Informatik 1

Programmieren mit Ausdrücken

1.1. Programmieren mit arithmetische Ausdrücke

Basics:

- Abfolge von Schritten = Algorithmus
- Sprache um Algorithmen zu programmien = Programmiersprache ⇒ bei uns BSL = Beginning Student Language
- Definitionsbereich – oberer DrRacket Bereich = Schreib & Editier Bereich
- Interaktionsbereich – unterer DrRacket Bereich = Auswertungsbereich

Einführung:

- Fragen bzw. Art von Programmen = Ausdrücke ⇒ Beispiel: (+ 2 2)
- Operationen (wie +/-/÷/*/sqr/expt/sin) = Funktionen ⇒ Bsp von Ausdrücke: +
- Operanden von Funktionen = Argumente ⇒ Bsp. von Ausdrücke: 2
- Argumente und Ergebnis sind Zahlen
- Zahl in Argument = Zahlenliteral
- "i" im Ergebnis = "inexact" → angenähertes Ergebnis
- Zahlen = atomare Ausdrücke
- Klammer Ausdrücke = zusammengesetzte Ausdrücke
- mehr als zwei Ausdrücke in einem Ausdruck = Schachtelung ⇒ $(+\ \overbrace{(/\ 4\ 2)}^{2}\ \overbrace{(*\ 2\ \overbrace{(-\ 2\ 1)}^{1})}^{2})$ ⇒ 4
  - ↳ rekursive Anwendung bei mehreren Unterausdrücken

1.2. Arithmetik mit nicht-numerischen Werten

- andere Arten von Ausdrücken (Text, Wahrheitswerte, Bilder, ...) = Datentypen
- Konstruktoren = Werte die Datentypen konstruieren ⇒ Bsp.: Zahlenliterale
- Funktionen = auf Datentypenwerte anwendbar → produzieren dadurch weitere Datentypen ⇒ Bsp.: + / * / -

Datentyp Text / String:

- Erkennungsmerkmal von Textkonstruktoren = Anführungszeichen "..."
- Textkonstruktor = Stringliteral
- String Aneinanderreihung = string-append
- alle Funktionen auf Strings = Arithmetik der Strings ⇒ nicht auswendig zu lernen!

Datentyp Wahrheitswerte / Boolsche Werte:

- Boolsche Werte Konstruktoren = #true und #false
- Funktionen von Boolschen Werten:
  - ↳ and ⇒ alle Konstruktoren müssen #true sein sonst kommt #false
  - ↳ or ⇒ mindestens ein Konstruktor muss #true sein sonst kommt #false
  - ↳ not ⇒ Gegenteil als Ergebnis

Datentyp Bilder

- Bildkonstruktor = Bildliteral
- mittels Copy & Paste einfügbar
- Bilder-Funktionen müssen mittels Teachpack freigeschaltet werden
- Bilder selbst konstruieren:
  - ↳ Kreis: (Circle Radius "Stärke Farbe" "Farbe")
  - ↳ Rechteck: (Rectangle Länge Breite "Stärke Farbe" "Farbe")
- Bilder zusammenfügen:
  - ↳ overlay: (overlay (oberstes Bild)
    ⋮
    (unterstes Bild))
  - ↳ place-image: (place-image (plazierendes Bild)
    x-Wert y-Wert ⇒ y-Koordinaten: von oben messend
    (Szene in der plaziert wird)) ⇒ x-Koordinaten: von links messend
  - ↳ empty-scene: (empty-scene x-Wert y-Wert)

Funktionsaufruf:

- Form: $(f\ e_1\ e_2\ \dots\ e_n)$
  - ↳ f = Name – muss kein Name einer existierender Funktion sein
  - ↳ e_1 - e_n = Ausdrücke

1.3 Auftreten & Umgang mit Fehlern

Zeitpunkte von Fehlern:

- Fehler **VOR** Programmausführung = statische Fehler
- Fehler **WÄHREND** Programmausführung = dynamische Fehler bzw. Laufzeitfehler

Arten von Fehlern:

