FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

Semestrální práce z předmětu KIV/FJP – Formální jazyky a překladače

Překlad latinské Javy do instrukční sady PL/0

Autor
Jiří Třesohlavý, bc. Jan Vašátko
jirtres@students.zcu.cz
vasatkoj@students.zcu.cz

Přednášející
Ing. Richard Lipka, Ph.D.
lipka@kiv.zcu.cz

Obsah

Kapitola 1

Popis zadání

Cílem semestrální práce bylo navrhnout a implementovat překladač, který přeloží zdrojový kód napsaný v modifikovaném jazyce `Java` s latinskými klíčovými slovy do instrukční sady `PL/0`. Tento úkol představuje specifickou výzvu v oblasti programovacího jazyka a překladových technik, neboť zahrnuje jak syntaktickou a sémantickou analýzu zdrojového kódu, tak i jeho následnou transformaci do nízkoúrovňové reprezentace.

Základním problémem bylo pracovat s hybridním jazykem, kde klíčová slova známá z tradiční `Javy` (např. `if`, `while`, `return`) byla nahrazena latinskými ekvivalenty (např. `si`, `dum`, `redi`). Tato změna vyžadovala úpravu lexikální analýzy a syntaktického analyzátoru tak, aby rozpoznával a správně zpracovával tyto nové instrukce, přičemž zůstala zachována celková struktura a pravidla jazyka `Java`.

Dalším klíčovým krokem bylo převést syntaktický strom vytvořený během analýzy na instrukce `PL/0`, což je jednoduchá zásobníková virtuální strojová sada příkazů. Tento proces zahrnoval návrh efektivní strategie generování kódu, při které se vyšší programovací konstrukce (např. podmínky, smyčky nebo volání funkcí) transformovaly do sekvencí instrukcí `PL/0`. Například konstrukce typu `dum` byla přeložena do sekvencí, které zahrnovaly podmíněné skoky a opakované vykonávání bloků kódu.

Výzvy této práce zahrnovaly především správné mapování mezi strukturami obou jazyků a zajištění konzistence generovaného kódu. Každý jazykový prvek musel být pečlivě analyzován, aby bylo zajištěno, že jeho ekvivalent v `PL/0` odpovídá zamýšlené funkcionalitě.

Výsledkem práce je funkční překladač, který ukazuje, jak lze propojit prvky dvou různých jazykových světů, přičemž demonstruje klíčové principy konstrukce překladačů a jejich praktickou aplikaci.

Kapitola 2

Použité technologie

Pro implementaci překladače jsme zvolili jazyk `Java` jako hlavní vývojový nástroj díky jeho robustnosti, široké podpoře knihoven a vhodnosti pro práci s textovou analýzou. `Java` nabízí platformovou nezávislost a bohaté prostředí pro práci s datovými strukturami, což usnadnilo zpracování jednotlivých fází překladače – od lexikální analýzy přes syntaktickou analýzu až po generování cílového kódu.

Hlavní roli v projektu sehrála knihovna `ANTLR`, která umožňuje snadné vytváření lexikálních a syntaktických analyzátorů. `ANTLR` jsme využili pro definici gramatiky modifikovaného jazyka `Java`, která zahrnovala latinská klíčová slova místo tradičních anglických výrazů. Na základě této gramatiky `ANTLR` automaticky vygeneroval parser, který dokázal rozpoznat syntaktické struktury zdrojového kódu.

Výhodou použití `ANTLR` bylo především zjednodušení práce s pravidly gramatiky a jejich snadná rozšiřitelnost. Díky nástrojům, které `ANTLR` poskytuje, jsme mohli rychle identifikovat a opravit případné nejednoznačnosti v gramatice, což přispělo k přesnosti a stabilitě překladače.

