Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung
Informatik I
Wintersemester 2006

Kapitel 2: Einführung in die Programmierung mit SML

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Elke Achtert, Arthur Zimek

Skript © 2006 Christian Böhm

http://www.dbs.ifi.lmu.de/Lehre/Info1

Inhalt (1)

1. Einführung
2. Ausdrücke, Werte, Typen und polymorphe Typüberprüfung
3. Präzedenz- und Assoziativregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken
4. Namen, Bindungen und Deklarationen
5. Fallbasierte Definition einer Funktion
6. Definition von rekursiven Funktionen

Inhalt (2)

7. Wiederdeklaration eines Namens – Statische Bindung – Umgebung
8. Totale und partielle Funktionen (Fortsetzung)
9. Kommentare
10. Die Standardbibliothek von SML
11. Beispiel: Potenzrechnung

Überblick

1. Einführung
2. Ausdrücke, Werte, Typen und polymorphe Typüberprüfung
3. Präzedenz- und Assoziativregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken
4. Namen, Bindungen und Deklarationen
5. Fallbasierte Definition einer Funktion
6. Definition von rekursiven Funktionen

SML-Sitzung

- SML-Aufruf: `sml`
- SML-Sitzung beenden: `Ctrl+D` bzw. `Strg+D`
- SML bietet eine dialogorientierte Benutzerschnittstelle:
  - Steht das Zeichen „`-`“ zu Beginn einer Zeile, dann
    - kann der Benutzer einen Ausdruck eingeben,
    - das Ende des Ausdrucks mit „`;`“ kennzeichnen und
    - die Auswertung des Ausdrucks mit „enter“ anfordern.
  - SML wertet den eingegebenen Ausdruck unter Verwendung der z. Zt. bekannten Definitionen aus und liefert den Wert in einer neuen Zeile.

Beispiel einer SML-Sitzung

achtert on kythera: /home/wiss/achtert

```
Standard ML of New Jersey, Version 110.0.6, October 31, 1999
val use = fn : string -> unit
-
- 2;
val it = 2 : int
-
- ~1;
val it = ~1 : int
-
- ~(~2);
val it = 2 : int
-
- 2 * 3 + 1;
val it = 7 : int
-
- (2 * 3) + 1;
val it = 7 : int

- 4 div 2;
val it = 2 : int
-
- 5 div 2;
val it = 2 : int
-
- 100 mod 4;
val it = 0 : int
-
- 012;
val it = 12 : int
-
-
- 
```

Datentyp „Ganze Zahl“ (1)

- Jeder Ausdruck in SML besitzt einen (Daten-)Typ.
- In diesem Kapitel: vorwiegende Behandlung von Ausdrücken, deren Werte ganze Zahlen oder Boolesche Werte sind (ausführliche Einführung der vordefinierten Typen von SML in Kapitel 6)
- „`int`“ bezeichnet den Typ „integer“ oder „ganze Zahl“
- „`it`“ bezeichnet den unbenannten Wert des Ausdrucks, dessen Auswertung angefordert wird

Datentyp „Ganze Zahl“ (2)

- Bei ganzen Zahlen sind führende Nullen zulässig, z.B. ist
  - `012` alternative Notation für `12`
  - `~0012` alternative Notation für `~12`
- Vordefinierte (Infix-) Operationen über den natürlichen Zahlen: `+`, `-`, `*`, `div` und `mod`
- Beachtung der Präzedenzen: `2*3+1` steht z.B. für `(2*3)+1`
- Vorsicht: ~ (Vorzeichen für negative ganze Zahlen) und – (Subtraktion) sind nicht austauschbar
- ~ ist ein unärer Operator, `-`, `+`, `*`, `div` und `mod` sind binäre Operatoren

Gleichheit für ganze Zahlen

- Zum Vergleich von ganzen Zahlen bietet SML die vordefinierte Funktion =

achtert on kythera: /home/wiss/achtert

```
- 2 = 2;
val it = true : bool
-
- 2 = 3;
val it = false : bool
-
-
```

- Eine Funktion, die als Wert entweder `true` oder `false` liefert, wird *Prädikat* oder *Test* genannt.

Datentyp „Boolescher Wert“ (1)

- `true` oder `false` sind so genannte Wahrheitswerte oder Boolesche Werte
- Operationen über Booleschen Ausdrücken `not` (präfix notiert), `andalso` (infix notiert), `orelse` (infix notiert)
- `not` ist ein `unärer` Operator, `orelse` und `andalso` sind binäre Operatoren

achtert on kythera: /home/wiss/achtert

```
- true;
val it = true : bool
-
- not true;
val it = false : bool
-
- not (not false);
val it = false : bool
-
- true andalso not false;
val it = true : bool
-
- false orelse true;
val it = true : bool
-
- not (2 = 3);
val it = true : bool
-
-
```

Datentyp „Boolescher Wert“ (2)

- Der Ausdruck `not not false` kann nicht ausgewertet werden, weil er von SML wie `(not not) false` verstanden wird.
- `(not not) false` ist aus zwei Gründen inkorrekt:
  1. Die Teilausdrücke `(not not)` und `false` sind nicht mit einer Operation verbunden, `(not not) false` ist genauso sinnlos wie `2 4`.
  2. Der Teilausdruck `(not not)` ist inkorrekt gebildet, weil die erste Negation auf keinen Booleschen Ausdruck angewandt wird, `(not not)` ist genauso sinnlos wie `(~~)`. Aus denselben Gründen ist z.B. auch `~~5` inkorrekt.

Gleichheit für Boolesche Werte (1)

- Boolesche Ausdrücke können mit der vordefinierten Funktion = verglichen werden

```
achtert on kythera: /home/wiss/achtert
- true = not (not true);
val it = true : bool
-
- false = (false andalso not true);
val it = true : bool
-
- 
```

Gleichheit für Boolesche Werte (2)

- Bemerkung: Der Vergleich von Wahrheitswerten mit = ist fast immer schlechter Programmierstil:
  - Statt `if Bedingung = true then Ausdruck else Ausdruck'` ist es einfacher und übersichtlicher
    `if Bedingung then Ausdruck else Ausdruck'`
    zu schreiben.
  - In ähnlicher Weise lässt sich
    `if Bedingung = false then Ausdruck else Ausdruck'` zu
    `if not Bedingung then Ausdruck else Ausdruck'` oder
    `if Bedingung then Ausdruck' else Ausdruck` vereinfachen.