- Syntaxfehler = BSL Programm passt nicht zur BSL Grammatik ⇒ Bsp. (+ 2 2 (
  - ↳ Syntax ist kontextfrei & lokal
  - ↳ **VOR** Programmausführung
- Typfehler = Funktion erwartet einen anderen Datentyp als gegeben
  - ↳ **WÄHREND** Programmausführung
- Aritätsfehler = Anzahl der gegebenen Argumente / Konstruktoren passt nicht zur Funktion
  - ↳ **WÄHREND** Programmausführung

1.4. Kommentare

- kein Einfluss auf die Programmbedeutung
- dienen besserer Lesbarkeit
- Semikolon (;) vor dem Kommentar

1.5. Bedeutung von BSL Ausdrücken

- Programmieren = Aufschreiben arithmetischer Ausdrücke
- Syntaktisch korrekt = mittels BSL Regeln konstruiert ⇒ nur dann hat Programm Bedeutung
- Bedeutung eines Programms:
  - ↳ v = Platzhalter für Wert-/Literal-Ausdrücke (Zahl, String, Bild, Boolsch Wert)
  - ↳ e = Platzhalter für beliebige Ausdrücke

Auswertungsvorschriften:

- Ausdruck = Literal → Bedeutung: Literal
- Ausdruck = $(f\ e_1 \dots e_n)$ → Bedeutung:
  - 1) $e_1 \dots e_n = v_1 \dots v_n$ ⇒ Anwendung von f auf $v_1 \dots v_n$ (f auf $v_1 \dots v_n$ definiert)
  - 2) $e_1 \dots e_n = v_1 \dots v_n$ ⇒ passende Fehlermeldung führt zum Abbruch (f nicht auf $v_1 \dots v_n$ definiert)
  - 3) Ausdrucksargumente kein v ⇒ Anwendung voriger Regeln bis geht
  - Funktionsname ↳ Ausdrücke
- Ausdruck = (and $e_1 \dots e_n$) → Bedeutung:
  - 1) $e_1 - e_{n-1}$ = #true & e_n = #false ⇒ #false
  - 2) $e_1 - e_n$ = #true ⇒ #true
  - 3) e_i = kein boolscher v ⇒ Fehlermeldungsabbruch
  - 4) e_i noch kein v ⇒ Anwendung voriger Regeln bis geht
- Ausdruck = (or $e_1 \dots e_n$) → Bedeutung: analog zu (and $e_1 \dots e_n$)

Reduktion von Programmen:

- $f(v_1 \dots v_n)$ [und Anwendung von f zu v führt] → v
- (and #false ...) → #false ⇒ "or" analog
- (and #true ...) → #true ⇒ "or" analog
- wenn e_1 nicht ausgewerteten Unterausdruck e_2 enthält: → zuerst $e_2 \to e_2'$ dann mittels e': $e_1 \to e_1'$
- Kongruenzregel: $e_1 \to e_2$, $e_2 \to e_3$ = $e_1 \to e_2 \to e_3$ ⇒ $e_1 \to^* e_3$
  - ↳ Konfluenz = wenn $e_1 \to e_4$ dann gilt $e_2 \to e_4$ und $e_3 \to e_4$
  - ↳ equational reasoning = Benutzung von "≡" um zu zeigen, dass Programme gleiche Bedeutung haben

Programmierer entwerfen Sprachen!

2.1. Funktionsdefinitionen

- **KEIN** Ausdruck
- arbeitet mit Variablen / Parametern
- ersparen von Redundanz
- DRY - Prinzip (Don't repeat yourself)
- allgemeine Form: (define (FunctionName Input1 Input2 ...) BodyExpression)
  → Bsp. (define (average x y) (/ (+ x y) 2))
- Bilder können ebenfalls zu Funktionsdefinitionen zusammengefasst werden

2.3. Bedeutung von Funktionsdefinitionen

- Auswertungsregeln aus 1.5 werden erweitert:
  ↳ Form $(f\ v_1 \dots v_n)$ [f = primitive Funktion & Anwendung auf $v_1 \dots v_n = v$] → v
  ↳ Form (define $(f\ x_1 \dots x_n)$ BodyExpression) dann: $(f\ x_1 \dots x_n)$ → NewBody aus $(f\ v_1 \dots v_n)$, $x_i \to v_i$ ersetzt
  ↳ Kongruenzregel gilt

