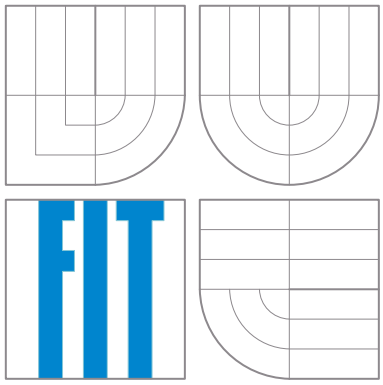

Dokumentace k projektu pro předměty IFJ a IAL

Implementace interpretu imperativního jazyka IFJ12

Tým XXX, varianta x/x/x

Vedoucí týmu:

12. prosince 2014

Řešitelé: 1BIT Pružina Tomáš, xpruzi01@stud.fit.vutbr.cz

Fakulta Informačních Technologií
Vysoké Učení Technické v Brně

Obsah

1 Úvod **1**

2 Moduly interpretu jazyka **1**
2.1 Lexikální analyzátor 1
2.2 Syntaktický analyzátor 1
2.2.1 Shunting-yard algoritmus 2
2.3 Interpret 2
2.3.1 Tabulka symbolů 3

3 Postup při implementaci řešení **3**
3.1 Funkce `sort` 3
3.2 Funkce `find` 3

4 LL-tabulka **4**

5 Závěr **5**

A Metriky kódu **5**

1 Úvod

Implementace překladače imperativního jazyka je netriviální úloha, proto je vhodné ji rozdělit na podúlohy (2), které jsou vhodně rozdělené mezi členy týmu, který je vedené a kontrolován týmovým vedoucím.

Tento dokument sa skládá z N částí, které popisují jednotlivé moduly interpretu imperativního jazyka IFJ14, a to jsou lexikální analyzátor (2.1), syntaktický analyzátor (2.2) využívající rekurzivní sestup (), který je pro potřeby syntaktické analýzy výrazů rozšířený o Shunting-Yard (2.2.1) algoritmus[1]. Kontrola syntaxe a sémantiky probíhá z velké části při jednom průchodu, následné kontroly sémantiky probíhají až za běhu.

2 Moduly interpretu jazyka

2.1 Lexikální analyzátor

Syntaktický analyzátor potřebuje ke své činnosti lexikální analyzátor. Ten předává na žádost syntaktického analyzátoru tzv. tokeny, které získává postupným čtením vstupního souboru, který obsahuje zdrojový kód napsaný v jazyce IFJ14. Samotný výsledný token je reprezentací lexému (identifikátor, klíčové slovo, příkaz přiřazení atp.).

Pro úspěšné rozlišení typu lexému se v naší implementaci používá konečný automat, jehož struktura je znázorněna na obrázku (**??**).

Jestliže se konečný automat v době zpracovávání řetězce zdrojového souboru document do chybového stavu `lexerrref`, řetězec je nepřijatý a jedná se o lexikální chybu. Samotná implementace lexikálního analyzátoru patří k jednodušším částem projektu, avšak jeho návrh a testování zabralo nemálo času.

2.2 Syntaktický analyzátor

Jak již bylo zmíněno, syntaktický analyzátor žádá lexikální analyzátor o tokeny a na základě sekvence po sobě jdoucích tokenů sestavuje abstraktní syntaktický strom (dále jen AST). Tato sekvence musí odpovídat pravidlům syntaxe jazyka IFJ14, v opačném případě se jedná o syntaktickou chybu.

Jazyk IFJ14 je staticky typovaný jazyk, tedy typové kontroly probíhají během překladu, v našem případě právě při průchodu syntaktickým analyzátorem. Je tedy nutné zachytávat chybové vstupy a přímo ukončovat průchod a tím i celou aplikaci.

Jako vnitřní kód programu jsme si zvolili AST, tedy binární strom a rozšířili jsme jeho strukturu o potřebné položky pro práci interpretu. Jak již bylo zmíněno výše, jedná se o binární strom, má tedy dva synovské uzly. Avšak pro určité typy uzlů je mít nemusí. Například uzel pro operaci `NOT` má pouze levý synovský podstrom.

Pro převod příkazů obsahujících tělo na AST uzly byla v týmu domluvena konvence, že primárním uzlem pro tělo bude levý podstrom. Tedy například cyklus `while` má svoje tělo uložené v levém podstromu. Zde je potřeba zmínit, že příkaz `if` má v podstatě dvě sekvence příkazů a to pravdivou a nepravdivou. Je tedy potřeba zajistit, aby se obě uchovaly v jednom uzlu AST.

[1] Byla nám dovolena výjimka, implementovat a náležitě zdokumentovat tento algoritmus oproti předepsané precedenční analýze

Řešení je jednoduché a více než samozřejmé, uložení pravdivé sekce do levého poduzlu stromu a nepravdivou sekci do uzlu pravého. U příkazů s podmíněným chováním (if, while, repeat-until, atp.) bylo nutné uchovat podmínku, tedy výraz, který určuje, jestli je příkaz proveditelný či není. Primárně pro tento účel slouží dodatečná položka uzlů AST, která svým typem `void*` umožňuje mnohostranné využití.

