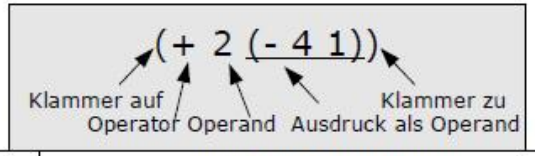
Grundlagen der Informatik 1 – Zwischenklausur 12.12.2009 – Scheme

Scheme Sprachübersicht (für Übungen und zur Klausurvorbereitung)



Sprache	*Definitionen*	*Ausdrücke (Expressions)*	*Primitive Operationen*
Beginner	> (define (NAME NAME NAME ...) EXPRESSION) *definiert eine Prozedur mit Parametern* > (define NAME EXPRESSION) *bindet einen Namen an einen Ausdruck („Variablendefinition“)* > (define-struct NAME (NAME ...)) *erstellt eine Struktur mit folgenden Prozeduren:* make-structname *(Konstruktor)*, structname? *(Abfrage) und* structname-attributname *(Selektoren)*	> (NAME EXPRESSION EXPRESSION ...) *führt eine Prozedur mit Parametern aus* > (PRIM-OP EXPRESSION ...) *wendet einen primären Operator (+, -, ...) an* > (if CONDITION EXPRESSION EXPRESSION) *falls Bedingung wahr werte ersten Ausdruck aus, ansonsten den zweiten* > (cond [CONDITION EXPRESSION] ... (else EXPRESSION)) *Abfrage mehrerer Fälle, falls Bedingung stimmt werte den Ausdruck aus* > (and EXPRESSION EXPRESSION EXPRESSION ...) *gibt wahr zurück wenn alle Ausdrücke wahr sind* > (or EXPRESSION EXPRESSION EXPRESSION ...) *gibt wahr zurück falls mindestens ein Ausdruck wahr ist* > NUMBER *Zahl*, empty *leere Liste*, true, false *Wahrheitswerte*	> +, *, /, -, = *bekannte Rechenzeichen (preorder beachten, siehe Beispiel oben)* > symbol=? : (symbol symbol -> boolean) *Ist das ein Symbol?* > number? : (any -> boolean) *Ist das eine Zahl?* > list? : (any -> boolean) *Ist das eine Liste?* > empty? : (any -> boolean) *Ist das eine leere Liste?* > first : ((cons y (listof x)) -> y) *Erstes Element der Liste* > rest : ((cons y (listof x)) -> (listof x)) *Rest der Liste* > cons : (x (listof x) -> (listof x)) *setze zwei Objekte (Listen) zusammen*
Beginner with List Abbrev.		> 'QUOTED *Symbol* > ('QUOTED ...) *Liste von Symbolen*	> list : (any ... (listof any) -> (listof any)) *erzeuge eine Liste*
Intermediate		> (local (DEFINITION ...) EXPRESSION) *lokale Umgebung*	
Intermediate w. Lambda		> (lambda (NAME NAME ...) EXPRESSION) *„namenlose“ Prozedur*	
Advanced		> (begin EXPRESSION EXPRESSION ...) *führt Ausdrücke in Sequenz (d.h. Nacheinander) aus und gibt den Wert des letzten zurück* > (set! NAME EXPRESSION) *ändert die Zuweisung einer Variable die mit* define *angelegt wurde*	> build-vector : (nat (nat -> x) -> (vectorof x)) *erzeuge Vektor mit Hilfe einer Prozedur* > make-vector : (number x -> (vectorof x)) *Konstruktor* > vector : (x ... -> (vector x ...)) *Konstruktor* > vector-length : ((vector x) -> nat) *Länge des Vektors* > vector-ref : ((vector x) nat -> x) *Selektor* > vector? : (any -> boolean) *Ist das ein Vektor?*

Mehr Informationen in der Hilfe von Dr.Scheme, auf http://www.plt-scheme.org/docs.html und auf http://www.htdp.org/.

