FAKULTA
APLIKOVANÝCH VĚD
ZÁPADOČESKÉ
UNIVERZITY
V PLZNI

Dokumentace k semestrální práci z KIV/FJP

Překladač jazyka Pascal0Like

Student: Martin Kružej, Jakub Šmaus
St. číslo: A17N0079P, A17N0089P
E-mail: kruzej@students.zcu.cz, smaus@students.zcu.cz

Datum: 25. prosince 2017

Obsah

1 Zadání

1.1 Tvorba překladače zvoleného jazyka

Cílem práce bude vytvoření překladače zvoleného jazyka. Je možné inspirovat se jazykem PL/0, vybrat si podmnožinu nějakého existujícího jazyka nebo si navrhnout jazyk zcela vlastní. Dále je také potřeba zvolit si pro jakou architekturu bude jazyk překládán (doporučeny jsou instrukce PL/0, ale je možné zvolit jakoukoliv instrukční sadu pro kterou budete mít interpret).

Jazyk musí mít minimálně následující konstrukce:

- definice celočíselných proměnných
- definice celočíselných konstant
- přiřazení
- 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

Překladač který bude umět tyto základní věci bude hodnocen deseti body. Další body (alespoň do minimálních 20) je možné získat na základě rozšíření, jsou rozděleny do dvou skupin, jednodušší za jeden bod a složitější za dva až tři body. Další rozšíření je možno doplnit po konzultaci, s ohodnocením podle odhadnuté náročnosti.

Jednoduchá rozšíření (1 bod):

- 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)
- datový typ string (s operátory pro spojování řětezců)

- rozvětvená podmínka (switch, case)
- násobné přiřazení (a = b = c = d = 3;)
- podmíněné přiřazení / ternární operátor (min = (a < b) ? a : b;)
- paralelní přiřazení (a, b, c, d = 1, 2, 3, 4;)
- příkazy pro vstup a výstup (read, write - potřebuje vhodné instrukce které bude možné využít)

složitější rozšíření (2 body):

- příkaz GOTO (pozor na vzdálené skoky)
- datový typ ratio (s celočíselnými instrukcemi)
- složený datový typ (Record)
- pole a práce s jeho prvky
- operátor pro porovnání řetězců
- parametry předávané hodnotou
- návratová hodnota podprogramu
- objekty bez polymorfismu
- anonymní vnitřní funkce (lambda výrazy)

Rozšíření vyžadující složitější instrukční sadu než má PL/0 (3 body):

- dynamicky přiřazovaná paměť - práce s ukazateli
- parametry předávané odkazem
- objektové konstrukce s polymorfním chováním
- instanceof operátor
- anonymní vnitřní funkce (lambda výrazy) které lze předat jako parametr
- mechanismus zpracování výjimek

Vlastní interpret (řádkový, bez rozhraní, složitý alespoň jako rozšířená PL/0) je za 6 bodů.

1.2 Bodované náležitosti

Kromě toho že by program měl fungovat se zohledňují i další věci, které mohou pozitivně nebo negativně ovlivnit bodování:

- testování - tvorba rozumné automatické testovací sady +3 body (pro inspiraci hledejte test suit pro LLVM nebo se podívejte na Plum Hall testy, ale jde skutečně jen o inspiraci, stačí výrazně jednodušší řešení). Užitečné a stručné povídání na dané téma najdete také tady.
- Kvalita dokumentace -x bodů až +2 body podle kvality a prohřešků (vynechaná gramatika, nesrozumitelné věty, příliš chyb a překlepů, bitmapové obrázky pro diagramy s kompresními artefakty, ...).
- Vedení projektu v GITu -x bodů až +2 body podle důslednosti a struktury příspěvků.
- Kvalita zdrojového textu -x bodů až +2 body podle obecně známých pravidel ze ZSWI, PPA a podobně (magická čísla, struktura programu a dekompozice problému, božské třídy a metody, ...)

2 Analýza

Překladač je obecně program, který převádí zdrojový kód do nějakého cílového kódu. Zadání nespecifikuje, které tyto kódy použít a dává nám volnost použít i vlastní. Cílový kód je interpretován tzv. interpretem. Součástí zadání je i jeho implementace ohodnocená body, proto bude v této práci také vytvořen.

2.1 Analýza překladače

Základem překladače je tedy určit, jaký zdrojový kód bude překládán. Definicí tohoto kódu je gramatika. Jde o souhrn pravidel nejčastěji ve formátu EBNF(*Extended Backus–Naur form*), která jednoznačně určují, jestli zadaný kód (či jazyk) patří či nepatří do našeho překládaného kódu.

Další částí překladače je lexikální analyzátor. Ten projde zdrojový kód a nahradí jeho jednotlivé části tzv. tokény. Výstup tohoto analyzátoru je předán syntaktickému analyzátoru, jenž kontroluje za pomocí definované gramatiky strukturu programu. Výstupem je často derivační strom. V této chvíli je možné zdrojový program dále zpracovat, např. kontrola typů, proměnných apod. dle definovaných pravidel vlastního programovacího jazyka. Poslední částí překladače je samotný překlad do instrukcí cílové platformy. Nyní přichází na řadu program, jenž cílový kód bude interpretovat.

Pro všechny tyto komponenty bude nutné vytvořit vlastní řešení s výjimkou lexikálního/syntaktického analyzátoru, kde je možnost použít některé existující nástroje jako YACC a ANTLR. Oba dva dělají ve své podstatě to samé, jen YACC vyžaduje implementaci překladače v programovacím jazyce C, zatímco ANTLR v Javě. Jde tedy o osobní preferenci, který z těchto nástrojů použít.

