1. Оценяване на комбинация – Първо се оценяват подизразите на комбинацията, а после се прилага процедурата, която е оценка на най-левия подизраз (оператора) към операндите, т.е оценките на останалит

е подизрази. Това правило е рекурсивно, т.к. всеки от подизразите може да бъде някаква комбинация. При този рекурсивен процес се стига до оценяване на числа, низове, вградени оператори и променливи (атоми).

2. Специални форми – вградени оператори, оценяването на обръщенията към които не се извършва по общото правилото за оценяване на комбинациите. (define, begin)

3. Модел на заместването при оценяването на обръщения към съставни (дефинирани) процедури – прилагането на дадена съставна процедура към получените аргументи се извършва, като съгласно правилото от 1, се оценят последователно изразите от тялото на процедурата, в които формалните параметри са заменени със съответните фактически. Оценката на последния израз от тялото става оценка на обръщение към съставна процедура.

4. Апликативен и нормален подход при оценяването на обръщения към съставни процедури – При апликативния подход отначало се оценяват операндите и след това към получените оценки се прилага резултатът от оценяването на оператора. При нормалния подход се замества името на процедурата с тялото ѝ, докато се получат означения на примитивни процедури и след това се прилагат техните правила за оценка .

5. Модел на средите за оценяване на изрази – Нека имаме процедура (define (<име> фо$П_1$ ... фо$П_n$) <тяло>), оценка на израза (<име> фо$П_1$ ... фо$П_n$) в средата Е се осъществява по същия начин, по който се оценява комбинация, чийто оператор е примитивна процедура. Оценката на <име> е съответния процедурен обект. Генерира се нова среда E_1, разширение на Е. В нея фо$П_1$ се свързва с оценката на фо$П_i$ и се оценява <тяло>. Оценката на последния израз от <тяло> е оценка на обръщението.

6. Абстрахирането чрез процедури и чрез данни – Най-същественият елемент на абстрахирането е разделянето на задачата на подзадачи, които могат да бъдат решавани поотделно. Основни преимущества на този подход: голям програмен проект може да бъде разделен на няколко независими една от друга части. По-лесно се извършва поправка и проверка на програмите. При абстрахирането чрез данни програмата се конструира така че да работи с „абстрактни” данни чиято структура може да не е (напълно) уточнена. След това представянето на данните се конкретизира с помощта на множество процедури наречени конструктори и селектори които реализират абстрактните данните по конкретен от програмиста начин.

7. Рекурсивни и итеративни изчислителни процеси – При рекурсивните процеси се поражда верига от обръщения към дефинираната функция с все по-прости в определен конкретен смисъл аргументи, докато се стигне до обръщението с т.нар. базов вариант на аргументите, след което започва последователно пресмятане на генерирани вече обръщения. При итеративните процеси във всеки момент състоянието на изчислителния процес се описва от няколко променливи (променливи на състоянието) и правило, с чиято помощ се извършва преходът от дадено състояние към следващото. При линейната рекурсия дефиницията включва (поражда) само едно рекурсивно обръщение към същата процедура с опростени аргументи. Опашковата (tail recursion) рекурсия е линейна рекурсия, при която общата задача се трансформира до нова по-проста, като при това решението на общата задача съвпада с решението на по-простата, а не се получава от него с помощта на допълнителни операции. Така общата задача директно се редуцира до по-проста и няма нужда от обратен ход за получаване на решението на

общата задача. (fact-iter). При дървовидната рекурсия процесът на оценяване на обръщението към процедурата може да бъде разгледан като процес на обхождане в дълбочина на дърво.

8. Процедура от по-висок ред – процедури, които манипулират други процедури.

9. S-изрази – 1) Атоми: числа, символни низове, символни атоми; 2) Ако s1 и s2 са S-изрази, то (s1 . s2) е S-израз (т. двойка).

10. Списъците – (Scheme) Точкови двойки, които представят крайни редици от елементи; (Haskell) Редица от (променлив брой) елементи от определен тип. За всеки тип t в езика е дефиниран и типът [t], който е списък от елементи с тип t. ([ ] празен списък)

11. Област на действие на локалните имена, дефинирани със специалните форми let, let*, letrec – Докато при let свързването на имената и стойностите на локалните променливи става едновременно (може да се ползват само в <тяло>), то при let* това става последователно, т.е. в оценяването на втората променлива, може да се използва вече оценената първа променлива (навсякъде след първата дефиниция и тялото). При letrec може да се извършват локални свързвания, при които рекурсията е възможна, т.е областта на действие на локалните променливи съвпада с изразите на всяка една от тях и тялото на letrec.

