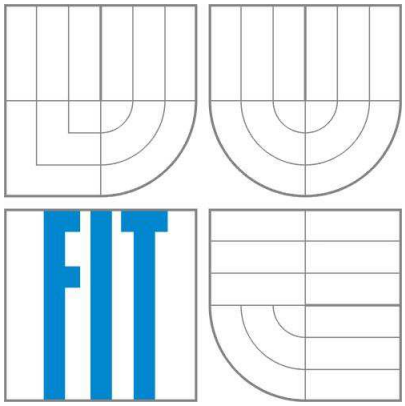
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Dokumentácia k projektu do predmetu IFJ a IAL

Interpret imperatívneho jazyka IFJ11

Tím 106, varianta b/1/II
Rozšírenia: LENGTH, MODULO, REPEAT, IFTHEN
LOGOP, LOCALEXPR, VOIDCALL

Zoznam riešiteľov:

Miroslav Lisik	xlisik00 – 25%
Pavol Loffay	xloffa00 – 25%
Dušan Maďarka	xmadar01 – 25%
Fridolín Pokorný	xpokor32 – 25% – vedúci tímu

22. apríla 2016

Obsah

1 Úvod

Tento dokument popisuje implementáciu a vnútornú štruktúru interpretru jazyka IFJ11, ktorý je podmnožinou skriptovacieho jazyka Lua. Navrhnutý program pracuje ako konzolová aplikácia, ktorej je ako parameter z príkazového riadku predaný súbor popisujpci algoritmus v jazyku IFJ11 a následne ho intepretuje. Aplikácia môže využívať štandardný vstup a štandardný výstup pre komunikáciu s užívateľom.

Dokument sa skladá z niekoľkých častí. V kapitole 2 sa venujeme analýze problému jednotlivých častí interpretru. Kapitola 4 sa zaoberá algoritmom a abstraktným datovým štruktúram navrhnutým pre implementáciu vnútornej štruktúry interpretru.

2 Analýza problému a princíp jeho riešenia

Cieľom úlohy je implementácia interpretru jazyka IFJ11, ktorý je podmnožinou jazyka Lua. Interpret je implementovaný v programovacom jazyku C podľa normy ISO-C99. Interpret sa skladá zo štyroch základných častí: lexikálna analýza, syntaktická analýza, sémantická analýza a samotný interpret, ktorý interpretuje predom vytvorený interný trojadresný kód. Zdrojový súbor bude spracovaný lexikálnou analýzou a následne bude vytvorená vnútorná štruktúra popisujúca algoritmus zapísaný v predanom zdrojovom súbore pre jeho interpretáciu.

Interpret je schopný detekovať chyby a adekvátne na ne reagovať chybovou hláškou pre užívateľa. Typ chyby je následne predaný operačnému systému ako navratová hodnota.

Jazyk IFJ11 je navrhnutý ako podmnožina skriptovacieho jazyka Lua. Jeho presný popis je možné nájsť v zadaní projektu [1].

3 Práca v tíme

Interpret bol implementovaný štvorčlenným tímom. Tím sa pravidelne stretával na predom dohodnutých schôdzach kde sa rozoberali implementačné problémy. Počas prvých stretnutí bolo navrhnuté celkové rozhranie projektu. Na základe návrhu boli na začiatku rozdelené úlohy podľa individuálnych znalostí členov tímu. Práca v tíme bola dynamická – úlohy boli časom prerozdelované na základe riešení novovzniknutých problémov. Pri vývoji boli využité znaky agilných metodík – automatizované testovanie, písanie testov pred implemetáciou, pravidelné stretnutia a pod..

Pri vývoji bol využitý nástroj *Subversion*[1] a systém *Redmine*[2], ktorý nám uľahčil komunikáciu a sprehľadnil vývoj projektu. Použili sme ho aj pre generovanie štatistík pre jednotlivých členov tímu a pre jednotlivé časti projektu. Taktiež nám pomohol sledovať vývoj projektu v čase, z čoho sme dokázali usúdiť plnenie termínov.