Náš tým se rozhodl pro ANTLR a tedy pro Javu. Překládaný jazyk bude postaven na vyučovací gramatice PL/0, doplněn o některé vlastnosti jazyka Pascal. Samotný PL/0 je jazyk velice jednoduchý a pouhou jeho implementací by zadání nebylo splněno na požadovaný počet bodů. Původní návrh také počítá s tím, že překladač bude překládat do instrukcí PL/0 tak, aby mohl být interpretován libovolným interpretem PL/0.

Zadání vyžaduje vedení práce na githubu, práci je tedy možno najít na adrese https://github.com/SmausJakub/heine-fluch.

2.2 Analýza jazyka

Náš nový vlastní jazyk nese pracovní název `Pascal0Like`, protože vychází z PL/0 a Pascalu.

Z PL0 si vezmeme tyto vlastnosti:

- Program je tvořen blokem a končí tečkou
- Blok je rozdělen na deklarační část a příkazovou část
- Deklarovat je možné konstanty a proměnné
- Příkazy jsou odděleny středníkem
- Proměnné jsou typu číslo
- Je možné deklarovat procedury, které obsahují blok
- Základní příkazy: přiřazení, podmínka if, cyklus while-do, příkaz volání procedury a strukturovaný příkaz begin-end
- Podpora výrazů: srovnání výrazů, test lichosti, násobení, dělení, modulo, součet, odčítání, unární mínus, číselná konstanta, závorkování výrazů

Tato základní gramatika bude za pomoci Pascalu udělána složitější a bude podporovat další věci:

- Deklarovat je možné návěští a paralelní deklarace proměnných
- Podporované typy jsou navíc: reálné číslo, logická hodnota, řetězec
- Přidání dalších cyklů: do-while, repeat-until, for
- Přidání else k podmínce if
- Další příkazy: skok goto, ternární přiřazení, I/O příkaz, case větvení
- Přidání výrazů: logická negace, logické AND a OR, konstanty reálného, logického typu a řetězce

3 Implementace

Vzhledem k použití nástroje ANTLR se implementace dělí do tří částí - samotný nástroj ANTLR pro lexikální a syntaktickou analýzu, náš překladač pro syntézu vstupního programu a převod do instrukcí PL/0, a nakonec interpret pro interpretaci těchto instrukcí.

3.1 ANTLR

Nástroj ANTLR poslouží v projektu pro lexikální a syntaktickou analýzu. Jako vstup vyžaduje gramatiku a jeho výstupem bude derivační strom. Pro úplnost dodejme, že derivační strom vytváří metodou zdola nahoru.

3.1.1 Gramatika

ANTLR má vlastní formát pro definici pravidel gramatiky, jež se ukládají do souboru s koncovkou `.g4`. Tento soubor lze najít v kořenovém adresáři programu pod jménem `Pascal0Like.g4`. Tato pravidla mají syntax podobnou EBNF, podívejme se na ukázku:

```
assignment_statement
    :
    IDENT ASSIGN expression SEMI
    ;
```

Odpovídající gramatika ve formátu EBNF by vypadala třeba takto:

```
assignment_statement = ident ":=" expression ";" ;
```

Pravidlo `ident` odpovídá definici identifikátoru a pravidlo `expression` definici výrazu.

Popis gramatiky implementován dle analýzy je v souboru `Pascal0Like EBNF Gramatika.pdf`. Gramatika v tomto souboru odpovídá té gramatice z `Pascal0Like.g4`, jen je vedena ve formátu EBNF pro lepší čitelnost.

Základní minimální gramatika programu PascaL0-like vypadá takto:

```
program(test);

use strict;

begin
end.
```

Na začátku je nutné deklarovat jméno programu. Následuje deklarace programového módu, o tom více v 3.2.4. Nyní je zde prostor pro deklarace, které ale nejsou povinné. Následuje povinná příakzová část ohraničená klíčovými slovy *begin* a *end*. Poté musí následovat tečka indikující konec programu.

3.1.2 Derivační strom

ANTLR vytvoří z definované gramatiky tokény (lexikální analýza). Následně vygeneruje derivační strom a třídy, jež obsahují metody pro práci s tímto stromem. Nabízí dvě možnosti, jak strom procházet - přes `Listenerey` nebo `Visitory`. `Listener` umožňuje pracovat s pravidlem při vstupu a výstupu z pravidla. `Visitor` nabízí jednu metodu na pravidlo, ale umožňuje vracet libovolnou třídu jako návratovou hodnotu. Pro použití těchto dvou metod stačí vytvořit novou třídu, která je zdědí a přepíše jejich metody pro práci s pravidly.

3.2 Překladač

Překladač má nyní za úkol projít derivační strom vygenerovaný ANTLR, provést syntézu a vygenerovat instrukce. Po zralé úvaze je vybrána možnost přes `Visitory`, jež nabízí možnost vytvořit si vlastní třídy. Tím otvírají možnost pro větší kontrolu a kvalitnější syntézu programu. Veškeré třídy překladače se nacházejí v balíku `compiler`.

3.2.1 Programové třídy

Předtím, než je možné se zabývat `visitory` je nutné podívat se na programové třídy. Tyto třídy jsou navrženy tak, aby reprezentovaly vstupní program. Svým způsobem se podobají derivačnímu stromu, jen neobsahují "zbytečnosti"navíc, jako symboly v pravidlech. Nacházejí se v balíku `types`.

