Zápočtový program

Interpret BASIC

2019 **Pavel Hrdý**

1. Uživatelská dokumentace

Vzhledem k tomu, že se jedná o interpret, předpokládá se, že uživatelem je člověk se základními znalostmi libovolného programovacího jazyka. V této části bude popsáno, jak interpret ovládat a rovněž budou popsány všechny důležité příkazy a konstrukce, které lze v interpreteru používat.

1.1 Ovládání interpreteru

Interpret se používá voláním příkazu z příkazové řádky. Pokud se program jmenuje např. `Interpreter.exe`, stačí otevřít příkazovou řádku ve složce, kde se program nachází a zadat příkaz `.\Intepreter.exe`.

Po zadání tohoto příkazu je uživatel upozorněn, že zapomněl zadat povinný argument – název souboru, který má být interpretován. Argument se píše ihned za název programu, chceme-li tedy např. interpretovat soubor `code.txt`, na příkazovou řádku napíšeme `.\Interpreter.exe code.txt`.

1.2 Hello, World!

Jak je již tradicí, první program, který si ukážeme, je pouhý výpis textu „Hello, World!“ na standardní výstup.

```
10 PRINT "HELLO, WORLD!"
```

Na tomto příkladu si můžeme ukázat hned několik věcí, které jsou pro BASIC typické. Každá řádka musí začínat číslem – to značí číslo řádky. Toto číslování začíná od 10 s tím, že další řádek má číslo vždy o 10 vyšší. Pokud dojde k chybě v číslování, program nebude moci běžet a program skončí chybou.

Další položkou na řádku je příkaz PRINT. Již podle názvu je zřejmé, že tento příkaz vypíše na standardní výstup svůj argument – v tomto případě textový řetězec „HELLO, WORLD!“. V BASICu bylo dříve povinností veškeré identifikátory funkcí psát velkým písmenem – v tomto interpreteru to povinností není, rovněž identifikátory proměnných lze psát malými písmeny, interpret je ale v tomto ohledu case-sensitivní, takže proměnné `W` a `w` jsou dvě rozdílné proměnné.

Obecně vzato, každý řádek v BASICu má formát:

```
ČÍSLO_ŘÁDKY      STATEMENT
```

Co vše může být statement, bude vysvětleno později.

1.3 Proměnné

Jméno proměnné v BASICu musí začínat písmenem, následující znaky mohou být libovolné alfanumerické znaky. Jméno řetězcové proměnné začíná validním jménem proměnné a musí končit znakem dolaru (např. `A$`).

Tento interpret operuje se třemi základními typy – Int, Real a String. Hodnoty těchto typů lze ukládat do proměnných, je ale třeba rozlišit, do jaké proměnné ukládáme příslušné typy. Hodnoty typu String lze ukládat pouze do řetězcových proměnných, do kterých nelze ukládat nic jiného. Hodnoty typu Int a Real lze pak ukládat do obyčejné proměnné.

V BASICu lze rovněž hodnoty ukládat i do polí, které se indexují přirozenými čísly od nuly. Takovou proměnnou může být např. `A(0)`.

1.4 Operátory

Tento interpret podporuje následující operátory (seřazeny sestupně podle priority)

- binární operátor mocnění ^
- unární operátor minus ~
- operátory násobení, dělení * /
- operátory sčítání, odčítání + -
- relační operátory <= < <> > >= =
- negace NOT
- konjunkce AND
- disjunkce OR

Jsou-li aritmetické operátory aplikovány na hodnoty shodného typu, výsledkem bude stejný typ. Jestli alespoň jeden z operandů je typu Real, výsledkem je Real.

Relační a logické operátory vracejí vždy typ Int, který je buď 0 či 1 - podle výsledku logické operace.

1.5 Povolené příkazy

V tomto oddíle budou zmíněny všechny příkazy, které lze v interpreteru použít jako statement na řádce. Všechny zmíněné příkazy jsou zároveň klíčovými slovy jazyka, není tedy možné je používat jako názvy proměnných.

1.5.1 END

Formát: END

Příklad: 10 END

Tento příkaz okamžitě přeruší další vykonávání příkazů.

1.5.2 DIM

Formát: DIM varName(Exp(,Exp,Exp…))

Příklad: DIM A(100)

Tento příkaz inicializuje pole, jehož dimenze je dána počtem argumentů, každý argument určuje maximální rozměr příslušné dimenze.

1.5.3 LET

Formát: LET varName = Exp

Příklad: 10 LET X = (3*8)+8

Tento příkaz definuje proměnnou varName a inicializuje ji hodnotou Exp.

1.5.4 REM

Formát: REM

Příklad: 10 REM Toto je pouhy komentar

Tento příkaz slouží ke komentování, veškerý text, který následuje za klíčovým slovem REM je interpreterem ignorován.

1.5.5 FOR NEXT

Formát: FOR varName = Exp to Exp

NEXT

Příklad: 10 FOR X = 0 TO 5

20 PRINT X

30 NEXT

Jedná se o typický příklad for-cyklu, který jistě znáte z jiných programovacích jazyků. Značný rozdíl od jiných jazyků tkví v příkazu NEXT, který se s příkazem FOR používá. Veškeré příkazy, které uživatel chce, aby se prováděly ve for-cyklu, je potřeba dát mezi příkazy FOR a NEXT.

1.5.6 GOSUB RETURN

Formát: GOSUB Exp

RETURN

Příklad: 10 GOSUB 20

20 PRINT 1

30 RETURN

Jedná se o příkaz, kterým se volá subrutina. Ta se musí označit číslem řádky, pro návrat ze subrutiny se používá příkaz RETURN. Číslo řádky může být vypočteno až během běhu interpreteru, je tedy možné jako argument použít expression.

