Datenabstraktion

Wird ein Typ in einer Schnittstelle (Signatur) nur deklariert, aber nicht definiert, so kann er von außen nur über die Operationen der Schnittstelle verarbeitet werden. Folgerungen:

- Die Implementierung des Typs kann später durch eine andere ersetzt werden. Beispiel: `Stapel0` ersetzt durch `Stapel1`.
- *Invarianten* des Typs können garantiert werden: `Stapel1.t` enthält nur solche Paare `(s,l)`, bei denen `List.length s = l`.

Ein Typ, dessen Definition von außen nicht sichtbar ist, heißt *abstrakter Datentyp*. Die Verwendung solcher heißt *Datenabstraktion*.

Separate Compilation

Sei `datei.ml` eine Datei mit ML-Definitionen.

Mit dem Befehl

```
ocamlc -c datei.ml
```

wird eine Datei `datei.cmo` erzeugt, die mit `#load "datei.ml";;` geladen werden kann.

Der Effekt ist derselbe wie wenn

```
module Datei = struct
```

Inhalt von `datei.ml`

```
end
```

eingegeben worden wäre. Es wird also ein Modul des Namens `Datei` erzeugt, dessen Komponenten die von `datei.ml` sind.

Beispiel

Datei `abc.ml` enthalte

```
let a = 0
let b = 1
let c = 2
```

Wir kompilieren mit `ocamlc -c abc.ml` und erhalten `abc.cmo`.
Folgende Sitzung ist möglich:

```
# #load "abc.cmo"
# Abc.a;;
- : int = 0
# Abc.b;;
- : int = 1
# Abc.c;;
- : int = 2
#
```

Schnittstellen

Sei `datei.mli` eine Datei, die ML Schnittstellenkomponenten, wie `val x : int` enthält.

Mit dem Befehl

```
ocamlc -c datei.mli
```

wird eine Datei `datei.cmi` erzeugt.

Wird nunmehr `ocamlc -c datei.ml` ausgeführt, so hat `#load "datei.cmo"`denselben Effekt wie

```
module type DATEI = sig
```

Inhalt von `datei.mli`

```
end
module Datei : DATEI = struct
```

Inhalt von `datei.ml`

```
end
```

Beispiel

Datei `abc.ml` wie zuvor.

Datei `abc.mli` enthalte nur `val a:int`.

Wir kompilieren mit `ocamlc -c abc.mli` gefolgt von `ocamlc -c abc.ml`. Folgende Sitzung ist möglich:

```
# #load "abc.cmo"
# Abc.a;;
- : int = 0
# Abc.b;;
  ^^^^^

Unbound value Abc.b
```

Ausführbare ML-Programme

Mit `ocamlc -o main datei.ml` wird eine ausführbare Datei `main` ("EXE-Datei") erzeugt. Der Effekt ist der des Auswertens aller Definitionen in `datei.ml`.

Natürlich macht das nur Sinn, wenn `datei.ml` Ausdrücke mit Seiteneffekten, wie die Grafikfunktionen oder `print_string` enthält:

Enthalte Datei `datei.ml` den folgenden Code

```
let _ = print_string "Hello, world!\n"
```

Compilieren mit

```
ocamlc -o main datei.ml
```

gefolgt von

```
./main
```

gibt `Hello, world!` aus.

Binden

Verwendet `datei.ml andere Dateien, z.B.: \verb`datei1.ml+ und `datei2.ml`, die man interaktiv mit `#load` hinzuladen würde, so muss man die entsprechenden `.cmo` Dateien bei der Kompilierung aufführen:

```
ocamlc -o main  datei1.cmo datei2.cmo datei.ml
```

oder auch

```
ocamlc -o main  datei1.cmo datei2.cmo datei.cmo
```

Die `.cmo` Dateien werden *gebunden* (*linked*).

Größeres Beispiel

Das Projekt der heutigen Übung besteht aus den Dateien

```
syntax.mli lex.mli norm.ml parse.mli graph.mli visualise.mli
lex.ml provided.cma visualise.ml main.ml
```

Dabei ist `provided.cma` ein Archiv, welches die Funktionen in `graph.mli` und `parse.mli` implementiert.

Sie müssen `norm.ml lex.ml visualise.ml` schreiben und mit `provided.cma` und `main.ml` zum ausführbaren Hauptprogramm zusammenbinden.

Aufgaben der Module

- `syntax.mli` Datentypen für arithmetische Ausdrücke und “Tokens”
- `parse.mli` Eine Funktion, die eine Tokenliste auf Ausrdücke abbildet. Eine Ausnahme.
- `lex.mli` Eine Funktion, die Strings auf Tokenlisten abbildet. Eine Ausnahme.
- `graph.mli` Eine Funktion, die Gegenstände zeichnet (Gegenstand = Strecke, Textbaustein). Eine Funktion, die die Größe eines Strings in Pixeln bestimmt.
- `visualise.mli` Eine Funktion, die einen Ausdruck auf eine Liste von Gegenständen abbildet (Baumdarstellung).
- `norm.mli` Eine Funktion, die Ausdrücke normalisiert.
- `main.ml` Ein Hauptprogramm, das einen String einliest, ggf. normalisiert und dann als Baum ausgibt.

