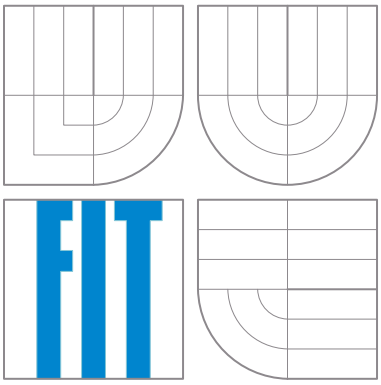


Dokumentace k projektu pro předměty IFJ a IAL

Implementace interpretu imperativního jazyka IFJ11

Tým 028, varianta b/2/II

10. května 2016

Rozšíření: IFTHEN: Podpora příkazu **if-then-elseif-else-end**
FUNEXP: Funkce mohou být součástí výrazu
LOCALEXPR: Inicializace proměnné přiřazením výrazu
REPEAT: Podpora cyklu **repeat ... until**

Autoři: Vojtěch Bartl, xbartl03, 25%
Pavel Frýz, xfryzp00, 25%
Jan Horák, xhorak34, 25%
Jiří Staněk, xstane31, 25%

Obsah

1 Úvod

Interprety jsou rozsáhlé programy, jejichž implementace spočívá především v rozčlenění na menší celky.

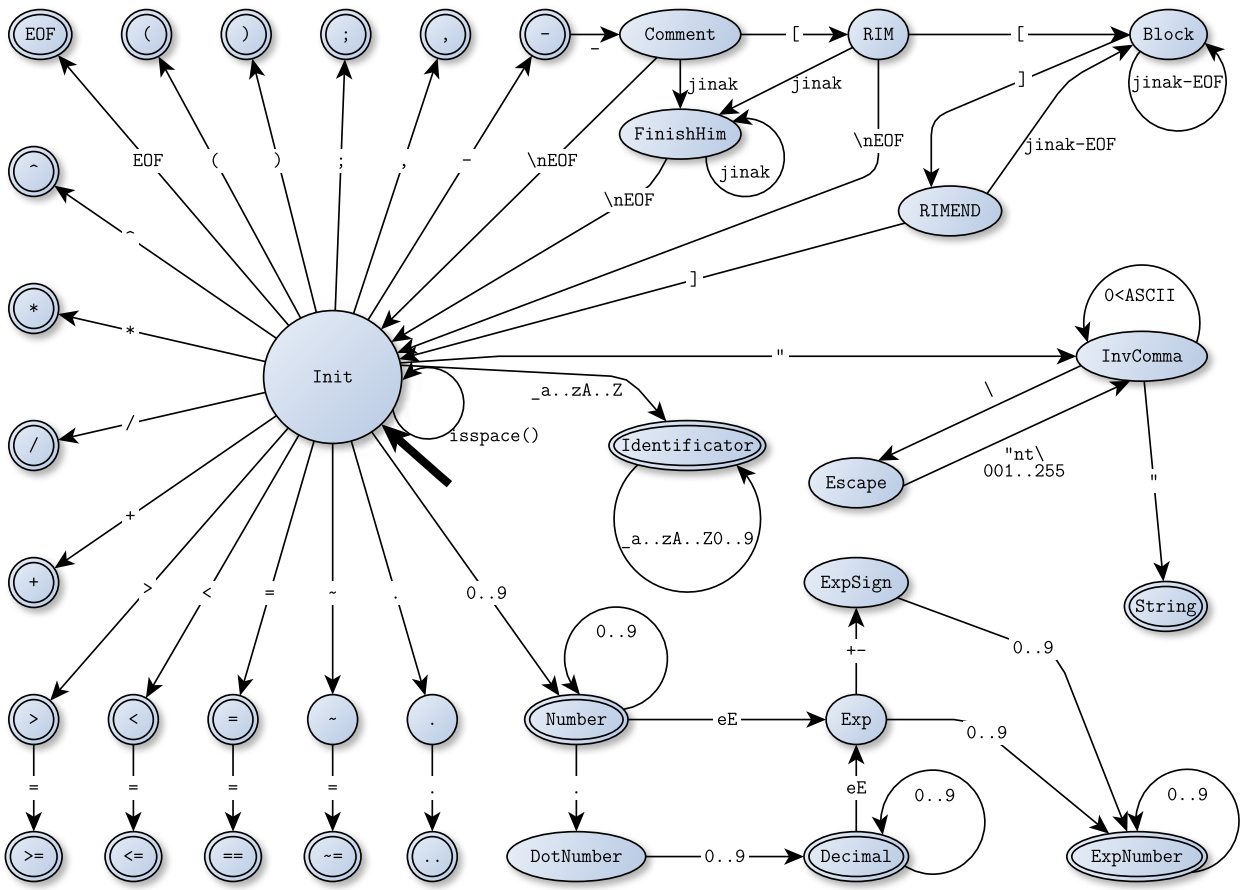
V tomto projektu bylo naším úkolem vytvořit interpret jazyka IFJ11, který je podmnožinou jazyka Lua.