4 Popis riešenia

4.1 Tabuľka symbolov

Tabuľka symbolov je implementovaná ako hashovacia tabuľka podľa zadania. Jednotlivými položkami tabuľky symbolov sú záznamy reprezentujúce informácie o funkcii. Každá funkcia má názov, počet parametrov, zoznam inštrukcii, ktoré sa môžu vykonávať nelineárne (obsahuje skoky), ktoré operujú nad premennými. Premenné jednotlivých funkcií sú odkazované z položky tabuľky symbolov a sú implementované pomocou lineárneho zoznamu. Špeciálnou premennou je návratová hodnota funkcie, ktorá je obsiahnutá už v položke hashovacej funkcie.

Takáto implementácia uľahčuje predávanie parametrov, pretože prvými položkami lineárneho zoznamu sú parametre funkcie. Počet parametrov je známy, preto pri predávani parametrov funkcii je jednoduchá iniciali-

[1] `http://subversion .tigris.org/`
[2] `http://www.redmine.org`

zácia parametrov na `nil`, prípadne kopírovanie hodnotou, či dokonca zahadzovanie premenných pri uvedení nadbytočných premenných pri volaní funkcie.

Každá funkcia má vygenerovaný trojadresný kód, ktorého operandy sú pevne dané. Pri volaní podprogramu sa všetky premenné funkcie nakopírujú do zoznamu odkazovaného z programového zásobníka, kde je uložená aj návratová adresa podprogramu. Záloha je nutná pri rekurzii, aby nedošlo k prepísaniu a tým k strate dát volajúcej funkcie pri vykonávaní volanej funkcie.

4.2 Boyer – Moorov algoritmus

Podľa zadania bolo požadované implementovať vstavanú funkciu `find`, ktorá vyhľadá podreťazec v reťazci znakov. Vybrali sme Boyer – Moorov algoritmus kvôli jeho časovej efektívnosti. Jeho účinnosť spočíva v tom, že neporovnáva každý znak v vyhľadávanom reťazci. Niektoré znaky za splnených podmienok môže preskočiť. Pričom platí, že čím dlhší je vyhľadávaný reťazec, tím väčší počet znakov môže preskočiť. Tento algoritmus s použitím známych heuristík, býva podstatne rychlejší ako Knuth – Morris – Prattov algoritmus.

Je zaujímavý tým že vyhľadávaný podreťazec porovnáva s vzorom sprava doľava. Ak vo vzore narazí na znak, s ktorým sa nezhoduje a daný znak vyhľadávaný podreťazec neobsahuje, vyhľadávanie sa môže posunúť o celú dĺžku podreťazca. Naopak ak sa dané znaky nezhodujú ale podreťazec ho obsahuje, posunie sa tak, aby pozície zhodných znakov súhlasili.

4.3 Quick sort

Vstavaná funkcia `sort` využíva algoritmus Quick sort. Vzľadom na to, že sa využíva radenie reťazových literálov je postačujúca rekurzívna implementácia. Pseudomedián (tiež uvádzaný ako pivot) je vyberaný prostredný znak reťazového literálu. Volanie funkcie `sort` zabezpečuje inštrukcia `sort`, ktorá vytvára kópiu zo zdrojového operandu, nakoľko algoritmus pracuje in situ. Je tak zabránené prípadnej straty užívateľských dát.

4.4 Lexikálny analyzátor

Lexikálny analyzátor je navrhnutý a implementovaný ako deterministický konečný automat (ďalej len DKA). Bol navrhovaný postupne – pre každú platnú lexému bol navrhnutý DKA, pričom neskôr boli tieto DKA zlúčené do jedného. Štruktúra DKA lexikálneho analyzátora je samostatne umiestnená v kapitole B.

Pomocou prechodov medzi stavmi konečného automatu sú definované platné lexémy jazyka. V prípade neplatnej lexémy vráti chybu, pričom je celý interpret ukončený chybovým hlásením. Ak je načítaný koniec súboru, je vrátený špeciálny token `TKN_EOF`. Pri spracovaní lexémy identifikátora sa zisťuje či nieje daný identifikátor kľúčové alebo rezervované slovo, príp. názov vstavanej funkcie. Toto zabezpečuje funkcia `scanner_is_special_word()`, ktorá v tabuľke kľúčových slov vyhľadáva daný identifikátor. Ak vyhľadávanie skončí úspešne, je token zmenený z tokenu identifikátora na špeciálny token, individuálny pre každé rezervované alebo kľúčové slovo, príp. názov vstavanej funkcie.

