



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: 5. ledna 2018

Obsah

1	Zadání	1
1.1	Tvorba překladače zvoleného jazyka	1
1.2	Bodované náležitosti	3
2	Analýza	4
2.1	Analýza překladače	4
2.2	Analýza jazyka	5
3	Implementace	6
3.1	ANTLR	6
3.1.1	Gramatika	6
3.1.2	Derivační strom	9
3.2	Překladač	9
3.2.1	Programové třídy	9
3.2.2	Visitory	11
3.2.3	Datové typy	12
3.2.4	Programový mód	12
3.2.5	Tabulka symbolů	13
3.2.6	Compilery	13
3.3	Interpret	26
3.3.1	Interpretace	27
3.3.2	Instrukční sada	27
3.4	Chyby programu	30
4	Testování	33
4.1	ANTLR	33
4.2	Překladač	33
4.3	Interpret	34
5	Uživatelská dokumentace	36
5.1	Postup přeložení a sestavení	36
5.2	Používání programu	36
6	Závěr	37
6.1	Nedosažené cíle	37
6.2	Omezení	37
6.3	Dosažené cíle	38

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 řetězců)

- rozvětvená podmínka (switch, case)
- násobné přiřazení ($a = b = c = d = 3;$)
- podmíněné přiřazení / ternární operátor ($\text{min} = (a < b) ? a : b;$)
- 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 proehřesků (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 pomoci 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 rozšířené 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 ukázkou:

```
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 `Pascal0Like` 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 příkazová část ohraničená klíčovými slovy *begin* a *end*. Poté musí následovat tečka indikující konec programu.

Všechny symboly gramatiky Pascal0Like jsou k dispozici v tabulce 3.1 a všechna klíčová slova v tabulce 3.2.

Tabulka 3.1: Symboly

Symbol	Název	Popis
+	PLUS	sčítání ve výrazech
-	MINUS	odečítání ve výrazech
*	MULTIPLY	násobení ve výrazech
/	DIVIDE	dělení ve výrazech
%	MODULO	modulo ve výrazech
=	EQUAL	kontrola rovnosti dvou výrazů
<>	NOT_EQUAL	kontrola nerovnosti dvou výrazů
<	LT	kontrola, že levý výraz je menší než pravý výraz
<=	LE	kontrola, že levý výraz je menší nebo stejně velký jako pravý výraz
>=	GE	kontrola, že levý výraz je větší nebo stejně velký jako pravý výraz
>	GT	kontrola, že levý výraz je větší než pravý výraz
:=	ASSIGN	přiřazení proměnné
?	TERNARY_ONE	ternární operátor 1
!	TERNARY_TWO	ternární operátor 2
;	SEMI	konec příkazu
,	COMMA	oddělovač proměnných
:	COLON	inicializace návěští
(LPAREN	levá závorka výrazu
)	RPAREN	pravá závorka výrazu
[LBRACK	počátek výrazů paralelní deklarace
]	RBRACK	konec výrazů paralelní deklarace
.	DOT	konec programu

Tabulka 3.2: Klíčová slova

Název	Použití
-------	---------

BEGIN	počátek bloku příkazů
END	konec bloku příkazů
ODD	kontrola, zda je výraz lichý
AND	logické AND ve výrazech
OR	logické OR ve výrazech
NOT	logická negace ve výrazech
TRUE	logická 1
FALSE	logická 0
CONST	deklarace konstanty
INTEGER	deklarace číselného datového typu
REAL	deklarace číselného datového typu s plovoucí čárkou
BOOLEAN	deklarace logického datového typu
VAR	deklarace číselného typu v <i>legacy</i> módu
LABEL	deklarace návěstí
PROCEDURE	deklarace procedury
PROGRAM	deklarace programu
USE	příkaz pro použití módu
LEGACY	<i>legacy</i> mód
DEFAULT	<i>default</i> mód
STRICT	<i>strict</i> mód
GOTO	příkaz skoku na návěstí
IF	součást <i>if-then</i> větve
THEN	součást <i>if-then</i> a <i>else-then</i> větve
ELSE	součást <i>else-then</i> větve a <i>case</i> větve
REPEAT	začátek <i>repeat-until</i> cyklu
UNTIL	konec <i>repeat-until</i> cyklu
WHILE	začátek <i>while-do</i> cyklu a konec <i>do-while</i> cyklu
DO	začátek <i>do-while</i> cyklu a konec <i>while-do</i> cyklu, součástí <i>for</i> cyklu
READ	příkaz I/O vstupu
WRITE	příkaz I/O výstupu
FOR	začátek <i>for</i> cyklu
TO	indikace pro <i>for</i> cyklus přičítat k iterační proměnné
DOWNTO	indikace pro <i>for</i> cyklus odečítat od iterační proměnné
CASE	začátek větvícího příkazu <i>case-of</i>
OF	součást větvícího příkazu <i>case-of</i>

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 `Listener` nebo `Visitor`. `Listener` umožňuje pracovat s pravidlem při vstupu a výstupu z pravidla. `Visitor` nabízí jednu metodu na pravidlo, ale umožňuje vrátit 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 `Visitor`, 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 `visitor` 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 zpracováváný 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 *Variable-Type*.

