Vysoké učení technické v Brně
Fakulta informačních technologií

Formální jazyky a překladače
2016/2017

Dokumentace interpretu imperativního jazyka IFJ16

Tým 013, varianta b/3/I
Rozšíření: BOOLOP

Šuba Adam	xsubaa00	25%	vedoucí týmu
Paliesek Jakub	xpalie00	25%	
Šuhaj Peter	xsuhaj02	25%	
Tóth Adrián	xtotha01	25%	

11. 12. 2016

Obsah

1 Úvod

Dokumentace popisuje vývoj interpretu imperativního jazyka IFJ16. Jazyk IFJ16 je podmnožinou jazyka Java SE 8, což je staticky typovaný, objektově-orientovaný jazyk. Úlohou našeho interpretu je kontrola zdrojového kódu napsaného v jazyku IFJ16 a jeho následná interpretace.

Zvolená varianta zadání Tým 013 má parametry b/3/I:

- (b) vyhledávání podřetězce s použitím Boyer-Mooreova algoritmu
- (3) řazení s použitím algoritmu Shell sort
- (I) implementace tabulky symbolů použitím binárního vyhledávacího stromu

Dokumentace je rozdělená na kapitoly, které blíže popisují jednotlivé části interpretu a způsob jejich řešení. Taktéž obsahuje i popis použitých algoritmů, rozdělení úloh a způsob práce v týmu. V závěru dokumentace se nachází shrnutí naší práce.
V příloze se nachází:

- diagram konečného automatu lexikální analázy
- LL-gramatika pro syntaktickou analázu shora dolů
- precedenční tabulka pro syntatickou analýzu zdola nahoru

2 Návrh

Plánování procesu vývoje interpretu nám bylo značně zjednodušeno závazně zadanými metodami implementace synktatické analýzy. V případě metody zhora dolů jsme se rozhodli pro rekurzivní sestup.

Samotný vývoj jsme naplánovali na 6 fází, s cílem mít při pokusném odevzdání plně funkční produkt.

1. Vývoj lexikálního analyzátoru
2. Vývoj syntaktického analyzátoru
3. Příprava tabulky symbolů
4. Vývoj sémantické analýzy a její integrace do syntaktického analyzátoru
5. Vývoj interpretu
6. Dokončení a vyladění celého produktu, tvorba dokumentace

Vývoj lexikálního analyzátoru zahrnuje přípravu konečného automatu rozpoznávajícího lexémy jazyka IFJ16 a jeho implementaci. První části vývoje syntaktického analyzátoru je implementace rekurzivního sestupu. To vyžaduje tvorbu LL-gramatiky a LL-tabulky. Druhou částí je implementace precedenční analýzy, která vyžaduje tvorbu precedenční tabulky. V mezičase je možné navrhnout a implementovat tabulku symbolů a připravit sémantickou analýzu pro integraci do synaktického analyzátoru. Jakmile je hotová kontrola správnosti vstupního programu a jsou generovány instrukce, je možné přistoupit

k implementaci interpretu. V průběhu vývoje jednotlivých modulů jsou připravovány testy jak samotných modulů, tak jejich spolupráce s ostatními moduly. Závěrečná fáze zahrnuje odstranění co nejvíce chyb a tvorba dokumentace.

3 Interpret

Implementaci interpretu lze rozdělit do 5 logických celků. Hlavní komponentou je syntaktický analyzátor, neboli "parser", který řídí činnost lexikálního analyzátoru, "scanneru", a také provádí sémantické akce sémantického analyzátoru. Sémantická analýza také vkládá informace o funkcicích a proměnných do tabulky symbolů. Poslední komponentou je samotný interpret, který provádí interpretaci instrukcí, vygenerovaných během kontroly vstupního programu.

Kontrola správnosti programu je prováděna ve dvou průchodech, v prvním se některé části zdrojového kódu úplně ignorují (zéjména výrazy, které jsou závislé na sémantických akcích, více dále). Účelem prvního průchodu je naplnění tabulky symbolů.