Lexikálny analyzátor používa pri prechode súborom cyklický kruhový buffer, ktorý je implementovaný tak, aby v prípade načítania ďalšieho znaku mohol byť použitý už predtým načítaný znak, ktorý bol vrátený do buffera z dôvodu ukončenia lexémy.

4.5 Syntaktický analyzátor

Syntaktický analyzátor kontroluje syntax zdrojového textu rekurzívnym zostupom. Avšak vyhodnocovanie výrazov je implementované precedenčnou analýzou. Je to dôkladná kombinácia postupu zhora – nadol, a naopak zdola – nahor. Týmto spôsobom sme schopní efektívne vykonávať syntaktické akcie. Tento modul sa taktiež stará o generovanie trojadresných inštrukcií.

4.5.1 Rekurzívny zostup

Rekurzívny zostup je implementovaný podľa pravidiel LL-gramatiky vytvorenej na základe popisu zadania jazyka IFJ11. V priebehu rekurzívneho zostupu sa volá funkcia lexikalného analyzátoru, vykonávajú sa sémanticke akcie a generuje sa trojadresný kód pre inštrukcie. Tabuľka zoznamu pravidiel našej gramatiky sa nachádza v kapitole C.

4.5.2 Precedenčná analýza

Na spracovávanie výrazov sme použili požadovanú precedenčnú analýzu. Pre správne určenie priority jednotlivých operácií využíva precedenčnú tabuľku priorít. Kde sú presne určené priority operátorov nášho jazyka. Pracuje systémom zdola – nahor.

Je súčasťou syntaktickej analýzy. Syntaktický analyzátor volá precedenčnú analýzu v prípade, ak narazí na obsah zdrojového textu programu, kde sa nachádza výraz. Preto musí popri vyhodnocovaní výrazov korektne kontrolovať syntax a sémantiku prevádzaných operácii. Súčasťou je taktiež generovanie trojadresného kódu. Inštrukcia sa vygeneruje vždy pri aplikácii redukčného pravidla.

Precedenčnú analýzu sme implementovali ako zásobníkový automat. V zásobníku sú uložené terminály v podobe operátorov a neterminály. Ktoré obsahujú ukazateľ do tabuľky symbolov na dané premenné. Tie môžu byť predom definované identifikátory, literály z výrazov, alebo odkazy na premenné, kde sa bude nachádzať výsledok už aplikovaných pravidiel. Veľkosť zásobníka je obmedzená, avšak zvolili sme ju dostatočnú, aby bolo možné spracovať výrazy s veľkým počtom zanorených zátvoriek. Podľa tabuľky sa určí, či výraz na vrchole zásobníka sa má zjednodušiť, alebo sa vloží nová položka – aktuálny token na vrchol.

$E \to id$	$E \to E/E$	$E \to not\ E$	$E \to E\ >\ E$
$E \to (E)$	$E \to E\ \%\ E$	$E \to E\ and\ E$	$E \to E\ >\ E$
$E \to E + E$	$E \to E\ \hat{}E$	$E \to E\ or\ E$	$E \to E\ <=\ E$
$E \to E - E$	$E \to E..E$	$E \to E == E$	$E \to E\ <=\ E$
$E \to E * E$	$E \to \#E$	$E \to E \sim= E$	

Tabuľka 1: Tabuľka zoznamu pravidiel.

4.6 Semantický analyzátor

Sémantické kontroly sa z jednoduchosti prevádzajú iba na literáloch. Pretože u premenných v dobe generovania inštrukcií nevieme jednoznačne určiť ich typ. Kontrola typov sa vykonáva pri zjednodušovaní výrazov v precedenčnej analýze. Sémantické kontroly identifikátorov sa vykonávajú v interprete. Prípadné nezhody sú vyhodnotené ako chyby interpretácie.