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 zabrání 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. Hlavním rozdílem oproti klasické deklaraci proměnných, je

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 úrovně (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 úrovně. Následně mohou nastat dvě věci 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 úroveň, 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:

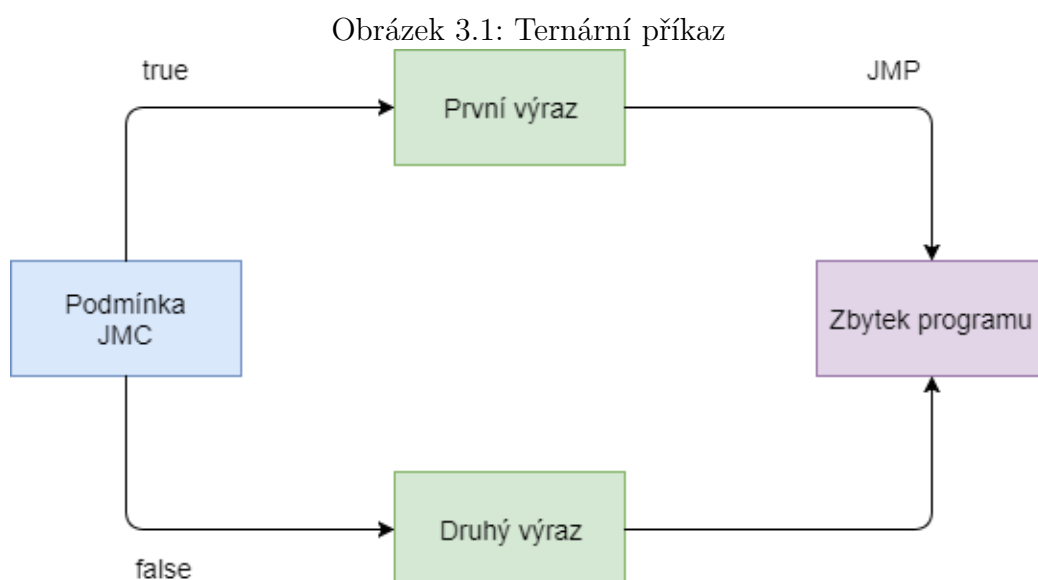
- **expression** – podmínka
- **expressionOne** – první výraz
- **expressionTwo** – druhý výraz

- **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.



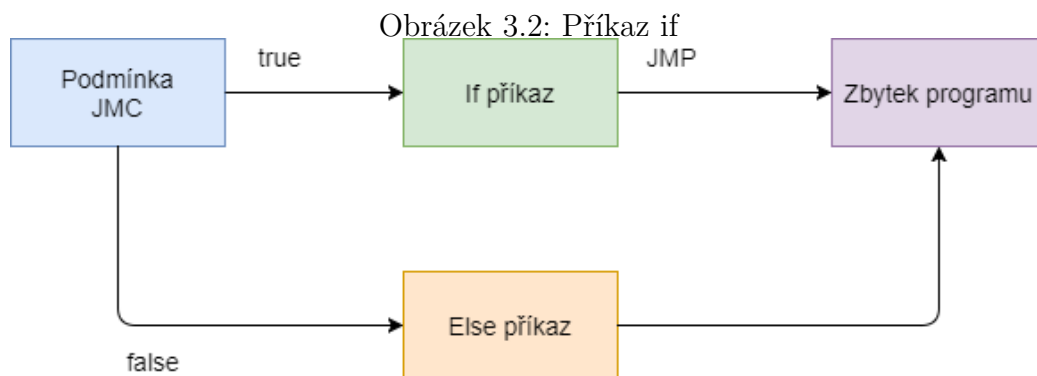
Příkaz if – pracuje se třídou Statementif, která obsahuje:

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

Příkaz if vyhodnocuje podmínku, při pravdě vykoná příkaz if, jinak přeskóčí 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.

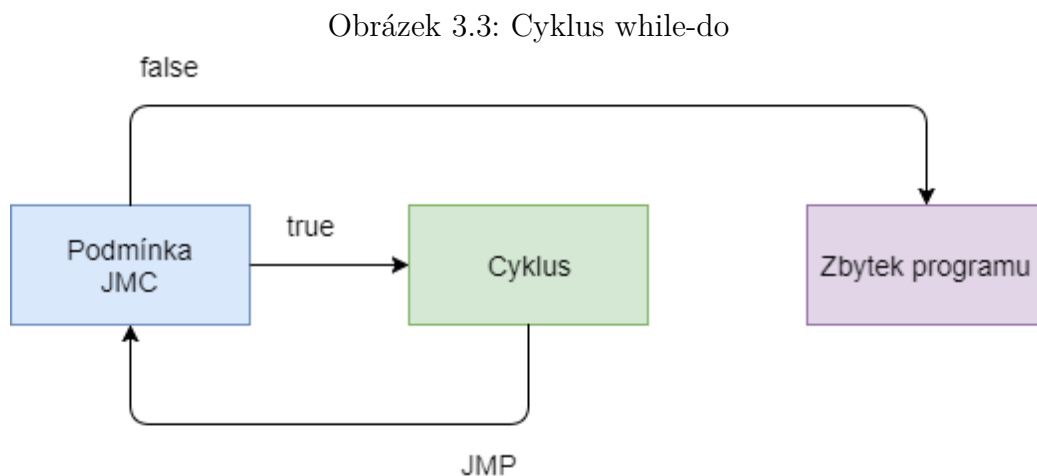


Příkaz while-do – pracuje s třídou StatementWhileDo, která obsahuje:

- **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.



