

# Dokumentace jazyka Výplod

Ondřej Veselý

10. června 2010

## 1 Úvod

Programovací jazyk Výplod je vyšší procedurální slabě typovaný strukturovaný jazyk. Tato dokumentace obsahuje jak jeho částečnou specifikaci, tak dokumentaci k jeho interpretu.

V části Sémantické akce se z větší částí zabývám popisem implementace.

### Ukázka

```
echon 'Hello world';
```

Interpret je napsaný v jazyce C++ za použití standardních knihoven. Implementuje mj. tři základní třídy:

- lexikální analyzátor
- syntaktický analyzátor
- sémantické akce.

## 2 Dokumentace jazyka

### 2.1 Lexikální analyzátor

Jazyk interpretu vypadá následovně:

$$\Sigma = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, ., *, \&, /, \backslash, |, !, (, ), <, >, =, :, ,, \{, \}, , TAB, \leftarrow, ;, ;, \_ , \$, \wedge, \#, , , , ? , \sim, ;, ;\}$$

Lexikální analyzátor znaky zdrojového kódu a generuje následující kategorie lexikálních symbolů (tokenů):

```
identifikator, cislo, retezec, datovotyp (int, float, string),  
znak_pirazeni, logand, logor, plus, minus, krat, deleno,  
vykricnik, operator_porovnani (<=, >=, <, >, ==), klicove_slovo  
(sub, read, reads, echo, echon, while, do, if, then, else,  
endif, open, close, for), zacatek_bloku, konec_bloku, lzavorka,  
pzavorka, strednik, empty
```

V relevantních případech je v závorce uveden úplný výčet možného obsahu tokenu. Oddělovače tokenů jsou bílé znaky.

Gramatika je podrobněji popsána v příloze.

## 2.2 Návrh gramatiky

Při návrhu jazyka jsem formálně postupoval pouze při návrhu gramatiky výrazů. Jedná se totiž o nejsložitější teoretickou část bez které by se implementace syntaktického analyzátoru výrazů neobešla.

Původní zjednodušená gramatika výrazů, ze které jsem při návrhu jazyka vycházel vypadala takto:

$$E \rightarrow E \pm T \mid T \mid E \_ T$$

$$F \rightarrow ( E ) \mid \underline{x}$$

$$T \rightarrow \bar{T} * \bar{F} \mid F \mid T / F$$

Po regularizaci a transformaci do Greibachovy normální formy:

$$E_{INC} \rightarrow E \text{ (accept)}$$

$$E' \rightarrow \epsilon \$ \mid \pm T E' \$ \mid \_ T E' \$$$

$$E \rightarrow ( E ) T' E' \$ \mid \underline{x} T' E' \$$$

$$F \rightarrow ( E ) \$ \mid \underline{x} \$$$

$$T' \rightarrow \epsilon \$ \mid * F T' \$ \mid / F T' \$$$

$$T \rightarrow ( E ) T' \$ \mid \underline{x} T' \$$$

A přechodová tabulka inkrementálního LL syntaktického analyzátoru:

	(	)	*	+	-	/
$\rightarrow E_{inc}$	E <i>accept</i>					
E'		$\epsilon \text{ pop}$		+ T E' <i>pop</i>	- T E' <i>pop</i>	
E	( E ) T' E' <i>pop</i>					
F	( E ) <i>pop</i>					
T'		$\epsilon \text{ pop}$	* F T' <i>pop</i>	$\epsilon \text{ pop}$	$\epsilon \text{ pop}$	/ F T' <i>pop</i>
T	( E ) T' <i>pop</i>					

