VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Formální jazyky a překladače
Překladač imperativního jazyka IFJ19

Tým 052, varianta II

Vedoucí:	**Jaroslav Hort**	**(xhortj04)**	**25%**
	Filip Dráber	(xdrabe09)	25%
	Iveta Strnadová	(xstrna14)	25%
	Norbert Pócs	(xpocsn00)	25%

11. prosince 2019

1 Scanner

Scanner obsahuje jedinou funkci `getToken`, která nepotřebuje parametry a vrací načtený token (typ `Token`, více 5.5) při úspěchu, `NULL` při neúspěchu.

Funguje na principu konečného automatu (viz obr. 1 str. 7). Ze vstupního souboru (určeného v rozhraní, viz 4) načítá znak po znaku. Pokaždé načtený znak uloží do vytvářeného dynamického stringu (více 5.2) a podle toho, jaký znak to byl, přejde do dalšího stavu. Pokud není z daného stavu přechod a stav byl jeden z koncových, vygeneruje token, který vrátí. Nebyl-li stav koncový, nastala lexikální chyba, kterou scanner správně zahlásí a skončí.

Scanner je volán v průběhu programu mnohokrát. Většinou prochází konečný automat opět od počátečního stavu, nicméně sám si uchovává flagy (static proměnné), které mohou jeho chování mírně pozměnit (např. generování dedentu nastaví flag, který při příštím spuštění provede opětovnou kontrolu odsazení).

1.1 Typy tokenů

Pokud scanner skončil ve stavu pro identifikátory, zkontroluje prvně, jestli se nejedná o jedno z klíčových slov. V takovém případě by vrátil token typu daného klíčového slovo. Není-li to klíčové slovo, vrátí token typu `ID`, kterému předá i načtený dynamický string. Jednalo-li se o číslo (celé či desetinné), scanner nastaví tokenu správný typ, převede znaky načtené v dynamickém řetězci na skutečnou hodnotu a tu vrátí v položce tokenu (`DEC` pro desetinné, `INT` pro celé). Hodnotu řetězce vrací stejně, jako id v dynamickém řetězci v tokenu typu `STR`. Byla-li v řetězci použita escape sekvence, byl nejprve výraz za ní převeden na správnou hodnotu a teprve poté vrácen token s již upraveným řetězcem. Ostatní záležitosti jako `indent`, `EOL` a další předává v tokenu odpovídajícího typu.

1.2 Indent/dedent

Pro určení, zda nastal `indent` nebo `dedent` používá parser `indent_stack.h`, což je stack s hodnotou 0 na začátku, která se nikdy nevyjímá. Pokaždé, když scanner načítá ze začátku řádku[1] a narazí na mezeru nebo tabulátor, začne počítat tyto načtené znaky. Výsledný počet před prvním jiným znakem poté porovná s vrcholem zásobníku. Je-li na zásobníku menší číslo, pushne aktuální počet mezer/tabulátorů na zásobník a vygeneruje `indent`. Je-li počet stejný, nic nedělá a pokračuje v načítání. Je-li však na vrcholu zásobníku menší číslo, vrchní hodnota se popne, zapamatuje se tento počet mezer/tabulátorů a vygeneruje `dedent`. Při příštím volání scaneru se díky zapamatované hodnotě přejde opět ke kontrole. Znovu se porovná zapamatovaný počet s vrcholem zásobníku. Je-li na zásobníku menší číslo, nastala chyba odsazení. Stejné číslo je v pořádku a pokračuje se načítáním dalších znaků, u většího čísla se tento postup opakuje (pop vrcholu, generace dedentu, příště znova kontrola).

2 Parser

Parser se skládá z rekurzivní syntaktické analýzy (RSA) a precedenční analýzy (PA), kterou volá na zpracování výrazů.

[1] Na toto má statickou proměnnou, podle které to pozná.

2.1 Rekurzivní syntaktická analýza

RSA komunikuje se scannerem (volá z něj funkci `getToken` na zisk tokenu) a PA (volá ji pomocí `expressionAnalysis`[2]

RSA vytváří tříadresný kód (3AC), který se přes funkci `appendAC` ukládá do seznamu, aby ho později mohl použít generátor. Pro kontrolu správné syntaxe používá rekurzivní sestup podle pravidel LL(1) gramatiky, kterou jsme pro tento účel vytvořili.

