[ZADEJTE NÁZEV SPOLEČNOSTI.]

Dokumentace k vývoji překladače

Skupinový projekt do předmětů IFJ a IAL

Vypracoval Boleslav Šídlo

9.12.2012

Stručný přehled informací o týmu vývojářů, základní popis fungování celého programu, objevených problémech a způsobech jejich řešení, zhodnocení celého projektu.

1. Informace o týmu

Vedoucí týmu:	**login**	**přiděleno bodů**
Tomáš Rajca	xrajca00	20 %
Další členové:		
David Buchta	xbucht18	20 %
Oto Pokorný	xpokor57	20 %
Matěj Radvanský	xradva00	20 %
Boleslav Šídlo	xsidlo02	20 %

Varianta zadání:	Tým 11, varianta b/2/I
Datum zadání:	17. 9. 2012
Datum odevzdání:	9. 12. 2012

2. Základní informace o projektu

Cílem tohoto projektu bylo napsat překladač z programovacího jazyka IFJ12, jenž je zjednodušenou verzí programovacího jazyka Falcon, do jazyka C. Při tomto projektu jsme se měli naučit skupinové práci a prokázat porozumění látce probírané v předmětech IFJ a IAL a následně tyto poznatky využít v praxi při vývoji vlastního překladače.

3. Organizace práce a rozdělení úkolů

Pro vzájemnou domluvu jsme kromě osobních setkání používali různé internetové komunikační nástroje, přičemž nejdůležitějším z nich server pro sdílení zdrojových kódů github.com, který nám značně usnadnil výměnu informací.

4. Průběh vývojeprogramu

Stejně jako u kteréhokoliv jiného softwaru bylo i zde nutné začít analýzou celého problému a jasnou definicí výchozí situace. Jelikož byly veškeré požadavky a vstupní podmínky přesně vymezeny v zadání projektu, nebylo potřeba tuto část nějak zpracovávat, ta prostě byla daná.

Při následné dekompozice celého problému jsme se řídili postupem uvedeným na přednáškách IFJ. Tedy vycházeli jsme z toho, že překladač se skládá z lexikálního analyzátoru, syntaktického analyzátoru, sémantického analyzátoru, generátoru vnitřního kódu, optimalizátoru a generátoru vnitřního kódu.

Rozhodli jsme se, že na začátku budeme paralelně pracovat po skupinách na lexikálním syntaktickém analyzátoru. Poté co tyto dva základní bloky celého projektu téměř hotové, jsme přistoupili k tvorbě dalších částí, opět paralelně-sémantického analyzátoru a generátorů vnitřního a cílového kódu. Tyto nové komponenty byly vyvíjeny, zatímco probíhaly dokončovací práce dvou prvotních.

V průběhu celého projektu, jsme několikrát zjistili, že některé naše prvotní plány a programové konstrukce nevedou, k takovému řešení problému, jaký bychom si představovali, tudíž jsme se museli ve vývoji vracet a zpětně předělávat některé části. Prakticky jsme tedy aplikovali vodopádový model vývoje softwaru. Konkrétní problémy, na které jsme narazili jsou popsány dále.

Co se týče přiřazení lidí k jednotlivým sekcím, má se to následovně:

Lexikální analýza	David Buchta, Boleslav Šídlo
Syntaktická a sémantická analýza	Oto Pokorný, Matěj Radvanský, Tomáš Rajca
Generátor vnitřního kódu	Tomáš Rajca,Matěj Radvanský
Generátor cílového kódu	David Buchta
Testování	Oto Pokorný, Boleslav Šídlo

5. Popis našeho řešení interpretu

Základem je lexikální analyzátor, který čte vstupní soubor a posílá syntaktickému analyzátoru posloupnost tokenů. Parser zároveň provádí i sémantickou analýzu. Pakliže je program lexikálně, syntakticky i sémanticky správně, parser vygeneruje pole abstraktních syntaktických stromů. Přičemž platí, že každá položka tohoto seznamu reprezentuje jeden řádek kódu vstupního programu. Tento seznam je poslán generátoru vnitřního kódu, který jej převede na posloupnost tříadresných instrukcí. Tento vnitřní kód je pak dále poslán generátoru cílového kódu, který interpretaci dokoná a předá na výstup přeložený program.

