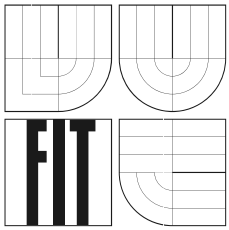
Dokumentácia k projektu z predmetov IAL a IFJ



Varianta zadania: tím 2, a/1/I

- Knuth-Moris-Prattov algoritmus pre vyhľadávanie podreťazcov
- quick-sort ako radiaci algoritmus funkcie `sort()`
- tabuľka symbolov ako binárny vyhľadávací strom

Riešitelia:

- Kontra Matúš, `xkontr00` (vedúci tímu), 25%
- Kobrtek Jozef, `xkobrt00`, 25%
- Smetana Matej, `xsmeta02`, 25%
- Šebeň Patrik, `xseben00`, 25%

Rozšírenia:

- spracovanie riadkových a blokových komentárov
- hexadecimálny, oktálový a binárny zápis celočíselných konštánt
- príkaz `if` bez časti `else`

Obsah

1 Úvod **2**

2 Lexikálny analyzátor **3**
2.1 Reťazcové konštanty 3
2.2 Číselné konštanty 3

3 Syntaktický analyzátor **5**
3.1 Syntaktická analýza zhora-nadol 5
3.2 Syntaktická analýza zdola nahor 6
3.3 Tabuľka symbolov ako binárny vyhľadávací strom 7

4 Sémantická kontrola **8**

5 Interprét **9**
5.1 Funkcia find() a algoritmus KMP 9
5.2 Funkcia sort() a algoritmus quick-sort 10

6 Záver, metriky kódu **11**

7 Prílohy **12**

1 Úvod

Táto dokumentácia popisuje riešenie projektu do predmetov IAL a IFJ. Zadanou úlohou bolo v jazyku C naprogramovať interprét jazyka `ifj2008`, ktorého špecifikácie sa nachádzajú v zadaní úlohy.

Interprét je rozdelený do jednotilvých funkčných častí, pričom každá z nich je analyzovaná v samostatnej kapitole, ako aj s analýzou použitých algoritmov a implementačným popisom.

2 Lexikálny analyzátor

Úlohou lexikálneho analyzátora je previesť text zdrojového programu jazyka `ifj2008` na postupnosť tokenov. Z matematického hľadiska predstavuje deterministický úplný konečný automat, pričom ak automat skončí v jednom z koncových stavov, je vstupný reťazec prijatý, v opačnom prípade ohlási lexikálnu chybu. Scanner tiež komunikuje s tabuľkou symbolov, v ktorej vyhľadáva zhody s kľúčovými slovami jazyka `ifj2008`.

Funkcia lexikálneho analyzátora `GetNextToken()` dostane ako parameter odkaz na premennú typu string, v ktorej vráti atribút načítaného tokenu. Návratová hodnota predstavuje identifikátor token (číselná konštanta), ktoré sú definované v hlavičkovom súbore `lexer.h`, alebo chybový kód (`codes.h`).

Nami implementovaný automat je schopný rozpoznať aj blokové komentáre, taktiež celočíselné konštanty v binárnom, hexadecimálnom a oktálovom tvare. V prípade načítavania relačných operátorov sa pozerá aj na nasledujúci znak, nakoľko jazyk `ifj2008` používa aj viacznakové relačné operátory. Schéma implementovaného konečného automatu nájdete v obrazových prílohách 2 a 1.

2.1 Reťazcové konštanty

Na spracovanie reťazcových konštánt sme použili dodané funkcie z modulu `str.c` pre načítavanie reťazcov teoreticky lubovoľnej dĺžky. Samotný reťazec sa uloží do na odkaz `attr`, ktorý funkcia `GetNextToken()` dostala ako parameter. Escape-sekvencie sa do výsledného reťazca ukladajú svojimi ASCII hodnotami.