2.4. Konditionale Ausdrücke (cond)

Motivation:

- Fallunterscheidung um bei verschiedenen Zuständen andere Aufrufe entstehen lassen

Bedeutung:

- Aufbau → (cond
  [ConditionExp1 ResultExp1]
  ...
  [ConditionExpN ResultExpN])
- ConditionExpression = Bedingung / Kondition
  ⇒ wenn #true kommt passendes Resultat raus
  ⇒ wenn #false kommt wird zweite Zeile angeschaut
  ⇒ wenn weder #true bzw. #false oder alle Bedingungen #false: Fehlermeldung
- ResultExpression = Resultat

Reduktionsregeln:

- Ausdruck: (cond [#false e] $[e_2\ e_3] \dots [e_{n-1}\ e_n]$) → (cond $[e_2\ e_3] \dots [e_{n-1}\ e_n]$)
- Ausdruck: (cond [#true e] $[e_2\ e_3] \dots [e_{n-1}\ e_n]$) → e
- Kongruenzregel trifft zu: (f v) → (cond $[e_0\ v_1] \dots [e_1\ v_2]$)

Syntaktischer Zucker:

- Synt. Zucker = kein hinzufügen von neue Sprachkonstrukten sondern Abkürzung bestehender Sprachkonstrukte
- Spezialfall 1: alle Zweige #false → Lösung: letzter Pfad mit "else"
  ↳ (cond $[e_0\ e_1][e_2\ e_3] \dots$ [else e_n]) → (cond $[e_0\ e_1][e_2\ e_3] \dots$ [#true e_n])
- Spezialfall 2: nur einen Zweig/Bedingung zu überprüfen → Lösung: "if" Konstrukt
  ↳ (if CondExpression ThenExpression ElseExpression) → (cond [CondExpression ThenExpression]
  ↳ "entweder das eine oder das andere" Situationen [else ElseExpression])

DRY Konzept:

- konditionale Ausdrücke vereinfachen wenn unnötig lang
- Beispiele:
  ↳ (if e #true #false) ⇒ e
  ↳ (if e #false #true) ⇒ (not e)
  ↳ (if e e #false) ⇒ e
  ↳ (if (not e-1) e-2 e-3) ⇒ (if e-1 e-3 e-2)
  ↳ (if e-1 #true e-2) ⇒ (or e-1 e-2)
  ↳ (if e-1 e-2 #false) ⇒ (and e-1 e-2)
  ↳ (if e-1 (f... e-2 ...) (f... e-3 ...)) ⇒ (f... (if e-1 e-2 e-3) ...)

2.5. Definition von Konstanten

- "place-image" Funktion platziert Zentrum des Bilder bei Koordinaten/Punkt
  ↳ (- yKoordinateHintergrund (/ (image-height Bild) 2))
- wichtig: DRY!
  ↳ (define HEIGHT ...)
  ↳ (define WIDTH ...)
  ↳ Definition von Konstanten ermöglicht leichtes neues Benutzen von Anforderungen
  ↳ Refactoring = Programmänderungen ohne Verhaltenänderung
  ↳ Verbesserung von Wartbarkeit, Lesbarkeit, Erweiterbarkeit
- globalen Scope = Definitionen können egal wo im Programm und können gelesen werden vom Programm
- Allgemein: (define CONSTANTNAME CONSTANTExpression)
- Konstanten werden sofort ausgewertet => Konstruktoren müssen bekannt sein
- Programm = Sequenz von Ausdrücken, Konstantendefinitionen
- Kontext = Menge von Funkions- & Konstantendefinitionen

Systematischer Programmentwurf

3.1. Funktionale Dekomposition

- Eine Funktion pro Aufgabe im Programm
- Aufgaben hierarisch angeordnet
  ↳ Top-Down Schema = Hauptfunktion am Anfang - oben
  ↳ Bottom-Up Schema = einfachste Funktionen am Anfang & ganz unten Hauptfunktion
  ↳ hierarchische Abstraktion = Aufrufstruktur so anordnen, dass einzelne Schichten alleine verständlich sind

