Fakultät IV – Elektrotechnik und Informatik
MPGI 1 WS 2013/14
Pepper / Höger / Reicherdt / Zuber / Tutoren

Commands und OPAL WINDOWS

1 Der Opal-Compiler `ocs`

Bisher haben wir nur den OPAL-Interpreter `oasys` benutzt. Dabei haben wir die Eingabe für unsere Programme (d. h. Funktionen) in den Interpreter eingegeben und das Ergebnis vom Interpreter anzeigen lassen. Das bedeutet, der Interpreter ist für die Ein-/Ausgabe zuständig, nicht unser funktionales Programm.

Funktionale Programmierung wäre aber für reale Probleme völlig ungeeignet, wenn wir immer einen Interpreter zur Kommunikation mit der Außenwelt zu Hilfe ziehen müssten. Glücklicherweise können wir aus einem funktionalen Programm auch eine eigenständig ausführbare Datei erzeugen. Im Falle von OPAL dient dazu der Compiler `ocs`.

Wir zeigen an dem berühmten „Hello, world!“-Programm (siehe Listing 1), wie der OPAL-Compiler benutzt wird. Die programmiersprachlichen Mittel, die wir in „Hello, world!“ benutzen, werden wir im Folgenden genau unter die Lupe nehmen.

Wir müssen dem Compiler `ocs` in einer Datei `SysDefs` mitteilen, welche Funktion den Einstiegspunkt in das Programm bildet und in welcher Struktur sich diese Einstiegsfunktion befindet.

Im Fall des „Hello, world!“-Programms soll die Funktion `helloWorld`, die sich in der Struktur `HelloWorld` befindet, als erstes ausgeführt werden. Dies halten wir in der Datei `SysDefs` folgendermaßen fest:

```
TOPSTRUCT=HelloWorld
TOPCOM=helloWorld
```

`TOPCOM` bezeichnet die Einstiegsfunktion, `TOPSTRUCT` die Struktur, in der die durch `TOPCOM` bezeichnete Funktion steht.

Die `TOPCOM`-Funktion muss den Typ `com[void]` haben (deswegen der Name `TOPCOM`).

Wir können nun das „Hello, world!“-Programm in eine ausführbare Datei übersetzen, indem wir den Compiler in dem Verzeichnis, das die `SysDefs` enthält, aufrufen:

```
$ ocs
Checking Signature of HelloWorld ...
Compiling Implementation of HelloWorld ...
Generating i686 object code for HelloWorld ...
Generating startup code for helloWorld ...
Linking helloWorld ...
```

In der letzten Zeile der Compiler-Ausgabe können wir erkennen, dass die ausführbare Datei `helloWorld` heißt — also genauso, wie die durch `TOPCOM` bezeichnete Funktion.

Wir können `helloWorld` nun direkt auf der Kommandozeile starten und erhalten die erwartete Begrüßung:

```
$ ./helloWorld
Hello, world!
```

Listing 1: *HelloWorld*

```
SIGNATURE HelloWorld

IMPORT Com  ONLY com
       Void ONLY void

FUN helloWorld : com[void]
```

```
IMPLEMENTATION HelloWorld

IMPORT BasicIO COMPLETELY

DEF helloWorld == writeLine("Hello, world!")
```

Weitere Hinweise Wenn man während der Programmentwicklung Änderungen an den Quelldateien vornimmt, übersetzt der Compiler um Zeit zu sparen nur die geänderten Dateien. Hin und wieder kann es vorkommen, dass dabei Fehler auftreten, obwohl das Programm fehlerfrei ist. Der Aufruf

```
$ ocs clean
```

löscht alle Zwischenergebnisse, so dass beim nächsten Compiler-Aufruf alle Dateien übersetzt werden. Dies hilft manchmal in der o. g. Situation.

Wenn es ganz schlimm kommt und gar nichts mehr geht, hilft es gelegentlich, das `OCS`-Verzeichnis, in dem der Compiler Zwischencompilate ablegt, zu löschen. Insbesondere nach Modifikation der `SysDefs`-Datei kann diese Massnahme helfen.

2 Einfache Ein-/Ausgabe

Wir schauen uns jetzt den Quelltext des „Hello, world!“-Programms an.

Funktionen, die Ein- oder Ausgabe betreiben, haben alle den (polymorphen) Typ `com` (für *command*). Die Funktion — im Zusammenhang mit Ein-/Ausgabe sprechen wir auch von Kommandos — `writeLine`, die eine Textzeile auf dem Terminal ausgibt, hat bspw. den Typ

```
FUN writeLine : denotation -> com[void]
```

Der Typ `void` hat genau ein Element, nämlich `nil`, und gibt an, dass `writeLine` kein Ergebnis liefert.

Mit der Funktion `ask` können wir eine Eingabe von der Tastatur einlesen.

```
FUN ask : denotation -> com[denotation]
```

Der Typ `com[denotation]` gibt an, dass `ask` ein Kommando ist, das einen Wert vom Typ `denotation` zurückgibt.

Um mehrere Kommandos nacheinander auszuführen, gibt es den Kombinator `&`:

```
FUN & : com[α] ** (α -> com[β]) -> com[β]
```

Das zweite Argument von `&` ist eine Funktion, die das Resultat des ersten Arguments (d. h. Kommandos) als Eingabe bekommt.

