Jazyk Scheme: jeho syntax a sémantika

Vilém Vychodil, vilem.vychodil@upol.cz

Syntaxe Scheme

program v jazyku Scheme = konečná posloupnost symbolických výrazů

symbolický výraz (S-výraz / symbolic expression / S-expression) = výraz definovaný použitím následujících bodů:

1. každé `číslo` je symbolický výraz (zapisujeme `12`, `-10`, `2/3`, `12.45`, `4.32e-20`, `-5i`, `2+3i`, a pod.)
2. každý `symbol` je symbolický výraz (zapisujeme `sqrt`, `+`, `quotient`, `even?`, `muj-symbol`, `++?4tt`, a pod.)
3. jsou-li $E_1, E_2, \ldots, E_n$ symbolické výrazy ($n \geq 1$), pak `(`$E_1 \sqcup E_2 \sqcup \cdots \sqcup E_n$`)` je symbolický výraz zvaný `seznam`, $E_1, E_2, \ldots, E_n$ se nazývají prvky seznamu `(`$E_1 \sqcup E_2 \sqcup \cdots \sqcup E_n$`)`.

Příklady:

- následující **jsou** symbolické výrazy: `3.141459`, `10.2.45`, `1+`, `list?`, `vector->list`, `(= 1 2)`, `(* 2 (sqrt (/ 2 3)))`, `(1 + 2)`, `(2+ (3 + 3))`, `((10) (20))`, `(1 (ahoj (3)))`, a pod.
- následující **nejsou** symbolické výrazy: `)`, `(+ 1 2))`, `(10 (20 ahoj)`, a pod.

Sémantika Scheme (interpretace S-výrazů)

Interprety nepracují s S-výrazy přímo, to jest ve formě sekvence znaků, ale převádějí je vždy do své interní reprezentace, která je pro ně efektivní. Například malé číslo `-123` je reprezentováno datovým typem integer, pro který je základní aritmetika implementována rovnou na procesoru. Stejně tak například seznam `(+ 1 (* 2 3))` je reprezentován efektivní strukturou (lineárním dynamickým spojovým seznamem). S-výraz převedený do interní reprezentace budeme nazývat **element jazyka**. Kromě interních reprezentací S-výrazů existují ještě další elementy jazyka. Jedněmi z nich jsou například **procedury** (například aritmetické procedury pro sčítání, odčítání, mocnění čísel, procedury pro vstupně / výstupní operace a podobně). Množina všech elementů jazyka spolu s pravidly jejich vzájemného vyhodnocování tvoří základní model **abstraktního interpretu jazyka Scheme**.

Interpretace programu probíhá v cyklu **REPL** (**R**ead, **E**val, **P**rint, **L**oop):

Read: Načtení prvního vstupního S-výrazu E a převedení do vnitřní (efektivní) reprezentace, kterou používá interpret, hovoříme o **interní reprezentaci S-výrazu**. Interní reprezentaci S-výrazu E budeme označovat $\lfloor E \rfloor$. Interpret dále provádí všechny úkony nad interní reprezentací S-výrazů. Během fáze „Read“ jsou odhalovány **syntaktické chyby** (neformálně řečeno: „chyby v zápisu S-výrazů“).

Eval: Vstupem je interní reprezentace $\lfloor E \rfloor$ vstupního S-výrazu E. Výstupem je **element jazyka** F, který vznikne **vyhodnocením** $\lfloor E \rfloor$ (viz dále). Pokud je F vyhodnocením $\lfloor E \rfloor$, tento fakt zapisujeme $F = \mathrm{Eval}[\lfloor E \rfloor]$.

Print: Provede se převod elementu F, vzniklého v předchozím kroku, na jeho **externí reprezentaci**, což je pro člověka čitelný S-výraz. Externí reprezentaci F značíme $\lceil F \rceil$. Vzniklý S-výraz $\lceil F \rceil$ je vypsán.

Loop: Odebere se S-výraz E ze vstupu a pokud je vstup neprázdný, pokračujeme krokem „Read“.