Grundlagen der Informatik 1 – Zwischenklausur 12.12.2009 – Scheme

Allgemeines	Strukturdefinitionen	Operationen für Listen	Symbol und String	Datentypen vector und map
(**define** (<function-name> <parameter> …) <expr>) *Namenszuweisung* (**check-expect** <actual> <expected>) (**check-within** <actual> <expected> <delta>) (**check-error** <test> <message>) (**error** <'function> <"message">) (**if** <test> <then> <else>) (**cond** [<question> <answer>] [<question> <answer>] …) (**and** <expr> <expr> …) *stoppt, sobald erstes false* (**or** <expr> <expr> …) *stoppt, sobald erstes true* (**not** <expr>) (**Boolean=?** <bool> <bool>) (**equal?** <expr> <expr>) (**time** <Funktionsaufruf>) *misst benötigte Zeit* (**begin** <expr> <expr> …) *führt mehrere Befehle nacheinander aus* (**set!** <variable> <expr>) *Zuweisung*	(**define-struct** <name> (<attribute1> <attribute2> …)) (define <expr> (**make-<name>** <expr> <expr> …)) *Konstruktor* (**<name>?** <expr>) *Prädikat* (**<name>-<field>** <expr>) *Selektor*	(**cons** <expr> <empty/cons>) (**list** <expr> <expr> …) *wertet expr aus* **'(**<expr> <expr> …) *macht expr zu symbol* **empty** *= leere Liste* (**append** <list> <list>) (**length** <list>) (**first** <list>) (**second** <list>) (**third** <list>) (**rest** <list>) (**list?** <list>) (**cons?** <list>) (**empty?** <list>) (**build-list** <length> <procedure>)	(**symbol?** <expr>) (**symbol=?** <symbol> <symbol>) **'symbol** (**string=?** <string> <string>) (**string-append** <string> <string>) (**number->string** <number>) (**symbol->string** <symbol>) **"string"**	(**vector** <value> <value> …) (**vector-ref** <vector> <index>) *extrahiert Wert (beginnt bei 0 zu zählen)* (**vector-length** <vector>) (**vector?** <expr>) (**build-vector** <length> <function>) (**map-create** <symbol-list> <value-list>) (**map-extend** <symbol> <value> <map>) (**map-remove** <symbol> <map>) (**map-remove-all** <symbol-list> <map>) (**map-lookup** <symbol> <map>)

Operationen für Zahlen	Blockstruktur, fold und map	Ein-/Ausgabe nach stdout	Teachpack draw.ss	Teachpack image.ss
(**sqr** <expr>) *quadriert* (**sqrt** <expr>) *Wurzel* (**zero**? <number>) (**even?** <number>) (**odd?** <number>) (**gcd** <number> <number>) *größter gemeinsamer Teiler* (**min** <number> <number> …) (**max** <number> <number> …) (**floor** <number>) *rundet ab* (**abs** <number>) *bildet Betrag* (**expt** <x> <y>) *berechnet (x^y)* (**remainder** <number> <number>) *Rest einer Ganzzahldivision*	(define (<function-name> <parameter> …) (**local** ((define (<name> <param> …) <expr>) <definition2> …) <local-body>) <expr>) (define (<function-name> <parameter> …) (**lambda** (<param> …)<lambda-body>)) (**map** <function> <list> …) *wendet Funktion auf eine oder mehrere Listen (synchron) an* (**foldl** <function> <basic-value> <list>) *faltet von links nach rechts* (**foldr** <function> <basic-value> <list>) *faltet von rechts nach links*	(**print** <expr>) *als Wert* (**display** <expr>) *ohne Hochkomma etc.* (**write** <expr>) *traditionell* (**pretty-print** <expr>) *Zeilenumbruch und Einrückung* (**printf** <expr>) *Formatierung wie erstes Element* (**newline**) *Zeilenumbruch* (**read**) *Nutzereingabe lesen*	(**make-posn** <x> <y>) (**draw-solid-disk** (make-posn <x> <y>) <radius> <'colour>) (**clear- solid-disk** (make-posn <x> <y>) <radius> <'colour>) (**draw-solid-line** <posn> <posn>) (**draw-solid-string** <posn> <string>) (**sleep-for-a-while** <delay>) (**start** <width> <height>) *erstellt Leinwand* (**stop**)	(**make-color** <red> <green> <blue>) (**image-width** <image>) (**image-height** <image>) (**image->color-list** <image>) (**color-list->image** <image> <width> <height> <pinhole-x> <pinhole-y>)

