VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace k projektu do předmětů IFJ a IAL

Implementace interpretu imperativního jazyka IFJ14

Tým 018, varianta a/1/II

Marek Bielik, xbieli05, 20 %

Filip Gulán, xgulan00, 20 %

Filip Ježovica, xjezov01, 20 %

Lubomír Matuška, xmatus29, 20 %

Eduard Rybár, xrybar04, 20 %

Obsah

1. Úvod

Tato dokumentace se zabývá implementací interpretu pro imperativní jazyk IFJ14. Dokumentace je členěna do čtyř hlavních kapitol a několika podkapitol, které popisují způsob práce v týmu, postupný růst interpretu a spojování jeho dílčích částí ve funkční celek. Popsány jsou všechny podstatné části interpretu, použité algoritmy a problémy, které nám naši programátorskou cestu komplikovaly. Součástí dokumentace je samozřejmě i LL-gramatika, která je jádrem parseru, precedenční tabulka použitá pro zpracování výrazů a konečný automat scanneru. Závěr dokumentace obsahuje shrnutí našeho snažení.

2. Práce v týmu

2.1. Pravidelné schůzky a komunikace

Začali jsme se jednou týdně scházet poměrně brzy po utvoření týmu a zaregistrování zadání. Zpočátku nikdo z našich členů neměl konkrétní představu, jak bychom měli projekt řešit. Určitá představa se rýsovala až s odpřednášenými přednáškami. Práci jsme si rozdělovali poměrně volně podle toho, kdo cítil, že by daný problém zvládl vyřešit. Neměli jsme stanovené striktní časy, do kdy je určitý problém třeba vyřešit. Počítali jsme se svědomitým přístupem každého z nás. S blížícím se deadline se schůzky stávaly častějšími.

2.2. Správa zdrojového kódu

Nezbytnou částí úspěšného projektu je určitě to, aby všichni členové měli přesnou představu o aktuálním stavu zdrojových kódů projektu. V případě našeho projektu jsme se rozhodli použít webovou službu GitHub, která posloužila dokonale. Každý z našich členů měl možnost kdykoli a prakticky odkudkoli přidávat, či upravovat libovolné zdrojové kódy. Všichni členové tak měli po celou dobu práce na projektu přesnou představu o fázi jeho vývoje.

2.3. Vývoj

Každá vyvinutá část projektu se díky pravidelným schůzkám, webové službě GitHub a patřičné aktivitě každého z členů dočkala rychlé zpětné vazby. Nebyl tedy problém každou část interpretu rychle a efektivně upravit tak, aby lépe vyhovovala a spolupracovala s dalšími částmi. Postupně jsme objevovali dílčí problémy, které si autor každé z částí třeba ihned při implementaci neuvědomoval. Problémy tedy byly odstraňovány včas a spíše v zárodku, než v pozdější fázi vývoje.

3. Implementace

Cílem projektu bylo implementovat interpret jazyka IFJ14, což je jazyk založený na bázi Pascalu. S Pascalem se mnozí z nás již dříve setkali, takže to byla příjemná výhoda. Celý interpret se skládá z několika částí, které spolu úzce spolupracují a vytvářejí funkční celek.

3.1. Quick Sort

Pro vestavěnou funkci řazení jsme měli dle zadání použít algoritmus Quick Sort. Tento algoritmus patří ve své kategorii k nejrychlejším, jak již název napovídá. Pro implementaci jsme použili jeho rekurzivní zápis. Jeho rychlost se odvíjí od vhodné volby mediánu. V našem případě jsme si jako takový „pseudomedián“ stanovili hodnotu ze středu intervalu. Experimentálně je prokázáno, že toto číslo splní svoji roli obstojně. Implementaci jsme provedli dle studijní opory IAL.[1]

3.2. Knuth-Moris-Prattův algoritmus

Pro vestavěnou funkci vyhledávání podřetězce v řetězci jsme měli dle zadání použít Knuth-Moris-Prattův algoritmus. Tento algoritmus funguje na principu konečného automatu. Funkci jsme implementovali dle studijní opory IAL.[1] Funkce vrací „-1“ pokud se podřetězec v řetězci nenachází, nebo index na kterém místě se podřetězec nachází.