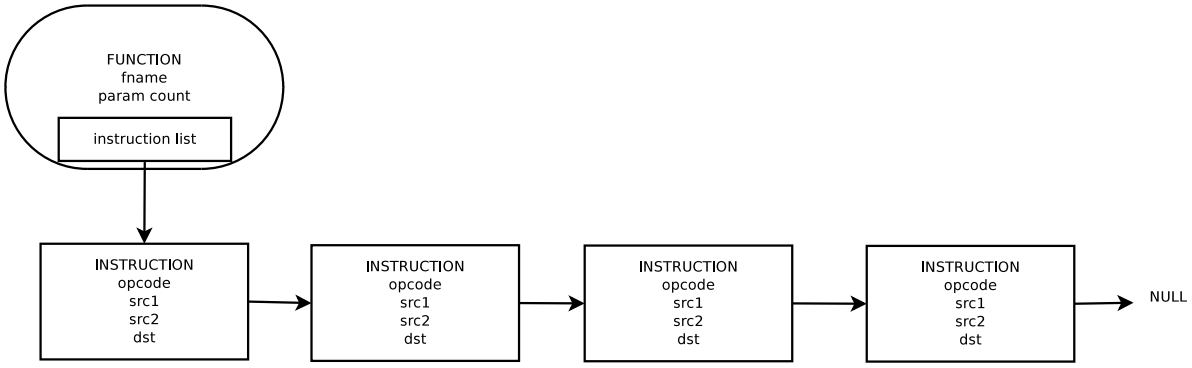
4.7 Interpret

Intepret prechádza lineárny zoznam inštrukcii a podľa sémantickej akcie danej významom práve vykonávanej inštrukcie operuje nad operandmi inštrukcie. Inštrukčná sada je trojadresná, preto každá inštrukcia má tri operandy – jeden cieľový a dva zdrojové. V inštrukčnej sade však existujú výnimky ako napríklad inštrukcia skoku, kde nie sú využité všetky operandy inštrukcie.

Interpret detekuje chyby, ktoré sa vyskytnú až za behu programu (príkladom môže byť delenie nulou).

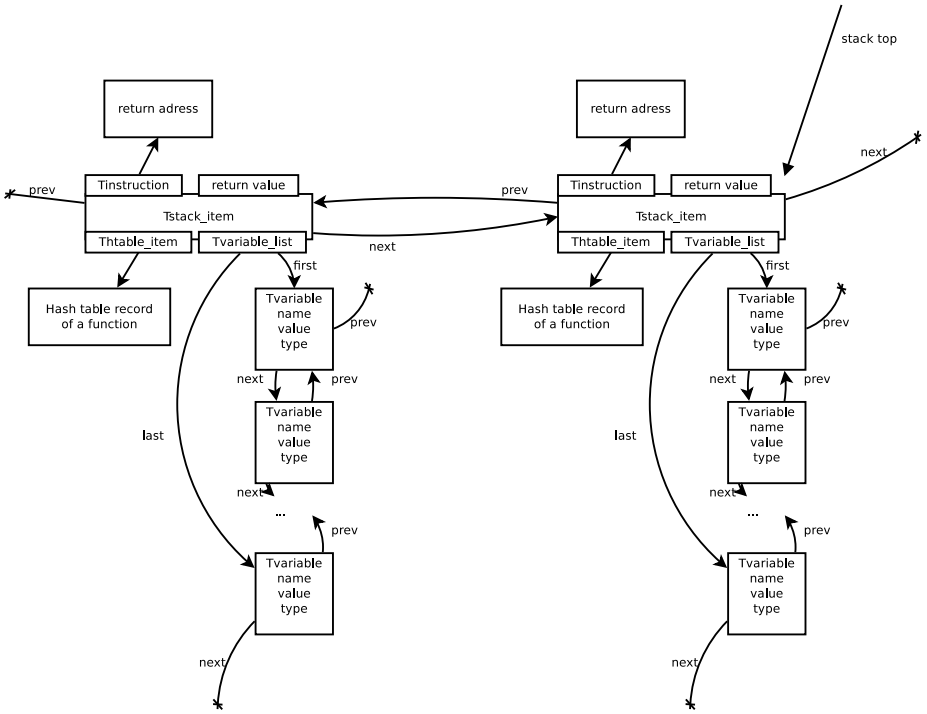
Operandom inštrukcie môže byť položka v tabuľke symbolov (inštrukcia `enter`, ktorá vytvára zálohu), premenná alebo inštrukcia v závislosti od sémantiky inštrukcie.

V niektorých prípadoch semantická analýza prenecháva kontrolu typov operandov pre interpret z dôvodu jednoduchosti. Dochádza k tomu hlavne pri premenných, ktorým hodnota je priradená vstavaným príkazom `read`. Preto interpret detekuje takéto chybné stavy pri aritmetických operáciach ako aj operáciach porovnánia.



Obr. 1: Lineárny zoznam inštrukcií.

