

ФУНКЦИОНАЛНО ПРОГРАМИРАНЕ

спец. Компютърни науки, II курс (3+0+2)

ФМИ, СУ „Св. Климент Охридски“
2017/2018

Магдалина Тодорова

magda@fmi.uni-sofia.bg

todorova_magda@hotmail.com

кабинет 517, ФМИ

Организация на курса

Оценяване:

- провеждат се 2 задължителни контролни (К1-8-ма седмица и К2 – 15-та седмица);
- провеждат се контролни и се дават домашни на лабораторните упражнения (ТК);
- провеждат се писмен и устен изпити.

Крайната оценка се получава по правилото:

$$\frac{K1 + K2 + ТК}{3} \cdot \frac{2}{5} + \frac{\text{писмен} + \text{устен}}{2} \cdot \frac{3}{5}$$

Освобождаване от писмен изпит

$$\frac{K1+K2+TK}{3} \geq 4.50$$

Материали за курса

<https://learn.fmi.uni-sofia.bg/>

ОСНОВНА ЛИТЕРАТУРА

(за Scheme)

- Тодорова М., *Езици за функционално и логическо програмиране, първа част – функционално програмиране*, преработено и допълнено издание, СИЕЛА СОФТ ЕНД ПУБЛИШИНГ, София, ISBN 978-954-28-0828-2, 2010, 227 стр.
- Нишева, М., П. Павлов. *Функционално програмиране на езика Scheme*. София, 2004
- Абелсон Х., Дж. Сасмън, *Структура и интерпретация на компютърни програми*, TEMPUS JEP 1497, СОФТЕХ, София, 1994

ДОПЪЛНИТЕЛНА ЛИТЕРАТУРА

- H. Abelson, G. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, 1985.
- R. Kent Dybvig, *The Scheme Programming Language*, Printice-Hall, 1987.
- *Среда за програмиране на Scheme/Racket, DrRacket*
(<http://racket-lang.org/>)

ТЕМА № 1

ОСНОВИ НА ФУНКЦИОНАЛНОТО ПРОГРАМИРАНЕ

I. Scheme/Racket

ПЛАН НА ЛЕКЦИЯТА

- Основни стилове за програмиране
- Силове на функционално програмиране
- Езици за функционално програмиране
- Среда Scheme/Racket
- Основни понятия в програмирането на Scheme/Racket
- Дефиниране на процедури
- Условни изрази и предикати

Стилове на програмиране

Съществуват два основни стила за програмиране:

- императивен (процедурен)
- дескриптивен (декларативен).

Стилове на програмиране (процедурен)

В процедурните езици програмата се реализира по известната схема

програма = алгоритъм + структури от данни

В основата на програмата при императивното програмиране стои **алгоритъмът**. Той определя последователните етапи на обработката на данните с цел получаване на търсения резултат. Решаването на една задача се свежда до точно описание на това **как** се получава съответният резултат.

Този стил е пряко свързан с архитектурата на съвременните компютри, която се нарича Нойманова по името нейния създател - Джон фон Нойман. Процедурните програми са абстрактна версия на този тип архитектура.

Стилове на програмиране (дискриптивен)

При декларативните езици за програмиране в програмата явно се посочва какви свойства има желаният резултат (или **какво** е известно за свойствата на предметната област и в частност на резултата), без да се посочва как точно се получава този резултат.

Т.е. при декларативните езици се посочва **какво** се пресмята, без да се определя **как** да се пресмята.

Стилове на програмиране (дискриптивен)

ПРОГРАМА = СПИСЪК ОТ ФУНКЦИИ ИЛИ
РЕДИЦА ОТ РАВЕНСТВА ИЛИ
ПРАВИЛА + ФАКТИ

Схемата на изпълнението има вида:

ИЗПЪЛНЕНИЕ = РЕДУКЦИЯ ИЛИ
ИЗВОД (ДОКАЗАТЕЛСТВО)

Дескриптивните езици описват:

КАКВО се пресмята, а не **КАК**.

Семантиката е скрита, вградена е в реализацията.

Стилове дескриптивни езици за програмиране

Съществуват три основни стила за дескриптивно програмиране:

- функционално;
- логическо;
- интегриращо функционалното и логическото програмиране.

Функционално програмиране

Функционалното програмиране е начин за съставяне на програми, при който единственото действие е обръщението към функции, единственият начин за разделяне на програмата на части е въвеждането на име на функция и задаването за това име на израз, който пресмята стойността на функцията, а единственото правило за композиция е суперпозицията на функции.

Съществуват следните стилове за функционално програмиране:

- ✓ лиспоподобен;
- ✓ равенстов;
- ✓ FP-стил

Лиспоподобно ФП

Програмите на лиспоподобен език са списъци от дефиниции на функции. Дефиницията на функция е списък, задаващ името, формалните параметри и тялото на функцията.

Теоретичен модел на лиспоподобното функционално програмиране е λ -смятането, въведено от Алонсо Чърч (1941 г.).

На базата на λ -смятането през 1958-61 год. Дж. Маккарти разработи езика за ФП Lisp, който е един от най-мощните езици за програмиране.

Програмиране с равенства

Програмите на равенстов език са съвкупности от равенства от вида $A == B$. Равенствата са ориентирани в смисъл, че B може да замени A , но A не може да замени B .

През 1977 год. М. О'Donnell математически обосновава изчисленията в системи, описани с равенства. Тези системи се наричат още системи за заместване на поддървета.

Пример:

$$D(c, x) == 0 \quad ; \text{ c е константа}$$

$$D(c * x, x) == c$$

$$D(u + v, x) == D(u) + D(v)$$

$$D(u * v, x) == u * D(v, x) + v * D(u, x)$$

$$D(x^n, x) == n * x^{n-1}$$

Програмиране с равенства

Пример:

$$D(c, x) == 0$$

$$D(c*x, x) == c$$

$$D(u+v, x) == D(u)+D(v)$$

$$D(u*v, x) == u*D(v, x) + v*D(u, x)$$

$$D(x^n, x) == n*x^{n-1}$$

$$D(2*x^3+4*x, x) \rightarrow D(2*x^3, x) + D(4*x, x) \rightarrow$$

$$2*D(x^3, x) + x^3*D(2, x) + 4 \rightarrow$$

$$2*3*x^2 + 0 + 4 \rightarrow$$

$$6*x^2 + 4$$

FR-стил

През 1978 г. J. Backus поставя основите на т. нар. FR-стил на програмиране. FR-програмите са или примитивни форми, или дефиниции, или функционални форми.

Backus доказва, че FR-системите имат следните важни свойства:

- а) проста формална семантика;
- б) ясна йерархична структура на FR-програмите, при която програми от високо ниво могат да се комбинират и да образуват програми от още по-високо ниво;
- в) основните FR-форми за комбиниране са операции в съответна алгебра на програми, която може да се използва за трансформиране на програми, за доказване на свойства на програми и др.

Езици за ФП

BETA	Interpreted	Pliant
Caml	Leda	POP-11
Clean	Lisp	R
CLOS	Logo	REBOL
Compiled	Lua	Refal
Compilers	Maple	Research
Dylan	MathematicaMercury	S
Emacs	Miranda	Scheme
Erlang	ML	Sisal
FP	Objective Caml	Spreadsheets
Functional Logic	Oz-Mozart	
Guile		
Haskell		

Типове езици за функционално програмиране

Програмата на един функционален език се състои от редица от уравнения, чрез които се описват някакви функции.

Редът, в който са подредени уравненията, не е съществен (за разлика от операторите в програмите на императивните езици).

Идеята за функционалност изключва използването на оператори за присвояване и управление (за преход, за цикъл).

Поради съображения за ефективност обаче голяма част от използваните в практиката функционални езици съдържат императивни конструкции, които са подобни на операторите за присвояване и операторите за управление.