2.2 Číselné konštanty

Spracovanie číslic začína príchodom čísla v stave `INITIAL`, pričom počas spracovania sa rozoznávajú nasledovné skupiny - 0,1,2-7,8-9. Ak je prvá nula, môže sa teoreticky jednať o ktorýkoľvej typ konštanty, preto sa rozhoduje podľa nasledujúceho symbolu. Ak je ním x, označí sa za hexadecimálne, ak je ďalšie 0 alebo 1 - binárne, 2-7 oktálové, 8-9 dekadické, pri načítaní desatinnej čiarky desatinné. Desatinné čísla sa navyše môžu vyskytovať aj v exponenciálnom zápise. Medzi jednotlivými stavmi reprezentujúcimi načítavanie daného typu konštanty sa preskakuje smerom z dola nahor - ak po načítaní sekvencie jednotiek a núl príde na vstup číslo 2-7, prejde sa k oktálovému, prípadne ak 8-9, k dekadickému a pod. Binárne číslo musí byť zakončené malým písmenom *b*, v opačnom prípade je označené za oktálovú konštantu (ak začínalo nulou), alebo dekadickú (ak začínalo jednotkou). Pri načítavaní čísla nedôjde k lexikálnej chybe - ak sa za číslom nachádza chybný text,

rozpozná sa na tomto mieste koniec čísla ako tokenu, načíta sa nasledujúci token, a až syntaktický analyzátor odhalí prípadnú chybu. Graf konečného automatu, ktorý spracováva čísla, sa nachádza v prílohe ako Obrázok 1.

3 Syntaktický analyzátor

V projekte sme implementovali syntaktický analyzátor zhora-nadol, ktorý spracúva všetky tokeny okrem výrazov, a parser zdola-nahor, určený pre spracovanie výrazov.

3.1 Syntaktická analýza zhora-nadol

Parser pracujúci zhora nadol je založený na množine pravidiel LL-gramatiky s ϵ-pravidlami, na základe ktorej bola zostavená LL-tabuľka. S využitím tejto tabuľky je možné simulovať tvorbu derivačného stromu.

Gramatické pravidlá:

1. $< prog >::=< int\_decl\_or\_main\_def >$
2. $< prog >::=< oth\_decls >$
3. $< oth\_decls >::=$ `double id ;`
4. $< oth\_decls >::=$ `string id ;`
5. $< int\_decl\_or\_main\_def >::=$ `int` $< id\_or\_main >$
6. $< id\_or\_main >::=$ `id ;`
7. $< id\_or\_main >::=$ `main()` $< comp\_stmnt >$
8. $< comp\_stmnt >::= \{< stmnt\_list >\}$
9. $< stmnt\_list >::=< stmnt >< stmnt\_list >$
10. $< stmnt\_list >::= \varepsilon$
11. $< stmnt >::=< comp\_stmnt >$
12. $< stmnt >::=$ `cin >>` $id < in\_id\_list >$
13. $< in\_id\_list >::=$ `>> id` $< in\_id\_list >$
14. $< in\_id\_list >::=$ `;`
15. $< stmnt >::=$ `cout <<` $< expr >< out\_expr\_list >$
16. $< out\_expr\_list >::=$ `<<` $< expr >< out\_expr\_list >$
17. $< out\_expr\_list >::=$ `;`

18. $< stmnt >::=< expr >$ `;`

19. $< stmnt >::=$ `while` $< expr >< comp\_stmnt >$

20. $< stmnt >::=$ `return` $< expr >$ `;`

21. $< stmnt >::=$ `if` $< expr >< comp\_stmnt >< else\_part >$

22. $< else\_part >::=$ `else` $< comp\_stmnt >$

23. $< else\_part >::= \varepsilon$

Výslednú LL-tabuľku je možno nájsť v prílohe ako Tabuľka 1. Samotný parser je implementovaný ako kaskáda funkcií obsluhujúce jednotlivé nonterminály, na ktorej začiatku je funkcia `parse()`, ktorá dostane otvorený zdrojový súbor, získa prvý token a spustí syntaktickú analýzu zavolaním procedúry `prog()`. Na záver syntaktickej analýzy kontroluje, či sa za funkciou `main()` nenachádza ďalší text. Procedúrou `prog()` začína samotné vetvenie syntaktickej analýzy - na deklaračnú časť, a programovú časť. Jednotlivé obslužné funkcie obsluhujúce príslušné nonterminály sa starajú o pridanie a kontrolu duplicity záznamu v tabuľke symbolov (ak šlo o deklaráciu premennej) prostredníctvom funkcií `bvs_insert()` a `bvs_lookup()`, prípadne vydávanie troj-adresných inštrukcií pomocou `emitInstr()` do globalneho zoznamu inštrukcií `IList`, ak procedúra spracováva funkcie alebo volá výrazový syntaktický analyzátor. Funkcie sa podľa poteby vzájomne volajú.

