FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Formální jazyky a překladače PL/0 překladač pro online interpret

KIV/FJP

Tomáš Linhart, Pavel Třeštík
A22N0055P, 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 vlastnosti, 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 .. until, 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

Překladače 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 překládá. 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 delší, 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 vychází 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 dodělané. 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 do upravit, aby bylo možné implementovat vybrané vlastnosti 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
                  ["(" expression {"," expression } ")"]
                | "?" 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
```

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

```
                | "return" expression; ]

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 nějaký má). Druhou je „matice“ identifikátorů proměnných, do které jsou vkládaný 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

Grafické rozhraní je realizováno ve webovém prohlížeči za využití frameworků Bootstrap, Jquery a editoru kódu Monaco. Pro uživatele je přístupná pouze jedna webová stránka *index.html*, která obsahuje kompletní editor.

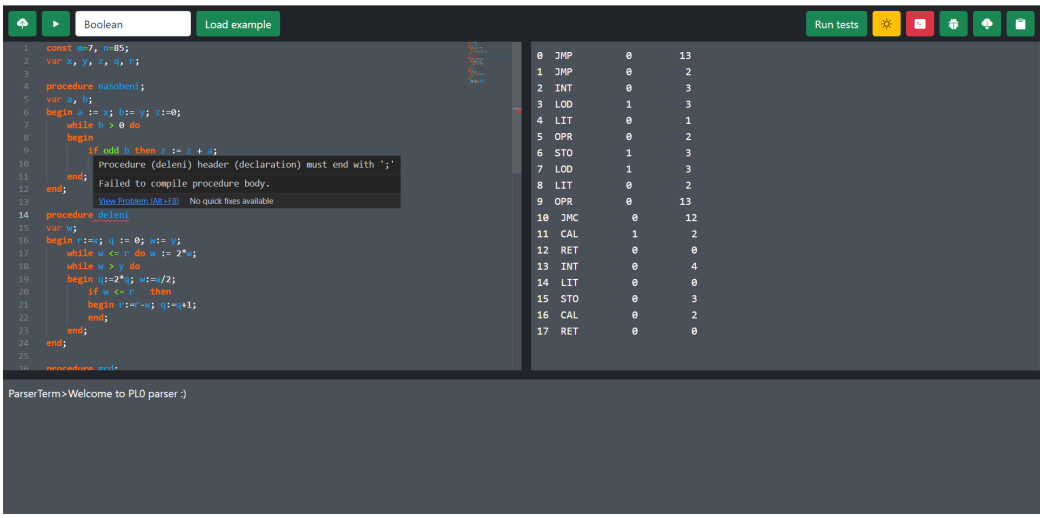
Editor je realizován podobně jako aktuální moderní IDE. Obrazovka obsahuje horní lištu s ovládacími tlačítky, levý vstupní editor pro zápis kódu a

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

pravý výstupní editor, ve kterém jsou obsaženy výstupní instrukce po kompilaci programu.

Vstupní editor je realizován s využitím volně dostupné knihovny Monaco editor. [4] Tato knihovna umožňuje vytvoření vlastního editoru s možností definice syntaxe vlastního programovacího jazyka a stylů. Pro tuto práci byla vytvořena zjednodušená verze syntaxe jazyka, která se využívá pro obarvení kódu a zobrazení dialogů s nápovědou programátorovi. Dále knihovna umožňuje pomocí funkce vkládat podtržení chyb a zobrazuje dialogové okno s detailním popisem chyby, pokud uživatel nad zvýrazněnou část kódu uživatel umístí kurzor myši.

Užitečnou funkčnosti uživatelského rozhraní je automatická průběžná kompilace během psaní vstupního kódu. Výstupní kód a zprávy v konzoli nejsou během psaní nijak ovlivněny, ale uživatel již průběžně vidí podtrhané chybné úseky kódu s informací, co způsobuje jednotlivé chyby. Ve výchozím nastavení probíhá tato kontrola vždy 1000 milisekund po ukončení zadávání vstupu, ale tuto hodnotu je možné změnit pomocí konfiguračního souboru.



Obrázek 1: Ukázka grafického rozhraní s chybovým dialogem a vygenerovanými instrukcemi

[4] `https://microsoft.github.io/monaco-editor/`

3.2.1 Ovládací prvky

Veškeré ovládací prvky jsou zobrazeny na horní ovládací liště, která je dále rozdělena na pravou a levou část. Nejčastěji používané ovládací prvky jsou umístěny v levé části. V následující části dokumentace jsou jednotlivé prvky zleva vyjmenovány a je popsána jejich funkčnost.