Procedury „Read“, „Print“ a „Loop“ jsou více méně triviální. Nutné je přesně specifikovat „Eval“. Bez znalosti „Eval“ neznáme interpretaci S-výrazů a tím pádem bychom neznali sémantiku (význam) jazyka. Potřebné pojmy:

výpočetní proces je generován programem pomocí **postupné aplikace procedur** (**funkcí**)

procedury = (speciální) elementy jazyka Scheme (v naší terminologii je „procedura“ totéž co „funkce“)

aplikace procedury (někdy též **aktivace procedury**) = provedení dané procedury nad danými **argumenty**, výsledkem aplikace procedury je buď „CHYBA: Proceduru nelze aplikovat s danými argumenty.“, nebo **výsledný element** (**výsledná hodnota**)

argumenty = elementy jazyka, které jsou při aktivaci předány proceduře (procedura používá argumenty ke stanovení výsledného elementu)

Příklad: Aplikací procedury „sčítání čísel“ na argumenty `1`, `10` a `-30` je výsledná hodnota `-19`. Aplikací procedury „zjisti zda-li je dané číslo sudé“ na argument `12.7` je „CHYBA: Vstupní argument musí být celé číslo.“.

Poznámka: nyní již známe dva typy elementů jazyka: *interní reprezentace S-výrazů* (viz výše) a *procedury.*

Při postupné aplikaci procedur (která je prováděna v části „Eval“ cyklu REPL) hrají důležitou roli symboly vyskytující se v S-výrazech, které vyhodnocujeme. Symboly se využívají (mimo jiné) k **pojmenování procedur.** Například intuitivně očekáváme, že symbol + uvedený v programu bude zastupovat „proceduru pro sčítání“. Upozorněme však na fakt, že symbol je něco jiného než procedura samotná: symbol je S-výraz, kdežto procedura je element jazyka – dvě naprosto odlišné věci.

Při interpretaci programu máme dány vazby symbolů na procedury, respektive obecněji, vazby symbolů na elementy. Tyto vazby jsou definovány v tabulce zvané prostředí. **Prostředí** je tabulka tvaru:

symbol	*element*
E_1	F_1
E_2	F_2
$\vdots$	$\vdots$
E_k	F_k
$\vdots$	$\vdots$

kde $E_1, E_2, \ldots$ jsou symboly a $F_1, F_2, \ldots$ jsou elementy.

Pokud je symbol E uveden v tabulce v levém sloupci, pak hodnotě v pravém sloupci na témže řádku říkáme **aktuální vazba symbolu E v prostředí**. Na počátku interpretace jsou v prostředí (**počátečním prostředí**) dány **počáteční vazby**, například aktuální vazbou symbolu + je procedura pro sčítání čísel, aktuální vazbou symbolu `sqrt` je procedura druhé odmocniny, a pod. Pokud symbol E není uveden v levém sloupci, pak říkáme, že **E nemá vazbu v prostředí**. Nyní můžeme popsat vyhodnocování výrazů (to jest část „Eval“ cyklu REPL).

Mějme S-výraz E. **Výsledek vyhodnocení (interní reprezentace) S-výrazu E**, značeno $\mathrm{Eval}[\lfloor E \rfloor]$, je definován:

(A) Pokud je E `číslo`, pak nastavíme hodnotu $\mathrm{Eval}[\lfloor E \rfloor]$ na $\lfloor E \rfloor$, což zapisujeme $\mathrm{Eval}[\lfloor E \rfloor] := \lfloor E \rfloor$.
(Slovně: každé číslo se vyhodnotí na sebe samo.)

(B) Pokud je E `symbol`, mohou nastat dvě situace:

(B.1) Pokud E *má aktuální vazbu* F, pak nastavíme $\mathrm{Eval}[\lfloor E \rfloor] := F$.
(Slovně: pokud má symbol aktuální vazbu, pak je vyhodnocením symbolu jeho aktuální vazba.)

(B.e) Pokud E *nemá aktuální vazbu,* skončíme vyhodnocování hlášením „CHYBA: Symbol E nemá vazbu.“.

