

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Formálne jazyky a prekladače – Projekt

Prekladač pre jazyk IFJ24

Tím xluptas00 varianta TRP-izp

Bodové rozdelenie: 30/25/20/25%
Implementované rozšírenia: žiadne

1. decembra 2024

Vedúci: Samuel Lupták (xluptas00)
Petr Němec (xnemecp00)
Lukáš Houzar (xhouzal00)
Mário Klopán (xklopam00)

1 Návrh

Prekladač sa skladá z 3 hlavných častí a 4 pomocných dátových štruktúr.

Hlavné časti prekladača (a ich podčasti):

- Lexikálny analyzátor ¹
- Dvojprechodový syntaktický a sématický analyzátor ²
 - Analýza kódu pomocou rekurzívneho zostupu
 - Analýza výrazov pomocou precedenčnej analýzy
- Generátor výsledného kódu

Pomocné štruktúry použité v prekladači:

- Tabuľka s rozptýlenými položkami s implicitným zret'azením položiek ³
- Abstraktný syntaktický strom ⁴
- Zásobník
- Fronta

Využitie jednotlivých štruktúr je nasledovné:

Hašovacia tabuľka: Bola použitá pre implementáciu tabuľky symbolov. Podmienka pre implicitné zret'azenie nám robila mierny problém, pretože teoreticky nekonečný počet identifikátorov sa nevmetí do konečne veľkej tabuľky

ASS: Slúži na komunikáciu medzi parserom a generátorom kódu

Zásobník: Je využitý precedenčnou analýzou, ktorá ho používa na spracovanie výrazov

Fronta: Má význam pri dvojprechodovej analýze ako úložisko tokenov. Pre dvojprechodovú analýzu sme sa rozhodli po zistení, že definícia funkcie nemusí lexikálne predchádzať jej volaniu.

