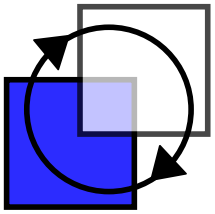
CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Prof. Bernd Finkbeiner, Ph.D.
Jana Hofmann, M.Sc.
Reactive Systems Group



Programmierung 1 (WS 2020/21)
Übungsblatt K

Lesen Sie im Buch Kapitel 13 - 14.3.

Hinweis: Über Aufgaben, die mit 🤔 markiert sind, müssen Sie eventuell etwas länger nachdenken. Falls Ihnen keine Lösung einfällt - kein Grund zur Sorge. Kommen Sie in die Office Hour, unsere Tutor:innen helfen gerne.

Konkrete Syntax

Aufgabe K.1

Im Folgenden sollen Sie einen Lexer `lex : string → int` schreiben, der aus einem String eine Zahl ausliest. Die Zahlen in dem String sind durch Ziffern in Wortdarstellung dargestellt. Die einzelnen Ziffern sind dabei durch mindestens ein Leerzeichen getrennt. Beispielsweise soll `lex ("eins␣sieben␣sechs␣null") = 1760` gelten. Ist der String keine gültige Darstellung einer Zahl, so sollen Sie eine geeignete Ausnahme werfen. Gehen Sie dazu wie folgt vor:

(a) Schreiben Sie eine Prozedur `lex': char list → int list`, die die Ziffern in Wortdarstellung durch Zahlen ersetzt. Orientieren Sie sich dabei an der Prozedur `lex` aus Kapitel 13.

(b) Schreiben Sie eine Prozedur `digitsToInt : int list → int`, die eine Liste von Ziffern zu einer Zahl zusammensetzt.

(c) Schreiben Sie mit Hilfe der ersten beiden Teilaufgaben die Prozedur `lex`.

Aufgabe K.2

Zeichnen Sie die Syntaxbäume für `ty` und die folgenden Wortfolgen:

(a) $int \to (bool \to int)$

(b) $(bool \to bool) \to int$

(c) $int \to (bool \to bool) \to int$

Aufgabe K.3

Zeigen Sie mit einem Beispiel, dass die Grammatik

$$exp ::= x \mid exp\ exp$$

nicht eindeutig ist. Geben Sie eine eindeutige Grammatik an, die die selben Wortfolgen darstellt.

Aufgabe K.4

Die beschriebene Grammatik für die Typen von F liefert für jeden Typ unendlich viele Wortdarstellungen. Sie sollen eindeutige Grammatiken angeben, die für jeden Typ genau eine Wortdarstellung liefern. Dabei soll der Typ $(int \to int) \to int \to int$ jeweils wie folgt darstellt werden: 🤔

(a) vollständig geklammert: $((int \to int) \to (int \to int))$

(b) minimal geklammert: $(int \to int) \to int \to int$

(c) Präfixdarstellung ohne Klammern: $\rightarrow \rightarrow \mathit{int}\ \mathit{int} \rightarrow \mathit{int}\ \mathit{int}$

Aufgabe K.5
Sei die folgende phrasale Syntax für die Typen von F gegeben:

$$\mathit{ty} ::= \mathit{bool} \mid \mathit{int} \mid \text{"->"}\ \mathit{ty}\ \mathit{ty}$$

Die Baumdarstellungen der Typen sollen in SML durch Werte des Typs

```
datatype ty  = Bool | Int | Arrow of ty * ty
```

dargestellt werden, und die erforderlichen Wörter durch Werte des Typs `token` aus Abschnitt 13.1.

(a) Deklarieren Sie einen Prüfer `test : token list → token list`.

(b) Deklarieren Sie einen Parser `parse : token list → ty * token list`.

(c) Deklarieren Sie eine Prozedur `rep : ty → token list`, die Typen als Wortfolgen darstellt.

(d) Deklarieren Sie eine Prozedur `str : ty → string`, die Typen als Zeichenfolgen darstellt.

Aufgabe K.6
Wir betrachten in dieser Aufgabe nun boolesche Ausdrücke, welche aus zwei Konstanten *true*, *false*, sowie zwei booleschen Operationen $\mathit{and} : \mathit{bool} \times \mathit{bool} \rightarrow \mathit{bool}$ und $\mathit{or} : \mathit{bool} \times \mathit{bool} \rightarrow \mathit{bool}$ bestehen.