Při nalezení syntaktické chyby (popř. při všech chybách kromě vnitřní nebo lexikální) kontrola nekončí, v pravidlech se nachází typy tokenů, na kterých se pokusí zotavit (např. `EOL` nebo `(`, pokud RSA najde chybu, pokusí se načítat tokeny tak dlouho, dokud na tento token nenarazí, a poté pokračuje).

2.2 LL(1) gramatika

Vytvořili jsme pravidla pro LL(1) gramatiku a tabulku k ní[3] (tabulka viz obr. 2 str. 8, pravidla viz listing 1 str. 9). Kostra RSA je implementována pomocí těchto pravidel. Každý neterminál je ve většině případů[4] jedna funkce, ve které derivace pravidla znamená volání dalších funkcí v případě rozkladu na neterminál a ověření typu tokenu v případě terminálu. Syntaktická chyba je rozpoznána, když v rámci rozkladu pravidla narazí na token, pro který není pravidlo (neodpovídá tokenu, který může v daný moment přijí).

2.3 Precedenční analýza

Precedenční analýza je v překladu využita pro zpracování výrazů. Je volána jako funkce `expressionAnalysis()` v knihově `precAnalysis_lib.c`. Tokeny získané buď přímo při volání nebo funkcí `getToken()` ukládá na stack a zpracovává je na základě precedenční tabulky (viz obr. 3 str. 10) uložené v souboru `PATable.c`. PA pracuje se speciálním ADT: „PA `char`“ zásobníkem. Funkce pro práci s tímto zásobníkem jsou uložené v knihovně `PACharStack_lib.c`. V rámci této knihovny se také generuje tříadresný kód v případě, že je na zásobníku aplikováno nějaké z pravidel pro zpracování výrazu ve funkci `PAApplyRule()` z knihovny `PACharStack_lib.c`.

2.4 Sémantická kontrola

V souboru `parser.c` se kromě RSA řeší i sémantické kontroly a generace tříadresného kódu. V kostře podle gramatiky se nejen kontroluje syntaktická stránka věci, ale jsou v ní i volání funkcí tabulky symbolů (více popsáno v sekci 5.6) a všechny akce potřebné ke generování 3AC. Umístili jsme to zároveň se syntaktickou analýzou, neboť se při rozkladu pravidel dalo dobře sledovat, v jaké části kódu jsme a jaké akce pro sémantiku/3AC je potřeba provést.

Sémantická kontrola z pohledu parseru spočívá čistě ve volání funkcí souboru `symtable`. Na začátku se zavolá funkce, která vytvoří potřebnou globální tabulku. Kdykoliv se narazí na ID (ať už v RSA nebo PA), volá se funkce `work_out_val_id` s parametry `Token` zpracovávaného id a `bool`, zda-li chceme id definovat nebo jsme na něj pouze narazili. U funkcí se jedná o `work_out_fce_id`,

[2]RSA volá PA, které předá jeden až dva tokeny, které jí již patří, ale byly omylem načteny RSA, a tokenem `res`, který má být použit pro uložení výsledku PA.

[3]S pomocí <https://www.fit.vutbr.cz/~ikocman/llkptg/>.

[4]Existují výjimky, kdy by pro syntaktickou analýzu toto stačilo, ale kvůli správné sémantické analýze nebo generování 3AC bylo potřeba za určitých podmínek provádět jinak (např. funkce print, ověření terminálu funkcí), nebo naopak v pravidlech to vyžadovalo další neterminál, ale při implementaci šlo jen o jednu kontrolu, bylo zbytečné vytvářet další funkci (např. return_item).

kde je opět `Token`, `bool` a ještě počet parametrů ve volání/definici této funkce. Aby symtable správně určila lokální id (při definici funkci) od globálních, je třeba při začátku definice funkce zavolat `go_in_local` a na konci `go_in_global`. O to, zda je id lokální/globální, správně použité, atd. se stará symtable. Z pohledu parseru pouze voláme tyto funkce.