Überladen (1)

- Gleichheit für Boolesche Ausdrücke und Gleichheit für ganze Zahlen sind grundverschiedene Funktionen, da ihre Argumente verschiedene Typen besitzen.
- Zur Feststellung der Gleichheit zieht das SML-System die Typen der Operanden in Betracht.
- Wird derselbe Name oder dasselbe Symbol (hier: „=“) zur Bezeichnung unterschiedlicher Operationen oder Funktionen verwendet, die vom System unterschieden werden, so spricht man von *Überladen* (*Overloading*).

Überladen (2)

- Weitere Fälle von Überladen in SML sind die arithmetischen Operationen für ganze und für reelle Zahlen: + und *.

```
achtert on kythera: /home/wiss/achtert
- 2 + 3;
val it = 5 : int
-
- 2.1 + 3.3;
val it = 5.4 : real
-
- 2 * 3;
val it = 6 : int
-
- 2.1 * 3.3;
val it = 6.93 : real
-
-
```

- Beachte: In SML sind (wie in den meisten Programmiersprachen) ganze Zahlen und reelle Zahlen unterschiedliche Typen.

Überladen (3)

- Grund: Unterschiedliche Darstellungen der Typen
  - Ganzzahlarithmetik in SML kennt keine Rundung und damit auch keine Ungenauigkeit
  - Genauigkeit der Arithmetik mit reellen Zahlen ist abhängig von der Gleitkommahardware des Computers
- Addition einer ganzen Zahl mit einer reellen Zahl führt zu einer Fehlermeldung

```
achtert on kythera: /home/wiss/achtert
- 2 + 4.83;
stdIn:127.1-127.9 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
  in expression:
    2 + 4.83
-
```

Weitere Typen

- SML bietet weitere häufig benötigte Typen, wie z.B. „Zeichen“ (wie `a`, `b`, `c`, usw.) und „Zeichenfolge“: siehe Kapitel 6

- SML ermöglicht die Definition von Typen, die für ein praktisches Problem maßgeschneidert werden können (z.B. eine beliebige Notenskala oder die Tage der Woche in beliebiger Sprache): siehe Kapitel 9

Vergleichsfunktionen für ganze und reelle Zahlen (1)

- Überladene vordefinierte Prädikate für ganze Zahlen und reelle Zahlen:
  - < (echt kleiner)
  - > (echt größer)
  - <= (kleiner gleich)
  - >= (größer gleich)
- Vordefiniertes Prädikat für ganze Zahlen: <> (Negation der Gleichheit)

Vergleichsfunktionen für ganze und reelle Zahlen (2)

- Vorsicht: = und <> sind für reelle Zahlen nicht zulässig, hier bietet SML `Real.compare(x,y)` an:
- `Real.compare` ist kein Prädikat (liefert keine booleschen Werte), sondern liefert Werte eines bisher nicht behandelten Typs `order`.
- SML bietet auch die Gleichheitsfunktion `Real.==` für reelle Zahlen an, die den Typ `order` nicht verwendet

```
achtert on kythera: /home/wiss/achtert
- Real.compare(1.0, 7.0);
val it = LESS : order
-
- Real.compare(100.0, 1.0);
val it = GREATER : order
-
- Real.compare(1.0, 1.0);
val it = EQUAL : order
-
```

```
achtert on kythera: /home/wiss/achtert
- Real.==(2.5, 2.5);
val it = true : bool
-
- Real.==(2.5, 3.0);
val it = false : bool
-
```


Weitere nützliche Funktionen für ganze Zahlen

- `Int.abs`: Betrag einer ganzen Zahl
- `Int.min`: Minimum zweier ganzer Zahlen
- `Int.max`: Maximum zweier ganzer Zahlen
- `Int.sign`: „Vorzeichen“ einer ganzen Zahl

```
achtert on kythera: /home/wiss/achtert
- Int.abs(~4);
val it = 4 : int
-
- Int.min(5,2);
val it = 2 : int
-
- Int.max(5,3);
val it = 5 : int
-
- Int.sign(0);
val it = 0 : int
-
- Int.sign(~5);
val it = ~1 : int
- Int.sign(6);
val it = 1 : int
-
```

Überblick

1. Einführung
2. Ausdrücke, Werte, Typen und polymorphe Typüberprüfung
3. Präzedenz- und Assoziativregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken
4. Namen, Bindungen und Deklarationen
5. Fallbasierte Definition einer Funktion
6. Definition von rekursiven Funktionen

Ausdrücke, Werte und Typen (1)

- SML wertet Ausdrücke aus, ein Ausdruck kann dabei atomar (z.B. `2` oder `false`) oder zusammensetzt sein (z.B. `not (false andalso true)`oder `2 + 4`).
- Jeder korrekt gebildete Ausdruck hat einen Typ, aber nicht immer einen Wert. Ein Typ ist eine Menge von Werten (z.B. die Menge der ganzen Zahlen).
- Hat der Ausdruck einen Wert, dann ist der Wert ein Element des Typs des Ausdrucks. Manche Ausdrücke wie z.B. `1 div 0`, in denen nicht-totale Funktionen verwendet werden, haben keinen Wert.

Ausdrücke, Werte und Typen (2)

- Atomare Ausdrücke wie `true` und `false` kann man oft mit ihren Werten identifizieren. Diese Betrachtungsweise ist aber in vielen (auch einfachen) Fällen problematisch:
  - `02` und `2` sind z.B. verschiedene atomare Ausdrücke, die denselben Wert haben.
  - zusammengesetzte Ausdrücke sind nie mit ihren Werten identisch, z.B. besitzt der Ausdruck `3+2` den Wert `5`
- Deshalb: Strikte Unterscheidung zwischen Ausdruck und Wert!