12. Действие на вградените функции eq? и equal? – (eq? S_1 S_2) - #t, ако [S_1] и [S_2] са идентични (т.е. ако [S_1] и [S_2] са означения на един и същи участък в паметта), else #f; (equal? S_1 S_2) - #t, ако [S_1] и [S_2] са еквивалентни изрази/обекти (т.е. или са еднакви символни атоми, или са еднакви низове, или са равни числа от един и същ тип, или са точкови двойки (в частност списъци) с еквивалентни car и cdr, else #f; Ако [S1] и [S2] са еднакви символни атоми, то те са равни в смисъла и на двете дефиниции, но ако става въпрос за точкови двойки те не са еднакви в смисъла на eq?, а на equal? са.

13. Map – (map <процедура> <списък>) – оценяват се <процедура> и <списък>, като оценената процедурата се прилага едновременно към всеки от елементите на оценения списък. Само веднъж се оценяват елементите на оценения списък и като оценка се връща списък на получените резултати. Същността на процесите на изобразяване се свежда до едновременно извършване на един и същи тип обработка върху елементите на даден списък и формирането на списък от получените резултати.

14. Apply – (apply <процедура> <списък>) . Оценяват се <процедура> (която не е специална форма!) и <списък>. Процедурата apply предизвиква прилагане на вече оценената процедура върху аргументите на списъка от аргументи, като при това тези аргументи се оценяват само веднъж и връща получения резултат; (apply + '(1 2 3 4)) => 10

15. Механизъм на прилагане на съставна процедура към съответните аргументи съгласно модела на средите – Дадена <процедура> се прилага върху определено множество от аргументи чрез конструиране на специална таблица която съдържа свързвания на формалните параметри на процедурата със съответните фактически параметри (аргументи) на обръщение, последвано от оценяване на тялото на дефиницията в контекста на новата среда. При това новопостроената таблица има родителска среда (същата среда която се сочи от указателя към средата в представянето на прилаганата процедура). Процедура се създава чрез оценяване на определен lambda израз в съответната среда. В резултат се създава специален обект (процедура) който може да се разглежда като двойка, съдържаща текста на lambda израза и указател към средата в която се съдържа процедурата.

16. Действие на вградените функции set-car! и set-cdr! – При set-car! има два аргумента, като първият

задължително трябва да бъде точкова двойка (в частност непразен списък). Процедурата set-car! модифицира първия си аргумент, като в него заменя съществуващия указател към car-а с указател към втория аргумент. Обръщението към set-cdr! има същия вид и същите аргументи като при set-car!, но в резултат на действието и указателят към cdr-а на първия аргумент се заменя с указател към втория. Оценката към обръщението на двете функции по стандарт е неопределена.

17. Принципи за реализация на потоци и действие на специалните форми delay и cons-stream – Една възможност за реализация на потоците е свързана с представянето им във вид на списъци. Ако се избере този подход тогава head съвпада с car, tail с cdr, cons-stream с cons, the-empty-stream с nil, empty-stream? с null?. Но тази реализация е неефективна. Основната идея на действителната реализация на потоците, която ги прави подходящи структури за представяне на редици, съдържащи голям брой елементи, дори безкрайно много, се основава на т.нар. забавени изчисления (отложено оценяване – delay evaluation), които са средство за „пакетиране" на съответните изрази и позволяват тези изрази да бъдат оценявани едва тогава, когато оценяването им е наистина необходимо. В този смисъл основната идея при реализацията на cons-stream е тази процедура да конструира потока само частично и да предава така конструирания от нея специален обект на процедурата, която го използва. Ако тази процедура иска да получи достъп до неконструираната част, то се конструира само тази част, която реално е необходима. Конструирането на останалата част се отлага до евентуалното възникване на необходимост от достъп до нови елементи. За целта се използва специалната форма delay: (delay <израз>) В резултат към това обръщение се получава пакетиран вариант на <израз>, който позволява реалното оценяване на <израз> да се извърши тогава, когато е необходимо.

18. Методи за конструиране на безкрайни потоци – Използват се два основни подхода при конструирането на безкрайни потоци: неявно (индиректно) и явно (директно) конструиране. При първия подход се използват специални процедури, наричани генератори, а при втория - рекурсия върху съответния генериран поток (по отношение на вече генерираните части на потока).

19. **Определяне обхвата на списък (list comprehension)** – Синтаксис: [expr | q_1, … ,q_n], където expr е израз, а q_i може да бъде: 1) генератор от вида p<=lExp, където p е образец и lExp е израз от списъчен тип; 2) тест, bExp, който е булев израз. При това в q_i могат да участват променливите използвани преди него. ([2*n | n<=[2, 4, 7], isEven n, n>3] => [8])

20. Примитивна рекурсия върху списъци – В дефиницията, която описва този тип рекурсия се описват следните типове случаи: Начален (прост, базов) случай и общ случай, в който се посочва връзката между стойността на функцията за дадената стойност на аргумента и стойността на функцията за по проста в определен смисъл стойност на аргумента

Пр. Sum:: [Int]-> Int; sum[] = 0;

Sum (x:xs) = x + sum xs;

Обща рекурсия: не се подчинява на ограниченията на примитивната. Схемата на дефиниране е специфична. Пример за нея е дървовидната рекурсия на qsort.

