typinferenzsystems für MML ableitbar ist. Wie sieht die Ableitung aus?

Aufgabe 3 *(Prolog)* (12 Punkte)

Wir definieren Binärbäume (ohne Knotenbeschriftung) in Prolog wie folgt:

- Jeder Grundterm, dessen Wurzelsymbol nicht der zweistellige Operator `pair` ist, ist ein Blatt.
- Jedes Blatt ist ein Baum.
- Wenn t_1 und t_2 Bäume sind, dann ist `pair(`t_1`,`t_2`)` ein Baum.

Beispiel: Der Baum



hat die Blätter `a`, `b`, `c` und `f(c)`; die Blätter `a` und `b` kommen je zweimal vor, die anderen Blätter je einmal. Implementieren Sie ein Prolog-Prädikat `maxfreqleaf(T,L)`, das dann erfüllt ist, wenn `T` ein Binärbaum und `L` ein Blatt maximaler Häufigkeit in `T` ist (d. h., ein Blatt in `T`, so daß kein anderes Blatt in `T` häufiger vorkommt). Die Anfrage `?- maxfreqleaf(`t`,X)`, wobei t der oben angegebene Baum ist, sollte also die beiden Lösungen `X = a` und `X = b` liefern. Sie können davon ausgehen, daß in der Anfrage das erste Argument ein Grundterm ist.

Hinweis: Es gibt verschiedene Ansätze, um diese Aufgabe zu lösen. Je nachdem, welchen Ansatz Sie wählen, sind möglicherweise die vordefinierten Prolog-Prädikate `@<`, `@=<`, `@>` und `@>=` nützlich. Mit diesen lassen sich beliebige Terme (Atome, Zahlen, zusammengesetzte Terme, Strings) miteinander vergleichen. Die durch diese Prädikate definierte Ordnung ist total, d. h., für je zwei verschiedene Terme t_1 und t_2 gilt entweder t_1 `@<` t_2 oder t_1 `@>` t_2.

Aufgabe 4 *(Java, Klassenhierarchie)* (8 Punkte)

Gegeben seien die nachfolgenden Java-Klassen:

```
interface I {
  public B iFunc(int n);
}

public class B {
  int ci = 5;
  public B bFunc(I i) {
    return i.iFunc(ci);
  }
}

public class C extends B implements I {
  public B iFunc(int n) {
    return new B();
  }
}
```

Welche der nachfolgenden Codefragmente führen zu einem Compilezeitfehler? Welche führen zu einem Laufzeitfehler? Erklären Sie kurz warum.

```
/* 1 */  I i1 = new I();

/* 2 */  I i2 = new C();

/* 3 */  B b3 = new C();
         C c3 = b3;

/* 4 */  Object o4 = new B();
         I i4 = (C)o4;

/* 5 */  B bx5 = new B();
         B by5 = new C();
         bx5 = bx5.bFunc(by5);

/* 6 */  C c6 = new C();
         Object o6 = c6.bFunc(c6);

/* 7 */  public class E implements I {
           public B iFunc(int n) {
             return new C();
           }
         }

/* 8 */  public class D extends B {
           int k;
           abstract public int dFunc(int n);
         }
```

Aufgabe 5 *(Implementierung zwischen Algebren)* (10 Punkte)

Wir definieren eine Signatur Σ für Stacks über Elementen eines beliebigen festen Typs α, sowie zwei Σ-Algebren, von denen die zweite die erste implementiert. In der ersten Algebra sind Stacks Listen, neue Elemente werden vorne angefügt, alte Elemente werden vorne entfernt. In der zweiten Algebra sind Stacks Paare, bestehend aus einer Liste und einem Zeiger (Index) auf das top-Element des Stacks. Beim Entfernen eines Elements wird hier nur der Zeiger verschoben, das Element selbst bleibt erhalten und wird, wenn ein neues Element angefügt wird, überschrieben.
Zunächst die Signatur $\Sigma = (S, \Omega)$ von Stacks wobei

$$\begin{array}{rcl} S & = & \{\alpha,\ \alpha\ \texttt{stack}\} \\ \Omega & = & \{\texttt{empty} : \alpha\ \texttt{stack}, \\ & & \quad \texttt{push} : \alpha\ \texttt{stack} * \alpha \to \alpha\ \texttt{stack}, \\ & & \quad \texttt{pop} : \alpha\ \texttt{stack} \to \alpha\ \texttt{stack}, \\ & & \quad \texttt{top} : \alpha\ \texttt{stack} \to \alpha\} \end{array}$$

Die Algebra B ist wie folgt definiert:

$$\begin{array}{rcl} B_\alpha & = & \alpha \cup \{\texttt{error}\} \\ B_{\alpha\,\texttt{stack}} & = & \alpha\ \texttt{list} \cup \{\texttt{error}\} \\ \texttt{empty}_B & = & [] \\ \texttt{push}_B([x_n, \ldots, x_1], x) & = & [x, x_n, \ldots, x_1], \quad \text{falls } x_i, x \in \alpha,\ n \geq 0 \\ \texttt{pop}_B([x_n, \ldots, x_1]) & = & [x_{n-1}, \ldots, x_1], \quad \text{falls } x_i \in \alpha,\ n \geq 1 \\ \texttt{top}_B([x_n, \ldots, x_1]) & = & x_n, \quad \text{falls } x_i \in \alpha,\ n \geq 1 \end{array}$$

für alle anderen Argumente ergebe sich `error` als Ergebnis.
Die Algebra A ist wie folgt definiert:

$$\begin{array}{rcl} A_\alpha & = & \alpha \cup \{\texttt{error}\} \\ A_{\alpha\,\texttt{stack}} & = & (\alpha\ \texttt{list} * \texttt{int}) \cup \{\texttt{error}\} \\ \texttt{empty}_A & = & ([], 0) \\ \texttt{push}_A(([x_n, \ldots, x_1], m), x) & = & ([x_n, \ldots, x_{m+2}, x, x_m, \ldots, x_1], m+1), \\ & & \qquad \text{falls } x_i, x \in \alpha,\ n > m \geq 0 \\ \texttt{push}_A(([x_n, \ldots, x_1], n), x) & = & ([x, x_n, \ldots, x_1], n+1), \quad \text{falls } x_i, x \in \alpha,\ n \geq 0 \\ \texttt{pop}_A([x_n, \ldots, x_1], m) & = & ([x_n, \ldots, x_1], m-1), \quad \text{falls } x_i \in \alpha,\ n \geq m \geq 1 \\ \texttt{top}_A([x_n, \ldots, x_1], m) & = & x_m, \quad \text{falls } x_i \in \alpha,\ n \geq m \geq 1 \end{array}$$

für alle anderen Argumente ergebe sich wiederum `error` als Ergebnis.
Geben Sie eine Repräsentationsinvariante $I \subseteq A_{\alpha\,\texttt{stack}}$ und eine Abstraktionsfunktion $a : A_{\alpha\,\texttt{stack}} \to B_{\alpha\,\texttt{stack}}$ an, sowie, falls a partiell ist, den Definitionsbereich von a. Zeigen Sie, daß die Algebra A die Algebra B auf $\alpha\,\texttt{stack}$ implementiert.