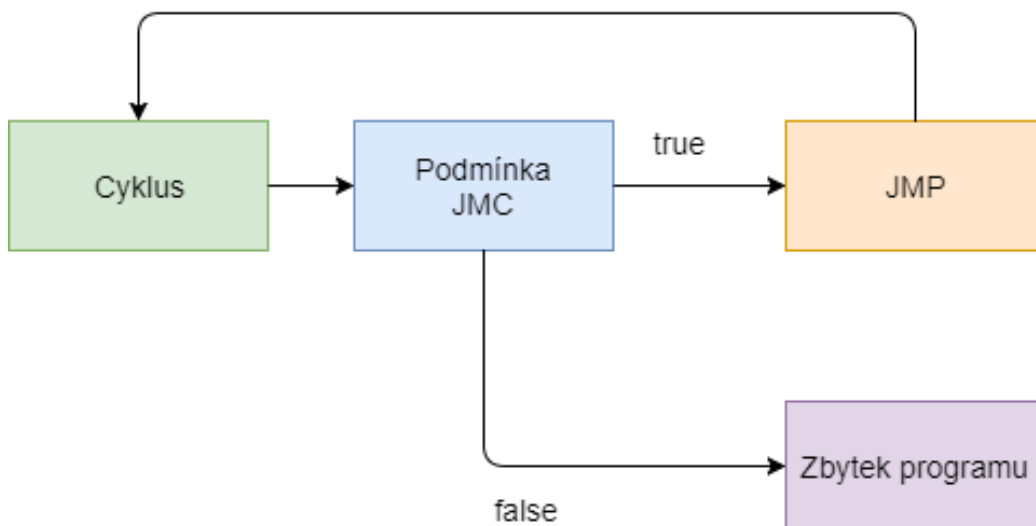
Příkaz do-while – pracuje s třídou `StatementDoWhile`, která obsahuje:

- **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



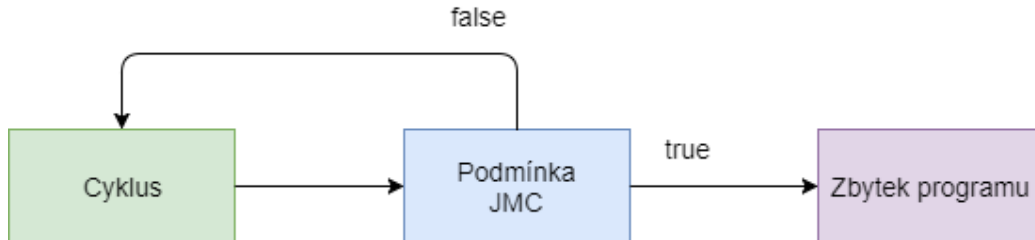
Příkaz repeat-until – pracuje s třídou `StatementRepeat`, která obsahuje:

- **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

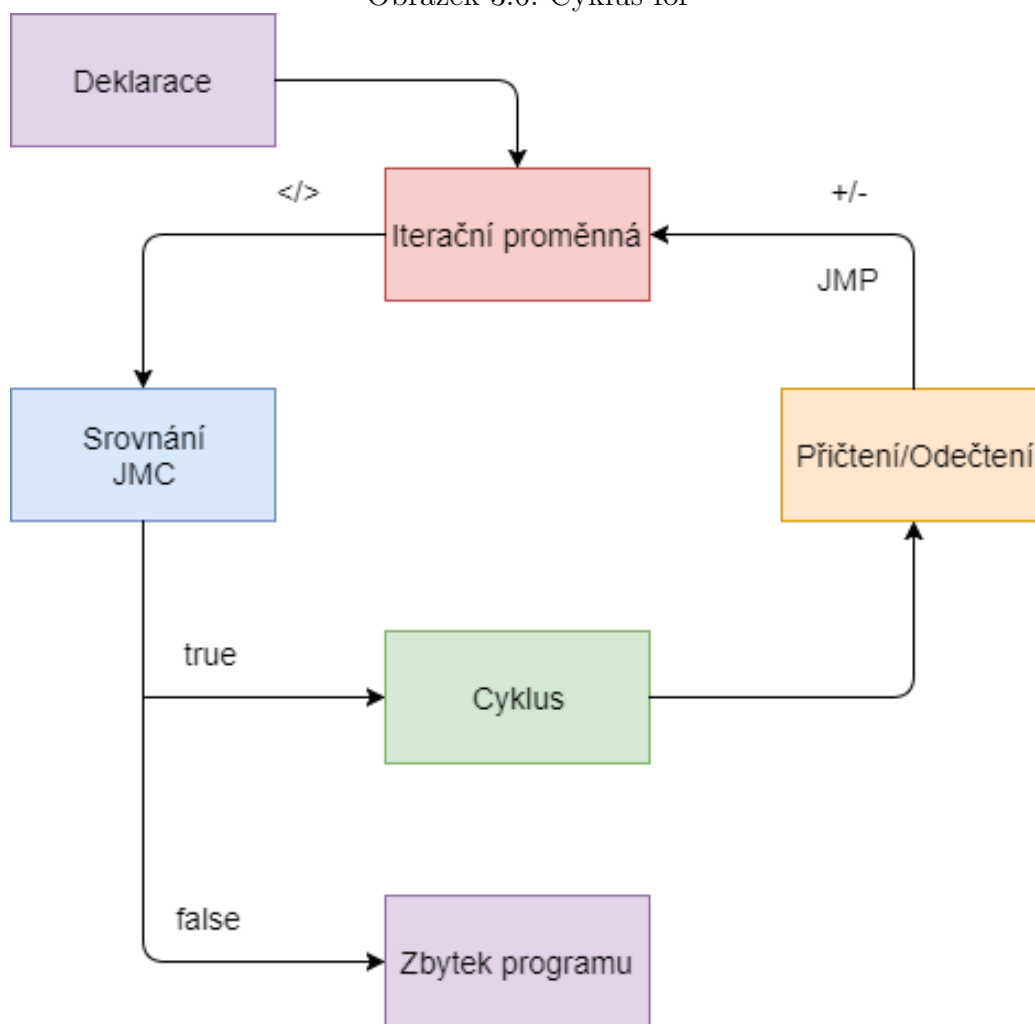
For cyklus je nejsložitější cyklus v programu. Na rozdíl od ostatních zmíněných cyklů nekontroluje pouze podmínku, ale cyklus je řízen tzv. iterační proměnnou. Cyklus spočívá v následujících krocích:

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, provedme cyklus, pokud ne, konec cyklu
 - (c) Na konci cyklu připočteme 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, provedme cyklus, pokud ne, konec cyklu
 - (c) Na konci cyklu odečteme konstantní 1 od iterační proměnné
 - (d) Skok zpět na srovnání

Implementace cyklu postupuje podle tohoto návodu. Je deklarována iterační proměnná a získává hodnotu dolní meze. Tuto hodnotu je nutné uložit na zásobník, abychom jí mohli v průběhu cyklu měnit. V závislosti na typu for cyklu 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
- **defaultStatement** – 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. Základní algoritmus pro příkaz case pascalovského typu by mohl vypadat následovně:

1. Vyhodnotíme kontrolní výraz
2. Vezmeme atomické hodnoty první větve. Srovnáme postupně s každou hodnotou kontrolní výraz, jestli se rovnají.
3. Pokud pro jednu z atomických hodnot vyhodnotíme pravdivě, vykonáme příkaz této větve, po tomto příkazu konec case.