Projekt je implementován v jazyce C. Funguje jako konzolová aplikace, která načte soubor se zdrojovým kódem v jazyce IFJ11 a interpretuje jej. Soubor je předán jako parametr příkazové řádky.

Při řešení projektu jsme využili poznatků z předmětů IFJ a IAL.

2 Lexikální analyzátor

Lexikální analyzátor rozděluje vstupní posloupnost znaků na lexémy, které jsou v praxi reprezentovány pomocí tokenů. Lexikální analyzátor jsme realizovali jako konečný automat, který je graficky znázorněn níže. Lexikální analyzátor je umístěn v souboru `scanner.c`.



3 Syntaktický a sémantický analyzátor

Syntaktický analyzátor tvoří z tokenů datovou strukturu zvanou derivační strom. Ta je poté využita v sémantickém analyzátoru, kde je transformována na abstraktní syntaktický strom. Zatímco syntaktický analyzátor kontroluje gramatickou korektnost posloupnosti tokenů, sémantický analyzátor kontroluje správnost operací.

V našem interpretu je syntaktická a sémantická analýza rozdělena na dvě části. První je umístěna v souboru `parser.c` a zpracovává celou strukturu programu mimo výrazů. Využili jsme metodu rekurzivního sestupu založenou na LL-gramatice s následujícím pravidly.

```
<program>    →  <functlist> "EOF"
<functlist>  →  "function" <funct>
<funct>      →  "id" "(" <paramlist> <declrlist> <functlist>
<funct>      →  "main" "(" ")" <declrlist> ";"
<paramlist>  →  "id" <param>
<paramlist>  →  ")"
<param>      →  "," "id" <param>
<param>      →  ")"
<declrlist>  →  "local" "id" <declr> <declrlist>
<declrlist>  →  <statlist> "end"
<declr>      →  "=" <expr> ";"
<declr>      →  ";"
<statlist>   →  <stat> <statlist>
<statlist>   →  "end"
<statlist>   →  "else"
<statlist>   →  "elseif"
<statlist>   →  "until"
<stat>       →  "while" <expr> "do" <statlist> "end" ";"
<stat>       →  "return" <expr> ";"
<stat>       →  "write" "(" <exprlist> ";"
<stat>       →  "id" "=" <assign> ";"
<stat>       →  "repeat" <statlist> "until" <expr> ";"
<stat>       →  "if" <expr> "then" <statlist> <ifstat> ";"
<ifstat>     →  "end"
<ifstat>     →  "else" <statlist> "end"
<ifstat>     →  "elseif" <expr> "then" <statlist> <ifstat>
<exprlist>   →  <expr> <writexpr>
<exprlist>   →  ")"
<writexpr>   →  "," <expr> <writexpr>
<writexpr>   →  ")"
<assign>     →  "read" "(" "format" ")"
<assign>     →  <expr>
```

V případě, že se během první části narazí na výraz `<expr>`, řízení se předá druhé části, která zpracovává výrazy. Tato část je zpracována pomocí precedenční syntaktické analýzy. V průběhu zpracování výrazu se zpracovává volání funkce metodou shora dolů dle následujícího pravidla.

```
<functcall>  →  "("[<expr> {,<expr>}] ")"
```

4 Interpret

Poslední částí našeho programu je interpret. Během syntaktické a sémantické analýzy se do listu instrukcí, který je implementován jako jednosměrný lineární seznam, přidávají instrukce. Po-

kud proběhne syntaktická a sémantická analýza bez chyby, interpret načte jednu instrukci ze seznamu, vykoná ji a poté pokračuje na další instrukci v seznamu.