Wir illustrieren die Funktionen `&`, `ask` sowie das bereits bekannte `writeLine` am Beispiel `Echo` (siehe Listing 2).

Das Kommando `echo` ruft zunächst das Kommando `ask` auf, um einen Text vom Terminal zu lesen. Dieses Lese-Kommando wird durch `&` mit der Funktion

```
\\input. writeLine("Output: " ++ input)
```

komponiert. Die o. g. Funktion hat den Typ `denotation -> com[void]`, d. h. wir instanziieren die Typvariablen im Typ von `&` wie folgt:

$$\alpha = \texttt{denotation}$$
$$\beta = \texttt{void}$$

Die Funktion `\\input. writeLine("Output: " ++ input)` gibt nun die eingegebene Zeile wieder auf dem Terminal aus:

```
$ ./echo
Input: Eine Zeile Text
Output: Eine Zeile Text
```

Listing 2: *Echo*

```
SIGNATURE Echo

IMPORT Com  ONLY com
       Void ONLY void

FUN echo echo2 : com[void]
```

```
IMPLEMENTATION Echo

IMPORT BasicIO    COMPLETELY
       ComCompose COMPLETELY
       Denotation COMPLETELY

DEF echo == ask("Input: ") & (\\input.
            writeLine("Output: " ++ input))

DEF echo2 == ask("First line: ")             & (\\line1.
             ask("Second line: ")            & (\\line2.
             writeLine("Line 1: " ++ line1)  &
             writeLine("Line 2: " ++ line2)))
```

Wir können auch mehr als zwei Kommandos hintereinander ausführen. Nicht immer ist dabei das zweite Argument für `&` eine Funktion. Aus diesem Grund gibt es die überlagerte Form

```
FUN & : com[α] -> com[β] -> com[β]
```

Wir betrachten als Beispiel die Funktion `echo2` (ebenfalls Listing 2). `echo2` liest zwei Zeilen ein und gibt sie nacheinander aus. Die beiden Aufrufe der `writeLine`-Kommandos sind durch die alternative Form des `&`-Kombinators verknüpft.

```
$ ./echo2
First line: A line of text
Second line: Another line of text
Line 1: A line of text
Line 2: Another line of text
```

Wir können uns auch eigene Kommandos schreiben, d. h. Funktionen, die den Resultattyp `com` haben. Einfache Beispiele sind die bereits gesehenen Funktionen `echo` und `echo2`, die allerdings den relativ uninteressanten Ergebnistyp `com[void]` haben. Wenn wir bspw. in der Funktion `echo2` das Einlesen vom Ausgeben trennen möchten, können wir das wie folgt tun:

```
DEF echo2 == readLines("Line: ") & writeLine

FUN readLines : denotation -> com[denotation]
DEF readLines(prompt) == ask(prompt) & (\\line1.
                         ask(prompt) & (\\line2.
                         succeed(line1 ++ ‘(newline) ++ line2)))
```

Die Funktion `succeed` hat den Typ

```
FUN succeed : α -> com[α]
```

und erzeugt ein Kommando, das den übergebenen Wert abliefert. Im gegebenen Beispiel lesen wir nacheinander zwei Zeilen ein und fügen sie zu einer einzelnen *Denotation* zusammen, bevor wir sie zurückgeben.

Bibliotheca Opalica

Struktur	*Funktion/Typ*
`Com`	`SORT com` Der Typ „Kommando“
	`FUN succeed: α -> com[α]` (Erfolgreiches) Zurückgebeben eines Wertes vom Typ α
`Void`	`SORT void` Der einelementige Typ für Kommandos ohne Resultat

`ComCompose`	`FUN & : com[α] -> (α -> com[β]) -> com[β]`
	`FUN & : com[α] -> com[β] -> com[β]`
	Komposition von Kommandos

`BasicIO`	`FUN writeLine : denotation -> com[void]`
	`FUN write : denotation -> com[void]`
	Ausgabe von Text mit und ohne Zeilenwechsel

	`FUN ask : denotation -> com[denotation]` Einlesen von Text (mit Eingabeaufforderung)

3 Kommandozeilenargumente

Oft möchte man einem Programm beim Aufrufen Argumente wie etwa zu bearbeitende Dateien o. ä. übergeben. Beispielsweise rufen wir einen Editor (i. d. F. Emacs) oft folgendermassen auf:

```
$ emacs -nw FooBar.sign
```

Dabei sind `-nw` und `FooBar.sign` Argumente auf der Kommandozeile.

3.1 Auslesen der Kommandozeilenargumente

Die Struktur `ProcessArgs` erlaubt es, die übergebenen Argumente auszulesen.

Listing 3 zeigt ein Programm, das die Anzahl der übergebenen Argumente sowie die Argumente selbst auf dem Terminal ausgibt.

Die Funktion `args` aus `ProcessArgs` liefert uns dabei eine Liste aller Argumente. Die beiden Kommandos `writeArgCount` sowie `writeArgs` geben die Länge der Kommandozeile und die einzelnen Argumente aus.

`writeArgs` ist dabei ein Beispiel für ein rekursiv geschriebenes Kommando. Interessant ist dabei der Terminierungsfall, der einfach nur den (uninteressanten) Wert `nil` liefert

Hier ist ein Beispielaufruf von `cmdLineArgs`:

```
$ ./cmdLineArgs 1stArg 2ndArg
Number of arguments: 3
Argument #0: ./cmdLineArgs
Argument #1: 1stArg
Argument #2: 2ndArg
```

(Wir bemerken, dass der Programmname das nullte Argument ist.)

