VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií

Projektová dokumentácia

Implementácia prekladača jazyka IFJ23

Tím xkruli03, varianta TRP-izp

Boris Hatala (xhatal02) 25%
František Holáň (xholan13) 25%
Michal Krulich (xkruli03) 25%
Stanislav Letaši (xletas00) 25%

6. decembra 2023

Obsah

1 Úvod

Táto dokumentácia obsahuje popis implementácie interpretu jazyka IFJ23, čo je zjednodušená verzia jazyka Swift 5. Swift 5 je staticky typovaný jazyk, ktorý kombinuje prvky objektovo orientovaného, funkcionálneho a imperatívneho programovania.

Nami zvolená varianta je TRP-izp, čo znamená, že tabuľka symbolov je implementovaná ako tabuľka s rozptýlenými položkami s implicitným zreťazením položiek.

1.1 Štruktúra zdrojových súborov

- `decode.h decode.c` - Konvertuje reťazec napísaný v zdrojovom jazyku na reťazec pre IFJcode23
- `dll.h dll.c` - Implementácia dvojsmerného zoznamu pre generované inštrukcie
- `exp.h exp.c` - Precedenčná analýza výrazov s generovaním cieľového kódu
- `generator.h generator.c` - Generátor cieľového kódu
- `logErr.h logErr.c` - Pomocné funkcie pre hlásenie chýb prekladu kódu či samotného prekladača
- `main.c` - Hlavné telo prekladača
- `parser.h parser.c` - Syntaktický a sémantický anayzátor
- `scanner.h scanner.c` - Lexikálny analyzátor
- `strR.h strR.c` - Implementácia reťazca s automatickou realokáciou veľkosti
- `symtable.h symtable.c` - Tabuľka symbolov

2 Implementácia

2.1 Skener

Hlavným cieľom implementácie lexikálnej analýzy je funkcia `getToken()`. Táto funkcia číta jednotlivé znaky zo štandardného vstupu a má za úlohu rozpoznať lexémy zdrojového jazyka a vrátiť príslušné tokeny. Token je reprezentovaný štruktúrou, v ktorej je uložený typ tokenu (rozpoznáva sa celkom 42 rôznych typov tokenov), reťazec načítaného lexému, riadok a stĺpec prečítaného lexému.

2.1.1 Konečný automat

Celý lexikálny analyzátor je implementovaný ako deterministický konečný automat obsahujúci ukončujúce a neukončujúce stavy. Tento automat bol naprogramovaný ako jeden `switch`, kde každé náveštie `case` reprezentuje jeden stav automatu. Okrem hlavného `switch` statementu vo funkcii `getToken()`, je tiež implementovaný podautomat v pomocnej funkcii, ktorý spracováva escape sekvencie v reťazcoch (za účelom sprehľadnenia kódu).

Počas čítania znakov zdrojového kódu zo štandardného vstupu sa prechádza medzi jednotlivými stavmi. Pokiaľ sa automat nachádza v ukončujúcom stave a ďalší načítaný znak už nezodpovedá tokenu, ktorý je reprezentovaný aktuálnym stavom, je vrátený príslušný token. Ak je načítaný znak ktorý nesúhlasí so žiadnym znakom, ktorý by jazyk IFJ23 povoľoval, je vrátený token s hodnotou `0`, značiaci chybu počas lexikálnej analýzy. Rovnako tak sa deje, ak sa nachádza automat v neukončujúcom stave a na vstupe je znak, po prečítaní ktorého by výsledná sekvencia načítaných znakov nezodpovedala žiadnemu typu tokenu.

Špecifická je situácia týkajúca sa rozpoznania identifikátorov a kľúčových slov. Automat nezahŕňa žiadny stav pre kľúčové slová, ale iba stav pre identifikátor. Ak nastane ukončenie čítania znakov v stave identifikátora,

skontroluje sa, či nie je načítaný reťazec (uložený vždy v príslušnom atribúte tokenu) zhodný s niektorým z kľúčových slov jazyka IFJ23. Takto sa rozpozná, či ide o identifikátor alebo o kľúčové slovo.

Okrem funkcie `getToken()` je tiež implementovaná špeciálna funkcia `storeToken(tkn)`, ktorá pri zavolaní uloží token `tkn` do globálnej premennej `storage`.

