Vysoké učení technické v Brně
Fakulta informačních technologií

Dokumentace projektu do předmětů IFJ a IAL

Implementace překladače imperativního jazyka IFJ17

Tým 118, varianta II

Furda Jiří xfurda00: 25%
Havan Peter xhavan00 – ved.: 25%
Juřica Jiří xjuric29: 25%
Stano Matej xstano04: 25%

6. 12. 2017

1. Obsah

1. Obsah 2
2. Práce v týmu 3
2.1. Rozdělení úkolů 3
3. Implementace 4
3.1. Lexikální analýza 4
3.2. Tabuľka symbolov 4
3.3. Syntaktická a sémantická analýza 4
3.4. Precedenční analýza 5
3.5. Generovanie trojadresného kódu 6
4. Seznam použitých obrázků 7

2. Práce v týmu

Na projektu jsme si vyzkoušeli hned několik nových věcí. Jelikož nikdo z nás zatím nedělal na žádné práci, která by vyžadovala kooperaci více lidí, nebylo při počátcích zcela jednoduché domluvit pravidelné schůzky, přípravy na ně a vhodně si rozdělit povinnosti. Díky průběžným novým informacím z přednášek se nám však záhy dařilo získávat lepší a lepší představu o tom, jak by kompilátor měl vypadat a brzy jsme se sladili. Ke sdílení jednotlivých modulů jsme začali požívat verzovací systém s repozitářem na Githubu a průběžně své části konzultovali s kolegy na Messengeru.

2.1. Rozdělení úkolů

V průběhu prací všichni z nás psali testy pro své části a kompilátor jako celek testovali jak našimi testy, tak těmi, které uvolnily naši spolužáci. Dále již měl každý na starost svoji část, jmenovitě:

- Jiří Furda – Pracoval na precedenční analýze, kterou navrhl a naprogramoval.
- Peter Havan – Jako vedoucí řídil běh návrhu a programování, mimo to naprogramoval část ke generování výsledných instrukcí a podílel se na tabulce symbolů.
- Jiří Juřica – Se postaral o návrh i implementaci lexikálního analyzátoru, vytvořil testovací skript a sestavil dokumentaci.
- Matej Stano – Pomáhal s řízením prací na projektu, navrhl a implementoval syntaktický analyzátor a tabulku symbolů.

3. Implementace

3.1. Lexikální analýza

Je zpracována v souborech scanner.c a scanner.h, řeší zpracování znaků ze stdin a definuje všechny tokeny, které pak zpracovávají ostatní moduly.
Návrh konečného automatu začal obrovskou tabulkou, kdy chybné stavy byly při zjištěných závadách průběžně opravovány. Tabulka byla nakonec převedena do grafu, který tvoří přílohu A. Pro scanner samotný je nejdůležitější funkce getToken, která při každém zavolání načte jeden lexém, návratovou hodnotou posílá stavový kód a pomocí ukazatele v atributu předává výsledný token, který obsahuje číslo s typem tokenu a strukturu s hodnotou, která se plní, pokud to daný token vyžaduje. Scanner byl navržen tak, aby nezáleželo, zda mezi operandy a operátory ve výrazu je či není bílý znak (mimo znak konce řádku). Pro jednoduchost je to tak mezi všemi termy, kdy scanner načítá nejdelší možnou posloupnost znaků, která splňuje požadavky rozeznaného typu a v případě následné chyby se lexikální chyba vyhodnotí až při druhém volání funkce getToken. Je to patrné např. na tomto příkladu:

```
sco#pe
```

Tento kód se rozloží do identifikátoru s textem „sco“ a při druhém volání by se zaslala chyba, ke které už nedojde, protože parser mezitím vyhodnotí syntaktickou chybu, kvůli očekávání klíčového slova scope, ne jména proměnné. To nás v týmu vedlo k diskuzi, zda je popisované chování správné, nakonec pomohla konzultace s cvičícími na diskuzním fóru, na jejíž základě jsme se toto chování rozhodli ponechat.
Funkce getToken dále využívá několik interních funkcí pro zpracování stringů, názvu proměnných, zpracování číselných hodnot a escape sekvencí. Samotné předávání tokenu z funkce musí proběhnou do již vytvořené struktury s inicializovaným stringem ve struktuře s hodnotou, postup byl dohodnut při práci na parseru pro jednoduší zpracování.

