Západočeská univerzita v Plzni
Dokumentace překladače PL/0 v PHP
Předmět KIV/FJP

Petr Sládek
slady@slady.cz
27. ledna 2005

Zadání

Cílem této semestrální práce je kontrola zdrojového kódu překladače jazyka PL/0 napsaného v PHP a zároveň sepsání dokumentace pro ostatní studenty.

Záměrem autora je poskytnout budoucím studentům vhodný odrazový můstek pro jejich vlastní práci. Pokud dostanete jako semestrální práci na FJP zadání v PHP, stačí si přečíst tuto dokumentaci a už budete vědět „kam sáhnout“. Zbytek práce tedy bude hračka.

Úvod

Ke spuštění překladače napsaného v PHP potřebujete server nebo alespoň samostatný interpret, který umožňuje spouštění PHP skriptů. Jediný zádrhel může spočívat v načtení vstupního zdrojového textu – většina serverů totiž není nastavena jako server katedry, takže se do proměnné `$text` nemusí zdrojový text PL/0 vůbec načíst. Pro zabránění těchto problémů s nastavením serveru byla přidána jedna z prvních řádek v PHP, která do proměnné `$text` uloží hodnotu zaslanou vstupním textovým polem s názvem „`text`“:

```
$text = $_POST["text"];
```

Činnost samotného překladače není nijak zvlášť komplikovaná. Překladač se dělí na několik částí:

- Lexikálního analyzátor
- Syntaktický analyzátor
- Pomocné funkce, například:
    - Práce s množinovými operacemi
    - Práce s tabulkou proměnných
    - Práce s tabulkou generovaného kódu

Činnost překladače

Překladač pracuje v jediném průchodu, při kterém provádí veškeré kontroly, generuje kód a tabulku proměnných. Není to tedy tak, že se nejprve spustí lexikální analýza, po úspěšném provedení této fáze se spustí syntaktická analýza, atd. Vše funguje v jediném průchodu.

Nejprve se po spuštění kompilátoru inicializují konstanty a proměnné. Poté je spuštěn syntaktický analyzátor, který pro svůj běh potřebuje „tokeny“ lexikálního analyzátoru. Zavolá se tedy funkce lexikálního analyzátoru pro načtení jednoho „tokenu“, která zase spustí funkci pro načítání ze vstupu. Když jsou všechny „tokeny“ vstupního zdrojového textu přečteny a zpracovány, je spuštěno vygenerování cílového kódu.

Lexikální analyzátor

Lexikální analyzátor je první jednotka překladače, která má za úkol relativně jednoduchým způsobem získat ze vstupního zdrojového textu tzv. lexému (některý základní prvek příslušného jazyka, např. klíčové slovo „`if`“) a tu pak vrací syntaktickému analyzátoru v globálních proměnných.

Všechny možné lexémy jsou ve vstupním jazyce popsány pomocí regulárních výrazů. Kromě samotné lexémy se ukládají další související údaje, např. jakého druhu je daná lexéma (operátor, klíčové slovo, identifikátor, …) nebo pokud se jedná o číslo, pak také jeho číselná hodnota. Tato skupina údajů, předávaná z lexikálního do syntaktického analyzátoru, se označuje anglickým termínem *token* (česky: známka, odznak).

Ve zdrojovém kódu je lexikální analýza implementována v jediné funkci: `getsym()`

Syntaktický analyzátor

Syntaktický analyzátor (v angličtině označovaný jako *parser*, z *to parse* – rozebrat, oddělovat, udělat větný rozbor) se dá považovat za mozek překladače, protože provádí samotnou analýzu vstupního jazyka. Úkolem syntaktického analyzátoru je rozpoznat, zda je program zapsán správným způsobem, např. zda na úvodní „`begin`“ bude navazovat někde koncové „`end`“, ale také určuje, v jakém pořadí se budou provádět jednotlivé části příkazů, například u zápisu „`x+y*z`“ rozpozná a určí, že nejdříve se bude násobit a pak až sčítat.

Úloha syntaktického analyzátoru je složitější než úloha lexikálního analyzátoru, který vyhodnocoval regulární jazyk. Syntaktický analyzátor musí rozpoznávat složitější bezkontextový jazyk, musí tedy vyhodnocovat vstup v delším záběru.

Používá se jeden z nejjednodušších způsobů konstrukce syntaktického analyzátoru – tzv. metoda rekurzivního sestupu. V této metodě se pomocí rekurzivních volání funkcí konstruuje tabulka výstupního kódu, přičemž struktura implementace překladače odráží strukturu příslušné gramatiky. Pokud je v gramatice na daném místě neterminální symbol, v překladači mu odpovídá (rekurzivní) volání funkce reprezentující příslušný neterminál. Tato rekurzivní volání končí u terminálních symbolů (lexémů jazyka).