2.2 StrR

Slúži na uchovanie reťazcov predom neznámej dĺžky. Tento reťazec automaticky realokuje svoju veľkosť vzhľadom na dĺžku vstupného reťazca. Je používaný v každej časti prekladača.

Okrem základných operácií ako inicializácia a deštrukcia reťazca sú implementované aj ďalšie funkcie. Pridanie znaku na koniec dynamického reťazca, zaplnenie reťazca s obsahom klasického reťazca jazyku C, spojenie dvoch dynamických reťazcov alebo spojenie dynamického reťazca s reťazcom jazyka C a vrátenie ukazateľa na dáta v dynamickom reťazci.

2.3 DLL

Na uchovávanie generovaného cieľového kódu je využívaný dvojsmerne viazaný zoznam reťazcov. Pri vkladaní nového reťazca do zoznamu je vytvorený nový prvok zoznamu `DLLstr_element` a alokovaná nová kópia reťazca, ktorá je uložená do tohto zoznamu. Dvojsmernosť zoznamu je hlavne využívaná pri generovaní kódu deklarácii premenných nachádzajúcich sa vo vnútri cyklu, kde musia byť tieto inštrukcie zapísané pred samotný cyklus. Zároveň je táto dátová štruktúra používaná pre uchovávanie názvov a identifikátorov parametrov funkcií.

2.4 Tabuľka symbolov

Tabuľka symbolov je v prekladači implementovaná pomocou zreťazených tabuliek s rozptýlenými položkami s implicitným zreťazením položiek, pričom tieto jednotlivé tabuľky/bloky sú usporiadané vo forme zoznamu. Každý blok tabuľky symbolov `TSBlock_T` obsahuje pole ukazateľov na prvky `TSData_T`, ktoré sú dátovými štruktúrami reprezentujúcimi informácie o premenných alebo funkciach. Každý prvok obsahuje informácie ako názov identifikátora/funkcie, typ (napríklad `i` pre Integer, `F` pre funkciu), informáciu o inicializácii či modifikovateľnosti premennej, prípadne signatúru funkcie `func_sig_T`.

Bloky tabuľky symbolov sú dvojsmerne zreťazené v zozname, kde prvý blok obsahuje informácie o globálnych premenných a funkciách. Metódy nad tabuľkou umožňujú pridávať a odstraňovať ďalšie (lokálne) bloky na konci zoznamu, ktorých význam spočíva v simulácii zanorovania a prekrývania premenných.

Hľadanie a vkladanie symbolov v tabuľke je realizované pomocou dvoch rozptylovacích funkcií, kde prvá (djb2[1]) určuje počiatočný index a druhá (upravený djb2) veľkosť kroku pri hľadaní ďalšieho synonyma. Implementované verejné metódy umožňujú vyhľadávať či vkladať priamo do globálneho alebo posledného lokálneho bloku.

2.5 Parser

Najväčšiu časť prekladača tvorí syntaktický analyzátor (parser), ktorý zároveň vykonáva syntaktickú analýzu, sémantickú analýzu a generovanie cieľového kódu. Parser komunikuje so všetkými podčasťami prekladača a to: získavanie tokenov od skenera, sémantické kontroly s údajmi ukladanými do tabuľky symbolov a generovanie cieľového kódu (`generator.h`).

Syntaktická analýza programu prebieha dvomi rôznymi technikami, ktoré si medzi sebou vymieňajú riadenie. Na začiatku behu prekladača sa spracúva program technikou *top-down* analýzy a pri nájdení začiatku výrazu je riadenie odovzdané *bottom-up* analýze, ktorá končí po zostavení najdlhšieho možného výrazu, prípadne zistení syntaktickej či sémantickej chyby vo vnútri výrazu, a odovzdáva riadenie naspäť. Samotná *top-down* analýza

[1] `http://www.cse.yorku.ca/~oz/hash.html`

je spustená v súbore `main.c` v hlavnom cykle, ktorý spracúva jednotlivé príkazy hlavného tela programu a definície funkcií, a končí načítaním konca programu.