Типове езици за функционално програмиране

Функционалните езици могат да се разделят на:

- ✓ **строги (чисти) функционални езици** – при тях единствените управляващи структури са функциите. Странични ефекти не се допускат. Такива езици са: Pure Lisp, HOPE, Miranda, Haskell, FP, W и др.;
- ✓ **нестроги функционални езици** – те съдържат императивни конструкции и допускат използването на нестроги функции. Такива езици са: Common Lisp, Scheme, ML и др.

Език за програмиране ЛИСП

Основни сведения за езика Lisp

Езикът Lisp (LISP = LISt Processing) е създаден от Дж. Маккарти в края на 50-те години на 20-ти век.

Той е функционален език, предназначен да обработва символни (аналитични, нечислови) данни, записани във вид на списък. Може да бъде определен като функционален език за символни (аналитични, нечислени) пресмятания (преобразования).

Lisp е сред най-широко използваните езици за програмиране в областта на изкуствения интелект.

Основни сведения за езика Lisp

Диалекти на езика Lisp:

Lisp 1.5, MacLisp, Franz Lisp, Standard Lisp, Scheme, Common Lisp.

Най-новият диалект на Lisp е Common Lisp. Той е утвърден като търговски стандарт за езика Lisp.

Scheme е създаден за целите на обучението. Той е с по-ограничени възможности и запазва в по-голяма степен и в по-чист вид идеите на функционалното програмиране.

Racket се сочи за наследник на Scheme.

Основни сведения за езика Lisp

Съвременните *среди за програмиране на Lisp* съдържат:

- интерпретатори (често и компилатори) на езика;
- редактори (за Common Lisp те са варианти на редактора Emacs);
- обектно-ориентирани разширения на езика;
- средства за връзка с програми, написани на други езици (Prolog, Ada, Pascal, C и др.).

Средата за програмиране на Scheme/Racket, която ще бъде използвана на упражненията по ФП, е DrRacket

<http://racket-lang.org/>

Основни сведения за езика Lisp

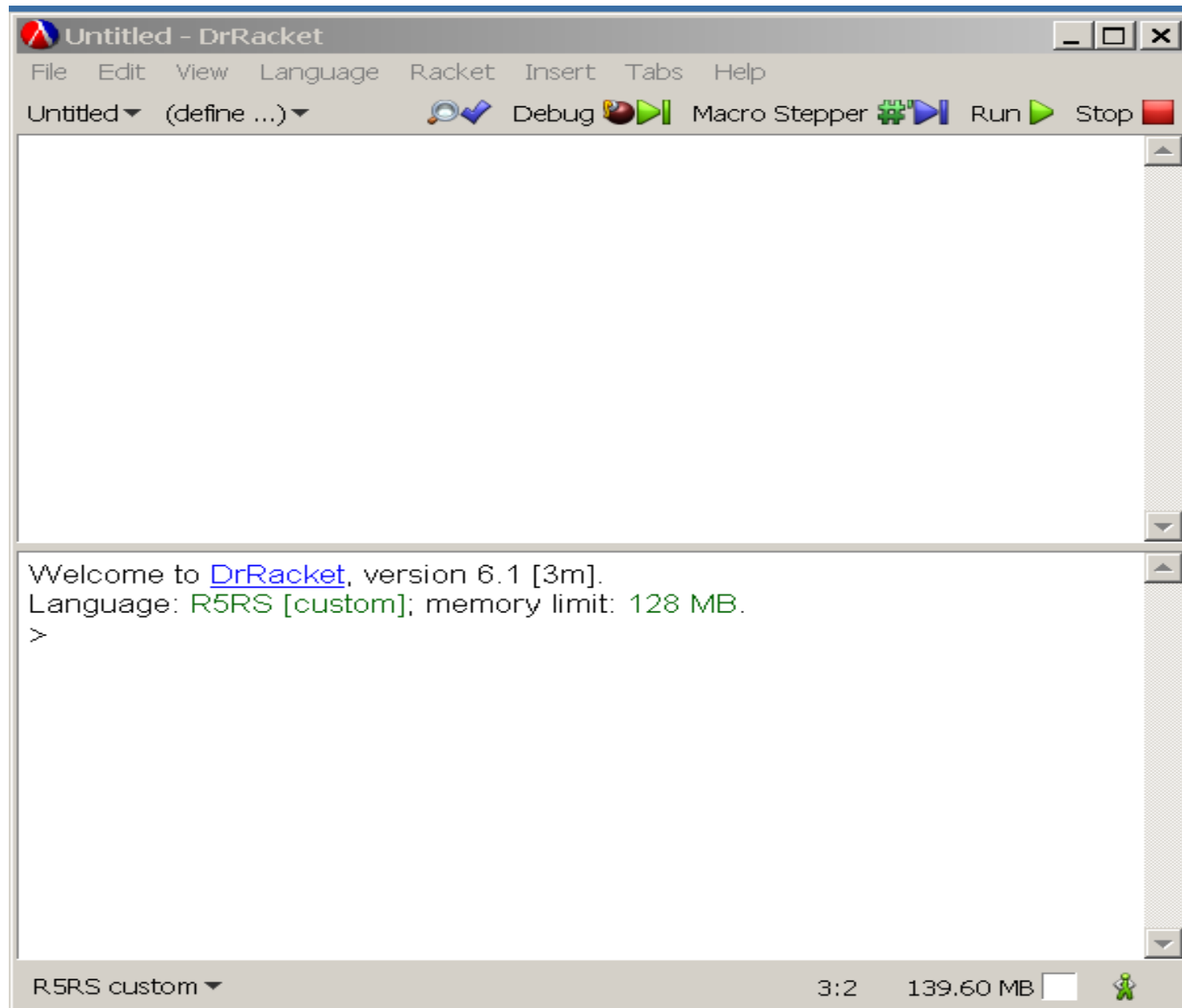
Забележка:

В Scheme като синоним на термина „функция“ се използва терминът „процедура“.

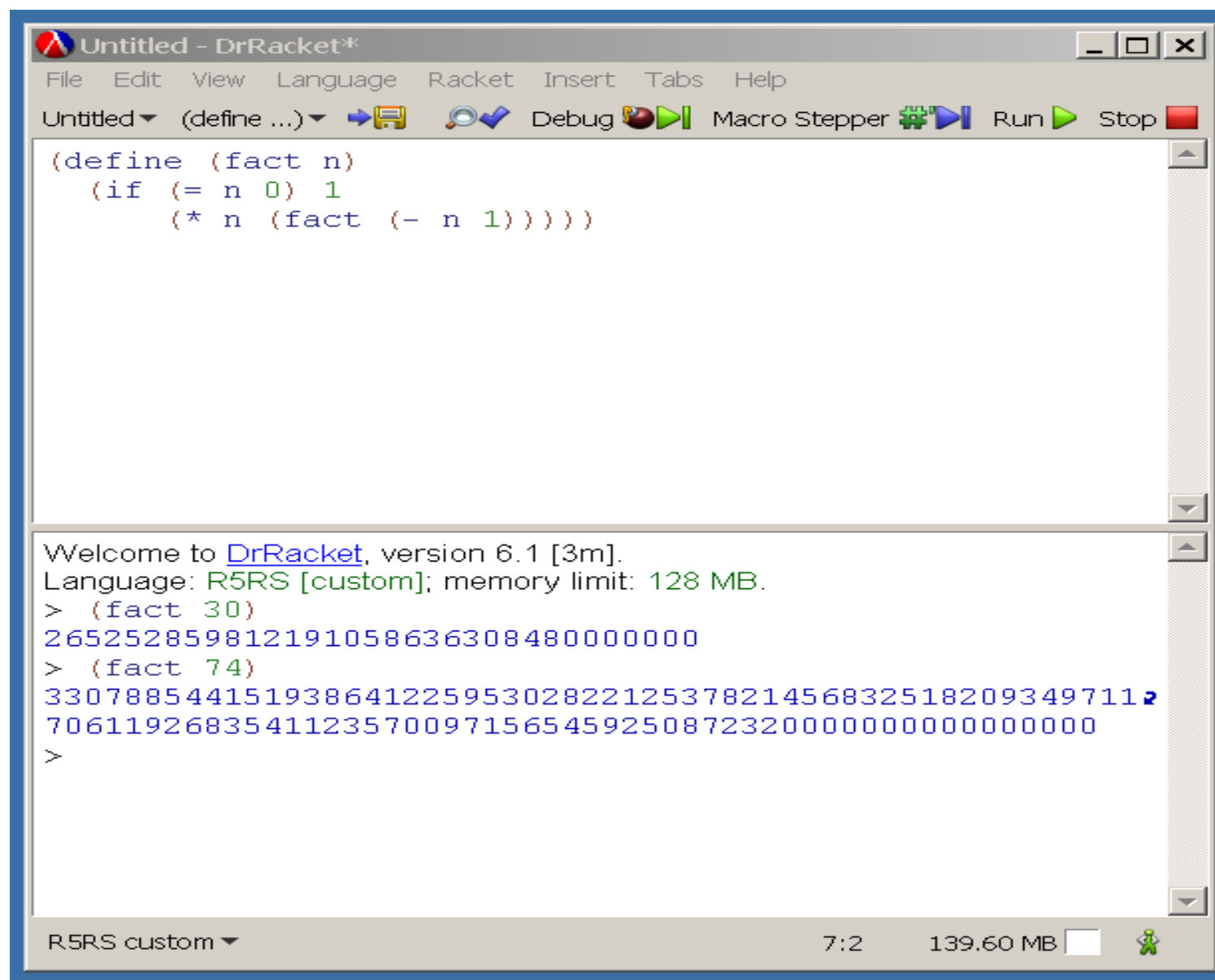
Терминът „процедура“ в смисъл на описание на даден изчислителен процес често се предпочита, тъй като в Scheme, както и в другите съвременни диалекти на езика Lisp, няма изисквания за строга функционалност. В този смисъл терминът „функция“ понякога не е съвсем точен.