Grundlagen der Informatik 1 – Zwischenklausur 12.12.2009 – Scheme

Allgemeines	Rekursion, Bäume, Sortierung	Abstraktion, Akkumulatoren
- **Primitive** Ausdrücke (Zahlen, boolesche Werte, eingebaute Prozeduren), **selbstauswertend** - **Kombinationsmittel** (Präfixdarstellung) - **Abstraktionsmittel** (z.B. define = Sonderform, Kombinationen mit Namen assoziieren und als Primitive behandeln) - **Umgebung** = Name-Objekt-Paare - **Programmkopf**: contract, purpose, example, 2 tests - **Applikative** Auswertung = von innen; **Normale** Auswertung = von außen - **Konfluenz** = Ergebnis unabhängig von Auswertungsreihenfolge (falls terminierend) - **Top-down**-design <> **bottom-up**-design <> **Wunschlisten**-Ansatz - **Abgeschlossenheit**seigenschaft: Ergebnis der Anwendung des Operators ist in Menge, auf die er angewendet wurde - **Verträge**: number, symbol, boolean, <struct-name>, (listof X), (vectorof X), (X -> Y), <name> - <name> in Verträgen = **parametrische Datendefinition** (Beschreibung: <name> is either symbol or …)	- **Strukturelle** Rekursion (folgt Datenstruktur) <> **Generative** Rekursion (Divide & Conquer + Rekombination) - Strukturelle = Spezialfall der generativen Rekursion - **Rekursionsanker** = trivialer, nichtrekursiver Fall - **Bäume** (make-tree-node <name> <attribute> <(listof children)>) → **wechselseitig rekursiv** da jedes Kind neuer Baumknoten ist - Generative Rekursion kann zu **Endlosschleifen** führen → Terminierungsargumente für alle möglichen Eingaben einfügen - **Insertion Sort**: sortierte Liste--- unsortierte Liste -> einfügen - **Quicksort**: Drehpunkt (Pivot-Element), teilt in Liste <= Pivot und Liste >= Pivot - **Mergesort**: teilt Liste in der Mitte und sortiert bei Rekombination	- In Hilfsprozeduren auslagern = **prozedurale** (Black-Box) Abstraktion - Zusammenfassen ähnlicher Prozeduren (z.B. contains?) = **funktionale** Abstraktion - **Abstraktionsbarriere** (z.B. map) unterdrückt Detaillevel - Abstraktion → eindeutiger **Kontrollpunkt**, separation of concerns, Wartbarkeit - Abstraktion anhand von Prozessphasen: aufzählen, filtern, map, akkumulieren → **modularer Programmentwurf** - Richtung von fold egal, wenn Funktion **kontextfrei** - **Akkumulatoren**: generativ rekursiv oft nicht ohne lösbar (z.B. Graphen mit Zyklen), strukturell rekursiv mit oft zeiteffizienter, sinnvoll bei Rekursion in der Rekursion - Lokale Funktion mit Startwert des Akkumulators aufrufen, **Akkumulator-Invariante** - Im local-Block vor define **Kommentar** zur Funktion des Akkumulators einfügen (was wird akkumuliert?)

Blockstruktur (local), Syntax, Semantik, Scope	Interpreter	Graphen, Vectoren. Maps
- Prozeduren **höherer Ordnung** = konsumieren oder produzieren Funktionen - **Lambda** = ad-hoc Funktionsdefinition, (z.B. polymorphe Funktionen mit **Gedächtnis**) - Kein lambda bei Rekursion, weil namenlos - Blockstruktur = lokale Namen - Auswertungsregel: local-Definitionen mit neuen Namen auf Top-Level ziehen - **Syntax** = Vokabular & Grammatik - **Semantik** = Bedeutung eines Programmes (Substitutionsmodell) - **Scope** = Bereich, in dem sich Auftreten eines Namens auf eine Namensbindung bezieht - Namen sind **frei oder gebunden** (relativ zum Ausdruck) - **Top-Level**-Definitionen = **globaler Scope**, Local-Definitionen = local-body als scope	- **Meta-Interpreter** (in der Sprache geschrieben, die er interpretiert) - Interpreter kann der **Definition der Semantik** dienen - **Basissprache <> interpretierte Sprache** - Interpreter: **Syntaxdefinition** (s-expressions), **Parser** (abstract syntax tree), **Auswertungsprozedur, Substitution, Umgebung** (=map)**, Startprozedur**	- Graphen als Liste von **Knoten und Kanten** '((A (B C)) (B (F)) (C ())) - **Backtracking**: Möglichkeit durchlaufen, bis Erfolg oder Sackgasse; wenn Sackgasse, zurück zu letztem Knoten mit Auswahlmöglichkeit; neue Alternative; wenn keine übrig, Misserfolg - Graphen **traversieren**, Weg finden → Startknoten = Zielknoten? - Terminiert nicht, wenn **Zyklus** → Akkumulator - **Vektoren** ermöglichen Datenzugriff in konstanter Zeit. Speichern Werte mit Indices (**Arrays**) - Abstrakter Datentyp **map** assoziiert Symbole mit Werten (vgl. Vektor) - Graphenknoten können als Vector-Indices dargestellt werden

