DOKUMENTACE PROJEKTU

IFJ a IAL

Implementace překladače imperativního jazyka IFJ19

Tým 65, varianta I

Diviš Jan	*xdivis12*		*25 %*
Kopáček Jiří	*xkopac06*	*vedoucí*	*25 %*
Pojsl Jakub	*xpojsl00*		*25 %*
Sasín Jonáš	*xsasin05*		*25 %*

11. 12. 2019

Obsah

1. Úvod/zadání

Cíl projektu bylo vytvořit překladač, který přeloží jazyk ifj19 (zjednodušená verze jazyka Python) do mezikódu ifjcode19, pro který už byl dodán interpret. Program tedy načítá na vstupu kód v jazyce ifj19 a na výstup tiskne instrukce v jazyce ifjcode19.

2. Práce v týmu

2.1 Rozdělení

Jan Diviš

- Makefile, Abstraktní datové typy, syntaktická analýza výrazů, generování kódu, dokumentace

Jiří Kopáček

- Vedení projektu, lexikální analýza, abstraktní datové typy, tabulka symbolů, generování vestavěných funkcí, dokumentace

Jakub Pojsl

- Abstraktní datové typy, návrh gramatiky, syntaktická a sémantická analýza výrazů, generování výrazů včetně potřebných typových kontrol, dokumentace

Jonáš Sasín

- Návrh gramatiky, abstraktní datové typy, syntaktická a sémantická analýza kódu (kromě výrazů), generování kódu (kromě výrazů), dokumentace

2.2 Komunikace

Snažili jsme se rozdělit práci tak, aby každý mohl z co největší části pracovat samostatně. Díky tomu nám stačilo minimum osobních setkání a většina komunikace probíhala elektronicky na našem vlastním Discord serveru. Většinou bylo potřeba se pouze domluvit na rozhraní některých funkcí, popřípadě se skupinově zamyslet nad určitým problémem (např. gramatika).

2.3 Správa kódu

Pro správu kódu jsme využívali Git, zdrojový kód jsme sdíleli přes GitHub.

3. Implementace

Projekt se skládá z několika hlavních částí:

3.1 Lexikální analýza

Celý lexikální analyzátor je implementován v souboru scanner.c a využívá pomocné datové struktury zásobník a dynamický řetězec. Scanner je implementován standardně podle předem navrženého konečného automatu (jehož diagram je přiložen na konci této dokumentace).

Mezi hlavní části scanneru patří struktura Token, která má 2 složky. První je typ tokenType, který slouží pro identifikaci typu tokenu a druhá je typ tString, což je dynamický řetězec sloužící pro ukládání atributu tokenu (návrh pro tuto implementaci byl inspirován staršími záznamy democvičení k předmětu IFJ).
Další hlavní částí je funkce get_next_token. Této funkci je při volání předán odkaz na nějaký výše zmíněný token a scanner následně přepíše složky tohoto tokenu podle přečtených informací na vstupu. Samotná funkce je převážně tvořena cyklem while, který čte znaky ze vstupu a v něm vnořeném příkazu switch, který přepíná mezi stavy konečného automatu.

Indentace

Asi největším problémem při implementaci scanneru byla detekce indentace. Tento problém vedl k vytvoření další části scanneru, a to indentačního zásobníku. Samotný zásobník ukládá typ integer a má jednoduchou strukturu (shodnou se způsobem implementace vyučovaným v předmětu IAL). Scanner si pamatuje, zdali byl poslední token konec řádku. Pokud ano, tak při získávání dalšího tokenu přejde do stavu počítání indentace. V něm počítá všechny mezery až do prvního jiného znaku na řádku. Pokud je tento první znak řádkový komentář nebo konec řádku, je počet mezer zahozen. Pokud ne, je porovnán počet mezer s hodnotou na zásobníku.

Větší hodnota než je na zásobníku vede k pushnutí počtu mezer na zásobník a generování tokenu indent. Stejná hodnota jen zahodí současný počet mezer. Nižší hodnota znamená, že došlo k dedentaci. To vede kromě generování tokenu dedent také k nastavení dedentační fáze scanneru. Ten existuje kvůli případu, kdy je potřeba generovat více dedentací zasebou. Při dalším volání get_next_token se před čtením dalších znaků nejdříve zkontroluje, zdali jsme v dedentační fázi. Pokud ano, tak znovu porovnáme dříve získaný počet mezer a hodnotu na vrcholu zásobníku. To budeme opakovat tak dlouho, dokud hledaný počet mezer nenalezneme na vrcholu zásobníku a tím vygenerujeme všechny potřebné tokeny dedent. Dedentační fáze je také užitečná při dosažení konce vstupu, kdy opět nastává dedentační fáze před generováním tokenu EOF s tím, že hledáme indentaci 0. Tím dosáhneme vyprázdnění zásobníku a generování všech zbývajících tokenů dedent.