Některé typy jsou rozděleny do dalších balíků pro lepší čitelnost a strukturu. Typy jsou následující:

- balík `atoms` - Atomy programu pro jednotlivé typy, tedy:
  - **AtomBoolean** - logická konstanta `true` nebo `false`
  - **AtomReal** - reálná konstanta
  - **AtomInteger** - číselná konstanta
  - **AtomId** - identifikátor

- balík `declarations` - jednotlivé deklarace programu, obsahuje:
  - **DeclarationConstant** - deklarace konstanty
  - **DeclarationLabel** - deklarace návěští
  - **DeklarationProcedure** - deklarace procedury
  - **DeclarationVariableParallel** - paralelní deklarace proměnných
  - **DeclarationVariableSimple** - deklarace proměnných
- balík `expressions` - jednotlivé podporované výrazy programu, jde o:
  - **ExpressionAdditive** - výraz sčítání
  - **ExpressionAtom** - atomický výraz - obsahuje Atom
  - **ExpressionLogic** - výraz logického AND nebo OR
  - **ExpressionMultiplication** - výraz násobení
  - **ExpressionNot** - logická negace
  - **ExpressionOdd** - test lichosti
  - **ExpressionPar** - závorkovaný výraz
  - **ExpressionRelational** - výraz srovnání
  - **ExpressionUnary** - unární mínus
- balík `statements` - jednotlivé příkazy programu, jde o:
  - **StatementAssignment** - příkaz přiřazení
  - **StatementCase** - příkaz case větvení
  - **StatementCompound** - strukturovaný příkaz
  - **StatementDoWhile** - příkaz do-while cyklu
  - **StatementFor** - příkaz for cyklu
  - **StatementGoto** - příkaz skoku
  - **StatementIf** - příkaz if podmínky
  - **StatementIO** - příkaz I/O
  - **StatementProcedure** - příkaz volání procedury
  - **StatementRepeat** - příkaz repeat-until cyklu
  - **StatementTernary** - příkaz ternárního operátoru
  - **StatementWhileDo** - příkaz while-do cyklu

- Block - blok programu a procedur
- CaseLimb - větev case
- Constant - konstanta
- Goto - uchovává informace o skoku
- Label - návěští
- Procedure - procedura
- Program - program
- Variable - proměnná

Tyto třídy uchovávají informace o jednotlivých částech programu a dohromady tvoří zpracovávaný program.

3.2.2 Visitory

`Visitory` mají za úkol projít ANTLR derivační strom a vytvořit naši vlastní reprezentaci programu z výše popsaných tříd. Všechny třídy `visitorů` se nacházejí v balíku `visitors`.

Každý `visitor` dědí ANTLR třídu `PascaL0LikeBaseVisitor`. Tato třída je typovaná a tento typ je právě návratová hodnota, kterou bude `visitor` vracet. Tím je nyní možnost vytvořit si třídy `visitorů` odpovídající různým typům, které jsme si popsali výše. Pro to, aby to fungovalo tak, jak požadujeme, je ještě třeba typy obalit do jednoho unikátního typu, abychom nemuseli mít pro každé pravidlo jeden `visitor`. Proto byly vytvořeny abstraktní třídy AbstractAtom, AbstractDeclaration, AbstractExpression a AbstractStatement. Nacházejí se v balíku `abstracts`.

Jednotlivé `visitory` jsou:

- **VisitorAtom** - vrací jednotlivé druhy atomů jako AbstractAtom
- **VisitorBlock** - vrací třídu Blok
- **VisitorCase** - vrací case větev (třída CaseLimb)
- **VisitorDeclaration** - vrací jednotlivé druhy deklarací jako AbstractDeclaration
- **VisitorExpression** - vrací jednotlivé druhy výrazů jako AbstractExpression

- **VisitorProgram** - vrací třídu Program
- **VisitorProgramMode** - vrací výčtový typ ProgramMode
- **VisitorStatement** - vrací jednotlivé druhy příkazů jako AbstractStatement

ANTLR `visitor` poskytuje metody pro jednotlivá pravidla. Tyto metody jsou v určitých `visitorech` přepsány a následně volány. Takto se projde celý derivační strom a vytvoří se programové třídy, jež mu odpovídají.

3.2.3 Datové typy

Před kapitolou o programovém módu je nejdříve nutné ukázat, jaké datové typy překladač podporuje. Jde o tři typy:

- **celé číslo** - datový typ celého čísla. V programu se značí klíčovým slovem *integer* nebo *var* v závislosti na módu (viz 3.2.4).
- **reálné číslo** - datový typ čísla s plovoucí čárkou. V programu se značí klíčovým slovem *real*.
- **logická hodnota** - datový typ logické hodnoty. V programu se značí klíčovým slovem *boolean*. Logické konstanty jsou *true* a *false*.

Datové typy jsou reprezentovány v programu výčtovým typem VariableType.

3.2.4 Programový mód

Před samotným popisem `compilerů` je třeba si ještě osvětlit programový mód. Překladač umí pracovat ve třech různých módech - *legacy*, *default* a *strict*. Tento mód je možné deklarovat před deklaracemi proměnných (viz 3.1.1). Pokud není deklarován mód, automaticky je použit *default*. Jednotlivé módy se od sebe liší v kompilační části. Popíšeme si je:

- **legacy mód** - tento mód slouží pro kompatibilitu s PL/0. V tomto módu je jediný povolený datový typ *var*, reprezentující celé číslo, a jsou zakázány logické výrazy AND, OR a negace. Tento mód umožňuje vzít libovolný programový kód základní PL/0 a bez problémů ho přeložit naším překladačem. Datový typ *var* je v ostatních módech zakázán.
- **strict mód** - tento mód je striktní v tom smyslu, že nepovoluje míchání typů. Není možné sečíst reálné číslo a celé číslo například. Přetypování není v tomto módu možné.