(C) Pokud je E `seznam` tvaru $(E_1 \sqcup E_2 \sqcup \cdots \sqcup E_n)$, pak nejprve provedeme vyhodnocení prvního prvku E_1 a výslednou hodnotu označíme F_1, což budeme zapisovat $F_1 := \mathrm{Eval}[\lfloor E_1 \rfloor]$. Mohou nastat dvě situace:

(C.1) Pokud F_1 (element vzniklý vyhodnocením interní reprezentace E_1) je **procedura**, pak se v nespecifikovaném pořadí vyhodnotí zbylé prvky $E_2, \ldots, E_n$ seznamu E, výsledky jejich vyhodnocení označíme $F_2, \ldots, F_n$. Formálně:

$$\begin{aligned} F_2 &:= \mathrm{Eval}[\lfloor E_2 \rfloor], \\ F_3 &:= \mathrm{Eval}[\lfloor E_3 \rfloor], \\ &\vdots \quad\quad \vdots \\ F_n &:= \mathrm{Eval}[\lfloor E_n \rfloor]. \end{aligned}$$

Výsledkem vyhodnocení $\lfloor E \rfloor$ je potom element F vzniklý **aplikací procedury F_1 na argumenty $F_2, \ldots, F_n$**, zapisujeme $\mathrm{Eval}[\lfloor E \rfloor] := \mathrm{Apply}[F_1, F_2, \ldots, F_n]$.
(Slovně: vyhodnocení seznamu probíhá tak, že je nejprve vyhodnocen jeho první prvek. V případě, že se první prvek vyhodnotí na proceduru, se v nespecifikovaném pořadí vyhodnotí ostatní prvky seznamu; potom je procedura aplikována na argumenty, kterými jsou právě výsledky vyhodnocení zbylých prvků seznamu – tedy všech prvků kromě prvního.)

(C.e) Pokud F_1 není **procedura**, skončíme vyhodnocování hlášením
„CHYBA: Nelze provést aplikaci: první prvek seznamu E se nevyhodnotil na proceduru.“.

V dalším textu budeme někdy kvůli přehlednosti ztotožňovat interní reprezentaci S-výrazů se samotnými S-výrazy. Pořád je ale nutné mít na paměti, že jde o dvě různé věci.

Příklady:

1. `(+ 1 2)` se vyhodnotí na `3`. Podrobněji: `(+ 1 2)` je seznam, tedy vyhodnocování postupuje dle bodu (C). Vyhodnotíme první prvek seznamu, to jest stanovíme $\mathrm{Eval}[+]$. Jelikož je + symbol, postupujeme dle bodu (B). Na symbol + je navázána procedura pro sčítání. V bodu (C) tedy postupujeme po větvi (C.1). Nyní v nespecifikovaném pořadí vyhodnotíme zbylé argumenty `1` a `2`. Oba argumenty jsou čísla, takže $\mathrm{Eval}[1] = 1$ a $\mathrm{Eval}[2] = 2$. Nyní aplikujeme proceduru sčítání na argumenty `1` a `2` a dostaneme výslednou hodnotu `3`. Výsledkem vyhodnocení `(+ 1 2)` je `3`.

2. `(+ (* 3 (+ 1 1)) 20)` se vyhodnotí na 26.
3. Vyhodnocení `(10 + 20)` končí chybou, která nastane v kroku (C.e).
4. Vyhodnocení `((+ 1 2) (+ 1 3))` končí chybou, která nastane v kroku (C.e).
5. Vyhodnocení `((* 2 3 4))` končí chybou, která nastane v kroku (C.e).
6. Vyhodnocení `(+ 2 (odmocni 10))` končí chybou, která nastane v kroku (B.e).
7. Vyhodnocení `(* (+ 1 2) (sqrt 10 20))` končí chybou při pokusu o aplikaci procedury „vypočti druhou odmocninu“: „CHYBA: Nepřípustný počet argumentů.“

Rozšíření Scheme o speciální formy

speciální forma = element jazyka

Poznámka: nyní již známe tři typy elementů jazyka: *interní reprezentace S-výrazů*, *procedury* a *speciální formy.*