2.5 Generování tříadresného kódu v parseru

Kód, který jsme si stanovili (více v 5.4), nám umožnil předávat generátoru všechny potřebné informace. V některých případech ovšem konstrukce byla problematická. Museli jsme udělat speciální generací pro funkci `print`, aby generátor nemusel pracovat s potenciálně nekonečným množstvím parametrů. Naše řešení bylo posílat každý parametr a za ním hned volání funkce print, namísto poslání vše a až následného volání.

Další problém, který stojí za zmínku, byla konstrukce `if-else` a `while`. Obojí je potřeba řešit vhodnou kombinací návěští a podmíněných/nepodmíněných skoků na ně. Ukázka 3AC pro `if-else` konstrukci:[5]

```
//volani PA, ta vygeneruje takovy 3AC, kde je
//v poslednim vysledek podminky v "cond"
(COND_JUMP, cond, , label_if)
(JUMP, , , label_else)
(LABEL, , , label_if)
//vlastni telo if casti
(JUMP, , , label_if_end)
(LABEL, , , label_else)
//vlastni telo else casti
(LABEL, , , label_end)
```

Obdobně bylo řešeno `while`, kde se ovšem podmínka vyhodnocovala uvnitř a v případě nepravdy se vyskočilo. Ještě však bylo třeba vyřešit zanořené cykly. Protože může být teoreticky nekonečno cyklů `while` nebo podmínek `if-else` zanořených v sobě, musely být návěští pro každou „vrstvu“ unikátní. Toto jsme vyřešili funkcí na generování žádaných návěští a proměnnou na počítání použitých konstrukcí, kde si každé pravidlo vzalo hodnotu, pro sebe uložilo, a k sdílené přičetlo jedničku.

3 Generátor výsledného kódu

Po úspěšné analýze je inicializován generátor výsledného kódu. Součástí inicializace je vyvtoření tabulek s rozptýlenými položkami pro ukládání názvu vytvořených proměnných a labelů pro funkce. Vlastní spuštění generátoru spočívá ve volání funkce `generate_code`.

3.1 Tříadresný kód

Tří adresný kód je generován na úrovni parseru a precedenční analýzy. Jeho struktura obsahuje v typ kódu, `ac_type`, a tři operandy typu `Token`, `op1`, `op2`, `res`. Výsledná struktura kódu je uložena do seznamu kódu, který je využit v generátoru. Hlavními funkcemi seznamu jsou funkce `setACAct`, která nastaví aktivitu seznamu na první položku, `actAC`, pro nastavení aktivity na následující položku a `isACActive` pro zjištení, zda je v seznamu aktivní prvek.

[5] Názorná reprezentace našeho 3AC, první položka je typ kódu, další tři cstringy předaných tokenů, po řadě op1, op2 a res. Není-li vypsán, je předán jako NULL. Jména návěští a tokenů se od skutečného kódu mírně liší, zde jsou pozměněna kvůli názornosti.

3.2 Průběh generování

Ze seznamu tři adresných kódu jsou postupně brány položky a podle typu kódu je rozhodnuto jaká instrukce bude zapsána do výsledného kódu. Výsledný kód je v průběhu generování rozdělen na dva bloky, kód hlavního programu (main) a kód funkcí a ukládán do dynamicého řetězce pro tyto bloky.

Tento způsob implementace byl zvolen z důvodu, že volání funkcí je realizováno pomocí skoku na návěští, pokud by byly funkce zapsány uvnitř hlavního kódu, bylo by nutné je přeskakovat, dokud nebudou volány, což by bylo horší na implementaci a výsledný kód by byl méně přehledný.

Po přečtení všech kódu a vygenerován instrukcí je výsledek vypsán na standardní výstup, nejprve kód funkci a poté kód hlavního bloku programu.

3.3 Vestavěné funkce

Vestavěné funkce jsou definovány v hlavičkovém souboru generátoru. V případě, že program poprvé volá jednu z těchto funkcí, jsou zapsnány do bloku funkcí a funkce je volána jako obyčejné uživatelské funkce. Výsledný kód tedy obsahuje pouze ty funkce, které jsou využívány, tímto rešením udržíme kód relativně čistý.