3.1 Lexikální analyzátor

Jediným úkolem lexikálního analyzátoru je na žádost syntaktického analyzátoru přečíst další symbol ze zdrojového souboru, klasifikovat jej a předat ve formě tokenu zpět syntaktickému analyzátoru.

Implementace je založena na konečném automatu. Bílé znaky a komentáře jsou ze zdrojového kódu odignorovány. Pokud řetězec dojde do koncového stavu reprezentující nějaký lexém a následující znak nepatří mezi znaky, které může daný lexém obsahovat a zároveň znak patří mezi znaky které mohou bezprostředně za daným lexémem následovat, je typ vraceného tokenu nastaven na typ odpovídající tomuto lexému. Některým tokenům je ještě přidělen atribut, obsahující užitečná data získaná z lexému. Pokud je v některém stavu přečten znak, který není v daném stavu podporován, popřípadě pokud je v koncovém stavu přečten znak, který nemůže následovat bezprostředně za daným lexémem, dojde k chybě. Poslední načtený znak, který rozhodl o ukončení konečného automatu, je vrácen zpět do vstupního souboru.

V případě lexikální chyby vrací interpet návratový kód 1.

Poznámka:
Při druhém průchodu syntaktické analýzy již není lexikální analyzátor činný. Při každém získání tokenu během prvního průchodu je daný token uložen do datového typu fronta a v druhém průchodu jsou tokeny brány z této fronty.

3.2 Syntaktický analyzátor

Mozkem celé kontroly správnosti zdrojového kódu je synktatický analyzátor. Samotnou syntaktickou analýzu můžeme rozdělit do dvou částí: kontrola rekurzivním sestupem a kontrola správnosti výrazů precedenční analýzou.

Rekurzivní sestup je metoda syntaktické analýzy shora dolů, která implementuje pravidla LL-gramatiky (viz příloha) na základě LL tabulky. Podle aktuálního stavu analyzátoru a vstupního tokenu je rozhodnuto, které pravidlo se zvolí. Metoda využívá rekurzivního volání funkce implementující dané pravidlo.

Pro kontrolu správnosti výrazů se využívá precedenční analýza, která je metodou syntaktické analýzy zdola nahoru. Neboť bylo nutné nějakým způsobem napojit precedenční analýzu na rekurzivní sestup, přistoupili jsme k zavedení speciálního pseudoterminálu E do naší LL-gramatiky. V místě kontroly, kde by se měl nacházet tento terminál, se analyzátor přepne do režimu, kdy pouze žádá další a další tokeny a ukládá je do datového typu fronta. Tento režim je ukončen v moment, kdy narazí na token, který není ve výrazech podporován. Takto vytvořená fronta je poté předána precedenční analýze ke kontrole. Speciální případem, kdy je nutné lehce pozměnit chování tohoto procesu, jsou výrazy podmínek ve větvení a v cyklu. Zde je nutné počítat levé a pravé závorky tak, aby byl analyzátor schopen poznat, kdy nastal konec podmínky a fronta je naplněná.

Neboť neimplementujeme rozšíření FUNEXP je pro nás funkce ifj16.print() speciální syntaktickou konstrukcí, neboť jí lze jako parametr dát jednoduchou konkatenaci, jejíchž výsledkem je vždy datový typ string. Z tohoto důvodu jsme se rozhodli nezahrnovat tohle chování do LL-gramatiky, ale vyřešit to jako speciální případ až při samotné implementaci analyzátoru. V druhém průchodu zdrojového souboru již díky sémantickým akcím analyzátor ví, že volaná funkce je ifj16.print() a na základě této informace se přepne do speciálního režimu, podobného přípavě na precedenční analýzu. Rozdíl je v tom, že povolené tokeny jsou pouze proměnné, literály a operátor konkatenace „+“. Při ukončení tohoto režimu je zkontrolováno, zda-li alespoň jeden token byl typu string. Poté je konkatenace vyhodnocena precedenční analýzou. V naší implementaci je tedy tato jednoduchá konkatenace vyhodnocována zleva doprava podle stejných pravidel jako jiné výrazy. Výsledkem `3 + 5 + "\n"` je tedy `"8\n"` a nikoliv `"35\n"`.