3.3. Lexikální analyzátor

Lexikální analyzátor funguje na principu konečného automatu, který načítá znaky ze zdrojového souboru a transformuje je na tokeny. Jednotlivé tokeny jsou reprezentovány strukturou, ve které je informace o stavu, ve kterém se automat nachází, datech tokenu a informaci o řádku (případně sloupci), na kterém se token ve zdrojovém souboru vyskytuje. Tyhle informace jsou nezbytné pro další činnost

interpretu. Tabulka klíčových slov je realizována jako pole polí znaků. Lexikální analyzátor navíc vrací konkrétní stav, o které klíčové slovo se jedná, což se ukázalo jako užitečné pro syntaktickou analýzu. Konečný automat lexikálního analyzátoru je detailně znázorněn v příloze.

3.4. Syntaktický analyzátor

Pro implementaci syntaktického analyzátoru jsme použili metodu prediktivní syntaktické analýzy, protože nám její implementace přišla elegantnější a případné opravy byly snáze implementovatelné. K implementaci jsme si museli vytvořit zásobník. Jádrem syntaktického analyzátoru je LL-gramatika, která je popsána v příloze.

Pro zpracování výrazů se přepínáme do precedenční analýzy, která je založena na, v příloze uvedené, precedenční tabulce, která je ve zdrojovém kódu opravdu fyzicky implementována, kde „E“ značí chybu, je to alternativa mezery použité v přednáškách z IFJ a „O“ je speciální stav, ktarý značí, že výraz je celý zkontrolován a je syntakticky správně.

3.5. Tabulka symbolů

Tabulku symbolů jsme dle zadání implementovali pomocí hashovací funkce. Implementace tabulky nabízí širokou škálu funkcí pro efektivní a elegantní práci. Během práce na projektu jsme mnohokrát změnili rozhodnutí ohledně toho, co vše bude do tabulky symbolů nezbytné ukládat. Nakonec si o každé položce vedeme poměrně mnoho informací. Každá položka tabulky nese informaci o svém identifikátoru, jakého je datového typu, ukazatel na data dané položky, pozici parametru z hlavičky funkce, adresu kde funkce začíná a příznak toho, jakém stavu se funkce nachází. Tyhle informace obsahují všechny položky, ale samozřejmě každá položka využívá pouze informace, které jsou pro ni podstatné. Například příznak, v jakém stavu se funkce nachází, se využívá pouze při dopředné deklaraci funkce a podobně.

3.6. Interpret

V projektu jsme využili generátor tříadresného kódu. Všechny tři adresy jsme využili jen zřídka. Obvykle se jednalo třeba jen o jednu adresu. Využívali jsme vhodnou instrukční sadu. Postupně se instrukce ukládaly na instrukční pásku, kde se později lineárně vykonávaly. V podmínkách, cyklech a funkcích by bylo lineární zpracování neúčinné. V těchto případech se generují vhodné instrukce skoků.

3.7. Testování

Testování jednotlivých částí probíhalo průběžně na systémech UNIXového typu. Využili jsme i druhý pokusný termín, který nedopadl úplně dobře, protože nám spousta částí v té době chyběla. V pozdější fázi jsme využili sadu asi 120 testů, kde jsme snadno a rychle doladili drobné, ale zásadní nedostatky.