6.

7. Realizace a problémy jednotlivých částí překladače

Lexikální analyzátor

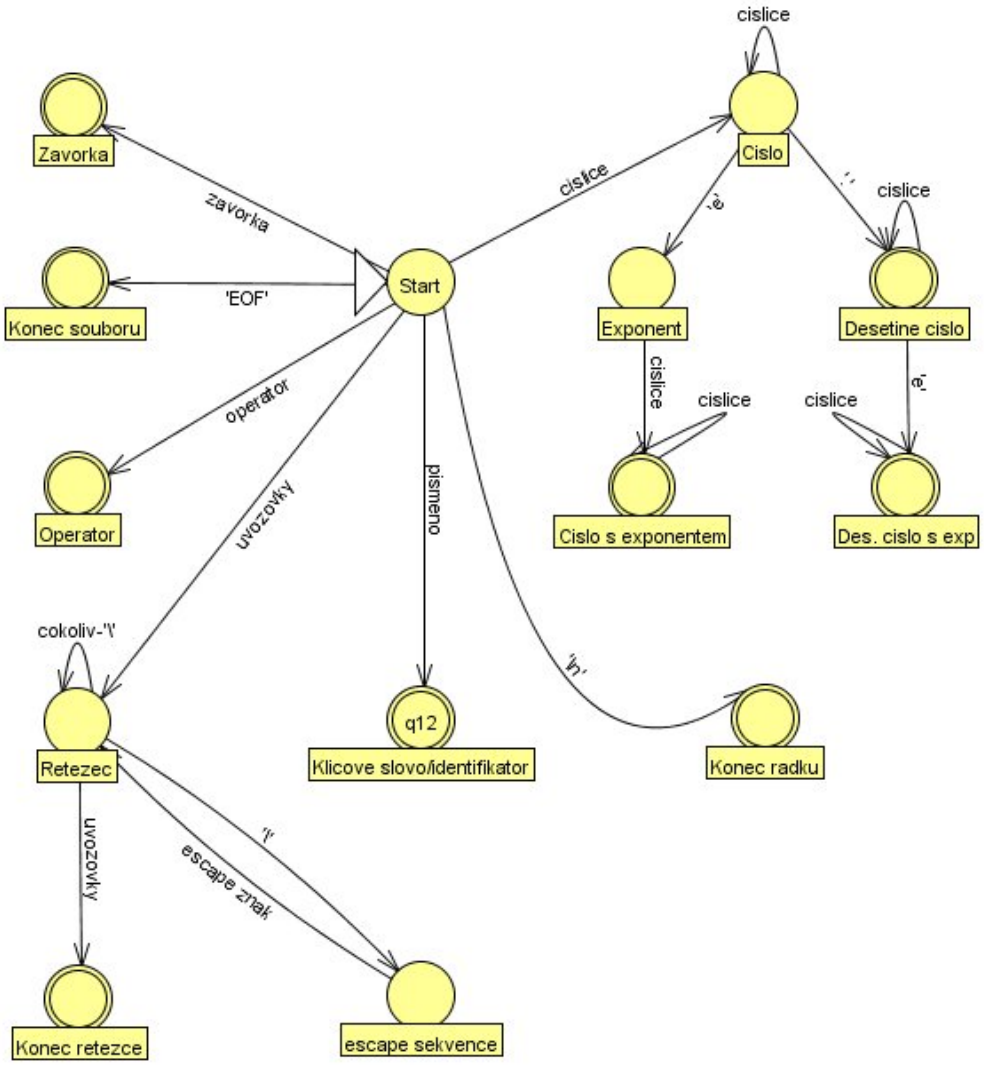
Lexikálního analyzátor měl tu výhodu, že bylo naprosto jasné, co bude na vstupu- posloupnost znaků vstupní abecedy. Co už bylo méně zřejmé, jak co nejlépe a nejefektivněji stanovit výstup, tedy posloupnost tokenů. Narazili jsme na několik problémů.

