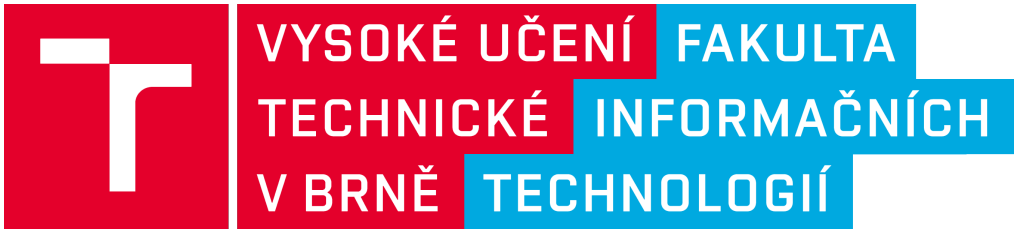
Vysoké učení technické v Brně
Fakulta informačních technologií



Dokumentace k projektu do předmětu IFJ a IAL

Implementace překladače jazyka IFJ24

Tým xhubacv00, varianta - TRP-izp

Vedoucí týmu:	Vojtěch Hubáček	xhubacv00	25%
Členové týmu:	Jakub Ramašeuski	xramas01	25%
	Jozef Ondrejička	xondre16	25%
	Lukáš Šidlík	xsidlil00	25%

4. prosince 2024

Obsah

1 Úvod

Cílem projektu bylo vytvořit překladač, který načte cílový kód jazyka IFJ24, který je podmnožinou programovacího jazyka Zig a přeloží ho do cílového jazyka IFJcode24.

V případě úspěchu překladu se na standardní výstup vypíše cílový kód jazyku IFJcode24, jinak program skončí s daným číslem chyby a chybovou hláškou.

2 Implementace překladače IFJ24

V této kapitole jsou rozepsané dílčí části programu a komunikace mezi nimi.

2.1 Lexikální analýza

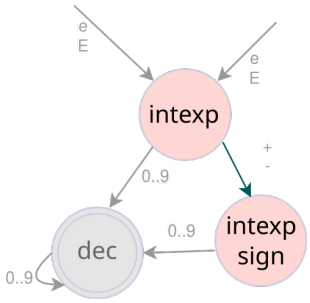
Skener se snaží co nejvíc přiblížit návrhu, je tedy zaveden jako orientovaný graf pod jmennými prostory `scn (SCaNner)` a `sca (SCanner Auxiliary)`, z nichž `scn` je využit zejména pro komunikaci skeneru ven a sca pro jeho konfiguraci.

V kódu `sca` tím vzniklo rozhraní pro konfiguraci orientovaného grafu, které dovoluje deklaraci hran s pomocí `SCA_PATH_DECL(src, dest)`, definici s `SCA_PATH_DEF(src, dest)` inicializaci s `SCA_PATH_INIT (name, args...)`, kterýmž lze dosadit hraně se jménem name dosadit disjunktní podmínky `(args)`, tedy stačí, aby z N podmínek platila jedna pro zvolení dané cesty, zmínění hrany `SCA_PATH(src, dest)` a deiniciaci orientovaných hran mezi uzly, které jsou sice deklarovány globálně, nicméně jim jsou hrany dosazeny až v rámci běhu programu za pomocí funkce `sca_assign_children(Scan_node_ptr node, int argc, ...)` s variabilním počtem argumentů.

Celý běh skeneru tím lze soustředit do funkce `scn_scan`, jenž iterativně prochází řetězec pomocí deterministického konečného automatu viz(A), dokud nenarazí na situaci, kdy pro načtený symbol nevede z daného uzlu přechod. Tehdy je načtený řetězec nejdříve vyhodnocen dle toho, zda se automat zastavil na terminálu definovaném v tabulce `sca_translation_table[]`, pokud ano, na jakém a vyhodí příslušný `token`, v opačném případě skener vyhodí chybu a ukončí program. V případě setkání se se symbolem konce řetězce `EOF` zastavuje načítání řetězce a v případě požadavku na další token jej pro usnadnění práce načte z dvousměrně vázaného seznamu.

