FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Formální jazyky a překladače PL0 překladač pro online interpret

KIV/FJP

Tomáš Linhart, Pavel Třeštík
Doplň své číslo, A22N0137P

21. ledna 2023

1 Úvod

Cílem této práce bude implementovat překladač z „vysoko úrovňového jazyka“ PL/0 do příslušných instrukcí. Jazyk PL/0 je založen nad syntaxí podobající se jazyku Pascal. Avšak PL/0 je značně omezen v poskytujících funkcí a je velmi zjednodušen, aby bylo snadné ho použít pro výuku formálních jazyků a překladačů.

K jazyku existují také vlastní instrukce, kterými lze popsat jednoduché výpočetní zařízení. Instrukce fungují na principu zásobníkového automatu, tedy všechny instrukce přidávají a ubírají data na zásobníku. Například součet dvou čísel funguje tak, že se na zásobník nejdříve vloží 2 čísla a pak se zavolá instrukce, která vyjme 2 hodnoty na vrcholu zásobníku a zpět vloží jejich sumu. Instrukce mají vždy 2 parametry, které ale nemusí mít pro instrukci význam.

V předchozích letech tohoto předmětu byl vytvořen online interpret takovýchto instrukcí. V rámci předmětu byly také vytvořeny překladače pro tento jazyk, avšak tyto překladače jsou buď zastaralé a nebo nejdou spustit online. Cílem této práce je vytvořit překladač, který bude provozován online a nevyžaduje běžící backendovou aplikaci. Ze zadání jsou na překladač kladeny požadavky na vlastnoti, které musí překládaný jazyk (a tedy i překladač) podporovat. Tyto požadavky nejsou částí původní gramatiky jazyka PL/0 a proto je třeba ho rozšířit. Zároveň je ale chtěné zachovat zpětnou kompatibilitu s jazykem PL/0, což může omezovat náš nový jazyk, který PL/0 rozšiřuje.

2 Analýza

Překladač má být online aplikace, která nesmí mít backendovou část, která by běžela na serveru. Z tohoto důvodu byl pro implementaci překladače navrhnut jazyk JavaScript, protože to je populární jazyk pro implementaci webových aplikací, a je možné v něm naprogramovat plně „client-side“ překladač.

Funkce jazyka, které jazyk musí podporovat jsou následující:

- definice celočíselných proměnných
- definice celočíselných konstant
- základní aritmetiku a logiku (+, -, *, /, AND, OR, negace a závorky, operátory pro porovnání čísel)

- cyklus (libovolný)
- jednoduchou podmínku (if bez else)
- definice podprogramu (procedura, funkce, metoda) a jeho volání

Funkce jazyka, které nejsou požadovaným minimem, ale byly zvoleny pro dosažení bodů nutných ke splnění práce:

- každý další typ cyklu (for, do .. while, while .. do, repeat .. unitl, foreach pro pole)
- else větev
- datový typ boolean a logické operace s ním
- datový typ real (s celočíselnými instrukcemi)
- podmíněné přiřazení / ternární operátor (min = (a < b) ? a : b;)
- paralelní přiřazení (a, b, c, d = 1, 2, 3, 4;)
- parametry předávané hodnotou
- návratová hodnota podprogramu

Překlače je možno implementovat různými způsoby. Mezi hlavní 2, které se učí v tomto předmětu, patří implementace pomocí zásobníkového automatu a nebo pomocí takzvaného rekurzivního sestupu. Implementace pomocí zásobníkového automatu může být jednodušší, protože existují nástroje, pro které stačí definovat tokeny jazyka a jeho gramatiku a tyto nástroje následně vygenerují automat, který jazyk překladá. Do automatu se akorát doplní logika generující instrukce. Na druhou stranu rekurzivní sestup je více v rukou programátora. Výhodou rekurzivního sestupu tedy je, že programátor si může překladač více řídit sám. Nevýhodou je, že implementace je časově dělší, náročnější na práci a také může být složitější optimalizovat výsledné instrukce, než to je u zásobníkového automatu.

2.1 Gramatika jazyka

Jak již bylo zmíněno, gramatika jazyka této práce výchází z PL/0 a musí na PL/0 zachovat zpětnou kompatibilitu. PL/0 má následující gramatiku (v EBNF [1]):

```
program = block "." ;

block = [ "const" ident "=" number
          {"," ident "=" number} ";"]
        [ "var" ident {"," ident} ";"]
        { "procedure" ident ";" block ";" } statement ;

statement = [ ident ":=" expression
              | "call" ident
              | "?" ident
              | "!" expression
              | "begin" statement {";" statement } "end"
              | "if" condition "then" statement
              | "while" condition "do" statement ];

condition = "odd" expression |
        expression ("="|"#"|"<"|"<="|">"|">=") expression ;

expression = [ "+"|"-"] term { ("+"|"-") term};

term = factor {("*"|"/") factor};

factor = ident | number | "(" expression ")";
```

