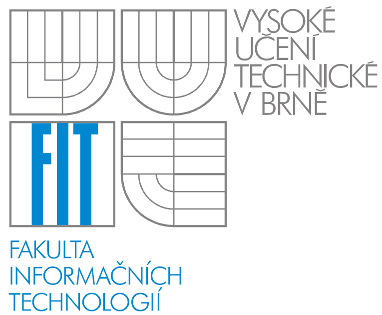


Dokumentace ke společnému projektu IFJ a IAL

Implementace překladače imperativního jazyka IFJ05

16. prosince 2005

řešitelé:

David Bařina	`xbarin02`
Kamil Dudka	`xdudka00`
Jakub Filák	`xfilak01`
Lukáš Hefka	`xhefka00`
Luděk Hradil	`xhradi08`

Úvod

Tato dokumentace popisuje implementaci překladače imperativního jazyka **IFJ05**. Vybrali jsme si zadání 2(c)/1, což nám udávalo, že máme do interpretu vytvořit vestavěnou funkci Heapsort s řazením řetězců a řešit tabulku symbolů pomocí binárního stromu.

Překladač se skládá ze tří hlavních bloků. Jádrem překladače je **Syntaktický analyzátor**, má na starosti překlad zdrojového kódu. Pomocí **Lexikálního analyzátoru** načítá zdrojový kód a podle gramatických pravidel jazyka IFJ05 jej překládá na posloupnost pseudoinstrukcí. V případě úspěšného překladu se spustí **Modul interpretace** (runtime), který posloupnost instrukcí vykoná – interpretuje.

Tyto základní 3 bloky budou dále v této dokumentaci stručně popsány.

Popis řešení

modul lexikálního analyzátoru - `scanner`

Definuje typ tokenu `TScannerToken` a funkci `ScannerGetNextToken()`, která na požádání syntaktického analyzátoru načte ze vstupního souboru jeden lexém. Ten je vrácen jako token, který obsahuje typ. Typem může být například číslo, středník, klíčové slovo apod. Token dále obsahuje ukazatel na obsah, který se liší v závislosti na typu tokenu. Další informací je číslo řádku, na kterém se token nachází. V případě, že scanner narazí na konstantu (celé číslo, reálné číslo nebo řetězec), uloží jej do tabulky konstant (modul `variables`). V tokenu je pak uložen odkaz na položku této tabulky. V případě, že narazí na identifikátor, uloží jej do tabulky symbolů (modul `symbols`). V tomto případě je v tokenu uložen odkaz na položku tabulky symbolů.

Vlastní scanner je řešen jako konečný automat. Načítá znaky ze vstupního souboru pomocí funkce `fgetc()`. Protože v některých případech je třeba skládat z načtených znaků řetězec, jsou v tomto modulu definovány také funkce pro práci s nekonečnými řetězci. V nich je použita funkce `realloc()`, která dokáže zvětšit alokovaný blok paměti. Při alokování paměti probíhá exponenciálně. V některých případech načte scanner znak patřící k dalšímu tokenu. V těchto případech volá funkci `ungetc()`, která vrátí znak zpět do vstupního souboru.

Jako jednoduché **rozšíření** jsme implementovali jednořádkový komentář, který začíná znakem # a končí koncem řádku.