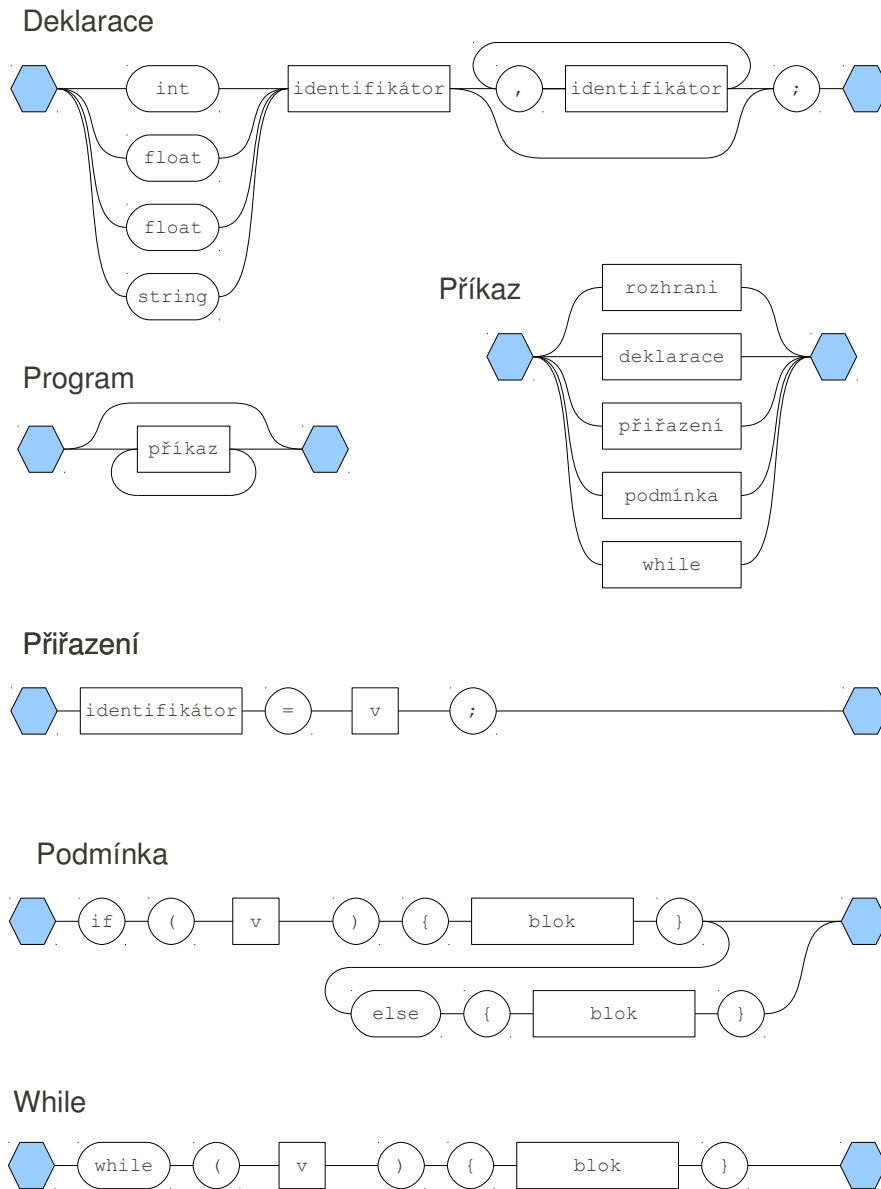
	x	$E_{inc}$	E'	E	F	T'	T	$\epsilon$
$\rightarrow E_{inc}$	E <i>accept</i>			E <i>accept</i>				
E'								$\epsilon \text{ pop}$
E	x T' E' <i>pop</i>							
F	x <i>pop</i>							
T'			$\epsilon \text{ pop}$					$\epsilon \text{ pop}$
T	x T' <i>pop</i>							

Výše uvedená tabulka popisuje chování syntaktického analyzátoru výrazů (pro tokeny *identifikator*, *cislo* a *retezec* je použitý symbol x). Do původního návrhu jsem později zanášel ještě drobné změny. Jisté změny si například vyžádal problém, které jsem nazval *aritní schizma*. Ten spočívá v dvojjakém chování operátorů + a -. Zatím co při výpočtu  $a - b$  se operátor chová jako binární,