3.4 Generování identifikátorů

Při generování instrukcí pracujících s identifikátory je nejprve prohledána tabulka, zda nebyl identifikátor generován již dříve. Identifikátory z hlavního bloku programu, takzvané globání identifikátory, mají přednost před identifikátory uvnitř funkcí, nebo-li lokálními identifikátory. Pokud nebyl nalezen, je zapsána insturkce do kódu a do příslušné tabulky je vloženo jméno identifikátoru

3.5 Problém generování cyklu

U generování cyklu `while` byl zjištěn problém možné definice identifikátoru. Tuto možnost jsme ošetřili tak, že parser vygeneruje tří adresný kód `WHILE_START` pro označení začátku cyklu a `WHILE_END` pro konec cyklu. Když generátor narazí na začátek cyklu, projde všechny kódy uvnitř cyklu, provede kontrolu definice identifikátorů a případně identifikátory nadefinuje předem, než cyklus začne.

4 Rozhraní

Rozhraní (`ifj19.c`) přesměruje standartní vstup na `source_file` pro scanner, poté předá slovo parseru. Až parser skončí (udělá mezitím celou analýzu), zkontroluje hodnotu `global_error_code` - pokud žádná chyba nenastala, zavolá generátor, aby dokončil práci. Vyčistí případné použité zdroje a skončí s návratovou hodnotou rovnou `global_error_code`.

5 Pomocné knihovny

5.1 Knihovna chyb `errors`

Knihovna obsahuje výčet kódu všech možných chyb, které mohou nastat při překladu programu, globání proměnnou `global_error_code`, kterou jednotlivé moduly nastavují na jeden z typu chyby v případě, že někde nastala. Na této proměnné zavisí spuštění generátoru. Dále máme dvě

funkce pro výpis chyby na standardní chybový výstup `stderr`, první pro výpis interní chyby a druhá pro kompilační chyby, která tiskne navíc název souboru a číslo řádku zdrojového programu na které se chyba vyskytuje.

5.2 Dynamický řetězec

Pro jednodušší práci s řetězci v programovacím jazyce C jsme vytvořili knihovnu `dynamic_string`. Knihovna nám umožnuje vytvořit nový řetězec, rozšířit ho připsáním nového řetězce na konec stávajícího, nebo ho skrátit na požadovanou délku.

5.3 Zásobník `PAStack`

Tento zásobník, někdy také popisován jako `PACharStack` slouží k ukládání tokenů v rámci precedenční analýzy. Jednotlivé položky jsou reprezentovány strukturou `PAStackElem`. Položka záznamu `content` ukazuje na token, který je v ní uložen. Položka záznamu `belowPtr` ukazuje na položku stacku přímo pod touto, zásobník je implementován jako lineárně vázaný seznam, kterým je nutno procházet při hledání nejvyššího terminálu. Mimo vrchol zásobníku pak lze považovat zásobník za rekurzivní, nicméně funkce, které nad tímto ADT pracují, nepracují s rekurzí. Nakonec se ve struktuře položky nachází znak `c`, který položku na základě obsahu reprezentuje jako `'i'`, `'E'`, `'+'`, `'('`, `'['` a `'$'`. Lze se na něj tím pádem dívat tak, jak je reprezentován v prezentacích v předmětu IFJ[6].

5.4 Tříadresný kód

Tříadresný kód je stuktura, kterou rekurzivní a precedenční analýza předávají všechny potřebné informace generátoru. Skládá se z typu operace `ac_type` a tří operandů typu `Token`, které označují první a druhý operand operace (`op1` a `op2`) a výsledek (`res`). Není-li u daného typu operace jeden či více operand využit, je místo něj předána hodnota `NULL`.

Knihovna `adress_code` implementuje funkce pro práci s tímto kódem. Parser využívá funkci `appendAC`, která s předanými tokeny vytvoří jeden tříadresný kód a uloží ho do seznamu. S touto frontou pak generátor pracuje skrz funkce `setACAct` (nastaví aktivitu seznamu na první položku), `actAc` (posune aktivitu na další tříadresný kód, existuje-li takový), `isACActive` (zjištění, zda je seznam ještě aktivní) a skrz funkce na čtení aktuálního tříadresného kódu (`ReadAc`, `readACtype`, `readACop1`, `readACop2`, `readACres`).