¹Ďalej len *skener*

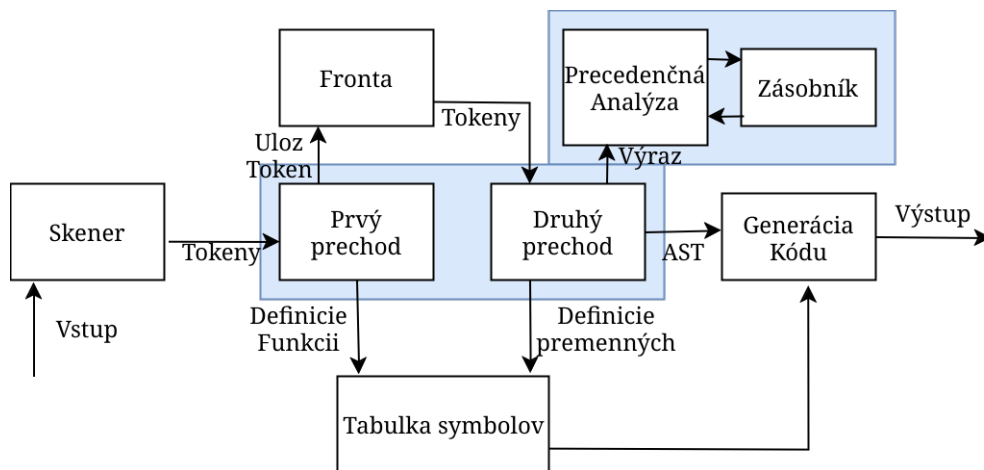
²Ďalej len *parser*

³Ďalej len *hašovacia tabuľka*

⁴Ďalej len *ASS*

2 Popis komunikácie

2.1 Diagram



Obr. 1: Diagram komunikácie

2.2 Stručný popis

Parser (modrá časť diagramu) inicializuje všetky operácie v prekladači. Preklad začína vložení programu v jazyku zig na vstup. Skener (Na zavolanie) postupne fragmentuje vstup na jednotlivé lexémy a posiela ich vo forme tokenov do parseru. Ako bolo už spomenuté, parser je dvojprechodový. Tokeny idú najprv cez prvý prechod, ktorý kontroluje syntax a sématicku iba pre hlavičky funkcií, ktoré následne ukladá do tabulky symbolov, aby informácie o nich boli dostupné v druhom prechode. Prvý prechod ukladá všetky prečítané tokeny do fronty. Z fronty si tokeny po jednom berie druhý prechod, ktorý kontroluje syntax a sématicku pre ostatok kódu. V prípade že v kóde sa nachádza výraz, zavolá sa precedenčná analýza, ktorá tento výraz pomocou zásobníka spracuje. Druhý prechod zároveň pridáva definované premenné do tabuľky symbolov (samotnú tabuľku symbolov však aj využíva, napr. pre kontrolu redefinície). Počas druhého prechodu sa zároveň vytvára ASS, ktorý po úspešnom dokončení analýzy slúži ako výstup a zároveň vstup do generátora kódu. Generátor kódu s pomocou tabuľky symbolou generuje cieľový kód.

3 Implementácia

Skener: *lexer.**, *token.h* (xhouzal00, xnemecp00)

Parser: *syntax.**, *queue_fill.**, *precedence.** (xluptas00, xklopam00)

Generátor kódu: *code_gen.** (xhouzal00, xnemecp00)

Hašovacia tabuľka: *syntable.** (xluptas00)

ASS: *tree.** (xklopam00)

Zásobník: *stack.** (xklopam00)

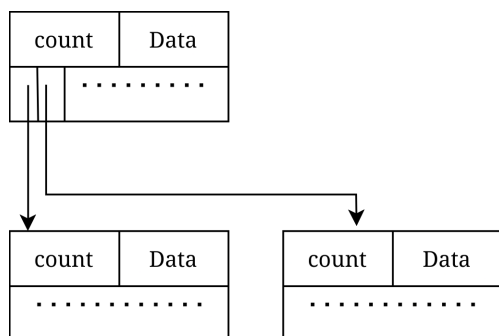
Fronta: *queue.** (xluptas00)

Ostatné časti: *error.** (xluptas00)

Implementácia dátových štruktúr sa nachádza v jednotlivých súboroch pomenovaných podľa danej štruktúry.

Implementácia častí samotného prekladača spočíva v súboroch skenera, parsera, generátora a súboru *error.**, ktorý implementuje základné pracovanie s chybami. Prvý prechod je implementovaný v súboroch *queue_fill.** a druhý prechod je implementovaný v súboroch *syntax.**. *syntax.c* zároveň obsahuje funkciu *Main*. Prílohy A, B, C, D ukazujú využitú teóriu, ktorá slúžila ako podklad pre jednotlivé časti prekladača.

3.1 Strom



Obr. 2: Štruktúra stromu

Pre kompletnosť sme sa rozhodli vizualizovať ASS, ktorý sme navrhli pre tento prekladač. Ako je vidno na diagrame, tak každý uzol obsahuje 3 hlavné časti a to sú: *count*, *data*, *children*, kde *count* určuje počet detí, ktoré daný uzol má. Hlavná časť, *data*, v sebe uchováva dáta potrebné na správne generovanie kódu (viac vid'. *tree.**). Posledná časť *children* je pole ukazovateľov na deti. V texte ďalej ukážeme ako sa "kódujú" jednotlivé časti kódu do tohto stromu.

Typy uzlov sú špecifikované v *tree.h:15*. Strom má presne 1 koreň typu **ROOT_NODE** a má presne toľko detí, koľko je funkcií vo vstupnom programe. Tieto uzly sú typu **TOP_FUNCTION_NODE** a sú v nich uložené základné informácie o funkciách potrebné pre generáciu kódu. Z jednotlivých uzlov funkcií vychádzajú uzly špecifikujúce jednotlivé časti kódu. Tieto časti kódu sa delia na: *priradenie*, *definíciu premennej*, *návrat*, *vetvenie*, *cyklus*, *volanie funkcie*.

Priradenie začína uzlom **ASSIGN_NODE**, jeho prvé dieťa určuje do akej premennej sa priradí a jeho druhé dieťa určuje čo sa priradí (funkcia alebo výraz).

Definícia premennej má ten istý tvar ako *priradenie* až na to, že vrchný uzol má typ **DEFINITION_NODE**

Návrat začína uzlom typu **RETURN_NODE**. Jeho jediným dieťaťom je strom výrazu, v prípade funkcie nevracajúcej hodnotu, nemá deti žiadne.