Затова в следващото изложение на езика Scheme термините „функция“ и „процедура“ обикновено ще бъдат използвани като синоними.

Средата за програмиране Scheme/Racket



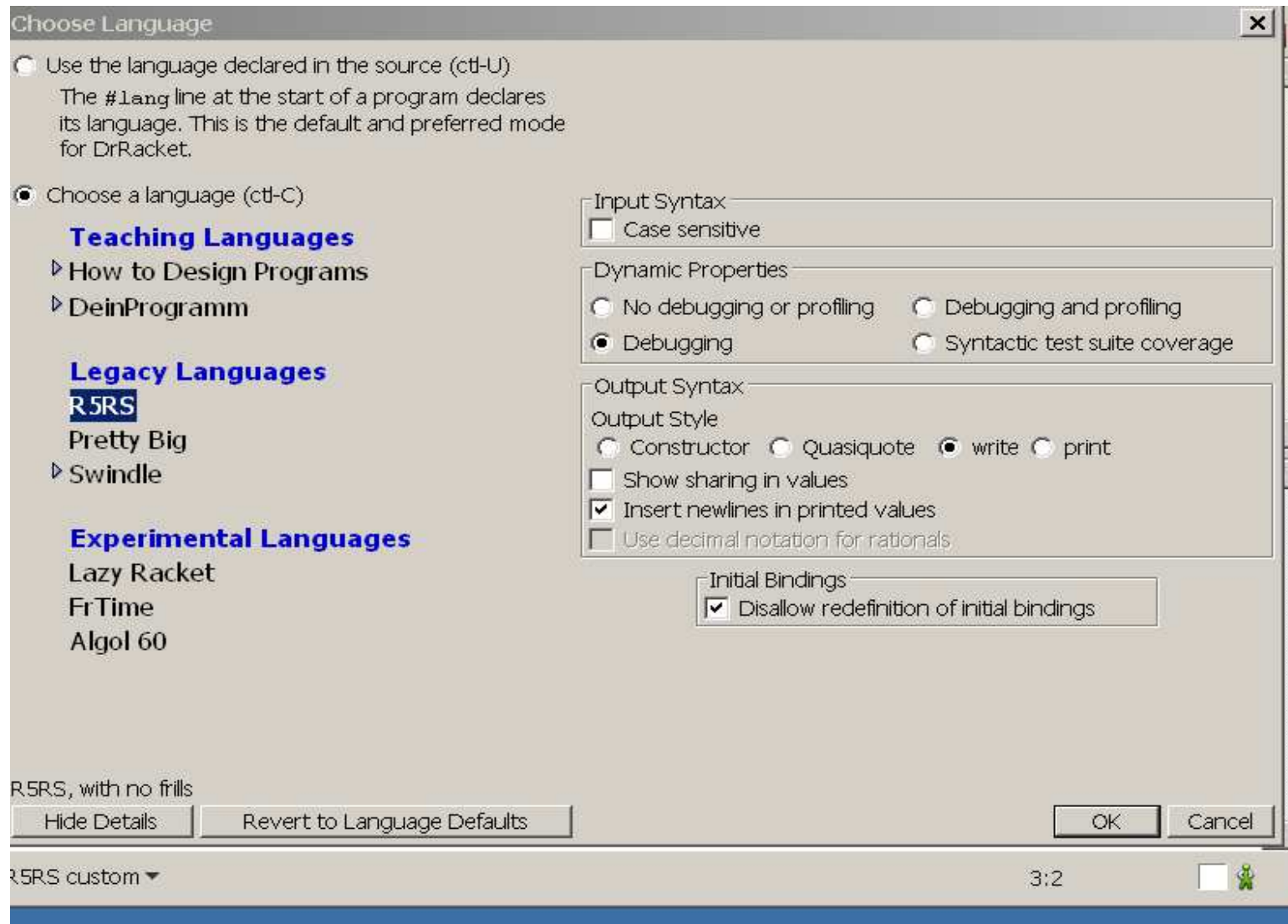
Средата за програмиране DrRacket

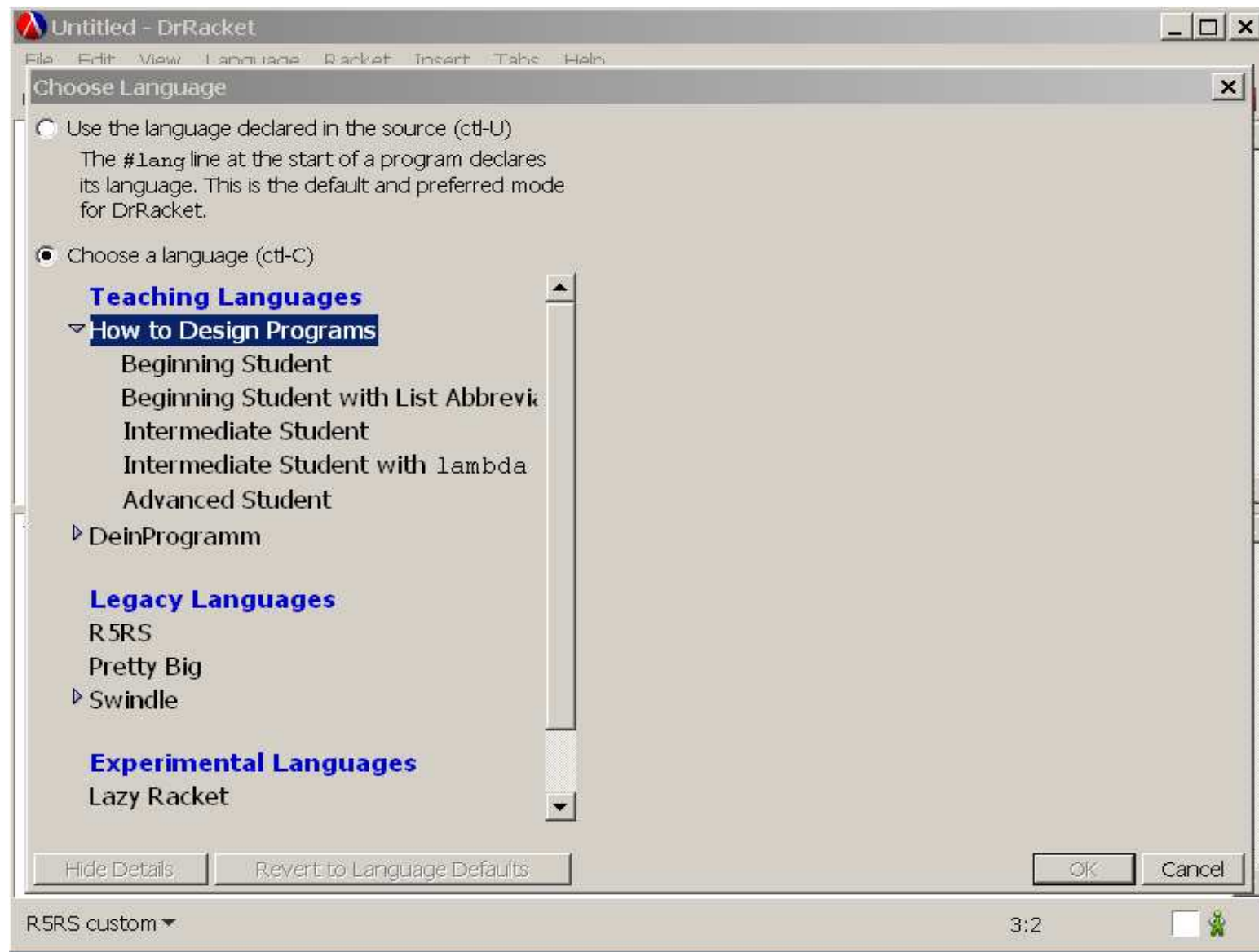


DrRacket предлага йерархия на езиковите нива (Language):

- Beginning Student language (две версии) – език за начинаещи; включва функции, структури и списъци.
- Intermediate Student language (две версии) – език за средно ниво на познание; добавя в обсега си и лексикални конструкции.
- Advanced Student language – език на напредналите; позволява мутации и поддържа разширения.

Езиковите нива на Scheme предлагат стандартен Scheme с или без графични възможности и дебъгер.





Основни понятия

► Примитивни изрази

▷ атоми

◆ символи

◆ числа

◆ низове

▷ списъци

} обекти

► Комбинации

► Средства за абстракция