Ausdrücke, Werte und Typen (3)

- Auch Operationen und allgemeine Funktionen haben Typen, z.B.
  - + ist eine Funktion, die als Argumente zwei Ausdrücke vom Typ „ganze Zahl" erhält und einen Wert vom Typ „ganze Zahl" liefert
  - die Gleichheit auf ganzen Zahlen ist eine Funktion, die als Argumente zwei Ausdrücke vom Typ „ganze Zahl" erhält und einen Wert vom Typ „boolescher Wert" liefert
  - Man schreibt:

```
+ : (int, int) -> int
= : (int, int) -> bool
```

Typen in Programmiersprachen (1)

- *Schwach typisierte Programmiersprachen* akzeptieren einen Ausdruck wie `8.0+1` (Summe einer reellen und einer ganzen Zahl) und wandeln die natürliche Zahl `1` automatisch in eine reelle Zahl um → (automatische) „Typanpassung“ (z.B. Prolog, Lisp).
- *Stark (oder streng) typisierte Programmiersprachen* verlangen vom Programmierer, dass er für jeden Namen einen Typ explizit angibt und jede notwendige Typanpassung selbst programmiert (z.B. Pascal, Modula).

Typen in Programmiersprachen (2)

- SML verfolgt einen Mittelweg zwischen schwach und stark typisierten Programmiersprachen: Anstatt die explizite Angabe von Typen zu verlangen, ermittelt SML wenn möglich selbst, was die Typen der Bezeichner sind → „*polymorphe Typüberprüfung*“.
- Vordefinierte Funktionen `real` und `round` zur Typanpassung zwischen reellen und ganzen Zahlen:

achtert on kythera: /home/wiss/achtert

```
- real(1);
val it = 1.0 : real
-
- round(8.12);
val it = 8 : int
-
- round(8.99);
val it = 9 : int
-
- round(8.0);
val it = 8 : int
-
```

Beachte: Der Zweck von `round` ist nicht nur die Typanpassung, sondern auch das Auf- und Abrunden.

Überblick

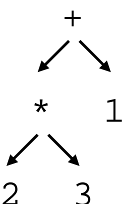
1. Einführung
2. Ausdrücke, Werte, Typen und polymorphe Typüberprüfung
3. **Präzedenz- und Assoziativregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken**
4. Namen, Bindungen und Deklarationen
5. Fallbasierte Definition einer Funktion
6. Definition von rekursiven Funktionen

Präzedenzen und Assoziativregeln für Operatoren

- *Präzedenzen* von Operatoren legen fest, welche implizite Klammerung bei unzureichend oder gar nicht geklammerten Ausdrücken gemeint sein soll, z.B. steht der Ausdruck `2*3+1` für `(2*3)+1`. Man sagt, dass `*` stärker bindet als `+`.
- *Assoziativregeln* legen fest, ob fehlende Klammerungen von links oder rechts her einzusetzen sind, d.h. ob `2+3+4` für `(2+3)+4` oder für `2+(3+4)` steht. In SML sind `+` und `*` linksassoziativ.
- Der Wert ist i.a. von der Assoziativregel abhängig: z.B. `10-7-2`
  - linksassoziativ (wie in SML): Wert `1`
  - rechtsassoziativ: Wert `5`
- In manchen Programmiersprachen sind Assoziativregeln auch deshalb wichtig, weil sie die Reihenfolge der Auswertung bestimmen.

Syntaxanalyse (1)

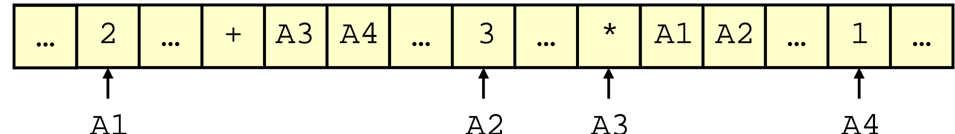
- Ausdrücke, die SML zur Auswertung weitergereicht werden, sind linear, da sie aus einer Folge von Zeichen bestehen, z.B. `(2*3)+1` und `2*3+1`
- Bevor solche Ausdrücke von SML ausgewertet werden, wird deren Syntax analysiert. Die Syntaxanalyse des linearen Ausdrucks `(2*3)+1` führt zur Bildung einer baumartigen Struktur im Speicher wie:



Die gerichteten Kanten stellen Speicheradressen dar (s. Info III).

Syntaxanalyse (2)

- Im (linear angeordneten) Speicher ist der Baum wie folgt repräsentiert (A_i = Speicheradressen):



- Die Syntaxanalyse ist aus zwei Gründen notwendig
  1. Sie ermöglicht die Auslegung (Interpretation) von unvollständig geklammerten Ausdrücken (wie z.B. `4+5+6`).
  2. Sie ersetzt die sog. „konkrete Syntax“ von Ausdrücken (d.h. die vom Programmierer verwendete Darstellung) durch die sog. „abstrakte Syntax“ (d.h. die Repräsentation im Speicher durch Bäume), die von SML zur Auswertung verwendet wird.

Syntaxanalyse (3)

- Beachte: die Baumdarstellung ist genauso wie die lineare konkrete Syntax eine abstrakte Wiedergabe der abstrakten Syntax.
- Da `2*3+1` in SML für `(2*3)+1` steht, führt die Syntaxanalyse von `2*3+1` zur Bildung desselben Baums wie die Syntaxanalyse von `(2*3)+1`.
- Die abstrakte Syntax ist nur dann wünschenswert, wenn die interne Repräsentation der Ausdrücke im Speicher eine Rolle spielt, sonst ist die konkrete Syntax von Vorteil, da sie für den Menschen einfacher (v.a. zu schreiben) ist.

Überblick

1. Einführung
2. Ausdrücke, Werte, Typen und polymorphe Typüberprüfung
3. Präzedenz- und Assoziativregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken
4. **Namen, Bindungen und Deklarationen**
5. Fallbasierte Definition einer Funktion
6. Definition von rekursiven Funktionen

Konstantendeklaration (1)

- Eine *Deklaration* bindet einen Wert an einen Namen. Mögliche Werte sind u.a. Konstanten und Funktionen.
- Eine Konstantendeklaration bindet eine Konstante an einen Namen.
- Bsp.:

achtert on kythera: /home/wiss/achtert

```
- val zwei = 2;
val zwei = 2 : int
-
```

- Anstelle von Konstantendeklaration spricht man auch von Wertdeklaration, daher das Kürzel `val` („value“).