Po provedení syntaktické analýzy vygeneroval parser abstraktní syntaktický strom `AST`, který představoval vnitřní reprezentaci zdrojového kódu. Tato reprezentace byla následně zpracována v `Javě`, kde jsme implementovali sémantickou analýzu a překlad do instrukční sady `PL/0`. `Java` nám zde umožnila efektivně manipulovat s datovými strukturami a řídit proces generování cílového kódu

Kapitola 3

Lexikální a syntaktická analýza

Lexikální a syntaktická analýza jsou dvě základní fáze procesu překladu zdrojového kódu, které zajišťují transformaci zdrojového kódu do reprezentace, s níž může pracovat překladač. V našem projektu jsme implementovali tyto fáze s využitím nástroje `ANTLR`.

Lexikální analýza

Lexikální analýza je prvním krokem při zpracování zdrojového kódu. Jejím úkolem je rozdělit vstupní text programu do základních stavebních jednotek (**tokenů**). Každý token reprezentuje určitou kategorii, například klíčové slovo identifikátor, literál nebo symbol.

V našem projektu jsme deklarovali pravidla pro rozpoznávání tokenů v gramatice `ANTLR`. Tato pravidla zahrnují definice klíčových slov v latinském jazyce.

```
1  ...
2  CASE          : 'casus';
3  CHAR          : 'char';
4  CLASS         : 'class';
5  CONTINUE      : 'persevera';
6  DEFAULT       : 'defaltam';
7  DO            : 'face';
8  ELSE          : 'aliter';
9  FINAL         : 'finalis';
10 FLOAT         : 'float';
11 FOR           : 'pro';
12 ...
```

Ukázka 1: Přepsání klíčových slov v souboru `IavaLexer.g4`

```
...
DECIMAL_LITERAL : '0' | [1-9] (Digits? | '_'+ Digits);
HEX_LITERAL     : '0' [xX] [0-9a-fA-F] ([0-9a-fA-F_]* [0-9a-fA-F])
    ?;
OCT_LITERAL     : '0' '_'* [0-7] ([0-7_]* [0-7])?;
BINARY_LITERAL  : '0' [bB] [01] ([01_]* [01])?;

FLOAT_LITERAL:
    (Digits '.' Digits? | '.' Digits) ExponentPart? [fF]?
    | Digits (ExponentPart [fF]? | [fF])
;

HEX_FLOAT_LITERAL: '0' [xX] (HexDigits '.'? | HexDigits? '.'
    HexDigits) [pP] [+-]? Digits [fF]?;
...
```

Ukázka 2: Definování číselných literálů v souboru `IavaLexer.g4`

Syntaktická analýza

Po vytvoření tokenů následuje syntaktická analýza, která ověřuje, zda je posloupnost tokenů syntakticky správná (dle definované gramatiky v souboru `IavaParser.g4`) a sestavuje z nich `AST`. V této fázi jsme definovali syntaktická pravidla na úrovni jazykové gramatiky, která zahrnují konstrukce jako deklarace tříd, metod, příkazové bloky, podmínky a cykly.

Definovaná gramatika

Následující popis obsahuje ty nejdůležitější struktury gramatiky. Celou gramatiku lze dohledat v souboru `IavaParser.g4`

Gramatika v souboru `IavaParser.g4` definuje syntaxi modifikovaného jazyka `Java` a je psaná v `ANTLR` formátu. Obsahuje pravidla pro zpracování deklarací tříd, metod, proměnných, výrazů a dalších konstrukcí.

Hlavní struktura `compilationUnit`

Pravidlo `compilationUnit` je výchozím bodem parseru. Určuje, že zdrojový kód může obsahovat jednu nebo více deklarací tříd či prázdných příkazů ukončených středníkem.

```
compilationUnit
    : (typeDeclaration | ';')* EOF
    ;
```

Deklarace tříd `classDeclaration`

Pravidlo pro deklaraci třídy definuje základní syntaxi tříd s klíčovým slovem `CLASS` (definovaným v `IavaLexer.g4`), identifikátorem a tělem třídy:

```
classDeclaration
    : CLASS identifier classBody
    ;
```

Tělo třídy (`classBody`) je obaleno {} a může obsahovat více deklarací, jako jsou metody, atributy nebo další třídy:

```
classBody
    : '{' classBodyDeclaration* '}'
    ;
```

