VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta informačních technologií

Dokumentace k projektu do předmětu IFJ a IAL

Interpret jazyka IFJ16

Tým 107, varianta b/2/I

7. prosince 2016

Autoři:

Petr Flajšingr (vedoucí projektu), xflajs00 – 20%
Igor Frank, xfrank12 – 20%
Dominik Dvořák, xdvora1t – 20%
Pavel Míča, xmicap02 – 20%
Martin Pospěch, xpospe03 – 20%

Obsah

1 Úvod

V dokumentaci budou popsány části interpretu imperativního jazyka IFJ16, implementace, metody a vývoj jednotlivých částí. Jelikož se jedná o projekt, který je řešen v týmu, je zde popsána i týmová spolupráce a celkový přínos práce na projektu.

Z pohledu předmětu IAL jsme implementovali a v dokumentaci detailněji popsali algoritmy k vyhledávání podřetězce v řetězci, algoritmus řazení a abstraktní datovou strukturu tabulky symbolů.

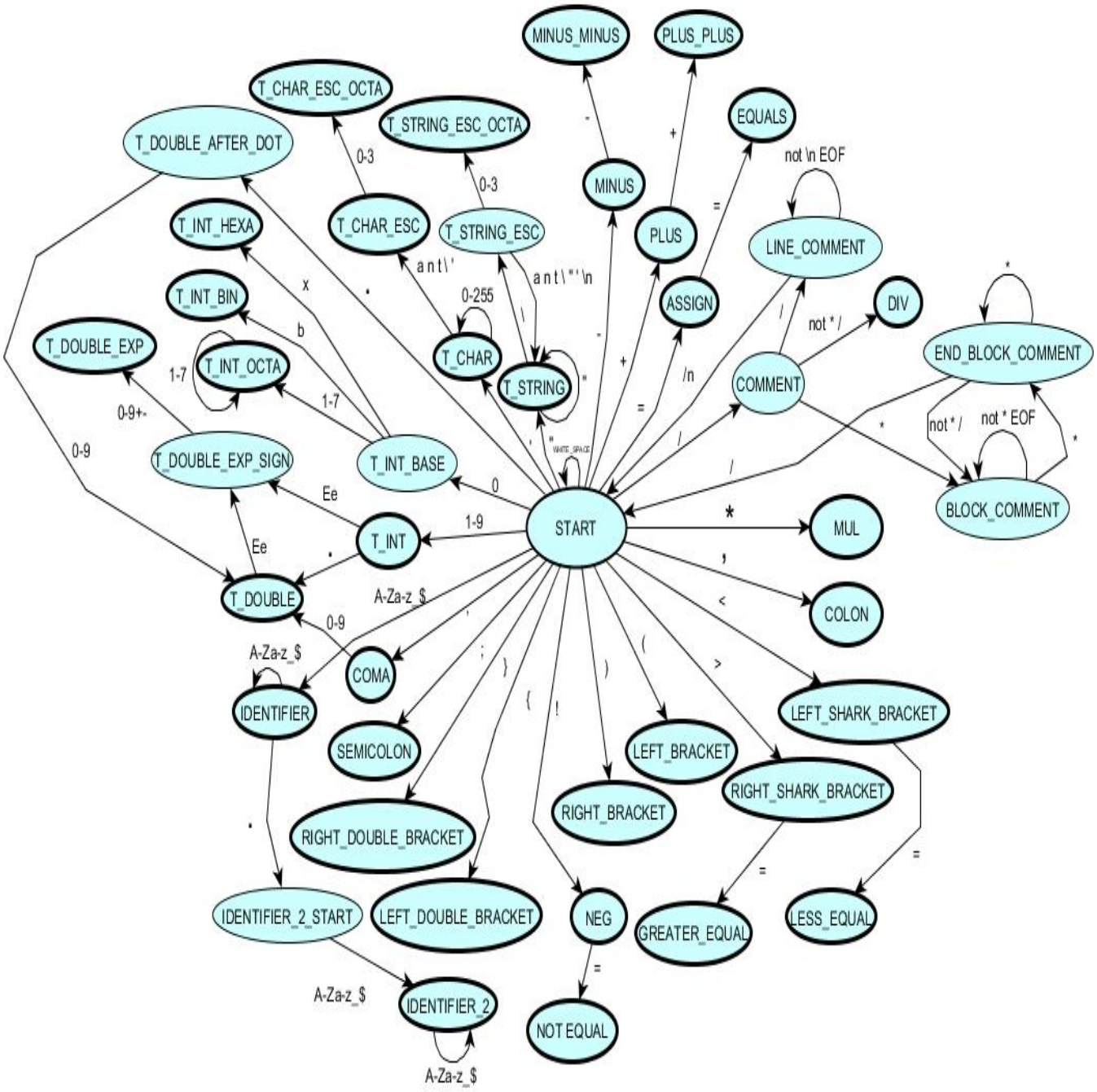
Konečný automat a tabulka pro precedenční analýzu jsou přidány jako přílohy, jelikož se jedná o zásadní prvky interpretu a pomůžou objasnit jejich samotnou implementaci.