Атоми

Атомите могат да се разделят на три основни групи:

а) символи (символни атоми) – крайни редици от букви (малки или главни), цифри, специалните знаци: +, -, *, /, <, >, =, ?, .., %, &, !, _, ^
без (,), [,], {, }, ', ", запетая и интервали.

Примери: Редиците по-долу са символи

X abC 1x 1+ 1-

Примери: Редиците по-долу не са символи

A(1) a'12 12345 "a12 a1,3

Числата не са символи.

Някои символи са идентификатори, например:

A – символ от една буква,

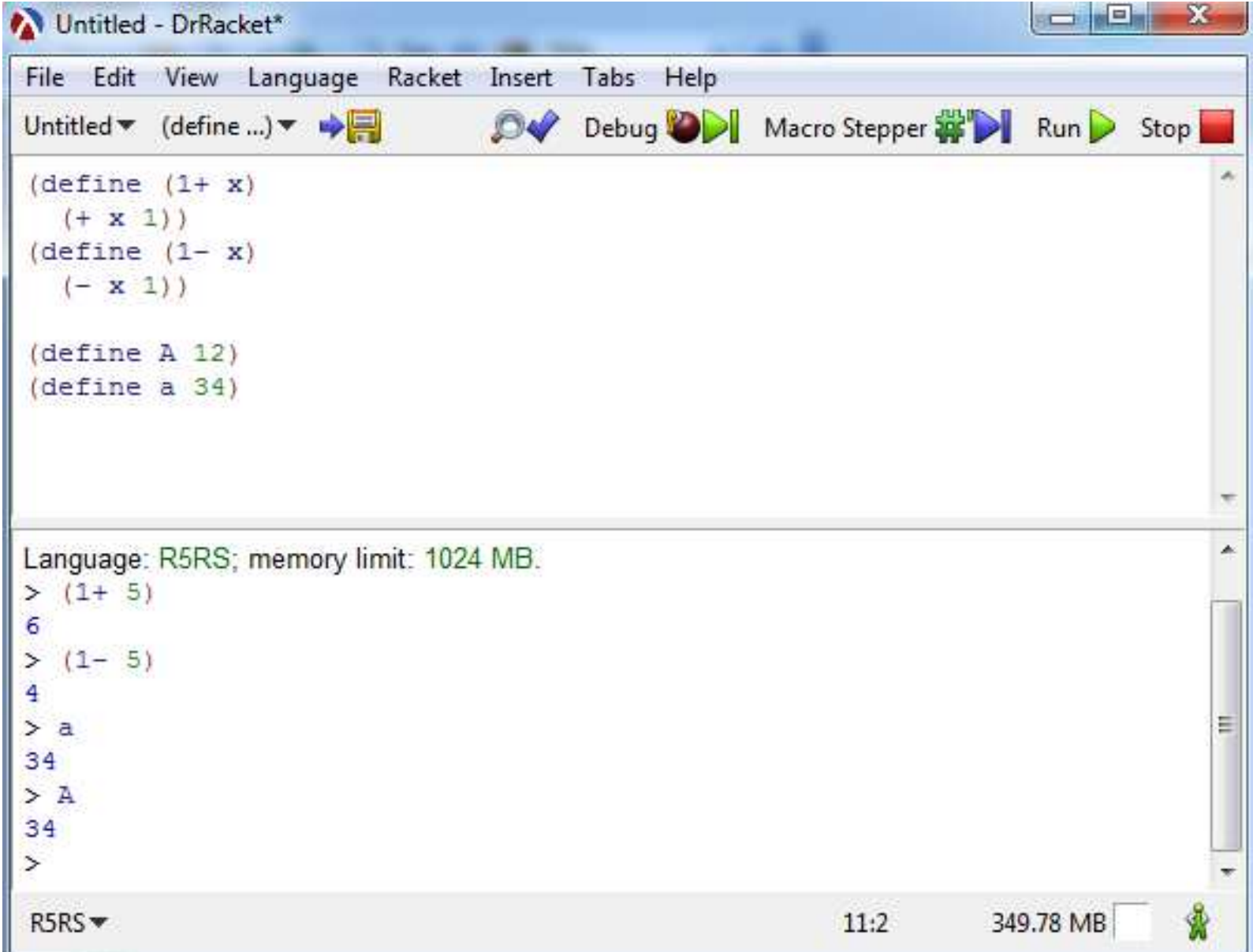
R12 – символ, в който има цифри

АТОМ – символ от 4 букви,

СПОРТ_ТОТО – символ от 10 букви
(знакът _ се възприема като буква).

Забележки:

1. DrRacket допуска Кирилски букви.
2. Стандартен Scheme не е чувствителен за малки и главни букви (Abc, aBc, abC, ABc, AbC са един и същ символ);
3. Някои версии на DrRacket са чувствителни, а други не са чувствителни за малки и главни букви.