Kompilieren

Soll alles kompiliert werden, so sind folgende Befehle auszuführen:

```
ocamlc -c syntax.mli
ocamlc -c lex.mli
ocamlc -c lex.ml
ocamlc -c visualise.mli
ocamlc -c visualise.ml
ocamlc -c norm.mli
ocamlc -c norm.ml
ocamlc -o main.ml
ocamlc -o main provided.cma visualise.cmo norm.cmo lex.cmo m
```

Ändert man eine Datei, so muss die entsprechende Datei neu kompiliert werden, ändert man eine Schnittstelle (`.mli`), so müssen alle Schnittstellen und Programme, die davon abhängen, neu kompiliert werden.

Makefiles

Das Unix-Werkzeug `make` erleichtert solche Kompilierungsaufgaben:

Man schreibt ein für alle Mal alle Aufgaben und deren Abhängigkeiten in eine Datei `Makefile`.

Danach genügt der Befehl `make` um nach einer Änderung die entsprechenden Neukompilierungen vorzunehmen.

Form des Makefiles

Ein Makefile enthält (neben anderen Komponenten) *Regeln* der Form.

```
datei1: datei2 datei3
        befehl1
        befehl2
```

Die Regel besagt folgendes:

- `datei1` hängt von `datei2`, `datei3` ab: haben `datei2`, `datei3` späteres Änderungsdatum als `datei1`, so ist `datei1` nicht mehr gültig und muss neu erstellt werden.
- Die Befehle[a] `befehl1` und `befehl2` erstellen `datei1` neu.

Wird `make datei1` aufgerufen, so wird geprüft, ob `datei2`, `datei3` aktuell sind (mithilfe entsprechender Regeln im Makefile). Wenn nein, dann werden zunächst diese "gemacht". Anschließend wird geprüft, ob `datei1` aktuell ist und ggf. mithilfe von `befehl1`, `befehl2` neu "gemacht".

[a] Achtung, vor jedem Befehl, wie `befehl1` und `befehl2` muss ein Tabulator stehen.

Ein Makefile für unser Projekt

```
main: visualise.cmo norm.cmo lex.cmo main.cmo
    ocamlc -o main provided.cma visualise.cmo norm.cmo lex.cmo main.cmo

visualise.cmi: visualise.mli
    ocamlc -c visualise.mli

visualise.cmo: syntax.cmi visualise.cmi
    ocamlc -c visualise.ml

lex.cmi: lex.mli
    ocamlc -c lex.mli

norm.cmi: norm.mli
    ocamlc -c norm.mli

lex.cmo: lex.cmi syntax.cmi
    ocamlc -c lex.ml

norm.cmo: norm.cmi syntax.cmi
    ocamlc -c norm.ml
```

Nachteil dieser Form

- Makefile enthält viele Redundanzen
- Wird die Modulstruktur geändert, so ändern sich ggf. die Abhängigkeiten und das Makefile muss geändert werden

Makefile mit impliziten Regeln und Abkürzungen

```
MAIN_OBJS = visualise.cmo norm.cmo lex.cmo main.cmo
.SUFFIXES: .ml .mli .cmo .cmi
.ml.cmo:
        ocamlc -c $<
.mli.cmi:
        ocamlc -c $<
main: $(MAIN_OBJS)
    ocamlc -o main $(MAIN_OBJS)
lex.cmi: syntax.cmi
norm.cmi: parse.cmi
norm.cmo: norm.cmi
visualise.cmi: graph.cmi syntax.cmi
lex.cmo: syntax.cmi lex.cmi
main.cmo: graph.cmi lex.cmi parse.cmi visualise.cmi
visualise.cmo: graph.cmi visualise.cmi
visualise.cmx: graph.cmx visualise.cmi
```

Erklärung

- Die implizite Regel

```
.ml.cmo:
        ocamlc -c $<
```

  besagt, wie eine `.cmo` Datei aus einer `.ml` Datei gemacht wird. `$<` steht für die `.ml` Datei, `$@` steht für die `.cmo` Datei (brauchen wir hier nicht).

- Die Abhängigkeiten sind weiter unten ohne Befehle angegeben.
- `$(MAIN_OBJS)` ist ein weiter oben definierte Abkürzung.
- Der Block mit den Abhängigkeiten kann mit `ocamldep *.mli *.ml` automatisch erzeugt werden.