V případě syntaktické chyby vrací interpret návratový kód 2.

3.3 Sémantická analýza

Sémantická analýza je soubor funkcí a kontrol volaných během syntaktické analýzy v obou průchodech. V 1. průchodu do tabulky symbolů vkládá informace o třídách, statických funkcích, statických proměnných, argumentech funkcí a kontroluje, zda nedošlo k redefinici. Těla funkcí nejsou v 1. průchodu sémanticky zpracována. V druhé fázi jsou pak generovány instrukce inicializačních výrazů statických proměnných a do tabulky symbolů jsou vkládány lokální proměnné. Rovněž jsou v této fázi zpracována těla funkcí, což zahrnuje kontrolu typů, kontrolu, zda byl identifikátor při jeho použití (u volání funkce, přiřazení) již definován a následné generování instrukcí, pokud nenastaly chyby zmíněné dříve.

V případě sémantické správnosti kódu je během precedenční analýzy vždy při redukci části výrazu na neterminál generovaná aritmetická instrukce, případně i dodateční konverzní instrukce, pokud je daná implicitní konverze podporovaná. Výstupem sémantické akce generující aritmetickou instrukci je pomocná proměnná, která je během precedenční analýzy přiřazena jako atribut cílovému neterminálu a může být dále použita při jiné redukci.

3.4 Tabulka symbolů

Tabulka symbolů (dále jen TS) je podle zadání implementovaná pomocí binárního vyhledávacího stromu. Vzhledem k tomu, že v jazyce IFJ16 existují různé rozsahy platnosti identifikátorů a třídy umožňují vytvářet oddělené jmenné prostory, navrhli jsme TS jako tříúrovňovou kaskádu vyhledávacích struktur. První úroveň je jediná tabulka tříd. Každá třída má potom vlastní tabulku členů (společnou pro statické proměnné a funkce). Funkce mají dále vlastní tabulky lokálních proměnných.

Hodnoty statických proměnných jsou uložené přímo v odpovídajících položkách TS. Hodnoty lokálních proměnných nejsou, kvůli možnému rekurzivnímu volání funkcí, uložené přímo v TS, ale jejich hodnoty si ukládá interpret na svém zásobníku během vykonávání instrukcí.

Různé typy proměnných (statická, lokální, pomocná) a literály jsou v naší implementaci reprezentovány různými strukturami, ale při sémantických akcích často není potřeba mezi nimi rozlišovat. Kvůli tomuto sme zavedli jednu extra strukturu, která je umístěna jako první člen ostatních struktur. Obsahuje informaci o tom, na kterou strukturu lze ukazatel na tuto univerzální strukturu dále přetypovat v případě potřeby.[1] Krom toho ještě obsahuje informaci o datovém typu, který může sémantická analýza skontrolovat bez nutnosti přetypování.

3.5 Interpret

Interpret obsahuje jednu velkou funkci, jejímž vstupem je instrukční páska. Funkce se pomocí cyklu posouvá po instrukční pásce až do dosažení konce, tj. na instrukční pásce je `NULL`.

Podle operačního kódu instrukce se zvolí příslušná větev interpretu a pomocná dekódovací funkce získá z instrukce operandy (jejich hodnotu, informaci o tom, jestli jsou inicializované apod.). Pro rámce funkcí má interpret svůj vlastní zásobník. Při volání funkce se vytvoří nový rámec, vyplní se potřebnými hodnotami (argumenty funkce, místo pro návratovou hodnotu, ukazatel na instrukci pro návrat) a vloží se na zásobník. Při návratu z funkce je návratová hodnota uložena na místa v rámci pro ni vymezeného a interpretace pokračuje od poznačené instrukce.