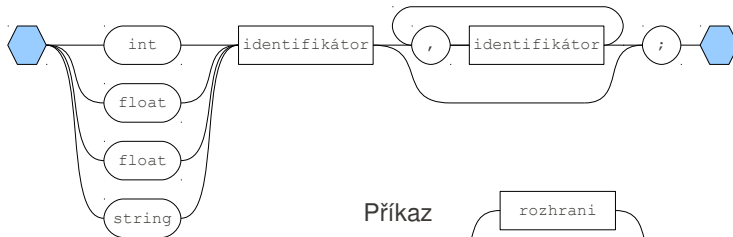
ve výrazu  $a + -(b + 1)$  vystupuje před závorkou jako unární. Pro ošetření většiny těchto potenciálně chybových stavů jsem implementoval funkci, která tento stav rozpozná a k výrazu se chová jako by byla před „schozifrenním“ operátorem nula. Tj., výraz je vypočten jako  $a + 0 - (b + 1)$ .

Zbytek gramatiky je popsán syntaktickými diagramy.

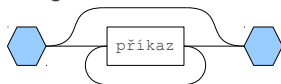
## 2.3 Syntaktické diagramy



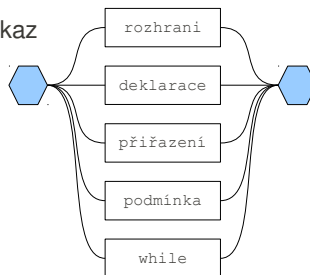
### Deklarace



### Program



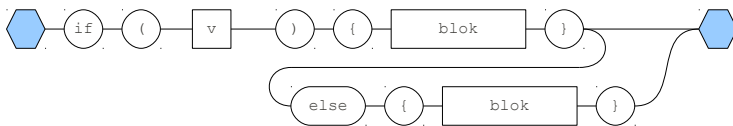
### Příkaz



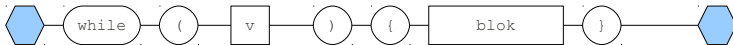
### Přiřazení



### Podmínka



### While



Implementace je založena na soustavě vzájemně se volajících metod třídy syntaktického analyzátoru. Přepínání stavů se v každé metodě realizuje konstrukcí *switch case*, viz ukázka kódu procedury, která zajišťuje definování více proměnných na jednom řádku k jednomu datovému typu (tj. opakování tokenů *carka* a *identifikator*).

```
void Syn::mdeklarace(char *typ) {
    TOKEN tmp; switch(lexx.type) {
        case (CARKA) : term(CARKA);
            tmp = term(IDENTIFIKATOR);
            /* sémantická akce: založení proměnné */
            this->seman->zalozeni(tmp.statement,typ,1);
            mdeklarace(typ);
            break;
        case (STREDNIK) : /*epsilon*/ break;
        default : chyba("ocekavam strednik nebo carku");
    }
}
```

Z ukázy je vidět, že na vhodných místech volá syntaktický analyzátor příslušné sémantické akce.

## 2.4 Sémantické akce

Realizují vlastní činnost prováděných příkazů. Jedná o deklaraci proměnných, typování, vyhodnocování výrazů a chování řídicích struktur.

**Proměnné** jsou implementovány dvourozměrnou tabulkou. Následující příklad ukazuje výpis obsahu tabulky proměnných tak, jak může v průběhu provádění programu vypadat:

jméno proměnné ( <i>řetězec</i> )	typ ( <i>řetězec</i> )	hodnota ( <i>řetězec</i> )	scope ( <i>integer</i> )
\$p	int	14	1
\$clenu	int	2	1
\$pozdrav	string	'ahoj'	1

Hodnota scope slouží k pozdější implementaci podprogramů.