Další funkce scanneru už zahrnují pouze běžné kontroly, jako například zdali je daný znak alfanumerický nebo zdali je daný řetězec rezervované klíčové slovo.

Ve výsledku lze tedy se scannerem pracovat pouze pomocí funkce get_next_token po inicializaci indentačního zásobníku.

3.2 Syntaktická analýza

3.2.1 Rekurzivní sestup

Základem pro implementaci syntaktické analýzy hlavního těla programu byla LL-gramatika, na které jsme pracovali společně, abychom se pokusili co nejvíce eliminovat mezery v návrhu.

Každé pravidlo v LL-gramatice je v programu reprezentováno samostatnou funkcí, které se navzájem rekurzivně volají. Data jsou mezi nimi předávány pomocí ukazatele na strukturu prog_data, která zapouzdřuje informace potřebné pro analýzu a generování, v rámci rekurzivního sestupu. Za zmínku stojí například lokální a globální tabulka symbolů, soubor, ze kterého čte scanner, aktuální token, data aktuální funkce nebo list tokenů, který využívá precedenční analýza.

Hlavní funkce parseru je analyse, ve které probíhá inicializace programových dat a volání funkce reprezentující počáteční neterminál „program“.

V každé funkci reprezentující pravidlo, se volá funkce get_next_token, která načte další token. Po načtení tokenu proběhne buď kontrola, zda jsme dostali očekávaný terminál, nebo rozhodnutí, kterou funkci (pravidlo) použít jako další.

Při chybě je kód chyby nejčastěji uložen do proměnné err, jejíž hodnotu pak každá funkce vrací své volající funkci, až nakonec vrátí funkce analyse kýžený error code přímo do mainu.

Problém s rozhodováním a tím i porušení LL(1) gramatiky nastal u výrazů a volání funkcí, konkrétně ve funkcích/pravidlech idwhat a assign. Museli jsme se rozhodovat na základě 2 načtených tokenů, kvůli čemuž jsme museli řešit problém komunikace s precedenční analýzou. Nakonec jsme se rozhodli načtené tokeny načítat do struktury tokenList (implementovaný jako jednosměrný lineární spojový seznam) až po konec výrazu, kdy výraz je v některých případech zakončen dvojtečkou a v jiných EOLem. Pokud první načtený token není identifikátor, vyhodnocujeme výraz.

Pokud první načtený token je identifikátor, zkopírujeme jej do pomocného tokenu a načteme další token. Pokud je nový token levá závorka, postupujeme jako při volání funkce. Pokud je cokoliv jiného, načteme do seznamu tokenů výraz, včetně předešlého/pomocného tokenu, a vyhodnocujeme pomocí precedenční analýzy.

3.2.2 Precedenční analýza

V rámci precedenční analýzy se provádí ověření syntaxe výrazů a případně sémantické kontroly proměnných vyskytujících se ve výrazu. Neimplementovali jsme rozšíření, umožňující volání funkcí ve výrazech, tudíž výskyt identifikátoru funkce ve výrazu vede na syntaktickou chybu.

Hlavní funkce precedenční analýzy je funkce expression, která je volána vždy, když se při rekurzivním sestupu narazí na výraz. Funkce dostane jako parametr ukazatel na strukturu data, ve které jsou všechny potřebné informace pro zpracování výrazu. Zásadní je již zmíněný seznam tokenů token list, který obsahuje všechny tokeny až do konce výrazu. Pro tuto implementaci jsme se rozhodli z důvodu zjednodušení komunikace s precedenční analýzou a také nám to zajistilo její větší nezávislost a možnost jednoduššího samostatného testování.

Tokeny jsou postupně načítány ze seznamu. Při načtení tokenu, je token podle typu převeden na „symbol“ z výčtu enum. To nám zjednoduší práci s daným symbolem, a především nám symbol jakožto index umožní získat pravidlo z precedenční tabulky, která je sestavena na základě priorit jednotlivých operátorů.