4. Pokud se ani jedna atomická hodnota nerovná s kontrolním výrazem, opakujeme pro další větev.
5. Pokud se výraz nerovná s žádnou atomickou hodnotou z žádných větví a neexistuje defaultní větev, program nic nevykoná, jinak vykoná defaultní větev.

Při implementaci case postupujeme podle pravidel pascalovského case, které zde ještě zdůrazníme:

- Kontrolní výraz může být pouze číselného typu
- Jedna větev může mít více atomických hodnot
- Atomické hodnoty musejí být číselné
- Po vykonání příkazu ve větvi program pokračuje za case příkazem
- Podpora defaultní větve

Na začátku vyhodnotíme kontrolní výraz a uložíme ho na zásobník. Nyní budeme postupně procházet všechny větve a srovnávat výsledek kontrolního výrazu s atomickými hodnotami.

Podívejme se na jednotlivá porovnávání. Zjistíme, jestli se kontrolní výraz rovná atomické hodnotě. Pokud se mu rovná, chceme přeskočit další vyhodnocování ostatních atomických výrazů ve stejné větvi a skočit rovnou do příkazové části. Jenže nemáme k dispozici instrukci skoku při hodnotě 1, tedy true, takže si vypomůžeme menším trikem - výsledek operace znegujeme

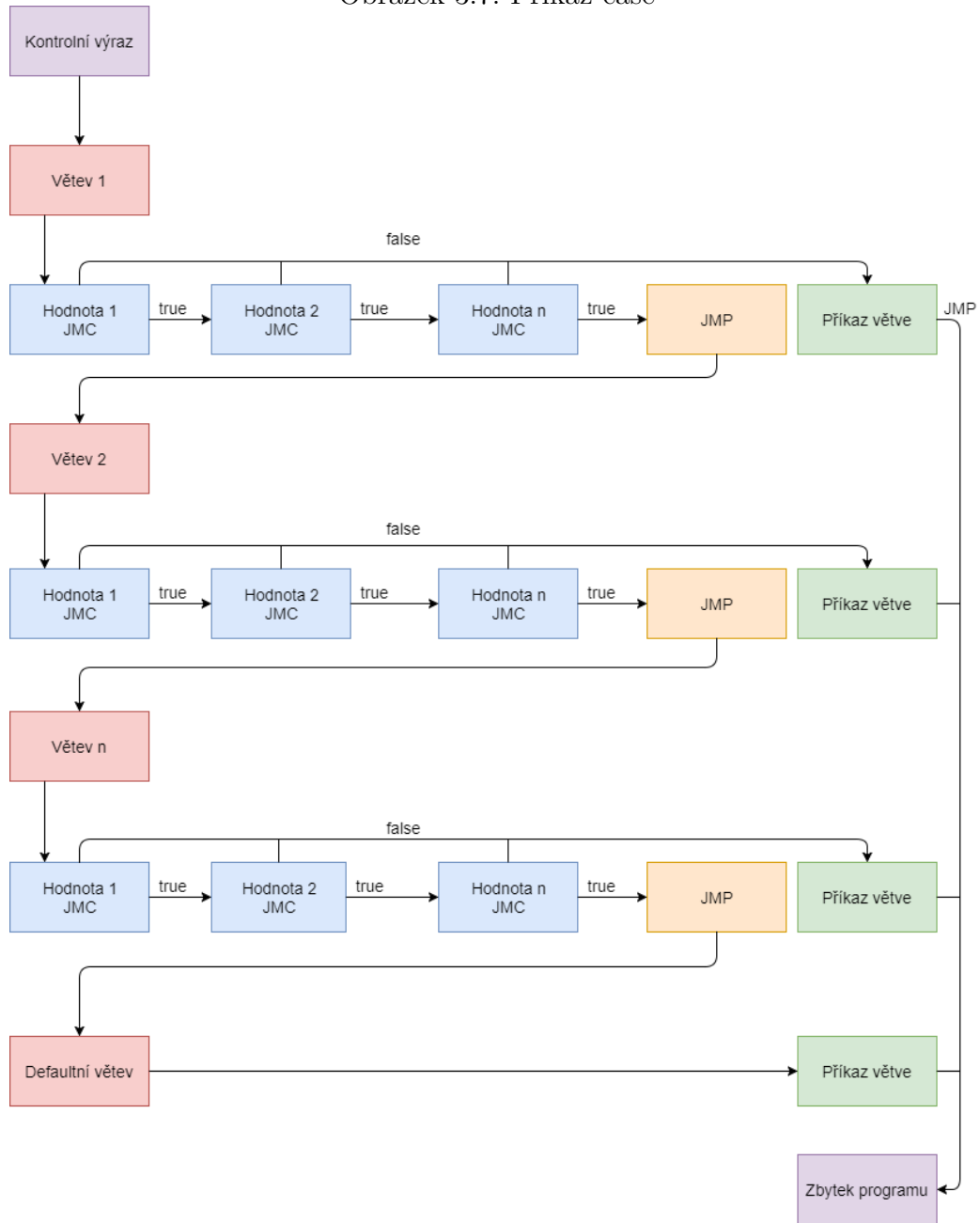
a doplníme instrukci JMC. Nyní budeme skákat na příkazovou část větve při false, tedy srovnání bylo vyhodnoceno pravdivě, a při true, tedy srovnání bylo vyhodnoceno nepravdivě, pokračujeme na další atomickou hodnotu.

Poslední atomická hodnota ve větvi je výjimkou od ostatních v tom, že při jejím vyhodnocení true, respektive false, potřebujeme skočit na další větev a tím se vyhnout příkazové části větve. Proto sem doplníme instrukci JMP. Tu samou instrukci doplníme i na konec příkazové části, abychom přeskočili zbylé větve a vyskočili z case příkazu (jde vlastně o automatickou obdobu break příkazu v jiných jazycích).