Ziel dieser Aufgabe ist es, einen Parser und Lexer für die konkrete Syntax von booleschen Ausdrücken anzugeben. Wir lassen Ihnen hierbei jegliche Freiheit bezüglich der Wahl der Zeichen, die Sie verwenden. Wir stellen lediglich 2 Anforderungen:

1. *and* klammert stärker als *or*:

$$\mathit{true} \wedge \mathit{false} \vee \mathit{false} \rightsquigarrow (\mathit{true} \wedge \mathit{false}) \vee \mathit{false}$$

2. *and* und *or* klammern jeweils links:

$$\mathit{true} \wedge \mathit{true} \wedge \mathit{false} \rightsquigarrow (\mathit{true} \wedge \mathit{true}) \wedge \mathit{false}$$
$$\mathit{true} \vee \mathit{true} \vee \mathit{false} \rightsquigarrow (\mathit{true} \vee \mathit{true}) \vee \mathit{false}$$

Bearbeiten Sie nun folgende Aufgaben:

(a) Geben Sie ihre abstrakte und phrasale Syntax für boolesche Ausdrücke an.

(b) Schreiben Sie einen Lexer für booleschen Ausdrücke.

(c) Schreiben Sie einen Parser für booleschen Ausdrücke.

Hinweis: Beachten Sie, dass sowohl *true* als auch *false* alleine gültige boolesche Ausdrücke sind.

Aufgabe K.7
Wir betrachten Ausdrücke, die mit Bezeichnern und den Operatoren :: und @ gebildet werden. Die phrasale Syntax sei durch die Grammatik

$$\begin{aligned} \mathit{exp} &::= \mathit{pexp}\ [(:: \mid @)\ \mathit{exp}] \\ \mathit{pexp} &::= \mathit{id} \mid (\mathit{exp}) \end{aligned}$$

gegeben. Die Operatoren :: und @ klammern also so wie in SML gleichberechtig rechts:

$$x :: y\ @\ z :: u\ @\ v \rightsquigarrow x :: (y\ @\ (z :: (u\ @\ v)))$$

Wörter und Baumdarstellungen seien wie folgt dargestellt:

```
datatype token = ID of string | CONS | APPEND | LPAR | RPAR
datatype exp = Id of string | Cons of exp * exp | Append of exp * exp
```

(a) Schreiben Sie einen Lexer `lex : char list → token list`.

(b) Schreiben Sie einen Parser für `exp`.

(c) Schreiben Sie eine Prozedur `exp : exp → string`, die Ausdrücke gemäß der obigen Grammatik mit minimaler Klammerung darstellt.

Aufgabe K.8
Betrachten Sie applikative Ausdrücke mit der folgenden abstrakten Syntax:

```
datatype exp = Id  of string          (* Identifier  *)
| App of exp * exp       (* Application *)
```

(a) Geben Sie eine eindeutige Grammatik an, die eine phrasale Syntax für diese Ausdrücke beschreibt. Wie in SML soll redundante Klammerung erlaubt sein, und Applikationen sollen bei fehlender Klammerung links geklammert werden. Verwenden Sie die Kategorien *exp*, *pexp* und *id*.

(b) Schreiben Sie eine Prozedur `exp : exp → string`, die applikative Ausdrücke gemäß Ihrer Grammatik mit minimaler Klammerung darstellt.

(c) Geben Sie eine rechtsrekursive Grammatik mit einer Hilfskategorie exp' an, aus der sich die Parsingprozeduren ableiten lassen.

(d) Schreiben Sie eine Prozedur `test : token list → bool`, die testet, ob eine Liste von Wörtern einen applikativen Ausdruck gemäß Ihrer Grammatik darstellt. Wörter sollen wie folgt dargestellt werden:

```
datatype token = ID of string | LPAR | RPAR
```

Aufgabe K.9
Schreiben Sie einen Lexer für F.

Aufgabe K.10
Schreiben Sie einen Parser für die syntaktischen Kategorien von F. Betrachten Sie dazu die phrasale Syntax für F auf Seite 275.

Datenstrukturen

Aufgabe K.11
Sei die folgende Struktur gegeben:

```
signature ISET = sig
type set
val set : int list → set
val union : set → set → set
val subset : set → set → bool
end
```

```
structure ISet :> ISET = struct
type set = int list
fun set xs = xs
fun union xs ys = xs @ ys
fun elem ys x = List.exists (fn y ⇒ y = x) ys
fun subset xs ys = List.all (elem ys) xs
end
```

Bearbeiten Sie nun die folgenden Aufgaben:

(a) Schreiben Sie eine Prozedur `member : int → ISet.set → bool`, die testet, ob ein Element in einer Menge enthalten ist.