Interpret ďalej detekuje chybu pri práci s nepovolenými hodnotami premenných interne typovaných na `double` - pretečenie na `NAN` a `INFINITY`, `-INFINITY`. [3].



Obr. 2: Programový zásobník.

Inštrukcia `substr` je zvláštna svojím vyhodnocovaním. Ako jediná z inštrukčnej sady prepisuje zdrojový operand, ktorý je zároveň cieľovým operandom. Je tak z dôvodu, že vstavaná funkcia má tri parametre. Pre jednoduchú implementáciu bola navrhnutá inštrukcia `substr`, ktorá je taktiež trojadresná, no výsledok operácie je zapísaný do cieľového operandu, v ktorom je predaný reťazec. Preto sa výnimočne v tomto prípade vygeneruje dodatočná inštrukcia, ktorá vytvorí kópiu prvého parametru vstavanej funkcie `substr` a vloží ho do tabuľky symbolov. Táto kópia môže byť následne prepísaná bez prípadnej straty dát.

Narozdiel od vstavanej funkcie `substr` je vstavaný príkaz `write` zvláštny svojím počtom parametrov. Zjednodušenie implementácie viedlo k tomu, že bola navrhnutá jednoduchá inštrukcia `write`, ktorá vypisuje

[3] Definované v štandardnom hlavičkovom súbore `float.h`

práve jeden parameter. Pokiaľ bol vstavaný príkaz volaný s viacero parametrami, vygeneruje sa samostatná inštrukcia `write` pre každý parameter.

Inštrukcia `leave` je určená pre návrat z funkcie definovanej užívateľom. Inštrukcia `leave` je automaticky vkladaná na koniec funkcie a za príkaz `return`. To spôsobí korektný návrat z podprogramu.

4.8 Voľba datových typov

Interpret jazyka IFJ11 nie je typovým jazykom. Avšak interpret je písaný v jazyku C, ktorý je typovým jazykom, preto je nutné interne určovať datový typ, ktorý aktuálne nadobúda premenná. Pre typ `string` je použitý ukazateľ na pole `char`, pre typ `bool` je použitá premenná typu `_Bool`, pre čísla typ `double` [4]. Typ `double` je vhodný pre svoj rozsah a presnosť, ktorá je požadovaná. Špeciálnym prípadom je typ `nil`, ktorý nenadobúda žiadnu hodnotu.

Pre prácu s premennými bol vytvorený abstraktný datový typ – štruktúra `Tvariable`, ktorú možno nájsť v hlavičkovom súbore `variable.h`. Táto štruktúra zapuzdruje všetky spomínané typy, ktoré podporuje jazyk IFJ11.

V lexikálnom analyzátore bol použitý abstraktný datový typ `string`, ktorý je definovaný v súboroch `str.h` a `str.c`. Ako základ bol použitý vzorový interpret jednoduchého jazyka zo stránok predmetu IFJ, ktorého autorom je Ing. Roman Lukáš PhD. Zdrojové súbory boli upravené pre použitie v interpretri IFJ11 (najmä pomenovanie metód/funkcií nad abstraktným datovým typom).

4.9 Volanie podprogramu a návrat

Imperatívny jazyk IFJ11 podporuje definovanie užívateľských funkcií s parametrami, ich volanie a následný návrat na miesto volania. Pre tento požiadavok boli navhnuté inštrukcie `enter`, `call` a `leave`.

Inštrukcia `enter` je prologom pred samotným volaním podprogramu. V dobe interpretácie je touto inštrukciou interpret informovaný, že nastane volanie podprogramu. Parametrom inštrukcie je odkaz do tabuľky symbolov na položku reprezentujúcu volajúcu funkciu. Interpret následne môže zahájiť tvorbu aktivačného záznamu na programovom zásobníku. Skopíruje hodnoty a typy všetkých premenných a vytvorí tak kópiu aktuálneho stavu premenných do zinicializovanej položky na programovom zásobníku. Nasleduje sled inštrukcií `mov` a `clr`.

Po vykonaní inštrukcie `enter` nastáva predávanie parametrov. Vygenerované inštrukcie predávajú hodnoty premenných priamo do zoznamu premenných odkazovaných z položky v tabuľke symbolov (inštrukcie `mov`) prípadne ich nulujú (inštrukciou `clr`) v prípade neuvedenenia dostatočného počtu.

