Dokumentácia projektu do predmetov IFJ a IAL

Interpret jazyka IFJ16

Tým 92, varianta b/2/II

4.12. 2012

Rozšírenia

.....

Filip Mik (vedúci) -20%	xmikfi00
Peter Žiška -20%	xziska02
Peter Pristaš -20%	xprist05
Daniel Florek -20%	xflore02
Martin Grnáč -20%	xgrnac00

1 Úvod

Táto dokumentácia popisuje implementáciu interpretu imperatívneho jazyka **IFJ16**, ako projekt do predmetov *Formální jazyky a překladače* a *Algoritmy.* **IFJ16** je podmnožinou jazyka **Java**. Projekt je možné rozdeliť na tri hlavné časti , z ktorých každej bude venovaná jedna samostatná kapitola.

- Lexikálny analyzátor (scanner),ktorý zo zdrojového súboru získava jednotlivé tokeny
- Syntaktický analyzátor(parser) ,ktorý je rozdelený na dve podčasti – syntaktický analyzátor jazykových konštrukcií a analyzátor výrazov
- Interpret ,ktorý vykonáva intepretáciu zadaného programu

2 Štruktúra projektu

Projekt je rozdelený do troch častí, ktoré sa mohli relatívne nezávislo na sebe rozvíjať. Syntaktický analyzátor je riadiacou časťou celého interpretu, riadi ostatné časti a zároveň vykonáva aj sémantickú kontrolu ,preto môžeme hovoriť tatkiež o syntaxou riadenom preklade. Preberá od lexikálneho analyzátoru tokeny ,ktoré spracováva a analyzuje. Ak je za potreby spracovať výraz ,je zavolaný na toto spracovanie precedenčný analyzátor výrazov. Akonáhle je spracovaný celý program , spúšťa sa interpret ,ktorý na základe inštrukčnej pásky vykoná finálnu interpretáciu programu.