2 Struktura projektu

Projekt jsme si rozdělili na tři části. Lexikální analyzátor, který má jako vstup zdrojový kód, rozdělí ho na tokeny, což jsou logicky oddělené lexikální jednotky a ty jsou vstupem pro další část, a to syntaktickou analýzu. Ta je základem celého interpretu, systematicky volá ostatní části. A nakonec z instrukční pásky interpretuje celý program.

2.1 Lexikální analyzátor

Scanner neboli lexikální analyzátor načte zdrojový kód ze souboru a jednotlivé znaky zanalyzuje a danou posloupnost znaků rozdělí na lexémy, ty jsou reprezentovány ve formě tokenu, který je jim ve výsledku přidělen. Ten je uložený do abstraktního datového typu. Dalším úkolem lexikálního analyzátoru je odstraňování bílých znaků a komentářů. Činnost analyzátoru je znázorněna na schématu konečného automatu. Syntaktický analyzátor se stará o tuto činnost a řídí toky tokenů, tedy žádá o další tokeny. V případě chyby, což znamená, že analyzátor narazil na neznámý lexém, se vrací lexikální chyba.



Obrázek 1: Konečný automat lexikálního analyzátoru

2.2 Syntaktický analyzátor

Syntaktický analyzátor tvoří jádro interpretu. Tento analyzátor, nazývaný také parser se stará o kontrolu správné syntaxe programu. Vstupem je otevřený soubor se zdrojovým kódem. Parser dále žádá od lexikálního analyzátoru řetězec tokenů a kontroluje, zda vše proběhlo bez syntaktické nebo sémantické chyby. Simuluje tvoru derivačního stromu, pokud se mu strom nepodaří sestavit, vzniká syntaktická chyba. Globální tabulka symbolů pro funkce je implementovaná na základě abstraktní datové struktury. K vyhodnocení výrazů používáme precedenční tabulku, ve které zjistíme prioritu matematických operací pro precedenční analýzu. Ten využívá vlastní zásobník pro ukládání údajů získaných od lexikální analýzy. Dále aplikuje pravidla, která jsou určena níže zobrazenou tabulkou. Pomocí těchto pravidel je vytvářen abstraktní syntaktický strom.

	*, /	+, -	<, >, <=, >=	!=, ==	(	)	i	$
*, /	>	>	>	>	<	>	<	>
+, -	<	>	>	>	<	>	<	>
<, >, <=, >=	<	<	>	>	<	>	<	>
!=, ==	<	<	<	>	<	>	<	>
(	<	<	<	<	<	=	<	Error
)	>	>	>	>	Error	>	Error	>
i	>	>	>	>	Error	>	Error	>
$	<	<	<	<	<	Error	<	Empty

Obrázek 2: Precedenční tabulka

2.3 Sémantický analyzátor

Jedná se o kontrolu dalších pravidel, jelikož syntaxe již byla zkontrolována, přichází na řadu kontrola sémantiky. Zde probíhá ověření, zda byly použity inicializované proměnné, správná definice a typ návratové hodnoty, volání funkce se správnými argumenty, dále také kontrola definovaných proměnných a správné použití jejich datových typů, zda neproběhla redefinice funkce, operandy správného datového typu. To vše je kontrolováno pomocí tabulky symbolů, se kterou analyzátor spolupracuje a kde se vyhledávají dané informace k ověření již zmíněných možných pochybení.

V naší implementaci však funkci sémantického analyzátoru plní interpret spolu s funkcemi, které má vykonávat (viz 2.4 Interpret). Pro kontrolu operací s proměnnými a funkcemi jsou použity v zastoupení interpretu různé instrukce.

2.4 Interpret

Pro náš interpret jsme vytvořili instrukční sadu, kterou používáme k vnitřní reprezentaci programu, což je tří adresný kód. Každá instrukce má však zásobník, který slouží k uchování parametrů u volání funkcí. Instrukce jsou uložené na instrukční pásce, která je tvořena na základě

lineárního seznamu. Tento seznam obsahuje kromě ukazatele na první a poslední prvek také ukazatel na aktivní prvek, aby při návratu interpretu zpět na instrukční pásku bylo zjištěno, která instrukce je na řadě.

Pro začátek je vytvořená funkce třídy s názvem IFJ16.init, kde se nachází třídní proměnné, které se inicializují výrazem.

Interpret se ukončí pomocí instrukce `CG_RET`, což značí ve funkci „return“ a ze zásobníku zjistí, kde bude pokračovat, nebo skončí pomocí NULL.

3 Implementované algoritmy

V rámci předmětu IAL jsme v rámci individuálního zadání implementovali následující algoritmy.

3.1 Boyer-Mooreův algoritmus

Hledání podřetězce v řetězci je v naší implementaci řešeno Boyer-Mooreovým algoritmem dle jeho první heuristiky pro postup v prohledávaném řetězci. Hlavním předností algoritmu je možnost přeskočit znaky, které vyhodnotí jako neodpovídající, a tím celé vyhledávání značně urychlit.