Po zkompileování všech větví se podíváme, jestli existuje defaultní větev. Pokud ano, zkompileujeme jí. Žádné atomické srovnání a skoky se zde neřeší. Po tomto je case příkaz hotov.

Příkaz case je ukázán na obrázku 3.7.

Obrázek 3.7: Příkaz case



3.3 Interpret

Posledním krokem implementační části je vytvoření interpretu. Ten má za úkol správně interpretovat instrukce, které vytvoří překladač. Tyto instrukce

jsou inspirovány instrukční sadou rozšířené PL/0.

3.3.1 Interpretace

V balíku `interpret` se nachází 2 třídy, které slouží k interpretaci:

- **Interpreter** – představuje instanci interpretu
- **Stack** – představuje zásobník a příslušné metody

Interpretace funguje tak, že se postupným průchodem souboru provádějí jednotlivé instrukce, které jsou v souboru obsaženy. V podstatě se jedná o jeden velký přepínač-switch, ve kterém se podle názvu instrukce provede to, co má. Většina základních instrukcí se shoduje s těmi, kterými disponuje základní i rozšířená verze jazyka PL/0. Protože jsme se rozhodli pro implementaci reálných čísel v **IEEE-format** bylo nutné pro tyto čísla vytvořit nové instrukce. Díky tomu je možné reálné číslo uložit pouze na jednu pozici v zásobníku.

- LRT – vloží konstantní reálnou hodnotu
- LOR – načte reálnou hodnotu vrcholu
- STR – uloží reálnou hodnotu vrcholu
- RER – načte reálné číslo ze vstupu a uloží jej na zásobník
- WRR – odebere reálné číslo z vrcholu zásobníku a vypíše jej na vstup

3.3.2 Instrukční sada

Pro úplnost, v tabulce 3.3 jsou uvedeny všechny instrukce a jejich popis, co interpret podporuje.