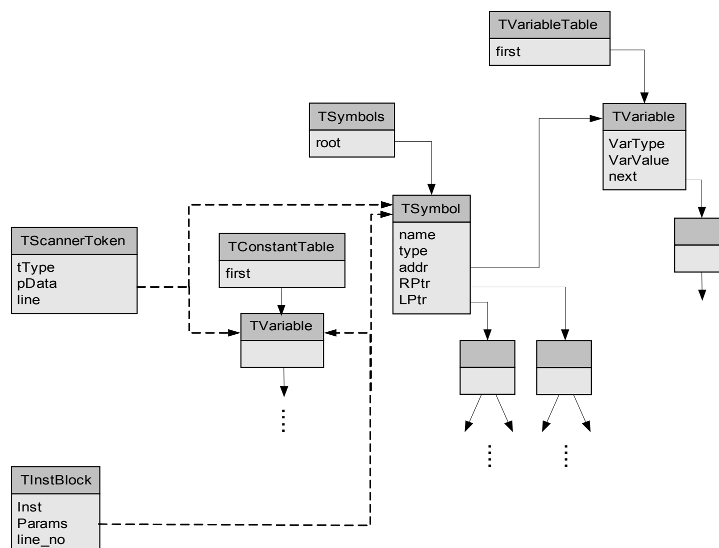
modul tabulky symbolů - `symbols`

V tomto modulu je definován datový typ `TSymbol`, který reprezentuje jeden symbol v tabulce symbolů. Tyto symboly jsou uspořádány do binárního vyhledávacího stromu. Každý symbol tedy obsahuje dva ukazatele (na levého a pravého syna). Symbol dále obsahuje jméno (identifikátor), typ (proměnná nebo návěští) a obsah. Obsahem se myslí index do pásky instrukcí u návěští. V případě proměnné je to ukazatel do tabulky proměnných (obdoba tabulky konstant). V tomto modulu jsou definovány funkce pro přidávání do symbolu do tabulky (v případě již existující položky se pouze vrátí ukazatel na již existující symbol). Tuto funkci využívá scanner. Dále je zde funkce pro označení symbolu návěštím (označuje parser) a funkce pro zjištění jména a typu symbolu, indexu pásky a adresy proměnné pro modul runtime. Tabulka symbolů neobsahuje informaci o typu proměnné (celé číslo, reálné číslo nebo řetězec). Tato informace je přímo součástí obsahu proměnné a je tedy měnitelná při běhu programu.

modul tabulky konstant a proměnných - `variables`

Pro uchování hodnot konstant a později při běhu programu i proměnných je potřeba někde je uchovat. V našem případě používáme dvě oddělené tabulky. Tabulka konstant je naplněna scannerem při analyzování vstupního souboru. Duplicitní konstanty jsou zde uloženy pouze jednou. Obsah této tabulky není možné později při běhu programu měnit. Její struktura je řešena jako jednosměrně lineárně vázaný seznam, do kterého se přidává vždy až na konec. To je důležité pro nalezení duplicitních položek. Každá položka této tabulky je typu `TVariable` a nese informaci o svém typu (neznámo, celé číslo, reálné číslo, řetězec znaků), obsahu (v závislosti na typu) a ukazatel na další prvek. Celou tabulku zastřešuje struktura `TConstantTable` obsahující pouze ukazatel na první prvek.

Tabulka proměnných je reprezentována velmi podobným způsobem. Základem je také jednosměrně lineárně vázaný seznam, do kterého se ovšem přidává na začátek, protože není potřeba hledat položky s duplicitním obsahem. Každý prvek je také typu `TVariable`. Tabulku zastřešuje podobná struktura `TVariableTable`. Obsah proměnných je naplněn a měněn za běhu programu. Na prvky tabulky proměnných se odkazuje symbol (typu proměnná).



Obrázek č. 1 Datové struktury

syntaktický analyzátor - `parser`

Syntaktický analyzátor tvoří jádro celého překladače. Zajišťuje přeložení zdrojového kódu v jazyce IFJ05 na posloupnost pseudoinstrukcí. Na základě úspěšnosti tohoto překladu se rozhodne, jestli spustí interpretaci, nebo jenom vypíše seznam chyb v zdrojovém kódu.

Na vstupu syntaktického analyzátoru je očekáván otevřený soubor obsahující zdrojový kód. Analyzátor si spustí vlastní instanci lexikálního analyzátoru a pomocí něj načítá *tokeny* ze zdrojového souboru. Ty potom zpracovává pomocí metod syntaktické analýzy a přímo vytváří pseudokód.