Podle získaného pravidla z precedenční tabulky pro daný symbol (=, <, >, “ “) se poté provádí korespondující operace na základě algoritmu pro precedenční SA. Na základě tohoto algoritmu jsou jednotlivé symboly vkládány na zásobník a poté je prováděna redukce jednotlivých symbolů na neterminální symboly (simulace vytváření ASS), pokud existuje pravidlo pro danou redukci (např. E -> i). Pokud pravidlo neexistuje, je výraz chybný a zpracování je ukončeno s návratem kódu syntaktické chyby.

Podle pravidel jsou průběžně generovány instrukce (volání funkcí pro generování ze souboru generator.c), potřebné pro skutečné vyhodnocení výrazu při interpretaci. O pravidlu se rozhoduje na základě počtu neterminálů pro vykonání operace (1 nebo 3). V případě 3, se o volbě generovaných instrukcí rozhoduje podle operátoru mezi 2 operandy. Nepřišlo nám významně přínosné kontrolovat kompatibilitu typů u operací s konstantami již při překladu a proto se veškeré typové kontroly provádí jednotně až při interpretaci, vzhledem k dynamickému typování proměnných jazyka.

3.3 Sémantická analýza

Sémantická analýza je velmi úzce navázána na rekurzivní sestup. Jestli se parser zrovna nachází v těle funkce detekuje proměnná in_function struktury prog_data. Analýza využívá lokální a globální tabulky symbolů a kontrolují se pomocí ní definice proměnných, definice funkcí a správný počet parametrů při jejich volání. Pokud se proměnná použije ve výrazu a nenajde se v lokální a poté ani v globální tabulce, je nedefinovaná. Pokud probíhá přiřazení a nastane stejná situace, proměnná se přidá do tabulky symbolů (lokální/globální) na základě toho, zda se nacházíme v těle funkce, což nám napoví proměnná in_function.

Záludnější byly definice funkcí, které mohou být lexikálně definovány až za jejich použitím. Funkci do globální tabulky přidáme hned při prvním kontaktu. Pokud je to při definici, nastavíme vlastnost defined na true. Pokud narazíme na definici a funkce již v tabulce je, s vlastností defined == false, nastavíme defined na true, v opačném případě se jedná o redefinici funkce.

Na konci funkce analyse poté projdeme všechny funkce v globální tabulce symbolů a zkontrolujeme, zdali byly všechny řádně definovány. Pokud narazíme na funkci, jejíž defined má hodnotu false, byla v průběhu použita funkce bez definice.

Na začátku funkce analyse jsou všechny vestavěné funkce přidány do tabulky symbolů, vlastnost defined je nastavena na true a je určen také očekávaný počet parametrů.

Při počítání parametrů u volání funkce zjistíme na základě globální tabulky symbolů, zdali byla funkce již zmíněna (volána/definována). Pokud byla použita, porovnáme počet parametrů u předchozí a současné zmínky funkce a pokud se počet rovná, je vše v pořádku.

Samostatný problém představovala funkce print, která nemá daný počet parametrů. Proto pro ni kontrola neprobíhá, ale funkce se volá pro každý parametr samostatně.

3.4 Generování kódu

Hlavní tělo generujeme funkcemi ze souboru generator.c. Zde jsou obsaženy zabudované funkce jazyka ifj19 a také hlavní funkce generate_main_body, která vygeneruje hlavičku a zabudované funkce a tiskne je na výstup.

Generování je poté provázáno s rekurzivním sestupem a jednotlivé instrukce se průběžně tisknou přímo na standartní výstup. Pomocné funkce jsou v souboru generator.c a jsou volány přímo z těl funkcí parseru nebo precedenční analýzy.

Pro unikátní indexaci návěští podmínek a cyklů jsme použily proměnné typu int ve struktuře prog_data. Zbytek návěští je pojmenován unikátně přímo podle reálných jmen funkcí, stejně jako u jmen proměnných.

Výsledná hodnota po vyhodnocení výrazu nebo ukončení volání funkce je uložena do globální proměnné exp_result.

Pokud jsou ve výrazech nebo při volání funkce použity konstanty, je použita funkce token_to_ifjcode_val, která na základě typu tokenu určí datový typ konstanty a případně její hodnotu převede do formátu konstant v ifjcode19.