Grundlagen der Informatik 1 – Zwischenklausur 12.12.2009 – Scheme

Komplexität von Algorithmen	Streams, Zuweisungen, Effekte
- Zeit, Speicher, Netzwerkbandbreite - unabhängig von PC und Eingabe (**abstraktes** Zeitmaß) - Kostenfunktion: Definitionsbereich = Eingabegröße n; Wertebereich = Anzahl Rechenschritte T(n) - **Anzahl der Rekursionsschritte → abstrakte Laufzeit**, Rekursion auf Rekursion = n^2 - Best-case & worst-case, durchschnittlicher Fall, Faktor ½ kann in Konstante integriert und vernachlässigt werden → **Größenordnung (Komplexitätsklasse)** O(1), O(log(n)), O(n), O(n*log(n)), O(n^2), O(2^n) - **Sortieralgorithmen:** Merge-Sort = n*log(n); Insertion-Sort n^2; Quicksort n*log(n) - **O-Notation:** Groß-O = Klasse von Funktionen auf natürlichen Zahlen, obere Schranke für Wachstum ab genügend großem n → nur für große Eingaben sinnvoll - Divide & Conquer → **Rekurrenzgleichung** - **Substitutionsmethode**, **Rekursionsbaummethode** (Ebenen addieren, innerhalb Ebene summieren), **Master-Theorem**	- **Streams** =Unendliche Listen (nur bei **normaler Auswertungsreihenfolge** möglich); (Abbruchbedingung z.B. wenn Ergebnisse sich näher als bestimmte Schranke annähern) - **Set! → Effekt**: alle Werte, die zur gesetzten Variable referenzieren evaluieren zum gesetzten Wert - Von Set! verwendete Variablen heißen **Zustandsvariablen** - Vorher auf Top-Level mit define **Startwert** festlegen - Funktion, die keine Parameter erhält und nur Set! Aufruft wird mit (<function-name!>) aufgerufen (**nicht mehr funktional/effektfrei, sondern destruktiv**), keine Konfluenz - Jetzt ist Ändern der Reaktion auf eine Eingabe möglich (z.B. neuer Eintrag in Adressbuch) - Funktionen mit Zuweisungen haben **!** im Namen - Objekte haben **Identität** zusätzlich zu momentanem Wert - **Sequenzieren** von Ausdrücken mit begin wertet in gegebener Reihenfolge aus - **Keine referentielle Transparen**z, da Wert der Variable vom **Zeitpunkt** abhängig - E/A- bzw. **I/O-Effekt**, Kommunikation mit Nutzer - **Purpose-statement** an Stelle wo Zustandsvariablen definiert und initialisiert werden - Funktionen mit Effekt: contract, purpose, **effect,** example (Zeitpunkt in example und test einbeziehen) - Zustandsvariable = **Kommunikationskanal** zwischen Prozeduren, verkomplizieren **Schnittstellen** → niemals global und möglichst selten verwenden - Jedes Programm kann auch ohne Zuweisungen geschrieben werden

Rekurrenzgleichung

$$T(n) = \begin{cases} \Theta(1) & n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & sonst \end{cases}$$

Das Master-Theorem

Betrachte

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n < c \\ aT(n/b) + f(n) & \text{falls } n > 1 \end{cases}$$

wobei a >= 1 und b >= 1

- Wenn $f(n) = O(n^{\log_b a - \varepsilon})$ für eine Konstante ε > 0, dann $T(n) = \Theta(n^{\log_b a})$
- Wenn $f(n) = \Theta(n^{\log_b a})$, dann $T(n) = \Theta(n^{\log_b a} \log_b n)$
- Wenn $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für eine Konstante ε > 0 und wenn $a\,f(n/b) <= c\,f(n)$ für eine Konstante $c < 1$ und alle hinreichend großen n, dann $T(n) = \Theta(f(n))$

Die Substitutionsmethode

Rekurrenz: $T(n) = 2T(\lfloor n/2 \rfloor) + n$

Geschätzte Lösung: $T(n) = O(n \log n)$

Zu zeigen ist, dass für eine geeignete Konstante c > 0 $T(n) \le cn \lg n$

Anfangsannahme: die Schranke gilt für $\lfloor n/2 \rfloor$ also: $T(\lfloor n/2 \rfloor) \le c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\le 2\left(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)\right) + n \\ &\le cn\log(n/2) + n \\ &= cn\log n - cn\log 2 + n \\ &= cn\log n - cn + n \\ &\le cn\log n \qquad c \ge 1 \end{aligned}$$

$n = 1 \Rightarrow 1 = T(1) \le c1\log 1 = 0$???

Lösung: die Abschätzung ist für $n > n_0$ zu zeigen

$n = 2 \Rightarrow T(2) = 2T(1) + 2 = 4$

$T(2) \le c2\lg 2 = c2 \Rightarrow c = 2$