3.3. systematischer Entwurf mit Entwurfsrezepten

- fördert Verständnis von Problembeschreibung
- Wissen über Daten die konsumiert werden und Daten die abhängig von den Eingabedaten produziert werden
- Beschreibung vom Programm
- Entwurfsrezept = Anleitung zur Schritt für Schritt Lösung von Entwurfsaufgaben

Testen:

- Programm(teil) mit Testdaten ausführen
- "check-expect" = Funktion die genauen Ergebnis Wert überprüft
  ↳ Ausdruck: (check-expect (ParameterInQuestion) ExpectedAnswer)
- "check-within" = Funktion die testet ob ein Ergebnis in einem Bereich liegt
- "check-range"

Information & Daten:

- Beziehung zwischen Informationen und Daten des Programmes
- Datendefinition als Kommentar
  ↳ Bsp: Distance is a number -> interprets .. -> (define WAYHOME 5) (Datenbeispiel)
- hilft Datendefinition zu verstehen
- können Datenbeispiele in Tests von Ausdrücken verwandeln

Entwurfsrezept zur Funktionsdefinition:

1. Datendefinition von den für die Funktion relevanten Informationen + Datenbeispiele
2. Signatur schreiben: Art(en) von Eingabe -> Art von Ausgabe
3. Aufgabenbeschreibung schreiben: Zusammenfassung des Zwecks der Funktion
4. Schreiben von Tests
5. Funktionskopf schreiben (Header/Stub): Funktionsdefinition passend zur Signatur mit Dummy-Werten im Body
6. Template: Dummy-Wert wird ersetzt durch "..." für jegliche Eingabedaten/Funktionen/Variablen
7. richtigen Funktionsbody implementieren mittels Template

3.4. Information Hiding

- Spezifikation = Name, Signatur, Aufgabenbeschreibung und Tests
- mittels Information Hiding / Geheimnisprinzip Programm besser lesbar, verstehbar und wartbar
  ↳ nur von Spezifikation abhängig nicht Implementierungsdetails

Überprüfbare Signaturen:

- Ausdruck: (: FunctionName Signatur / Art des Wertes)

Batchprogramme und interaktive Programme

4.1. Batchprogramme

- besteht aus Hauptfunktion
- Hauptfunktion besteht aus Hilfsfunktionen
- Kommandozeilenparameter wichtig
- Stärke: Leicht & vielfältig kombinieren mit anderen Batchprogrammen
- Nicht für Interaktion mit Benutzer geeignet

4.2. interaktive Programme

- mehrere Hauptfunktionen → können dazu noch Hilfsfunktionen verwenden
- Informationsausdruck für Computer bezüglich Art von Eingaben und Ausgaben Funktionen

Universe teachpack:

- unterstützt interaktive Programme
  ↳ Programme die auf Zeitsignale, Mausklicks, Tastatureingaben oder Netzwerkverkehr reagieren
- Produzieren grafische Ausgaben
- "WorldState" = Konzept welches den bestimmten Zustand der interaktiven Anwendung beschreibt
  ↳ Zustand ändert sich bei bestimmten Ereignissen

Datendefinition durch Alternative: Summentypen

5.1. Aufzählungstypen

- Datentyp enthält endliche aufzählbare Menge an Werten = Aufzählungstypen bzw. Enumerationtypen
- Programm enthält typischerweise Fallunterscheidung mit aller Alternativen des Aufzählungstyps
  ↳ bedeutet meistens mit "cond" bzw. "if"

5.2. Intervalltypen

- Datentyp enthält unendlich große Menge an Werten = Intervalltypen
- Verwendung von Intervallen mittels Grenzen
  ↳ entweder obere & untere Grenze, nur eine Grenze & zur anderen Seite offen
- ebenfalls eine Fallunterscheidung
  ↳ Nutzung von "cond" bzw. "if"