3.2 Prüfen der Kommandozeilenargumente

Oft kann ein Programm ohne geeignete Argumente auf der Kommandozeile gar nicht arbeiten. In solchen Fällen müssen wir prüfen, ob die Argumente den Erfordernissen entsprechen. Falls dies nicht der Fall ist, bleibt oft nichts anderes übrig, als das Programm mit einer Fehlermeldung zu beenden.

Das folgende Beispiel illustriert diesen Fall. Das Programm `twice` erwartet eine Zahl auf der Kommandozeile und verdoppelt diese.

Wenn vergessen wird, das erforderliche Argument zu übergeben, wird das Programm mit einer Fehlermeldung beendet:[1]

```
$ ./double
ERROR: usage: ./double <number>
```

Ansonsten wird das Ergebnis ausgegeben:

```
$ ./double 6
12
```

Listing 4 zeigt das Programm. Das Kommando `exit: nat -> com[`α`]` aus der Struktur `Com` beendet das Programm mit dem gegebenen Exit-Code. Unter Unix bedeutet ein Exit-Code von 0 ein fehlerfreies und ein Code > 0 ein fehlerhaftes Beenden des Programms.

[1] Genaugenommen müssen wir selbstverständlich noch prüfen, dass tatsächlich eine Zahl übergeben wird und nicht ein x-beliebiger String wie z. B. `dgxg%'&u`, den wir schlecht verdoppeln können.

Listing 3: *CmdLineArgs*

```
SIGNATURE CmdLineArgs

IMPORT Com  ONLY com
       Void ONLY void

FUN cmdLineArgs : com[void]
```

```
IMPLEMENTATION CmdLineArgs

IMPORT BasicIO     COMPLETELY
       Com         COMPLETELY
       ComCompose  COMPLETELY
       Denotation  COMPLETELY
       Identity    COMPLETELY
       Nat         COMPLETELY
       NatConv     COMPLETELY
       ProcessArgs COMPLETELY
       Seq         COMPLETELY
       Void        COMPLETELY

DEF cmdLineArgs == args                    & (\\argList.
                   writeArgCount(argList)  &
                   writeArgs(argList))

FUN writeArgCount : seq[denotation] -> com[void]
DEF writeArgCount(args) == writeLine("Number of arguments: " ++ `(args#))

FUN writeArgs : seq[denotation] -> com[void]
DEF writeArgs(args) == writeArgs(argNums, args)
    WHERE
      argNums == (0 .. (#(args)-1))(id)

FUN writeArgs : seq[nat] ** seq[denotation] -> com[void]
DEF writeArgs(<>, <>) == succeed(nil)
DEF writeArgs(num::nums, arg::args) == writeLine(argText) &
                                       writeArgs(nums, args)
    WHERE
      argText == "Argument #" ++ `(num) ++ ": " ++ arg
```

Listing 4: *Double*

```
SIGNATURE Double

IMPORT Com  ONLY com
       Void ONLY void

FUN double : com[void]
```

```
IMPLEMENTATION Double

IMPORT BasicIO     COMPLETELY
       Com         COMPLETELY
       ComCompose  COMPLETELY
       Denotation  COMPLETELY
       Nat         COMPLETELY
       ProcessArgs COMPLETELY
       Seq         COMPLETELY
       Void        COMPLETELY


DEF double == getArgument & (\\n.
              writeLine(2*n))

FUN getArgument : com[nat]
DEF getArgument == argCount  & (\\count.
                   args      & (\\args.
                   IF count = 2 THEN succeed(!(ft(rt(args))))
                   ELSE writeLine("ERROR: usage: " ++ ft(args) ++ " <number>") &
                        exit(1) FI))
```

Struktur	*Funktion/Typ*
`ProcessArgs`	`FUN args : com[seq[denotation]]` Liste der Kommandozeilen-Argumente
	`FUN argCount : com[nat]` Anzahl der Kommandozeilen-Argumente
`Com`	`FUN exit : nat -> com[`α`]` Das Programm beenden

4 Lesen und Schreiben von Dateien

Neben der Ausgabe auf dem Terminal und dem Einlesen von der Tastatur ist das Lesen und Schreiben von Dateien eine weitere wichtige Interaktionsmöglichkeit mit der Umgebung.

Basis-Kommandos zum Arbeiten mit Dateien sind in der Struktur `File` zu finden. Dateien werden dabei durch sog. *Handles* des (abstrakten) Typs `file` repräsentiert.

Wir illustrieren die wichtigsten Datei-Operationen anhand eines Beispiels: Das Programm `reverse` liest alle Zeilen einer Textdatei und schreibt sie in umgekehrter Reihenfolge in eine Ausgabedatei.

Das Herz des Programms bildet das Kommando `reverse`. `reverse` erwartet die Namen der Eingabe- und Ausgabedatei als Argumente. Beide Dateien werden zunächst mittels `open` geöffnet, um ein Handle vom Typ `file` zu erhalten. `open` hat den Typ

```
FUN open : denotation ** denotation -> com[file]
```

wobei das erste Argument der Dateiname ist und das zweite angibt, ob die Datei gelesen (`"r"`) oder geschrieben (`"w"`) werden soll.[2] Ist eine Datei zum Schreiben geöffnet, wird eine evtl. bestehende Datei desselben Namens überschrieben; existiert die Datei noch nicht, wird sie angelegt.