3.2 Syntaktická analýza zdola nahor

Parser vykonávajúci analýzu zdola nahor a sémantickú kontrolu je metricky najrozsiahlejšou časťou projektu. Do projektu bol implementovaný precedenčný syntaktický analyzátor ako zásobníkový automat, ktorý pracuje na základe posunovacích a redukčných pravidiel uvedených v precedenčnej tabuľke.

Precedenčná tabuľka (Tabuľka 2) je pomerne obsiahla, preto aj kód simulujúci vykonávanie týchto pravidiel je relatívne dlhý. Základom je funkcia `parseExpr()` je volaná syntaktickým analyzátorom vtedy, keď narazí na počiatok výrazu (môže ním byť identifikátor, konštanta, funkcia alebo zátvorka). Ako parameter dostane odkaz na tabuľku symbolov, do ktorej zapisuje medzivýsledky pri spracovávaní výrazu.

Redukcie prebiehajú nasledovne:

- konštanta na zredukuje na $< expr >$ s hodnotou konštanty, a uloží na zásobník
- dvojica zátvoriek ako volanie funckie, alebo sa vyhodnotí ako výraz v zátvorkách (opäť $< expr >$)
- relačné a aritmetické operátory sa uložia ako $< expr >$ s (pravdivostnou) hodnotou vyrazu
- priradenie ako $< expr >$ s hodnotou priradenia

3.3 Tabuľka symbolov ako binárny vyhľadávací strom

Jednou zo špecifikácii zadania bolo implementovať tabuľku symbolov ako binárny vyhľadávací strom. Výhodou binárneho vyhľadávacieho stromu je logaritmická časová zložitosť vyhľadania $O(\log n)$, v najhoršom prípade $O(n)$, ak má BVS tvar lineárneho zoznamu.

Záznam (uzol) v BVS obsahuje:

- meno premennej
- dátový typ premennej
- príznak užívateľská/pomocná
- dáta premennej
- odkaz na pravý a ľavý podstrom

Vyhľadávanie prebieha nerekurzívne pomocou funkcie `bvs_lookup()`, pričom ako kľúč sa používa meno premennej, ktoré bola buď užívateľské, alebo ak sa jedná o pomocnú premennú, tak generované funkciou `uniqueID()`.

4 Sémantická kontrola

Sémantická analýza prebieha v rámci oboch syntaktických analyzátorov. LL parser má za úlohu kontrolovať správnosť deklarácie premennej (nie je možné priradiť do nedefinovanej premennej), redundanciu (viacnásobná deklarácia), kontroluje správnosť dátových typov v podmienkach a návratové typy pri ukončení funkcie príkazom `return`.

Precedenčný syntaktický analyzátor vykonáva sémantickú kontrolu pri aplikácii redukčných pravidiel. Jedná sa najmä o implicitné pretypovania pri aritmetických operáciách, kde dochádza ku kombinácii dátových typov `double` a `int`. Vtedy sa premenná typu int pretypuje na double. Ďalej je to kontrola L-hodnoty, do ktorej by sa mal priradiť výsledok určitého výrazu - na tomto mieste nesmie figurovať napr. konštanta alebo funkcia.

Pri spracovaní aritmeticko-logických, volaní funkcií, podmienok alebo cyklu `while` sa vydávajú trojadresné inštrukcie funkciou `emitInstr()`, ktorá vytvorí novú položku na konci zreťazeného zoznamu a uloží do nej typ operácie a príslušné operandy, resp. odkazy do tabuľky symbolov.