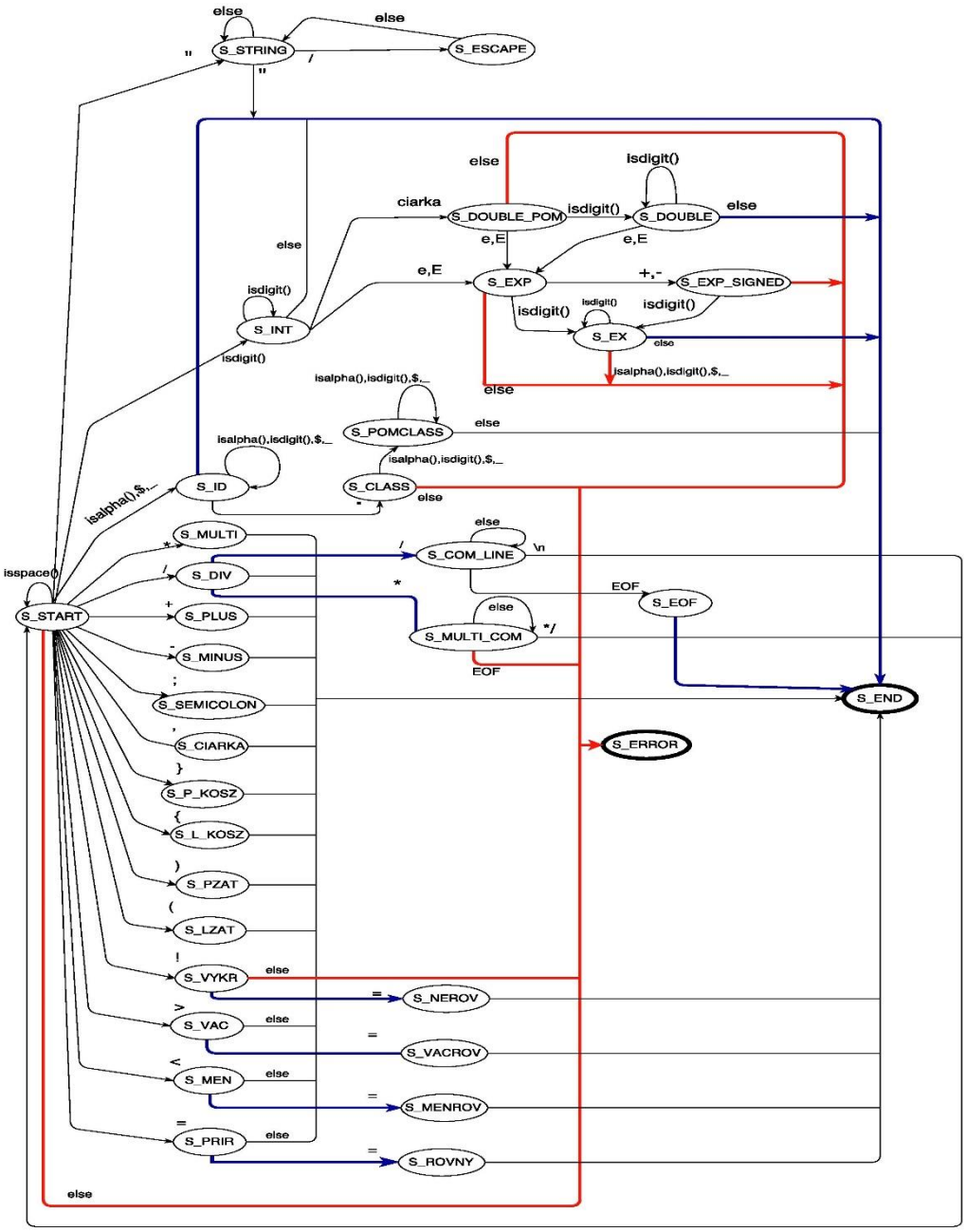
2.1 Lexikálny analyzátor

Lexikálny analyzátor (ďalej len LA) čítá zo zdrojového súboru postupnosť znakov (lexémy) a spracováva ich na základe lexikálnych pravidiel jazyka.

LA na základe požiadavky syntaktického analyzátora číta lexém a predáva mu ho. Spolu s lexémom predáva aj načítanú hodnotu(napr. reťazec, číslo) číslo riadku a pozíciu v riadku ,v ktorom sa nachádzal ,pokiaľ by tento lexém spôsobil chybu pri preklade, môžeme uživateľovi oznámiť ,kde sa daná chyba vyskytla a vraciame chybový kód.

Činnosť LA je riadená po celý priebeh syntaktickým analyzátorom ,ktorý žiada ďalší token(ten obsahuje dodatočné informácie). V prípade ,že LA bude spracovávať neznámy token ,ohlási túto chybu a do konca spracovania zdrojového súboru zostane v stave interpretovanej ako lexikálnej chyby. Princíp fungovania LA je odvodený z konečného automatu(viz obrázok uvedený nižšie), ktorého aktuálny stav je vždy vyjadrený aký bol detekovaný lexém.

Ku svojej funkčnosti LA používa jednoduché premenné ,z ktorých je najdôležitejší aktuálny stav tohto automatu a samotná štruktúra obsahujúca informácie o spracovávanom tokene ,ktorý je následne posielaný ďalej syntaktickému analyzátoru.



Graf konečného automatu LA

2.2 Syntaktický analyzátor

Syntaktický analyzátor (ďalej len SA) kontroluje program z hľadiska správnosti syntax. Správna konštrukcia syntax je definována pomocou *LL gramatiky*(syntax jazyka) zároveň taktiež vykonáva sémantické kontroly a generuje časti trojadresného kódu.

Vstupom SA je postupnosť tokenov ,ktoré zasiela LA taktiež obsahuje podčasť precedenčnej analýzy ,ktorá vyhodnocuje výrazy. Výstupom je potom informácia ,či bol program sémanticky a syntakticky správny.

Simuluje vytváranie derivačného stromu z tokenov ,ak derivačný strom nebolo možné vytvoriť ,tak program je syntakticky nesprávny. Náš SA je vytvorený na princípe rekurzívneho vzostupu ,ktorého základom je *LL tabuľka* vytvorená z *LL gramatiky*.

Základnou dátovou štruktúrou ,ktorú SA používa je tabuľka symbolov. Je používaných niekoľko tabuliek symolov – pre funkcie je definovaná s globálnou platnosťou a pre každú funkciu je ešte definovaná jej lokálna tabuľka symbolov. Vygenerované inštrukcie sa vkladajú do inštrukčného zoznamu(do poľa inštrukcií) a používané konštanty(reťazcové literály, číselné konštanty atď.) sú vkladané do tabulky konštánt, ktoré majú globálnu platnosť.