Vetvenie a *cyklus* začínajú uzlom typu **WHILE_NODE** alebo **IF_NODE**. Na prvom mieste sa na-

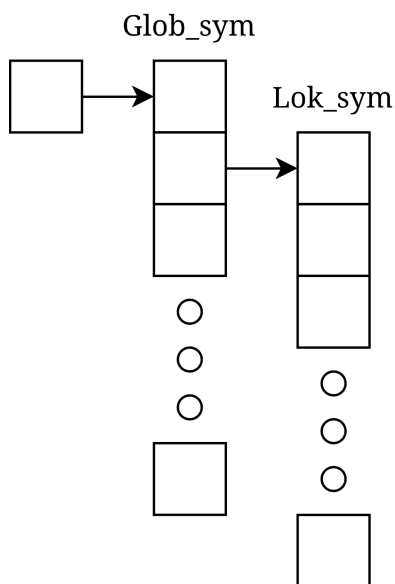
chádza podmienka. Nasleduje postupnosť častí kódu. Pokiaľ sa jedná o podmienku, začiatok druhej vetvy označuje dieťa typu **ELSE_NODE** nasledovaná postupnosťou častí kódu.

Volanie funkcie začína uzlom **FUNCTION_NODE** a jeho deti sú postupnosť argumentov. Podobne fungujú aj vstavané funkcie s miernou zmenou typu hlavného uzla.

Zaujímavým doplnkom stromu je inklúzia spätného odkazu na otca v každom uzle. Toto umožňuje celkom elegantné zanorovanie a vynorovanie v rekurzívnom zostupe. Pokiaľ je napríklad volaná funkcia, program sa zanoří do uzlu **FUNCTION_NODE** a zaplní ho uzlami argumentov. Po jeho spracovaní sa pomocou spätného odkazu program vynorí z tohto uzlu a je pripravený spracovať ďalšie riadky kódu.

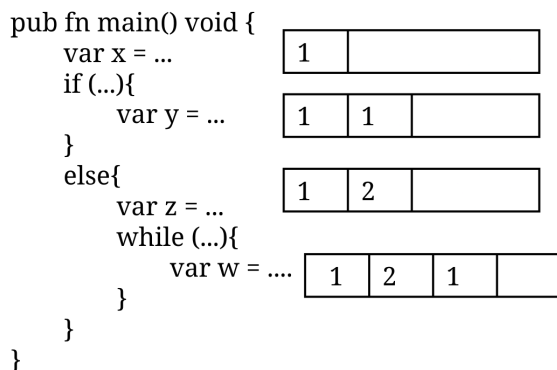
Generácia kódu následne generuje kód jednoduchým prechádzaním tohto stromu.

3.2 Tabuľka symbolov



Obr. 3: Tabuľka symbolov

Tabuľku symbolov sme implementovali podľa zadania ako hashovaciu tabuľku. V prekladači existuje ukazovateľ na *globálnu tabuľku* v ktorej sú záznamy definovaných funkcií. Každá funkcia má v sebe odkaz na *lokálnu tabuľku* symbolov. Lokálna tabuľka symbolov, obsahuje záznamy premenných v daných funkciách. Parametre funkcie sú na začiatku tiež interpretované ako lokálne premenné. Zaujímavosťou je určite riešenie rozsahu platnosti lokálnych premenných. Namiesto riešenia zásobníka tabuliek symbolov sme vymysleli jednoduchý systém, kde každá premenná má svoj vlastný zásobník čísel. Tento zásobník určuje rozsah premennej následovne: premenné definované v tele hlavnej funkcie majú veľkosť zásobníku 1 a na zásobníku je číslica 1. Pokiaľ definujeme premennú v nejakom podbloku, na zásobník vložíme hodnotu, ktorá je jedinečná pre daný podblok (ilustrujeme obrázkom). Tabuľka je kvôli požadovanému implicitnému zret'azaniu obmedzená na 1000 položiek, čo si uvedomujeme, že nie je najvhodnejšie riešenie, avšak veríme, že pre projekt bolo dostačujúce.