21. Операторите (деф. и свойства) – В Haskell биват булеви, аритметични, целочислени, за сравнения, оператори за работа с числа плаваща запетая. Чрез заграждане на името на всяка двуаргументна функция в обратни апострофи е възможно записа на обръщението към тази функция да стане инфиксен. Пример: 14 `div` 3 (ако не са дефинирани с име не е задължително, например за + и -)

22. Образци (pattern) видове и правила – Пример: mystery 0 y = y;
mystery x y = x;
В тези две равенства последователно са описани отделните възможни случаи за стойността на mystery, като за целта се използват образци - литералът 0 и променливите x y. Равенствата се прилагат последователно, като до всяко следващо се стига само ако предните не са дали резултат. Видове образци: 1) Литерал, като например 24 'f' true, даден аргумент се съпоставя успешно с такъв образец, ако е равен на неговата стойност; 2) Променлива като например x. Образец от такъв вид се съпоставя успешно с аргумент с произволна стойност; 3) специален израз за безусловно съпоставяне '_' (wildcard), който е съпоставим с произволен аргумент; 4) Вектор образец (p_1, p_2, …, p_n), за да бъде съпоставим с него аргумента трябва да има вид (v_1, v_2, …, v_n), като всяко v_i трябва да е съпоставимо с p_i;
5) Конструктор, приложим към дадено множество от аргументи.

23. Дефиниране на функция на функционално ниво – предполага тази функция да се опише не в термините на резултата, който връща тя при прилагане върху подходящо множество от аргументи, а като директно се посочи връзката и с други функции. Пр.ако са дефинирани функциите:
f :: b => c и g :: a => b
То тяхната композиция е:
(.) :: (b => c) => (a => b) => (a => c)

24. Частично прилагане на функция и тип на получения резултат – Всяка функция на 2 или повече аргумента може да бъде приложена частично към по-малък брой аргументи. Тази идея дава богати възможности за конструиране на функции като оценки на обръщения към други функции.
multiply :: Int => Int = > Int
multiply x y = x*y
Правило на изключването: Ако дадена функция f е от тип t1 => t2 => ... => tn => t и тази функция е приложена към аргументи e1 :: t1, e2 :: t2, ..., ek :: tk (където k <= n) то типът на резултатът се определя чрез изключването на типовете t1, t2, …, tk, т.е. резултатът е от тип tk+1 => … => tn => t.

25. Същност на понятието „сечение на оператор" (operator section) – Операторите могат да бъдат прилагани частично, като за целта се задава това което е известно, под формата на т.н. сечения на оператори. Общото правило гласи, че сечението на оператора *op* „добавя" аргумента си по начин, който завършва от синтактична гледна точка записа на приложението на оператора (обръщение към оператора); Пример:
(op x) y = y op x ; (x op) y = x op y;
например (+ 2) е функция, която прибавя към аргумента си числото 2; (2 +) е функция, която прибавя аргумента си към числото 2; (> 2) е функция, която проверява дали дадено число е по-голямо от 2.

26. Същност на „мързеливото" оценяване (lazy evaluation) – Интерпретаторът оценява даден аргумент на дадена функция само, ако стойността на този аргумент е необходима за пресмятането на целия резултат. Ако даден аргумент е съставен (например вектор или списък), то се оценяват само тези негови компоненти, чиито стойности са необходими от гледна точка на оценяването на резултата. При това дублиращите се подизрази се оценяват по не повече от 1 път.

27. Дефиниране на класове в Haskell – Понятието клас се определя като колекция от типове, за които се поддържа множество додефинирани операции, наречени методи. Множеството от типове, за които са дефинирани съответно множество от функции, се нарича клас от/на типовете (type class) или накратко клас. Например, множеството от типове, за които е дефинирана функция за проверка на равенството (==), се означава като клас Eq. За да може да се дефинира един клас, е необходимо да се избере неговото име и да се опишат ограниченията, които трябва да удовлетворяват даден тип a, за да принадлежи на този клас. Типовете, които принадлежат на даден клас се наричат екземпляри на даден

клас. Най-важното за класа Eq е наличието на функцията == от тип a->a-> Bool, която проверява дали 2 елемента на даден тип a, който е екземпляр на Eq, са равни:

Class Eq where (==) :: a->a->Bool