2.1.1 Příklad užití rozhraní lexikální analýzy

Zde jsou uvedeny příklady užití. Pro tento příklad uvažujme existenci hrany mezi uzlem, který v číslu představuje exponent (`sca_intexp`) a znaménko (`sca_intexpsign`). Průchod přes hranu je tedy dovolen pouze pokud je načtený symbol plus nebo minus.



Obrázek 1: uváděné uzly a hrany (barevně)

- `SCA_PATH_DEF(sca_intexp, sca_intexpsign)`
  - tímto je definována hrana mezi uzlem `sca_intexp` a `sca_intexpsign`
- `SCA_PATH_INIT(SCA_PATH(sca_intexp, sca_intexpsign), SCA_MATCH(plus), SCA_MATCH(minus))`
  - tímto jsou za běhu hraně dosazeny podmínky požadující znaménka plus nebo minus
- `sca_assign_children(&sca_intexp, 2, &SCA_PATH (sca_intexp, sca_dec2), &SCA_PATH(sca_intexp, sca_intexpsign));`
  - tímto jsou uzlu dosazeny hrany, které z něj vedou. Z uzlu tímto lze přejít do jiného s jiným stavem, zároveň je uzel zodpovědný za následnou deiniciaci hran při ukončení programu. V našem případu to znamená, že z uzlu vedou hrany do uzlů `sca_intexpsign` a terminálního `sca_dec2`

2.2 Syntaktická analýza

Syntaktická analýza v tomto projektu je implementována jako dvoufázový proces, který zajišťuje správné zpracování zdrojového kódu a přípravu na generování výsledného kódu. Naše syntaktická analýza byla implementována pomocí metody rekurzivního sestupu založeného na LL-gramatice (vše kromě výrazů), viz (B), což umožňuje přehledné a systematické zpracování jednotlivých syntaktických konstrukcí.

První fáze, realizovaná funkcí `parse_fn_first`, provádí základní syntaktickou kontrolu zdrojového kódu, identifikuje deklarace funkcí, jejich parametry a ukládá je do tabulky symbolů. Druhá fáze, která začíná inicializací zásobníku a řetězců pro generování kódu, spouští syntaktickou analýzu z neterminálu `<program>` pomocí funkce program a postupně zpracovává jednotlivé části programu, jako jsou deklarace, parametry, tělo funkcí a příkazy.

Syntaktická analýza zahrnuje hlavně práci s neterminály, jako například `<prolog>`, `<function>`, `<body>`, `<statement>` či `<call>`. Tento dvoufázový přístup zaručuje efektivní sémantickou kontrolu a generování správného výstupního kódu, což tvoří základ pro další zpracování výrazů, ve kterých se volají funkce a generování tříadresného kódu.

2.2.1 Zpracování výrazů

Syntaktická analýza výrazu je v souboru `parser.c` řízena precedenční analýzou, k níž je vytvořena odpovídající precedenční tabulka. Syntaktická analýza přistupuje v podstatě ke všemu jako k výrazu, přestože se může skládat například jen z jednoho literálu. Precedenční analýza je řízena operacemi posuvu `'<'`, pouhého vložení `'='`, prázdné operace `' '` a redukce `'>'` s použitím odpovídajících pravidel. ($\texttt{E} \rightarrow \texttt{ID}$, $\texttt{E} \rightarrow \texttt{(E)}$,$\texttt{E} \rightarrow \texttt{!E}$, $\texttt{E} \rightarrow \texttt{E.?}$, $\texttt{E} \rightarrow \texttt{E}\{+|-|*|/|orelse|==|!=|<|>|<=|>=|and|or\}\texttt{E}$). Současně s precedenční analýzou je vytvářena postfixová notace výrazu pro kontrolu sémantickou analýzou a pro následnou generaci cílového kódu.

2.3 Sémantická analýza

Se syntaktickou analýzou je v `parser.c` zároveň implementována analýza sémantická. Ve struktuře `parser_tools_t`, která je definována v `parser.h` jsou uloženy všechny struktury a proměnné, které jsou nezbytné pro sémantickou kontrolu.

Při prvním průchodu se naplní symbolická tabulka viz(3.2) funkcí, jelikož v jazyce IFJ24 nezáleží jestli je volaná funkce definována nad nebo pod jejím voláním. V tomhle průchodu se zkontroluje i redefinice funkcí.