Zvláštní případ použití dodatečné položky je například u definice či volání funkce. V tomto případě obsahuje strukturu `String` se jménem funkce. Levý podstrom opět obsahuje tělo funkce a v pravém podstromu je uložen uzel se strukturou `varspars`, která se složená z dvou front, jedné pro lokální proměnné funkce a druhá pro parametry funkce. Tyto fronty jsou naplněny při definici funkce, při deklaraci (použití `forward` deklarace) je naplněna pouze fronta parametrů. Více o funkcích v **??**.

2.2.1 Shunting-yard algoritmus

Algoritmus zpracování výrazů je založen na prioritě operátorů, které buď vkládá nebo odebírá ze zásobníku a vkládá do výstupní sekvence. Tento algoritmus primárně slouží k převodu infixového zápisu výrazů do postfixového, tedy lépe zpracovatelného a převeditelného do AST.

Tento algoritmus byl pro potřeby projektu upraven, aby přímo generoval AST místo výstupního postfixového zápisu a zároveň u tvorby jednotlivých uzlů zastupujících operace kontroloval, zdali datový typ vrcholových uzlů synovských uzlů odpovídá sémantickým pravidlům zpracovávání výrazů.

Při tvorbě výrazů je nutné kontrolovat spoustů aspektů validního výrazu. To jsou nejen datové typy jednotlivých operandů, ale i v případě proměnných jejich inicializovanost. Tedy pokud proměnná nebyla inicializovaná, není možné ji použít.

2.3 Interpret

Interpret je závěrečnou částí projektu. Rekurzivně zpracovává AST reprezentovaný binárním stromem a postupně vykonává operace nad ním definované.

Kořenem tohoto AST je vždy typ převedené operace, přičemž obsahuje předem definované nepovinné parametry (příkladem je podstrom volání funkce, který obsahuje předávání proměnné). Tedy každý uzel AST představuje jednu atomickou operaci, kterou je řízený samotný běh programu.

Interpret samozřejmě nepracuje jen se samotným syntaktickým stromem, ale taktéž s tabulkou symbolů, která obsahuje data potřebná na to, aby interpret mohl vykonávat nějakou užitočnou práci definovanou v těle programu.

Interpret tedy spravuje tabulky symbolů, dynamicky je za běhu vykonávaného programu vytváří a (podla potřeby) ruší.

V neposlední řadě interpret vykonává sémantickou kontrolu u operacích, které syntaktický analyzátor v době zpracování vstupního programu a generování abstraktního syntaktického stromu neřešil.

Příkladem takovéto kontroly je používání (čtení) proměnné před její řádnou definicí (přiřazením hodnoty) anebo dělení nulou v aritmetickém výrazu.

Takovéto běhové chyby jsou v interpretě řešené ukončením interpretace a navrácením chybového kódu podle zadání imperativního jazyka IFJ14.

2.3.1 Tabulka symbolů

Tabulka symbolů je implementována pomocí binárního vyhladávacího stromu, přičemž se rozlišuje několik úrovní tabulky symbolů.

Každý program obsahuje globální tabulku symbolů (ve které jsou uložené ukazatele na funkce) a lokální tabulku symbolů, která obsahuje běhové proměnné vykonávaného programu.

Z implemetačního hlediska jsou lokální tabulky symbolů (stromy) ukládané na zásobník, přičemž platí, že při volání funkce se vytvoří nová (lokální) tabulka symbolů, do které se skopírují příslušné parametry volané funkce z nižší vrstvy lokální tabulky (popř. z globální tabulky symbolů).

Taková lokální tabulka taktéž obsahuje návratovu hodnotu vykonávané funkce, která umožňuje rekurzivní volání funkcí a samozřejmě umožňuje vykonávat tělo funkce bez nutnosti jakkoliv zasahovat do abstraktního syntaktického stromu.

3 Postup při implementaci řešení

3.1 Funkce `sort`

K implementaci funkce sort jsme podle zadání použili algoritmus quicksort. Kromě funkce quicksort jsme potřebovali ještě pomocnou funkci `partition`, která rozděluje řazené pole na dvě části. V jedné takto vzniklé části jsou prvky s menší hodnotou než je prvek rozdělující tyto části (pseudomedián, dále jen PM) a v druhé části prvky s hodnotou větší než PM. Index PM určíme jako $(l + p)/2$, kde i je index nejlevějšího prvku řazeného úseku pole a j index nejpravějšího prvku tohoto pole. Všechny prvky s hodnotou nižší než je PM poté přesuneme na levou stranu od PM a prvky s vyšší hodnotou na pravou stranu. Poté aplikujeme algoritmus quicksort na obě poloviny pole.