Deklarace metod `methodDeclaration`

Metody mohou být `void` nebo vracet určité typy. Pravidlo pro deklaraci metody obsahuje:

- Návratový typ `typeTypeOrVoid`
- Název metody
- Parametry `formalParameters`
- Tělo metody

```
methodDeclaration
    : typeTypeOrVoid identifier formalParameters ('[' ']')*
      methodBody
    ;
```

Tělo metody je definováno jako blok

```
methodBody
    : block
    ;
```

Proměnné `fieldDeclaration`

Deklarace atributů třídy zahrnují typ, názvy proměnných a jejich inicializace:

```
fieldDeclaration
    : typeType variableDeclarators ';'
    ;
```

Více proměnných může být deklarováno na jednom řádku pomocí čárek:

```
variableDeclarators
    : variableDeclarator (',' variableDeclarator)*
    ;
```

Výrazy a literály

Gramatika definuje širokou škálu literálů, jako jsou celá čísla, řetězce a logické hodnoty:

```
literal
    : integerLiteral
    | floatLiteral
    | CHAR_LITERAL
    | STRING_LITERAL
    | BOOL_LITERAL
    | NULL_LITERAL
    ;
```

Tvorba AST

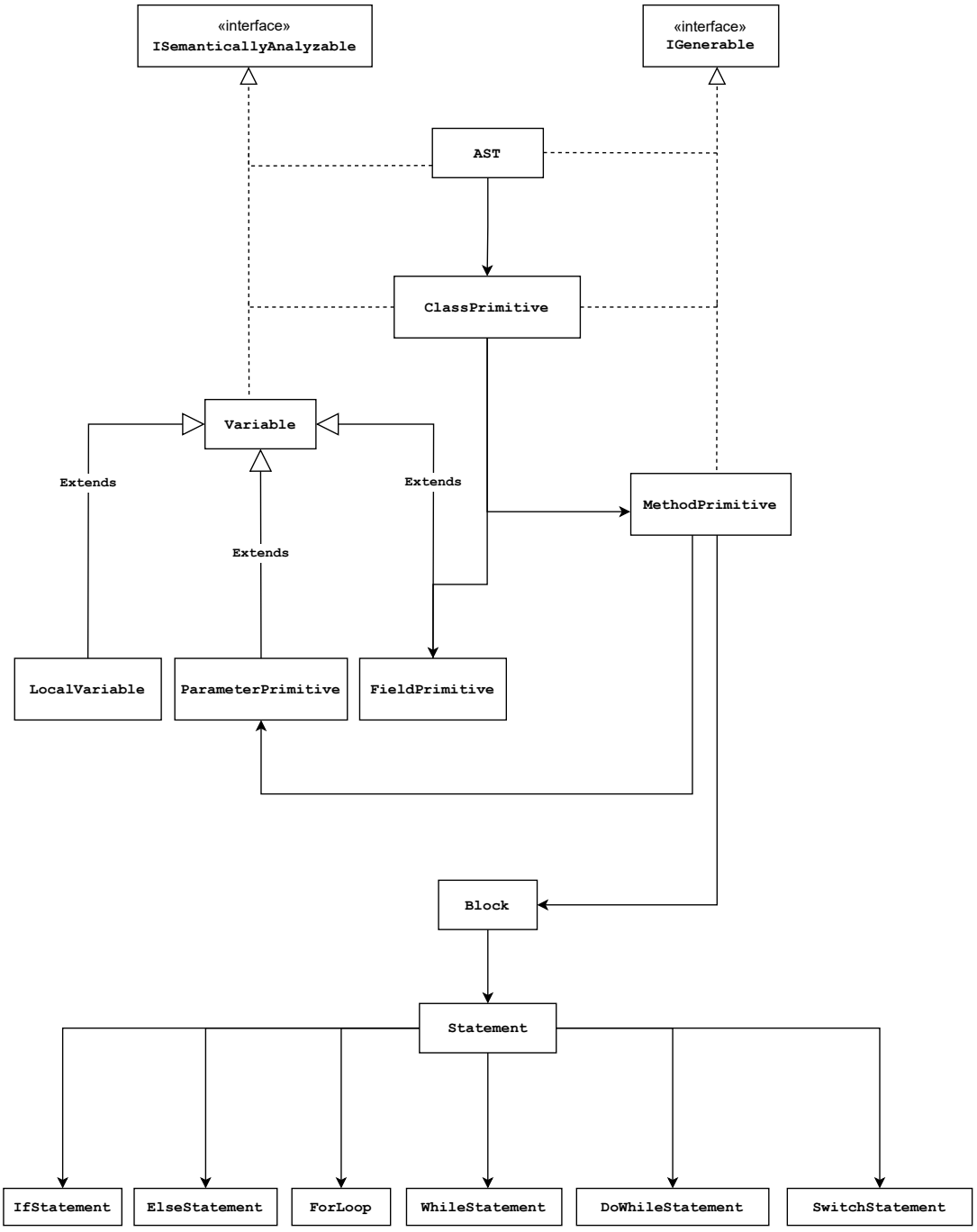
V projektu jsme využili knihovnu `ANTLR`, která umožňuje generovat `ParseTree` (syntaktický strom) na základě gramatiky. Syntaktický strom obsahuje všechny informace o struktuře zdrojového kódu, ale jeho formát není ideální pro následné fáze překladače, jako je semantická analýza nebo generování kódu. Proto jsme se rozhodli vytvořit vlastní abstraktní syntaktický strom (`AST`), který z `ParseTree` extrahuje pouze relevantní informace a reprezentuje strukturu programu v jednodušším a kompaktnějším formátu.