Pro kontrolu kompatibility datových typů a případnou implicitní konverzi, byly implementovány speciální funkce, které se vygenerují na začátku, společně s vestavěnými funkcemi. Tyto funkce jsou poté volány pouze v případě nutnosti vykonání konkrétní kontroly typů. Tento způsob řešení zredukuje počet vygenerovaných instrukcí při překladu delších programů obsahujících výrazy, jelikož není nutné generovat instrukce pro typové kontroly při každém provádění operace, která kontrolu vyžaduje.

3.5 Pomocné datové struktury

Během implementace jednotlivých částí jsme využili několik datových struktur, většinou probíraných v předmětu IAL.

3.5.1 Tabulka symbolů

Tabulka slouží pro ukládání uživatelských proměnných a funkcí a je implementována pomocí binárního vyhledávácího stromu v souboru symtable.c. Detaily implementace této struktury byly navrhnuty na základě poznatků z předmětu IAL. Funkce vyhledávání a vkládání uzlu (a smazání s tím, že smazání samostatného uzlu nakonec nebylo využito) jsou implementovány nerekurzivně, protože se jedná o časté operace. Funkce smazání všech uzlů a prohledání všech uzlů kvůli nedefinovaným funkcím jsou implementovány rekurzivně, obě tyto funkce jsou volány pouze na konci práce s tabulkou symbolů.

Samotné uzly obsahují složky klíč, dále proměnnou, která určuje, zdali je v daném uzlu uložena proměnná nebo funkce, odkazy na levý a pravý uzel a také strukturu tSymdata, která obsahuje informace o symbolu. Konkrétně se jedná o počet parametrů funkce a informaci, zdali byla funkce tohoto uzlu definovaná. Tyto informace původně mohly být obsažený přímo v uzlu stromu, při návrhu jsme si ale ještě nebyli jisti, co vše budeme potřebovat znát o symbolu a tak jsme pro něj vytvořili samostatnou strukturu.

3.5.2 Zásobník

Jak už bylo zmíněno, zásobník byl využit při počítání indentace v lexikální analýze a je implementován v souboru stack.c. Dále byl implementován také druhý zásobník využitý při analýze výrazů, který má podobný účel jako běžný zásobník, ale prvky v něm uložené také obsahují odkaz na následující prvek a vrchol tohoto zásobníku je vždy první položka. Kvůli tomu už se také trochu jedná o jednosměrně vázaný seznam, ve kterém lze přidávat a odebírat prvky nejen na začátku seznamu.

3.5.3 Dynamický řetězec

Dynamický řetězec je struktura, která obsahuje odkaz na pole znaků s daným řetězcem a také proměnné délka a alokovaná délka. Tato struktura je využita téměř všude, kde je potřeba skládat nějaký řetězec. Hlavní využití má pro ukládání atributů tokenů, které mohou být nekonečně dlouhé. Při přidávání do řetězce se kontroluje přesáhnutí alokované délky a případně je řetězec realokován.