5.3. Summentypen

- Verallgemeinern Intervall- & Aufzählungstypen
- existierende Datentypen & individuelle Werte beliebig kombinieren
- auch Vereinigungstypen genannt
- vereint verschiedene Alternativen an Wert-Typen
- ebenfalls eine Fallunterscheidung
  ↳ Nutzung von "cond" bzw. "if"

Entwurf mit Summentypen:

1. Werte in unterschiedliche Klassen unterteilt ⇒ Datendefinition definieren
2. Signatur bleibt gleich
3. pro Datentyp Alternative mind. ein Beispieltest
   ↳ bei Intervalltypen beide Grenzen testen
4. Template: meistens TemplateBody mit "cond" Ausdruck
5. "..." im Template nach und nach mit passendem Code ersetzen

5.4. Unterscheidbarkeit der Alternativen

- anstelle absoluten Werten ein kontinuierliches Spektrum
  ↳ um Alternativen zu differenzieren. zusätzliche Information "tag" wozu der Wert gehört

Datendefinition durch Zerlegung: Produkttypen

Basics

- mehrere Werte zusammenstellen dass sie wie ein einzelner Wert behandelt werden
- "structs" / Strukturen = mehrere Daten zu einem Datum & bei Bedarf wieder zerlegen möglich
  ↳ oft auch Produkte genannt

6.1. posn Struktur

- Aussprache: position
- ist eine Struktur, die einen Wert repräsentiert
  ↳ besteht aus zwei Werten: x-koordinate & y-koordinate einer Position
- "make-posn" produziert Instanz der Struktur
  ↳ Signatur: (number number → posn)
- um zu extrahieren:
  ↳ posn-x & posn-y Funktionen

6.2. Strukturdefinitionen

- Strukturen (Beispiel posn) werden nicht fest im Programmiersprache implementiert
- struct zu der Mechanismus um strukturen zu definieren
  ↳ posn: (define-struct posn (x y))
- Allgemeine Form: (define-struct StructurName (FieldName ... FieldName))
- define-struct = gibt an, dass eine Struktur definiert wird
- ermöglicht Definition eines ganzen Satzes an Funktionen:
  ↳ Konstruktor = Funktion mit n-Parameter wie n vorhandene Fields ⇒ liefert Instanz der Struktur
    → Form: (make-StructureName ...)
  ↳ Selektor = pro Field je ein Selektor ⇒ liefert Field-Wert aus Instanz
    → Form: (StructurName-FieldName ...)
  ↳ Strukturprädikat = boolsche Funktion zur Feststellung ob ein Wert Instanz einer Struktur ist
    → Form: (StructureName ? ...)

6.3. Verschachtelte Strukturen

- verschachtelt viele Strukturen um mehr Übersicht zu schaffen
- Daten werden hierarchisch in Form eines Baums repräsentiert

6.4. Datendefinitionen für Strukturen

- Zweck ist zu beschreiben welche Art von Daten jedes Feld erhält
- nicht immer eine Struktur mit genau einer zugehörigen Datendefinition
- könnten rein theoretisch immer dieselbe Struktur verwenden und alle Produkttypen damit kodieren
- keine Unterscheidung von Daten möglich

6.7. Erweiterung des Entwurfsrezepts

1. Information vorhanden, die zusammengehören oder Ganzes beschreiben => Strukturen nötig
   ↳ Beispiele für Instanzen der Struktur
2. wie zuvor - Signatur, ...
3. Tests entwerfen mittels obiger Beispiele
4. Template sollte Selektorausdrücke verwenden wenn nötig
5. mittels Selektorausdrücke Funktion implementieren
6. Testen des Funktionsheaders

Datentypen durch Alternativen und Zerlegung: Algebraische Datentypen

=> Kombination von Summen- und Produkttypen = algebraische Datentypen (ADT)