Speciální formy mají podobný účel jako procedury – jejich aplikacemi lze generovat, případně modifikovat, výpočetní proces. Zcela zásadně se však od procedur liší v tom, jak probíhá jejich aplikace během vyhodnocování S-výrazů. Procedury jsou aplikovány s argumenty, které jsou předem získány vyhodnocením (tedy procedura přísně vzato nemůže zjistit, vyhodnocením jakých symbolických výrazů dané argumenty vznikly). Neformálně řečeno: speciální formy jsou aplikovány s argumenty jimiž jsou S-výrazy (respektive jejich interní reprezentace) v jejich nevyhodnocené podobě a každá speciální forma si sama určuje jaké argumenty a v jakém pořadí (zda-li vůbec) bude vyhodnocovat. Přesnou aplikaci speciálních forem zavedeme nyní.

Pro použití speciálních forem rozšíříme bod (C) části „Eval“ cyklu REPL následovně:

(C) Pokud je E `seznam` tvaru $(E_1 \sqcup E_2 \sqcup \cdots \sqcup E_n)$, pak nejprve provedeme vyhodnocení prvního prvku E_1 a výslednou hodnotu označíme F_1, to jest $F_1 := \mathrm{Eval}[\lfloor E_1 \rfloor]$. Mohou nastat tři situace:

- (C.1) Pokud F_1 je **procedura**, pak se postup shoduje s (C.1) popsaným v předchozí sekci.
- (C.2) Pokud F_1 je **speciální forma**, pak výsledkem vyhodnocení $\lfloor E \rfloor$ je element F vzniklý **aplikací speciální formy F_1 na argumenty $\lfloor E_2 \rfloor, \ldots, \lfloor E_n \rfloor$**, zapisujeme $\mathrm{Eval}[\lfloor E \rfloor] := \mathrm{Apply}[F_1, \lfloor E_2 \rfloor, \ldots, \lfloor E_n \rfloor]$.
  (Slovně: narozdíl od procedur, speciální formy jsou aplikovány na argumenty jimiž jsou interní reprezentace prvků seznamu E v jejich **nevyhodnocené podobě**).
- (C.e) Pokud F_1 není **procedura ani speciální forma**, skončíme vyhodnocování hlášením „CHYBA: Nelze provést aplikaci: první prvek seznamu E se nevyhodnotil na proceduru ani spec. formu.“.

V jazyku Scheme budeme mít vždy snahu mít co možné nejméně (nezbytné minimum) speciálních forem, protože zavádění speciálních forem do jazyka a jejich využívání programátorem je potenciálním zdrojem chyb. Důvod je zřejmý: jelikož si každá speciální forma určuje vyhodnocování svých argumentů, na uživatele formy (to jest programátora) to narozdíl od procedur klade další požadavek (pamatovat si, jak se argumenty zpracovávají). Druhý úhel pohledu je více méně epistemický – zařazováním nadbytečných forem do jazyka se zvyšuje jeho celková složitost.

Pojmenování: vytváření abstrakcí pojmenováním hodnot

define = speciální forma, která je v počátečním prostředí navázána na symbol `define`

Poznámka: Pokud budeme dále hovořit o speciálních formách, budeme je pro jednoduchost ztotožňovat se symboly, na které jsou navázány v počátečním prostředí. Opět je ale nutné si uvědomit, že speciální forma je element jazyka, tedy něco jiného než symbol (S-výraz).

Speciální forma **define** se aktivuje se dvěma argumenty ve tvaru: `(define` ⟨*jméno*⟩ ⟨*výraz*⟩`)`, přitom forma nejprve ověří zda-li je ⟨*jméno*⟩ symbol. Pokud tomu tak není, aktivace formy je ukončena hlášením „CHYBA: První výraz musí být symbol.“ V opačném případě je vyhodnocen ⟨*výraz*⟩, to jest provede se $F := \mathrm{Eval}[\lfloor$⟨*výraz*⟩$\rfloor]$. Dále je v prostředí vytvořena nová vazba symbolu ⟨*jméno*⟩ na element F. Pokud již symbol ⟨*jméno*⟩ měl vazbu, tato původní vazba je nahrazena elementem F. Speciální forma **define** má při své aktivaci **vedlejší efekt**: modifikuje prostředí. Výsledná hodnota `(define` ⟨*jméno*⟩ ⟨*výraz*⟩`)` je **nedefinovaná**.