Prípravu volania podprogramu ukončuje inštrukcia `call`, ktorá na vrchol programového zásobníka uloží adresu nasledovnej inštrukcie, ktorá bude vykonaná po návrate z podprogramu. Nasleduje vykonávanie prvej inštrukcie podprogramu.

Každá funkcia musí obsahovať aspoň jednu inštrukciu `leave`. Inštrukcia `leave` sa generuje na úplnom konci funkcie a po príkaze `return`, ktorý najprv predá návratovú hodnotu. Pri interpretácii inštrukcie `leave` sa obnoví záloha premenných funkcie, do ktorej sa podprogram vracia a zruší položky na vrchole zásobnika. Inštrukcia `leave` je preto komplementárna s inštrukciou `enter` a je epilogom pri volaní funkcie.

5 Záver

Projekt IFJ11 bol naším prvým tímovým projektom. Počas jeho tvorby sme si rozšírili svoje znalosti a získali skúsenosti s prácou na rozsiahlejšom projekte a prácou v tíme. Spoznali sme náročnosť rôznych fáz tvorby projektu od návrhu až po testovanie. Využili sme pokusné odovzdanie, ktoré nám pomohlo získať predstavu o stave jednotlivých častí projektu, vďaka čomu sme sa vedeli zamerať na slabšie miesta našej implementácie. Projekt hodnotíme veľmi pozitívne. Nadobudnuté znalosti uplatníme v profesionálnej kariére.

[4] Číslo s plávajúcou desatinnou čiarkou s dvojitou presnosťou podľa štandardu ISO-IEEE 754.

Literatúra

[1] KŘOUSTEK, J., KŘIVKA, Z. a VRÁBEL, L. *Zadání projektu z předmětu IFJ a IAL.* září 2011.