Kvůli problému definice proměnných při generování `while` jsou v knihovně také funkce `set_ac_breakpoint` a `goto_ac_breakpoint`, které umožňují nastavení jednoho breakpointu a skok na něj.

5.5 Knihovna `token.h`

Knihovna pro práci se strukturou `Token`. Ta se skládá z `double` (uložení desetinné hodnoty pro desetinné číslo, jinak 0), `int` (uložení hodnoty pro celé číslo, jinak 0), `e_type` (typ tokenu) a dynamického řetězce (uložení jména pro id nebo řetězec, př. `NULL` pokud není využit).

[6] Znak `'['` nevyplývá přímo z prezentace, ale z komentáře a preferencí prof. Alexandra Meduny. Záměna `'<'` za `'['` navíc zabraňuje použití operátoru jako pomocného znaku.

Pro práci zvenčí jsou tu funkce `init_token` na inicializaci, množství funkcí na doplnění dat pro různé typy (např. `add_string`, `add_id`, ...), funkce pro zisk informace z tokenu (`getIntValue` a další) a `copy_token`, která slouží pro vytvoření identického tokenu.[7]

5.6 Tabulka symbolů

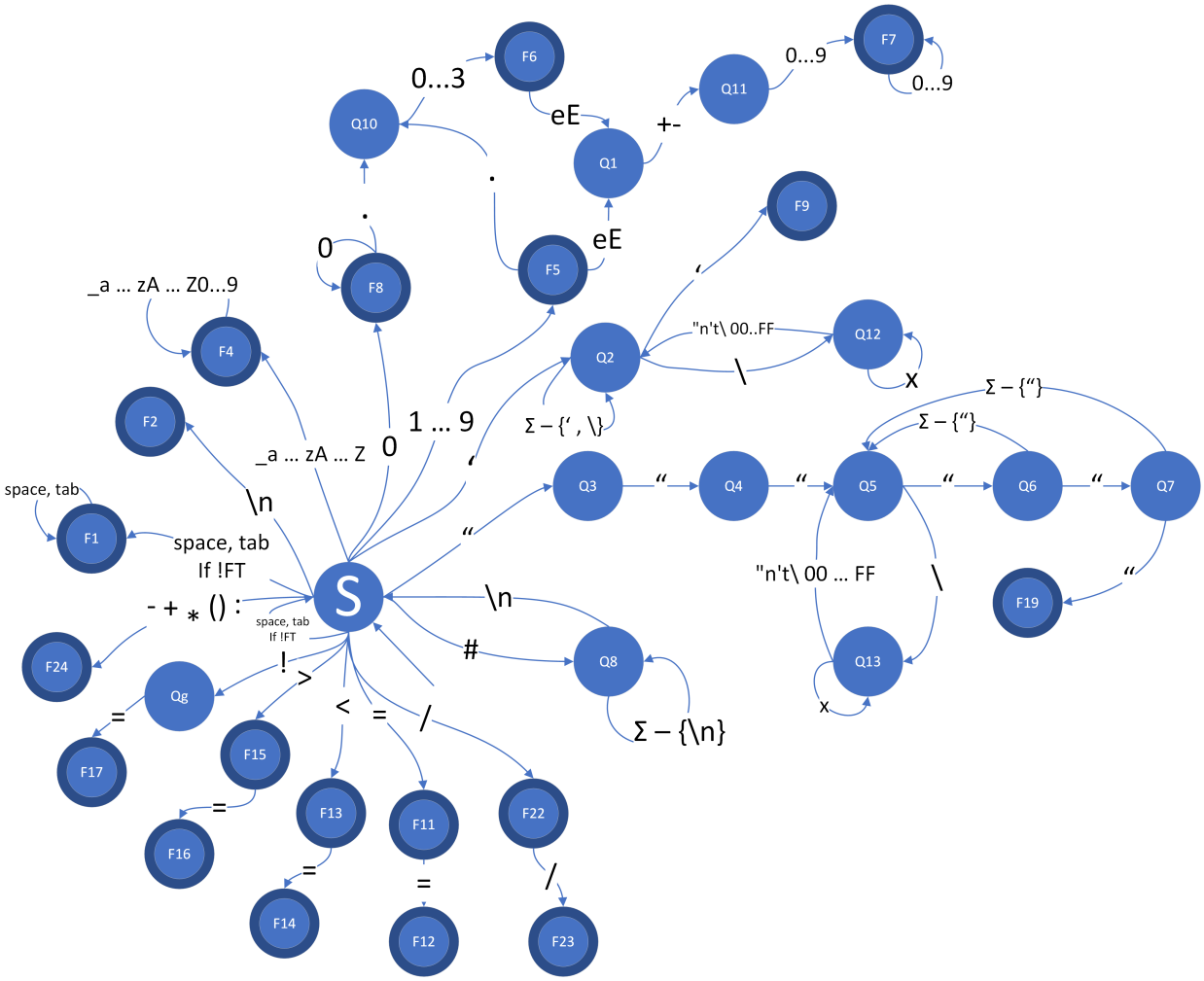
Tabulka symbolů implementovaná jako `hashtable` slouží pro sémantickou analýzu a jeden specifický případ syntaktické.