Mit `readLines` erhalten wir alle Zeilen eines `file`s als Sequenz von `string`s. Das Gegenstück zu `readLines` ist `writeLines`, das eine Sequenz von `string`s in eine Datei schreibt.

Zum Schluß dürfen wir nicht vergessen, die Dateinhandles mittels `close` wieder freizugeben.

```
FUN reverse : denotation ** denotation -> com[void]
DEF reverse(input, output) ==
    open(input, "r")                & (\\in.
    open(output, "w")               & (\\out.
    readLines(in)                   & (\\lines.
    writeLines(out, revert(lines))  &
    close(in)                       &
    close(out))))
```

Das vollständige Programm ist in Listing 5 zu sehen.

[2] Es gibt weitere Modi wie `"b"` für Binärdateien oder `"a"` zum Anhängen weiterer Inhalte am Dateiende.

Listing 5: *Reverse*

```
SIGNATURE Reverse

IMPORT Com  ONLY com
       Void ONLY void

FUN reverse : com[void]
```

```
IMPLEMENTATION Reverse

IMPORT BasicIO     COMPLETELY
       Com         COMPLETELY
       ComCompose  COMPLETELY
       Denotation  COMPLETELY
       File        COMPLETELY
       Nat         COMPLETELY
       Pair        COMPLETELY
       ProcessArgs COMPLETELY
       Seq         COMPLETELY


DEF reverse ==
    getArguments  & (\\args.
    reverse(1st(args), 2nd(args)))


FUN reverse : denotation ** denotation -> com[void]
DEF reverse(input, output) ==
    open(input, "r")                &  (\\in.
    open(output, "w")               &  (\\out.
    readLines(in)                   &  (\\lines.
    writeLines(out, revert(lines))  &
    close(in)                       &
    close(out))))


FUN getArguments : com[pair[denotation, denotation]]
DEF getArguments ==
    argCount  & (\\count.
    args      & (\\args.
    IF count = 3 THEN LET in  == ft(rt(args))
                          out == ft(rt(rt(args)))
                      IN succeed(in&out)
    ELSE writeLine("Error: usage: " ++ ft(args) ++ " <infile> <outfile>")  &
         exit(1) FI))
```

Struktur	*Funktion/Typ*
`File`	`SORT file` Filehandles
	`FUN open : denotation ** denotation -> com[file]` Öffnen einer Datei
	`FUN close : file -> com[void]` Schliessen einer Datei
	`FUN readLines : file -> com[seq[string]]` Lesen aller Zeilen einer Datei
	`FUN writeLines : file ** seq[string] -> com[void]` Schreiben einer Liste von Zeilen in eine Datei

5 Opal Windows

OPAL verfügt über eine Bibliothek zur Programmierung graphischer Benutzerobeflächen (OPAL WINDOWS), die auf dem Tcl/Tk-System[3] basiert.

Um OPAL WINDOWS benutzen zu können, muss der `SysDefs`-Datei die Zeile

```
OPAL_LIBS += $(OPAL_WIN)
```

hinzugefügt werden, da die Bibliothek nicht standardmäßig verfügbar ist.

In diesem Abschnitt betrachten wir ein kleines Beispiel, dass die für die Projektaufgabe notwendigen Aspekte illustriert. Wir beschränken uns dabei auf Zeichenflächen sowie einfache Buttons.

Abbildung 1 zeigt das Program `textdemo`: Unterschiedlich angeordnete Texte und Linien werden über einem „Exit“-Button angezeigt.

Wir gehen die Konstruktion des Programms, das in voller Länge Listing 6 (auf S. 18) gezeigt ist, nun Schritt für Schritt durch.

[3] `www.tcl.tk`

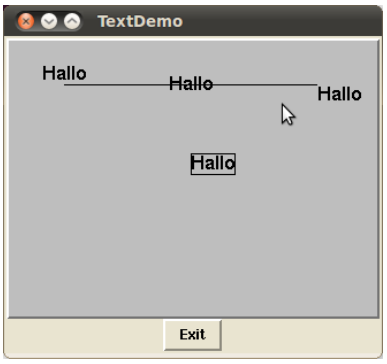


Abbildung 1: Das Program `textdemo` in Aktion.

Wir implementieren das Programm `textdemo` der Kürze halber in einer einzigen Struktur `TextDemo`. Prinzipiell wäre es besser, es auf mehrere Strukturen aufzuteilen (vgl. Vorlesung: Beispiel `MovingBall`, Modell-View-Control).