Konstantendeklaration (2)

- Nach der Konstantendeklaration kann der Name `zwei` genauso wie die Konstante `2` verwendet werden:

achtert on kythera: /home/wiss/achtert

```
-  zwei + zwei;
val it = 4 : int
-
- zwei + 8;
val it = 10 : int
-
-
```

- Beachte: alle Konstantendeklarationen sind Wertdeklarationen, aber nicht alle Wertdeklarationen sind Konstantendeklarationen (vgl. nächste Folie).

Funktionsdeklaration (1)

- Eine Funktionsdeklaration (auch Funktionsdefinition) bindet eine Funktion an einen Namen.
- Bsp.:

```
achtert on kythera: /home/wiss/achtert
- fun zweimal(x) = 2 * x;
val zweimal = fn : int -> int
-
```

- SML gibt anstelle des eigentlichen Wertes des Namens `zweimal` das Kürzel „`fn`“ (für Funktion) und den Typ der Funktion an, hier `int -> int`.
- Gleichwertige Deklarationen sind
  - `fun zweimal (x) = 2 * x;`
  - `fun zweimal x = 2 * x;`

Funktionsdeklaration (2)

- Der *Typ* der Funktion `zweimal` wird wie folgt ermittelt:
  - Da `2` eine ganze Zahl ist, steht die überladene Operation `*` für die Multiplikation ganzer Zahlen, folglich muss `x` vom Typ `int` sein (daher `int ->`).
  - Da `*` die Multiplikation ganzer Zahlen ist, ist der von `zweimal` berechnete Wert eine ganze Zahl (daher `-> int`).
  - Die Typermittlung `int -> int` der Funktion `zweimal` ist ein Beispiel der „Polymorphen Typermittlung“ von SML!
- Der *eigentliche Wert* des Namens `zweimal` ist die Funktion, die als Eingabe eine ganze Zahl erhält und das Doppelte dieser Zahl liefert.

Funktionsdeklaration (3)

- Nachdem eine Funktion deklariert wurde, kann sie aufgerufen werden:

```
achtert on kythera: /home/wiss/achtert
- zweimal(8);
val it = 16 : int
-
- zweimal(zweimal(8));
val it = 32 : int
-
```


Funktion als Wert – Anonyme Funktion

- Für SML ist eine Funktion ein Wert, daher kann das Deklarationskonstrukt `val` verwendet werden, um einen Namen an eine (anonyme) Funktion zu binden.
- Die Funktion `zweimal` kann z.B. wie folgt definiert werden:

```
achtert on kythera: /home/wiss/achtert
- val zweimal = fn x => 2 * x;
val zweimal = fn : int -> int
-
```

  - An den Namen `zweimal` wird dabei die anonyme Funktion gebunden, die eine Zahl `x` als Argument erhält und `2 * x` liefert.
  - Der Teil `fn x => 2 * x` definiert eine anonyme Funktion.
  - `fn` wird oft „lambda“ (λ) ausgesprochen
- VORSICHT: Verwechseln Sie die Konstrukte `fn` und `fun` von SML nicht!

Formale und aktuelle Parameter einer Funktion

- In der Funktionsdeklaration
  ```
  fun zweimal(x) = 2 * x;
  ```
  wird `x` formaler Parameter (der Funktionsdeklaration) genannt.
- Im Funktionsaufruf `zweimal(8)` wird `8` aktueller Parameter (des Funktionsaufrufs) genannt.
- Formale Parameter besitzen in Funktionsdeklarationen eine ähnliche Bedeutung wie Pronomen in natürlichen Sprachen.

Rumpf oder definierter Teil einer Funktionsdeklaration

- Der Rumpf oder definierte Teil einer Funktionsdeklaration ist der Teil nach dem Zeichen „=“.
- In der Funktionsdeklaration
  ```
  fun zweimal(x) = 2 * x;
  ```
  ist der Rumpf `2 * x`.

Typ-Constraints (1)

- Da x + x = 2 * x, hätte man die Funktion zweimal wie folgt definieren können:
  ```
  fun zweimal(x) = x + x;
  ```
- Nicht alle SML-Systeme nehmen eine solche Deklaration als korrekt an, da der Typ des formalen Parameters `x` nicht eindeutig feststeht:
  - Manche Systeme nehmen an, dass `x` den Typ ganze Zahl hat, weil sie im Zweifel annehmen, dass + für die Addition von ganzen Zahlen steht.
  - Andere SML-Systeme treffen keine solche Annahme und verwerfen die o.a. Funktionsdeklaration als inkorrekt.

Typ-Constraints (2)

- Typ-Constraints (auch Typisierungsausdrücke) ermöglichen es, die fehlenden Informationen anzugeben.
- Angabe des Ergebnistyps:
  ```
  achtert on kythera: /home/wiss/achtert
  - fun zweimal x: int = x + x;
  val zweimal = fn : int -> int
  -
  ```
- Angabe des Parametertyps:
  ```
  achtert on kythera: /home/wiss/achtert
  - fun zweimal(x: int) = x + x;
  val zweimal = fn : int -> int
  -
  ```
- Angabe des Ergebnis- und Parametertyps:
  ```
  achtert on kythera: /home/wiss/achtert
  - fun zweimal(x : int): int = x + x;
  val zweimal = fn : int -> int
  -
  ```

Typ-Constraints (3)

- Mit einem Typ-Constraint kann die folgende Funktion für reelle Zahlen definiert werden:

```
achtert on kythera: /home/wiss/achtert
fun reell_zweimal x:real = x + x;
val reell_zweimal = fn : real -> real
-
- val pi = 3.1416;
val pi = 3.1416 : real
-
- reell_zweimal pi;
val it = 6.2832 : real
-
- 
```

- Beachte: vor und nach dem Zeichen „:“ in einem Typ-Constraint sind ein oder mehrere Leerzeichen zulässig.