The screenshot shows the DrRacket IDE window titled "Untitled - DrRacket*". The menu bar includes File, Edit, View, Language, Racket, Insert, Tabs, and Help. The toolbar contains icons for saving, debugging, macro stepping, running, and stopping. The main text area contains the following R5RS code:

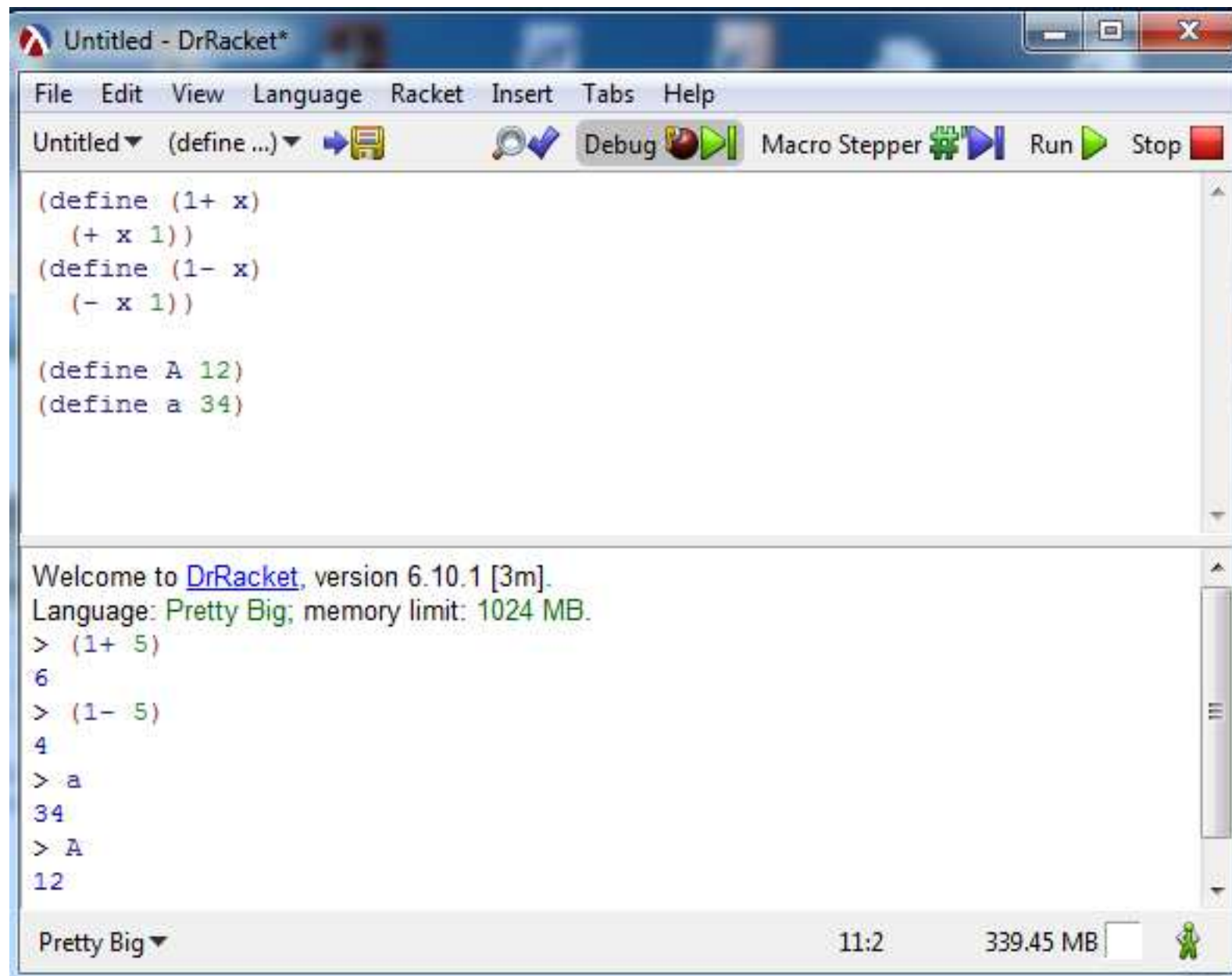
```
(define (1+ x)
  (+ x 1))
(define (1- x)
  (- x 1))

(define A 12)
(define a 34)
```

Below the code, the language is set to R5RS with a memory limit of 1024 MB. The interaction area shows the following commands and results:

```
> (1+ 5)
6
> (1- 5)
4
> a
34
> A
34
>
```

The status bar at the bottom indicates the language is R5RS, the version is 11:2, and the memory usage is 349.78 MB.



б) Числа (числени атоми) – цели и реални числа, които се записват по общоприетия в езиците за програмиране начин – например както на езика C++.

► **двоични #b (binary)**

▷ #b0001

► **осмични #o (octal)**

▷ #o23401

► **шестнадесетични #x (hexadecimal)**

▷ #xe1e10

► **десетични #d (decimal)**

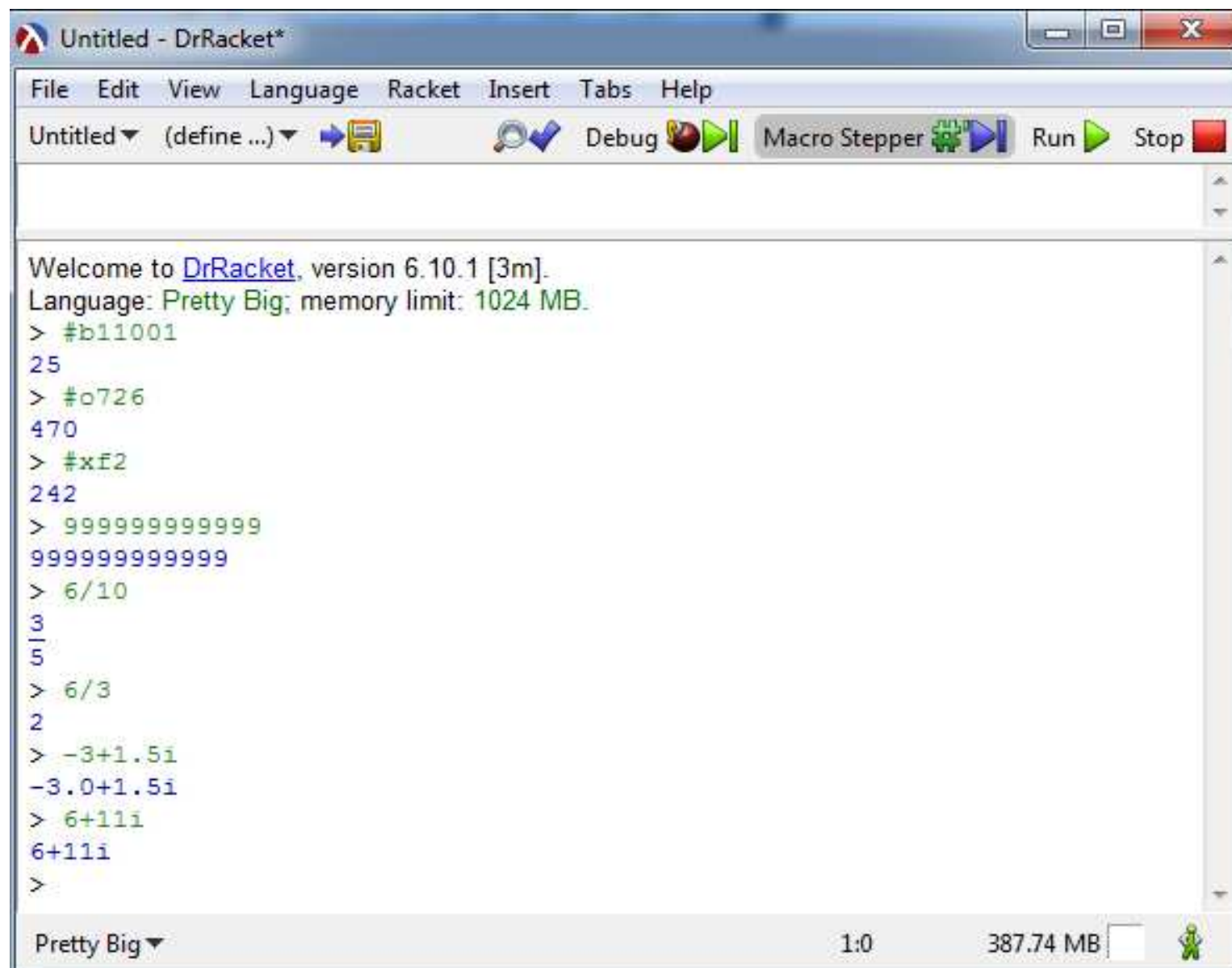
▷ #d59

▷ 99

▷ 876438763486

б) числени атоми

- ▶ **рационални** (числител и знаменател, разделени с наклонена черта)
 - ▷ $6/10$
 - ▷ $6/3$
- ▶ **комплексни**
 - ▷ $3+4i$
 - ▷ $3-i$
- ▶ **реални с неограничена точност и пр.**
 - ▷ 334.92387923879287
 - ▷ -2.5



The screenshot shows the DrRacket IDE window titled "Untitled - DrRacket*". The menu bar includes File, Edit, View, Language, Racket, Insert, Tabs, and Help. The toolbar contains buttons for saving, debugging, macro stepping, running, and stopping. The main text area displays the following text:

```
Welcome to DrRacket, version 6.10.1 [3m].  
Language: Pretty Big; memory limit: 1024 MB.  
> #b11001  
25  
> #o726  
470  
> #xf2  
242  
> 9999999999999999  
9999999999999999  
> 6/10  
3  
5  
> 6/3  
2  
> -3+1.5i  
-3.0+1.5i  
> 6+11i  
6+11i  
>
```

The status bar at the bottom shows "Pretty Big" as the selected language, a zoom level of "1:0", and a memory usage of "387.74 MB".