5.1 Grundlegende Einstellungen

Wir legen zunächst einige Größen und Einstellungen fest:

- Die Größe der oberen Zeichenfläche (in Millimetern)

```
FUN canvasWidth canvasHeight : real
DEF canvasWidth == "320" px
DEF canvasHeight == "240" px

FUN canvasSize : size
DEF canvasSize == canvasWidth x canvasHeight
```

- Die Breite eines umlaufenden Randes sowie die linke, mittlere und rechte x-Position.

```
DEF margin == "50" px
DEF left == margin
DEF right == canvasWidth-margin
DEF middle == canvasWidth/2
```

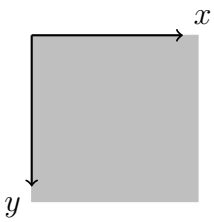
- Die Hintergrundfarbe der Zeichenfläche, den angezeigten Text „Hallo“ sowie die Schriftart (eine Standardschrift mit variabler Breite).

```
FUN canvasBackground : color
DEF canvasBackground == grey

FUN sampleText : denotation
DEF sampleText == "Hallo"

FUN stdFont : font
DEF stdFont == variable
```

5.2 Die Zeichnung

Wir wenden uns nun der aus Linien und Zeichenketten bestehenden Zeichnung zu. In OPAL WINDOWS wird eine solche Zeichnung durch Werte des Typs `drawing` aus `WinCanvasEditor` repräsentiert.

`drawing`s werden in einem Koordinatensystem gezeichnet, dessen Ursprung links oben in der Zeichenfläche ist. Die x-Achse zeigt nach rechts, die y-Achse nach unten:



Der obere Teil der Zeichnung Wir beginnen mit der waagerechten Linie. Die Funktion `line` nimmt eine Liste von Punkten und liefert ein `drawing`, das die Verbindungslinie zwischen diesen Punkte repräsentiert.

```
FUN line : drawing
DEF line == line(%(left @ ("40" px), right @ ("40" px)))
```

Punkte werden durch den Konstruktor `@` gebildet.

Nun wollen wir den Text „Hello“ an drei Punkten relativ zu der Line abbilden:

1. Am linken Ende oberhalb der Linie
2. In der Mitte auf der Linie
3. Am rechten Ende unterhalb der Line

Wir beginnen mit der linken Position:

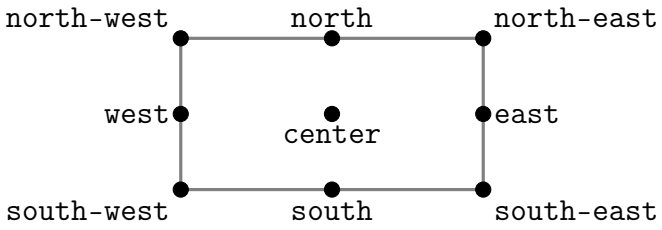
```
above == text(left @ ("40" px))
         with (text(sampleText) ++ anchor(south) ++ font(stdFont))
```

Mittels der Funktion `text` erstellen wir ein Text-Objekt an einer gegebenen Position, i. d. F. `left @ ("40" px)`.

Den eigentlichen Text fügen wir als sog. *Konfiguration* an das `drawing` an. Die Funktion `with` fügt einem `drawing` eine Konfiguration hinzu. Mehrere Konfigurationen lassen sich durch `++` kombinieren. In diesem Falle setzen wir mit `text` den anzuzeigenden Text, wählen mit `font` die gewünschte Schriftart und setzen mit `anchor` den Referenzpunkt. All diese drei Konfigurationen werden mit `++` kombiniert und per `with` dem Text-`drawing` hinzugefügt.

Der Referenzpunkt ist wichtig, da er bestimmt, wo der Text relativ zum gegebenen Punkt erscheint. Wir haben als Referenzpunkt `south` gewählt, so dass der Text über dem Punkt `left @ ("40" px)` erscheint (also *über* der Linie).

OPAL WINDOWS kennt die folgenden Referenzpunkte:



Analog zu dem linken Text erstellen wir nun den mittleren und rechten, wobei wir jeweils den Referenzpukt mittels `anchor` so ändern, dass der Text genau auf bzw. unter der Linie landet:

```
on    == text(middle @ ("40" px))
         with (text(sampleText) ++ anchor(center) ++ font(stdFont))
below == text(right @ ("40" px))
         with (text(sampleText) ++ anchor(north-west) ++ font(stdFont))
```

Einzelne `drawings` lassen sich mit der Funktion `++` zu komplexeren `drawings` kombinieren. Wir fügen nun die drei Texte zusammen:

```
FUN text : drawing
DEF text == above ++ on ++ below
```

Der untere Teil der Zeichnung Im unteren Teil der Zeichnung wollen wir den Text „Hallo“ mit einem Rahmen versehen.

Des Zeichnen des Texts erfolgt wie oben gezeigt. Wir wählen die obere linke Ecke als Referenzpunkt:

```
txt        == text(upperLeft)
              with (text(sampleText) ++ anchor(north-west) ++ font(stdFont))
upperLeft  == middle @ ("100" px)
```

Die Box können wir mit der Funktion `rectangle` zeichnen. Allerdings müssen wir erst wissen, wie breit und hoch der Text auf dem Bildschirm ist, damit wir die Koordinaten des Rechtecks bestimmen können. Um die Abmessungen eines Textes zu ermitteln, stellt die Struktur `WinFontMetrics` die Funktion `dimensions` zur Verfügung, die Breite und Höhe in einem Tupel liefert.

Nun können wir das Rechteck um den Text zeichnen:

```
(w, h)     == dimensions(metr)(sampleText)
lowerRight == xc(upperLeft)+w @ yc(upperLeft)+h
box        == rectangle(upperLeft, lowerRight)
```

Mit den Selektoren `xc` und `yc` erhalten wir die x- bzw. y-Koordinate eines Punktes. Mittels der Breite `w` und Höhe `h` des Rechtecks können wir die untere rechte Ecke des Rechtecks berechnen.