3.2. Tabuľka symbolov

Tabuľka uchováva všetky potrebné informácie o funkciách a ich premenných potrebné pri sémantickej kontrole. V našom prípade sme sa rozhodli, že prvky tabuľky, ktoré sú uložené ako parametre funkcie budú odkazovať okrem ďalšieho prvku v rámci tabuľky aj na ďalší a predchádzajúci parameter. Teda parametre medzi sebou vytvoria dvojsmerne viazaný zoznam, ktorý uľahčí kontrolovanie typov, počtu a poradia parametrov. V jazyku free basic môže byť ID parametru iné v deklarácii a iné v definícií, preto bolo treba vytvoriť funkciu ktorá v prípade potreby zmeny ID parametru presunie štruktúru na správne miesto v tabuľke, tak aby odpovedal hash nového ID, ale neporuší viazanie v rámci parametrov.

3.3. Syntaktická a sémantická analýza

Syntaktická analýza programu je riešená metódou rekurzívneho zostupu, pomocou gramatiky priloženej nižšie, a precedenčnou analýzou. Pred začatím zostupu parser alokuje miesto pre tabuľku symbolov a pre štruktúru token cez ktorú od scanneru prijíma informácie. Pretože generovanie inštrukcií prebieha zároveň s analýzou, tak sme token častokrát využili aj na posielanie informácii modulu ilist. Parser okrem syntaktickej kontroly vykonáva aj sémantické kontroly. V prípade že sa vykonáva pravidlo priradenia do premennej, parser načítava aj jeden token za „rovná sa“ z dôvodu rozlíšenia medzi výrazom a volaním funkcie. V prípade výrazu predáva riadenie expr. Volania funkcií (typovú kontrolu parametrov, počet parametrov a generovanie príslušných inštrukcií) vykonáva parser nerekurzívne pomocou informácii uložených v tabuľke symbolov.

LL pravidlá:

1. <prog> -> SCOPE EOL <stats> END SCOPE EOL EOF
2. <prog> -> <function-declaration> <prog>
3. <prog> -> <function-definition> <prog>
4. <function-declaration> -> DECLARE FUNCTION ID (<function-parameters> AS <type> EOL
5. <function-definition> -> FUNCTION ID AS (<function-parameters> AS <type> EOL <stats> END FUNCTION EOL
6. <type> -> INTEGER
7. <type> -> DOUBLE
8. <type> -> STRING
9. <function-parameters> ->)
10. <function-parameters> -> ID AS <type> <more-func-params>
11. <more-func-params> ->)
12. <more-func-params> -> , ID AS <type> <more-func-params>
13. <assign> -> = <expr> EOL
14. <assign> -> EOL
15. <stats> -> ε
16. <stats> -> INPUT ID EOL <stats>
17. <stats> -> PRINT <expr> ; EOL <stats>
18. <stats> -> DIM ID AS <type> <assign> <stats>
19. <stats> -> ID = <expr> EOL <stats>
20. <stats> -> RETURN <expr> EOL <stats>
21. <stats> -> IF <expr> THEN EOL <stats> ELSE EOL <stats> END IF EOL <stats>
22. <stats> -> DO WHILE <expr> EOL <stats> LOOP EOL <stats>

	SCOPE	EOL	END	DECLARE	FUNCTION	ID	(	)	AS	STRING	INTEGER	DOUBLE	,	INPUT	PRINT	RETURN	DIM	=	DO	WHILE	ELSE	IF	LOOP	EOF [$]
<prog>	1			2	3																			
<function-declaration>				4																				
<function-definition					5																			
<type>										8	6	7												
<function-parameters>						10		9																
<more-func-params>								11					12											
<assign>		14																13						
<stats>			15			19								16	17	20	18		22		15	21	15	

Obr. 3.1. LL tabulka, Matej Stano

3.4. Precedenční analýza

Syntaktická analýza metodou zdola nahoru je použita pro vyhodnocování výrazů. Modul expr se nachází v souborech expr.c a expr.h. Dále jsou využity dva typy zásobníků – jeden pro ukládání terminálů pro zpracování (obsažený v souborech stack.c a stack.h) a druhý obsahující datové typy jednotlivých hodnot přímo odpovídající zásobníku ve tříadresném kódu (obsažený v souborech tokstack.c a tokstack.h)

Tento modul je volán pouze z modulu parser a to vždy v těchto čtyřech případech:

- Přiřazení (např. int = 4)
- Porovnání (např. if 1 < 2 then)
- Výpis (např. print !”Slovo”;)
- Návratová hodnota funkce (např. return 3)