Syntax von Namen (1)

- SML unterscheidet zwischen *alphabetischen* und *symbolischen* Namen, je nachdem wie sie gebildet werden.
- Alphabetische Namen beginnen mit einem Buchstaben, dem endlich viele (auch 0) Buchstaben `(a…zA…Z)`, Ziffern(`012…9`), Underscore (_), Hochkommata (single quote: ´) folgen.
- Symbolische Namen sind (endliche) Folgen der Zeichen `! % & $ # + - * / : < = > ? @ \ ~ ‘ ^` und `|`.

Syntax von Namen (2)

- Sowohl alphabetische, als auch symbolische Namen können in Konstanten- und Funktionsdeklarationen verwendet werden.

```
- val @#!@@@ = 12;
val @#!@@@ = 12 : int
-
- fun $@#? x = 5 * x;
val $@#? = fn : int -> int
-
```

- VORSICHT: Die folgenden symbolischen Namen haben in SML eine vordefinierte Bedeutung: `: | = => -> #`

Dateien laden

- Funktionsdeklarationen können in einer Datei (z.B. meine_datei.sml) gespeichert werden, die dann wie folgt geladen (eingelesen) werden kann:

```
- use("meine_datei.sml");
[opening meine_datei.sml]
val it = () : unit
-
```

- Dabei ist `()` („unity") der einzige Wert des besonderen Datentyps `unit`.
- `unit` liefert einen Wert für Funktionsaufrufe, die eigentlich keinen Wert berechnen, sondern einen Nebeneffekt bewirken (im Falle der vordefinierten Funktion `use` z.B. das Einlesen einer Datei).

Überblick

1. Einführung
2. Ausdrücke, Werte, Typen und polymorphe Typüberprüfung
3. Präzedenz- und Assoziativregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken
4. Namen, Bindungen und Deklarationen
5. **Fallbasierte Definition einer Funktion**
6. Definition von rekursiven Funktionen

if-then-else (1)

- SML ermöglicht fallbasierte Funktionsdefinitionen.
- Eine Funktion `vorzeichen`, die der vordefinierten Funktion `Int.sign` entspricht, kann z.B. wie folgt definiert werden:

```
achtert on kythera: /home/wiss/achtert
- fun vorzeichen(x: int) = if x > 0 then 1
=                          else if x < 0 then ~1
=                          else 0;
val vorzeichen = fn : int -> int
-
```

- Das Konstrukt `if Test then E1 else E2` stellt die Anwendung einer wie folgt definierten Funktion auf `Test` dar: `(fn true => E1 | false => E2)`
- `if Test then E1 else E2` steht also für `(fn true => E1 | false => E2)(Test)`

if-then-else (2)

- Im Gegensatz zu (imperativen) Programmiersprachen wie Pascal ist in SML der `else`-Teil nicht abdingbar. Grund: ein Ausdruck ohne `else`-Teil wie `if B then A` hätte keinen Wert, wenn die Bedingung `B` den Wert `false` hätte, was in der funktionalen Programmierung unmöglich ist.
- In einem SML-Ausdruck `if B then A1 else A2` müssen `A1` und `A2` denselben Wert besitzen

Pattern Matching („Musterangleich“) (1)

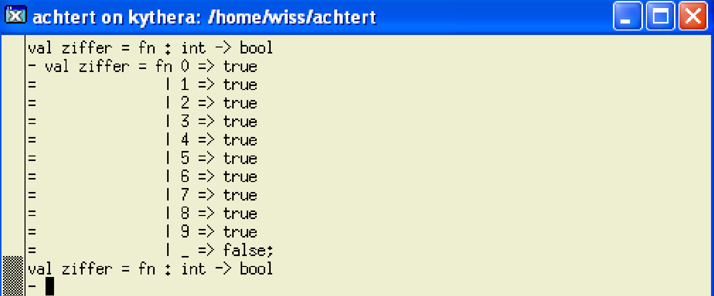
- In der Definition der anonymen Funktion `(fn true => E1 | false => E2)` sind zwei Aspekte bemerkenswert:
  1. „|“ drückt eine Alternative aus
  2. Die Ausdrücke `true` und `false` stellen sog. Muster (Patterns) dar. Matcht der Wert des aktuellen Parameters mit dem ersten Muster, so wird der Wert des Ausdrucks `E1` geliefert. Ansonsten wird getestet, ob der Wert des aktuellen Parameters mit dem zweiten Muster matcht.

Pattern Matching („Musterangleich“) (2)

- Kommen mehr als zwei Fälle vor, werden die Muster sequentiell in der Reihenfolge der Definition probiert, bis eines mit dem Wert des aktuellen Parameters matcht.
- Das Muster _ (Wildcard) stellt einen Fangfall dar, d.h. es matcht mit jedem möglichen Wert des aktuellen Parameters. Das Wildcard-Symbol wird nicht im Rumpf eines Falles (also hinter =>) verwendet.

Pattern Matching („Musterangleich“) (3)

- Das folgende Prädikat liefert `true`, wenn es auf eine ganze Zahl angewandt wird, die eine (nicht negierte) Ziffer ist:



- VORSICHT: Ein Muster ist kein Test, wie etwa `(x < 0)`, sondern repräsentiert mögliche Werte des Parameters.

Überblick

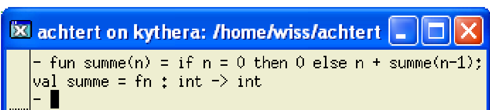
1. Einführung
2. Ausdrücke, Werte, Typen und polymorphe Typüberprüfung
3. Präzedenz- und Assoziativregeln, Notwendigkeit der Syntaxanalyse, Baumdarstellung von Ausdrücken
4. Namen, Bindungen und Deklarationen
5. Fallbasierte Definition einer Funktion
6. **Definition von rekursiven Funktionen**

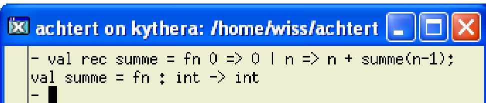
Rekursive Berechnung der Summe der *n* ersten ganzen Zahlen

- Die Funktion `summe`, die zu jeder natürlichen Zahl n die Summe aller natürlichen Zahlen von 0 bis einschließlich n liefert, kann u.a. wie folgt definiert werden:

$$\text{summe}(n) = \begin{cases} 0 & \text{falls } n = 0 \\ n + \text{summe}(n-1) & \text{falls } n > 0 \end{cases}$$

- In SML kann `summe` in einer der folgenden Weisen programmiert werden:





- Beachte: Bei der *Wertdefinition* einer rekursiven Funktion, muss `rec` verwendet werden..