2.5.1 Rekurzívny zostup

Rekurzívny zostup sa vykonáva pomocou funkcií definovaných v `parser.c`, kde každej tejto funkcii náleží jedno pravidlo definovanej LL-gramatiky a nasledujúci zostup je rozhodnutý podľa LL-tabuľky. Špeciálny prístup sa uplatňuje pri volaniach funkcií, keďže rozlíšenie beznávratového volania funkcie od priradenia a volania funkcie od výrazu nie je na základe jedného tokenu možné. V takom prípade je prečítaný ešte jeden token a v prípade nepotvrdenia volania funkcie je tento token vrátený naspäť skeneru.

Návratová hodnota všetkých funkcií rekurzívneho zostupu je v prípade správneho programu `0` alebo v prípade nájdenia chyby rovná číslu chyby, pričom chybová hláška je vypísaná v najnižšom rekurzívnom zanorení. Niektoré z týchto funkcií obsahujú zvláštne parametre, ktoré slúžia pre ďalšie sémantické kontroly a generovanie kódu. Pomocné dáta ako aktuálne načítaný token, ukazateľ na tabuľku symbolov, názov aktuálne spracovávanej funkcie či cyklu, alebo zoznam premenných definovaných v cykle sú používané naprieč všetkými funkciami *top-down* analýzy a preto sú deklarované ako globálne premenné.

Počas rekurzívneho zostupu prebieha aj sémantická analýza kódu, pričom je často využívaná tabuľka symbolov, a to pre kontrolu deklarovaných funkcií a premenných, ich signatúr/dátových typov, informácii, či boli definované/inicializované ale aj simuláciu blokov rozsahu a vnorení.

2.5.2 Precedenčná analýza

Analýza výrazov začína syntaktickou kontrolou. Spracovávaný je každý token výrazu, pričom je uchovávaný aj typ predošlého tokenu. Syntaktická analýza nie je implementovaná ako konečný automat, ale skôr ako „filter“ syntaktických chýb. Počas syntaktickej analýzy prebieha aj konverzia pôvodného výrazu do postfixovej formy, pričom sa z tabuľky symbolov získavajú ďalšie informácie o premenných a kontroluje sa ich stav deklarácie a inicializácie. Syntaktická analýza úspešne končí, ak spracovávaný token nemôže patriť do výrazu, a doteraz spracovaný výraz je syntakticky správny.

Počas sémantickej analýzy sú spracovávané tokeny z postfixového výrazu, prebieha generácia kódu a na pomocnom zásobníku tokenov je vytváraný finálny dátový typ výrazu, ktorý bude po skončení predaný naspäť pre *top-down* analýzu. Sémantická analýza taktiež vykonáva konverziu int konštánt na double, ak je to potrebné, a pre *top-down* analýzu vracia späť informáciu, či je finálny výraz možné implicitne prekonvertovať na double.

2.6 Generovanie cieľového kódu

Cieľové inštrukcie sú generované použitím pomocných funkcií v rozhraní `generator.h`, ktoré sú v rekurzívnom zostupe volané popri syntaktickej analýze a v precedenčnej analýze popri sémantickej analýze. Vygenerovaný kód je uchovávaný v dvoch zoznamoch, kde `code_fn` uchováva inštrukcie užívateľom definovaných funkcií a `code_main` prestavuje hlavné telo programu.

Výsledný vygenerovaný kód je vytlačený na štandardný výstup pomocou funkcie `printOutCompiledCode()`. Táto funkcia pridá potrebnú hlavičku .IFJcode23, tri špeciálne pomocné globálne premenné, inštrkuciu skoku do hlávneho programu a následne vytlačí obsah `code_fn` a `code_main` spolu s náveštím na hlavné telo programu.

2.6.1 Názvy premenných a náveští

Pretože nie je možné v interprete medzikódu IFJcode23 používať dočasný rámec a zásobník lokálnych rámcov ako spôsob zanorovania a prekrývania premenných, tak je každému identifikátoru premennej pomocou funkcie `genUniqVar` pridané unikátne číslo, ktoré zaisťuje rozlíšenie od možných prekrytých premenných. Podobný prístup priraďovania unikátnych čísel je zavedený aj pri náveštiach konštrukcií `while`, `if` a operátorov `??`, kedy je používaná funkcia `genUniqLabel`.

2.6.2 Definície funkcií a ich volanie

Definície užívateľských funkcií začínajú náveštím s názvom funkcie, vytvorením vlastného lokálneho rámca uloženého na vrchol zásobníka rámcov a spracovaním argumentov, ktoré boli predané cez klasický zásobník. Nasleduje samotný kód funkcie, a vždy končí inštrukciou `RETURN`. Ak funkcia vracia nejakú hodnotu, tak táto hodnota je vložená na vrchol zásobníka pred samotným `RETURN`.