Konkrétní implementace algoritmu je převzata z přednášek předmětu Algoritmy (IAL2014 [2] – 10. přednáška 33. slide). Pro snazší konverzi do jazyka C a pro pochopení algoritmu samotného jsme vytvořili vývojový graf a podle grafu jsme sepsali samotný kód v jazyce C.

Modifikace: Algoritmus popsaný v přednáškách obsahoval prohození prvku se sebou samým, takže jsme provedli drobnou úpravu, která tomuto zabránila.

3.2 Funkce `find`

Pro funkci find jsme dle zadání použili KMP (Knuth-Morris-Prattův algoritmus). Jedná se o algoritmus využívající konečný automat. Tento konečný automat využívá hrany ANO/NE. Nejprve bylo nutné vytvořit si pole `FAIL`, do kterého se uloží zjištěné hodnoty posunů. Poté přijde na řadu samotné vyhledávání řetězce. Nejprve zarovnáme podřetězec a řetězec na první index. Pak se jen porovnávají znaky, pokud jsou shodné, zvyšuje se index o 1 v podřetězci, který hledáme (`pattern`) a zároveň i v řetězci, ve kterém vyhledáváme (`text`). Pokud nikoliv, tak se vracím po hraně "N"(NE hrana) neboli `FAIL` vektor. Je-li pak hodnota indexu v hledaném řetězci rovna délce hledaného podřetězce, tak byl nalezen vzorek na indexu $TInd - PL$, kde $TInd$ je index řetězce a PL je délka hledaného podřetězce, jinak nic nenašel a vrací se 0.

Algoritmus byl převzán a přepsán do jazyka C z přednášek předmětu Algoritmy (IAL2014 [2] – 11. přednáška 18. slide).

Modifikace: Pro návrat 0 v případě neúspěchu byla provedena modifikace algoritmu z přednášek, kde byl vracen poslední dosažený index, tedy délka řetězce.

4 LL-tabulka

```
PROGRAM         -> VARS FUNC begin CMD_LIST enddot

VARS            -> var VAR_DEF VAR_DEFS
VARS            -> eps
VAR_DEFS        -> VAR_DEF VAR_DEFS
VAR_DEFS        -> eps
VAR_DEF         -> id : DT_TYPE ;

DT_TYPE         -> integer
DT_TYPE         -> real
DT_TYPE         -> boolean
DT_TYPE         -> string

FUNC            -> eps
FUNC            -> HEADER

HEADER          -> function id ( DEF_PARAMS ) : DT_TYPE ; AFTER_HEADER

AFTER_HEADER    -> forward ;
AFTER_HEADER    -> VARS BODY ;

DEF_PARAMS      -> eps
DEF_PARAMS      -> DEF_PAR DEF_PAR_LIST

DEF_PAR_LIST    -> eps
DEF_PAR_LIST    -> ; DEF_PAR

DEF_PAR         -> id : DT_TYPE

BODYN           -> begin CMD_LIST_N end
CMD_LIST_N      -> eps
CMD_LIST_N      -> CMD CMD_FOLLOW
CMD_FOLLOW      -> eps
CMD_FOLLOW      -> ; CMD CMD_FOLLOW

BODY            -> begin CMD_LIST end
CMD_LIST        -> CMD CMD_FOLLOW

CMD             -> ASSIGN
CMD             -> BODYN
CMD             -> IF
```

```
CMD             -> WHILE
CMD             -> WRITE
CMD             -> READLN

ASSIGN          -> id := AFTER_ASSIGN
AFTER_ASSIGN    -> EXPR
AFTER_ASSIGN    -> CALL

IF              -> if EXPR then BODYN IF_ELSE
IF_ELSE         -> eps
IF_ELSE         -> else BODYN
WHILE           -> while EXPR do BODYN

CALL            -> id ( TERM_LIST )
CALL            -> sort ( dt_str )
CALL            -> find ( dt_str , dt_str )
CALL            -> length ( dt_str )
CALL            -> copy ( dt_str , dt_str , dt_int )

READLN          -> readln ( id )
WRIT            -> write ( TERM_LIST )

TERM_LIST       -> TERM
TERM_FOLLOW     -> , TERM TERM_FOLLOW
TERM_FOLLOW     -> eps

TERM            -> id
TERM            -> dt_int
TERM            -> dt_real
TERM            -> dt_bool
TERM            -> dt_str

EXPR            -> vyraz
```

5 Závěr

A Metriky kódu

Reference

[1] HONZÍK J. M.: *Studijní opora pro předmět Algoritmy.* Elektronický text. FIT VUT v Brně

[2] HONZÍK J. M.: *Přednášky k předmětu Algoritmy.* Soubor elektronických textů. FIT VUT v Brně

[3] MEDUNA A., LUKÁŠ R., *Podklady k přednáškám.* Elektronický text. FIT VUT v Brně