Die Funktion `dimension` benötigt ein zusätzliches Argument `metr` des Typs `metrics`, das Informationen über die verwendete Schriftart enthält. Wir zeigen später, woher wir diese Informationen bekommen, vorläufig sehen wir sie als Parameter der Funktion `boxedText` an, die die Box und den inneren Text zusammenfasst:

```
FUN boxedText : metrics -> drawing
DEF boxedText(metr) == txt ++ box
    WHERE
      txt        == text(upperLeft)
                    with (text(sampleText) ++ anchor(north-west) ++ font(stdFont))
      upperLeft  == middle @ ("100" px)
      (w, h)     == dimensions(metr)(sampleText)
      lowerRight == xc(upperLeft)+w @ yc(upperLeft)+h
      box        == rectangle(upperLeft, lowerRight)
```

Zum Abschluß dieses Teils fassen wir noch den oberen und unteren Teil der Zeichnung zusammen:

```
FUN picture : metrics -> drawing
DEF picture(metr) == line ++ text ++ boxedText(metr)
```

5.3 Zeichenfläche und Button

Wir erstellen nun die Zeichenfläche und den „Exit“-Button. Sowohl Zeichenfläche als auch Button sind sog. *Views*, also graphische Anzeigeelemente. Views können, ähnlich wie `drawing`s, zu komplexeren Views kombiniert werden.

Zeichenfläche Die Zeichenfläche kann mit der Funktion `canvas` aus `WinCanvas` erstellt werden. Um die Zeichnung zu modifizieren, müssen wir einen `canvasEditor` an den View binden. Ähnlich den `drawing`s werden einem View Konfigurationen mittels `with` hinzugefügt, wobei wiederum einzelne Konfigurationen mit `++` zusammengesetzt werden können.

Wir nehmen fürs Erste den `canvasEditor` als gegeben an und erstellen die Zeichenfläche:

```
FUN pad : canvasEditor -> view
DEF pad(editor) == canvas with (size(canvasSize)
                                ++ edit(editor)
                                ++ background(canvasBackground))
```

Mit `size` legen wir die Grösse der Fläche fest, `edit` bindet den `canvasEditor` an den View und `background` legt die Hintergrundfarbe der Zeichenfläche fest.

Button Die Erstellung des Buttons erfolgt fast analog zur Zeichenfläche. Wir müssen allerdings einen *Emitter* an den Button binden, der uns mitteilt, dass der Button geklickt wurde. Dies geschieht mit der Konfiguration `bind(emit, act)`, wobei `act` der Wert ist, den der Emitter beim Klicken liefert.

Da wir in unserem Beispiel nur einen Button haben, reicht hier ein einzelner Wert aus, den wir durch den Datentyp `action` repräsentieren:

```
DATA action == exit
```

Die Funktion `exitButton` erstellt den Button-View:

```
FUN exitButton : emitter[action] -> view
DEF exitButton(emit) == button with (bind(emit, exit)
                                     ++ text("Exit"))
```

5.4 Erstellen der graphischen Umgebung

Die Definitionen der bisherigen Abschnitte waren rein-funktional, d. h. keine Werte vom Typ `com`. Natürlich ist eine graphische Oberfläche letztlich Ein-/Ausgabe. Diese Verbindung der rein-funktionalen Graphik-Beschreibung zur Aussenwelt haben wir an drei Stellen gemerkt:

1. Eingabe: Wir brauchen einen `emitter`, um einen Klick auf den „Exit“-Button zu registrieren
2. Ausgabe: Zum Zeichnen benötigen wir einen `canvasEditor`.
3. Eingabe: Da die Schriften vom Betriebssystem verwaltet werden, müssen wir letztlich dort nachfragen, um die gewünschte `metrics` zu erhalten.

Wir definieren uns einen Datentyp `gui`, der die benötigten Werte enthält:

```
DATA gui == gui(edit  : canvasEditor,
                emit  : emitter[action],
                title : denotation,
                metr  : metrics)
```

Zu Beginn des Programms müssen wir diese Werte initialisieren. Wir erledigen dies mit der Funktion `setupGui`, die die notwendigen Aufrufe erledigt:

```
FUN setupGui : com[gui]
DEF setupGui == canvasEditor          & (\\edit.
                emitter               & (\\emit.
                fontMetrics(stdFont)  & (\\metr.
                succeed(gui(edit, emit, "TextDemo", metr)))))
```

Sobald wir die gui-Datenstruktur initialisiert haben, können wir den Button und die Zeichenfläche zu einem Gesamtview komponieren:

```
FUN frame : gui -> view
DEF frame(G) == pad(edit(G))
                ^^
                exitButton(emit(G))
```

Neben der Komposition übereinander durch ^^ bietet OPAL WINDOWS noch viele weitere Möglichkeiten, um Views zu komponieren, z. B. << zur Anordnung nebeneinander.

5.5 Erzeugen des Fensters

Um einen View tatsächlich anzeigen zu können, benötigen wir ein Fenster, das wir mit Hilfe des Kommandos `window` aus `WinWindow` erstellen:

```
FUN setupWindow : gui -> com[window]
DEF setupWindow(G) == window(frame(G))               & (\\win.
                      set(win, titleName(title(G)))  & (\\ _ .
                      grab(win)                      & (\\ _ .
                      succeed(win))))
```

Dem Kommando `window` übergeben wir den View, der in dem Fenster angezeigt werden soll, i. d. F. `frame(G)`. Wir setzen noch zusätzlich den Titel des Fensters (mit `set`) and setzen den Eingabefokus auf das neue Fenster (mittels `grab`).