1.5.7 GOTO

Formát: GOTO Exp

Příklad: 10 GOTO 30
20 PRINT 1
30 PRINT 1
40 GOTO (20+30)

GOTO je příkaz nepodmíněného skoku, který skočí na číslo řádkem, jenž je daný expressionem.

1.5.8 IF THEN

Formát: IF Exp THEN Statement

Příklad: 10 X = 100 THEN PRINT 1

Pokud je expression vyhodnocen na 1, vykoná se Statement, což může být libovolný příkaz.

1.5.9 READ / DATA

Formát: DATA Exp, Exp …

READ varName, varName …

Příklad: DATA 1, 2, 3

READ X, Y, Z

Tyto příkazy se používají k tomu, aby se do většího počtu proměnných načetly programátorem zadané hodnoty. Příkaz DATA musí vždy předcházet příkazu READ. Počet proměnných a hodnot může být 1 a více.

1.5.10 PRINT

Formát: PRINT Exp/String/varName

Příklad: PRINT 10

PRINT X

PRINT "HELLO"

Na nový řádek vypíše hodnoty zadané jako argument. Argumentem může být výraz, řetězec či proměnná. Argumentů může být více, pak musejí být oddělené čárkou. V tomto případě dojde k vypsání všech argumentů na jeden řádek, s tím že hodnoty jsou oddělené mezerou.

1.5.11 POP

Formát: POP

Příklad: POP

K tomu, aby se mohlo při příkazu GOSUB / RETURN vrátit na původní adresu, se používá interní zásobník, na který se ukládá adresa, na kterou se musí při RETURNu skočit. Příkazem POP je možné adresu z tohoto zásobníku vyhodit.

1.5.12 INPUT

Formát: INPUT varName(,varName,varName…)

Příklad: INPUT X

Slouží k zadávání hodnot do standardního vstupu, tyto hodnoty jsou postupně ukládány do proměnných, které jsou argumenty tohoto příkazu.

1.6 Aritmetické funkce

V interpreteru je rovněž k dispozici množství aritmetických funkcí, které lze používat pro výpočty. Použití těchto funkcí je vždy stejné: NAZEV_FUNKCE(ARGUMENT). Argument může být libovolný expression. Výsledky funkcí je možné přiřazovat (a dělá se to, neboť by jinak funkce neměly význam) do proměnných.

1.6.1 ABS

Vrací absolutní hodnotu čísla.

1.6.2 CLOG

Vrací logaritmus argumentu o základu 10.

1.6.3 INT

Vrací nejvyšší přirozené číslo, které je menší či rovno argumentu.

1.6.4 LOG

Vrací logaritmus argumentu o přirozeném základu.

1.6.5 RND

Tato funkce je výjimkou, přijímá sice argument, ale nijak ho nepoužívá. Vygeneruje náhodné číslo v polouzavřeném intervalu [0,1).

1.6.6 SGN

Vrací -1, jestliže argument je záporné číslo, 0 pokud je 0 a 1, pokud je argument kladné číslo.

1.6.7 SQR

Vrací druhou odmocninu z argumentu.

1.6.8 ATN

Vrací arkustangens argumentu.

1.6.9 COS

Vrací kosinus argumentu.

1.6.10 SIN

Vrací sinus argumentu.

1.7 Funkce pro práci s řetězci

Tyto funkce opět pouze přijímají argument v závorkách.

1.7.1 ASC

Vrací ASCII číslo prvního znaku stringu, který je argumentem.

1.7.2 CHR

Výraz, který je argumentem, vyhodnotí jako ASCII kód a vrátí string, který obsahuje příslušný znak.

1.7.3 LEN

Vrátí délku stringu, který je argumentem.

1.7.4 STR

Převede číslo na string.

1.7.5 VAL

Převede string na číslo.

2. Programátorská dokumentace

Práce interpreteru je rozdělena do tří základních kroků: lexikální analýza, sémantická analýza, a nakonec vykonávání kódu intermediate code virtual machine (ICVM). Jednotlivé kroky budou v této kapitole blíže popsány.

2.1 Lexikální analýza

V lexikální analýze se provádí čtení kódu a vytváření tokenů, které reprezentují kód. Veškerá lexikální analýza probíhá ve třídě Lexer, kde nejdůležitější metoda, GetNextToken(), vrací další token, který byl v kódu nalezen.

Tokeny jsou řešeny hierarchií tříd, kde všechny tokeny dědí od abstraktní třídy TokenType.

2.2 Sémantická analýza

Sémantickou analýzu provádí recursive-descent parser. Nejprve musela být navržena gramatika celého jazyka.[1] Sémantická analýza pak postupně prochází tokeny, které mu vrací lexikální analýza a dává jim význam – konkrétně v tomto případě je převádí do instrukcí ICVM.

Hlavní kód sémantické analýzy se nachází ve třídě Parser.

2.3 Vykonávání intermediate kódu

Po tom, co parser vytvoří instrukce pro ICVM, dochází následně k jejich vykonávání. ICVM je zásobníkový stroj, který pro svou činnost, jak již z názvu vyplývá, používá zásobník. Na ten jsou ukládány všechny používané hodnoty, se kterými pak ICVM pracuje, je-li k tomu instrukcí vyzván.

Samotné instrukce jsou řešeny hierarchií tříd, kde kořen hierarchie je abstraktní třída Instruction.

[1] K návrhu byl použit tento zdroj: https://rosettacode.org/wiki/BNF_Grammar#BASIC