Po naplnení zoznamu inštrukcií sa v hlavnom programe zavolá samotná funkcia interprétu `runIList()`, ktorý ich vykoná.

5 Interprét

Koncovou časťou projektu je interprét trojadresného kódu generovaného sémantickým analyzátorom. Stará sa tiež o volanie funkcií jazyka `ifj2008` a ošetrovanie behových výnimiek programu - napr. pokus o delenie nulou, kontrola vstupného formátu pri načítavaní z konzoly do premennej. Komunikuje tiež s tabuľkou symbolov, z ktorej načítava hodnoty identifikátorov a kam ukladá spracované výsledky.

Vstavané funkcie jazyka `ifj2008` nad dátovým typom `string`:

- `length()` - vráti parameter `length` zo štruktúry `string`
- `concat()` - vytvorí nový reťazec do ktorého nakopíruje najprv prvý reťazec, potom k nemu pripojí druhý
- `find()` - vyhľadá podreťazec v reťazci (viď nižšie)
- `sort()` - usporiada vzostupne znaky v reťazci (viď nižšie)

5.1 Funkcia find() a algoritmus KMP

Knuth-Morris-Parratov algoritmus, známy tiež ako KMP-algoritmus, je veľmi dômyselným a jednoduchým spôsobom ako nájsť v reťazci podreťazec.

Pred samotným vyhľadávaním sa hľadaný (vzorový) reťazec spracuje, a zostaví sa tzv. *chybová funkcia*. Následne pracujeme s plávajúcim oknom v rámci prehľadávaného reťazca, a zľava sa testuje znak po znaku na zhodu s prehľadávaným reťazcom. V prípade nezhody posunieme začiatok okna so vzorom o hodnotu indexu chyby doprava, ale vrátime o toľko pozícii doľava, aká je funčná hodnota chybovej funkcie pre daný index nezhody.

Chybová funkcia $f(x)$ má pre index $x = 0$ hodnotu -1, pre $x = 1$ je rovná 0, pre ostatné indexy udáva dĺžku najdlhšej predpony vzoru, ktorá je zhodná s príponou $V[0 \ldots i-1]$, kde `V` je vzorový reťazec a `i` je index nezhody v okne vzoru. Takýmto spôsobom sa pri nezhode nemusí porovnávať časť reťazca o dĺžke, ktorú udáva hodnota chybovej funkcie pre daný index nezhody.

Samotná implementácia pozostávala z vytvorenia dynamického poľa o dĺžke vzorového reťazca, ktoré sa následne naplní hodnotami chybovej funkcie pre daný reťazec. Potom nasleduje jednoduchý porovnávací cyklus `while`, ktorý skončil buďto nájdutím zhody, alebo narazením na koniec prehľadávaného reťazca.

5.2 Funkcia sort() a algoritmus quick-sort

Quicksort je radiaci algoritmus patriaci do kategórie porovnávacích algoritmov. Jeho priemerná zložitosť je linearitmická, vyjadriteľná ako $O(n \cdot \log n)$, avšak v najhoršom prípade môže byť až $O(n^2)$, teda kvadratická. Zložitosť algoritmu je daná najmä výberom tzv. *pivota*, pričom najlepší výsledok by sme získali výberom mediánu z triedeného poľa. Nakoľko je počítanie mediánu zdĺhavé, je medián nahradený lubovoľnou hodnotou z daného poľa.

Po výbere pivota nasleduje mechanizmus *partition*, ktorým sa pole zotriedi tak, aby pivot ostal na mieste, kde má byť - teda všetky väčšie hodnoty od neho budú napravo, a všetky menšie vľavo. To sa dosiahne tak, že prvý prvok zľava väčší ako pivot sa vymení s prvkom zprava menším ako pivot, až kým sa idexy zľava a zprava neskrížia. Nakoniec sa pivot vymení s indexom, ktorý má väčšiu hodnotu (myslí sa pravý alebo ľavý), a zavolá sa ten istý algoritmus najprv na časť poľa od pivota naľavo, potom napravo.