Obr. 4: Riešenie rozsahu platnosti

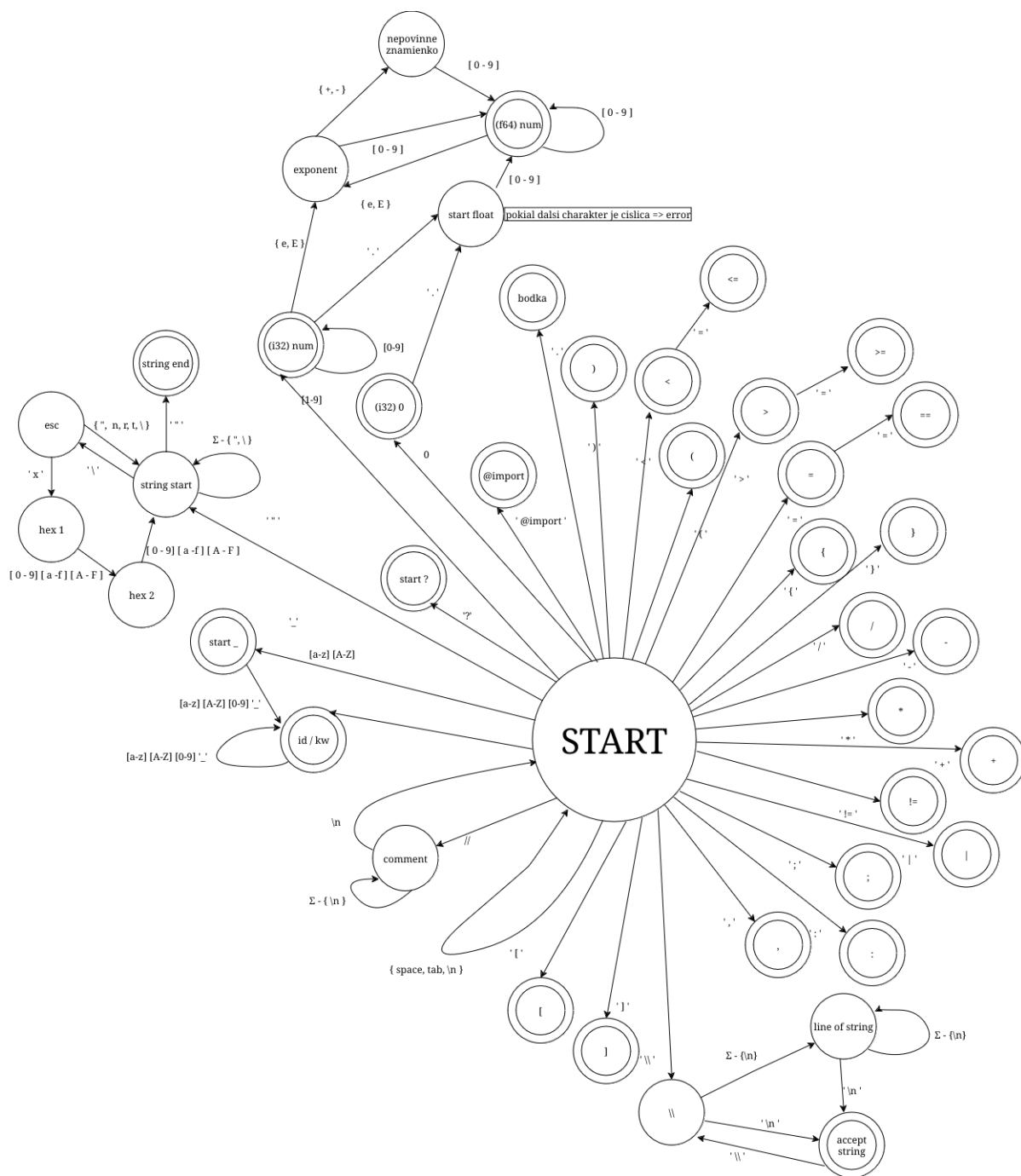
3.3 Implementácia ostatných častí

Ostatné časti (skener, parser, generácia kódu, zásobník, fronta) sú implementované štandardne metódami prednášanými na FIT VUT, preto ich špecifikáciu vynecháme.