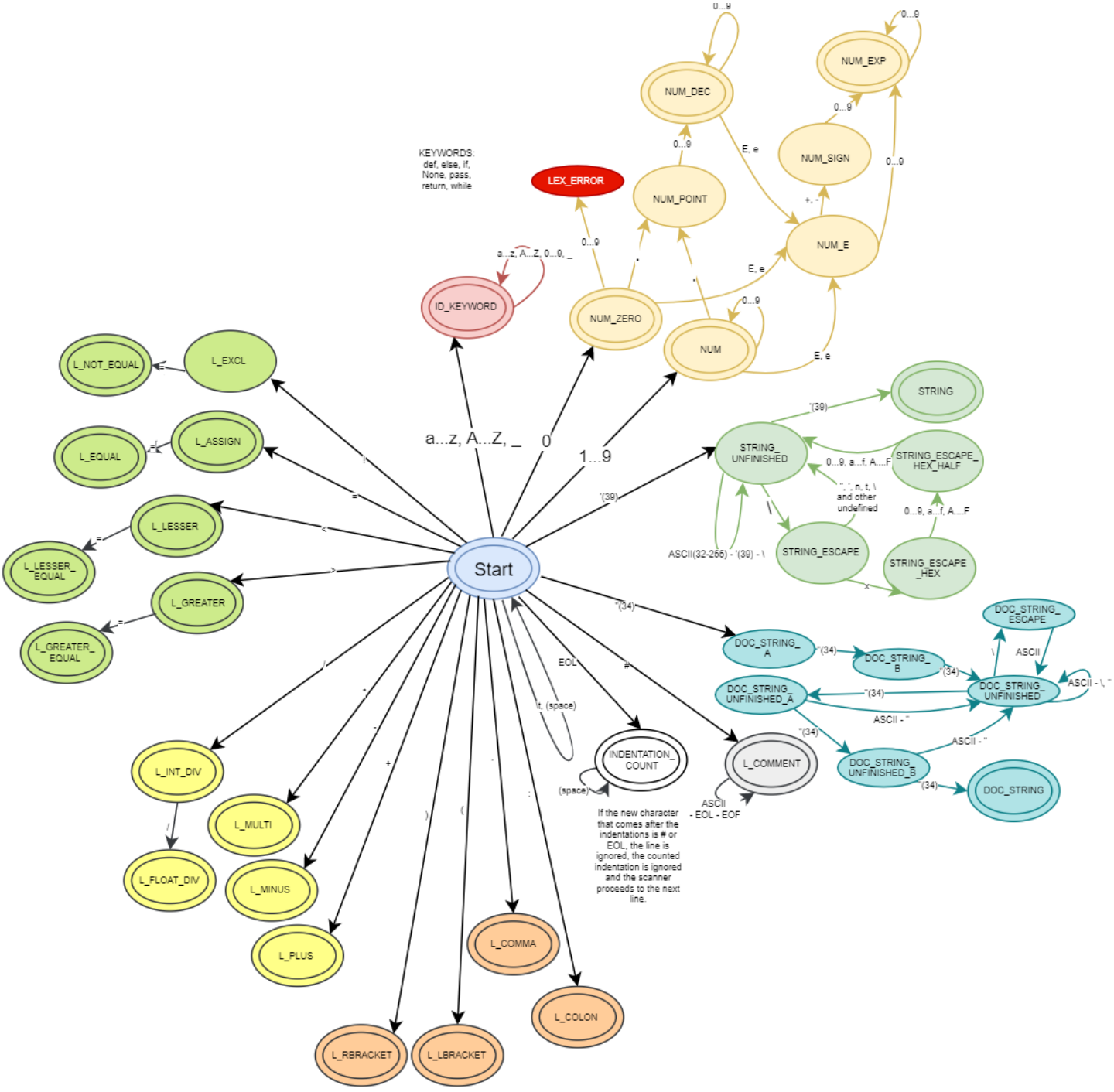
3.5.4 Seznam tokenů

Seznam tokenů je struktura určená pro předání tokenů mezi parserem a precedenční analýzou. Parser precedenční analýze vždy zprostředkuje právě seznam tokenů, ve kterém je načtený celý výraz pro vyhodnocení. Každý prvek obsahuje kromě tokenu také ukazatel na další token. Do seznamu se prvky přidávají vždy na konec a odebírají se ze začátku, takže je sémanticky podobný frontě. Precedenční analýza poté pracuje vždy s prvním prvkem seznamu.

4. Závěr

Tým na projekt jsme měli sestavený už před začátkem semestru. Kromě toho už někteří z nás spolu dříve na týmovém projektu pracovali (v rámci předmětu IVS), tudíž jsme s týmovým vývojem už měli zkušenosti a spolupráce nepředstavovala žádný problém. Trochu se nám nepodařilo odhadnout časovou náročnost některých potřebných úkonů, a proto jsme nestihli mít na pokusné odevzdání celý projekt hotový (což jsme původně plánovali) kvůli absenci generování kódu. Tuto část jsme však dělali společně, a tak nebyl problém ji stihnout do finálního odevzdání. Přestože byl projekt poměrně časově náročný, všichni v týmu by asi souhlasili s tvrzením, že se jednalo o jednu z nejzajímavějších a nejzábavnějších částí informatiky, se kterou jsme se doposud na této fakultě setkali. Kromě toho bylo IFJ zatím ten největší projekt, na kterém každý z nás pracoval, což samo o sobě přinese spoustu užitečných zkušeností.