4. Závěr

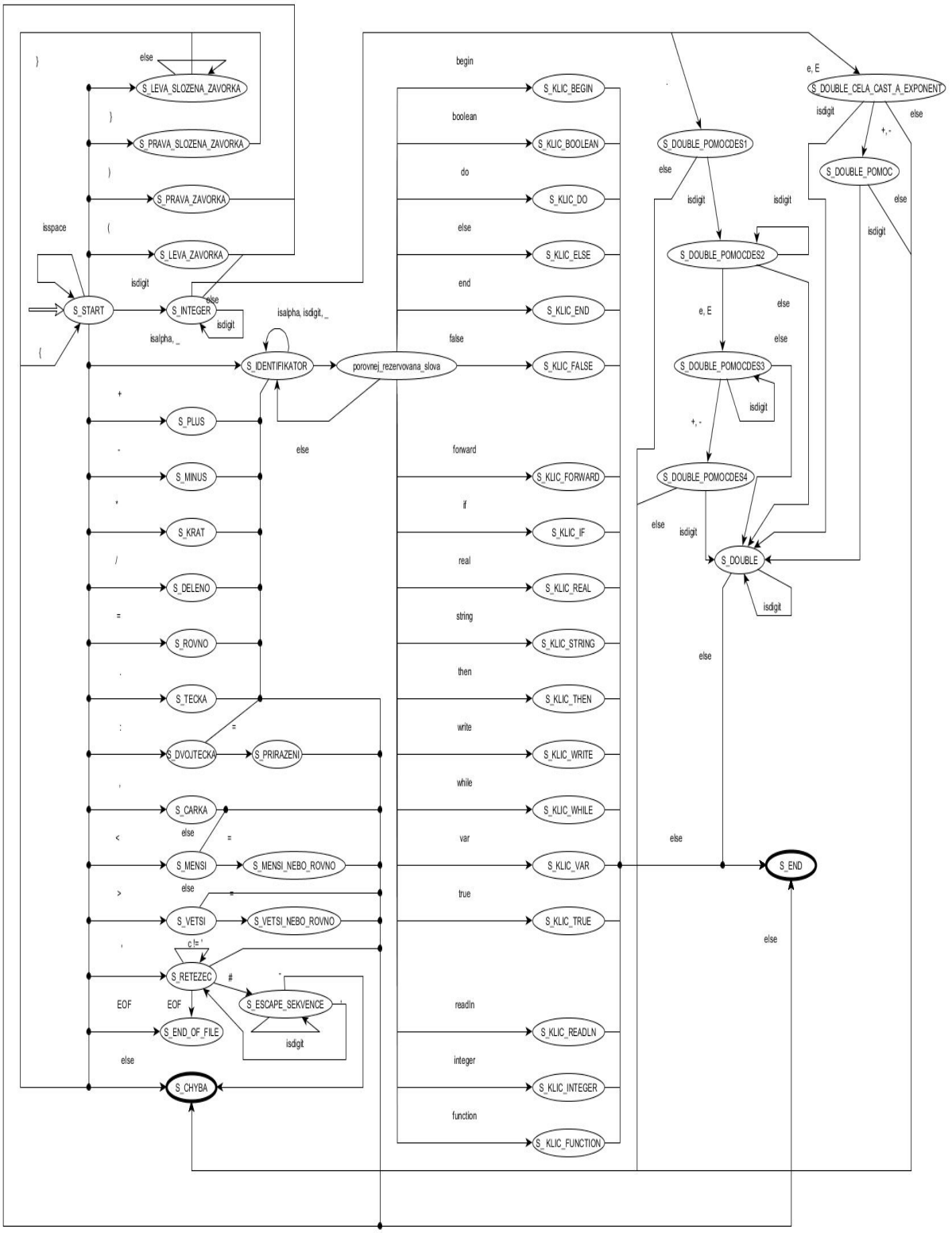
S projektem podobného rozsahu a obtížnosti se nikdo z našich členů zatím nesetkal. Z tohoto pohledu se určitě jednalo o zajímavou zkušenost a pořádnou výzvu. Při tvorbě projektu jsme prakticky využili znalostí nabytých v předmětech IFJ a IAL. Nebytnou součástí úspěšného projektu bylo udržovat patřičnou komunikaci v týmu. Pravidelné schůzky kompletního týmu probíhaly minimálně jedenkrát týdně, kdy jsme si vzájemně sdělovali a vysvětlovali princip a fungování nově implementovaných částí projektu a rozdělili si další práci. Výsledným produktem naší spolupráce by měl být funkční interpret jazyka IFJ14.