V našom riešení sme algororitmus implementovali rekurzívne a in-situ, t.j. k presunom dochádza priamo v poli. Pivot je vybraný náhodne, a následne je prehodený s posledným prvkom poľa, resp. jeho časti. Modifikovali sme však posun po poli - oba indexy vychádzajú zľava, pričom prvý sa inkrementuje v každej iterácii, ale druhý len ak dôjde k výmene. Prechod končí vtedy, ak prvý index narazí na koniec poľa. Vtedy ostane druhý index nastavený na pozícii, kam sa uloží pivot, ktorý bol dočasne na konci poľa. Po správnom umiestnení pivota sa algoritmus rekurzívne aplikuje na ľavú stranu poľa od pivota, potom na pravú.

6 Záver, metriky kódu

Projekt bol pre nás veľmi cennou programátorskou skúsenosťou, nakoľko sme si vyskúšali prácu v tíme a s tým spojené problémy. Najväčšie komplikácie spôsobovala najmä voľba rozhrania, mnohokrát totiž návrh jedného člena tímu nevyhovoval predstavám ostatným, ktorí by si o niečo viac skomplikovali návrh. Novinkou pre nás bolo tiež ladenie a testovanie projektu o takomto rozsahu, navyše napísaného viacerými ľuďmi.

Výsledný interprét nepozná cykly do-while a for, čo by mohlo byť predmetom neskoršieho rozšírenia. Veľmi prínosná by bola tiež možnosť užívateľsky definovaných funkcií.

Výsledný program bol testovaný ako pod operačným systémom Windows, tak aj v prostredí UNIX na serveri merlin.fit.vutbr.cz. Na základe výsledkov testov môžeme konštatovať, že sa nám podarilo splniť požiadavky kladené na funkčnosť a prevedenie.

Počet riadkov kódu, vrátane hlavičkových súborov: 5286
Veľkosť spustiteľného súboru (Windows, gcc 3.4.5): 85,3 KiB