Nejdříve jak samotnou strukturu T_Token_struct definovat, abychom pokryli všechny možnosti, ale zároveň zbytečně neplýtvali pamětí. Bylo rozhodnuto, že struktura se bude předávat odkazem a dále, že pokud se bude jednat o token, jemuž kromě typu přísluší i hodnota (číselný, nebo řetězcový literál) bude se tato hodnota zapisovat na pozici určenou void*value ukazatelem. Ten se pochopitelně musí příslušně přetypovat.

Dalším problémem byla situace, kdy scanner poslal token, ovšem při jeho zpracování načetl znak navíc, který už do daného tokenu nepatřil, tedy v následujícím tokenu mohl chybět. V tomto případě bylo řešení jednoduché. Prostě jsme do struktury token přidali proměnou charprevious společně s indikátorem intstate, který informoval, jestli byl načten znak navíc.

Jinak co se týče samotného typu tokenu, tak je uložena v proměnné int type, jež nabývá jedné z hodnot tabulky terminálů

Struktura konečného automatu-lexikální analyzátor



Z důvodu přehlednosti, je schéma automatu značně zjednodušeno. Nejsou zde detailně popsány všechny možné kombinace, ani zobrazeny chybové stavy. Jedná se pouze o hrubý náčrt. Skutečný analyzátor je pochopitelně mnohem složitější.

K vytvoření obrázku byl použit program JFLAP verze 7.0, který je k dispozici na:

http://www.jflap.org/jflaptmp/

Syntaktický analyzátor

Syntaktický analyzátor(SA) využívá lexikální analyzátor k tomu, aby mohl postupně načítat strukturu programu a vyhodnocovat její syntaxi, dostává od něj na požádání tokeny pomocí funkce *getNextToken()*.

Abychom vůbec mohli začít s implementací, bylo nutné přepsat slovní zadání do formy, se kterou se lépe pracuje – pravidla gramatiky. Z těchto pravidel jsme pak sestrojili LL tabulku, která posloužila jako základ pro vytvoření rekurzivního sestupu. Samotný návrh pravidel a LL tabulky ale probíhal v několika fázích, kdy bylo nutné je předělat.

Rekurzivní sestup je postup, jak má SA postupovat pro různé kombinace tokenů, tedy jak se má zachovat pro konkrétní situace, kam má postoupit dále. Je vytvořen podle LL tabulky, platí zde tedy determinismus a proto je jasné, kam se má kdy sestoupit. Pokud přijde token, který nemá podle definovaných pravidel gramatiky přijít, jedná se o syntaktickou chybu - SA vždy ví, jak se má zachovat. Každý neterminál je realizován svou vlastní funkcí, kde je popsán postup, co dělat při terminálu, který je na vstupu.

Tento přístup se ale nedá reálně použít u vyhodnocování výrazů, protože priorita operátorů dokáže pořadím načtených tokenů dost zamíchat, proto je potřeba zvolit jiný přístup-precedenční syntaktickou analýzu. Vždy, když je potřeba vyhodnotit výraz, volá se funkce *expr()*. K tomu využíváme zásobníky, kdy si načtený výraz postupně převádíme z infixové notace na postfixovou notaci. Zároveň je brána v úvahu i priorita operátorů. Postfixový výraz už pak lze snadno vyhodnotit a pokračovat dále.

Vzhledem k tomu, že sémantický analyzátor provádí kontrolu v rámci SA, výstupem SA je abstraktní syntaktický strom ve formě kombinace lineárního seznamu a binárního stromu. S tímto stromem dále pracuje generátor vnitřního kódu.