nedefinovaná hodnota: specifikace jazyka Scheme R^{5}RS neupřesňuje, co je pod pojmem „nedefinovaná hodnota“ myšleno. Jednotlivé interprety tuto otázku řeší různě. Náš (abstraktní) interpret Scheme ji může vyřešit třeba tak, že **nedefinovaná hodnota** bude *speciální element jazyka,* který se bude vyhodnocovat na sebe sama (analogicky jako čísla)

a jehož externí reprezentací bude „prázdný řetězec“ (takto je nedefinovaná hodnota řešena i v interpretu DrScheme a mnoha dalších).

Příklady:

1. Po vyhodnocení `(define ahoj (* 2 3))` se na symbol `ahoj` naváže hodnota `6`.
2. Po vyhodnocení `(define blah +)` se na symbol `blah` naváže hodnota, která byla navázaná na symbolu `+`.
3. Po postupném vyhodnocení `(define tmp +)`, `(define + *)`, a `(define * tmp)` se zamění vazby na `+` a `*`.
4. Vyhodnocení `(define 10 (* 2 3))` končí chybou (`10` není symbol).

Vytváření abstrakcí pomocí procedur

primitivní procedury = procedury, které jsou na počátku vyhodnocování (po spuštění interpretu) navázány na symboly `+`, `sqrt`, `modulo`, ...

uživatelsky definované procedury = procedury, které lze dynamicky vytvářet během vyhodnocování programů

uživatelsky definované procedury **vznikají vyhodnocením lambda-výrazů (λ-výrazů)**

lambda-výraz je S-výraz ve tvaru `(lambda (`$\triangleleft p_1 \triangleright \cdots \triangleleft p_n \triangleright$`) `$\triangleleft$`tělo`$\triangleright$`)`, kde $\triangleleft p_1 \triangleright, \ldots, \triangleleft p_n \triangleright$ $(n \geq 0)$ jsou vzájemně různé symboly a $\triangleleft$`tělo`$\triangleright$ je S-výraz; $\triangleleft p_1 \triangleright, \ldots, \triangleleft p_n \triangleright$ se nazývají **formální argumenty** (**parametry**).

Poznámka: Účelem formálních argumentů je „pojmenovat hodnoty“ se kterými bude procedura aplikována (proceduru můžeme aplikovat s různými argumenty, proto je potřeba je při definici procedury zastoupit symboly, aby byly pokryty „všechny možnosti aplikace procedury“). Tělo procedury představuje vlastní *předpis procedury* – neformálně řečeno, tělo vyjadřuje „co bude procedura s danými argumenty provádět“.

Vazba na pojem prostředí: Při rozšíření interpretu o uživatelsky definovatelné procedury je třeba rozšířit pojem *prostředí*. Už si nevystačíme pouze s jedním prostředím jak tomu bylo doposud, ale během vyhodnocování programu bude obecně existovat celá řada prostředí, které budou mít mezi sebou vazby.

prostředí $\mathcal{P}$ je tabulka vazeb symbolů (jako doposud); každé prostředí (vyjma *globálního prostředí*, viz dále) má navíc ukazatel na svého **předka** (předek je opět prostředí).

globální (počáteční) prostředí, označujeme $\mathcal{P}_G$, je prostředí ve kterém jsou nastaveny počáteční vazby symbolů a ve kterém jsme doposud uvažovali vyhodnocování. Globální prostředí *nemá předka*. Jinými slovy, globální prostředí je v hierarchii prostředí úplně na vrcholu.