Tabulka 3.3: Instrukce

Kód	Instrukce	Popis
1	LIT 0, M	Vloží konstantní celou hodnotu (literál) M do zásobníku
2	LRT 0, M	Vloží konstantní reálnou hodnotu (literál) M do zásobníku
3	OPR 0, M	Operace , která se provede nad vrcholem zásobníku pro celé čísla

	OPR 0, 1	Negace ; vybere vrchol a vrátí negativní hodnotu
	OPR 0, 2	Sčítání ; vybere dvě hodnoty, sečte a vrátí
	OPR 0, 3	Odečítání ; vybere dvě hodnoty, odečte druhou první a vrátí výsledek
	OPR 0, 4	Násobení ; vybere dvě hodnoty, vynásobí a vrátí výsledek
	OPR 0, 5	Dělení ; vybere dvě hodnoty, vydělí druhou první
	OPR 0, 6	Lichost ; vybere vrchol a vloží 1 když liché, 0 když sudé
	OPR 0, 7	Modulo ; vybere dvě hodnoty, vydělí druhý prvním a vloží zbytek
	OPR 0, 8	Rovnost ; vybere dvě hodnoty, a vloží 1 pokud se rovnají, jinak 0
	OPR 0, 9	Nerovnost ; vybere dvě hodnoty a vloží 0 pokud se rovnají, jinak 0
	OPR 0, 10	Menší než ; vybere dvě hodnoty a vloží 1 pokud je první menší než druhá, jinak 0
	OPR 0, 11	Větší nebo rovno než ; vybere dvě hodnoty a vloží 1 pokud je první větší nebo rovno než druhá, jinak 0
	OPR 0, 12	Větší ; vybere dvě hodnoty a vloží 1 pokud je první větší nebo rovno než druhá, jinak 0
	OPR 0, 13	Menší nebo rovno než ; vybere dvě hodnoty a vloží 1 pokud je první menší nebo rovno než druhá, jinak 0
4	LOD L, M	Načtení ; načte hodnotu vrcholu z umístění dané offsetem M od L lexikografických úrovní dolů
5	STO L, M	Uložení ; uloží hodnotu vrcholu z umístění dané offsetem M od L lexikografických úrovní dolů
4	LOR L, M	Načtení ; načte reálnou hodnotu vrcholu z umístění dané offsetem M od L lexikografických úrovní dolů
5	STR L, M	Uložení ; uloží reálnou hodnotu vrcholu z umístění dané offsetem M od L lexikografických úrovní dolů
6	CAL L, M	Volání procedury v kódovém indexu M

7	RET 0, 0	Návrat z procedury ; vrátí se z procedury do volající procedury
8	INT 0, M	Alokování místa pro M hodnot na vrcholu zásobníku
9	JMP 0, M	Provede skok do instrukce M
10	JMC 0, M	Vybere vrchol a skočí k instrukci M pokud je rovna 0, podmíněný skok
11	REA L, M	Načte celé číslo ze vstupu a uloží jej na zásobník
12	WRI L, M	Odebere celé číslo z vrcholu zásobníku a vypíše jej na vstup
13	RER L, M	Načte reálné číslo ze vstupu a uloží jej na zásobník
14	WRR L, M	Odebere reálné číslo z vrcholu zásobníku a vypíše jej na vstup
15	OPF 0, M	Operace , která se provede nad vrcholem zásobníku pro reálná čísla
	OPF 0, 1	Negace ; vybere vrchol a vrátí negativní hodnotu
	OPF 0, 2	Sčítání ; vybere dvě hodnoty, sečte a vrátí
	OPF 0, 3	Odečítání ; vybere dvě hodnoty, odečte druhou první a vrátí výsledek
	OPF 0, 4	Násobení ; vybere dvě hodnoty, vynásobí a vrátí výsledek
	OPF 0, 5	Dělení ; vybere dvě hodnoty, vydělí druhou první
	OPF 0, 6	Lichost ; vybere vrchol a vloží 1 když liché, 0 když sudé
	OPF 0, 7	Modulo ; vybere dvě hodnoty, vydělí druhý prvním a vloží zbytek
	OPF 0, 8	Rovnost ; vybere dvě hodnoty, a vloží 1 pokud se rovnají, jinak 0
	OPF 0, 9	Nerovnost ; vybere dvě hodnoty a vloží 0 pokud se rovnají, jinak 0
	OPF 0, 10	Menší než ; vybere dvě hodnoty a vloží 1 pokud je první menší než druhá, jinak 0
	OPF 0, 11	Větší nebo rovno než ; vybere dvě hodnoty a vloží 1 pokud je první větší nebo rovno než druhá, jinak 0

	OPF 0, 12	Větší ; vybere dvě hodnoty a vloží 1 pokud je první větší nebo rovno než druhá, jinak 0
	OPF 0, 13	Menší nebo rovno než ; vybere dvě hodnoty a vloží 1 pokud je první menší nebo rovno než druhá, jinak 0
16	RTI 0, 0	Reálné číslo na celé číslo ; vybere jednu hodnotu ze zásobníku a vloží celou část čísla do zásobníku
17	ITR 0, 0	Celé číslo na reálné číslo ; vybere jednu hodnotu ze zásobníku a vloží číslo jako reálné do zásobníku
18	NEW 0, 0	Alokace na haldě ; alokuje se jedno místo na haldě, na zásobník vloží hodnotu představující pozici místa v haldě
19	DEL 0, 0	Uvolnění místa na haldě ; odebere ze zásobníku jednu hodnotu a to adresu na haldě, kterou uvolní
20	LDA 0, 0	Načtení hodnoty z haldy ; odebere ze zásobníku hodnotu a vloží hodnotu z haldy
21	STA 0, 0	Uložení hodnoty na haldy ; odebere dvě hodnoty zásobníku. Na první představující adresu uloží druhou v haldě
22	PLD 0, 0	Dynamické načtení hodnoty z místa určeného L/A ; odebere ze zásobníku dvě hodnoty. První je úroveň zanoření a druhá je relativní pozice. Z té se načte hodnota na vrchol zásobníku
23	PST 0, 0	Dynamické uložení hodnoty z místa určeného L/A ; odebere ze zásobníku tři hodnoty. První je úroveň zanoření, druhá relativní pozice a třetí odebraná hodnota představuje hodnotu k uložení

3.4 Chyby programu

Program při vykonávání může narazit na několik různých chyb při překlada. Problémy nalezené při analýze řeší ANTLR a program je pouze zachycuje a vyhazuje jako výjimku. Chyby při syntéze a interpretaci jsou již v rukách tvůrců a tak byly sestaveny jednotlivé chyby, jejich kód a popis chyby. Program zároveň vrací tento kód chyby jako svůj návratový kód. Tyto chyby jsou zaneseny do tabulky 3.4.