**Typování jazyka** se dá označit za slabé a statické. Proměnné je nutné před použitím deklarovat a svázat je s určitým datovým typem. Při přiřazování nebo porovnávání různých datových typů je hodnota „vhodně“ konvertována:

- při porovnání (nebo přiřazování proměnné typu řetězec na číslo) řetězce s číslem, dojde ke konverzi řetězce na číslo o hodnotě počtu znaků v řetězci
- operátor + aplikovaný na řetězce sloučí k jejich spojování (jiné operátory mezi řetězci nejsou povoleny)

- při přiřazování čísla do řetězce, konvertuje jazyk číslo na řetězec
- porovnávací operátory vrací vždy číselnou hodnotu (1 pro *true*, 0 pro *false*)

Automatickou konverzi při přiřazování do různých datových typů nejlépe demonstruje následující příklad:

```
int $i;
string $s;
$i='nazdar';
$s='nazdar';
echon $i;
echon $s;
```

Vypíše:

```
6
nazdar
```

Další záludnost jazyka spočívá v mnohoúčelovosti operátoru + mezi proměnnými různých typů. Zde se operátor řídí jednoduchým pravidlem – pokud je právě jeden z operandů řetězec, dojde ke konverzi druhého operandu na řetězec. Vyhodnocování přitom probíhá zleva doprava s respektem k uzávorkování. Příklad:

```
echon 'hello ' + 2 + 1;
echon 2 + 1 + ' hello';
echon 'hello ' + (2 + 1);
```

Vypíše:

```
hello 21
3 hello
hello 3
```

Toto chování považuji za nejpraktičtější, protože se dá závorkováním jednoduše směřovat chování automatického přetypování.

**Zpracování výrazů** probíhá v okamžiku, kdy syntaktický analyzátor začne kontrolovat výraz. V tom okamžiku spouští sémantickou akci, která až do konce výrazu ukládá všechny přečtené tokeny do zásobníku.

Po skončení výrazu je zásobník s tokeny nejprve převeden pomocí Shunting-yard algoritmu do postfixové notace, pak je výraz teprve počítán.

**Větvení programu** je řešeno pomocí tzv. *if-flagů*. Třída implementující sémantiku vždy po vyhodnocení výrazu v podmínce určí, zda se budou při dalším zpracování provádět sémantické akce, nebo zda bude probíhat jen syntaktická analýza. Tento stav trvá až do konce bloku ke kterému se podmínka vztahuje. V případě výskytu else se hodnota if-flagu invertuje. Tato jednobitová informace je jen hodnotou v zásobníku – s vnořenými podmínkami je potřeba odlišovat aktuální stav if-flagu a ukládat si hodnoty těch ostatních (vnějších resp. mateřských).

**Cykly** jsou implementovány pomocí prioritní fronty tokenů, pole zásobníků tokenů a pomocných atributů. Chování interpretu při vstupu do cyklu je následující:

- je-li splněna vstupní podmínka vstupu do cyklu, interpret provádí příkazy cyklu a zároveň ukládá tokeny do zásobníku  $[n]$ , kde  $n$  odpovídá aktuální hloubce zanoření
  - narazí-li interpret při provádění cyklu na další (vnořený) cyklus, inkrementuje hodnotu  $n$  (která se tedy mění s hloubkou zanoření aby nedojde ke kolizi ukládaných tokenů) a zanoří se do cyklu
  - po dosažení konce cyklu je podmínka testována znovu
    - \* pokud je podmínka po prvním průchodu cyklem opět splněna, provede se nakopírování zásobníku  $[n]$  na začátek prioritní fronty lexikálního analyzátoru – de facto tak dojde k nakopírování kódu cyklu na vstupní pásce před čtecí hlavu; poté se cyklus provádí znovu
    - \* v případě, že podmínka splněna není, je zásobník  $[n]$  vyprázdněn a interpret pokračuje dál ve čtení
- není-li splněna podmínka, interpret pouze kontroluje syntaktickou správnost kódu až do konce cyklu

### 3 Závěr

Navržený jazyk je dostatečně použitelný pro zapsání velkého množství jednoduchých programů – namátkou třeba výpis násobilky, výpočet největšího společného dělitele, rozvoj Fibonacciho posloupnosti nebo výpočet faktoriálu.

Za nedostatek jazyka lze považovat absence podprogramů. Silnými stránkami jazyka jsou jeho slabá typovost, schopnost automatického přetypování a bezchybné provádění vnořených cyklů.