в) низове – произволна редица от знаци, заградени от специален знак, зависещ от конкретната реализация.

Примери:

"А" – низ от един знак,

"А = В + С" – низ от 9 знака,

"((:=8иг *НН" – низ от 11 знака.

S-изрази

S-изразите са основна конструкция на езика. Чрез тях се задават съставни структури на езика.

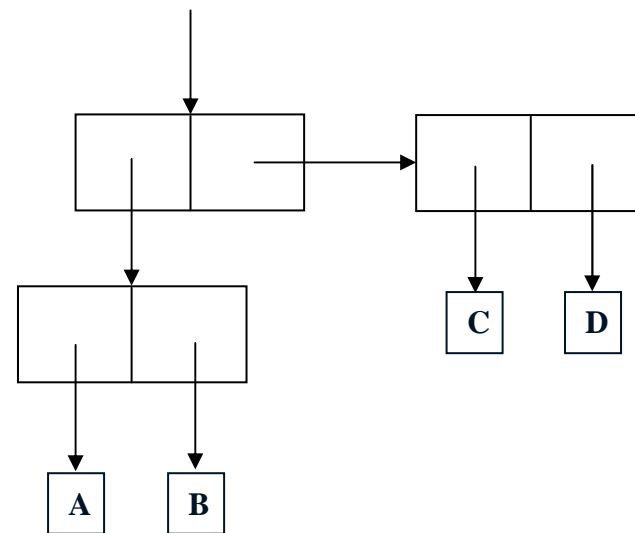
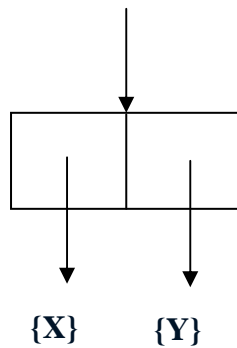
Те се дефинират посредством следните три правила:

- а) всеки атом е S-израз;
- б) ако X и Y са S-изрази, то $(X . Y)$ е също S-израз;
- в) S-изразите се определят само по правилата а) и б).

Двойката $(X . Y)$ се нарича още точкова двойка. Така всеки S-израз е или атом, или точкова двойка.

Пример: Нека A , B , C , и D са атоми. Тогава $((A . B) . (C . D))$ е S-израз.

Удобен начин за представяне на точковите двойки са двойните кутии:



Списъци

Нека $X_1, X_2, X_3, \dots, X_N$ са S-изрази. Тогава изразът:

$$(X_1 . (X_2 . (X_3 . (\dots . (X_N . \text{NIL}) \dots))))$$

се нарича **списък**.

Прието е такъв списък да се записва съкратено във вида:

$$(X_1 X_2 X_3 \dots X_N)$$

където $X_1, X_2, X_3, \dots, X_N$ са елементите на списъка.

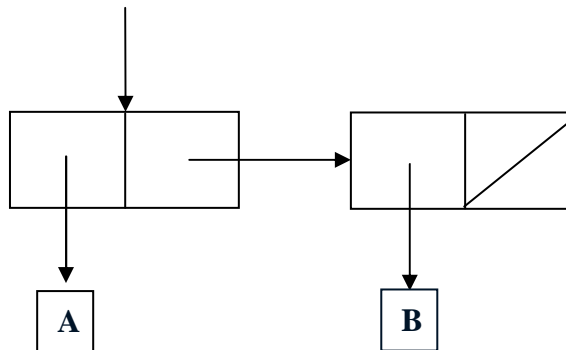
Тъй като $X_1, X_2, X_3, \dots, X_N$ са S-изрази, то те могат да бъдат атоми или точкови двойки. По този начин могат да се влагат списъци един в друг.

*По дефиниция празният списък $()$ е еквивалентен на атома **NIL!!!***

*Забележка: В DrRacket вместо **nil** се използва **null**, или се използват и двете.*

Примери

Съкратеният запис на S-израза (**A . (B . NIL)**) е (**A B**), а графичното му представяне е



Комбинации

Дефиниция: Списък от изрази (примитивни или съставни), най-левият елемент на който е процедура, се нарича **комбинация** (*процедурно прилагане*).

Най-левият елемент се нарича **оператор**, а следващите го – **операнди** или **аргументи на оператора**.

Синтаксис:

$(\text{proc } \text{arg}_1 \text{ arg}_2 \dots \text{arg}_n)$

- proc е израз, оценката на който е процедурен обект;
- $\text{arg}_1 \text{ arg}_2 \dots \text{arg}_n$ са изрази, които представят фактическите параметри (операндите) на процедурния обект.

Синтаксис:

$(\text{proc } \text{arg}_1 \text{ arg}_2 \dots \text{arg}_n)$

- *proc* е израз, оценката на който е процедурен обект;
- $\text{arg}_1 \text{ arg}_2 \dots \text{arg}_n$ са изрази, които представят фактическите параметри (операндите) на процедурния обект.

Семантика:

Оценяват се *proc*, arg_1 , arg_2 , ... и arg_n отляво надясно. Оценката на комбинацията се получава като се прилага процедурният обект, който е оценка на *proc* към оценките на операндите. Тази семантика определя т.н. **основно правило** или **апликативен модел на оценяване**.

Основно правило за оценяване (апликативен модел):

- оценяват се подизразите, съставлящи комбинацията;
- прилага се процедурният обект, който е оценка на най-левия подизраз (т.е. операторът), към операндите (аргументите), които са оценки на останалите подизрази.

Примери:

(+ 2 12)

(- 34 12)

(* 8 6 9 8)

(/ 34 2 5)

(+ 32 45 6 8 2)

(+ (- 12 5) (+ 7 8 2) (* 4 2 3))

Пример: Оценяването на

$$(* (+ 2 (* 4 6)) (+ 3 5))$$

изисква прилагане на основното правило върху четири различни комбинации.

Забележка:

В комбинациите операторът е в началото, а операндите са след него. Това означение се нарича префиксно.

Предимства на префиксния запис:

- Дава възможност процедурите да имат произволен брой аргументи.

(/ 1234.567 8.4 6.2 3.4)

(* 3 5 6 7 8 9 123 2)

- Позволява влагане на комбинации (възможно е операндите на оператора да са комбинации)

(+ (- 3.4 4) (* 1.2 8.9 2.3))

Няма ограничения за дълбочината на влагане и за сложността на изразите, които интерпретаторът на езика може да оцени.

Прегледно записване на комбинациите (pretty print)

Комбинцията

$$(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))$$

Може да се запише по следния по-ясен начин:

$$\begin{aligned} & (+ (* 3 \\ & \quad (+ (* 2 4) \\ & \quad \quad (+ 3 5))) \\ & (+ (- 10 7) \\ & \quad 6)) \end{aligned}$$

Интерпретаторът оценява изразите (примитивни или съставни) по следния начин:

- Чете израз, въведен от клавиатурата (**R**ead);
- Оценява го (**E**val);
- Извежда получения резултат (**P**rint).

Тези три стъпки се повтарят за всеки оценяван израз и затова се казва, че интерпретаторът работи в цикъла **REP**.

Как интерпретаторът различава обикновен списък от комбинация?