4 Algoritmy

Všechny popsané algoritmy jsou implementovány v souboru `ial.c`

4.1 Vyhledávání podřetězce použitím Boyer-Mooreova algoritmu (b)

Algoritmus byl objeven Robertem S. Boyerem a J Strotherem Moorem v roce 1977. Jedná se o nejefektivnější algoritmus na vyhledávání podřetězce v řetezci. Funguje na základě dvou heuristik a na principu zpracování řetězce zprava doleva. Podřetězec je vyhodnocován od posledního znaku k prvnímu. Heuristické funkce spracují podřetězec nejprve do pole, kam se uloží hodnoty skoků. ????

4.2 Řazení použítím algoritmu Shell sort (3)

Shell sort je řadící algoritmus, který pracuje na principu bublinového vkládání. Neřadí prvky nacházející se výhradně vedle sebe, ale i posloupnosti oddělené určitým počtem prvků (krokem). V naší implementaci je na počátku tento krok stanoven na polovinu počtu prvků. Při každé další iteraci je hodnota kroku snížena na polovinu. V různých implementacích existují jiné způsoby určení těchto kroků, od čeho se odvijí jejich časová složitost. Při vyšších hodnotách kroku se neuspořádanost snižuje rychleji. Algoritmus skončí jakmile krok dosáhne hodnoty 0. Shell sort pracuje „in situ“ a je nestabilní, což nám

[1] A pointer to a structure object, suitably converted, points to its initial member and vice versa. C Standard, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, s. 103

nevadí. Ve srovnání s jinými algoritmy je pomalejší než Quicksort a rychlejší než Heapsort. Složitost naší implementace je $O(n^2)$.

4.3 Implementace tabulky symbolů použitím binárního vyhledávacího stromu (I)

Binární vyhledávací strom je binární strom, platí tedy, že je složen z kořene a levého a pravého podstromu. Všechny uzly nalevo od kořene mají klíče menší než kořen a všechny uzly napravo mají klíče větší než kořen. Vyhledávacím klíčem je v naší implementaci identifikátor symbolu. Funkce pro práci s binárními vyhledávacími stromy (v našem případě vyhledání uzlu, vložení uzlu a zrušení celého stromu průchodem PostOrder) byly implementovány rekurzivně. Implementace samotné tabulky symbolů byla popsána detailně v kapitole 3.4.

5 Práce v týmu

5.1 Rozdělení týmu

- Adam Šuba – lexikální analyzátor, syntaktická analyza shora dolů a zdola nahoru, dokumentace, vedoucí
- Jakub Paliesek – sémantický analyzátor, tabulka symbolů, algoritmy, generátor 3-adresného kódu, dokumentace
- Peter Šuhaj – interpret, algoritmy, dokumentace
- Adrián Tóth – testy, algoritmy, dokumentace

5.2 Způsob práce

Pro práci byl zvolen systém správy verzí Git, hostovaný na serveru GitHub. Kromě osobní komunikace byla hlavním komunukačním kanálem skupinová konverzace na sociální síti Facebook.

6 Závěr

Tým pracoval spolehlivě a časový plán se nám dařilo plnit. Jediné období, kdy nebylo možné na projektu aktivně pracovat bylo během působemestrálního období, s čímž jsme ovšem během plánování počítali a problémy to tedy nezpůsobilo. Při prvním pokusném odevzdání jsme měli až na několik drobných chyb funkční interpret.

7 Literatura

1. HONZÍK, Jan M., Algoritmy: Studijní opora. Verze: 16-B, Brno: Vysoké učení Technické, 2014
2. https://sk.wikipedia.org/wiki/Shell_sort