Daný podřetězec se v řetězci hledá zleva doprava. Znaky se však porovnávají z pravé strany hledaného výrazu. Pokud se vybraný znak řetězce neshoduje s posledním znakem vzorku, dále se zjistí, zda v něm tento vybraný znak není obsažen. V případě, že tomu tak je, posune se tento znak na místo shodujícího znaku řetězce a začnou se znovu porovnávat znaky z pravé strany. V případě shody všech znaků vzorku je vyhledávání úspěšné, v opačném případě se posune o určitý počet míst, který se rovná počtu znaků hledaného podřetězce a proces se opakuje do konce řetězce.

3.2 Heap sort

Pro řazení požíváme algoritmus, založený na porovnávání prvků, který řadí pomocí stromové struktury tzv. hromady. To umožňuje nalezení maxima v konstantním čase. Reprezentace hromady binárním stromem má následující vlastnosti, a to že všechny úrovně, kromě poslední, jsou zaplněné maximálně dvěma uzly a v kořenu hromady se nachází dané maximum. Dále oba synové jsou vždy menší nebo rovny otci.

V naší implementaci však řadíme od nejmenšího prvku, což znamená že v kořenu hromady bude vždy nejmenší hodnota, která se následně z hromady odebere.

Z prvků v poli k seřazení se sestaví hromada, avšak ta je rozbitá a v kořenu je prvek, který nedodržuje pravidla hromady, a proto je nutné ji správně seřadit tak, aby v hlavním kořenu byl nejmenší prvek. Po odebraném prvku z kořenu se vloží na tento index prvek z nejnižší úrovně a levé strany. Následně se opět provede znovu seřazení a nejmenší prvek se z hromady odebere. Tato operace se opakuje, dokud není hromada prázdná. Následně je daná posloupnost seřazená.

Časová náročnost tohoto algoritmu pro seřazení posloupnosti je O = (n . log n).

3.3 Tabulka symbolů

Tabulku symbolů jsme implementovali pomocí abstraktní datové struktury vyhledávacího binárního stromu. Implementace je však rozdělena do několika stromových struktur a ve výsledku jsou spojené v jeden celek, který tvoří právě tabulku symbolů.

První tabulka obsahuje název třídy, který je taktéž klíčem daného uzlu a dle tohoto pravidla jsou pojmenovány i klíče následujících tabulek. Tou další je tabulka funkcí. Zde se uloží název, návratový typ funkce a argumenty, uložené v zásobníkové struktuře pro jejich snadnější manipulaci. V této tabulce se nachází ukazatel na tabulku lokálních proměnných dané funkce. Do uzlu ukládáme opět název proměnné a její obsah. Stejná tabulka se nachází i pod tabulkou třídy, kam se ukládají třídní proměnné.

Po celý čas běhu programu se pracuje s jedním existujícím stromem, do kterého se postupně přidávají jednotlivé uzly a dále jsou využívány k různým operacím pro správný chod překladače. Dané uzly vyhledáváme pomocí jména. Pokud se však jedná o tabulku funkcí nebo proměnných funkce, musíme v tomto případě znát jméno třídy, které tato funkce náleží.

Tabulka je velmi důležitou součástí interpretu, protože je základem mnoha jeho operací.

4 Vývoj a týmový managment

Práce v týmu pro nás byla po loňském neúspěšném projektu výzvou. Každý se ji však tento rok zhostil výrazně lépe. Na projektu se začalo pracovat téměř hned po zveřejnění zadání a základ projektu byl vytvořen takřka za pár dní. Měli jsme tak časovou rezervu pro časový nápor dalších projektů a půlsemestrálních zkoušek.

Pro řešení otázek jednotlivých částí projektu jsme se, především ze začátku, scházeli ve škole, kde jsme využili prostředí pro vzájemnou diskusi a vyřešení jednotlivých problémů, které se v průběhu vývoje projektu naskytli. Kromě osobních schůzek jsme komunikovali přes sociální síť Facebook, kde jsme využili možnost skupinové konverzace. Zdrojové kódy jsme měli uložené na verzovacím systému Git přes službu BitBucket. Ta poskytuje vytvoření soukromého projektu, takže přístup k těmto kódům měli všichni členové týmu odkudkoliv, kde byl internet. Zároveň je však zabráněno nechtěnému šíření.

4.1 Rozdělení práce v týmu

- Lexikální analyzátor – Dominik Dvořák
- Syntaktický analyzátor – Petr Flajšingr, Martin Pospěch
- Interpret – Igor Frank
- Implementované algoritmy do IAL – Petr Flajšingr
- Tabulka symbolů – Pavel Míča
- Dokumentace – Pavel Míča

5 Závěr

Projekt nám pomohl pochopit jednotlivé části překladače, jejich funkce a zároveň tyto části poskládat do jednoho celku, který tvoří právě implementovaný překladač. Pro každého z nás byl tento projekt velkou zkušeností. Ať už to jsou získané vědomosti o překladači, problémy a jejich řešení v rámci týmu, implementace jednotlivých částí nebo prostá diskuse na dané téma v rámci projektu. To vše hodnotíme jako velký přínos pro budoucí projekty na fakultě nebo v budoucím zaměstnání.