Volanie funkcie začína vkladaním argumentov od posledného k prvému (prvý argument bude na vrchole). Následne sa zavolá daná funkcia pomocou inštrukcie `CALL` a zahodí sa lokálny rámec vytvorený funkciou a prípadne sa získa návratová hodnota zo zásobníka alebo sa zásobník vyprázdni pomocou `CLEARS`. Pretože premenné deklarované v hlavnom tele programu sa nachádzajú v globálnom rámci a každé volanie funkcie vytvára nový lokálny rámec na zásobniku rámcov, tak je zabránené akejkoľvek kolízii pri rekurzívnom volaní.

2.6.3 Vstavané funkcie

Väčšina vstavaných funkcií je implementovaných pomocou jednej alebo dvoch inštrukcií a nie je preto potrebné pri ich volaní generovať inštrukciu `CALL` či vytvárať nový lokálny rámec. Výnimku tvorí vstavaná funkcia `substring`, ktorá je natoľko komplexná, že jej vlastný súbor inštrukcií bude dodatočne vložený k bežným užívateľom definovaným funkciám, pokiaľ bola v priebehu programu aspoň raz volaná.

3 Vývojový cyklus

3.1 Práca v tíme

Prácu na projekte sme zahájili úvodnou schôdzou na začiatku októbra. Na tejto schôdzi sme si rozdelili hlavné časti projektu medzi jednotlivých členov tímu, pripravili sme si približný časový harmonogram (stanovili sme si termíny, do kedy chceme mať hotové konkrétne časti projektu) a dohodli sme sa na ďalších záležitostiach potrebných na prácu. Na konkrétnych celkoch sme pracovali väčšinou jednotlivo.

3.2 Vývojové prostredie

Dohodli sme sa, že pri vývoji budeme používať prostredie Visual Studio Code. S týmto vývojovým prostredím sme mali všetci skúsenosti, preto sme túto voľbu považovali za najprijateľnejšiu. Výhodou pri vývoji v tomto prostredí bolo okrem iného prepojenie s GitHub serverom, čo viedlo k jednoduchšej práci s naším repozitárom.

3.3 Repozitár

Na správu súborov sme používali systém Git, pričom ako vzdialený repozitár sme využili GitHub.

3.4 Komunikácia v tíme

Komunikácia prebiehala hlavne prostredníctvom serveru Discord. Poväčšine sme komunikovali v skupinovom chate, aby všetci členovia tímu boli informovaní o vývoji. Nevyhnutnou súčasťou boli aj osobné stretnutia, ktoré sme organizovali vždy pred začiatkom práce na rozsiahlejšom celku, aby sme vyriešili ďalší postup.

3.5 Rozdelenie práce medzi členov tímu

Prácu v tíme sme sa snažili rozdeliť čo najrovnomernejšie. Dohodli sme sa, že každý člen tímu dostane percentuálne hodnotenie 25%. Tabuľka nižšie zobrazuje rozdelenie práce medzi členov tímu.

Boris Hatala	tabuľka symbolov, pomocné funkcie pre generovanie kódu, funkcie pre prácu s řeťazcami, dokumentácia
František Holáň	lexikálna analýza, pomocné funkcie pre generovanie kódu, dokumentácia
Michal Krulich	vedenie tímu, organizácia práce, syntaktická a sémantická analýza – *top-down*, testovanie, generovanie kódu, dokumentácia
Stanislav Letaši	syntaktická a sémantická analýza – *bottom-up*, finálne zpracovanie dokumentácie