Transformace na `AST`

Vlastní `AST` jsme vytvořili extrakcí klíčových částí `ParseTree`. Tato transformace probíhá v několika krocích:

- Návštěva uzlů `ParseTree` pomocí návrhnového vzoru `visitor`
- Extrahovali jsme z `ParseTree` potřebné informace, které budeme potřebovat k sémantické analýze a ke generování instrukcí.

- **Vytvoření hierarchie** `AST`: Pro každý důležitý konstrukční prvek, jako jsou deklarace tříd, metod nebo výrazů, jsme vytvořili odpovídající uzel `AST`. Každý uzel obsahuje pouze informace potřebné pro další fáze.



Obrázek 3.1: UML diagram `AST` stromu

Kapitola 4

Sémantická analýza

Semantická analýza je klíčovou fází překladače, která ověřuje, zda program odpovídá pravidlům jazyka na úrovni sémantiky. Zatímco lexikální a syntaktická analýza zajišťují správnost struktury programu, sémantická analýza kontroluje významové aspekty, jako je správné použití typů, existence proměnných nebo platnost výrazů. Implementace této fáze v našem projektu probíhá na základě vlastního abstraktního syntaktického stromu (`AST`).

Průběh semantické analýzy

Sémantická analýza probíhá procházením uzlů `AST`, který jsme vytvořili z `ParseTree`. Každý typ uzlu odpovídá určité konstrukci programu (např. deklarace třídy, metoda, výraz) a má definován svůj způsob kontroly. Každý uzel abstraktního syntaktického stromu (`AST`) implementuje rozhraní `ISemanticallyAnalyzable` (viz `UML diagram 3.1`).Celá sémantická analýza je spuštěna pomocí třídy `SemanticAnalyzer`. Tato třída přijímá parametr (v našem případě náš `AST`) typu `ISemanticallyAnalyzable` a zprvu zavolá metodu `collectData()`, která naplní tabulku symbolů potřebnými daty. Tato data jsou posléze analyzována zavoláním metody `analyze()`.

