Fachrichtung 6.2 — Informatik
Universität des Saarlandes
Tutorenteam der Vorlesung Programmierung 1

Programmierung 1 (Wintersemester 2012/13)

Lösungsblatt 11

(Kapitel 12)

Aufgabe 11.1 (*Offizielle Übungsaufgabe aus dem Wintersemester 11/12*)
Geben Sie Typumgebungen an, für die der Ausdruck *if true then x else y* zulässig beziehungsweise unzulässig ist.

Lösung 11.1:
Der Ausdruck ist beispielsweise zulässig fr die Typumgebung $[x := int, y := int]$, und unzulässig fr die leere Umgebung [].

Aufgabe 11.2 (*Offizielle Übungsaufgabe aus dem Wintersemester 11/12*)
Rekapitulieren Sie, dass die Phrasen in F lediglich mathematische Objekte sind (oder als solche dargestellt werden können). Man kann Konstanten, Operatoren, Typen etc... ähnlich wie die Werte von Konstruktortypen mit Hilfe von Variantennummern und Tupleln darstellen. Hier sind 5 Beispiele fr Ausdrücke mit den korrespondierenden Variantennummertupeln:

true	$\langle 1, \langle 2 \rangle \rangle$
3	$\langle 1, \langle 3, \langle 3 \rangle \rangle \rangle$
$7 \leq 13$	$\langle 3, \langle 4 \rangle, \langle 1, \langle 3, 7 \rangle \rangle, \langle 1, \langle 3, 13 \rangle \rangle \rangle$
if true then 72 *else* 33	$\langle 4, \langle 1, \langle 2 \rangle \rangle, \langle 1, \langle 3, 72 \rangle \rangle, \langle 1, \langle 3, 33 \rangle \rangle \rangle$
x	$\langle 2, \langle x \rangle \rangle$

Schreiben Sie nun folgende Prozeduren:

(a) *compile* : *variant* → *exp* Welche als Eingabe ein Variantennummertupel erhält und den dazu korrespondierenden Ausdruck zurückgibt (siehe Beispiele).

(b) *decompile* : *exp* → *variant* Welche genau die Umkehrfunktion von a) berechnen soll: Als Eingabe erhält man einen Ausdruck e, aus welchem dann ein Variantennummertupel generiert wird.

(*Hinweis:* Wie Sie sicherlich bemerkt haben, ist der Typ *variant* nicht genauer spezifiziert. Das ist keine Willkür der Tutoren, es soll Sie lediglich dazu ermutigen den Typ selbstständig zu erarbeiten. Sie werden wahrscheinlich schnell merken, dass der erste naive Ansatz nicht funktionieren wird. Überlegen Sie, wie Sie geschachtelte Tupel in dieser Form darstellen und verwenden können. Seien Sie dabei kreativ!
Verwenden sie weiterhin Strings anstatt Zahlen um Variablen mit Hilfe von Variantennummertupeln darzustellen!)

Lösung 11.2:
Zunächst sei zu sagen, dass Variantennummertupeln in der Form wie wir sie verwenden sowohl Werte vom Typ *int* als auch Werte vom Typ *String* beinhalten. Aufgrund dieser Tatsache brauchen wir also zunächst einen Datentyp welcher dieses Problem löst:

```
datatype variant' = V of int | S of string
```

Aufgrund der beliebigen Schachtlung der Ausdrücke können wir uns jedoch nicht auf Listen und Tupel beschränken sondern müssen auf markierte Bäume zurückgreifen:

```
datatype 'a ltr = L of 'a * 'a ltr listt type variant = variant' ltr
```

Nun können wir recht einfach die beiden Prozeduren schreiben:

(a)Die Prozedur *compile* ist gegeben durch:

```
fun compilety (L(V 1, nil)) = Bool | compilety (L(V 2, nil)) = Int | compilety (L(V 3, [t1,
t2])) = Arrow (compilety t1, compilety t2) fun compileCon (L(V a, t)) = case (a, t) of (1,
nil) => True | (2, nil) => False | (3, [L(V a, nil)]) => IC a fun compileOpr 1 = Add | compileOpr
2 = Sub | compileOpr 3 = Mul | compileOpr 4 = Leq fun compile (L(V 1, [t])) = Con (compileCon
t) | compile (L(V 2, [L(S s, nil)])) = Id s | compile (L(V 3, [L(V a, nil), t1, t2])) = Opr(compileOpr
a, compile t1, compile t2) | compile (L(V 4, [t1, t2, t3])) = If(compile t1, compile t2,
compile t3) | compile (L(V 5, [L(S s, nil), a, t1])) = Abs (s, compilety a, compile t1) |
compile (L(V 6, [t1, t2])) = App(compile t1, compile t2)
```

(b) Die Prozedur *decompile* ist gegeben durch:

```
fun decompilety Bool = (L(V 1, nil)) | decompilety Int = L(V 2, nil) | decompilety (Arrow(ty1,
ty2)) = L(V 3, [decompilety ty1, decompilety ty2]) fun decompileCon True = L(V 1, nil) |
decompileCon False = L(V 2, nil) fun decompileOpr Add = 1 | decompileOpr Sub = 2 | decompileOpr
Mul = 3 | decompileOpr Leq = 4 fun decompile (Con c) = L(V 1, [decompileCon c]) | decompile
(Id s) = L(V 2, [L(S s, nil)]) | decompile (Opr(opr, e1, e2)) = L(V 3, [L(V (decompileOpr
opr), [decompile e1, decompile e2])]) | decompile (If(e1, e2, e3)) = L(V 4, [decompile e1,
decompile e2, decompile e3]) | decompile (Abs(s, t, e)) = L(V 5, [L(S s, nil), decompilety
t, decompile e]) | decompile (App(e1, e2)) = L(V 6, [decompile e1, decompile e2])
```

Aufgabe 11.3 (*Offizielle Übungsaufgabe aus dem Wintersemester 11/12*)
Erweitern Sie *elab* um 2 unäre Operatoren:

(a) $\sim: \mathbb{Z} \to \mathbb{Z}$, welche das additive Inverse für ganze Zahlen berechnet.