Listing 1: Originální PL/0 gramatika

Tato gramatika téměř splňuje minimální zadání práce. Chybí v ní akorát logické operátory. Gramatiku tedy stačí rozšířit vybrané funkce. Tyto funkce byly záměrně vybrány tak, aby byly do gramatiky snadno dodělatelné. Návrh výsledné gramatiky vypadá následovně:

```
program = block "." ;
```

[1] Extended Backus-Naur form - `https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_form`

```
block = [ "const" ident [":" data_type] "="
          value {"," ident [":" data_type] "=" value} ";"]
        [ "var" ident [":" data_type]
          {"," ident [":" data_type]} ";"]
        { "procedure" ident [ "(" ident [ : data_type ]
          {"," ident [ : data_type ]} ")" ] ";" block ";" }
          statement ;

statement = [ ident ":=" {ident ":="} expression
                | "{" ident {, ident} "} :=
                  {" value{, value} "}"
                | "call" ident
                | "?" ident
                | "!" expression
                | "begin" statement {";" statement } "end"
                | "if" condition "then" statement
                  [ "else" statement ]
                | "(" condition ") ? " "return" statement
                  ":" "return" statement
                | "while" condition "do" statement
                | "for" number "to" number "do" statement
                | "return" value; ]

condition = "odd" expression |
            expression ("="|"#"|"<"|"<="|">"|">=") expression ;

expression = [ "+"|"-"] term { ("+"|"-") term};

term = factor {("*"|"/") factor};

factor = ident | number | value | "(" expression ")";
```

Listing 2: Navržená gramatika rozšiřující PL/0

3 Realizace

Jak bylo navrženo v analýze, tak překladač byl implementován v jazyce JavaScript. Pro implementaci překladače byl zvolen rekurzivní sestup. Pro parsování kódu jazyka na tokeny byl použit nástroj **jison-lex** [2], což je JavaScript alternativa nástroje **lex**, který je učen v tomto předmětu.

Při realizaci bylo zjištěno několik chyb v gramatice, a navrženou gramatiku bylo třeba ještě doupravit, aby bylo možné implementovat vybrané vlastnoti jazyka. Finální verze gramatiky vypadá následovně:

```
program = block "." ;

block = [ "const" ident [":" data_type] "="
          value {"," ident [":" data_type] "=" value} ";"]
        [ "var" ident [":" data_type]
          {"," ident [":" data_type]} ";"]
        { "procedure" [data_type] ident [ "(" ident
          [ : data_type ] {"," ident [ : data_type ]} ")" ]
          ";" block ";" } statement ;

statement = [ ident ":=" expression
                | "{" ident {, ident} "} :=
                  {" value{, value} "}"
                | "call" ident
                | "?" ident
                | "!" expression
                | "begin" statement {";" statement } "end"
                | "if" condition_expression
                  "then" statement
                  [ "else" statement ]
                | "(" condition_expression ") ?
                  " expression ":" expression
                | "while" condition_expression
                  "do" statement
                | "for" expression "to" expression
                  "do" statement
                | "return" expression; ]
```

[2] jison-lex - `https://github.com/zaach/jison-lex`

```
condition_expression = ["~"] condition
                       { ("&"|"|") ["~"] condition }

condition = "odd" expression |
            expression ("="|"#"|"<"|"<="|">"|">=")
            expression ;

expression = ["+"|"-"] term {("+"|"-") term} | "call" ident;

term = ["~"] factor { ("*"|"/"|"&"|"|") ["~"] factor};

factor = ident | value | "(" expression ")";
```

Listing 3: Finální gramatika

3.1 Rekurzivní sestup

Jison-lex vytváří skript, který parsuje vstup a podle specifikovaných regexů vrací tokeny. Rekurzivní sestup implementuje 2 funkce, které z lexu podávají tokeny. První je **next_sym()**, která načte další symbol (token) z lexu. Druhá je **bool accept(symbol)**, která zavolá **next_sym()** a poté porovná, zda-li načtený symbol je rovný parametru. Tímto lze ověřit, že je načten správný token, když je očekáván. Pokud **accept()** vrátí false, tak byl načten neočekávaný vstup a překladač se může rozhodnout, jak pokračovat - ačkoliv v této práci vždy vede k chybě překladu.