Funkce `start_symtable_with_functions` a `clean_all_symtables` slouží k inicializaci (s vestavěnými funkcemi) na začátku analýzy a zničení na jejím konci. Pro pohyb mezi lokální a globální tabulkou používá parser funkce `go_in_global` a `go_in_local`, které lokální tabulku vytvoří/zničí a upravují, kde se budou hledat a zapisovat nová id. Parser volá funkce pro zpracování id (`work_out_fce_id` a další), které podle situace vyhodnotí správnost použití id a zanesou potřebné informace do tabulky. Použití `global_check_def` na konci parseru zkontroluje, zda byly všechny funkce definovány.[8]

Speciální případ je funkce `is_this_ret_okay`, kterou volá parser, když najde klíčové slovo `return`. Protože symtable si uchovává, zda je ve funkci (lokálně) nebo těle programu (globálně), pomáhá určit, zda je return korektní nebo ne (v takovém případě vrací syntaktickou chybu).

[7] Toto je důležité pro vytváření tříadresného kódu, kde jsou dočasné proměnné použity vždy dvakrát, jako výsledek předchozí operace a jako operand následující.

[8] Analýzu provádíme jednoprůchodově. U funkcí se zaznamenává flag, zda už byly definovány, na konci je zkontrolováno, zda někde není flag hodnoty `false`.



Obrázek 1: Konečný automat scanneru

	EOF	def	id	'('	')'	':'	EOL	ind	ded	pass	ret	wh	expr	if	else	int
P	1	1	1	1			1			1	1	1		1		1
MEOL	30	30	30	30			29		30	30	30	30		30		30
PBWD	4	2	3	3						3	3	3		3		3
PL			20		21											20
NEPB			5	5			5			5	5	5		5		5
C			16	15						8	9	10		11		12
PI			27													24
MP					23											
C			16	15						8	9	10		11		12
PB			6	6					7	6	6	6		6		6
RI							19						17			
NS1				31			34									
NS2			35	39												36
OP																
NS3				40			42									

	float	str	Nn	','	'='	'+'	'-'	'*'	'/'	'//'	'<'	'>'	'<='	'>='	'!='	'=='
P	1	1														
MEOL	30	30														
PBWD	3	3														
PL	20	20	20													
NEPB	5	5														
C	13	14														
PI	25	26	28													
MP				22												
C	13	14														
PB	6	6														
RI			18													
NS1					32	33	33	33	33	33	33	33	33	33	33	33
NS2	38	37														
OP						43	44	45	46	47	48	49	50	51	52	53
NS3						41	41	41	41	41	41	41	41	41	41	41

P prog_body
MEOL more_EOL
PBWD prog_body_with_def
PL param_list
NEPB nonempty_prog_body
C command
PI param_item
MP more_item
RI return_item
OP op
C command
PB prog_body
NS1 not_sure1
NS2 not_sure2
NS3 not_sure3
ind indent
ded dedent
ret return
wh while
Nn None