- **default mód** - základní mód překladače. Přetypování je možné a vždy se řídí podle levého výrazu. Například při sečtení reálného čísla a celého čísla (*integer x := 3.14 + 1*) bude pravý výraz převeden na reálné číslo, poté se provede operace a až poté se výsledek převede na celé číslo a uloží do proměnné *x*. Při obrácení výrazů (*integer x := 1 + 3.14*) se pravý výraz převede na celé číslo, provede se operace a výsledek se uloží do proměnné *x*.

3.2.5 Tabulka symbolů

Tabulka symbolů je struktura, ve které se uchovávají proměnné a informace o nich. V programu reprezentováno ve třídách SymbolTable a SymbolTableItem v balíku *symbol*. Jednotlivý záznam v tabulce symbolů (tedy symbol) obsahuje:

- **name** - název proměnné (identifikátor). Unikátní pro záznam.
- **level** - úroveň, ve které byla proměnná deklarována
- **type** - typ proměnné (konstanta, proměnná, procedura, návěští)
- **address** - v závislosti na typu symbolu má různý význam:
  - **proměnná** - adresa na zásobníku
  - **návěští** - začíná na 0, po použití v programu obsahuje číslo instrukce, kam je třeba skočit
  - **procedura** - číslo instrukce, kde procedura začíná
- **size** - v závislosti na typu symbolu má různý význam:
  - **proměnná** - jestli byla inicializována (1) nebo ne (0)
  - **návěští** - jestli bylo použito (1) nebo ne (0)
  - **procedura** - velikost procedury (počet instrukcí)
- **variableType** - pouze proměnné, určuje jejich datový typ

Tabulka symbolů je staticky přístupná celému programu, ale používají ji pouze `compilery`.

3.2.6 Compilery

`Compilery` mají za úkol projít vygenerované programové třídy, provést syntézu a vytvořit odpovídající instrukce. Nacházejí se v balíku `compilers`. Existují různé třídy `compilerů` v závislosti na tom, co kompilují. Všechny třídy pracují se třídou CompilerData, která obsahuje globální proměnné a metody, které jsou využívány všemi třídami.

Jednotliví `compilery` budou nyní popsány.

CompilerProgram

První volaný kompilátor. Pracuje s třídou Program, která obsahuje:

- **block** - hlavní blok programu
- **identifier** - název programu
- **programMode** - programový mód programu

Tento kompilátor uloží programový mód do CompilerData a předá hlavní programový blok CompilerBlock. Jeho poslední úlohou je zkontrolování seznamu Goto, ale to popíšeme až později u kompilace příkazu skoku.

CompilerBlock

Druhý volaný kompilátor. Pracuje s třídou Block, která obsahuje:

- **declarationList** - seznam deklarací
- **statementList** - seznam příkazů

Úkolem kompilátoru je předat deklaračním a příkazovým kompilátorům jednotlivé deklarace a příkazy. Jenže zde nastávají komplikace.

Ukládání proměnných na zásobník vyžaduje zvýšení vrcholu zásobníku o počet deklarací. Kompilátor tedy počítá, kolik deklarací se provedlo, a zásobník zvýší o požadovaný počet.

Výraznější problém nastává při deklaraci procedury. Ta totiž obsahuje svůj vlastní blok s dalšími deklaracemi a příkazy. Kvůli tomu je nutné nulovat používané proměnné - počet deklarací a stávající adresu. Tím se zamezí chybám v deklaracích proměnných procedury. Kompilátor poté přidá instrukci skoku, aby přeskočil tělo procedury, jelikož procedura se smí provádět až tehdy, kdy je zavolána a první kód, který je čten je hlavní příkazový blok programu. Kompilátor kompiluje proceduru ihned, ale skokem, kterým

proceduru přeskočíme, zajistíme, že interpret bude interpretovat instrukce v pořadí takovém, jaké chceme.

Posledním problémem může být deklarace proměnných po proceduře obzvláště pokud se v proceduře tyto proměnné používají. Proto kompilátor deklarace řadí podle jejich typu tak, že deklarace procedur jsou vždy poslední.

CompilerAtom

Kompilátor nejnižší úrovně jazyka - atomů. Pracuje v závislosti na typech atomu:

- **AtomInteger** - celočíselná konstant
- **AtomReal** - reálná konstanta
- **AtomBoolean** - logická konstanta
- **AtomId** - identifikátor

V závislosti na typu je použita určitá instrukce. V případě identifikátor musí kompilátor načíst požadovaný záznam z tabulky symbolů a dle datového typu postupuje obdobně, jako s ostatními atom typy.

Tento kompilátor nikdy nic nekontroluje, tedy při reálné konstantě nekontroluje, že je program kompilován v *non-legacy módu*, či nedívá se, jestli identifikátor v tabulce symbolů existuje. Toto řeší kompilátory o úroveň výš předtím, než předají pokyn ke zpracování atomu.

CompilerDeclaration

Tento kompilátor zpracovává deklarace v závislosti na jejich typu.

Deklarace návěští - pracuje s třídou DeclarationLabel, která obsahuje seznam objektů Label. Kompilátor pro každý z nich vytvoří záznam v tabulce symbolů

Deklarace procedury - pracuje s třídou DeclarationProcedure, ta obsahuje:

- **identifier** - identifikátor procedury
- **procedureBlock** - blok procedury

Kompilátor vytvoří záznam pro novou proceduru a předá řízení CompilerBlock, který řeší náležitosti týkající se nového bloku. Kompilátor jen zvyšuje stávající úroveň, která se tímto zvětší. Po návratu z bloku je úroveň opět snížena.