5. Použitá literatura

[1] Studijní opora IAL

[2] Přednášky IFJ

A Konečný automat lexikálního analyzátoru



B LL gramatika

INIT → VLIST FLIST begin STLIST end .

VLIST → ε
VLIST → var VDEC ; NVLIST

VDEC → id : TYPE

NVLIST → ε
NVLIST → VDEC ; NVLIST

TYPE → integer
TYPE → real
TYPE → string
TYPE → boolean

FLIST → ε
FLIST → function id (PLIST) : TYPE ; FUNC

FUNC → forward ; FLIST
FUNC → VLIST begin STLIST end ; FLIST

PLIST → ε
PLIST → id : TYPE NPLIST

NPLIST → ε
NPLIST → ; id : TYPE NPLIST

STLIST → ε
STLIST → STAT NSTLIST

NSTLIST → ε
NSTLIST → ; STAT NSTLIST

STAT → BSTAT
STAT → id := RHS
STAT → if E then BSTAT else BSTAT
STAT → while E do BSTAT
STAT → readln (id)
STAT → write (SPLIST)

RHS → E
RHS → fid (SPLIST)

BSTAT → begin STLIST end

SPLIST → ε
SPLIST → id NSPLIST
SPLIST → integer NSPLIST
SPLIST → real NSPLIST
SPLIST → string NSPLIST
SPLIST → true NSPLIST
SPLIST → false NSPLIST

NSPLIST → ε
NSPLIST → , id NSPLIST
NSPLIST → , integer NSPLIST
NSPLIST → , real NSPLIST
NSPLIST → , string NSPLIST
NSPLIST → , true NSPLIST
NSPLIST → , false NSPLIST

C Precedenční tabulka

	*	/	+	-	<	>	<=	>=	=	<>	(	)	id	$	int	real	string	true	false
*	>	>	>	>	>	>	>	>	>	>	<	>	<	>	<	<	<	<	<
/	>	>	>	>	>	>	>	>	>	>	<	>	<	>	<	<	<	<	<
+	<	<	>	>	>	>	>	>	>	>	<	>	<	>	<	<	<	<	<
-	<	<	>	>	>	>	>	>	>	>	<	>	<	>	<	<	<	<	<
<	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<	<	<
>	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<	<	<
<=	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<	<	<
>=	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<	<	<
=	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<	<	<
<>	<	<	<	<	>	>	>	>	>	>	<	>	<	>	<	<	<	<	<
(	<	<	<	<	<	<	<	<	<	<	<	=	<	E	<	<	<	<	<
)	>	>	>	>	>	>	>	>	>	>	E	>	E	>	E	E	E	E	E
id	>	>	>	>	>	>	>	>	>	>	E	>	E	>	E	E	E	E	E
$	<	<	<	<	<	<	<	<	<	<	<	E	<	O	<	<	<	<	<
int	>	>	>	>	>	>	>	>	>	>	E	>	E	>	E	E	E	E	E
real	>	>	>	>	>	>	>	>	>	>	E	>	E	>	E	E	E	E	E
string	>	>	>	>	>	>	>	>	>	>	E	>	E	>	E	E	E	E	E
true	>	>	>	>	>	>	>	>	>	>	E	>	E	>	E	E	E	E	E
false	>	>	>	>	>	>	>	>	>	>	E	>	E	>	E	E	E	E	E

Pravidla pro precedenční analýzu:

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 → id

E → integer

E → real

E → string

E → true

E → false