Effiziente Berechnung der Summe der *n* ersten ganzen Zahlen

- Die Funktion `summe` kann auch wie folgt definiert werden: summe(n) = n * (n + 1) / 2
- Diese Definition führt zu wesentlich effizienteren Berechnungen, da sie für jedes n nur drei Grundoperationen verlangt.
- Diese Definition kann in SML wie folgt programmiert werden:

```
achtert on kythera: /home/wiss/achtert
- fun summe(n) = n * (n + 1) div 2;
val summe = fn : int -> int
-
```

- Wir untersuchen nun die Korrektheit dieser Implementierung und wiederholen hierzu nochmals die Technik des Induktionsbeweises.

Vollständige Induktion (1)

- Wenn für eine Eigenschaft E, die in Abhängigkeit von $n \in N$ formuliert werden kann, gilt:
  (i) $E(0)$ ist wahr;
  (ii) $E(m)$ ist wahr für ein $m \in N \Rightarrow E(m+1)$ ist wahr,
  dann folgt: $E(n)$ ist wahr für alle $n \in N$.
- (i) heißt *Induktionsanfang* (oder *Induktionsbasis*).
- (ii) heißt *Induktionsschluss* (oder *Induktionsschritt*).
- „$E(m)$ ist wahr für ein $m \in N$“ heißt *Induktionsvoraussetzung* (oder *Induktionsannahme*).
- Die vollständige Induktion gehört zu den unabdingbaren Techniken der Informatik.

Vollständige Induktion (2)

Induktiver Beweis der Korrektheit von

$$\text{summe}(n) = n * (n + 1) / 2 :$$

- *Induktionsanfang: m*=0

  $$\text{summe}(0) = 0 * (0 + 1) / 2 = 0$$

- *Induktionsschritt*
  - *Induktionsannahme*: summe(m) = $m * (m + 1) / 2$
  - z.z.: summe(m + 1) = $(m + 1)*(m + 2) / 2$

    Beweis: summe(m + 1) = $m + 1$ + summe(m) =
    $m + 1 + (m * (m + 1) / 2) =$
    $[2(m + 1) + (m * (m + 1)] / 2 =$
    $(m + 1) * (m + 2) / 2$ **qed.**

Alternativer Beweis (1)

- Sei $n \in N$
- *Fall 1*: n ist gerade
  - Die ganzen Zahlen von 1 bis n können in Paaren (n,1), (n-1,2), (n-2,3), … gruppiert werden.
  - Das letzte solcher Paare ist ((n/2)+1, n/2), weil kein weiteres solches Paar (a,b) die beiden Eigenschaften $a+b = n+1$ und $a \leq b$ besitzt (*).
  - Die Summe der Zahlen jedes Paares ist n+1 und es gibt n/2 solche Paare, also summe(n) = $n * (n + 1) / 2$.

Alternativer Beweis (2)

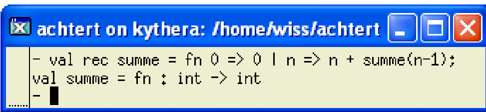
- *Fall 2*: *n* ist ungerade
  - Die ganzen Zahlen von 0 bis *n* können in Paaren (n,0), (n-1,1), (n-2,2), … gruppiert werden.
  - Das letzte solcher Paare ist ((n/2)+1, n/2) (**).
  - Die Summe der Zahlen jedes Paares ist *n* und es gibt (n+*1*)/2 solche Paare, also summe(n) = n * (n + 1) / 2.

- Bemerkung: Die Aussagen (*) und (**) verlangen im Grunde (einfache) Induktionsbeweise, die hier der Übersichtlichkeit halber ausgelassen werden

Terminierungsbeweis (1)

- Die Programmierung von rekursiven Funktionen kann (z.B. durch Denk- oder Programmierfehler) zu einer nichtterminierenden Funktion führen.
  Bsp.: `fun s(n) = n + s(n+1);`
- Die Terminierung einer rekursiven Funktion wie z.B. `summe` kann mit Hilfe der vollständigen Induktion gezeigt werden.
- In der Regel gibt es unendlich (oder zu viele) mögliche Aufrufparameter, so dass durch Testen zwar Fehler gefunden werden können, allerdings ohne Garantie, dass alle Fehler gefunden wurden. Deshalb sind Beweise unabdingbare Bestandteile der Programmentwicklung.

Terminierungsbeweis (2)



Terminierungsbeweis für `summe`

- *Induktionsanfang: m*=0
  `summe(0)` terminiert, weil nach Funktionsdeklaration `summe(0)` den Wert 0 liefert
- *Induktionsschritt*:
  - *Induktionsannahme*: `summe(`*m*`)` terminiert für eine natürliche Zahl $m \in N$
  - z.z: `summe(m+1)` terminiert
    Beweis:
    - Nach Funktionsdeklaration liefert der Aufruf `summe(m+1)` den Wert von `m+1+summe(m)`.
    - Nach Induktionsannahme terminiert der Aufruf `summe(m)`.
    - Folglich terminiert auch der Aufruf `summe(m+1)`. **qed.**

Überblick

7. Wiederdeklaration eines Namen – Statische Bindung – Umgebung
8. Totale und partielle Funktionen (Fortsetzung)
9. Kommentare
10. Die Standardbibliothek von SML
11. Beispiel: Potenzrechnung

Wiederdeklaration eines Namen

- Betrachten wir die folgende Sitzung

achtert on kythera: /home/wiss/achtert

```
- val zwei = 2;
val zwei = 2 : int
-
- fun zweimal(n) = zwei * n;
val zweimal = fn : int -> int
-
- zweimal(9);
val it = 18 : int
-
- val zwei = 0;
val zwei = 0 : int
-
- zweimal(9);
val it = 18 : int
-
- fun zweimal'(n) = zwei * n;
val zweimal' = fn : int -> int
-
- zweimal'(9);
val it = 0 : int
-
```

- Es ist zulässig, die Bindung eines Wertes (Konstante oder Funktion) an einen Namen durch eine neue Deklaration zu ändern (*Wiederdeklaration*).
- Wird der Name in einer Deklaration verwendet, dann gilt seine letzte Bindung an einen Wert.