A Prílohy

A.1 Pravidlá bezkontextovej gramatiky používanej parserom

1. <STAT> → ϵ
2. <STAT> → let id <DEF_VAR> <STAT>
3. <STAT> → var id <DEF_VAR> <STAT>
4. <DEF_VAR> → : <TYPE> <INIT_VAL>
5. <DEF_VAR> → = <ASSIGN>
6. <INIT_VAL> → = <ASSIGN>
7. <INIT_VAL> → ϵ
8. <ASSIGN> → exp
9. <ASSIGN> → id (<PAR_LIST>)
10. <STAT> → id = <ASSIGN> <STAT>
11. <STAT> → { <STAT> } <STAT>
12. <STAT> → id (<PAR_LIST>) <STAT>
13. <PAR_LIST> → id : term <PAR_IN_NEXT>
14. <PAR_LIST> → term <PAR_IN_NEXT>
15. <PAR_LIST> → ϵ
16. <PAR_IN_NEXT> → , <PAR_IN> <PAR_IN_NEXT>
17. <PAR_IN_NEXT> → ϵ
18. <PAR_IN> → id : term
19. <PAR_IN> → term
20. <STAT> → func id (<FN_SIG>) <FN_RET_TYPE> { <STAT> } <STAT>
21. <FN_SIG> → id id : <TYPE> <FN_PAR_NEXT>
22. <FN_SIG> → _ id : <TYPE> <FN_PAR_NEXT>
23. <FN_SIG> → ϵ
24. <FN_PAR_NEXT> → , <FN_PAR> <FN_PAR_NEXT>
25. <FN_PAR_NEXT> → ϵ
26. <FN_PAR> → id id : <TYPE>
27. <FN_PAR> → _ id : <TYPE>
28. <FN_PAR> → id _ : <TYPE>
29. <FN_PAR> → _ _ : <TYPE>
30. <FN_RET_TYPE> → − > <TYPE>
31. <FN_RET_TYPE> → ϵ
32. <STAT> → return <RET_VAL> <STAT>
33. <RET_VAL> → exp
34. <RET_VAL> → ϵ
35. <STAT> → if <COND> { <STAT> } else { <STAT> } <STAT>
36. <COND> → exp
37. <COND> → let id
38. <STAT> → while exp { <STAT> } <STAT>
39. <TYPE> → Integer <QUESTMARK>
40. <TYPE> → Double <QUESTMARK>
41. <TYPE> → String <QUESTMARK>
42. <QUESTMARK> → ?
43. <QUESTMARK> → ϵ

A.2 Precedenčná tabuľka

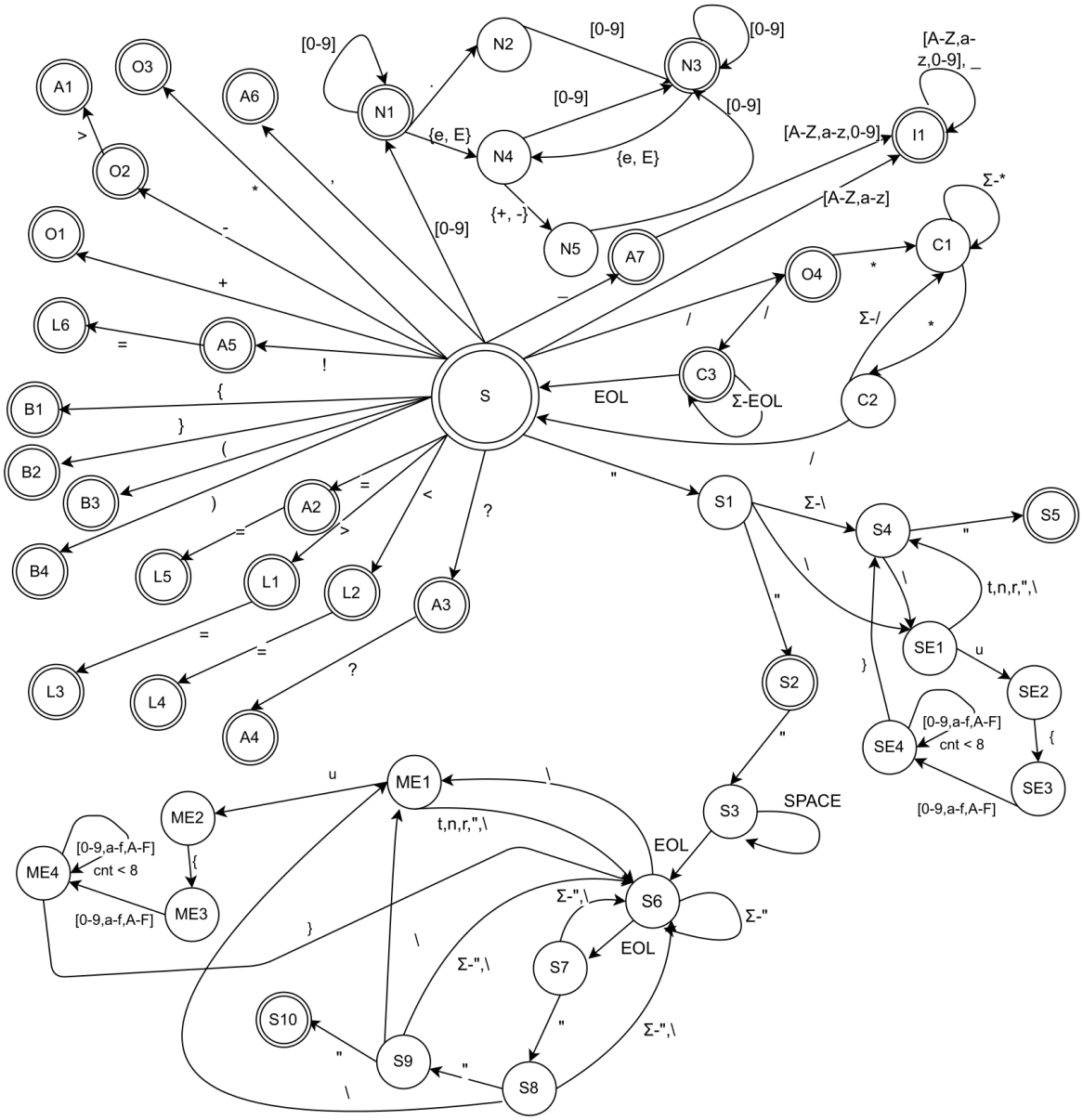
	!	*** /**	**+ -**	**id**	**rel**	**??**	**(**	**)**
!		>	>		>	>		>
*** /**	<	>	>	<	>	>	<	>
+ -	<	<	>	<	>	>	<	>
id	>	>	>		>	>		>
rel	<	<	<	<		>	<	>
??	<	<	<	<	<	<	<	>
(	<	<	<	<	<	<	<	=
)	>	>	>		>	>		>