7.2. Programmentwurf mit ADTs

1. Informationen als Eingabe bzw. Ausgabe am besten als ADTs repräsentiert => ADT definieren
   ↳ ADT sinnvoll wenn: unterschiedliche Arten von Informationen aber gemeinsames Konzept
   ↳ mittels ADT Daten hierarchisch organisieren
   ↳ Produkttyp mit vielen Feldern bzw. Summentyp sehr viele Alternativen => deutet auf ADT hin
2. wie zuvor - Signatur, Aufgabenstellung, ...
   ↳ Verwendung der definierten ADTs
3. Tests
   ↳ bei zu großer Verschachtelung nicht mehr möglich jede Alternative zu testen
4. Funktionstemplate mit zwei Dimensionen
   ↳ Summentyp Fälle mittels cond unterscheiden
   ↳ Produkttypen Alternativen: Hilfsfunktionen definieren (bis zum Template-Schritt)
5. von Template zu lauffähiges Programm
6. Testen nochmals
7. Refactoring, DRY-Prinzip überprüfen

7.3. Refactoring von ADTs

- gibt mehrere Arten Daten zu modellieren

Daten beliebiger Größen

9.3. Listen

- Alternative des Summentyps
- Entwurf genauso wie aller anderen Bäume
- List-of-Numbers is either:
  - (make-1st Number List-of-Numbers)
  - #false
  - -> interprets the head and rest of a list, or the empty list

Listen aus der Konserve:

- Vordefinierte Funktionen
  - ↳ cons = Konstruktorfunktion ⇒ entspricht dem Strukturkonstruktor make-name
  - ↳ cons? = Prädikat von cons ⇒ überprüft ob es sich um eine Liste handelt
  - ↳ empty = leere Liste
  - ↳ empty? = Prädikat von empty ⇒ überprüft ob es sich um eine leere Liste handelt
  - ↳ first = der erste Teil der Liste
  - ↳ rest = der restliche Teil der Liste ohne first
- cons:
  - ↳ baut eine Liste
  - ↳ überprüft, dass zweites Argument auch Liste ist
- Syntaktischer Zucker: list
  - ↳ (cons x (cons y (cons z empty))) ⇒ (list x y z)
  - ↳ Allgemein: (list exp-1 ... exp-n)
  - ↳ Abkürzung der Schreibweise (cons exp-1 (cons ... (cons-n empty)))
  - ↳ konsumiert Listen
- syntaktischer Zucker: make-list
  - ↳ konsumiert natürliche Zahl und einen beliebigen Wert
  - ↳ produziert daraus Liste
  - ↳ Allgemein: (make-list Number V)

9.4. mehrere rekursive Datentypen gleichzeitig

- Konkatenation zweier Listen = append
  - ↳ bedeutet eine Verkettung bzw. Verknüpfung zweier Listen

Pattern Matching

Basics

- Funktionen mit Summen- oder algebraischen Datentyp wird erst unterschieden was vorliegt und dann auf Komponenten zugegriffen
- Pattern Matching = solche Funktionen mit reduziertem Aufwand definieren
  - ↳ definiert Bedingung analog zu Bedingungen der Funktion
  - ↳ definiert Namen welche anstelle Projektionen verwendet werden können
  - ↳ alle Arten von Summentypen anwendbar

Pattern Matching Allgemein

- match = Pattern matching Konstruktor
- erstes matching Pattern gewinnt
- Allgemein: (match v [p-1 e-1] ... [p-n e-n])
- Aufgabe: möglichst kleines i finden, sodass p-i auf v matcht
- es wird nur zum ersten passenden Unterausdruck ausgewertet

Quote und Unquote

11.1. Quote

- kompakte Notation für große & verschachtelte Listen
- Beispiel: (quote (1 2 3)) ≡ (cons 1 (cons 2 (cons 3 empty))) ≡ (list 1 2 3)
- Abkürzung von quote = gerades Hochkomma '

- Bedeutung über rekursive, syntaktische Transformation:
  - ↳ '(e-1 ... e-n) ≡ (list 'e-1 ... 'e-n), 'L ≡ (
    - ↳ 'e-1 : Anwendung von rekursiver Transformation

11.2. Symbole

- dienen zur Repräsentation symbolischer Daten
- Symbole ähnlich zu strings
  - ↳ "strings"
  - ↳ 'symbole'
  - ↳ nicht zur Text Repräsentation gedacht
  - ↳ nur eine wichtige Operation: symbol=? ⇒ Vergleich von Symbolen