Statische und dynamische Bindung

- Die Wiederdeklaration eines Namens gilt jedoch nicht für Deklarationen, die diesen Namen *vor* der Wiederdeklaration verwendet haben.
- So steht im Beispiel `zwei` für `2` in der Deklaration der Funktion `zweimal`, für `0` in der Deklaration der Funktion `zweimal'`.
- Man sagt, die Bindung in SML ist *statisch* (oder lexikalisch).
- Hätte die Wiederdeklaration eines Namens *N* Einfluss auf Funktionen, deren Rümpfe sich auf *N* beziehen, so würde man von *dynamischer* Bindung sprechen.

Umgebung (1)

- Das SML-System verwaltet mit jeder Sitzung und jeder eingelesen Datei eine geordnete Liste der Gestalt Name=Wert (dargestellt als Paare (Name,Wert)). Diese Liste heißt *Umgebung*.
- Jede neue Deklaration eines Wertes `W` für einen Namen `N` führt zu einem neuen Eintrag `N=W` am Anfang der Umgebung.
- Um den Wert eines Namens zu ermitteln, wird die Umgebung von Anfang an durchlaufen. So gilt immer als Wert eines Namens `N` derjenige Wert, der als letztes angegeben wurde.

Umgebung (2)

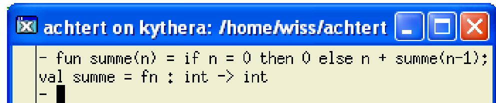
- Kommt ein Name `A` im Wertteil `W` einer Deklaration `val N=W` oder `val rec N=W` oder `fun N=W` vor, so wird der Wert von `A` ermittelt und in `W` anstelle `A` eingefügt, bevor der Eintrag für `N` in der Umgebung gespeichert wird.
- Deshalb verändert eine spätere Widerdeklaration von `A` den Wert von `N` nicht.

Überblick

7. Wiederdeklaration eines Namen – Statische Bindung – Umgebung
8. Totale und partielle Funktionen (Fortsetzung)
9. Kommentare
10. Die Standardbibliothek von SML
11. Beispiel: Potenzrechnung

Totale und partielle Funktion (Fortsetzung)

- Die rekursive Funktion `summe`



  ist über den ganzen Zahlen nicht total, weil ein Aufruf von `summe` mit einem nicht positiven Eingabeparameter (z.B. `~25`) nicht terminiert. Über den natürlichen Zahlen ist die Funktion aber total.
- Um sicherzustellen, dass die Funktion auch nur mit den entsprechenden Parametern aufgerufen wird, ist es ist wichtig, zu ermitteln, über welchen Bereich eine programmierte Funktion total ist. Oftmals werden zudem weitere Eigenschaften der Funktion nur bzgl. des totalen Bereichs angegeben.

Überblick

7. Wiederdeklaration eines Namen – Statische Bindung – Umgebung
8. Totale und partielle Funktionen (Fortsetzung)
9. Kommentare
10. Die Standardbibliothek von SML
11. Beispiel: Potenzrechnung

Kommentare

- In SML sind Kommentare beliebige Texte, die mit den Zeichen `(*` anfangen und mit den Zeichen `*)` enden.
- SML lässt auch geschachtelte Kommentare zu.
- Bsp.:

achtert on fiji: /home/wiss/achtert

```
- (*
= fun Vorzeichen(x:int) = if x > 0 then 1
=                         else if x < 0 then ~1
=                              else (* x = 0 *) 0;
= *)
-
```

- BEACHTE: Klare und präzise Kommentare sind in jedem Programm unabdingbar! Es ist naiv anzunehmen, dass ein Programm selbsterklärend ist.

Überblick

7. Wiederdeklaration eines Namen – Statische Bindung – Umgebung
8. Totale und partielle Funktionen (Fortsetzung)
9. Kommentare
10. Die Standardbibliothek von SML
11. Beispiel: Potenzrechnung

Die Standardbibliothek von SML (1)

- Manche vordefinierte Funktionen von SML wie `Real.compare` sind in sog. *Modulen* (Programmen) programmiert, andere Funktionen wie + sind Teile des SML-Systems.
- Die SML-Bezeichnung für Modul ist „Struktur“ (structure).
- Eine Funktion `F`, die in einem Modul `M` definiert ist, wird außerhalb dieses Moduls als `M.F` bezeichnet und aufgerufen.

Die Standardbibliothek von SML (2)

- Die Standardbibliothek stellt eine Sammlung von Modulen für herkömmliche Typen wie reelle Zahlen dar.
- Die Module der Standardbibliothek werden vom SML-System automatisch geladen. Das Laden von anderen Modulen muss aber vom Programmierer explizit angefordert werden (s. Kapitel 12).
- Dokumentation zur SML-Standardbibliothek unter
  http://www.smlnj.org/doc/basis/

Überblick

7. Wiederdeklaration eines Namen – Statische Bindung – Umgebung
8. Totale und partielle Funktionen (Fortsetzung)
9. Kommentare
10. Die Standardbibliothek von SML
11. **Beispiel: Potenzrechnung**

Einfache Potenzrechnung

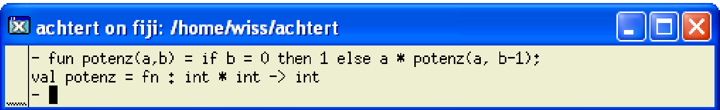
- Die folgende (nicht vordefinierte) Funktion soll in SML programmiert werden: $\text{potenz} : Z \times N \rightarrow Z$

$$(a,b) \mapsto a^b$$

- Die folgenden Gleichungen liefern die Grundlage für ein rekursives Programm: $a^b = 1$, falls $b = 0$

$$a^b = a \cdot a^{b-1} \text{, sonst}$$

- Implementierung in SML



Terminierungsbeweis für die einfache Potenzrechnung

Induktiver Beweis, dass für alle $(a,b) \in (Z \times N)$ der Aufruf `potenz(a,b)` terminiert:

Sei a eine beliebige ganze Zahl.