Následuje zápis syntaxe jazyka PL/0 tak, jak odpovídá voláním názvů funkcí:

```
program    -> block .

block      -> [ const identifikátor = číslo {, identifikátor = číslo } ; ]
              [ var identifikátor { , identifikátor } ; ]
              [ procedure identifikátor ; block ; ]
              statement

statement  -> identifikátor := expression |
              call identifikátor |
              begin statement { ; statement } end |
              if condition then statement |
              while condition do statement |
              e

condition  -> odd expression |
              expression ( = | <> | < | > | <= | >= ) expression

expression -> [ + | - ] term { ( + | - ) term }

term       -> factor { ( * | / ) factor }

factor     -> identifikátor |
              číslo |
              "(" expression ")"
```

Vysvětlivky:

- **Tučně zapsané** symboly jsou neterminální symboly gramatiky
- *Symboly zapsané kurzívou* jsou složitější terminální symboly (tedy čísla a názvy identifikátorů)
- Obyčejným písmem jsou zapsané jednoduché terminální symboly
- [Výraz] v hranatých závorkách je nepovinný
- {Výraz} ve složených závorkách se může několikrát opakovat (tzv. iterace)
- Znak | je symbol pro „nebo“
- Kulaté závorky oddělují části výrazu
- Kulaté závorky v uvozovkách jsou terminální symboly reprezentující kulaté závorky

Zvláštnosti programu – množinové operace

Zajímavost oproti implementaci v jazyku Pascal je absence množinových funkcí v PHP. Jedná se vlastně o bitové pole logických hodnot, se kterým je možné provádět operace jako například sčítání. V této implementaci a v implementaci v jazyku C je tato vlastnost implementována normálním polem logických hodnot.

Potíží, které tato náhražka množin přináší, je několik. Nejde jednoduše pojmenovávat jednotlivé prvky, takže jsou nadefinovány konstanty indexů, kterými se přistupuje do pole. Dále je také velmi těžké sloučit dvě logická pole, takže je tu k tomu jednoduchá funkce pojmenovaná `sjednot()`, která k poli v prvním formálním parametru přidruží prvky z pole v druhém formálním parametru (což je vlastně sjednocení dvou logických polí).

Jediná změna, kterou jsem v implementaci provedl, byla nahrazení vytváření takového logického pokaždé znovu. Namísto nulování pole je možné volat funkci `nuluj()`, která vrátí pole plné nul o délce třicet prvků.

Popis globálních proměnných a funkcí

Následuje popis nejdůležitějších globálních proměnných, datových struktur a všech funkcí programu.

Globální proměnné

`$wsym` … tabulka symbolů (klíčových slov)

`$ssym` … tabulka matematických operátorů (+, -, /, <)

`$text` ... zdrojový kód (jeden dlouhý řetězec s kódem a oddělovači řádků)

`$pozice` ... index (ukazatel) do proměnné `$text` – pozice ve zdrojáku

`$ch` ... jeden znak načtený ze zdrojáku – plněno funkcí `getch()`

`$id` ... jeden „token“ načtený ze zdrojáku – plněno funkcí `getsym()`

`$num` ... hodnota číselného „tokenu“ – plněno funkcí `getsym()`

`$sym` ... číselná identifikace načteného „tokenu“ (z tabulky `$wsym`)

`$cx` ... index (ukazatel) na vygenerovaný kód do tabulky `$code`

`$code` ... pole vygenerovaného kódu

Datové struktury

Tabulka proměnných `$TABLE` má položky o tomto formátu:

```
class PrvekTabulky:
      $name      ... jméno proměnné z globální prom. $id
      $kind      ... typ položky: Kconstant, Kvariable či Kprocedure
      $val       ... číselná hodnota z globální proměnné $num
      $level     ... tzv. „level“ – zanoření procedury
      $adr       ... adresa proměnné či procedury
      $size      ... délka procedury
```

Při každém přístupu k proměnné či při volání procedury se prohledává tato tabulka a podle jména (uloženého v položce `$name`) se určí index do tabulky. Pak se může načíst adresa, ze které se má číst hodnota proměnné, nebo adresa začátku procedury, na kterou se má skákat.