11.3. Quasiquote und Unquote

- quasiquote Konstrukt vom Verhalten wie quote
- Abkürzung von quasiquote = schräges Hochkomma `
- ermöglicht innerhalb gequoteten Bereichs zurückspringen in Programmiersprache
- unquote = Möglichkeit des Zurückspringens
- Abkürzung von unquote = Kommazeichen ,
- wenn quasiquote und unquote aufeinander treffen = neutralisation beider
  - ↳ `,e = e = 'e

11.4 S-Expressions

- universelles Datenformat ⇒ ermöglicht beliebig strukturierte Daten zu repräsentieren
- S-Expression sind:
  - ↳ Number
  - ↳ String
  - ↳ Symbol
  - ↳ Boolean
  - ↳ Image
  - ↳ empty
  - ↳ a (list-of S-Expression)
- Beispiel: (list 1 (list 'two 'three) "four"), "Hi"

Sprachunterstützung für Datendefinition und Signaturen

12.1. Ungetypte Sprachen

- jede Operation ist auf alle Daten anwendbar
  - ↳ unabhängig ob es Sinn ergibt oder nicht
- Assembler-Sprachen als Beispiel
- Verantwortung des Programmierers Daten-Eigenschaften sicherzustellen

12.2. Dynamisch getypte Sprachen

- jedem Wert wird ein Typ zugeordnet ⇒ Markierung zur Erkennung der Art des Werts
- Laufzeitsystem: repräsentiert Werte passed zur ständigen Abfrage des Typs
- BSL ist ein Beispiel dafür
  - ↳ Boolean, Number, String, Symbol ⇒ fest eingebaute Typen
  - ↳ define-struct ermöglicht Definition neuer Typen

12.3. Dynamisch überprüfte Signaturen und Contracts

- Signaturen & Datendefinitionen als Programme definieren
  ↳ laufen & werden überprüft während Programm läuft
- Contracts = dynamische Prüfungen von Signaturen
- Fehler können früher gefunden werden
- klar benennbaren Schuldigen bzw. Fehlerquelle

12.4. Statisch getypte Sprachen

- jedem Programmteil **VOR** Ausführung Typ zugeordnet
- Kompositionalität = Typ eines zusammengesetztes Programmteil nur abhängig der Typen der Komponenten
  ↳ (+ e-1 e-2) ist Typ Number **NUR** wenn e-1 & e-2 Number sind

Sprachunterstützung für Algebraische Datentypen (ADT)

13.1. ADTs mit Listen und S-Expressions

- algebraische Datentypen mit verschachtelten Listen ⇒ S-Expressions
- dadurch möglich Interpreter für Ausdrücke zu definieren

13.2. ADTs mit Strukturdefinitionen

- mittels Strukturen Interface implementieren

13.3. ADTs mit define-type

- bietet direkte Unterstützung für Summentypen
  ↳ ermöglicht sofortige Definition
- zu jedem Feld einer Alternative je eine Prädikatsfunktion (immer)
  ↳ Bsp: (define-type Expression
  (literal (value number?))
  (addition (left Expression?) (right Expression?)))
- Erweiterung von match: type-case
  ↳ Typ wird direkt mit angegeben ⇒ Fallunterscheidung / Falldeckung wird bereits **VOR** Programmausführung geprüft
  ↳ erlaubt nur sehr eingeschränktes Pattern Matching
  ↳ Allgemein: ((variant (nam-1 ... name-n) body expression))

13.6. ADTs mit Dictionaries

- Dictionary = repräsentieren partielle Funktionen mit keys (endliche Definitionsmengen) mit je max. einem Wert
  ↳ {name: 'Höhn', vorname: 'Gesa'}
  (name = Key, 'Höhn' = Wert, vorname = Key, 'Gesa' = Wert)
- wie Listen nur Elemente werden durch Key adressiert (nicht Position)
- können auch als universelle Datenstruktur verwendet werden (wie S-Expressions) wegen Verschachtelung