8 Příloha

8.1 LL-gramatika

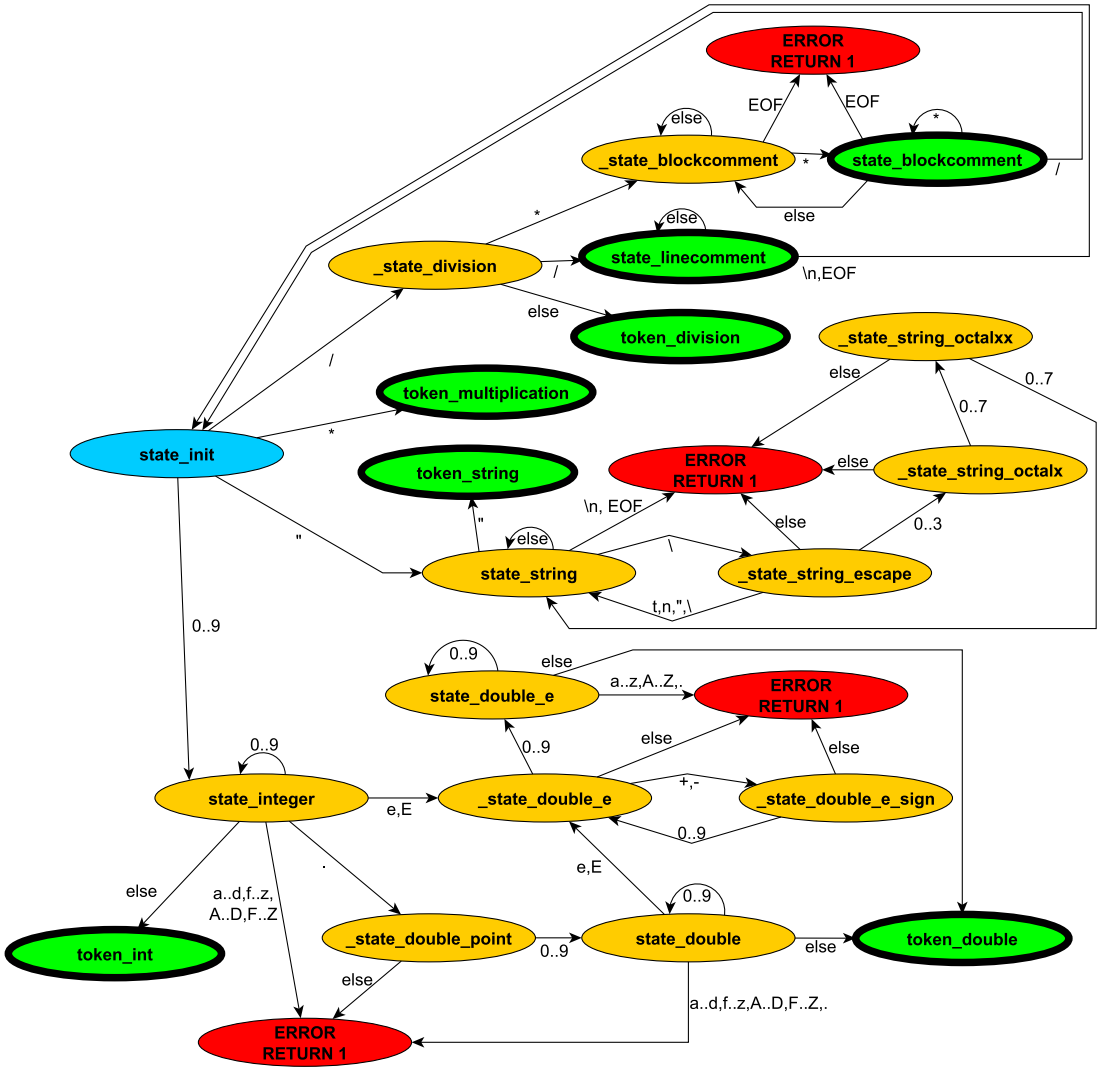
1. <c-list> → class id { <memb-list> } <c-list>
2. <c-list> → $
3. <memb-list> → <c-memb> <memb-list>
4. <memb-list> → ε
5. <c-memb> → static <c-memb1>
6. <c-memb1> → <type> id <c-memb2>
7. <c-memb1> → void id <c-membfunc>
8. <c-memb2> → = E ;
9. <c-memb2> → ;
10. <c-memb2> → <c-membfunc>
11. <c-membfunc> → (<fn-def-plist>) { <fn-body> }
12. <type> → int
13. <type> → double
14. <type> → String
15. <type> → boolean
16. <fn-def-plist> → <par-def> <fn-def-plist1>
17. <fn-def-plist> → ε
18. <fn-def-plist1> → <par-def> <fn-def-plist1>
19. <fn-def-plist1> → ε
20. <par-def> → <type> id
21. <fn-body> → <stat> <fn-body>
22. <fn-body> → <type> id <opt-assign> ; <fn-body>
23. <fn-body> → ε
24. <stat-com> → { <stat-list> }
25. <stat-list> → <stat> <stat-list>
26. <stat-list> → ε
27. <stat> → <id> <as-ca> ;
28. <stat> → <stat-com>
29. <stat> → if (E) <stat-com> else <stat-com>
30. <stat> → while (E) <stat-com>
31. <stat> → return <ret-val> ;
32. <id> → id
33. <id> → fqid
34. <as-ca> → (<fn-plist>)
35. <as-ca> → = <assign>