Специална форма quote

Синтаксис:

(quote <израз>)

където <израз> е произволен израз.

Семантика:

Оценката на (quote <израз>) е <израз> без каквито и да са промени.

Примери:

>(quote (1 2 3 4 5))

(1 2 3 4 5)

>(quote ("a" "b" "c"))

("a" "b" "c")

>(quote 15.5)

15.5

Специална форма quote

Синтаксис:

'<израз>

където <израз> е произволен израз.

Примери:

> '(1 2 3 4 5)

(1 2 3 4 5)

> '("a" "b" "c")

("a" "b" "c")

Отговор на въпроса:

Как интерпретаторът различава обикновен списък от комбинация?

Трябва точно да се каже на интерпретатора как да разглежда списъка – като обикновен или като процедурно прилагане.

Ако списъкът е цитиран, разглежда се като обикновен. В противен случай, ако оценката на първия му обект е процедурен обект, списъкът е комбинация, а ако не е – е обикновен списък.

Как интерпретаторът оценява изрази?

а) Оценка на атом

>12.5

12.5

>"abcdef"

"abcdef"

>+

#<procedure: +>

>a

a: undefined;

cannot reference undefined identifier

Как интерпретаторът оценява изрази?

б) Оценка на израз, който е цитиран

>'a

a

>'(1 2 3 4 5)

(1 2 3 4 5)

>'(+ 2 5)

(+ 2 5)

в) Оценката на комбинация се осъществява по основното правило (апликативния модел).

г) Оценката на списък, който не е комбинация, предизвиква грешка.

>(1 2 3 4 5)

error

Средства за абстракция

Специална форма define

I.Формат

Синтаксис:

(define <име> <израз>)

- <име> е произволен символ;
- <израз> е произволен израз.

Семантика:

Интерпретаторът чете, оценява и извежда на екрана <име>. Оценката се осъществява по следния начин: В средата (паметта), в която define-изразът се оценява, се записва <име> и се свързва с оценката на <израз> в същата среда.

Средства за абстракция

Специална форма define

I. Формат

Примери:

>(define R 2)

R

>R

2

>>(* R 5)

10

Средства за абстракция

Специална форма define

Оценката на всеки define-израз се осъществява в някаква среда (памет). Оценката на най-външно ниво се осъществява в т. нар. глобална среда.

В езика има изрази, които не се оценяват по основното правило. Наричат се *специални форми*. Разгледахме специалните форми:

quote и define (1-ва форма)

Семантика на Scheme = основно правило

+

правила за специалните форми

Дефиниране на процедури

*Специална форма **define***

II. Формат (Дефиниране на съставни процедури)

Пример: Да се дефинира процедура, която връща като резултат квадрата на дадено число.

(define (square x) (* x x))

В процеса на оценяване на обръщението към *define* името **square** се свързва със зададената дефиниция в текущата среда. В случая се казва, че **square** е име на **съставна процедура (дефинирана функция)**.

Средства за абстракция

Синтаксис:

(define (<име> <формални параметри>) <тяло>)

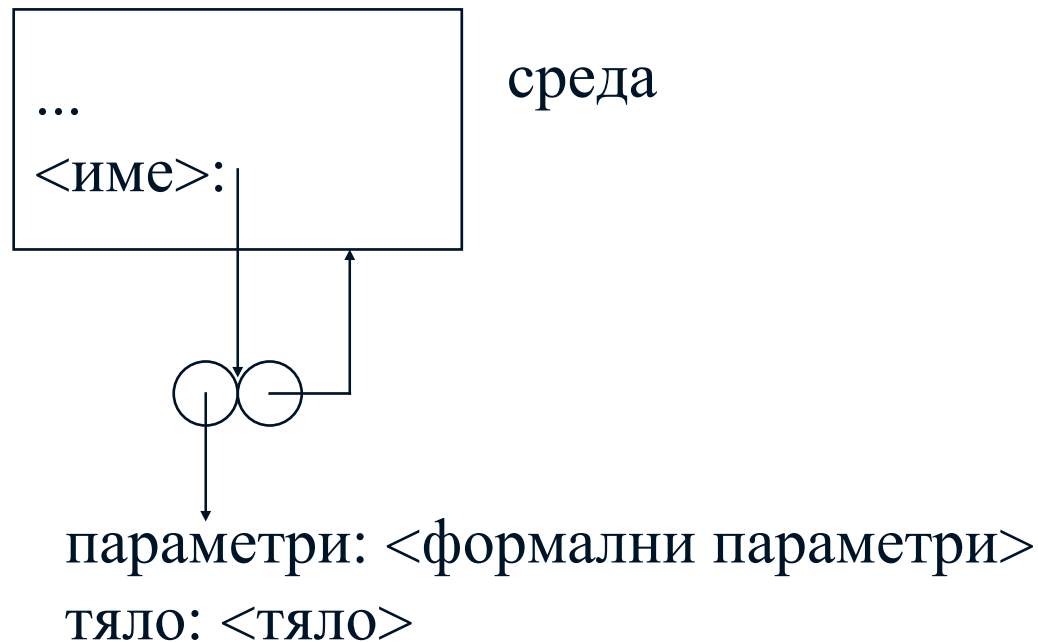
където

- **<име>** е символ, който задава името на дефинираната процедура;
- **<формални параметри>** е символ или редица от различни символи, означаващи имената на формалните параметри. Използват се в тялото на дефиницията за означаване на аргументите на процедурата. Отделят се един от друг с бял символ. Наричат се още **локални имена**.
- **<тяло>** е редица от изрази и определя оценката на процедурното прилагане.

Средства за абстракция

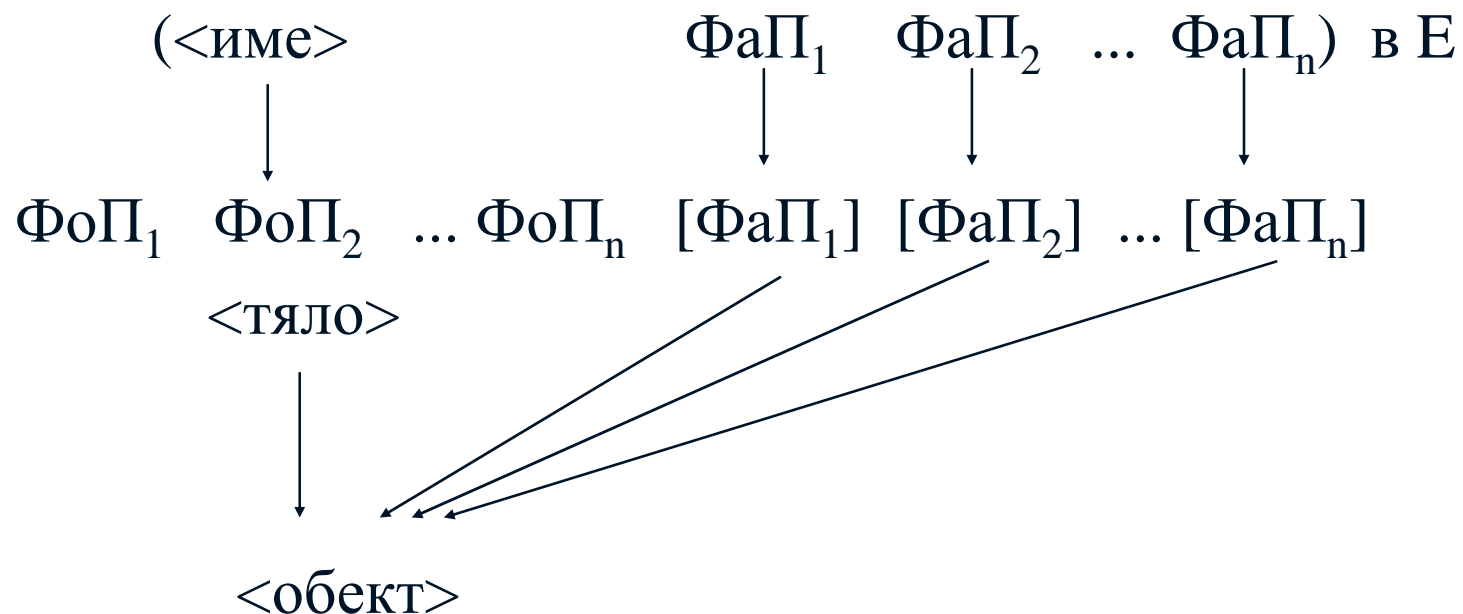
Семантика:

Интерпретаторът чете, оценява и извежда името на дефинираната процедура. Оценяването се състои в следното:



Обръщение към процедура

(define (<име> $\Phi o\Pi_1$ $\Phi o\Pi_2$... $\Phi o\Pi_n$) в средата E
<тяло>)



Използване на дефиниции на процедури

```
(define (sum_of_squares x y)
  (+ (square x) (square y)))
```

```
(define (g a)
  (sum_of_squares (+ a 3) (* a 2)))
```

Оценка на (g 5) в глобалната среда.