Obrázek 2: LL-tabulka

Listing 1: Pravidla LL(1) gramatiky

```
<prog> -> <more_EOL> <prog_body_with_def> EOF

<prog_body_with_def> -> epsilon
<prog_body_with_def> -> def id ( <param_list> ) : EOL indent <nonempty_prog_body>
                        dedent <more_EOL> <prog_body_with_def>
<prog_body_with_def> -> <command> <prog_body_with_def>

<nonempty_prog_body> -> <more_EOL> <command> <prog_body>

<prog_body> -> <command> <prog_body>
<prog_body> -> epsilon

<command> -> pass EOL <more_EOL>
<command> -> return <return_item> EOL <more_EOL>
<command> -> while expr : EOL indent <nonempty_prog_body> dedent <more_EOL>
<command> -> if expr : EOL indent <nonempty_prog_body> dedent else : EOL
             indent <nonempty_prog_body> dedent <more_EOL>
<command> -> int expr EOL <more_EOL>
<command> -> float expr EOL <more_EOL>
<command> -> str expr EOL <more_EOL>
<command> -> ( expr EOL <more_EOL>
<command> -> id <not_sure1> EOL <more_EOL>

<return_item> -> expr
<return_item> -> epsilon
<return_item> -> None

<param_list> -> <param_item> <more_params>
<param_list> -> epsilon

<more_params> -> , <param_item> <more_params>
<more_params> -> epsilon

<param_item> -> int
<param_item> -> float
<param_item> -> str
<param_item> -> id
<param_item> -> None

<more_EOL> -> EOL <more_EOL>
<more_EOL> -> epsilon

<not_sure1> -> ( <param_list> )
<not_sure1> -> = <not_sure2>
<not_sure1> -> <op> expr
<not_sure1> -> epsilon

<not_sure2> -> id <not_sure3>
<not_sure2> -> int expr
<not_sure2> -> float expr
<not_sure2> -> str expr
<not_sure2> -> ( expr

<not_sure3> -> ( <param_list> )
<not_sure3> -> <op> expr
<not_sure3> -> epsilon

<op> -> +               <op> -> -
<op> -> *               <op> -> /
<op> -> //              <op> -> <
<op> -> >               <op> -> <=
<op> -> >=              <op> -> !=
<op> -> ==
```

	ID	ind	dec	str	<	<	´>	>=	==	!=	+	-	*	/	//	(	)	None	$
ID					]	]	]	]	]	]	]	]	]	]	]		]		]
int					]	]	]	]	]	]	]	]	]	]	]		]		]
dec					]	]	]	]	]	]	]	]	]	]	4		]		]
str					]	]	]	]	]	]	]	4	4	4	4		]		]
<	[	[	[	[							[	[	[	[	[	[	]	4	]
<=	[	[	[	[							[	[	[	[	[	[	]	4	]
>	[	[	[	[							[	[	[	[	[	[	]	4	]
>=	[	[	[	[							[	[	[	[	[	[	]	4	]
==	[	[	[	[							[	[	[	[	[	[	]	[	]
!=	[	[	[	[							[	[	[	[	[	[	]	[	]
+	[	[	[	[	]	]	]	]	]	]	]	]	[	[	[	[	]	4	]
-	[	[	[	4	]	]	]	]	]	]	]	]	[	[	[	[	]	4	]
*	[	[	[	4	]	]	]	]	]	]	]	]	]	]	]	[	]	4	]
/	[	[	[	4	]	]	]	]	]	]	]	]	]	]	]	[	]	4	]
//	[	[	4	4	]	]	]	]	]	]	]	]	]	]	]	[	]	4	]
(	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	=	[	
)					]	]	]	]	]	]	]	]	]	]	]		]		]
None					4	4	4	4	]	]	4	4	4	4	4		]		
$	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[	[			

Obrázek 3: Tabulka pro precedenční analýzu (prázdné políčko – syntaktická chyba, políčko ”4”– sémentická chyba typové kompatibility