(b) $! : \mathbb{B} \to \mathbb{B}$, welche die boolesche Negation berechnet.

Lösen Sie diese Aufgabe nicht indem sie blind drauf los schreiben und direkt den Code für *elab* erweitern. Auch wenn das in diesem Fall vielleicht funktionieren würde, da die Aufgabe nicht sehr schwer ist, macht es generell immer mehr Sinn strukturierter vorzugehen:

- Erweitern Sie formal die abstrakte Grammatik für die abstrakte Syntax von F. (Siehe S. 243 Abbildung 12.2)
- Erweitern Sie nun die Typdeklarationen für die abstrakte Syntax von F.
- Geben Sie die Inferenzregeln für die statische Semantik der neuen unären Operatoren an.
- Erweitern Sie nun die Deklaration der Prozedur *elab*

Lösung 11.3:
[Offizielle Übungsaufgabe aus dem Wintersemester 11/12] Gehen wir strukturiert vor, so erhalten wir folgende Schritte:

- Wir erweitern die abstrakte Grammatik für die abstrakte Syntax von F wie folgt:

$$o \in UnOpr \quad = \quad \sim \quad | \quad ! \qquad \text{Unäre Operatoren}$$

- Wir erweitern die Typdeklaration vür die abstrakte Syntax von F wie folgt:

```
datatype unopr = Neg | Not ... datatype exp = ... | UnOpr of unopr * exp
```

- Wir erhalten folgende Inferenzregeln:

Snot $\dfrac{T \vdash e : bool}{T \vdash !e : bool}$

Sneg $\dfrac{T \vdash e : int}{T \vdash \sim e : int}$

- Aus den Inferenzregeln folgt die Erweiterung:

```
fun elab f ... | elab f (UnOpr(unopr e)) = case (unopr, elab f e) of (Neg, Int) => Int
| (Not, Bool) => Bool | s=> raise Error ''T UnOpr''
```

Aufgabe 11.4
Betrachten Sie erneut Aufgabe 11.3. Erweitern Sie nun ebenfalls die dynamische Semantik um die entsprechenden unären Operatoren.

Lösung 11.4:
Da die Syntax bereits in Aufgabe 12.5 angepasst wurde, müssen wir lediglich die neuen Inferenzregeln für die dynamische Semantik aufstellen und dementsprechend die Prozedur *eval* erweitern:

- Wir erhalten folgende Inferenzregeln für die dynamische Semantik:

DnotT $\dfrac{V \vdash e \rhd 1}{V \vdash !e \rhd 0}$ **DnotF** $\dfrac{V \vdash e \rhd 1}{V \vdash !e \rhd 0}$

Dneg $\dfrac{V \vdash e \rhd v}{V \vdash \sim e \rhd -v}$

- Mit den Inferenzregeln erhalten wir folgende Erweiterungen in *eval*:

```
fun eval f ... | eval f (UnOpr(unopr, e1)) = case (unopr, eval f e1) of (Not, IV 0) =>
IV 1 | (Not, IV 1) => IV 0 | (Neg, IV n) => IV ( n)
```

Aufgabe 11.5 (*Schriftliche Aufgabe aus dem Wintersemester 11/12*)
Erweitern Sie *elab* und *eval* um wirklich spannende Dinge wie *Tupel, Let-Ausdrücke, Listen* oder gar *Typvariablen*. Diese Aufgabe ist nicht ganz einfach, jedoch sehr lohnenswert: Es gibt keine bessere Methode, um die statische und dynamische Semantik von SML richtig zu verstehen. Damit ist diese Aufgabe auch eine ideale Klausurvorbereitung! Nutzen Sie diese Chance!

Sollten Sie an manchen Stellen in Ihrer Implementierung unlösbare Schwierigkeiten entwicklen, so dokumentieren Sie Ihre Probleme und Ihre verschiedenen Lösungsansätze ausführlich.

Tipp: Wir haben die möglichen Erweiterungen oben nach ansteigender Schwierigkeit sortiert aufgelistet. Im Forum sollten Sie wie immer Fragen stellen und diskutieren.

Aufgabe 11.6 (*Schriftliche Aufgabe aus dem Wintersemester 11/12*)
Die Endklausur der Vorlesung steht schon wieder vor der Tür. Schätzen Sie Ihren gegenwerten Könnensstand realistisch ein. Welche Inhalte der Vorlesung beherrschen Sie schon recht sicher, was müssen Sie noch besser verstehen? Machen Sie sich einen Plan, wie sie sich die kommenden Wochen auf die Klausur vorbereiten wollen.