Deklarace proměnné - pracuje s třídou DeclarationVariableSimple, která obsahuje:

- **variableList** - seznam deklarovaných proměnných
- **expression** - přiřazovaný výraz
- **init** - logická hodnota, jestli jsou proměnné inicializovány uživatelem
- **type** - datový typ proměnných

Kompilátor postupuje po jedné proměnné, nejdříve vytvoří záznam v tabulce symbolů. Z důvodu optimalizace provede jen a pouze napoprvé kompilaci výrazu. Pokud proměnné nebyly inicializovány uživatelem, kompilátor je inicializuje na implicitní hodnoty.

Deklarace paralelních proměnných - pracuje s třídou DeclarationVariableParallel, která obsahuje:

- **variableList** - seznam deklarovaných proměnných
- **expressionList** - seznam přiřazovaných výrazů
- **type** - datový typ proměnných

Kompilátor postupuje po jedné proměnné, vytvoří záznam v tabulce symbolů a poté kompiluje daný výraz. V každé iteraci cyklu kompiluje výraz, jelikož je tento výraz jiný pro každou proměnnou. Počet proměnných a výrazů musí souhlasit, jinak je vyhozena chyba.

Deklarace konstanty - pracuje s třídou DeclarationConstant, která obsahuje:

- **constantList** - list konstantních proměnných
- **value** - atomická přiřazovaná hodnota
- **type** - datový typ proměnných

Kompilátor postupuje stejně jako při deklaraci proměnných, jen nekompiluje výraz, ale atom.

CompilerExpression

Tento kompilátor zpracovává výrazy v závislosti na jejich typu. I přesto, že v programu rozeznáváme různé druhy výrazů, dají se generalizovat na tři typy:

- **Obalovací výrazy** - tyto výrazy obsahují další výraz, který obalují. Jde o ExpressionOdd, ExpressionUnary, ExpressionPar, ExpressionNot.
- **Atomický výraz** - tento výraz neobsahuje žádné další výrazy, ale pouze atomickou hodnotu. Jde o ExpressionAtom.
- **Operativní výrazy** - tyto výrazy obsahují vždy dva další výrazy, levou a pravou stranu, a operátor. Jde o ExpressionMultiplication, ExpressionAdditive, ExpressionRelational, ExpressionLogic.

Kompilace výrazů probíhá rekurzivně, kde návratová hodnota je datový typ. Každý výraz vždy zkontroluje datový typ, se kterým pracuje, v závislosti na svém typu a programovém módu potom rozhoduje, zda je možné s tímto typem pracovat. Operativní výrazy srovnávají oba typy svých dvou výrazů a jejich možné kombinace.

Jakousi výjimku tvoří ExpressionPar a ExpressionAtom, které žádné typy nekontrolují. ExpressionPar vrací typ výrazu, který obaluje, a ExpressionAtom vrací datový typ odpovídající typu svého atomu.

CompilerStatement

Tento kompilátor kompiluje příkazy v závislosti na jejich typu. Ještě předtím se podívá, jestli příkaz neobsahuje návěští. V případě, že ano, aktualizuje daný záznam tabulky symbolů a řízení předá dále.

Příkaz přiřazení - pracuje se třídou StatementAssignment, jež obsahuje:

- **identifier** - identifikátor proměnné
- **expression** - přiřazovaný výraz

Kompilátor musí zkontrolovat několik věcí:

- Identifikátor musí mít záznam v tabulce symbolů
- Tento záznam musí být přístupný dle úrovní (tedy úroveň záznamu musí být menší nebo rovna stávající úrovni)
- Záznam nesmí být typu konstanty či procedury

Příkaz volání procedury - pracuje se třídou StatementProcedure, jež obsahuje **identifier** - identifikátor procedury. Kompilátor zkontroluje, že je možné tuto proceduru zavolat a pak jí zavolá.

Příkaz skoku - pracuje se třídou StatementGoto, jež obsahuje **value** - číslo návěští, kam se má skočit. Nyní je třeba zkontrolovat, že návěští existuje v tabulce symbolů (tedy bylo deklarováno) a že je přístupné dle úrovní. Následně mohou nastat dvě věcí dle atributu *size*:

- **size je 1** - to znamená, že víme kam skočit, takže to není problém a vytvoříme instrukci skoku na adresu návěští
- **size je 0** - v této chvíli nevíme, kam skočit, jelikož návěští nebylo v programu ještě použito. Instrukce skoku se stejně připravíme s dočasnou nulovou hodnotou jako adresou. Následně se do seznamu objektů *Goto* uloží nový záznam. Tento záznam obsahuje vytvořenou instrukci, název návěští a stávající úroveň. Právě tento seznam později projíždí CompilerProgram. Podívá se na adresu záznamu a pokud je různá od nuly (tedy návěští bylo někde použito a adresa je číslo instrukce skoku), aktualizuje instrukci skoku objektu Goto poté, co zkontroluje úrovně, jinak vyhazuje chybu, protože skok nelze provést.

Příkaz I/O - pracuje se třídou StatementIO, která obsahuje:

- **identifier** - identifikátor proměnné
- **type** - I/o typ - vstupní či výstupní

Kompilátor zkontroluje dostupnost záznamu daného identifikátor a poté již jen doplní instrukce dle I/O typu.

Příkaz strukturovaný - pracuje se třídou StatementCompound, která obsahuje seznam příkazů. Tento kompilátor sám o sobě nic nedělá, vzhledem k tomu, že strukturovaný příkaz je obalovací příkaz dalších příkazů. Kompilátor tedy jen zavolá další kompilátory na jednotlivé příkazy.