5.6 Anzeigen der Zeichnung

Bis jetzt haben wir nur einen Wert vom Typ `drawing`. Den müssen wir jetzt noch mit dem `canvasEditor` auf den Bildschirm bringen. Dies erledigt die Funktion `display`:

```
FUN drawText : gui -> com[void]
DEF drawText(G) ==
    display(edit(G), picture(metr(G)))
```

5.7 Kombination der Einzelteile

Nun müssen wir noch die einzelnen Bausteine geeignet zusammenfügen:

```
DEF textdemo == setupGui        & (\\G.
                setupWindow(G)  & (\\win.
                drawText(G)     &
                waitForTermination(G, win)))
```

Als letztes fehlt uns noch die Funktion `waitForTermination`, die wartet, bis der „Exit“-Button geklickt wird.

Mit Hilfe des Kommandos `await` können wir auf das Ereignis eines Emitters warten:

```
FUN waitForTermination : gui ** window -> com[void]
DEF waitForTermination(G, win) ==
    await(emit(G))  & (\\act.
    IF act exit? THEN release(win) & delete(win) & exit(0) FI)
```

Erhalten wir den Ergebniswert `act` des Emitters prüfen wir, ob es der erwartete `exit`-Wert ist, geben den Fokus mittels `release` frei, löschen das Fenster (`delete`) und beenden das Programm.

Da wir nur einen Button haben, kann der Emitter natürlich keinen anderen Wert als `exit` liefern. Wir haben dennoch diese allg. Form hier gewählt, um zu zeigen, wie man mit mehreren Buttons umgehen kann: durch die `bind`-Konfiguration ordnen wir jedem Button einen charakteristischen Wert zu, über den wir eine Fallunterscheidung machen können.

5.8 Bemerkung zum Thema Schriftarten

In OPAL WINDOWS sind nur zwei Schriftarten vordefiniert: `variable` und `fixed`, die eine Proportional- bzw. Festbreitenschriftart auswählen. Möchte man andere Schriftarten verwenden, kann die Funktion `font` verwendet werden, die den Namen einer Schriftart als Argument erwartet.

Die Namen der verfügbaren Schriftarten kann man sich bspw. mit `xlsfonts` anzeigen lassen. Hier ist ein Ausschnitt:

```
$ xlsfonts
-unregistered-texgyretermes-bold-i-normal--0-0-0-0-p-0-iso8859-15
-unregistered-texgyretermes-bold-r-normal--0-0-0-0-p-0-iso8859-1
-unregistered-texgyretermes-bold-r-normal--0-0-0-0-p-0-iso8859-15
-unregistered-texgyretermes-medium-r-normal--0-0-0-0-p-0-iso8859-15
-urw-century schoolbook l-bold-i-normal--0-0-0-0-p-0-iso8859-1
-urw-century schoolbook l-bold-i-normal--0-0-0-0-p-0-iso8859-15
-urw-century schoolbook l-bold-i-normal--0-0-0-0-p-0-iso8859-2
-urw-century schoolbook l-bold-r-normal--0-0-0-0-p-0-iso8859-1
-urw-century schoolbook l-bold-r-normal--0-0-0-0-p-0-iso8859-15
-urw-dingbats-regular-r-normal--0-0-0-0-p-0-adobe-fontspecific
-urw-nimbus mono l-bold-o-normal--0-0-0-0-p-0-iso8859-1
-urw-nimbus mono l-bold-o-normal--0-0-0-0-p-0-iso8859-15
-urw-nimbus mono l-bold-o-normal--0-0-0-0-p-0-iso8859-2
```

Auch das Programm `xfontsel` ist nützlich, um den Namen einer Schriftart herauszufinden.

Hinweis: In der Dokumentation der Bibliotheca Opalica wird fälschlicherweise auf eine Struktur `WinFontName` verwiesen, die nicht existiert, der o. g. Weg ist der einzige, um andere Schriftarten in OPAL WINDOWS nutzen zu können.

BIBLIOTHECA OPALICA

Aufgrund der Vielzahl der im Beispiel `TextDemo` verwendeten Funktionen und Datentypen führen wir hier nur die verwendeten Strukturen aus der Bibliotheca Opalica auf:

`WinButton`, `WinCanvas`, `WinCanvasEditor`, `WinConfig`, `WinEmitter`, `WinWindow`, `WinView`

---

6 Debugging

Hin und wieder ist es nützlich, sich an einer beliebigen Stelle im Programm einen Wert auf dem Terminal ausgeben zu lassen. Ausgabe funktioniert in OPAL allerdings nur in Kommandos, aber die meisten Funktionen haben nicht den Typ `com`.

Aus diesem Grund gibt es die Struktur `DEBUG`, die eine Funktion

```
FUN PRINT : bool ** denotation ** α -> α
```

zur Verfügung stellt, die Ausgaben an beliebiger Stelle erlaubt. Allerdings ist die Verwendung etwas umständlich.