Dolaďování podmínek a detailů uvnitř SA nám zabralo asi nejvíce času i díky tomu, že vyhodnocování výrazů nebylo zprvu tak jasné jak se zdá teď a skončilo spoustou neúspěšných pokusů.

LL gramatika-syntaktický analyzátor

	Prog	St_list	Stat	Params	Params_n	Expr	Term	Term_n	Opr	Func	Assign
eol	1	2	10			15		18			38
function	1	2	4								
id	1	2	5	11		15	19				38
(						15		17			
)				12	14						
end		3									
if	1	2	6								
else		3									
while	1	2	7								
return	1	2	8								
,					13						
[						15		17			
:											
]											
+						15		16	24		38
-						15		16	25		38
/						15		16	27		38
*						15		16	26		38
**						15		16	28		38
sort										35	9
find										34	9
len										33	9
typeof										32	9
print										31	9
numeric										30	9
input										29	9
false						15	23				38
literal						15	22				38
“”						15	21				38
nil						15	20				38
$		3									

Sémantický analyzátor

Sémantický analyzátor je v tomto případě zahrnut uvnitř syntaktického analyzátoru. Ke kontrole sémantiky využívá tabulku symbolů, kam vždy při definici nového symbolu(tedy názvu proměnné nebo funkce) uloží záznam, že tento symbol existuje pomocí funkce *Tree_insert_new()*. A při každém výskytu daného symbolu(volání funkce nebo použití proměnné ve výrazu) pomocí funkce *Tree_Find_node* zkontroluje tabulku symbolů a pokud nenajde tento symbol, jedná se o sémantickou chybu.

Tabulka symbolů je implementována ve formě binárního stromu, a je rozdělena na dvě části – globální *gts* a lokální *lts*. Do globální části se ukládají záznamy o funkcích, zatímco do lokální části záznamy o jejich proměnných(ale ne hodnoty).

Dále vyhodnocuje, zda uvnitř výrazu jsou nějaké nekoretní operace nad literály – pokud se ve výrazu vyskytuje proměnná, nelze to vyhodnotit, ale v případě, že jsou zde jen literály, dá se hned na základě pravidel pro každou operaci zjistit, jestli se provádí korektní operace.

Generátor vnitřního kódu

Jeho úkolem je zajistit, aby existovala sekvence příkazů pro generátor cílového kódu. Zvolili jsme přístup pomocí tří-adresného kódu. Je nutné vytvářet pomocné označení každé operace, aby bylo možné provádět skoky.

Prochází abstraktní syntaktický strom pomocí funkce *inorder()* a generuje instrukce. Konkrétně existuje pole typu *TAC*, kam se do každého prvku pole ukládají operandy a operátor, adresa kam se má uložit je generována unikátně. Toto pole slouží jako páska, kterou projíždí generátor cílového kódu.

Každá instrukce je tvořena maximálně jedním operátorem a dvěma operandy, pracujeme tedy s jednoduchou instrukční sadou pro operace(všechny operace jsou binární).

Generátor cílového kódu

Projíždí pole *TAC,* kde je uložen seznam instrukcí a podle něj generuje finální zpracování programu. Dochází zde k vyhodnocování vestavěných funkcí a vyčíslování proměnných a funkcí. Vyhodnocují se chyby interpretu, tedy chyby které nebylo možné dříve zjistit(než byly vyčísleny promenné). V podstatě dochází k porovnávání typů u každé instrukce.

Vzhledem k tomu, že jsme se k řešení této části dostali až jako k úplně poslední, museli jsme řešit problémy s časem a také došlo k zanesení množství chyb.