Jednotlivé příkazy jsou načítány pomocí konečného automatu. Pokud příkaz vyžaduje načtení výrazu (přiřazení, podmíněný skok), je volána funkce `ReadExpr()`. Pro zpracování výrazu je použita precedenční syntaktická analýza řízená precedenční tabulkou, která udává prioritu a asociativitu všech operátorů.

Součástí výrazu může být také volání funkce `Heapsort()`, jehož syntaxe není načítána

pomocí precedenční syntaktické analýzy. Za tímto účelem je volána speciální funkce `Readsort()`, která tuto syntaxi načte. Tato funkce musí načíst také jednotlivé parametry, což jsou opět výrazy. K tomu se rekurzivně volá funkce `ReadExpr()` a tím vzniká nepřímá rekurze.

Pseudoinstrukce jsou na výstupní pásku zapisovány přímo při aplikaci gramatických pravidel. Instrukční sada našeho pseudokódu je velmi jednoduchá a využívá zásobník. To znamená, že každá pseudoinstrukce má nejvýše jeden operand. Zásobník je užitečný zejména při vyhodnocování výrazů. Každé úspěšné volání funkce `ReadExpr()` vygeneruje sled instrukcí zajišťující, že na vrcholu zásobníku zůstane výsledná hodnota výrazu. Sémantická analýza není prováděna v době překladu, ale až při interpretaci.

Jako **rozšíření** jsme implementovali velmi jednoduché zotavení z chyb pomocí metody pevných klíčů. V našem případě se jedná o jeden pevný klíč a tím je znak `';'`. To znamená, že pokud je v jednom příkazu syntaktická chyba, nezabrání to detekci chyb ve zbývajícím zdrojovém kódu. Interpretace kódu se však nespustí, protože vygenerovaná posloupnost instrukcí může (a bude) vygenerována špatně.

Z hlediska implementace je syntaktická analýza rozdělena do dvou modulů. Modul `parser` zastřešuje celé provedení syntaktické analýzy a případné spuštění interpretace. Pro vyhodnocení výrazů (včetně načtení syntaxe volání funkce `Heapsort()`) je volána funkce `ExprRead()`, což je vstupní bod modulu `expr`.

interpretace a sémantická analýza - `runtime`

Interpretaci a sémantickou analýzu řeší soubor runtime. Po vytvoření vnitřního kódu je volána funkce `RuntimeProcess()` ze souboru runtime, což je proces, který provede sémantickou analýzu a pokud se nevyskytne žádná sémantická chyba dojde k interpretaci vnitřního kódu. Sémantická analýza je součástí jednotlivých instrukcí vnitřního kódu, provádí se až při vykonávání těchto instrukcí. Sémantická analýza provádí kontroly, které neprovádí syntaktická analýza a další akce. Jedná se zejména o přetypování proměnných u proměnných, které byly již dříve definovány, kontrolu inicializace proměnných při funkci **print**, kontrolu typů proměnných a jejich inicializace u binárních operací a jiných operací, u kterých je tohoto třeba a typování jejich výsledků, kontrolu korektnosti dat zadaných na standardním vstupu při funkci **read,** kontrolu deklarací navěští u skoků, nepoužití proměnných u skoků místo návěští atd.

V prvním parametru funkce RuntimeProces je této funkci předána páska. V souboru runtime je implementován jednoduchý zásobník, který uchovává v každé své položce ukazatel na data a typ dat. K zásobníku patří funkce pro jeho inicializaci, uložení na dat na vrchol zásobníku, přečtení a odstranění dat, zvětšení kapacity a nakonec pro zničení.

V hlavní funkci `RuntimeProcess()` je cyklus, který se provádí tak dlouho, dokud se neprovedou všechny instrukce na pásce nebo nenastane sémantická chyba. Výjimkou jsou instrukce `read` a `print` a instrukce nepodmíněného skoku, které se zásobníkem vůbec nepracují. Před vykonáním akce v instrukci se provádí sémantická analýza.