<P>	->	EOF
<P>	->	<C> <P>
<C>	->	**class id {** <AC> **}**
<AC>	->	**static** <SD> <AC>
<AC>	->	**epsilon**
<SD>	->	**void id** <SDA>
<SD>	->	<PARS> <DECL>
<PARS>	->	<TYPE> **id**
<TYPE>	->	**int | double | string**
<DECL>	->	**;**
<DECL>	->	= <E>
<DECL>	->	<SDA>
<SDA>	->	**(** <PA> **) {** <MB> **}**
<PA>	->	**epsilon**
<PA>	->	<PARS> <NP>
<NP>	->	**,** <PARS> <NP>
<NP>	->	**epsilon**
<MB>	->	<SL> <MB>
<MB>	->	**epsilon**
<SL>	->	**id** <EXPRorFC>
<EXPR>	->	<E> **;**
<EXPR>	->	<FC>
<EXPRorFC>	->	= <EXPR>;
<EXPRorFC>	->	<FC>
<SL>	->	<PARS> <VD>
<VD>	->	= <EXPR> **;**
<VD>	->	**;**
<SL>	->	<FC>
<FC>	->	**(** <FP> **) ;**
<FP>	->	**epsilon**
<FP>	->	<E> <NE>
<NE>	->	**,** <E> <NE>
<NE>	->	**epsilon**
<SL>	->	**if (**<E>**) {**<MBr>**} else {** <EB> **}**
<EB>	->	<MBr>
<EB>	->	**epsilon**
<SL>	->	**while (** <E> **) {** <MBr> **}**
<SL>	->	**return** <RV> **;**
<RV>	->	<E>
<RV>	->	**epsilon**
<SL>	->	**print(** <E> <PrA> **) ;**
<PrA>	->	**+** <E> <PrA>
<PrA>	->	**epsilon**
<MBr>	->	**id** <EXPRorFC> <MBr>
<MBr>	->	<FC>
<MBr>	->	**if (**<E>**) {**<MBr>**} else {** <EB> **}**
<MBr>	->	**while (** <E> **) {** <MBr> **}**
<MBr>	->	**return** <RV> **;**
<MBr>	->	**print(** <E> <PrA> **) ;**

LL gramatika

Kedykoľvek syntaktický analyzátor narazí na výraz zavolá si precedenčnú analýzu(ďalej len PA) ,ktorá mu následne potom spracovaný výraz vráti.

Ako vstup je aktuálna tabuľka symbolov obsahujúca informácie o premenných a glob. tabuľka symbolov ,v ktorej sú uložené informácie o funkciách, ďalším parametrom pre

vstup je odkaz na zásobník ,ktorý vytvára a spracováva SA. Taktiež je za potreby tabuľka konštánt a list inštrukcií. Výstupom PA sú vygenerované inštrukcie a chybový kód ,ak prebehla operácia úspešne bez chýb ,tak SA očakáva výsledok výrazu v unikátnej premennej uloženej v tabulke symbolov ,ktorá je vytvorená len za týmto účelom.

Dôležitou časťou PA je precedenčná tabuľka(viz. obrázok uvedený nižšie) ,v ktorá obsahuje definície syntaktických pravidiel pre všetky výrazy. Algoritmus ako celok je tvorený cyklom, v ktorom sú postupne vyhodnotené a redukované pravidlá získané z precedenčnej tabuľky . Táto štruktúra implementovaná formou zásobníka.

	+	-	*	/	(	)	id	idf	<	>	<=	>=	'=='	!=	,	$
'+'	>	>	<	<	<	>	<	<	>	>	>	>	>	>	>	>
'-'	>	>	<	<	<	>	<	<	>	>	>	>	>	>	>	>
*	>	>	>	>	<	>	<	<	>	>	>	>	>	>	>	>
/	>	>	>	>	<	>	<	<	>	>	>	>	>	>	>	>
(	<	<	<	<	<	=	<	<	<	<	<	<	<	<	>	E
)	>	>	>	>	E	>	E	E	>	>	>	>	>	>	=	>
id	>	>	>	>	E	>	E	E	>	>	>	>	>	>	>	>
idf	E	E	E	E	R	E	E	E	E	E	E	E	E	E	E	E
<	<	<	<	<	<	>	<	<	E	E	E	E	E	E	>	>
>	<	<	<	<	<	>	<	<	E	E	E	E	E	E	>	>
<=	<	<	<	<	<	>	<	<	E	E	E	E	E	E	>	>
>=	<	<	<	<	<	>	<	<	E	E	E	E	E	E	>	>
'=='	<	<	<	<	<	>	<	<	E	E	E	E	E	E	>	>
!=	<	<	<	<	<	>	<	<	E	E	E	E	E	E	>	>
,	<	<	<	<	<	=	<	<	<	<	<	<	<	<	=	E
$	<	<	<	<	<	E	<	<	<	<	<	<	<	<	E	E

Precedenčná tabuľka

2.3 Interpret

Interpret vykonáva typovú kontrolu ,tzv. či je určitá operácia nad danými datovými typmi definovaná. K svojej činnosti potrebuje inštrukčnú pásku a tabuľku konštánt ,v ktorej sa uchovávajú konštanty ,ktoré boli načítané zo zdrojového súboru zadané užívateľom.