Testování

K závěru celé práce bylo třeba začít s náležitým testováním celého programu. K tomuto účelu jsme si vytvořili zhruba 200 testovacích zdrojových kódu pro různé typy chyb. Abychom nemuseli všechny ručně spouštět jeden program po druhém a následně zjišťovat zdali se náš překladač chová korektně, vytvořili jsme si v bashi skript, který většinu rutinní kontroly obstaral za nás. Testy ale bylo nutné navrhovat ručně, proto jsme ani zdaleka nebyli schopni pokrýt testování dostatečně(také kvůli špatnému rozvrhnutí času).

8. Implementace vestavěných funkcí

Řadící algoritmus

Pro řešení řazení znaku dle jejich ordinální hodnoty, jsme podle zadání použili algoritmus HEAP SORT.

Tento algoritmus funguje na principu řazení v binárním stromu, který je realizován za pomocí pole, kde kořen má index 0, a pro otcovské a synovské uzly platí vztah, že jestliže je otcovská uzel na pozici n levý syn se pak nachází na pozici 2n+1 a pravý syn je na pozici 2n + Výhodou tohoto to algoritmu je velmi dobrá časová složitost.

Pro vlastní realizaci algoritmu jsme použili tři funkce. Hlavní funkce, která je volaná generátorem cílového kód a provádí vlastní řazení, nese název bi_sort. Tato funkce je void a jako parametr dostává na vstup řetězec určen k seřazení. Prochází obě strany binárního stromu a postupně prosívá nejvyšší prvek. K této funkci je pomocná funkce bi_repair_top, která zajistí porovnáním znaku s oběma syny, že nesprávně umístěny vrchol haldy propadne na správné místo.

Vyhledávácí algoritmus

Vyhledávání podřetězce v řetězci zajišťuje Boyer-Mooreův algoritmus.

Algoritmus hledá výskyt vzorku v daném řetězci od posledního znaku. Velkou výhodou zde je že se nemusí porovnávat všechny znaky z řetězce, ale skoro vždy se nějaké přeskočí. Prvním porovnávaným znakem v řetězci je znak, který má stejnou pozici od začátku řetězce jako je dílka podřetězce. Pokud se srovnávané znaky nerovnají, zavolá si algoritmus dvě heuristiky, které mu doporučí vzdálenost kterou je možnost přeskočit a algoritmus si vybere tu větší.

Hledací funkce bi_find, která se nachází v souboru ial.h má dva vstupní parametry a to jsou v tomto pořadí: řetězec, ve kterém se vyhledává a hledaný vzorek. Hlavní funkce má k dispozici celkem čtyři pomocné funkce. Funkce bi_find_delta1 , a bi_find_delta nám sestaví delta tabulky. Funkce bi_find_prefix nám vrátí, jestli je právě porovnávány znak prefixem našeho hledaného řetězce a funkce bi_find_suffix_length, která vrací nejdelší příponu končící na právě porovnávaném znaku.

Hlavní funkce vrací buď pozici znaku v řetězci (číslovaném od nuly), nebo -1 což znamená, že se hledaný vzorek nenachází v řetězci. Ošetření parametrů a chyby pokud funkce nedostane dva řetězce, zajišťuje přímo soubor cilovy_kod.c.

9. Závěr

Celkově projekt hodnotíme jako úspěšný. Podařilo se nám základním způsobem porozumět problematice překladačů, rozebrat a následně vyřešit problémy, na které jsme narazili a najít efektivní způsob kolektivní práce. Což následně vedlo k dokončení funkčního kompilátoru.

Nedostatky a možnost případného rozvoje, bychom viděli v efektivitě celého návrhu a implementace. Toto lze zajisté přičíst na vrub naší naprosté nezkušenosti na začátku tohoto projektu, malému množství přiděleného času a nutnosti plnit jiné povinnosti v rámci studia.

10. Použité zdroje

MEDUNA,A,LUKÁŠ,R.*Formální jazyky a překladače:Studijníopora.*Verze 1.2006+revize 2012. Brno

HONZÍK,J,M.*Algoritmy:Studijníopora.*Verze 12-L. Brno