Máme-li k dispozici víc prostředí, musíme přesně specifikovat vyhodnocování S-výrazů v daném prostředí. Proto rozšíříme „Eval“ následovně: místo $\mathrm{Eval}[\lfloor E \rfloor]$ budeme psát $\mathrm{Eval}[\lfloor E \rfloor, \mathcal{P}]$, kde $\mathcal{P}$ je prostředí; $\mathrm{Eval}[\lfloor E \rfloor, \mathcal{P}]$ označuje **výsledek vyhodnocení (interní reprezentace) S-výrazu E v prostředí $\mathcal{P}$**, který definujeme:

(A) Pokud je E `číslo`, pak nastavíme $\mathrm{Eval}[\lfloor E \rfloor, \mathcal{P}] := \lfloor E \rfloor$.
(Slovně: každé číslo se v libovolném prostředí vyhodnotí na sebe samo.)

(B) Pokud je E `symbol`, mohou nastat dvě situace:

(B.1) Pokud E *má vazbu* F *v tabulce prostředí* $\mathcal{P}$, pak nastavíme $\mathrm{Eval}[\lfloor E \rfloor, \mathcal{P}] := F$.
(Slovně: pokud má symbol vazbu v $\mathcal{P}$, pak je vyhodnocením symbolu v $\mathcal{P}$ tato vazba.)

(B.2) Pokud E *nemá vazbu v* $\mathcal{P}$ *a pokud má* $\mathcal{P}$ *předchůdce* $\mathcal{P}'$, pak nastavíme $\mathrm{Eval}[\lfloor E \rfloor, \mathcal{P}] := \mathrm{Eval}[\lfloor E \rfloor, \mathcal{P}']$.
(Slovně: pokud není vazba symbolu nalezena v tabulce prostředí $\mathcal{P}$, pak se vazbu pokusíme hledat v tabulce nadřazeného prostředí.)

(B.e) Pokud E *nemá v* $\mathcal{P}$ *vazbu a pokud* $\mathcal{P}$ *nemá předchůdce* (to jest pokud $\mathcal{P} = \mathcal{P}_G$), pak skončíme vyhodnocování hlášením „CHYBA: Symbol E nemá vazbu.“.

(C) Pokud je E `seznam` tvaru $(E_1 \sqcup E_2 \sqcup \cdots \sqcup E_n)$, pak nejprve provedeme vyhodnocení prvního prvku E_1 v prostředí $\mathcal{P}$ a výslednou hodnotu označíme F_1, to jest $F_1 := \mathrm{Eval}[\lfloor E_1 \rfloor, \mathcal{P}]$. Mohou nastat tři situace:

(C.1) Pokud F_1 (element vzniklý vyhodnocením interní reprezentace E_1) je **procedura**, pak se v nespecifikovaném pořadí vyhodnotí zbylé prvky $E_2, \ldots, E_n$ seznamu E v prostředí $\mathcal{P}$, výsledky jejich vyhodnocení označíme $F_2, \ldots, F_n$. Formálně:

$$\begin{aligned} F_2 &:= \mathrm{Eval}[\lfloor E_2 \rfloor, \mathcal{P}], \\ F_3 &:= \mathrm{Eval}[\lfloor E_3 \rfloor, \mathcal{P}], \\ &\vdots \quad\quad \vdots \\ F_n &:= \mathrm{Eval}[\lfloor E_n \rfloor, \mathcal{P}]. \end{aligned}$$

Výsledkem vyhodnocení $\lfloor E \rfloor$ v $\mathcal{P}$ je potom element F vzniklý **aplikací procedury F_1 na argumenty $F_2, \ldots, F_n$**, zapisujeme $\mathrm{Eval}[\lfloor E \rfloor, \mathcal{P}] := \mathrm{Apply}[F_1, F_2, \ldots, F_n]$.
(Slovně: vyhodnocení seznamu v daném prostředí probíhá tak, že je nejprve vyhodnocen jeho první prvek v tomto prostředí. V případě, že se první prvek vyhodnotí na proceduru, se v nespecifikovaném pořadí vyhodnotí ostatní prvky seznamu (opět v daném prostředí); potom je procedura aplikována na argumenty, kterými jsou právě výsledky vyhodnocení zbylých prvků seznamu – tedy všech prvků kromě prvního.)