7 Prílohy

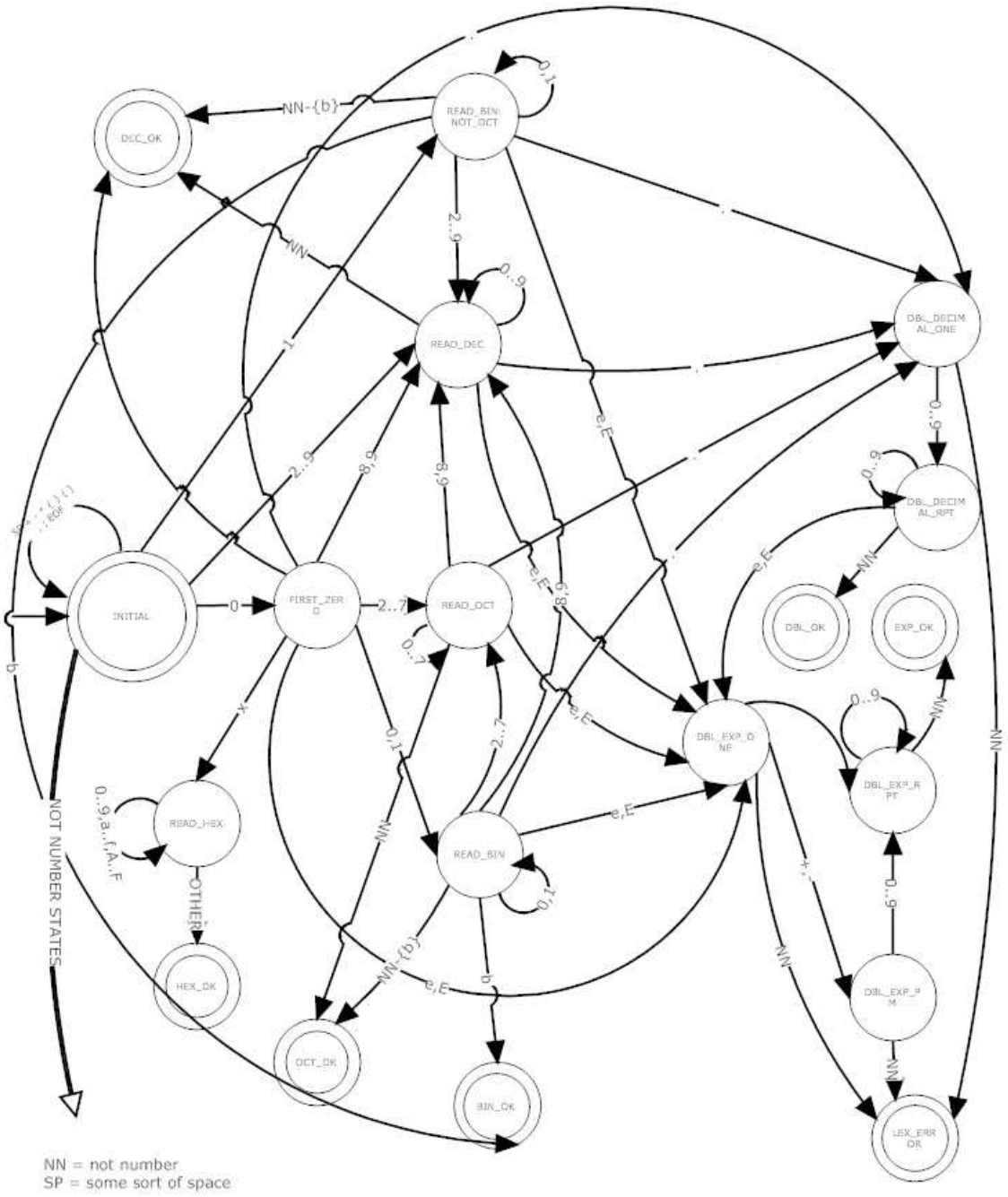
	id	int	double	string	main	;	cin	cout	if	else	while	>>
< *prog* >		1	2	2								
< *int_decl_or_main_def* >		5										
< *oth_decls* >			3	4								
< *id_or_main* >	6				7							
< *comp_stmnt* >												
< *stmnt_list* >	9								9		9	
< *stmnt* >	18						12	15	21		19	
< *in_id_list* >						14						13
< *out_expr_list* >						17						
< *else_part* >	23						23	23	23	23	23	
< *expr* >												

	<<	return	{	}	(,const,func
< *prog* >					
< *int_decl_or_main_def* >					
< *oth_decls* >					
< *id_or_main* >					
< *comp_stmnt* >			8		
< *stmnt_list* >		9		10	
< *stmnt* >		20			18
< *in_id_list* > 13					
< *out_expr_list* >	16				
< *else_part* >		23		23	23
< *expr* >					