- *Induktionsanfang:* $b=0$
  Nach Funktionsdeklaration terminiert der Aufruf und liefert 1.
- *Induktionsschritt*:
  - *Induktionsannahme*: `potenz(a,b)` terminiert für eine natürliche Zahl $b \in N$
  - z.z: `potenz(a,b+1)` terminiert
    Beweis:
    - Nach Funktionsdeklaration liefert der Aufruf `potenz(a,b+1)` den Wert von `a*potenz(a,b)`.
    - Nach Induktionsannahme terminiert der Aufruf `potenz(a,b)`.
    - Folglich terminiert auch der Aufruf `potenz(a,b+1)`. **qed.**

Zeitbedarf der einfachen Potenzrechnung

- Der Zeitbedarf wird als die Anzahl der Multiplikationen zweier ganzer Zahlen geschätzt. Diese Schätzung stellt eine (übliche) Vergröberung dar, da die Multiplikation kleiner Zahlen weniger Zeit verlangt, als die Multiplikation großer Zahlen.
- Die Berechnung von `potenz(a, b+1)` benötigt eine Multiplikation mehr als die Berechnung von `potenz(a, b)`, die Berechnung von `potenz(a, 0)` benötigt keine Multiplikation. Also benötigt die Berechnung von `potenz(a,b)` insgesamt `b` Multiplikationen.
- Man sagt, dass der Zeitbedarf der Funktion `potenz` linear im zweiten Argument ist.

Effizientere Potenzrechnung

- Ist b gerade mit $b = 2k$, so gilt: $a^b = a^{2k} = (a^k)^2$
- Es ist also möglich, für gerade natürliche Zahlen b die b-Potenz einer ganzen Zahl a mit weniger als b Multiplikationen zu berechnen.
- Dies führt zu folgenden Funktionsdeklarationen:

achtert on fiji: /home/wiss/achtert

```
val potenz' = fn : int * int -> int
- fun potenz'(a, b) = if b = 0
=                     then 1
=                     else if gerade(b)
=                          then quadrat(potenz'(a, b div 2))
=                          else a * potenz'(a, b-1);
val potenz' = fn : int * int -> int
-
-
```

achtert on fiji: /home/wiss/achtert

```
- fun gerade(a) = (a mod 2 = 0);
val gerade = fn : int -> bool
-
- fun quadrat(a : int) = a * a;
val quadrat = fn : int -> int
-
-
```

Zeitbedarf der effizienteren Potenzrechnung (1)

- Der Zeitbedarf der Funktion `potenz'` wird ebenfalls als die Anzahl der Multiplikationen zweier ganzer Zahlen geschätzt. Dabei werden die Rechenzeiten für die Aufrufe des Prädikats `gerade` vernachlässigt.
- Somit ist die Rechenzeit abhängig von `b` und unabhängig von `a`. Sei also *rz*(`b`) die Rechenzeit eines Aufrufs `potenz'(a, b)` (für eine bel. ganze Zahl `a` und eine natürliche Zahl `b`). Es gilt:
  - $rz(2^b) = rz(2^{b-1})+1$
  - $rz(0) = 0$
  - Daraus folgt: $rz(2^b) = b$
- Auf die Potenzen von 2 ist also *rz* die Umkehrung der Funktion $b \rightarrow 2^b$, d.h. der Logarithmus zur Basis 2, genannt $\log_2$. Diese Beobachtung liefert keinen präzisen Wert für Zahlen, die keine Potenzen von 2 sind.

Zeitbedarf der effizienteren Potenzrechnung (2)

- Für große Zahlen ist der Zeitbedarf von `potenz'` viel geringer als der Zeitbedarf von `potenz`, z.B. bedarf `potenz'(a, 1000)` nur 14 Multiplikationen anstatt der 1000 Multiplikationen von `potenz(a, 1000)`. Für wachsende Werte von `b` vergrößert sich sehr schnell der Berechnungszeitabstand zwischen `potenz'` und `potenz`:

b	potenz(a, b)	potenz'(a,b)
1	1 Multiplikation	1 Multiplikation
10	10 Multiplikationen	5 Multiplikationen
100	100 Multiplikationen	9 Multiplikationen
1000	100 Multiplikationen	14 Multiplikationen
…	…	…

Bessere Implementierung der effizienteren Potenzrechnung

- Die folgende Implementierung der effizienteren Potenzrechnung ist auch möglich:

achtert on fiji: /home/wiss/achtert

```
- fun potenz''(a, b) = if b = 0
=                      then 1
=                      else if gerade(b)
=                           then potenz''(quadrat(a), b div 2)
=                           else a * potenz''(a, b-1);
val potenz'' = fn : int * int -> int
-
```

- Es ist leicht zu überprüfen, dass die Zeitbedarfsanalyse für die Funktion `potenz'` auch auf die Funktion `potenz''` zutrifft.

- Im Abschnitt 4.3 werden wir sehen, dass der `then`-Fall der Funktion `potenz''` *endrekursiv* ist, d.h. dass der rekursive Aufruf außer im `if-then-else`-Ausdruck in keinem weiteren zusammengesetzten Ausdruck vorkommt. Man beachte, dass der `else`-Fall der Funktion `potenz''` nicht endrekursiv ist. In diesem Abschnitt wird dann erläutert, warum Funktionen mit nur endrekursiven Aufrufen gegenüber Funktionen mit nicht-endrekursiven Aufrufen vorzuziehen sind.