Při druhém průchodu už se provádí všechny zbylé sémantické kontroly v tělech funkcí, a plnění symbolický tabulek proměnných, které jsou uloženy v dvousměrně vázaném seznamu (ten je implementovaný `symDLList.c`), který se nachází v `perser_tools_t`. U každého paměťovém bloku (`if, while, else`) se vytvoří nová symbolická tabulka a vloží se na konec obousměrně vázaného seznamu. Po odejití z bloku se zkontroluje použití všech proměnných, které byly v daném bloku definovány a následně se tabulka zničí. Na konci sémantické analýzy se zkontroluje jestli existuje funkce `main`.

Kontrola datové kompatibility ve výrazech se řeší v `expression.c`, kde se odvodí datový typ výsledku a zkontrolují validní implicitní konverze. Dále se podle kontextu zkontroluje, jestli daný výsledek má správný datový typ, vůči čemu byl volán `(if, return)` nebo přiřazení.

2.4 Generování cílového kódu

Generace kódu probíhá přímo v souboru `parser.c` při druhém průchodu parseru zdrojovým kódem. Všechny potřebné funkce, formáty instrukcí a další pomocné konstrukce pro generování cílového kódu jsou umístěny v souboru `codegen.c` a `codegen.h`.

2.4.1 Generace instrukcí

Pro instrukce bylo v souboru `codegen.c` předpřipraveno pole formátu instrukcí cílového jazyka pro bezpečnější práci s instrukcemi [2]. Vytvoření jednoznačných proměnných a návěští je řešeno pomocí počítadel dostupných v nástrojích parseru určených pro generátor kódu.

2.4.2 Průběžná generace celkového kódu

K výpisu jednotlivých instrukcí zdrojového kódu se souboru `parser.c` využívá předpřipravené makro `printi`. Toto makro přijímá formátovaný řetězec s již danou instrukcí a argumenty pro formátovaný řetězec, jimiž jsou hodnoty užívané instrukcí. Cílový kód programu je následně makrem vypisován v závislosti na typu instrukce do dynamických řetězců typu `str_t` nacházejících se v nástrojích parseru.

Tedy takřka všechny instrukce jsou zapisovány do dočasného řetězce, až na instrukci definice proměnné `DEFVAR`, která je přednostně zapisována do hlavního řetězce instrukcí za vstupem do dané funkce (jelikož definice proměnných nelze pro provádět uvnitř cyklů, kvůli redefinici proměnných). Na konci každé funkce je hlavní a dočasný řetězec sjednocen.

Vestavěné funkce a pomocné funkce zpracování výrazů jsou generovány na konci průchodu parseru zdrojovým programem. Na závěr v případě, že je zdrojový kód korektní, tak se cílový kód vypíše na standardní výstup.

2.4.3 Generace výrazů

Výrazy jsou z postfixové notace, vytvářené během precedenční analýzy, generovány a zpracovávány operacemi nad zásobníkem. Pro operace, které umožňují přetypování operandů, jsou přegenerovány funkce, které umožňují případné přetypování operandů uskutečnit. Předpřipravené funkce jsou nachystány i pro operace, pro něž neexistuje ekvivalentní instrukce zdrojového kódu. Vyskytuje-li se ve výrazu funkce, je již během vytváření postfixové notace proveden úkon, kdy je přednostně vygenerováno volání dané funkce a ve výsledném výrazu se tak nachází pouze návratové hodnoty dané funkce.

2.5 Chování při chybě a hlídání paměti

Pro odchytávání chyb je postavena sada funkcí pro jejich zpracování, každý chybový výstup má vlastní volatelnou funkci či makro, které ukončí program. Projekt představuje tabulku alokovaných zdrojů ve formě globální hašovací tabulky o velikosti 9973 a s explicitně zřetězenými prvky, současně i wrapper okolo funkcí `malloc()`, `realloc()` a `free()` → `imalloc()`, `irealloc()` a `ifree()`.

Tyto funkce za běžné situace fungují naprosto identicky jako funkce, jež obalují, nicméně při iniciaci `memory_ht_init(&_memory_table);` a nastavení globální proměnné `safe_memory` na `true` dochází k interakci s tabulkou alokovaných zdrojů, tedy při alokaci či dealokaci dojde přidání ukazatele na zdroj do tabulky zdrojů, respektive jeho odebrání a následné uvolnění. Funkce, jež požaduje neočekávané ukončení programu tedy nejdříve zavolá `memory_ht_dispose(&_memory_table);`, čímž uvolní zdroje alokované přes `imalloc()`, načež ukončí chod programu s požadovaným návratovým kódem.