> *Wichtig:* `PRINT` sollte nur zur Fehlersuche eingesetzt werden und hat in abgeschlossenem Code nichts verloren.

Sei beispielsweise das Programm

```
DEF debug == writeLine(‘(f(6)))

FUN f : nat -> nat
DEF f(x) == (x-1)*2 + (x+1)
```

gegeben, können wir mittels `PRINT` den Ausdruck `(x-1)*2` ausgeben:

```
FUN f : nat -> nat
DEF f(x) == LET e == PRINT(true, ‘((x-1)*2), (x-1)*2)
            IN e + (x+1)
```

Die Ausgabe auf dem Terminal ist:

```
$ ./debug
DEBUG PRINT:
10
17
```

Die ersten beiden Zeilen stammen von `PRINT`.

Weitere Informationen zu `DEBUG` gibt es im Handbuch des Opal-Compilers[4] auf S. 16.

[4] `https://projects.uebb.tu-berlin.de/opal/trac/raw-attachment/wiki/Documentation/userguide.pdf`

Listing 6: *TextDemo*

```
SIGNATURE TextDemo

IMPORT Com  ONLY com
       Void ONLY void

FUN textdemo : com[void]
```

```
IMPLEMENTATION TextDemo

IMPORT Com        COMPLETELY
       ComCompose COMPLETELY
       Denotation COMPLETELY
       Nat        COMPLETELY
       Real       COMPLETELY
       Seq        COMPLETELY
       Void       COMPLETELY

IMPORT WinButton       COMPLETELY
       WinCanvas       COMPLETELY
       WinCanvasEditor COMPLETELY
       WinConfig       COMPLETELY
       WinEmitter      COMPLETELY
       WinFontMetrics  COMPLETELY
       WinWindow       COMPLETELY
       WinView         COMPLETELY


DATA gui == gui(edit  : canvasEditor,
                emit  : emitter[action],
                title : denotation,
                metr  : metrics)

DATA action == exit


DEF textdemo == setupGui        & (\\G.
                setupWindow(G)  & (\\win.
                drawText(G)     &
                waitForTermination(G, win)))


-- ------------------------------------------------------------------------
-- Setup of window
-- ------------------------------------------------------------------------

FUN setupGui : com[gui]
DEF setupGui == canvasEditor          & (\\edit.
                emitter               & (\\emit.
                fontMetrics(stdFont)  & (\\metr.
                succeed(gui(edit, emit, "TextDemo", metr)))))
```

```
FUN setupWindow : gui -> com[window]
DEF setupWindow(G) == window(frame(G))               & (\\win.
                      set(win, titleName(title(G)))  & (\\ _ .
                      grab(win)                      & (\\ _ .
                      succeed(win))))


-- ---------------------------------------------------------------------
-- Main program
-- ---------------------------------------------------------------------

FUN drawText : gui -> com[void]
DEF drawText(G) ==
    display(edit(G), picture(metr(G)))


FUN picture : metrics -> drawing
DEF picture(metr) == line ++ text ++ boxedText(metr)


FUN line : drawing
DEF line == line(%(left @ ("40" px), right @ ("40" px)))

FUN text : drawing
DEF text == above ++ on ++ below
    WHERE
      above == text(left @ ("40" px))
               with (text(sampleText) ++ anchor(south) ++ font(stdFont))
      on    == text(middle @ ("40" px))
               with (text(sampleText) ++ anchor(center) ++ font(stdFont))
      below == text(right @ ("40" px))
               with (text(sampleText) ++ anchor(north-west) ++ font(stdFont))


FUN boxedText : metrics -> drawing
DEF boxedText(metr) == txt ++ box
    WHERE
      txt        == text(upperLeft)
                    with (text(sampleText) ++ anchor(north-west) ++ font(stdFont))
      upperLeft  == middle @ ("100" px)
      (w, h)     == dimensions(metr)(sampleText)
      lowerRight == xc(upperLeft)+w @ yc(upperLeft)+h
      box        == rectangle(upperLeft, lowerRight)


FUN waitForTermination : gui ** window -> com[void]
DEF waitForTermination(G, win) ==
    await(emit(G))  & (\\act.
    IF act exit? THEN release(win) & delete(win) & exit(0) FI)
```

```
-- ---------------------------------------------------------------------
-- The layout (views)
-- ---------------------------------------------------------------------

FUN frame : gui -> view
DEF frame(G) == pad(edit(G))
                ^^
                exitButton(emit(G))



FUN pad : canvasEditor -> view
DEF pad(editor) == canvas with (size(canvasSize)
                                ++ edit(editor)
                                ++ background(canvasBackground))


FUN exitButton : emitter[action] -> view
DEF exitButton(emit) == button with (bind(emit, exit)
                                     ++ text("Exit"))


-- ---------------------------------------------------------------------
-- Basic settings
-- ---------------------------------------------------------------------

FUN canvasWidth canvasHeight : real
DEF canvasWidth == "320" px
DEF canvasHeight == "240" px

FUN canvasSize : size
DEF canvasSize == canvasWidth x canvasHeight


FUN margin left right middle : real
DEF margin == "50" px
DEF left == margin
DEF right == canvasWidth-margin
DEF middle == canvasWidth/2


FUN canvasBackground : color
DEF canvasBackground == grey

FUN sampleText : denotation
DEF sampleText == "Hallo"

FUN stdFont : font
DEF stdFont == variable
```