4 Práca v tíme

Na projekte sme pracovali počas celého semestra. Na správu verzií sme využívali platformu GitHub, ktorá nám umožnila prehľadnú organizáciu zdrojového kódu, sledovanie zmien a jednoduché riešenie prípadných konfliktov v kóde. Vďaka tomu mal každý člen tímu vždy prístup k aktuálnej verzii projektu. Na komunikáciu sme používali Discord, ktorý nám poskytol priestor na rýchlu výmenu informácií, plánovanie úloh a riešenie problémov v reálnom čase. Táto platforma bola užitočná najmä pri každodennom zdieľaní poznatkov alebo konzultáciách ohľadom implementácie jednotlivých častí projektu. Okrem online komunikácie sme sa raz týždenne stretávali osobne, aby sme spoločne zhodnotili doterajší pokrok, stanovili priority na nadchádzajúce obdobie a riešili zložitejšie časti projektu, ktoré si vyžadovali detailnú diskusiu. Tento pravidelný rytmus spolupráce zabezpečil plynulý vývoj projektu a pomohol nám efektívne rozdeliť prácu medzi jednotlivých členov tímu. Navyše sme sa rozdelili do tímov po dvoch podľa toho, ako sme ubytovaní na internátoch, čo nám umožnilo ešte rýchlejšiu komunikáciu pri problémoch špecifických pre jednotlivé tímy. Bodové rozdelenie odráža čas vložený do projektu jednotlivými členmi tímu.

5 PRÍLOHA A - Konečný automat



6 PRÍLOHA B - LL1 gramatika

<xxx> - xxx je neterminál

'yyy' - yyy je terminál

Pravidlá tokenov

1. <ifj> -> 'IFJ'
2. <at_import> -> 'IMPORT'
3. <eq> -> 'ASSIGN'
4. <comma> -> 'COMMA'
5. <colon> -> 'COLON'
6. <op_bracket> -> 'OPENING_BRACKET'
7. <cl_bracket> -> 'CLOSING_BRACKET'
8. <op_cr_bracket> -> 'OPENING_CURLY_BRACKET'
9. <cl_cr_bracket> -> 'CLOSING_CURLY_BRACKET'
10. <op_sq_bracket> -> 'OPENING_SQUARE_BRACKET'
11. <cl_sq_bracket> -> 'CLOSING_SQUARE_BRACKET'
12. <string_prolog> -> 'STRING' (tento string sa lisi od pravidla <string> tym ze musi nadobudat hodnotu "ifj24.zig")
13. <semicolon> -> 'SEMICOLON'
14. <pub> -> 'PUB'
15. <fn> -> 'FN'
16. <dot> -> 'DOT'
17. <return> -> 'RETURN'
18. <if> -> 'IF'
19. <else> -> 'ELSE'
20. <vertical_bar> -> 'VERTICAL_BAR'
21. <while> -> 'WHILE'
22. <eof> -> 'END_OF_FILE'
23. <const> -> 'CONST'
24. <var> -> 'VAR'
25. <void> -> 'VOID'
26. <type_keyword> -> <op_sq_bracket> <cl_sq_bracket> 'U8'
27. <type_keyword> -> 'F64'
28. <type_keyword> -> 'I32'
29. <string> -> 'STRING'
30. <int> -> 'I32_VAR'
31. <float> -> 'F64_VAR'
32. <null> -> 'NULL_VALUE'
33. <id> -> 'ID'
34. <id_ifj> -> 'ID' (id.val sa musi zhodovat s nazvom neakej vstavanej funkcie)
35. <underline> -> 'UNDERSCORE'
36. <nullable> -> 'NULLABLE'
37. <nullable> -> ϵ

Výrazi

-
- 38. <var_exp> -> <ifj_call>
 - 39. <var_exp> -> <fn_call>
 - 40. <var_exp> -> <exp> <semicolon>
 - 41. <null_exp> -> <exp> <cl_bracket>
 - 42. <null_exp> -> <id> <cl_bracket> <vertical_bar> <id> <vertical_bar> // zober 2 tokeny, pokiaľ prvý je ID a druhý je cl_bracket, vol túto variantu, inak vol druhú
 - 43. <return_exp> -> ϵ
 - 44. <return_exp> -> <exp>
 - 45. <exp> -> // call precedence analyzer