3.2.1.1 Nahrání souboru - ikona mráčku s šipkou nahoru

Toto tlačítko slouží pro načtení vstupního souboru z počítače uživatele. Povoleny jsou pouze soubory ve formátu *.txt*. Před tím než uživatel vybere soubor je upozorněn, že tato akce smaže jeho aktuální kód. Po potvrzení vybere soubor a ten je načten do vstupního okna.

3.2.1.2 Spuštění překladu - ikona přehrání

Stisknutím tohoto tlačítka uživatel spustí překlad kódu z vstupního editoru. V případě, že je kód validní jsou do výstupního okna vloženy přeložené instrukce. Pokud ovšem kód obsahuje nějakou chybu, je tato chyba vypsána do konzole ve spodní části obrazovky a zároveň je kód způsobující chybu podtržen. Obsah ve výstupním okně zůstává nezměň, může obsahovat instrukce z předchozích korektních běhů.

3.2.1.3 Načtení příkladů - selectbox a tlačítko Load example

Pro uživatele bylo připraveno několik příkladů, které demonstrují funkčnosti vytvořeného programovacího jazyka. Ty lze načíst do vstupního okna kliknutím na selectbox a vybráním příslušného příkladu. Po stisknutí tlačítka Load example je bez zobrazení varování kód ve vstupním editoru nahrazený daným příkladem. Všechny příklady jsou funkční a obsahují například ukázku aritmetiky, podmínek a procedur.

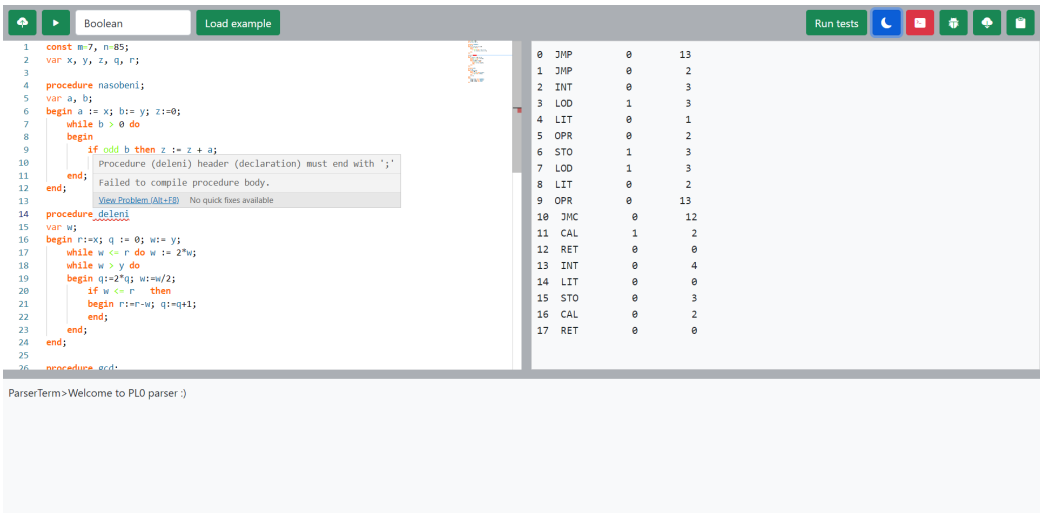
3.2.1.4 Spuštění testů - tlačítko Run tests

Po stisknutí tlačítka je spuštěno automatické testování. Testovací případy jsou pomocí asynchronního volání načteny ze serveru a následně spuštěny. Uživateli je pak zobrazeno okno s výsledky testování, viz 5 Testování.

3.2.1.5 Přepnutí režimu - tlačítko sluníčko nebo měsíček

Tímto tlačítkem se přepíná mezi světlým a tmavým režimem editoru. Editor je ve výchozím stavu spuštěn v tmavém režimu, po stisknutí na tlačítko

sluníčka je ikona a barva tohoto tlačítka změněna a je nahrazeno tmavé schéma editoru schématem světlým. Opětovným stisknutím stejného tlačítka se editor přepne opět na tmavé schéma.