Tabulka 3.4: Chybové kódy

Kód	Chyba	Komentář
0	Žádná chyba	Program proběhl správně
Překladač		
1	Variable not initialized	Použili jste hodnotu předtím, než byla inicializována
2	Variable already declared	Pokoušíte se deklarovat proměnnou, která již byla deklarována
3	Constant reassign	Pokoušíte se změnit hodnotu konstanty
4	Unknown identifier	Vyskytl se neznámý identifikátor
5	Unknown label	Vyskytlo se neznámé návěští
6	Unknown procedure	Vyskytla se neznámá procedura
7	Parallel declaration number mismatch	Paralelní deklarace vyžaduje stejný počet proměnných a výrazů
8	Incompatible types	Typový výsledek výrazu neodpovídá očekávanému typu
9	Expression variable type mismatch	Ve výrazu se vyskytly neslučitelné typy
10	Strict mode expression variable type mismatch	Striktní mód nedovoluje smíšení různých typů ve výrazu
11	Can not assign loop variable	Nepoužil jste unikátní proměnnou jako iterační proměnnou for cyklu
12	Label used elsewhere	Návěští již bylo použito jinde
13	Label out of reach	Návěští je mimo dosah
14	Label never used	Skáčete na návěští, které není v programu použito
15	Label not allowed in for cycle	Návěští není povoleno ve for cyklu
16	No boolean type in case	Case větev nepodporuje logický datový typ

17	Case atom used multiple times	Definujete <code>case</code> větev, která již byl použita
18	Legacy mode only allows var type	Legacy mód povoluje pouze typ <code>var</code>
19	Legacy mode real constant not allowed	Legacy mód nepovoluje konstatní reálné číslo
20	Legacy mode boolean constant not allowed	Legacy mód nepovoluje logickou konstantu
21	Legacy mode logical expression not allowed	Legacy mód nepovoluje logické výrazy
22	Legacy mode negation expression not allowed	Legacy mód nepovoluje výraz negace
23	Non legacy mode var type not allowed	<code>Var</code> typ není povolen v <code>non-legacy</code> módu
Interpret		
24	Stack overflow	Došlo k přetečení zásobníku
25	Stack underflow	Došlo k podtečení zásobníku
26	Unknown instruction	Tato instrukce není známa
27	Program counter is out of range	Čítač instrukcí je mimo rozsah

4 Testování

Program byl kromě pečlivého ručního otestování autorů otestován sadou automatických funkcionálních testů za podpory knihovny JUnit. Veškeré třídy se nacházejí v balíku *test.zcu.cz.kiv.fjp* a veškeré testovací soubory ve složce *testFiles*.

4.1 ANTLR

Testování ANTLR, potažmo analytické části překladače, je o otestování, že přijmeme správné vstupy a odmítneme nesprávné. Jde tedy o testování funkčnosti pravidel gramatiky. Správně vstupy jsou předpokladem pro další část překladu a proto byla věnována zvýšená pozornost této části testování.

Jednotlivé testovací třídy jsou v balíku *ANTLR*.

- **TestProgram** - testovací sada na základní vlastnosti programu týkající se především hlavičky programu.
- **TestDeclarations** - testovací sada na deklarace. Důraz kladen na správné přijetí všech druhů deklarací včetně všech jejich povolených tvarů.
- **TestExpressions** - testování vlastností výrazů. Kontrola, že dodržují předepsaná pravidla o počtu výrazů, povolené symboly, klíčová slova apod.
- **TestStatements** - testovací sada na příkazy. Správné tvary všech příkazů včetně návěští.
- **TestSuiteANTLR** - testovací sada, která spustí všechny zmíněné testy.

4.2 Překladač

Testování samotného překladače je složitý problém. Nasnadě je využít přeložené instrukce pro další budoucí testování. Jenže to vyvolává mnohem více otázek než zodpovídá. Kde tyto instrukce vezmeme? Náš překladač podporuje velké množství příkazů, mnohem více než PL/0, navíc využívá vlastní instrukce pro převod čísel a reprezentaci reálného čísla. Musel by se tedy použít samotný překladač. To by znamenalo srovnávat starší výsledek překladače

s novými výsledky překladače. Smysl takového testování je velice malý. Přes srovnávání instrukcí otestování překladače nevede.

Zaměřme se nejdříve na to, co testovat snadno lze - otestování chybových stavů. Překladač při naražení na problém, vyvolá chybový stav, vypíše chybovou hlášku na konzoli a ukončí program přes *System.exit*. Návrátová hodnota tohoto volání odpovídá číslu chyby uvedené v tabulce 3.4. Knihovna JUnit neumožňuje testovat tuto návratovou hodnotu a tak si vypomůžeme další knihovnou, která tuto funkcionalitu do JUnit doplňuje - *System rules*.