Poté co parser předá řízení tomuto modulu, je zde voláno načítání tokenů z modulu scanner. Terminály na vstupu jsou zpracovány na základě algoritmu probíraném v předmětu IFJ s využitím zásobníku stack, současně s redukcí je volána funkce modulu ilist pro generování příslušné instrukce tříadresného kódu (s výjimkou závorek) včetně přetypování operandů, pokud je třeba. Pro datový typ

Integer a Double jsou použity zásobníkové instrukce, kdežto pro typ String jsou použity instrukce nezásobníkové v kombinaci s dvěma pomocnými proměnnými. Zásobník tokStack je v tento okamžik aktualizován tak, aby odpovídal hodnotám uloženým na zásobníku ve tříadresném kódu.
Jakmile je načten token s typem nepatřícím do výrazu (s výjimkou tokenu pro znak středníku, který celý modul odstartuje znovu), je token odkazem vrácen modulu parser, modul expr je následně ukončen s návratovou hodnotou 0 a řízení je předáno zpět modulu parser.
Pokud je v kterémkoliv okamžiku běhu modulu objevena chyba, je modul ihned po uvolnění dynamické paměti ukončen s návratovou hodnotu odpovídající dané chybě a řízení je opět předáno zpět modulu parser.

	+	-	\	*	/	(	)	i	$	=	<>	<	<=	>	>=	str
+	>	>	<	<	<	<	>	<	>	>	>	>	>	>	>	<
-	>	>	<	<	<	<	>	<	>	>	>	>	>	>	>	#
\	>	>	>	<	<	<	>	<	>	>	>	>	>	>	>	#
*	>	>	>	>	>	<	>	<	>	>	>	>	>	>	>	#
/	>	>	>	>	>	<	>	<	>	>	>	>	>	>	>	#
(	<	<	<	<	<	<	=	<	#	<	<	<	<	<	<	<
)	>	>	>	>	>	#	>	#	>	>	>	>	>	>	>	#
i	>	>	>	>	>	#	>	#	>	>	>	>	>	>	>	#
$	<	<	<	<	<	<	#	<	#	<	<	<	<	<	<	<
=	<	<	<	<	<	<	>	<	>	#	#	#	#	#	#	<
<>	<	<	<	<	<	<	>	<	>	#	#	#	#	#	#	<
<	<	<	<	<	<	<	>	<	>	#	#	#	#	#	#	<
<=	<	<	<	<	<	<	>	<	>	#	#	#	#	#	#	<
>	<	<	<	<	<	<	>	<	>	#	#	#	#	#	#	<
>=	<	<	<	<	<	<	>	<	>	#	#	#	#	#	#	<
str	>	#	#	#	#	#	>	#	>	>	>	>	>	>	>	#