Na základe aktuálnej inštrukcie vykonáva tomu odpovedajúce akcie pri prechádzaní inštrukčnou páskou. Ak by narazil na definíciu funkcie, preskočí ju. Keď narazí na volanie funkcie vytvára sa nový zásobník ,nahrá sa naň návratová adresa(t.j. index do inštrukčnej pásky ,kde sa po jej vykonaní bude pokračovať vo funkcii), jej parametre, adresa premennej, kam sa má zapísať jej návratová hodnota a vykoná sa skok na

definíciu funkcie. Používané dátové štruktúry pre pole inštrukcií (inštrukčná páska), tabuľka konštánt (implementovaná pomocou zásobníka) , z ktorej sa načítavajú konštanty na zásobník hodnôt ,kde sa prebiehajú operácie(aritmeticko-logické a pod.).

2.4 Boyer Mooreov algoritmus

Boyer Mooreov algoritmus (ďalej len BMA) vychádza z úvahy ,že pri porovnávani vzorkov s reťazcom je možné preskakovať znaky ,ktoré sa nemôžu rovnať. Výhodou pre to je ,že čím väčší je vyhľadávaný vzor o to viac môže tento algoritmus preskočiť v prehľadávanom reťazci. BMA používa prvú heurestiku ,ktorej princípom je ,ak nájdeme nezhodu znakov ,môžeme sa posunúť začiatkom vzorku o N miest ďalej. Porovnávanie vzorku začína sprava. Výsledkom je nájdenie zadaného vzorku. Správanie BMA závisí na kardinalite abecedy a na opakovaní podreťazcov vo vzorku.

2.5 Tabuľka s rozptýlenými prvkami

Tabuľka symbolov je tvorená pomocou hashovacej tabuľky, ktorá má v najlepšom prípade zložitosť O(1) a v najhoršom prípade lineárnu zložitosť. Hashovacia tabuľka je štruktúra obsahujúca pole ukazateľov N prvkov a integer s číslom N, ktoré sa požíva pri hashovacej funkcií.

Tieto ukazatele ukazujú na prvky ,ktoré tvoria jednosmerne viazaný zoznam avšak sú rozptýlené vďaka hashovacej funkcií a každý prvok obsahuje informácie o názve a typu ,premennej či funkcie, názvu prislúchajúcej triedy ukazateľ na ďalšiu položku atď. Vyhľadávanie prebieha získaním indexu z hashovacej funkcie pre prvok ,ktorý hľadáme na základe jeho mena. Vďaka tejto funkcii je vyhľadávanie prvku veľmi rýchle.

2.6 Heap sort – radenie hromadou

Heapsort je riadiaca metóda s linearitmickou zložitosťou ,pretože sift vie nájsť extrém medzi N prvkami s log zložitosťou. Hromada je štruktúra stromového typu ,pre ktorej všetky uzly platí ,že medzi otcovským uzlom a všetkými jeho synovskými je rovnaká relácia usporiadania. Využíva operáciu sift ,ktorej princíp spočíva v tom ,že prvok z koreňa postupnými výmenami prepadne na svoje miesto a do koreňa sa dostane prvok ,ktorý spĺňa pravidlá hromady. Táto operácia má v najhoršom prípade zložitosť $\log_2 N$.

Koreň hromady obsahuje vždy maximálnu hodnotu. Ak pole má veľkosť N prvkov ,tak najnižší a najpravší otcovský uzol odpovedajúci hromade má index [N div 2]. Ak je dvojnásobok indexu uzla väčší než počet prvkov pola N, tak odpovedajúci uzol je terminálny, avšak ak je dvojnásobok uzlo rovný počtu prvkov N ,odpovedajúci uzol má len ľavého syna alebo ak je dvojnásobok indexu uzla menší než počet prvkov N ,odpovedajúci uzol má oboch synov.

3 Práca v tíme a použité prostriedky

Na začiatku sme sa s vedúcim tímu dohodli a rozdelili si projekt na jednotlivé časti medzi sebou k online komunikácii sme využívali Messenger a aktívne sa stretávali a zisťovali v akej sme fáze poprípadne ,kde sa vyskytuje problém a pod. Ako úložisko dát sme využívali pre spoločný vývoj systém Git ,pričom ako hosting sme použili pre náš repozitár službu BitBucket. Pre dôkladnosť a porozumiteľnosť sme dostatočne komentovali kód.

4 Záver

Projekt bol časovo náročný a pracovali sme na ňom cez viac ako 2 mesiace. Naučil nás schopnosti tímovej spolupráce a komunikácie, plánovania alebo používania verzovacieho systému. Už od začiatku sme si naštudovali vopred problematiku ,pretože daná problematika bola preberaná s miernym oneskorením voči postupu pri vývoji interpreta.

5 Literatúra

- Prednášky,skriptá a podklady k predmetu IFJ a IAL