V kapitole datových struktur je letmo zmíněno, že jednosměrně vázaný seznam jako jediná ze struktur dovoluje alokaci přes `malloc()`. Tato vlastnost je nezbytná k validnímu fungování tabulky, neboť by jinak přidání jediného ukazatele do tabulky ukazatelů způsobilo řetězovou reakci v podobě rekurzivního přidávání prvků bez žádného vrchního limitu mimo pád překladače.

2.5.1 Omezení funkcí pracujících s pamětí

V této pasáži je nezbytné upozornit, že dané obalové funkce přicházejí i se svými výzvami – na jedné straně jsou sice určené k tomu, aby zakládaly ukazatele na alokované zdroje, nicméně pro ně platí omezení ve formě nekombinovatelnosti s tradičními funkcemi na alokaci – uvolnění zdroje získaného přes `malloc()` s pomocí `ifree()` vede k úniku paměti, neboť je nejprve kontrolována existence ukazatele v tabulce ukazatelů, naopak při dealokaci zdroje alokovaného funkcí `imalloc()` s pomocí `free()` dojde k dvojitému uvolnění dané paměti, neboť ukazatel na ni nadále přebývá v tabulce ukazatelů.

3 Datové struktury

V této kapitole jsou speciální datové struktury, které jsme implementovaly a převážná inspirace je z přednášek ze předmětu IAL.

3.1 Vázané seznamy

S pomocí maker byly vytvořeny jednosměrně a dvousměrně vázané seznamy. Vzhledem k jejich trivialitě k nim nebude napsáno tolik, mimo rozdíl mezi nimi, možnosti alokace - dvousměrně vázané seznamy nenabízí možnost přímé alokace zdrojů s pomocí `malloc()`, u seznamu jednosměrného tak muselo býti učiněno z důvodů vyplývajících z jeho užití v tabulce ukazatelů alokovaných zdrojů.

3.2 Tabulka s rozptýlenými položkami

Tato datová struktura je zde využita ve dvou formách - s implicitním a explicitním zřetězením.

3.2.1 Tabulka s rozptýlenými položkami s implicitním zřetězením položek

Tato struktura je použita pro tabulky symbolů. Její největší výhodou je rychlost nalezení daného symbolu. Základem struktury je pole ukazatelů na položky o velikosti 4001, která zaručuje dostatečné rozptýlení položek. Každá položka obsahuje klíč, ukazatel na data a odkaz na další položku. V případě kolize se vloží na první volné místo za synonymem a odkaz na něj, se vloží do předchozí položky. Pro implementaci jsme zvolili hashovací funkci `djb2 hash`, která je známá svou jednoduchostí a dostatečnou efektivitou pro běžné použití [1].

3.2.2 Tabulka s rozptýlenými položkami s explicitním zřetězením položek

Tato datová struktura nabízí mnohem větší upravitelnost vlivem užití maker, které dovolují určovat počet klíčů, používání vlastních rozptylujicích a dalších funkcí. Pro synonyma je užit jednosměrně vázaný seznam.

3.3 Dynamické pole

Dynamické pole v našem programu používáme na dvě věci. Za prvé si do něj ukládáme parametry funkcí, jako výčtové číslo typu tokenu s daným datovým typem. Jako další věc pomocí pole kontrolujeme, zdali funkce s neprázdnou návratovou hodnotou obsahuje `return` ve všech dostupných větvích funkce. Jako základní velikost pole jsme zvolili 16 a při přeplnění se jeho velikost zdvojnásobí.

3.4 Dynamický řetězec

Struktura dynamického byl vytvořen a využit speciálně pro zápis generovaných instrukcí. Zadávání nových instrukcí je možno primárně přidáváním na konec řetězce (nikoli dovnitř řetězce). Počáteční velikost dynamického řetězce je 16 a je případně dynamicky zvětšován na potřebnou výslednou velikost, která je vypočítána jako velikost stávajícího a přidávaného řetězce [3]. V neposlední řadě disponuje funkcí vyhledání podřetězce v řetězci, která je specificky využita pro vyhledání již dříve definované stejnojmenné proměnné. Dále disponuje funkcemi vypsání řetězce, spojení dvou dynamických řetězců a vyčištění obsahu.