Tabulka `$code`, do které se generuje výstupní kód, má položky s tímto formátem:

`class INSTRUCTION:`

- `$f` … kód instrukce
- `$l` … tzv. „level“ – úroveň operandu zanoření procedury
- `$a` … adresa, na kterou se má skočit, nebo kód operace

Funkce

Následují všechny funkce v logickém seskupení, tedy nikoli v takovém pořadí, v jakém jsou implementované v programu.

`function` **`error`**`($n)`

Vypíše chybu, která se objevila v programu.

`function` **`getch`**`()`

Přečte jeden znak ze zdrojového textu. Uloží ho do `$ch`

Množinové funkce

`function` **`sjednot`**`(&$s1, $s2)`

Náhrada množinové operace sjednocení. Nastaví příznaky v `$s1`, které jsou nastaveny v `$s2`

`function` **`test`**`($s1,$s2,$n)`

Jedná se o test syntaktického analyzátoru. Testuje, zda je v `$sym` uloženo číslo „tokenu“, který je povolen v logické množině `$s1`

`function` **`nuluj`**`()`

Vrátí nově vytvořené čisté pole logických hodnot (pole „nepravda“).

Práce s tabulkou proměnných

`function` **`enter`**`($k, &$tx, $lev, &$dx)`

Vloží jeden prvek (proměnnou nebo proceduru) z globálních proměnných (`$id`, `$num`) a lokálních (`$k`, `$lev`, `$dx`) do tabulky `$TABLE` tvořenou objekty `PrvekTabulky` na pozici `$tx`

`function` **`position`**`($id, $tx)`

Vyhledá symbol podle jména v tabulce symbolů $TABLE, hledá od pozice `$tx` dolů do nuly

Generování kódu

`function` **`gen`**`($x, $y, $z)`

Vloží jeden prvek o hodnotách (`$x`, `$y`, `$z`) do pole vygenerovaného kódu `$code` na pozici `$cx`

`function` **`listcode`**`()`

Generuje assembler na výstupu.

Vypisuje všechny položky tabulky `$code` až do hodnoty indexu `($cx-1)`

Lexikální analyzátor

`function` **`getsym`**`()`

Načte jeden symbol (identifikátor, klíčové slovo, číslo) ze zdrojového textu. Uloží ho do `$id`
Pokud se jedná o identifikátor nebo klíčové slovo, uloží do `$sym` číslo (kód z tabulky `$wsym`), nebo když se jedná o číslo, uloží jej do proměnné `$num`

Syntaktický analyzátor

`function` **`constdeclaration`**`(&$tx, $lev, &$dx)`

Zpracovává definici konstanty typu „`const size = 8;`“
Používá funkci `enter()` pro uložení hodnot.

`function` **`vardeclaration`**`(&$tx,$lev,&$dx)`

Zpracovává definici proměnné typu „`var i := 0;`“
Používá funkci `enter()` pro uložení hodnot.

`function` **`factor`**`($fsys, $tx, $lev)`

Součást výrazu (**`expression`**), viz syntaktický analyzátor dříve.

`function` **`term`**`($fsys, $tx, $lev)`

Součást výrazu (**`expression`**), viz syntaktický analyzátor dříve. Volá `factor()`

Použito pro matematické operace násobení, dělení a modulo.

`function` **`expression`**`($fsys, $tx, $lev)`

Výraz (**`expression`**), viz syntaktický analyzátor dříve. Volá `term()`
Použito pro matematické operace sčítání a odčítání.

`function` **`condition`**`($fsys, $tx, $lev)`

Vyhodnocení logických výrazů, viz syntaktický analyzátor dříve. Volá `expression()`

`function` **`statement`**`($fsys, $tx, $lev)`

Vyhodnocení výrazů, viz syntaktický analyzátor dříve.

`function` **`block`**`($lev, $tx, $fsys)`

Vyhodnocení bloků příkazů, viz syntaktický analyzátor dříve.

Závěr

Pokud máte konkrétní zadání na rozšíření či změnu funkčnosti překladače jazyka PL/0 v PHP a přečetli jste si tuto dokumentaci, měli byste vědět, kde se dá místo pro požadovanou změnu rychle najít.

Mimo sepsání této dokumentace pro vaší snazší práci je snad můj jediný viditelný přínos do zdrojového kódu počítání čísel řádek pro oživení výpisu chybových hlášení.

Zdroje

[1] Domácí stránky KIV/FJP o PL/0, http://www.kiv.zcu.cz/~lobaz/fjp/fjp4.html

[2] Článek Wikipedie o překladačích, http://cs.wikipedia.org/wiki/Překladač

[3] Implementace PL/0 v dalších jazycích, http://www.kiv.zcu.cz/~lobaz/fjp/fjp.html