Obrázek 2: Ukázka grafického rozhraní ve světlém režimu

3.2.1.6 Skrytí a zobrazení konzole - ikona terminálu

Tlačítko slouží pro skrytí okna konzole. Po jejím skrytí je editor roztažen na celou výšku okna (kromě horní lišty). Výstupy z aplikace jsou do konzole stále vypisovány, po jejím zobrazení stisknutím stejného tlačítka uživatel vidí veškeré aktuální výpisy.

3.2.1.7 Zobrazení debuggeru - ikona brouka

Napojení na debugger lze spustit stisknutím tlačítka pro debug. Po stisknutí je výstupní kód automaticky odeslán do napojeného debuggeru a debugování je spuštěno. Okna editorů jsou nahrazeny připraveným debuggerem, pro opětovné otevření editorů je nutné stisknout stejné tlačítko. Okno spuštěného debuggeru nelze poté vyvolat zpět, lze ho pouze načíst od začátku stejným tlačítkem. Pokud je napojení na debugger nefunkční, je zobrazena chybová zpráva do konzole a uživateli zůstávají k dispozici okna editoru. Dále viz 3.2.2 Napojení na debugger

3.2.1.8 Stažení kódu - ikona mráčku s šipkou dolů

Soubor obsahující výsledné instrukce lze stáhnout pomocí předposledního tlačítka na ovládací liště. Soubor s názvem *outputCode.txt* obsahuje jednoduchý seznam výsledných instrukcí podobně jako ve výstupním okně editoru.

3.2.1.9 Zkopírování kódu - ikona psací podložky

Poslední ovládací tlačítko je alternativou ke stažení souboru. Výsledné instrukce jsou po stisknutí nakopírovány do schránky uživatele namísto stažení.

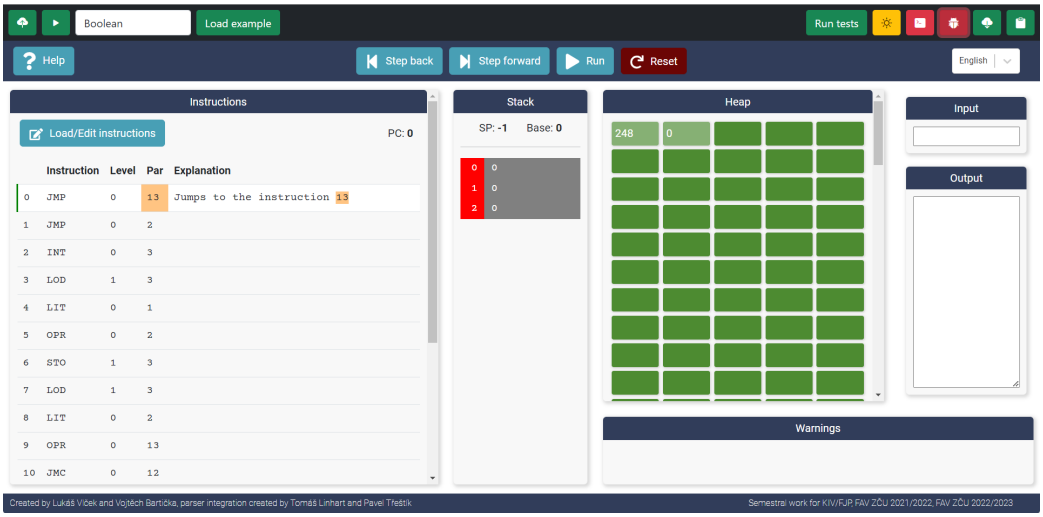
3.2.2 Napojení na debugger

Jako debugger je využita práce z minulých let pana Vlčka a Bartičky, pro informace pro spuštění a ovládání viz dokumentace anebo Github [5]. Pro integraci bylo nutné definovat jednoduchý protokol, kterým budou přenášeny informace mezi překladačem a debuggerem. Z tohoto důvodu bylo nutné debugger upravit tak, aby dokázal přečíst data z datové sběrnice a spustit jejich krokování. Debuggger je v překladači zobrazený jako Iframe na většině obrazovky, pro komunikaci je využito Message eventů [6], které umožňují předávání dat mezi IFrame a rodičovským oknem a naopak.