4.1 Volání funkcí

Volání a návrat z funkce je ralizován pomocí tří instrukcí. Instrukce `IMARK` označí vrchol zásobníku, od tohoto místa se poté počítají pozice parametrů a lokálních proměnných. Poté jsou na zásobník přidány argumenty funkce. Následně se vykoná instrukce `ICALL`, která uloží návratovou adresu na označené místo a provede volání funkce. Pro návrat z funkce poté slouží instrukce `IRET`, která smaže zásobník až po označené místo a vloží návratovou hodnotu na zásobník.

4.2 Lokální proměnné a parametry funkcí

Hodnoty proměnných a parametrů jsou uloženy na zásobníku od posledně označeného místa instrukcí `IMARK` v relativním pořadí jejich deklarací ve zdrojovém souboru. Pokud má funkce `n` parametrů a `m` lokálních proměnných, pak parametry jsou uloženy na pozicích `1..n` a proměnné na pozicích `n+1..n+m`.

K vložení hodnoty na zásobník (i literálů) slouží instrukce `IPUSH`. Pro uložení hodnoty z vrcholu zásobníku do proměnné slouží instrukce `IPOP`.

4.3 Vyčíslovaní výrazů

Pro vyčíslování výrazů jsme vytvořili 12 instrukcí, kde každá instukce odpovídá jednomu aritmetickému nebo relačnímu operátoru. Operandy se berou z vrcholu zásobníku, kam se následně uloží výsledek operace.

4.4 Další instrukce

Kromě výše uvedených instrukcí jsme vytvořili 6 instrukcí pro vestavěné funkce a příkazy read a write, 2 skokové instrukce pro nepodmíněný a podmíněný skok a instrukci `IHALT` pro ukončení interpretace.

5 Algoritmy do předmětu IAL

5.1 Boyer-Mooreův algoritmus

Boyer-Mooreův algoritmus je implementován v souboru `ial.c` ve funkci `find` a `computeJumps`. Jedná se o metodu vyhledání podřetězce v řetězci. Pole `charJump` má velikost 256, což je počet znaků v ASCII tabulce, a pro každý znak obsahuje velikost posunu podřetězce. Funkce `computeJumps` je pomocná funkce, která naplní pole `charJump` podle vyhledávaného podřetězce. Funkce `find` porovnává podřetězec zprava znak po znaku. V případě neúspěšného porovnání se použije hodnota posunu z pole charJump, podřetězec se posune a porovnávání pokračuje.

5.2 Heap sort

Řazení metodou Heap sort je implementováno v souboru `ial.c` ve funkcích `heapSort` a `heapExtend`. Funkce heapExtend je pomocná funkce, která dokáže prodloužit haldu o jeden prvek. Funkce `heapSort` obsahuje dva kroky. V obou krocích je využitá právě pomocná funkce `heapExtend`. V prvním kroku je halda rozšířena na celé pole. Druhá polovina pole pravidla haldy automaticky splňuje, několikanásobným voláním funkce `heapExtend` postupně haldu rozšiřujeme vždy o jeden prvek. Ve druhé fázi vybereme z haldy největší prvek (ten první) a vyměníme ho s posledním prvkem. V této chvíli jsme však porušili pravidla haldy a proto musíme znovu zavolat pomocnou funkci `heapExtend`.