Tabulka symbolů

V rámci analýzy je využita tabulka symbolů, která uchovává informace o deklarovaných proměnných, třídách a metodách. Tabulka umožňuje:

- Kontrolu, zda jsou proměnné a metody deklarovány před použitím
- Zajištění, že názvy proměnných a metod v jedné oblasti viditelnosti nejsou v konfliktu

Kontrola typové kompatibility

Typová kontrola ověřuje, zda jsou operace a přiřazení mezi datovými typy validní. Například:

- Operace mezi neslučitelnými typy (např. `int + boolean`) je označena jako chyba.
- Přiřazení hodnoty proměnné kontroluje kompatibilitu typu proměnné a výrazu.

Rozsah platnosti (scope)

Semantická analýza ověřuje rozsah platnosti proměnných a správné uzavření bloků, aby nedocházelo k přístupu k proměnným mimo jejich rozsah.

Kontrola deklarací funkcí a parametrů

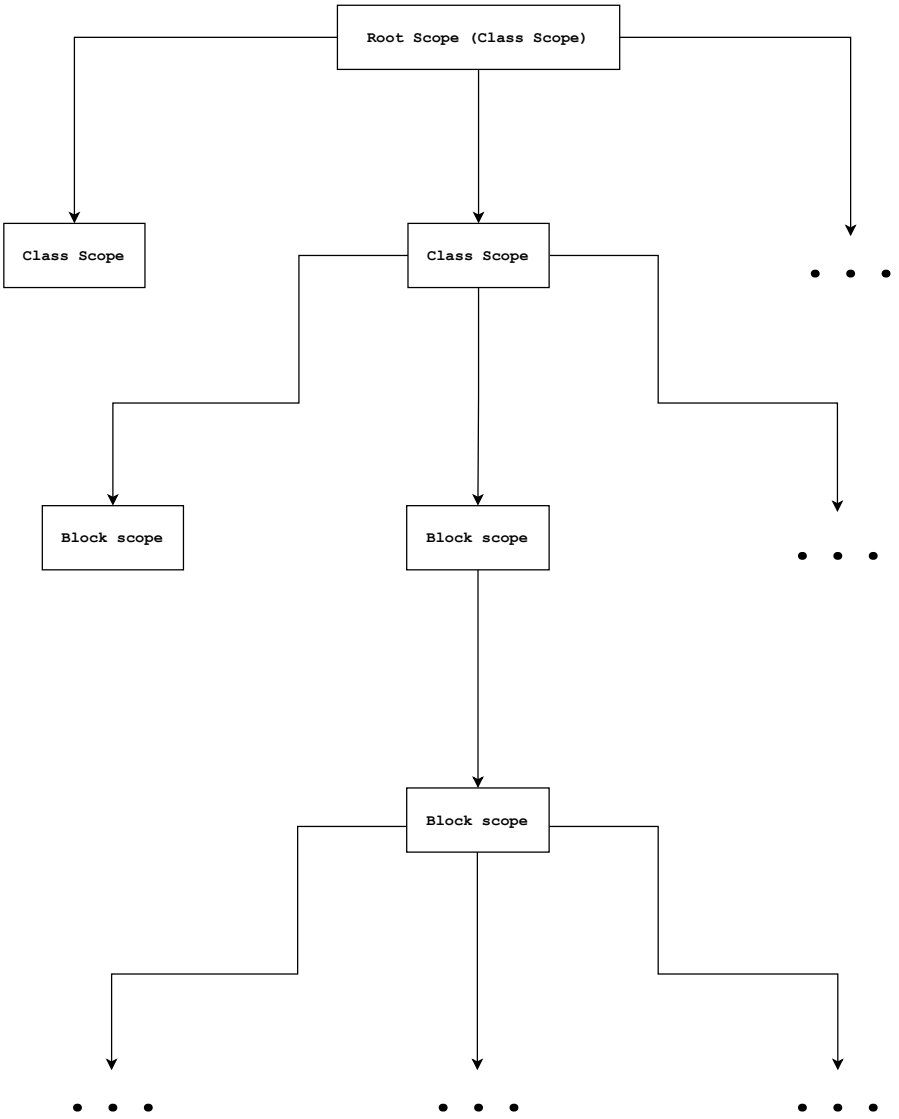
Metody a jejich parametry jsou kontrolovány na počet a typy při volání, aby odpovídaly deklaraci.