3.5 Zásobník

Pro potřeby syntaktické analýzy jsme byla vytvořena struktura zásobníku. Zásobník je implementován polem a disponuje obvyklými funkcemi vložení prvku, odebrání prvku či nahlédnutí na nejvyšší prvek. Položka zásobníku je datového typu ukazatele na token `Token_ptr`. Díky tomuto příspupu lze během rekurzivního sestupu, či precedenční analýzy ukládat potřebné tokeny a zpětně k nim přistupovat bez nutnosti další alokace, jelikož tokeny již alokuje a udržuje `scanner`.

4 Testování

Pro testování jsme si vytvářeli vlastní soubory testů. Jednotlivé soubory testů byly navrženy pro otestování určité funkcionality překladače. Využili jsme i možnosti testování našeho překladače na testech poskytovaných dalšími studenty.

5 Práce v týmu

Na projektu jsme začali týden po vytvoření týmů. Na začátku jsme vytvořili repozitář ve verzovacím systému `Git` a stručný plán rozdělení práce. Schůzky se konaly přes platformu `Discord` jednou týdně do začátku listopadu, následně 2–3x týdně, na kterých se řešila funkčnost jednotlivých částí či problémy. Každý člen týmu se podílel při vytváření návrhu konečného automatu viz(A) a LL-gramatiky viz(B) prostřednictvím sdíleného online přístupu. Další části se řešily samostatně, popřípadě online konzultací. Ke konci vývoje se vytvářely krátké testíky, které testovali celkovou funkčnost programu.

5.1 Rozdělení práce

Vojtěch Hubáček	⇒	Vedoucí týmu, codegen, celková korekce programu
Jakub Ramašeuski	⇒	Lexikální analýza, hlídání paměti
Jozef Ondrejička	⇒	Syntaktická analýza
Lukáš Šidlík	⇒	Sémantická analýza, dokumentace

6 Závěr

Projekt pro nás na začátku představoval výzvu svou velikostí a složitostí. Postupem času, kdy jsme přednášky z IFJ sledovaly s předstihem získávali potřebné znalosti o tvorbě překladačů, jsme se do jeho řešení mohli pustit.

Tým jsme sestavili rychle a předem jsme se dohodli na způsobu komunikace, pravidelných setkáních a využití verzovacího systému. Díky tomu jsme neměli žádné problémy s týmovou spoluprací a práce probíhala bez větších problémů.

Tento projekt nám poskytl nejen nové znalosti o fungování překladačů, ale také praktické propojení látky z předmětů IFJ a IAL s reálnými aplikacemi. Získali jsme také cenné zkušenosti s projektem takového rozsahu.