Příkaz ternární - pracuje se třídou StatementTernary, jež obsahuje:

- **expression** - podmínka
- **expressionOne** - první výraz
- **expressionTwo** - druhý výraz

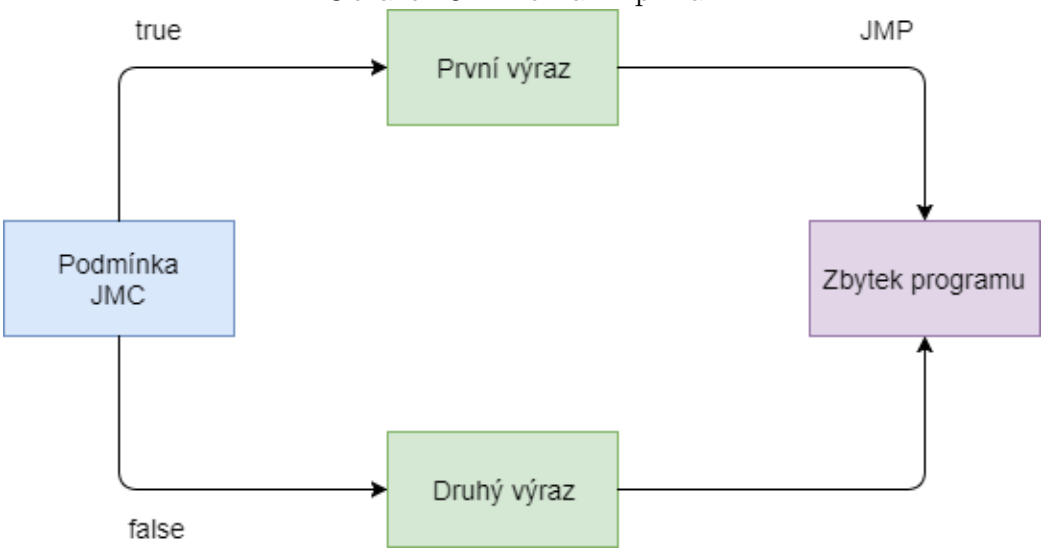
- **identifier** - identifikátor proměnné

Ternární operátor je kombinací podmíněného výrazu a přiřazovacího. Pokud platí podmínka, přiřadíme první výraz, pokud ne, přiřadíme druhý. Instrukční sada je vysvětlena později v dokumentu, ale nyní nám budou stačit dvě instrukce - instrukce skoku JMP a instrukce skoku při 0 JMC.

Kompilátor nejdříve kompiluje podmínku. Poté připraví skok JMC. Pokud byla podmínka vyhodnocena jako pravda, program dále pokračuje do prvního výrazu. Pokud byla vyhodnocena jako nepravda, program skáče na druhý výraz. Nyní je třeba si dávat pozor, jelikož bez dalších úprav by po dokončení prvního výrazu začal program vykonávat druhý výraz, proto je nutné na konec prvního výrazu doplnit instrukci skoku JMP a vyhnout se tak druhému výrazu. Na konci druhého výrazu není třeba nic měnit.

Ternární příkaz je ukázán na obrázku 3.1.

Obrázek 3.1: Ternární příkaz



Příkaz if - pracuje se třídou Statementif, která obsahuje:

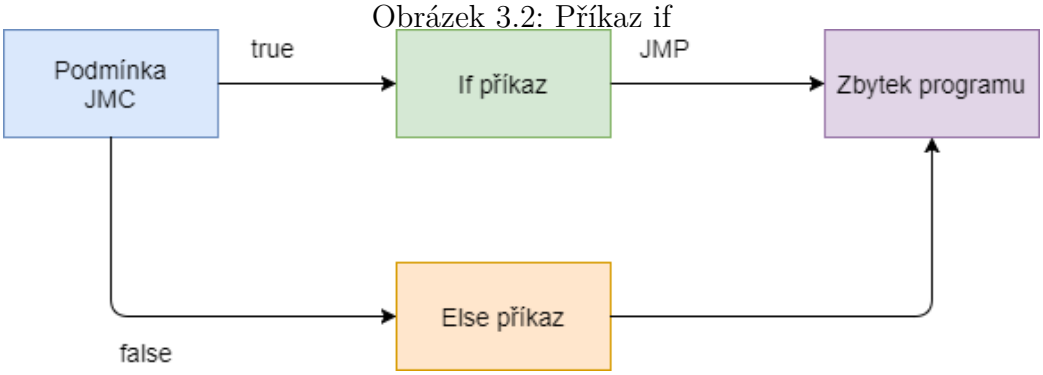
- **condition** - podmínka
- **statement** - příkaz if
- **elseStatement** příkaz else

Příkaz if vyhodnocuje podmínku, při pravdě vykoná příkaz if, jinak přeskočí na příkaz else, pokud existuje, jinak do zbytku programu. Implementace spočívá ve skoku JMC, kterým buď vykonáme příkaz if nebo skočíme

do příkazu else. Pak jen stačí zajistit přidání skoku na konec if příkazu pro přeskočení else příkazu.

Příkaz if je ukázán na obrázku 3.2.

Obrázek 3.2: Příkaz if



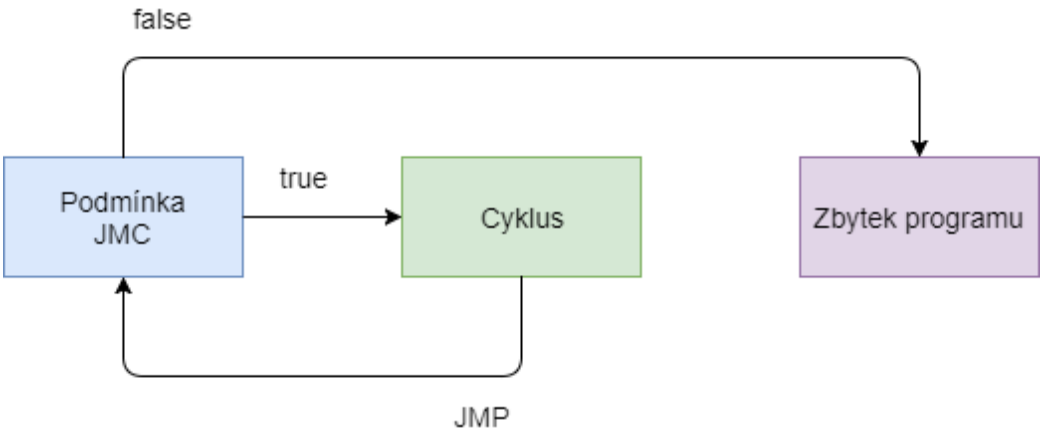
Příkaz while-do - pracuje s třídou StatementWhileDo, která obsahuje:

- **condition** - podmínka
- **statement** - příkaz cyklus

Tento cyklus testuje podmínku na začátku cyklu. Pokud tato podmínka je pravdivá, provede cyklus. Pokud není, provede skok mimo cyklus do zbytku programu. Toto kompilujeme pomocí instrukce JMC. Na konec cyklu nyní stačí přidat instrukci skoku JMP, kterou se dostaneme zpátky na začátek k testování podmínky.

Cyklus while-do je zobrazen na obrázku 3.3.

Obrázek 3.3: Cyklus while-do



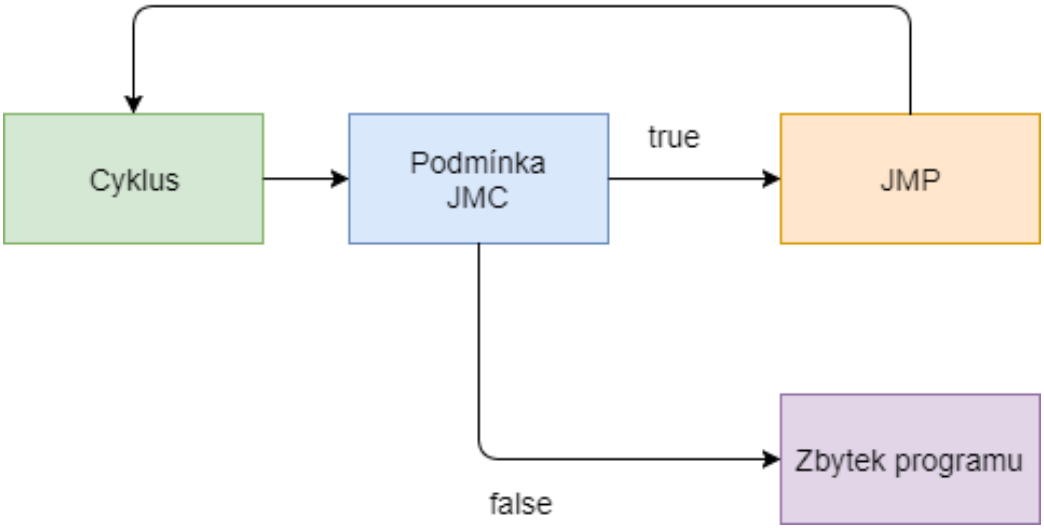
Příkaz do-while - pracuje s třídou StatementDoWhile, která obsahuje:

- **condition** - podmínka
- **statement** - příkaz cyklu

Tento cyklus testuje podmínku na konci cyklu. Nejdříve vykoná cyklus. Poté vyhodnotí podmínku. Pokud je pravdivá, potřebujeme skočit zpátky a pokud nepravdivá, tak pryč. Dosáhneme toho tak, že připravíme instrukci JMC a za ní instrukci skoku JMP. Skok JMP bude skákat vždy na začátek cyklu, tím dosáhneme toho, že při vyhodnocení podmínky pravdivě skočíme zpět. Nyní jen stačí nastavit JMC ven z cyklu, kam při nepravdě skočí do zbytku programu.

Cyklus do-while je zobrazen na obrázku 3.4

Obrázek 3.4: Cyklus do-while



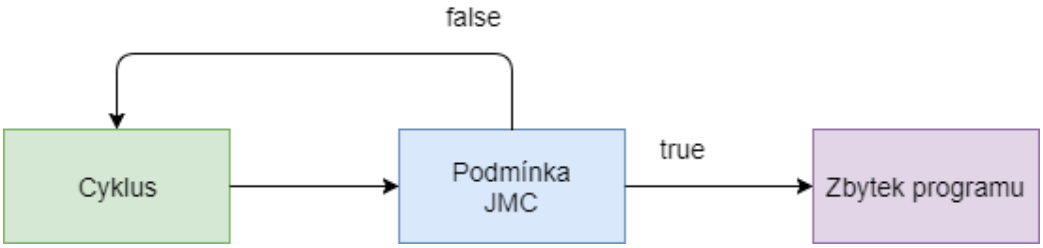
Příkaz repeat-until - pracuje s třídou StatementRepeat, která obsahuje:

- **condition** - podmínka
- **statement** - příkaz cyklu

Tento cyklus testuje podmínku na konci cyklu. Co ho odlišuje od cyklu do-while je to, že tento cyklus probíhá, dokud je podmínka nepravdivá. Toho dosáhneme jednoduše tak, že nám stačí skok JMC na začátek cyklu.

Cyklus repeat-until je zobrazen na obrázku 3.5.