(b) Schreiben Sie eine Prozedur `empty : ISet.set → bool`, die testet, ob eine Menge leer ist.

(c) Schreiben Sie eine Prozedur `equal : ISet.set → ISet.set → bool`, die testet, ob zwei Mengen identisch sind.

Aufgabe K.12
Unter einer Umgebung wollen wir eine Funktion verstehen, deren Definitionsbereich eine Menge von Strings ist. Sie sollen eine Struktur `Env` deklarieren, die Umgebungen wie folgt implementiert:

```
type α env
exception Unbound
val env : (string * α) list → α env
val lookup : α env → string → α
val update : α env → string → α → α env
```

- `env` liefert zu einer Liste von Paaren die entsprechende Umgebung. Wenn die Liste zu einem String mehr als ein Paar enthält, soll nur eines der Paare verwendet werden.
- `lookup` liefert zu einer Umgebung f und einem String s den Wert $f\ s$. Falls f auf s nicht definiert ist, wirft `lookup` die Ausnahme `Unbound`.
- `update` liefert zu einer Umgebung f, einem String s und einem Wert x die Umgebung $f[s := x]$.

Deklarieren Sie die Struktur `Env` mit einem Signaturconstraint, der die Signatur direkt angibt (also ohne Rückgriff auf eine Signaturdeklaration).

kNobelpreis

Sie sind an den Lösungen für kNobelaufgabe I interessiert? Der/die Aufgabensteller:in bietet ein kNobeltutorium für alle Interessierte an:

Donnerstag, 28.1. um 16:15
Zoom-Link:
https://cs-uni-saarland-de.zoom.us/j/98947936655?pwd=TXdBOXUxSmI2SzRDeWNsellnUEhGQT09

Hinweis: Dies ist die letzte kNobelaufgabe. Das ganze Programmierung 1 Team und alle kNobeltutoren bedanken sich herzlich für all die tollen Abgaben und interessanten Tutorien. Es hat sehr viel Spaß gemacht!

Um gut auf die Klausur vorbereitet zu sein, hat Dieter Schlau sich diese Woche vorgenommen, den alten Vorlesungsstoff zu wiederholen. Gerade schauen Sie sich mit ihm zusammen Listenprozeduren an. „Hey Dieter, so geht das doch nicht. Du hast schon wieder den Basisfall vergessen“, stellen Sie fest. Aber Dieter will seinen Fehler nicht eingestehen. Da kommt ihm die rettende Idee: „Die brauche ich gar nicht. Ich rufe meine Funktionen einfach mit Listen auf, die unendlich groß sind. Problem gelöst!“ „Unendliche Listen gibt's doch gar nicht“, erwidern Sie. Aber wenn Dieter nach einem Semester Programmierung 1 eines gelernt hat, dann, dass das Wunder der Rekursion alles möglich macht! Also macht er sich an die Arbeit:

```
fun nats n = n :: nats (n+1)
```

„Damit kann ich eine unendliche Liste mit allen natürlichen Zahlen erzeugen“, verkündet Dieter stolz. Natürlich möchte er Ihnen seine Entdeckung sofort demonstrieren. Doch als er den Code in den Interpreter eingibt, muss er feststellen, dass sein Programm leider nicht terminieren möchte. Dieter ist am Boden zerstört.

Sie müssen Ihren Freund trösten. Vielleicht gibt es ja doch einen Weg, unendliche Datenstrukturen in SML darzustellen?

Aufgabe K.13 *(Aufwärmen)*
Betrachten Sie folgenden Datentyp

```
datatype α ilist = inil | icons of α * (unit → α ilist)
```

(a) Wie können Sie damit das Problem umgehen, das Dieter mit regulären Listen hatte?

(b) Implementieren Sie mit `ilist` die unendliche Liste $[1, 1, 1, ...]$, Dieters Liste $[0, 1, 2, ...]$ und die Liste $[42]$.

(c) Schreiben Sie zum Testen eine Prozedur `take`, die die ersten n Zahlen einer `ilist` als normale Liste zurückgibt. Entwickeln Sie außerdem Versionen der bekannten Listenprozeduren `nth`, `map`, `append` und `filter` für `ilist`.