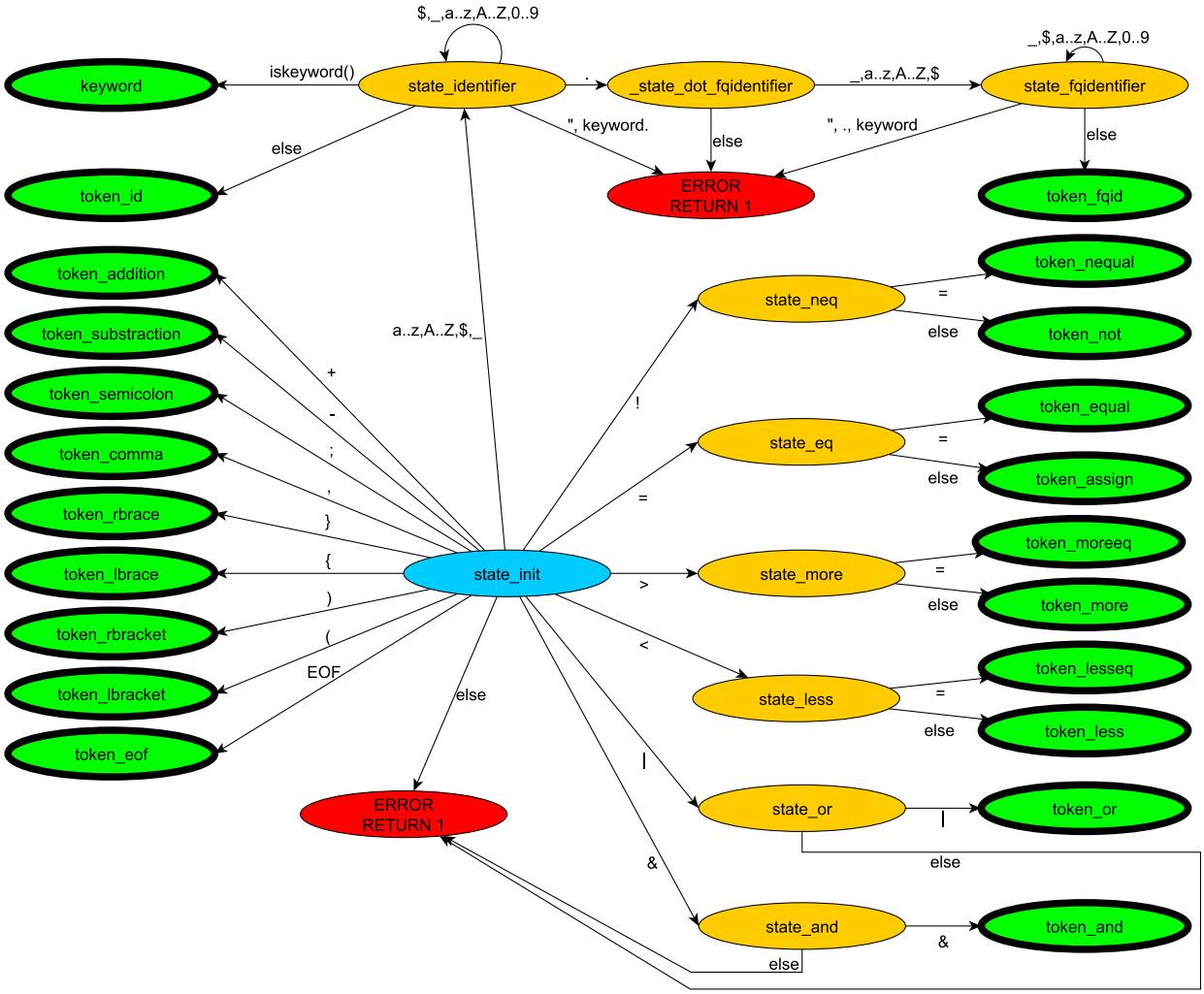
36. <assign> → E
37. <assign> → <id> (<fn-plist>)
38. <opt-assign> → = <assign>
39. <opt-assign> → ε
40. <fn-plist> → <val-id> <fn-plist1>
41. <fn-plist> → ε
42. <fn-plist1> → , <val-id> <fn-plist1>
43. <fn-plist1> → ε
44. <val-id> → <id>
45. <val-id> → int-literal
46. <val-id> → double-literal
47. <val-id> → string-literal
48. <val-id> → boolean-literal
49. <ret-val> → E
50. <ret-val> → ε