Pomocné pravidlá

- 46. <type> -> <nullable> <type_keyword>
- 47. <var_const> -> <var>
- 48. <var_const> -> <const>
- 49. <type_fndef> -> <type>
- 50. <type_fndef> -> <void>
- 51. <type_vardef> -> <colon> <type>
- 52. <type_vardef> -> ϵ
- 53. <id_assign> -> <underline>
- 54. <id_assing> -> <id>

Hlavné pravidlá

- 55. <call_params> -> ϵ
- 56. <call_params> -> <string> <comma> <call_params>
- 57. <call_params> -> <int> <comma> <call_params>
- 58. <call_params> -> <float> <comma> <call_params>
- 59. <call_params> -> <null> <comma> <call_params>
- 60. <call_params> -> <id> <comma> <call_params>
- 61. <if_while_body> -> <op_cr_bracket> <fn_body> <cl_cr_bracket>
- 62. <var_def> -> <var_const> <id> <type_vardef> <eq> <var_exp>
- 63. <assign> -> <id_assign> <eq> <var_exp>
- 64. <fn_call> -> <id> <op_bracket> <call_params> <cl_bracket> <semicolon>
- 65. <ifj_call> -> <ifj> <dot> <ifj_id> <op_bracket> <call_params> <cl_bracket> <semicolon>
- 66. <if_else> -> <if> <op_bracket> <null_exp> <if_while_body> <else> <if_while_body>
- 67. <cycle> -> <while> <op_bracket> <null_exp> <if_while_body>
- 68. <fn_return> -> <return> <return_exp> <semicolon>
- 69. <fn_body> -> <var_def> <fn_body>
- 70. <fn_body> -> <assign> <fn_body> // nahliadnutie do tabuľky symbolov, pokiaľ je ID variable, tak sa volá táto varianta
- 71. <fn_body> -> <fn_call> <fn_body> // nahliadnutie do tabuľky symbolov, pokiaľ je ID function alebo nedefinované, tak sa volá táto varianta
- 72. <fn_body> -> <ifj_call> <fn_body>
- 73. <fn_body> -> <if_else> <fn_body>
- 74. <fn_body> -> <cycle> <fn_body>

- 75. $\langle \text{fn_body} \rangle \rightarrow \langle \text{fn_return} \rangle \langle \text{fn_body} \rangle$
- 76. $\langle \text{fn_body} \rangle \rightarrow \epsilon$
- 77. $\langle \text{params} \rangle \rightarrow \langle \text{id} \rangle \langle \text{colon} \rangle \langle \text{type} \rangle \langle \text{comma} \rangle \langle \text{params} \rangle$
- 78. $\langle \text{params} \rangle \rightarrow \epsilon$
- 79. $\langle \text{fn_def} \rangle \rightarrow \langle \text{pub} \rangle \langle \text{fn} \rangle \langle \text{id} \rangle \langle \text{op_bracket} \rangle \langle \text{params} \rangle \langle \text{cl_bracket} \rangle \langle \text{type_fndef} \rangle$
- 80. $\langle \text{functions} \rangle \rightarrow \langle \text{fn_def} \rangle \langle \text{op_cr_bracket} \rangle \langle \text{fn_body} \rangle \langle \text{cl_cr_bracket} \rangle \langle \text{functions} \rangle$
- 81. $\langle \text{functions} \rangle \rightarrow \langle \text{eof} \rangle$
- 82. $\langle \text{import_def} \rangle \rightarrow \langle \text{const} \rangle \langle \text{ifj} \rangle \langle \text{eq} \rangle \langle \text{at_import} \rangle \langle \text{op_bracket} \rangle \langle \text{string_prolog} \rangle \langle \text{cl_bracket} \rangle \langle \text{semicolon} \rangle$
- 83. $\langle \text{start} \rangle \rightarrow \langle \text{import_def} \rangle \langle \text{functions} \rangle$

7 PRÍLOHA C - LL1 tabulka

[illegible]

[illegible]

8

[illegible]