“Professionelles” Makefile für das gesamte Projekt

```
OCAMLC=ocamlc
OCAMLOPT=ocamlopt
OCAMLDEP=ocamldep
OCAMLYACC=ocamlyacc
INCLUDES=
OCAMLFLAGS=$(INCLUDES)
PROVIDED_OBJS=parsen.cmo parse.cmo graph.cmo
MAIN_OBJS=provided.cma visualise.cmo lex.cmo main.cmo
DEPEND += parsen.ml

main: $(MAIN_OBJS)
        $(OCAMLC) -o main $(OCAMLFLAGS) $(MAIN_OBJS)

provided.cma: $(PROVIDED_OBJS)
        $(OCAMLC) -a -o provided.cma $(OCAMLFLAGS) graphics.cma $(PROVIDED_OBJS)

.SUFFIXES: .ml .mli .cmo .cmi .cmx .mly

.ml.cmo:
        $(OCAMLC) $(OCAMLFLAGS) -c $<
.mli.cmi:
        $(OCAMLC) $(OCAMLFLAGS) -c $<
.ml.cmx:
        $(OCAMLOPT) $(OCAMLOPTFLAGS) -c $<
.mly.ml:
        $(OCAMLYACC) $<
clean:
        rm -f main;rm -f *.cm[iox];rm parsen.ml;rm parsen.mli
depend: $(DEPEND)
        $(OCAMLDEP) $(INCLUDES) *.mli *.ml > .depend
include .depend
```

Was man über `make` wissen muss

- `make` ist eine Möglichkeit, größere Projekte zeitsparend und fehlerfrei zu kompilieren.
- Kooperiert mit Abhängigkeitsgenerator (`ocamldep`)
- Abhängigkeitsgeneratoren gibt es genauso für C, C++.
- `make` funktioniert unabhängig von der Programmiersprache und ist nicht auf Kompilierbefehle beschränkt.
- Für manche Programmiersprachen gibt es spezielle Kompilationsmanager (NJ-SML, Visual C++,etc), die `make` ersetzen, wenn nur in dieser Sprache gearbeitet wird. Für Projekte mit mehreren Sprachen braucht man dann doch wieder `make`.
- Man sollte wissen, dass `make` existiert; ggf. Dokumentation studieren.

Kapitel 6: Effiziente Algorithmen

Begriffsklärungen

Algorithmus ist `effizient`, wenn seine Ausführung möglichst wenig Aufwand verursacht.

Aufwand: Rechenzeit, Speicherplatz, Anzahl bestimmter Elementaroperationen.

Die *Komplexität* eines Algorithmus gibt den Aufwand als Funktion der Eingabe oder deren *Größe* an.

Bei Angabe der Komplexität als Funktion der Eingabegröße unterscheidet man

- *Best case:* Aufwand bei am günstigsten gewählter Eingabe zu fester Größe
- *Worst case:* Aufwand bei am ungünstigsten gewählter Eingabe zu fester Größe
- *Average case:* Erwartungswert des Aufwands bei Eingaben einer festen Größe bzgl. einer bestimmten Verteilung.

Beispiel: `insertel`

```
let rec insertel = function
   (a, []) -> [a]
 | (a, h::t) -> if a <= h then a::h::t else h :: insertel(a,t)
```

Anzahl der Auswertungsschritte von `insertel(a,l)` als Funktion von $n =$ `length(l)`.

- Best case: 4 (Element `a` wird am Anfang eingefügt)
- Worst case: $3n$ (Element `a` wird am Ende eingefügt)
- Average case: $1.5 \cdot n$ (Element `a` wird "in der Mitte" eingefügt)

Größenordnungen

Oft interessiert man sich nur für die *Größenordnung* der Komplexität: konstant, linear, quadratisch, exponentiell, ...

Man gibt diese mit der O-Notation an:

Sei $f : \mathbb{N} \to \mathbb{N}$:

$$O(f) = \{g \mid \exists N.\exists c > 0.\forall N \geq n.g(n) \leq c \cdot f(n)\}$$

Insbesondere: $\lim_{n\to\infty} \frac{g(n)}{f(n)}$ existiert:$g \in O(f)$.

Notationen

Man verwendet oft folgende Abkürzungen:

- $f(n)$ statt f, z.B.: $2n^2 \in O(n^2)$ statt $\mathbf{function}(n)2n^2 \in O(\mathbf{function}(n)n^2)$,
- $g = O(f)$ statt $g \in O(f)$, z.B.: $n^2 = O(2^n)$,
- $f + O(g)$ statt $\{h \mid \exists u \in O(g).h = f + u\}$, z.B.: $\sum_{i=1}^{n} i^k = \frac{1}{k+1} n^{k+1} + O(n^k)$.

Beispiele

- $2n^2 = O(n^2)$
- $n^{1000} = O(2^n)$
- $\log(n) = O(n)$
- $1000/n = O(1)$
- Best case Laufzeit von `insertel` $= O(1)$
- Average case Laufzeit von `insertel` $= O(n)$
- Worst case Laufzeit von `insertel` $= O(n)$
- Worst case Laufzeit von `inssort` $= O(n^2)$