Rekurzivní sestup je potom implementován po vzoru gramatiky. Vstupním bodem překladače je funkce **program()**, která po vzoru gramatiky volá funkci **block()** a následně **accept(Symbols.dot)**. Tedy všechny neterminální symboly gramatiky jsou funkcemi v implementaci a na všechny terminální symboly je volána funkce **accept()** v těchto funkcích. Tyto funkce jsou nakonec rozšířeny o logiku, která jim umožňuje generovat výsledné instrukce. Příkladem takovéto funkce je například pravidlo `statement = "while" condition_expression "do"statement`:

```
statement_while: function() {
  // while verified by caller
  this.accept(Symbols.while);
```

```
    if (!this.condition_expression()) {
      this.error("Failed to compile while condition.");
      return false;
    }

    if (!this.accept(Symbols.do)) {
      this.error("Expected 'do' before while statement.");
      return false;
    }

    if (!this.statement()) {
      this.error("Failed to compile while statement.");
      return false;
    }

    return true;
  }
```

Listing 4: statement_while příklad bez logiky

Po rozšíření o logiku pro generování instrukcí, které plní funkci `while` pak vypadá následovně:

```
  statement_while: function() {
    // while verified by caller
    this.accept(Symbols.while);

    let while_start_addr = instruction_list.length;

    if (!this.condition_expression()) {
      this.error("Failed to compile while condition.");
      return false;
    }

    push_instruction(Instructions.JMC, 0, 0);
    let while_end = instruction_list[instruction_list.length - 1];

    if (!this.accept(Symbols.do)) {
      this.error("Expected 'do' before while statement.");
```

```
      return false;
    }

    if (!this.statement()) {
      this.error("Failed to compile while statement.");
      return false;
    }

    push_instruction(Instructions.JMP, 0, while_start_addr);
    while_end.par2 = instruction_list.length;

    return true;
  }
```

Listing 5: statement_while příklad s logikou

Samozřejmě překladač drží více logiky, než pouze jednoduché generování instrukcí jako v příkladu `while`. Některé funkce reprezentující gramatiku souvisí s jinými a je třeba nějak tyto souvislosti udržovat, aby se daly využívat všude, kde jsou potřeba. Příkladem tohoto je tabulka symbolů[3]. Ta udržuje informace o všech identifikátorech v programu - tedy názvech proměnných, konstant či funkcí. V této práci jsou tyto informace také udržovány, ale nejsou v jednotné tabulce. Práce rozděluje tabulka hlavně na 2 části. První je list identifikátorů procedur a informací o proceduře (jako například návratový typ, pokud nějáký má). Druhou je „matice“ idetifikátorů proměnných, do které jsou vkládany řádky, každého kontextu, který je zrovna překládán a tedy které proměnné jsou zrovna dostupné. Sestup také udržuje několik „flag“ proměnných, které slouží ke správné funkci překladače.

3.2 Grafické rozhraní

4 Řešené problémy

Řešeno byla dlouhá řada problémů, různých složitostí. Například volání procedur je realizováno tak, že je vložena instrukce **CAL 0 par**, kde **par** je index v listu kontextů. Index odpovídá instrukci, na který je nalezena instrukce **JMP 0 x**, která skočá na tělo procedury. Toho je docíleno tak, že po

[3]Tabulka symbolů - `https://en.wikipedia.org/wiki/Symbol_table`

vygenerování všech instrukcí jsou na začátek seznamu instrukcí vloženy instrukce **JMP** tak, aby odpovídaly indexům listu kontextů. Následkem tohoto vložení je potom nutné všechny skokové instrukce posunout o počet vložených instrukcí. Problém byl takto řešen, protože takto byly realizovány volání v příkladech na cvičení předmětu. Možná lepším řešením by ale mohlo být do instrukcí **CAL** rovnou doplňovat instrukci začínající tělo procedury, protože tato informace je udržována. Ušetřilo by se tím vkládání na začátek listu a iterování celého listu, ovšem za cenu přehlednosti. Takto je zřejmé, že je volána procedura a navíc díky indexům, který odpovídají pořadí překládání procedur, lze odhadnou, která procedura je volána.

Dalším příkladem problému je rozdílné pořadí operací online interpretu, pro který je překladač zaměřen. Překladač byl implementován s použitím „manuálu“[4], ovšem během testování bylo zjištěno, že několik operací neodpovídá manuálu. Z interpretu poté bylo zjištěno, že pro některé operace používá jiné čísla. Tento problém není problém implementační, ale stále způsobil značnou ztrátu času.

Mezi opravdu těžké problémy by pak patřila například implementace `for` cyklu. `for` cyklus vezme výsledky dvou výrazů a z nich vypočte počet iterací, a poté iteruje do 0. Alespoň takto je navržen `for` cyklus této práce. Implementovat ho ale bylo těžké, protože PL/0 instrukce nemají žádný způsob pro kopírování hodnot na vrcholu zásobníku či zjištění ukazatele na vrcho zásobníku. Kvůli tomuto nešlo cyklus implementovat bez pomocné proměnné. Řešením tedy je, že pro kontext se vytváří navíc proměnné, pokud obsahuje `for` cyklus. Navíc bylo třeba ošetřit možnost noření cyklů a je tedy třeba počítat kolik pomocných proměnných pro `for` je vlastně potřeba.

5 Testování

To ti rád přenechám

6 Závěr

To bych ti klidně taky nechal :D

[4] `https://usermanual.wiki/Pdf/PL020Users20Manual.15518986/help`