Řadící funkce – Heapsort

Princip řadící funkce je stejný jak pro čísla, tak pro řetězce. Místo relačních operátorů použije `strcoll()` a předtím pomocí `setlocale(LC_ALL, "cs_CZ")` nastavíme českou lokalizaci. Heapsort J. W. J. Williamse je řadící algoritmus, který má v nejhorším i v průměrném případě časovou složitost *O(N*lg N)*. V průměru je Heapsort asi dvakrát pomalejší než Quicksort, ale při jeho použití máme jistotu, že v nejhorším případě nedojde ke katastrofické degradaci

výkonu, jako je tomu u Quicksortu. Heapsort je nestabilní a nechová se přirozeně.

Heap (hromada) je struktura stromového typu, u níž pro všechny prvky platí, že mezi otcovským a všemi jeho synovskými uzly je stejná relace. Nejčastějším případem je binární heap, založený na binárním stromu.

Pro řazení pomocí Heapsortu budeme mít binární heap uložený v poli. Potom platí, že prvek v poli na pozici k má syny na pozici $2*k$ a $2*k+1$. Pokud je $2*k$ nebo $2*k+1$ větší než počet prvků má uzel buď jednoho následníka nebo ani jednoho. Předchůdce prvku na pozici k je vždy na pozici $k / 2$.

Postup přidávání prvků na konec. Pokud máme seřazený heap v poli a přidáme na konec jeden prvek a porovnáme-li jej s jeho předchůdcem a předchůdce bude menší, tak jsme skončili, ale je-li větší, tak je spolu vyměníme. Tak pokračujeme dokud není vkládaný prvek větší nebo není na začátku. V každém kroku se nám index zmenšuje alespoň dvakrát, a tak provedeme nejvýše *O(log N)* výměn.

Odebíráme-li minimum, dáme na jeho místo poslední prvek a porovnáváme jej s jeho následníky. Pokud je větší, jak jeden z nich, tak je vyměníme, pokud je větší jak oba, tak jej vyměníme s menším z nich. S každým krokem se nám index v poli zdvojnásobuje, a tak provedeme nejvýše *O(log N)* výměn.

V našem případě máme v poli *N* řetězců, které chceme seřadit. Nejdříve znovu ustanovíme heap pomocí funkce `sift()`. V tomto heapu bude v kořeni maximum. Funkce to provádí tak, že na jednom konci pole vzniká heap a na druhém je zbytek vstupního neseřazeného pole. Když je heap hotov začne se vykonávat druhá polovina algoritmu. Potupně se vybírá maximum a maže se z heapu. Vybrané maximum se vymění s posledním prvkem. Po výměně maxima se volá funkce `sift()`, které se předá velikost pole o jeden prvek menší a funkce znovu ustaví heap. Vybírání maxima se provádí tak dlouho, dokud z heapu nezůstane vzestupně seřazené pole.

Závěr

Interpret provádí vše podle zadání. Navíc je rozšířen o zpracování jednořádkového komentáře a jednoduchého zotavení z chyb. Program byl testován se všemi vzorovými zdrojovými kódy a všechny testy proběhly správně podle vzorových výstupů. Taktéž byl testován našimi testy, které byly navrženy pro otestování určitých vlastností interpretu. Výsledky testů prokázaly správnost našeho řešení interpretu.

Program přesně dodržuje požadavky kladené na formát vstupních a výstupních dat, takže může být bezproblémově používán spolu s dalšími programy ve skriptech nebo jiných programech.

Navržené řešení je bez problému přenositelné na všechny platformy, které používají alespoň 32 bitové registry. Program byl úspěšně otestován v prostředí operačních systémů Linux a MS Windows.

Metriky kódu

Počet souborů : 17
Počet řádků zdrojového textu: 4 976
Velikost spustitelného souboru: 37 kB (OS Linux 32-bit)