(C.2) Pokud F_1 je **speciální forma**, pak výsledkem vyhodnocení $\lfloor E \rfloor$ v $\mathcal{P}$ je element F vzniklý **aplikací speciální formy F_1 na argumenty $\lfloor E_2 \rfloor, \ldots, \lfloor E_n \rfloor$**, zapisujeme $\mathrm{Eval}[\lfloor E \rfloor, \mathcal{P}] := \mathrm{Apply}[F_1, \lfloor E_2 \rfloor, \ldots, \lfloor E_n \rfloor]$.
(Slovně: narozdíl od procedur, speciální formy jsou aplikovány na argumenty jimiž jsou interní reprezentace prvků seznamu E v jejich **nevyhodnocené podobě**).

(C.e) Pokud F_1 není **procedura ani speciální forma**, skončíme vyhodnocování hlášením
„CHYBA: Nelze provést aplikaci: první prvek seznamu E se nevyhodnotil na proceduru ani spec. formu.“.

V cyklu REPL probíhá vyhodnocování ve fázi „Eval“ v prostředí $\mathcal{P}_G$ (cyklus REPL probíhá v globálním prostředí).

Poznámka: Všimněte si, že předchozí popis „Eval“ jsme rozšířili v podstatě jen velmi nepatrně – oproti „Eval“ pracujícím se speciálními formami jsme pouze přidali bod (B.2) zajišťující možnost hledat vazby v nadřazených prostředích; u všech ostatních výskytů „Eval“ provádíme vyhodnocení v témže prostředí $\mathcal{P}$. Dále si všimněte, že aplikace (primitivní) procedury F_1 na argumenty $F_2, \ldots, F_n$, což značíme $\mathrm{Apply}[F_1, F_2, \ldots, F_n]$, není závislá na prostředí.

Doposud jsme nespecifikovali *aplikaci uživatelsky definovatelných procedur*. To jest, neřekli jsme co je výsledkem provedení $\mathrm{Apply}[F_1, F_2, \ldots, F_n]$ pokud je F_1 procedura vzniklá vyhodnocením lambda-výrazu. Na to se zaměříme nyní.

Vyhodnocením lambda-výrazu `(lambda (`$\triangleleft p_1 \triangleright \cdots \triangleleft p_n \triangleright$`) `$\triangleleft$*tělo*$\triangleright$`)` **v prostředí $\mathcal{P}$ vzniká procedura**, což je trojice

$$\langle (\triangleleft p_1 \triangleright \cdots \triangleleft p_n \triangleright), \triangleleft \mathit{tělo} \triangleright, \mathcal{P} \rangle.$$

Trojice $\langle (\triangleleft p_1 \triangleright \cdots \triangleleft p_n \triangleright), \triangleleft \mathit{tělo} \triangleright, \mathcal{P} \rangle$ je tedy element jazyka Scheme reprezentující proceduru, který se skládá ze **seznamu formálních argumentů**, **těla**, a **prostředí vzniku procedury**. Prostředí vzniku procedury je prostředí, ve kterém byl vyhodnocen daný lambda-výraz. Zápis procedur budeme dále zkracovat na $F = \langle V, E, \mathcal{P} \rangle$, kde V je seznam formálních argumentů a E je tělo.

Poznámka: Z předchozího je zřejmé, že každá procedura vzniklá vyhodnocením lambda-výrazu si v sobě nese informaci o prostředí, ve kterém vznikla. Tato informace bude dále použita při aplikaci procedury, viz dále.

Mějme dánu uživatelský definovanou proceduru $F = \langle V, E, \mathcal{P} \rangle$ a argumenty $F_1, \ldots, F_n$. **Aplikaci procedury F na argumenty $F_1, \ldots, F_n$**, značenou $\mathrm{Apply}[F, F_1, \ldots, F_n]$, definujeme takto:

1. Předpokládejme, že V je m-prvkový seznam symbolů tvaru `(x1 ⋯ xm)`. Pokud se m (počet formálních argumentů) neshoduje s n (počet argumentů se kterými chceme proceduru aplikovat), pak skonči hlášením „CHYBA: Chybný počet argumentů, předáno n, očekáváno m“. V opačném případě přejdi na další bod.