Модели на оценяване:

- ✓ Апликативен модел
- ✓ Нормален модел на заместване

Оценка по нормалния модел на заместване на (g 5)

(g 5)

(sum_of_squares (+ 5 3) (* 5 2)) -->

(+ (square (+ 5 3)) (square (* 5 2))) -->

(+ (* (+ 5 3) (+ 5 3)) (* (* 5 2) (* 5 2))) -->

(+ (* 8 8) (* 10 10)) -->

(+ 64 100) -->

164

Да се оцени редицата от изрази:

```
(define (f x) (f x))
```

```
(define (g x y)
```

```
  (if (= x 5) 5 y
```

```
(g 5 (f 3)))
```

а) по апликативния модел – зацикляне

б) по нормалния модел на заместване

```
(g 5 (f 3)) -->
```

```
(if (= 5 5) 5 (f 3)) -->
```

```
5
```

Условни изрази и предикати

Специална форма cond

Синтаксис -1:

(cond (<предикат₁> <редица_от_изрази₁>
(<предикат₂> <редица_от_изрази₂>
...
(<предикат_n> <редица_от_изрази_n>)))

} клаузи

- cond е специален символ (condition);
- <предикат_i> (i = 1, 2, ..., n) е предикат, т.е. израз, чиято оценка е true (#t) или false (#f). В Лисп false са константата #f, специалният символ nil (null) и празният списък. True са всички останали обекти. **В някои реализации на Racket nil (null) и празният списък са true.**
- <редица_от_изрази_i> е произволна редица от изрази. Може и да бъде празна.

Условни изрази и предикати

Специална форма cond

Синтаксис - 2:

(cond [$\langle \text{предикат}_1 \rangle \langle \text{редица_от_изрази}_1 \rangle$]
[$\langle \text{предикат}_2 \rangle \langle \text{редица_от_изрази}_2 \rangle$]
...
[$\langle \text{предикат}_n \rangle \langle \text{редица_от_изрази}_n \rangle$])

} клаузи

Условни изрази и предикати

Частен случай на cond

Синтаксис- 3:

```
(cond (<предикат1> <редица_от_изрази1>)  
      (<предикат2> <редица_от_изрази2>)  
      ...  
      (<предикатn> <редица_от_изразиn>)  
      ( else      <редица_от_изразиn+1>))
```

Забележка: else не е предикат, а специален символ, който може да се използва на мястото на предикат в последната клауза на cond.

Условни изрази и предикати

Частен случай на cond

Синтаксис - 4:

```
(cond [<предикат1> <редица_от_изрази1>]  
      [<предикат2> <редица_от_изрази2>]  
      ...  
      [<предикатn> <редица_от_изразиn>])  
[ else      <редица_от_изразиn+1>])
```

Условни изрази и предикати

Семантика:

В средата E се оценяват последователно предикатите, докато се достигне първият, чиято стойност е `true`. Нека това е $\langle \text{предикат}_i \rangle$. Оценява се $\langle \text{редица_от_изрази}_i \rangle$ в средата E и тя е оценката на `cond`-израза.

Ако няма $\langle \text{предикат}_i \rangle$ с оценка „истина“, то по стандарт оценката на `cond` е неопределена (в много реализации е `()`, `nil/null` или `#f`).

Ако първият срещнат $\langle \text{предикат} \rangle$ с оценка „истина“ няма съответна $\langle \text{редица_от_изрази} \rangle$, оценката на `cond` съвпада с оценката на този $\langle \text{предикат} \rangle$.

Ако `cond` съдържа специалният символ `else` и оценките на всички предходни предикати са били „лъжа“, оценката на $\langle \text{редица_от_изрази}_{n+1} \rangle$ в средата E е оценката на `cond`-израза.

Условни изрази и предикати

Примери:

```
(define (absol x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x)) ))
```

```
(define (absol x)
  (cond [(> x 0) x]
        [(= x 0) 0]
        [(< x 0) (- x)] ))
```

```
(define (absol x)
  (cond ((< x 0) (- x))
        (else x) ))
```

```
(define (absol x)
  (cond [(< x 0) (-x)]
        [else x] ))
```

Това решение е еквивалентно на:

```
(define (absol x)
  (if (< x 0) (- x) x) )
```

Специална форма if

Използва се когато трябва да бъдат анализирани случай и отрицанието му.

Синтаксис:

(if <предикат> <следствие> <алтернатива>)

- <следствие> и <алтернатива> са изрази.

Семантика:

Ако <предикат> е истина, оценява се <следствие>. В противен случай се оценява <алтернатива> и това е оценката на if-израза.

Разликата между специалните форми if и cond е, че и <следствие>, и <алтернатива> в if са точно един израз.

Функции – предикати в Scheme

Примитивни предикати за сравнения

Такива са предикатите: $>$, $=$, $<$, $>=$, $<=$, $<>$.

Те са двуаргументни процедури, които оценяват аргументите си (оценките на аргументите трябва да са числа) и връщат стойност *#t* точно когато оценките на аргументите удовлетворяват съответното отношение.

Примери:

$>(>= 5 2)$

#t

$>(< 8 3)$

#f

Функции – предикати в Scheme

Съставни предикати

Такива са процедурите: *and* (конюнкция), *or* (дизюнкция), *not* (логическо отрицание).

Оператор за конюнкция (and)

Има произволен брой аргументи. Процесът на оценяване на обръщението към *and* е следният. Последователно се оценяват аргументите и ако всички аргументи имат стойност „истина“, като резултат се връща *#t*. Ако в процеса на оценяване на аргументите се срещне аргумент, чиято оценка е „лъжа“, то останалите аргументи не се оценяват и оценката на обръщението към *and* е *#f*.

Функции – предикати в Scheme

Съставни предикати

Такива са процедурите: *and* (конюнкция), *or* (дизюнкция), *not* (логическо отрицание).

Оператор за конюнкция (and)

Пример: Проверка, дали реално число принадлежи на интервала (5, 15).

```
(define (member x)
  (and (> x 5) (< x 15)))
```

Функции – предикати в Scheme

Съставни предикати

Оператор за дизюнкция (or)

Има произволен брой аргументи. Процесът на оценяване на обръщението към *or* е следният. Последователно се оценяват аргументите и ако всички аргументи имат стойност „лъжа“, оценката на обръщението към *or* е *#f*. Ако в процеса на оценяване на аргументите се срещне аргумент, чиято оценка е „истина“, то останалите аргументи не се оценяват и оценката на обръщението към *or* съвпада с оценката на последния оценен аргумент.

Функции – предикати в Scheme

Съставни предикати

Оператор за дизюнкция (or)

Пример:

```
(define (>= x y)
  (or (> x y) (= x y)))
```

Функции – предикати в Scheme

Съставни предикати

Оператор за логическо отрицание (not)

Има един аргумент. Аргументът на *not* се оценява. Оценката на обръщението към *not* е *#t*, ако оценката на аргумента е „лъжа“; ако оценката на аргумента е „истина“, оценката на обръщението към *not* е *#f*.

Пример:

```
(define (>= x y)
  (not (< x y)))
```