8.2 Precedenční tabulka

	+	-	*	/	<	>	<=	>=	==	!=	(	)	&&	||	!	id	$
+	>	>	<	<	>	>	>	>	>	>	<	>	>	>	<	<	>
-	>	>	<	<	>	>	>	>	>	>	<	>	>	>	<	<	>
*	>	>	>	>	>	>	>	>	>	>	<	>	>	>	<	<	>
/	>	>	>	>	>	>	>	>	>	>	<	>	>	>	<	<	>
<	<	<	<	<					>	>	<	>	>	>	<	<	>
>	<	<	<	<					>	>	<	>	>	>	<	<	>
<=	<	<	<	<					>	>	<	>	>	>	<	<	>
>=	<	<	<	<					>	>	<	>	>	>	<	<	>
==	<	<	<	<	<	<	<	<	>	>	<	>	>	>	<	<	>
!=	<	<	<	<	<	<	<	<	>	>	<	>	>	>	<	<	>
(	<	<	<	<	<	<	<	<	<	<	<	=	<	<	<	<	
)	>	>	>	>	>	>	>	>	>	>		>	>	>	>		>
&&	<	<	<	<	<	<	<	<	<	<	<	>	>	>	<	<	>
||	<	<	<	<	<	<	<	<	<	<	<	>	<	>	<	<	>
!	>	>	>	>	>	>	>	>	>	>	<	>	>	>	<	<	>
id	>	>	>	>	>	>	>	>	>	>		>	>	>	>		>
$	<	<	<	<	<	<	<	<	<	<	<		<	<	<	<	

8.3 Diagram konečného automatu lexikální analýzy




$,_,a..z,A..Z,0..9
keyword
iskeyword()
state_identifier
.
_state_dot_fqidentifier
_,a..z,A..Z,$
state_fqidentifier
_,$,a..z,A..Z,0..9
else
", keyword.
else
", ., keyword
else
token_id
ERROR
RETURN 1
token_fqid
token_addition
token_substraction
token_semicolon
token_comma
token_rbrace
token_lbrace
token_rbracket
token_lbracket
token_eof
a..z,A..Z,$,_
+
-
;
,
}
{
)
(
EOF
else
state_init
!
=
>
<
|
&
state_neq
=
else
token_nequal
token_not
state_eq
=
else
token_equal
token_assign
state_more
=
else
token_moreeq
token_more
state_less
=
else
token_lesseq
token_less
state_or
|
token_or
else
state_and
&
token_and
else
ERROR
RETURN 1