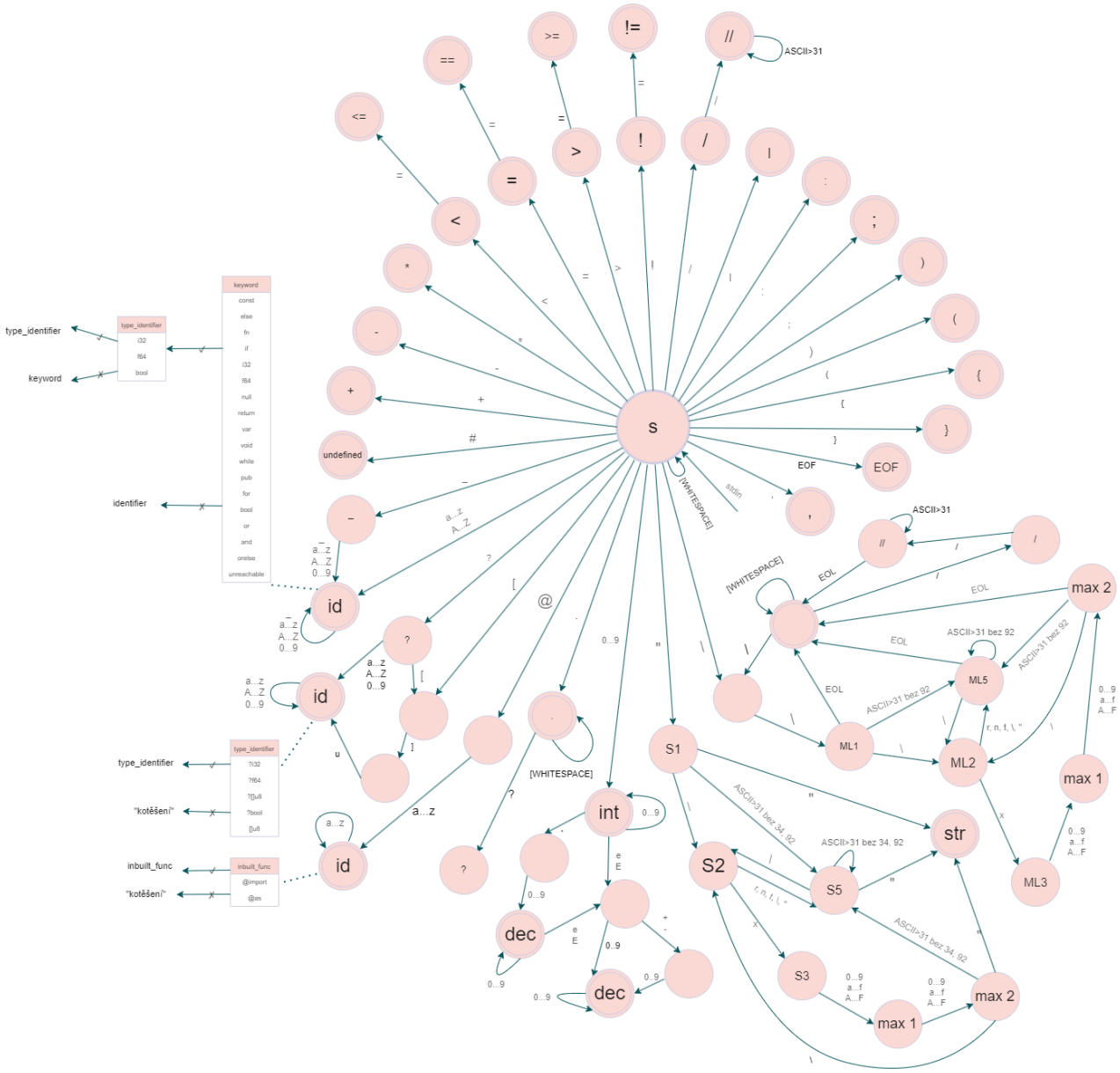
Reference

[1] CSE.YORKU.CA. Hash Functions [online]. Rev. 22.9.2003. [cit. 15.10.2024]. Odkaz: http://www.cse.yorku.ca/~oz/hash.html.

[2] HUSA, J. Slidy ze cvičení ISU [online]. Rev. 8.11.2024. [cit. 8.11.2024]. Odkaz: https://www.fit.vut.cz/person/ihusa/teaching/.cs#nav.

[3] STEVEB. *C Language Dynamic String* [online]. Rev. 20.12.2019. [cit. 20.11.2024]. Odkaz: https://www.codeproject.com/Articles/1259074/C-Language-Dynamic-String.

A Konečný automat lexikálního analyzátoru



Obrázek 2: Diagram konečného automatu pro lexikální analýzu

B LL – gramatika

1. <program> → <prolog> <fuction>
2. <prolog> → const ID @import (<string_value>) ;
3. <function> → pub fn ID (<parameter>) <return_type> { <body> } <function_next>
4. <function_next> → <function>
5. <function_next> → EOF
6. <parameter> → ID : <type> <parameter_next>
7. <parameter> → ε
8. <parameter_next> → <parameter>
9. <parameter_next> → ε
10. <body> → <statement> <body>
11. <body> → ε
12. <statement> → <id_option> <id_statement> ;
13. <statement> → <prefix> ID <definition> = <value> ;
14. <statement> → if (<value>) <not_null_value> <then> <else_then>
15. <statement> → while (<value>) <not_null_value> <then> <else_then>
16. <statement> → for (<for_value>) | <id_option> | <then>
17. <statement> → return <return_value> ;
18. <statement> → break ;
19. <statement> → continue ;
20. <definition> → : <type>
21. <definition> → ε
22. <id_statement> → = <value>
23. <id_statement> → <call>
24. <id_continue> → <call>
25. <id_continue> → ε
26. <call> → . ID (<call_params>)
27. <call> → (<call_params>)
28. <call_params> → <call_value> <call_params_next>
29. <call_params> → ε
30. <call_params_next> → , <call_params>
31. <call_params_next> → ε