Obrázek 3.5: Cyklus repeat-until



Příkaz for - pracuje se třídou StatementFor, která obsahuje:

- **identifier** - identifikátor iterační proměnné
- **from** - výraz dolní meze
- **to** - výraz horní meze
- **statement** - příkaz cyklus

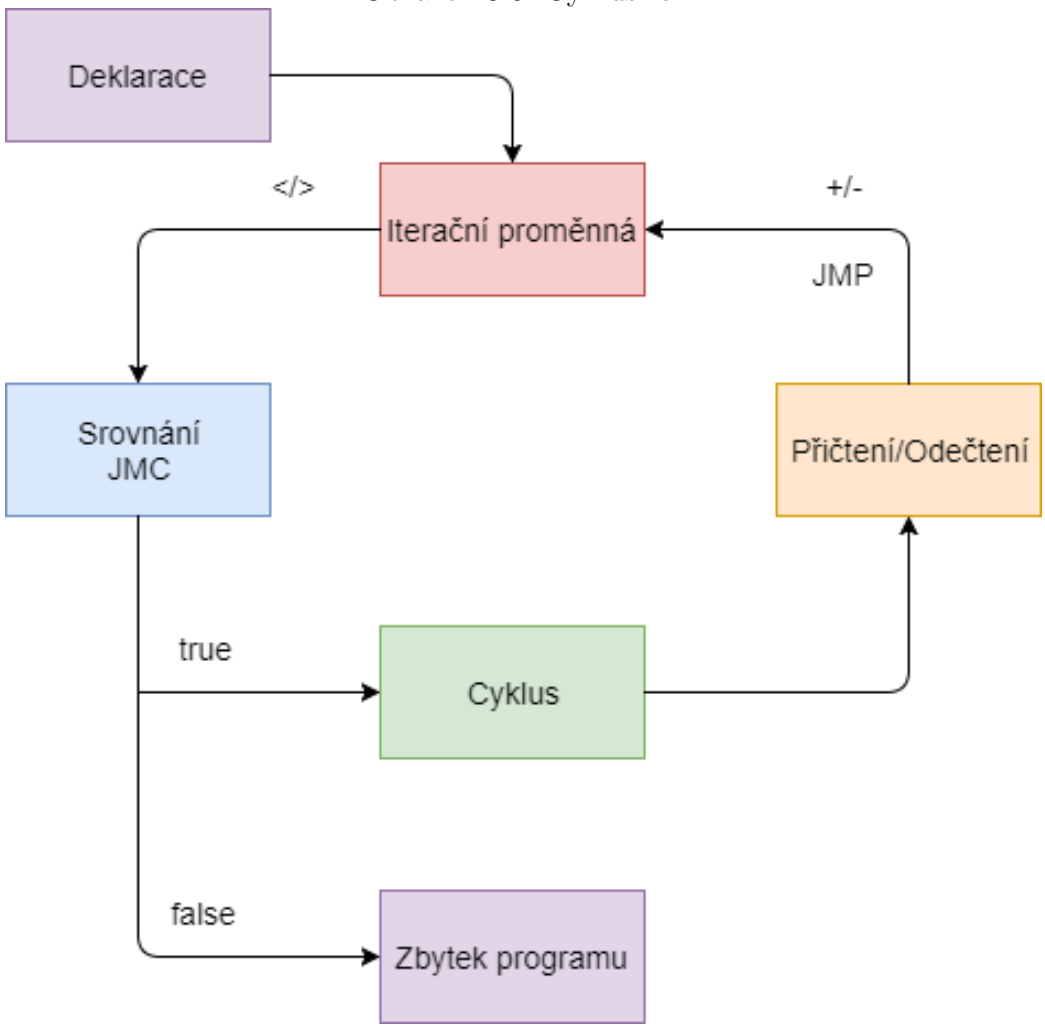
For cyklus je nejsložitější cyklus v programu. Na rozdíl od ostatních zmíněných cyklů nekontroluje pouze podmínku, ale cyklus je řízen tzv. iterační proměnnou. Cyklus spočívá v následujících krocích:

1. Deklarace iterační proměnné, dolní a horní meze
2. Přiřazení dolní meze do iterační proměnné
3. Cyklus for vzrůstající:
   (a) Srovnání, jestli je iterační proměnná menší než horní mez
   (b) Pokud ano, proveďme cyklus, pokud ne, konec cyklu
   (c) Na konci cyklu připočtěme konstantní 1 k iterační proměnné
   (d) Skok zpět na srovnání
4. Cyklus for klesající:
   (a) Srovnání, jestli je iterační proměnná větší než horní mez
   (b) Pokud ano, proveďme cyklus, pokud ne, konec cyklu
   (c) Na konci cyklu odečtěme konstatní 1 od iterační proměnné
   (d) Skok zpět na srovnání

Implementace cyklu postupuje podle tohoto návodu. Je deklarována iterační proměnná a získává hodnotu dolní meze. V závislosti na typu for cyklu je je provedeno srovnání. Skokem JMC nyní budeme kontrolovat vykonání cyklu nebo výskok z cyklu pryč. Na konci cyklu již jen stačí přičíst/odečíst konstantní 1 a skokem JMP se vrátit do bodu srovnání.

Cyklus for je zobrazen na obrázku 3.6.

Obrázek 3.6: Cyklus for



Příkaz case - pracuje se třídou StatementCase, která obsahuje:

- **expression** - kontrolní výraz
- **defaultStateemt** - příkaz defaultní větve

- **limbList** - seznam objektů CaseLimb, jež obsahují:
  - **statement** - příkaz větve
  - **atomList** - seznam objektů Atom

Příkaz case je nejsložitější příkaz v programu. Jde o obdobu příkazu if-else s tím rozdílem, že umožňuje definovat více větví s více možnostmi.

3.3 Interpret

3.3.1 Instrukční sada

3.3.2 Interpretace

3.4 Chyby programu

4 Uživatelská dokumentace

4.1 Postup přeložení a sestavení

4.2 Používání programu

5 Závěr