5. Diagram konečného automatu



6. LL-gramatika

1. <program> -> <statement> <program>
2. <program> -> <def_function> <program>
3. <program> -> EOF

4. <term> -> value <term_n>
5. <term> -> id <term_n>
6. <term> -> None <term_n>
7. <term> -> Ɛ
8. <term_n> -> , <term>
9. <term_n> -> Ɛ

10. <def_function> -> def id (<param>) : EOL INDENT <statement_fun> DEDENT
11. <param> -> id <param_n>
12. <param> -> Ɛ
13. <param_n> -> , <param>
14. <param_n> -> Ɛ

15. <statement_fun> -> <expression> EOL <statement_fun>
16. <statement_fun> -> id <idwhat> EOL <statement_fun>
17. <statement_fun> -> if <expression> : EOL INDENT <statement_fun> DEDENT else : EOL INDENT <statement_fun> DEDENT <statement_fun>
18. <statement_fun> -> while <expression> : EOL INDENT <statement_fun> DEDENT <statement_fun>
19. <statement_fun> -> pass EOL <statement_fun>
20. <statement_fun> -> return <return_value> EOL <statement_fun>
21. <statement_fun> -> Ɛ

22. <return_value> -> <expression>
23. <return_value> -> Ɛ

24. <statement> -> <expression> EOL <statement>
25. <statement> -> id <idwhat> EOL <statement>
26. <statement> -> if <expression> : EOL INDENT <statement> DEDENT else : EOL INDENT <statement> DEDENT <statement>
27. <statement> -> while <expression> : EOL INDENT <statement> DEDENT <statement>
28. <statement> -> pass EOL <statement>
29. <statement> -> Ɛ

30. <idwhat> -> = <assign>
31. <idwhat> -> (<term>)
32. <idwhat> -> Ɛ

33. <assign> -> id(<term>)
34. <assign> -> <expression>

7. LL-tabulka

	def	value	id	None	,	return	expression	if	while	pass	=	(	ε	EOF
<program>	2		25				24	26	27	28				3
<def_function>	10													
<term>		4	5	6									7	
<term_n					8								9	
<param>			11										12	
<param_n>					13								14	
<return_value>							22						23	
<statement_fun>			16			20	15	17	18	19			21	
<statement>			25				24	26	27	28			29	
<idwhat>											30	31	32	
<assign>			33				34							

8. Precedenční tabulka

```
  0    1    2    3    4    5    6    7    8    9    10   11   12   13   14  15
  <    >    <=   >=   ==   !=   +    -    *    /    //   (    )    t    =    $    */
{'>', '>', '>', '>', '>', '>', '<', '<', '<', '<', '<', '<', '>', '<', ' ', '>'}, // <
{'>', '>', '>', '>', '>', '>', '<', '<', '<', '<', '<', '<', '>', '<', ' ', '>'}, // >
{'>', '>', '>', '>', '>', '>', '<', '<', '<', '<', '<', '<', '>', '<', ' ', '>'}, // <=
{'>', '>', '>', '>', '>', '>', '<', '<', '<', '<', '<', '<', '>', '<', ' ', '>'}, // >=
{'>', '>', '>', '>', '>', '>', '<', '<', '<', '<', '<', '<', '>', '<', ' ', '>'}, // ==
{'>', '>', '>', '>', '>', '>', '<', '<', '<', '<', '<', '<', '>', '<', ' ', '>'}, // !=
{'>', '>', '>', '>', '>', '>', '>', '>', '<', '<', '<', '<', '>', '<', ' ', '>'}, // +
{'>', '>', '>', '>', '>', '>', '>', '>', '<', '<', '<', '<', '>', '<', ' ', '>'}, // -
{'>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '<', '>', '<', ' ', '>'}, // *
{'>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '<', '>', '<', ' ', '>'}, // /
{'>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '<', '>', '<', ' ', '>'}, // //
{'<', '<', '<', '<', '<', '<', '<', '<', '<', '<', '<', '<', '=', '<', ' ', ' '}, // (
{'>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', ' ', '>', ' ', ' ', '>'}, // )
{'>', '>', '>', '>', '>', '>', '>', '>', '>', '>', '>', ' ', '>', ' ', '=', '>'}, // t (term - id,int,str,float,None)
{'<', '<', '<', '<', '<', '<', '<', '<', '<', '<', '<', '<', '>', '<', ' ', '>'}, // =
{'<', '<', '<', '<', '<', '<', '<', '<', '<', '<', '<', ' ', '<', ' ', ' ', ' '}, // $
```