32. <not_null_value> → | <id_option> |
33. <not_null_value> → ε
34. <id_option> → ID
35. <id_option> → _
36. <then> → <statement>
37. <then> → { <body> }
38. <else_then> → else <then>
39. <else_then> → ε
40. <prefix> → var
41. <prefix> → const
42. <value> → NULL
43. <value> → INTEGER
44. <value> → FLOAT
45. <value> → BOOLEAN
46. <value> → <expression>
47. <value> → @as (i32 , ID)
48. <value> → ID <id_continue>
49. <return_value> → <value>
50. <return_value> → ε
51. <call_value> → <value>
52. <call_value> → <string_value>
53. <for_value> → ID <id_continue>
54. <for_value> → <string_value>
55. <string_value> → STRING
56. <string_value> → MULTILINE_STRING
57. <type> → i32
58. <type> → f64
59. <type> → []u8
60. <type> → bool
61. <type> → ?i32
62. <type> → ?f64
63. <type> → ?[]u8
64. <return_type> → void
65. <return_type> → <type>

C LL – tabulka

	const	var	pub	EOF	ID	,	if	while	for	return	break	continue	_	:	=	.	(	)	{	else	NULL	INTEGER	FLOAT	BOOLEAN	EXPRESSION	@as	STRING	MULTILINE_STRING	i32	f64	[]u8	bool	?i32	?f64	?[]u8	void	###
<program>	1																																					
<prolog>	2																																					
<function>			3																																			
<function_next>			4	5																																		
<parameter>					6																																		7
<parameter_next>						8																																	9
<body>	9	9			9		9	9	9	9	9	9	9																									10
<statement>	13	13			12		14	15	16	17	18	19	12																									
<definition>														20																								21
<id_statement>															22	23	23																					
<id_continue>																24	24																					25
<call>																26	27																					
<call_params>					28																28	28	28	28	28	28	28	28									29
<call_params_next>						30																																	31
<not_null_value>																		32																				33
<id_option>					34								35																									
<then>	36	36			36		36	36	36	36	36	36	36						37																			
<else_then>																				38																		39
<prefix>	40	41																																				
<value>					48																42	43	44	45	46	47											
<return_value>					49																49	49	49	49	49	49											50
<call_value>					51																51	51	51	51	51	51	52	52									
<for_value>					53																						54	54										
<string_value>																											55	56										
<type>																													57	58	59	60	61	62	63		
<return_type>																													65	65	65	65	65	65	65	64	

Tabulka 1: LL – tabulka pro syntaktickou analýzu

D Precedenční tabulka

	$+-$	$*/$	rel	and	or	(	)	!	$orls$	$.?$	ID	\$
$+-$	$>$	$<$	$>$	$>$	$>$	$<$	$>$	$<$	$<$		$<$	$>$
$*/$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$		$<$	$>$
rel	$<$	$<$		$>$	$>$	$<$	$>$	$<$	$<$		$<$	$>$
and	$<$	$<$	$<$	$>$	$>$	$<$	$>$	$<$	$<$		$<$	$>$
or	$<$	$<$	$<$	$<$	$>$	$<$	$>$	$<$	$<$		$<$	$>$
(	$<$	$<$	$<$	$<$	$<$	$<$	$=$	$<$	$<$		$<$	
)	$>$	$>$	$>$	$>$	$>$		$>$		$>$	$=$		$>$
!		$>$	$>$	$>$	$>$	$<$	$>$	$<$	$>$		$<$	$>$
$orls$	$>$	$<$	$>$	$>$	$>$	$<$	$>$	$<$	$>$		$<$	$>$
$.?$	$>$	$>$	$>$	$>$	$>$		$>$		$>$	$=$		$>$
ID	$>$	$>$	$>$	$>$	$>$		$>$		$>$	$=$		$>$
\$	$<$	$<$	$<$	$<$	$<$	$<$		$<$	$<$		$<$	

Tabulka 2: Precedenční tabulka pro zpracování výrazů