Implementace

Jako řešení jsme zvolili víceprůchodovu sémantickou analýzu. V prvním průchodu si sbíráme z `AST` potřebná data (jména tříd, proměnné, metody a bloky kódu (`for`, `while`, ...) a ukládáme je do tabulky symbolů. V druhém průchodu analyzujeme, zda se neporušuje typová kompatibilita, volání neexistujících metod, používání proměnných mimo svůj `scope` nebo zda konstanty nejsou nijak přepisovány. Námi implementovaná sémantická analýza se drží regulí jazyka `Java`, tudíž téměř všechna sémantika je v našem jazyce stejná jako v jazyce `Java`.

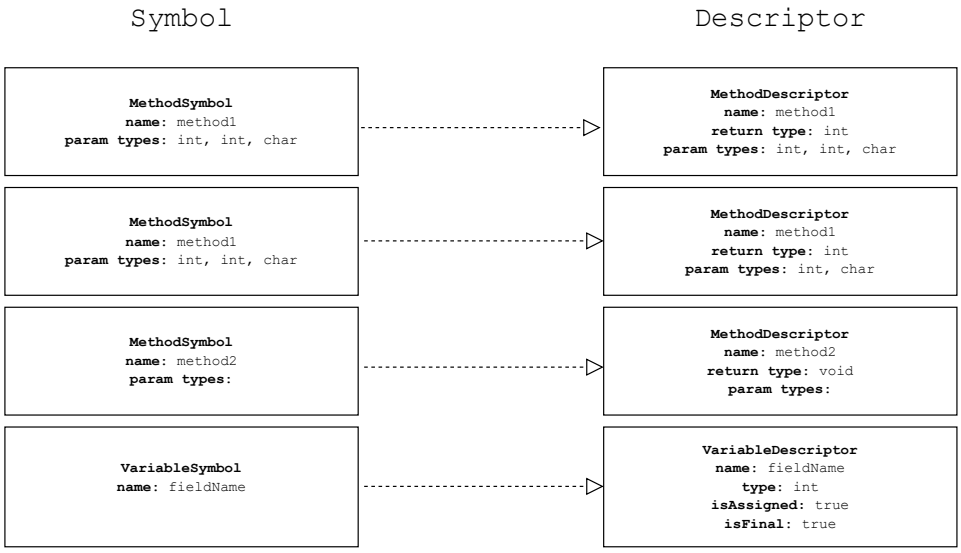
Tabulka symbolů je definována jako strom scopů (`AbstractScope`). Každý scope může být dvojího typu – třída (`ClassScope`) nebo block (`BlockScope`). Tyto druhy byly zvoleny z důvodu viditelnosti proměnných. Zatímco proměnná deklarovaná přímo ve třídě (nebo-li `field`) je viditelná po celé třídě, nehledě na to kde je deklarována, tak proměnná v bloku kódu je viditelná pouze po její deklaraci jen v tom bloku samotném (a v jeho podblocích).

Každý scope si drží svou tabulku (mapu) symbolů, které jsou deklarovány pro tento scope. Mimo jiné si drží také i seznam svých potomků (tj. podscopů), které obsahuje.