5.3 Tabulka symbolů implementovaná pomocí hashovací tabulky

Tabulka symbolů je implementována v souboru `ial.c`. Do tabulky symbolů ukládáme tři typy symbolů. Symbol typu funkce, typu proměnná a typu literál (konstanta). Pro každý typ si uchováváme specifická data. Pro funkci si ukládáme její adresu a počet parametrů. Pro proměnné si ukládáme název funkce ve které je daná proměnná deklarovaná a pozici proměnné ve funkci (jako kolikatý parametr proměnná ve funkci vystupuje). Pro literál si ukládáme jeho typ (string, number, boolean, nil) a jeho hodnotu.

Hashovací funkci jsme převzali z předmětu IJC, kde byla součástí jednoho z projektů. Seznam synonym je zřetězen explicitně do jednosměrně vázaného lineárního seznamu, kde se každý nový prvek vkládá na začátek seznamu. Velikost hashovací tabulky je zvolena jako prvočíslo 193, což by měla být adekvátní velikost pro běžné programy.

6 Rozšíření

V rámci projektu jsme se rozhodli implementovat následující rozšíření:

- IFTHEN
- FUNEXP
- LOCALEXPR
- REPEAT

Rozšíření jsou vyřešena dodatečnými pravidly v LL-gramatice, viz sekce 3. Těmi je vyřešen i problém návaznosti příkazu `else` a `elseif`, který se přiřadí k poslednímu příkazu `if` případně `elseif`.

Pro tato rozšíření jsme se rozhodli ještě před implementací parseru a zabudovali jsme je rovnou při jeho implementaci.

Při rozšíření `FUNEXP` jsme se museli zamyslet nad voláním funkcí, viz sekce 4.1, a také nad zpracováním volání funkce ve výrazu. To je implementováno v souboru `expr.c` ve funkci `functcall`, která se zavolá ve chvíli, kdy se při zpracování výrazů narazí na identifikátor funkce. Tato funkce kontroluje syntaxi volání a počet zadaných argumentů, po překročení počtu parametrů volané funkce již pro další argumenty negeneruje instrukce. V případě nedostatku argumentů, doplní chybějící počet konstantou `NIL`.

7 Práce v týmu

Náš tým se dal dohromady velmi prozaickým způsobem. Chodili jsme na stejné gymnázium, sedávali jsme spolu na přednáškách, a tak při skládání týmu nebylo co řešit.

První problém, se kterým jsme se v týmu potkali, byla role vedoucího. Nikdo se totiž na tuto pozici nijak netlačil, což vyvrcholilo tím, že jsme se při přihlašování vedoucích přihlásili rovnou dva (prostě proto, aby tam aspoň někdo byl). Naštěstí se dalo po uzávěrce ještě odhlásit, takže jsme se nakonec i oficiálně šťastně potkali v jednom týmu.

Rozdělení práce probíhalo hlavně na bázi „když se někomu chce, tak ať něco udělá“. První oficální rozdělení proběhlo někdy v půlce října, kdy jsme se rozdělili na dvě dvojice, z nichž jedna dělala interpret a druhá lexikální analyzátor.

Domluva v týmu probíhala hlavně na pondělních přednáškách a ve středu ve dvouhodinové pauze mezi dvěma přednáškami. Ke komunikaci jsme už někdy v září vytvořili i diskuzní fórum, které zůstalo na celém jednom příspěvku. Mimo osobní domluvu jsme komunikovali ještě e-mailem.

K uchovávání aktuálního stavu projektu jsme používali svn na školním serveru merlin. Měli jsme snahu zaznamenávat až větší změny, abychom měli méně commitů, přesto se jejich počet vyšplhal na 49.

8 Závěr

Tento projekt pro nás byl jednoznačně pozitivní zkušeností. Naučili jsme se spolupracovat ve skupině lidí a vytvářet společně větší projekty obsahující více navzájem kooperujících modulů.

Program byl testován na několika zdrojových kódech jazyka IFJ11 a všechny proběhly v pořádku. Doufáme, že tomu tak bude i u těch, které budou použity při testování správnosti programu.

A Metriky kódu

Počet souborů: 24 souborů

Počet řádků zdrojového textu: 4423 řádků

Velikost statických dat: 31444B

Velikost spustitelného souboru: 39612B (systém Linux, 64 bitová architektura, při překladu bez ladicích informací)