V balíku *compiler* se nachází **TestErrors**. Tato třída průběžně otestuje správnou reakci překladače na chybné vstupy kontrolou návratové hodnoty *System.exit* pomocí výše popsané knihovny.

Další třídou je **TestComplex**. Tato třída testuje komplexnější programy. Bez výpomoci třetího si zde nepomůžeme a tak toto testování probíhá za asistence interpretu. Všechny testy využívají předpřipravená data ve třídě **TestData**. Všechny programy vždy vypisují výsledek pomocí instrukcí pro I/O, a právě tyto výstupy poté kontrolujeme v testovacích sadách.

V rozhodovacích problémech odpovídají programy buď **1** (ano) nebo **-1** (ne).

Testujeme následující programy:

- **armstrong** - Program rozhodne o zadaném čísle, jestli se jedná o Armstrongovo číslo.
- **factorial** - Program vypočítává faktoriál zadaného čísla.
- **fahrenheit** - Program převádí zadané číslo z celsia do fahrenheit.
- **fibonacci** - Program vypíše prvních n čísel Fibonacciho posloupnosti.
- **palindrome** - Program rozhodne o zadaném čísle, jestli se jedná o palindrom.
- **prime** - Program rozhodne o zadaném čísle, jestli se jedná o prvočíslo.

Všechny testovací sady překladače lze spustit najednou pomocí třídy **TestSuiteCompiler**.

4.3 Interpret

Aby bylo co nejlépe otestováno správné fungování interpretu, je zapotřebí pro každou instrukci z instrukční sady vytvořit testovací scénář a na něm

potřebnou funkcionalitu ověřit. Zároveň si sám interpret dokáže poradit i s nekorektním vstupním souborem a proto je nutné otestovat i toto.

Otestování správné funkčnosti instrukcí je docíleno porovnáním vypsaného čísla ze zásobníku a správné hodnoty. Korektní hodnotu jsme zjistili pomocí ručního zpracování instrukcí a jiných dostupných interpretů. Chybové stavy jsou řešeny obdobným způsobem, jako tomu je u testové sady pro chybné vstupy u překladače.

V balíku *interpreter* se proto nachází dvě třídy.

- **TestComplex** - testovací sada ověřující základní funkcionalitu interpretu.
- **TestErrors** - testovací sada ověřující korektní reakci programu na chybný vstupní sadu instrukcí.

5 Uživatelská dokumentace

5.1 Postup přeložení a sestavení

5.2 Používání programu

6 Závěr

6.1 Nedosažené cíle

Od původní analýzy se implementovaný program liší ve dvou věcech.

Za prvé původním cílem byl překlad našeho jazyka do instrukcí PL/0. To bylo splněno částečně vzhledem k tomu, že pro realizaci překladu reálných čísel byly použity vlastní instrukce a nikoli instrukce rozšířené PL/0.

Za druhé nebyl realizován datový typ řetězec. Z důvodů časové tísně a dalších povinností byla implementace tohoto datového typu nejdříve odložena načež zrušena.

6.2 Omezení

I přes implementaci některých funkcí jsou tyto funkce omezené v určitých aspektech:

- Návěští musí být použito pouze na začátku příkazu. To sebou nese určitá omezení v případě nutnosti skočit z cyklu ven, pokud za tímto cyklem nic není.
- Hlavička *program(název)* je vyžadována i přesto, že v programu ničemu neslouží.
- I/O příkazy nepodporují konstanty
- Překladač nepodporuje logické konstanty v číselných výrazech (pouze v logických)
- Iterační proměnná *for* cyklu musí být nedeklarována
- *Case* pracuje pouze s číselnými typy
- Konstanta může být pouze atomická
- Deklarovat proměnné lze pouze v deklaracioní části

6.3 Dosažené cíle

Hlavní předsevzatý cíl bylo splnit zadání, tedy vytvořit překladač vlastního jazyka. Tento cíl byl později doplněn o vlastní interpret. Překladač i interpret byly otestovány automatickými testy na jednoduchých i složitějších případech. Funkcionalitu programu shrňme v následujícím seznamu:

- Programový mód - *legacy*, *default*, *strict*
- **Datové typy**
 - Celé číslo
 - Reálné číslo
 - Logická hodnota
- **Deklarace**
 - Deklarace konstant
 - Deklarace návěští
 - Deklarace proměnných
 - Deklarace paralelních proměnných
 - Deklarace procedury
- **Výrazy**
 - Unární mínus
 - Lichost
 - Negace
 - Násobení, dělení, modulo
 - Sčítání, odčítání
 - Srovnání
- **Příkazy**
 - I/O příkaz
 - Příkaz přiřazení
 - Ternární příkaz
 - Goto příkaz
 - Volání procedury

- Blokový příkaz (*begin end*)
- Cyklus *do while*
- Cyklus *while do*
- Cyklus *repeat until*
- Cyklus *for*
- Příkaz podmínky *if*
- Příkaz větvení *case*
- **Interpret podporuje**
 - Základní PL/0
 - Rozšířená PL/0
 - Naše instrukce pro reálná čísla