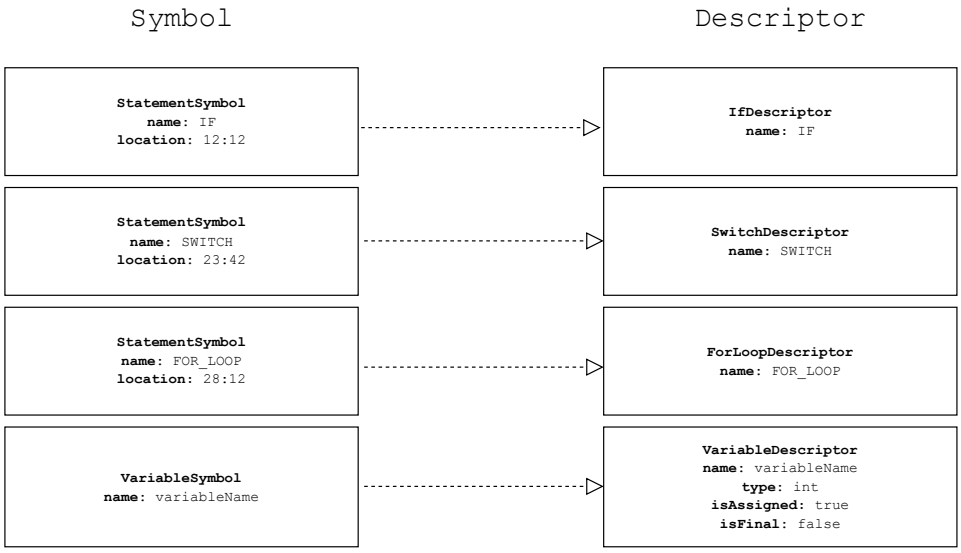


Obrázek 4.1: Diagram tabulky symbolů

Následující obrázek ukazuje příklad, jak jsou jednotlivé symboly uchovávány v tabulce symbolů pro `ClassScope`. Je zřejmé, že tato tabulka bude pouze uchovávat symboly pro metod a pro proměnné (textttfieldy. Každý tento symbol obsahuje svůj deskriptor, který přesněji poisuje co je daný element zač.



Na následujícím obrázku můžeme vidět vizualizaci implementace pro `BlockScope` – scope pro blok kódu. Lze si všimnout, že symbol může být jakýkoliv statement nebo jakákoliv proměnná. Deskriptor každého symbolu obsahuje pouze podrobnější data o symbolu.



Kapitola 5

Generování instrukcí

Hlavním cílem generování kódu je přeložit zdrojový program do posloupnosti instrukcí, které odpovídají logice původního programu a jsou spustitelné na cílovém stroji. Proces začíná transformací abstraktního syntaktického stromu (AST) nebo jiné mezijazykové reprezentace na konkrétní sekvenci instrukcí. Každý uzel AST odpovídá jedné nebo více instrukcím v PL/0, přičemž je nutné brát v úvahu jak syntaktický kontext (například bloky kódu), tak semantické vazby (například správné přiřazení proměnných nebo zachování zásobníkového rámce).

Instrukční sada PL/0 zahrnuje základní operace, jako je manipulace se zásobníkem, podmíněné a nepodmíněné skoky, aritmetické operace, přístup k lokálním i globálním proměnným a volání podprogramů. Generování kódu proto vyžaduje přesné řízení zásobníkového rámce, aby byly správně obsluhovány proměnné a návratové adresy. Při překladu výrazů se používá zásobníkový model, kde se operandy postupně ukládají na zásobník a operace jsou prováděny na hodnotách uložených na jeho vrcholu.

Pro generování latinské javy je zásadní cyklická struktura a podmíněné skoky. Kód musí zajistit iterativní generování hodnot podle pravidel latinské javy, což zahrnuje kontrolu, zda zadané hodnoty splňují podmínky uniqueness v řádcích a sloupcích. To se implementuje pomocí opakovaných smyček a logiky pro testování podmínek. Výsledný kód obsahuje jak inicializační sekvenci, tak sekvenci odpovídající hlavnímu algoritmu a výstupní část, která zajistí správné zobrazení výsledků.

Důležitým aspektem generování kódu je také minimalizace počtu instrukcí a zajištění efektivity, přičemž se dbá na to, aby výsledný kód byl správně interpretován cílovým strojem podle pravidel PL/0.