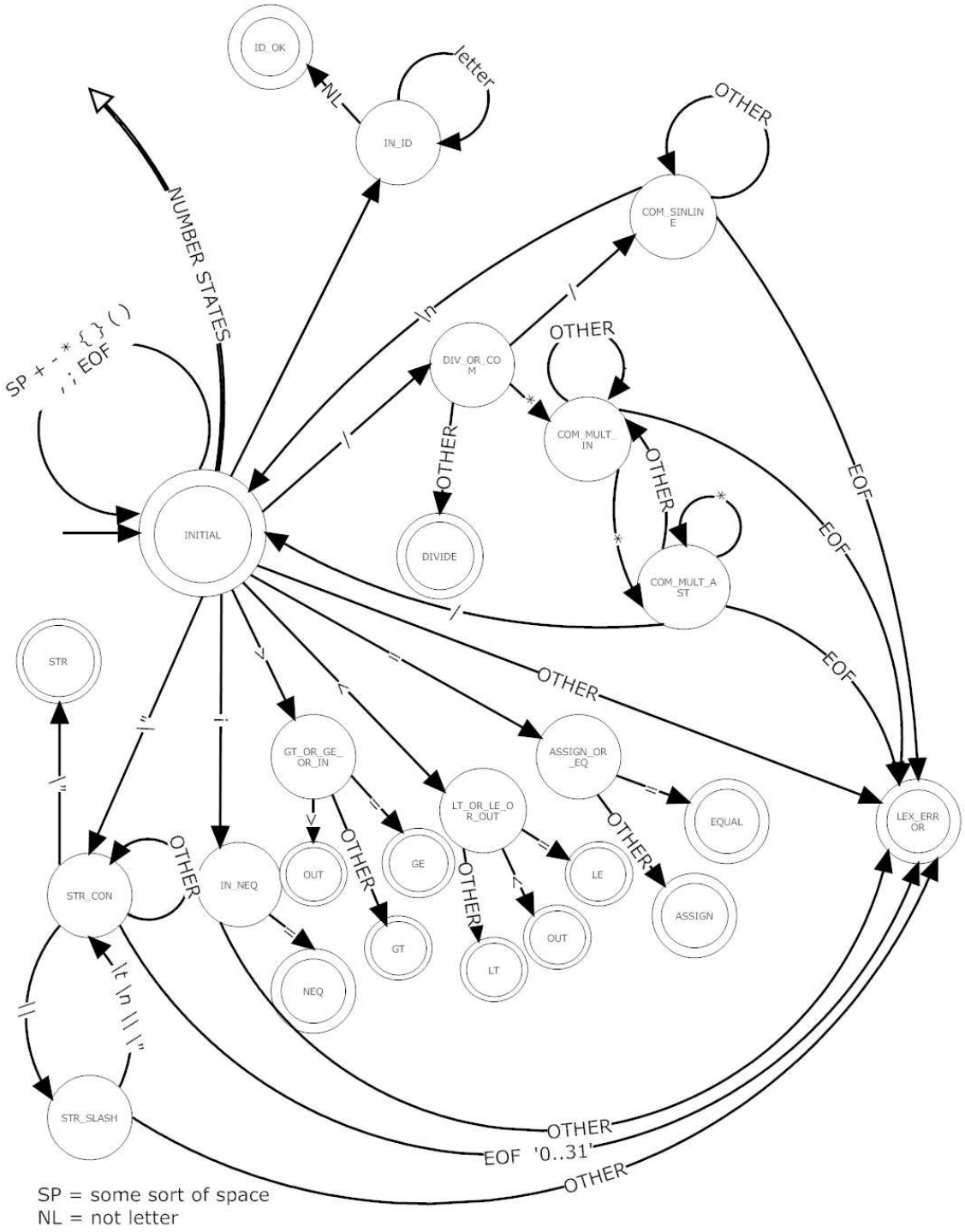
Tabulka 1: LL-tabuľka

	+	-	·	/	<	>	=<	=>	==	!=	=	id	const	func	,	(	)	$
+	>	>	<	<	>	>	>	>	>	>	>	<	<	<	>	<	<	>
-	>	>	<	<	>	>	>	>	>	>	>	<	<	<	>	<	<	>
·	>	>	>	>	>	>	>	>	>	>	>	<	<	<	>	<	>	>
/	>	>	>	>	>	>	>	>	>	>	>	<	<	<	>	<	>	>
<	<	<	<	<	>	>	>	>	>	>	>	<	<	<	>	<	>	>
>	<	<	<	<	>	>	>	>	>	>	>	<	<	<	>	<	>	>
=<	<	<	<	<	>	>	>	>	>	>	>	<	<	<	>	<	>	>
=>	<	<	<	<	>	>	>	>	>	>	>	<	<	<	>	<	>	>
==	<	<	<	<	<	<	<	<	>	>	>	<	<	<	>	<	>	>
!=	<	<	<	<	<	<	<	<	>	>	>	<	<	<	>	<	>	>
=	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	>	>
id	>	>	>	>	>	>	>	>	>	>	>				>		>	>
const	>	>	>	>	>	>	>	>	>	>	>				>		>	>
func																=		
,	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	=	
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	=	
)	>	>	>	>	>	>	>	>	>	>	>				>		>	>
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<		<		

Tabulka 2: Precedenčná tabuľka



Obrázek 1: Graf konečného automatu načítavajúceho číselné konštanty



Obrázek 2: Graf konečného automatu načítavajúceho nečíselné tokeny