2. Vytvoří se **nové prázdné prostředí $\mathcal{P}_l$**, které nazýváme **lokální prostředí procedury**
(tabulka prostředí $\mathcal{P}_l$ v tomto okamžiku *neobsahuje žádné vazby*).

3. Nastavíme předka $\mathcal{P}_l$ na $\mathcal{P}$ (předkem nového prostředí je *prostředí vzniku procedury* F).

4. V prostředí $\mathcal{P}_l$ se na symboly `x1`, …, `xn` naváží argumenty $F_1, \ldots, F_n$ (což jsou elementy jazyka).

5. Položíme $\mathrm{Apply}[F, F_1, \ldots, F_n] := \mathrm{Eval}[E, \mathcal{P}_l]$.
(Slovně: výsledkem aplikace procedury je vyhodnocení těla E v lokálním prostředí $\mathcal{P}_l$).

Výše popsaný princip aplikace procedur odpovídá **lexikálnímu rozsahu platnosti symbolů** – pokud není vazba daného symbolu nalezena v lokálním prostředí procedury, pak je hledána v prostředí **vzniku procedury** – to jest v **lexikálně nadřazeném prostředí** (prostředí **lexikálního předka**). Z tohoto důvodu je potřeba uchovat v každé proceduře informaci o prostředí ve kterém vznikla.

Druhým typem rozsahu platnosti je **dynamický rozsah platnosti,** který je v současnosti prakticky nepoužívaný. Jedinou odlišnosti od předchozího modelu je, že pokud není vazba symbolu nalezena v lokálním prostředí procedury, pak je hledána v **prostředí odkud byla procedura aktivována (aplikována)**, to jest v **dynamicky nadřazeném prostředí.**

Pokud bychom chtěli v našem abstraktním interpretu zavést dynamický rozsah platnosti místo lexikálního, stačilo by pouze uvažovat procedury jako dvojice $F = \langle V, E \rangle$ ($\mathcal{P}$ už si není potřeba pamatovat). Aplikaci procedury bychom museli uvažovat vzhledem k prostředí $\mathcal{P}_A$, ve kterém ji provádíme, a změnili bychom bod 3. aplikace následovně:

3. Nastavíme předka $\mathcal{P}_l$ na $\mathcal{P}_A$ (předkem nového prostředí je *prostředí aplikace procedury* F).

Dále bychom upravili „Eval“ v bodech (C.1) a (C.2) tak, aby se při každé aplikaci předávala informace o prostředí $\mathcal{P}_A$, ve kterém aplikaci provádíme (to jest v podobném duchu jako jsme rozšířili samotný Eval o dodatečný argument reprezentující prostředí, rozšíříme nyní Apply o další argument reprezentující prostředí – v tomto případě prostředí aktivace procedury).

Poznámka: Implementace dynamického rozsahu platnosti je výrazně jednodušší než implementace lexikálního rozsahu platnosti. Proto byl dynamický rozsah platnosti populární v rané fázi vývoje interpretů a překladačů programovacích jazyků. Programování s dynamickým rozsahem platnosti však vede k častému vzniku chyb (programátor se musí neustále zamýšlet nad tím, odkud bude daná procedura volána a jaké je potřeba mít v daném prostředí vazby symbolů) a proto jej v současnosti nevyužívá skoro žádný programovací jazyk (jedním z mála jazyků s dynamickým rozsahem platnosti, který je v praxi používán je FoxPro).

Příklad: Výsledkem vyhodnocení následujícího programu v interpretu jazyka Scheme

```
(define x 100)

(define curry+
  (lambda (x)
    (lambda (y)
      (+ x y))))

((curry+ 10) 20)
```

je číslo 30. Kdybychom používali dynamický rozsah platnosti, pak by výsledkem vyhodnocení bylo číslo 120, protože procedura vytvořená vyhodnocením lambda-výrazu `(lambda (y) (+ x y))` je aplikována z globálního prostředí (s argumentem 20), kde má symbol `x` vazbu 100.