Aufgabe K.14 *(To infinity...)*
Mittlerweile sind Sei beide ganz vertieft in das Implementieren von unendlichen Listen. Nach einiger Zeit schaut Dieter auf und prahlt: „Ich habe gerade einen Weg gefunden, die Liste der natürlichen Zahlen aufzustellen, ohne in dem Funktionsargument hochzählen zu müssen."

```
fun nats () = [redacted]
```
[1]

Können Sie da mithalten? Seien Sie kreativ! Deklarieren Sie folgende Listen auf möglichst viele verschiedene Arten als `ilist`. Versuchen Sie z.B. wie Dieter nicht mit den Funktionsargumenten zu arbeiten.

(a) Die Liste der natürlichen Zahlen $[0, 1, 2, ...]$

(b) Die alternierende Liste $[\mathit{true}, \mathit{false}, \mathit{true}, \mathit{false}, ...]$

(c) Die Liste der Fakultäten $[1, 1, 2, 6, 24, 120, ...]$

(d) Die Liste der Fibonacci-Zahlen $[0, 1, 1, 2, 3, 5, ...]$

(e) Die Liste der Primzahlen $[2, 3, 5, 7, 11, 13, ...]$

(f) Die Liste aller binären Strings, die keine zwei aufeinander folgenden Einsen enthalten (z.B. `"100010"` ist enthalten, `"011"` aber nicht)

(g) Für wahre Knobler: Die Liste aller endlichen Teilmengen von $\mathbb{N}$, wobei wir Mengen als Listen darstellen (vergessen Sie nicht zu begründen, warum Ihr Ansatz korrekt ist). Wie sähe es mit einer Liste für ganz $\mathcal{P}(\mathbb{N})$ aus?

Aufgabe K.15 *(...and beyond)*
Da Sie nun Experten auf dem Gebiet der unendlichen Listen sind, wollen wir nun etwas hinter die Kulissen schauen. Dieter zeigt Ihnen folgendes (recht ineffizientes) Berechnungsschema für die Fibonacci-Liste aus der vorherigen Aufgabe:

$$\begin{aligned} \mathit{fibs}'\, n &= f\, n :: \mathit{fibs}'\, (n+1) \\ \mathit{fibs} &= \mathit{fibs}'\ 0 \end{aligned} \qquad\qquad \begin{aligned} f\, 0 &= 1 \\ f\, 1 &= 1 \\ f\, n &= f\, (n-1) + f\, (n-2) \end{aligned}$$

Nachdem Dieter das Schema als `ilist` implementiert hat, führt er zum Testen `take 20 fibs` aus. Wie erwartet dauert der Aufruf einige Sekunden. Als er anschließend `nth 19 fibs` aufruft, stellt er enttäuscht fest: „Das dauert ja nochmal genau so lange! Dabei wurden durch `take` die ersten 20 Elemente doch schon ausgerechnet". Dieter hat Recht: Bei jedem Zugriff auf die Liste muss alles neu berechnet werden. Können wir das vermeiden und unsere Listen effizienter machen?

Um zu erreichen, dass Listen die Werte nur einmal berechnen, müssen wir leider die wunderbare Welt der Funktionalität verlassen. Lesen Sie dazu Kapitel 15.1 – 15.3, um sich mit Speicher in SML vetraut zu machen. Realisieren Sie damit unendliche Listen mit dem gewünschten Verhalten.

- Die Aufgabe ist lösbar, ohne den Datentyp `ilist` zu verändern. Wenn Sie möchten, können Sie aber auch Änderungen vornehmen. Behalten Sie aber auf jeden Fall die Struktur mit einem nil und einem cons Konstruktor bei.
- Testen Sie Ihren Ansatz, indem Sie die Listen aus Aufgabe KK.2 erneut deklarieren. Ihre Ideen sollten weiterhin funktionieren. Werden sie vielleicht sogar effizienter?
- Schaffen Sie es, dass die Liste $[1, 1, 1, ...]$ nur konstant viel Speicherplatz verbraucht?
- Mit der Sequenzialisierung `(print "x"; e)` können Sie in SOSML zum Testen ausgeben lassen, wann ein Ausdruck `e` ausgewertet wird. Das ist etwas präziser als die Laufzeit zu messen.

[1] Dieter hat die schlechte Angewohnheit, immer fürchterlich zu nuscheln, wenn er aufgeregt ist. Sie haben leider kein Wort verstanden. Aber Kopf hoch, Sie kommen bestimmt selbst drauf.