A Metriký kódu

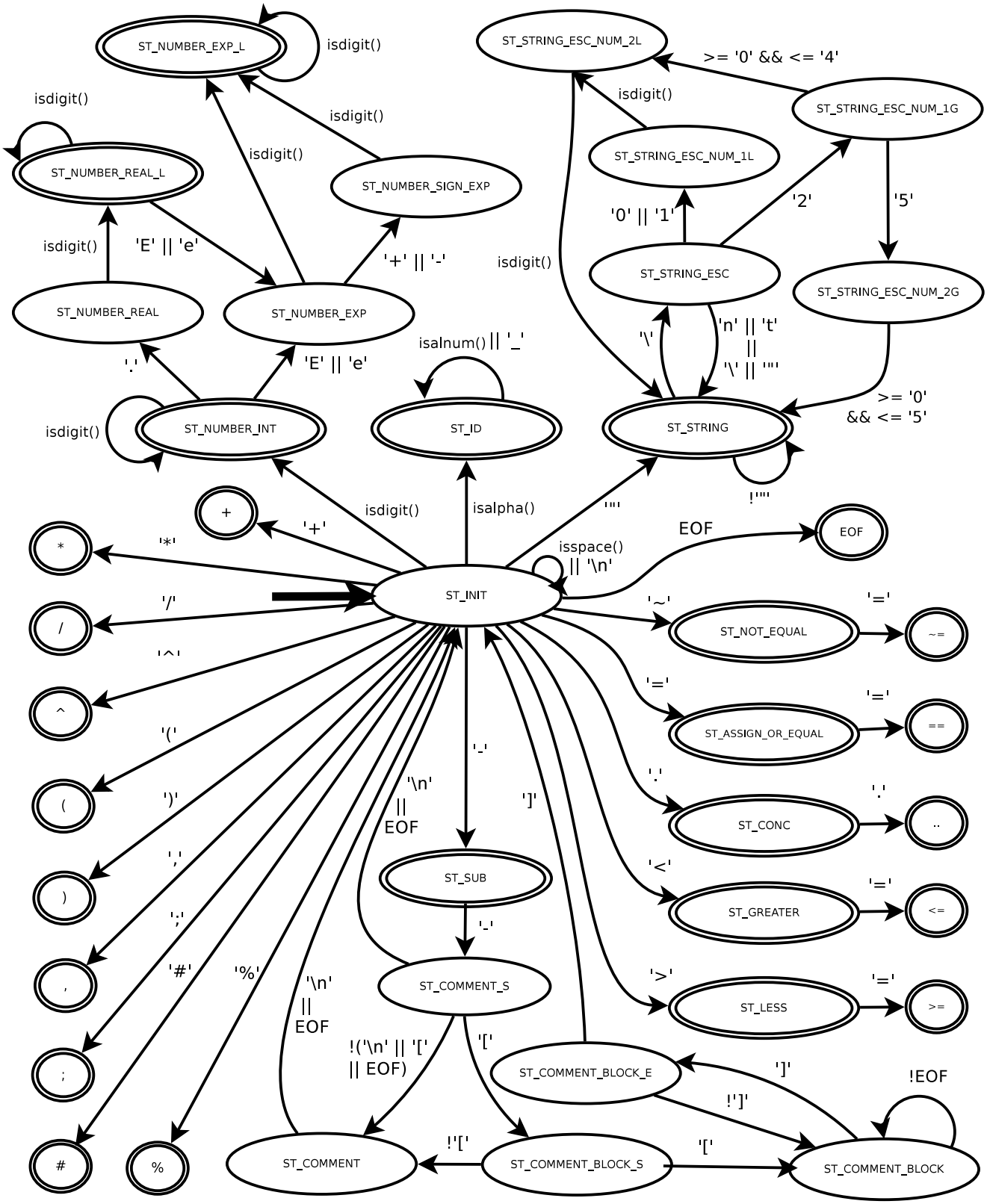
Počet súborov: 30 súborov

Počet riadkov zdrojového kódu: 9378 riadkov

Veľkosť statických dát: 968B

Veľkosť spustiteľného súboru: 52kB (systém GNU/Linux, 64 bitová architektúra, pri preklade bez ladiacich informácií)

B Štruktúra konečného automatu lexikálneho analyzátora



Obr. 3: Konečný automat lexikálneho analyzátora

C Tabuľka pravidiel LL-gramatiky

1.	<prog>	→	<df-list> EOF
2.	<df-list>	→	FUNCTION ID (<param-list> <var-list>
3.	<param-list>	→	)
4.	<param-list>	→	ID <param>
5.	<param>	→	, ID <param>
6.	<param>	→	)
7.	<var-list>	→	LOCAL ID <dclr-type> <var-list>
8.	<var-list>	→	<stat-list>
9.	<dclr-type>	→	;
10.	<dclr-type>	→	= <expr> ;
11.	<stat-list>	→	<stat > ; <stat-list>
12.	<stat-list>	→	END <end>
13.	<stat>	→	ID <id-or-fun>
14.	<stat>	→	WRITE (<w-args-list>
15.	<stat>	→	RETURN <expr>
16.	<stat>	→	WHILE <expr> DO <stat-list>
17.	<stat>	→	REPEAT <stat-repeat-list> <expr>
18.	<stat>	→	IF <expr > THEN <stat-if-list>
19.	<end>	→	;
20.	<end>	→	<df-list >
21.	<stat-repeat-list>	→	UNTIL
22.	<stat-repeat-list>	→	<stat> ; <stat-repeat-list>
23.	<stat-if-list>	→	<stat> ; <stat-if-list>
24.	<stat-if-list>	→	ELSE <stat-list>
25.	<stat-if-list>	→	ELSEIF <stat-if-list>
26.	<stat-if-list>	→	END <end>
27.	<id-or-fun>	→	= <rfe>
28.	<id-or-fun>	→	(<args-list>
29.	<w-args-list>	→	<expr> <w-arg>
30.	<w-arg>	→	, <expr > <w-arg>
31.	<w-arg>	→	)
32.	<rfe>	→	READ (<read-format>)
33.	<rfe>	→	ID <fe >
34.	<fe>	→	<expr >
35.	<fe>	→	(<args-list>
36.	<args-list>	→	)
37.	<args-list>	→	<lit-or-id> <arg>
38.	<arg>	→	, <lit-or-id> <arg>
39.	<arg>	→	)
40.	<lit-or-id>	→	ID
41.	<lit-or-id>	→	NUMBER
42.	<lit-or-id>	→	STRING
43.	<lit-or-id>	→	FALSE
44.	<lit-or-id>	→	TRUE
45.	<lit-or-id>	→	NIL
46.	<read-format>	→	NUMBER
47.	<read-format>	→	STRING
48.	<expr>	→	PSA

Tabuľka 2: Tabuľka pravidiel LL-gramatiky