Tabuľka 1: **id** - identifikátor, **rel** - relačné operátory (==, <=, . . .)

A.3 LL tabuľka

	id	**:**	**=**	**{**	**term**	**return**	**Integer**	**Double**	**String**	**exp**	**func**	**if**	**let**	**var**	**while**	**?**	**_**	**,**	**−>**	**$**
<STAT>	10, 12			11		30					20	35	2	3	38					1
<DEF_VAR>		4	5																	
<INIT_VAL>			6																	7
<ASSIGN>	9									8										
<PAR_LIST>	13				14															15
<PAR_IN_NEXT>																		16		17
<PAR_IN>	18				19															
<FN_SIG>	21																22			23
<FN_PAR_NEXT>																		24		25
<FN_PAR>	26, 28																27, 29			
<FN_RET_TYPE>																			30	31
<RET_VAL>										33										34
<COND>										36			37							
<TYPE>							39	40	41											
<QUESTMARK>																42				43

A.4 Diagram konečného automatu Lexikálnej analýzy



Legenda

S start
O1 plus
O2 minus
O3 mul
O4 div
L1 greater_than
L2 less_than
L3 greater_or_equal
L4 less_or_equal
L5 equal
L6 not_equal
A1 arrow
A2 assign
A3 question_mark
A4 test_nil
A5 exclamation_mark
A6 comma
A7 underscore
N1 number
N2 number_point
N3 number_double
N4 number_exp
N5 number_exp_sign
I1 identificator
S1 string
S2 empty_string
S3 pre_multiline_string
S4 singleline_string
S5 singleline_string_end
S6 multiline_string
S7 multiline_string_end1
S8 multiline_string_end2
S9 multiline_string_end3
S10 multiline_string_end4
SE1 singleline_escape
SE2 singleline_escape_u1
SE3 singleline_escape_u2
SE4 singleline_escape_u_end
ME1 multiline_escape
ME2 multiline_escape_u1
ME3 multiline_escape_u3
ME4 multiline_escape_u_end
C1 comment_block
C2 comment_block_end
C3 singleline_comment

Poznámka

Pro úsporu místa a zpřehlednění, byl u stavů ME4 a SE4 zaveden čítač, který značí, že tento stav by se opakoval celkem 8x