Kapitola 6

Spuštění

Spuštění programu zahrnuje 2 kroky. V prvním kroku je potřeba pomocí technologie `maven` sestavit projekt pomocí příkazu

```
mvn clean package
```

V tomto kroku byly vygenerovány potřebné zdrojové kódy pro správný chod `ANTLR` – třídy `IavaParserBaseVisitor`, `IavaParserBaseListener` a `IavaParser`. Tyto třídy se starají o vygenerování `AST`. V tomto kroku se také vytvoří `jar` soubor, který se dá spustit příkazem:

```
java -cp
target/FJP-semestralka-1.0-SNAPSHOT-jar-with-dependencies.jar
org.example.Main <zdrojovy_soubor.iava>
<vystupni_soubor_s_instrukcemi>
```

Ve druhém kroku je potřeba přidat parametry programu. Program očekává dva parametry:

```
<zdrojovy_soubor.iava> <vystupni_soubor_s_instrukcemi>
```

Bude-li předám jiný počet argumentů, program vypíše smysluplnou hlášku o špatném formátu argumentů.

Kapitola 7

Shrnutí

V rámci projektu se nám úspěšně podařilo implementovat sémantickou analýzu i generování kódu, přičemž jsme využili nástroj `ANTLR` pro zpracování vstupního programu. Díky této technologii jsme mohli efektivně definovat gramatiku, vytvořit syntaktický strom a provádět analýzu a transformaci zdrojového kódu do cílové instrukční sady.

Během implementace jsme se zaměřili na zpracování klíčových programových konstrukcí, včetně podmínek a všech běžných typů cyklů: `if`, `for`, `while`, `do-while`, a také `switch-case-default`. Dále se nám podařilo správně implementovat metody s parametry, podporu pro primitivní datové typy, řetězce, přiřazení hodnot a různé typy výrazů.

Díky této funkcionalitě náš překladač nyní dokáže plně přeložit program napsaný ve vstupním jazyce do cílové instrukční sady. Tím jsme vytvořili robustní a univerzální řešení, které je schopno zpracovávat širokou škálu programů a podporuje důležité aspekty moderního programování.

Implementované funkcionality

Datové typy a literály

- `char`: char literal
- `int`: hex literal, oct literal, binary literal
- `float`: float literal, hex float literal
- `boolean`: boolean literal

Deklarace proměnných

- hromadná deklarace s možností odložené inicializace (tj. např. `int a, b = 4, c = 5; a = b;`)

- deklarace konstant pomocí `finalis`

Výrazy

- unární: unární plus, unární minus, pre-increment, pre-decrement, post-increment, post-decrement, logická negace `!`, bitová negace `~`
- binární
  - aritmetické: `+`,`-`, `*`, `/`, `%`
  - relační: `==`,`!=`, `<`,`<=`,`>`,`>=`
  - logické: `&&`,`||`
  - přiřazovací: `=`, `+=`,`-=`,`*=`,`/=`,`%=` (včetně násobného přiřazení, tj. např. `a = b = c;`)
- ternární operátor
- přetypování, závorky, inicializátor, indexování pole

Přetypování

- `Implicitní`: char, int → float (ve výrazech, v argumentech metody, návratových hodnotách metody)
- `Explicitní` – implicitní + (float → int, char)

Volání metody

- Bezparametrické, s parametry
- Bez návratové hodnoty (volitelné použití `return` příkazu), návratovou hodnotou (lze ji ignorovat)
- Přetěžování metod

Příkazy

- `if`, `if-else`
- Smyčky `for` (včetně čárkovaných výrazů – `pro (x = 1, y = 1; x * y < 10; x++, y++)` – a prázdných výrazů – `pro (;;;)`, `while`, `do-while`
- `Switch` (s default case), bez použití `break` propadávající
- `break, continue`

Pole

- vytvoření pole o délce dané konstantou (`novus int[7]`) i proměnným výrazem (`novus int[a + 1]`)
- vytvoření pole výčtem prvků (konstant i výrazů) (`novus int[] {1, 2, 3, 4, a * 2};`)
- přístup k prvkům pomocí `[]`
- lze předávat do metod
- pole typu `float[]` není implementováno

Ostatní

- Kvalita zdrojového kódu – použití návrhových vzorů `strategie`, atd.
  - Vhodná dekompozice programu – `SRP`
  - žádné božské třídy
- Vedení `git` repozitáře
- Vypisování smysluplných syntaktických a sémantických chyb