Obr. 3.2. Precedenční tabulka, Jiří Furda

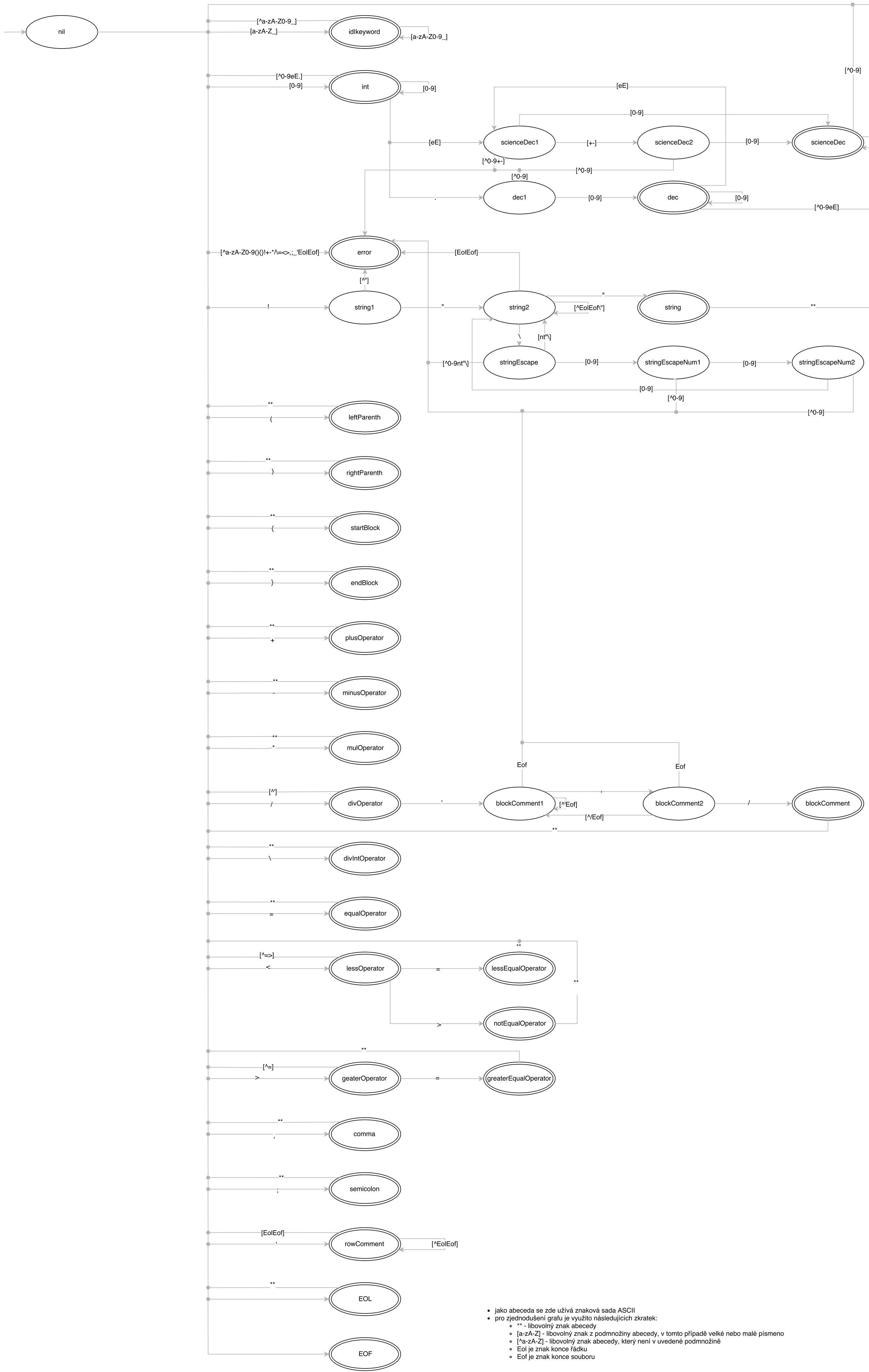
3.5. Generovanie trojadresného kódu

Samotné generovanie inštrukcií trojadresného kódu prebieha v module ilist. Modul ilist sa nachádza v súboroch ilist.c a ilist.h. Cieľom bolo vytiahnuť samotné generovanie inštrukcií, správu nad použitými štruktúrami a prácu s pamäťou mimo moduly starajúce sa o sémantickú analýzu.
Najväčšiu prekážku predstavoval fakt, že inštrukcie sa nemali na štandartný výstup vypisovať v prípade nájdenej chyby. Keďže sme sa rozhodli prechádzať vstup len jedenkrát, bolo nutné inštrukcie postupne ukladať. Štruktúra na uloženie inštrukcií bola implementovaná jednoduchým poľom reťazcov jazyka C. Reťazce ako aj samotné pole sú dynamicky alokované, aby mohli v prípade potreby meniť svoju veľkosť.
Ďalšiu prekážku predstavovala tvorba a správa unikátnych náveští pri podmienkach a cykloch. Bola nutná úzka spolupráca so súborom parser.c, ktorý bol vhodný na vytvorenie počítadla pre pomenovania náveští.
V neposlednom rade je modul zodpovedný za tvorbu vstavaných funkcií a ich výpis v prípade, že boli zavolané.
Najdôležitejšou funkciou modulu je funkcia na pridanie inštrukcie, ktorá je volaná zo súborov parser.c a expr.c. Ďalšie funkcie sa týkajú obsluhy štruktúr ako je inicializácia, alokácia a uvoľnenie, volané najmä zo súboru main.c.

4. Seznam použitých obrázků

Obr. 3.1. LL tabulka, Matej Stano 5

Obr. 3.2. Precedenční tabulka, Jiří Furda 6



- jako abeceda se zde užívá znaková sada ASCII
- pro zjednodušení grafu je využito následujících zkratek:
  - ** - libovolný znak abecedy
  - [a-zA-Z] - libovolný znak z podmnožiny abecedy, v tomto případě velké nebo malé písmeno
  - [^a-zA-Z] - libovolný znak abecedy, který není v uvedené podmnožině
  - Eol je znak konce řádku
  - Eof je znak konce souboru