Při inicializaci očekává překladač zprávu *READY*, která indikuje, že je debugger dostupný. Debugger tuto zprávu odešle po načtení a bez přijetí této zprávy není možné integraci na debugger využívat. Poté lze odesílat do debugger instrukce pomocí zprávy s identifikátorem *debug*. Tato zpráva dále obsahuje instrukce, které debugger zpracuje a v případě, že je vše v pořádku a debugování je spuštěno, vrátí překladači zprávu *DEBUGGER_START*. Na tu zareaguje překladač a zobrazí okno s debuggerem uživateli. V případě, že dojde k chybě překladače a zašle do debuggeru instrukce, které nedokáže přečíst, odešle debugger překladači zprávu *COMPILATION_ERROR*, ve které se dále nachází jaké konkrétní instrukce způsobily chybu.

[5] `https://github.com/lukasvlc3k/fjp/tree/main/pl0`

[6] `https://developer.mozilla.org/en-US/docs/Web/API/Window/message_event`



Obrázek 3: Ukázka grafického rozhraní ve světlém režimu

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 skočí na tělo procedury. Toho je docíleno tak, že po 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“[7], ovšem během testování bylo zjištěno, že několik operací neod-

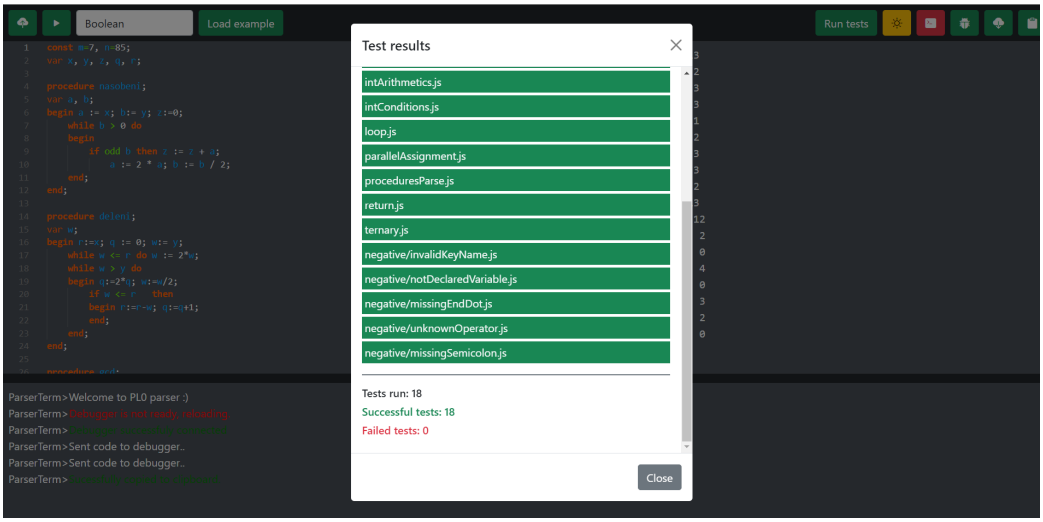
[7] `https://usermanual.wiki/Pdf/PL020Users20Manual.15518986/help`

povídá 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 vrchol 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

Testování je spouštěno přímo z uživatelského rozhraní pomocí tlačítka, viz 3.2.1.4 Spuštění testů - tlačítko Run tests. Jednotlivé testy soubory s testy jsou uloženy v souboru *tests.json*. Očekává se, že jeden testovací soubor odpovídá jednomu testovacímu případu. Po načtení souboru s testy pomocí AJAX volání jsou načteny jednotlivé testovací případy. Každý testovací případ vytvoří metodu *runTestCase*, která představuje vstupní bod daného testu. Framework pro testování tuto metodu spustí, pokud dojde k chybě v testu, je to indikováno vyhozením výjimky. Testovací framework tuto výjimku zpracuje a data o s informacemi o chybě testu jsou zobrazeny uživateli v modálním okně.



Obrázek 4: Ukázka modálního okna s výsledky testů

Pro práci bylo připraveno více jak 15 testovacích případů, které testují správnost lexeru a správnost vygenerovaných instrukcí, nebo správnost zobrazených chybových zpráv (negativní testy).

6 Závěr

Výsledkem práce je funkční webový překladač námi definovaného jazyka, který překládá vstupní kód na výstupní instrukce jazyka PL/0. Překladač splňuje všechny základní body zadání, zároveň ale umožňuje zpracování pokročilých, bonusových instrukcí. Pro používání veškerých funkcí překladače je připraveno webové rozhraní s využitím primárně jazyka HTML a JavaScript. Správná funkčnost překladače je ověřena pomocí několika testů. Překladač je napojený na mírně upravený debugger vytvořený studenty z předchozích let a umožňuje výstupní instrukce automaticky spouštět a ladit ve stejném okně.