# **VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

# **FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

# **Dokumentace projektu z předmětů IFJ a IAL**

# Implementace překladače imperativního jazyka IFJ23

**Tým xzelni06, varianta TRP-izp**

6.12.2023

|  |  |  |
| --- | --- | --- |
| Robert Zelníček | xzelni06 | 20% |
| Šimon Drábek | xdrabe10 | 15% |
| Ivan Chodák | xchoda00 | 50% |
| Dominik Šagát | xsagat01 | 15% |

## Obsah

[1. Práce v týmu 2](#_Toc152778277)

[2. Lexikální analýza 2](#_Toc152778278)

[3. Syntaktická analýza 2](#_Toc152778279)

[4. Sémantická analýza 2](#_Toc152778280)

[5. Tabulka symbolů 2](#_Toc152778281)

[5.1. Kolize a odstranění symbolu 2](#_Toc152778282)

[5.2. Parametry funkce 3](#_Toc152778283)

[6. Generování Kódu 3](#_Toc152778284)

[6.1. Rozhrání generátoru kódu 3](#_Toc152778285)

[6.2. Generování výrazů 3](#_Toc152778286)

[6.3. Generování přiřazení a definic proměnných 3](#_Toc152778287)

[6.4. Podmínky a cykly 3](#_Toc152778288)

[6.5. Vestavěné funkce 4](#_Toc152778289)

[7. Příloha 5](#_Toc152778290)

[7.1. Diagram stavového automatu 5](#_Toc152778291)

[7.2. Precedenční tabulka 5](#_Toc152778292)

[7.3. LL gramatika 6](#_Toc152778293)

[7.4. LL tabulka 7](#_Toc152778294)

## Práce v týmu

Nechám kapitánovi

### Rozdělení práce

**Robert Zelníček**

* Generování kódu

**Šimon Drábek**

* Lexikální analýza

**Ivan Chodák**

* Syntaktická analýza
* Sémantická analýza

**Dominik Šagát**

* Tabulka symbolů
* Dokumentace

## Lexikální analýza

Základem lexikální analýzy je načítání lexémů a jejich převádění na tokeny, které reprezentují daný lexém v dalších částech překladače. Lexikální analýza je implementována v souboru scanner.c, pracujícím s konečným stavovým automatem.

Tokenizování je provedeno ve funkci scan za pomocí funkce Switch, který rozhoduje, jaký token vytvořit podle aktuálního stavu stavového automatu.

Struktura tokenu obsahuje typ, kategorii a strukturu atributu. Tyto atributy poskytují potřebné informace pro token, jako jsou identifikátory, řetězce, celočíselné literály a desetinné literály. Za určitými tokeny mohou následovat znaky ? pro String, Int a Double nebo ! pro identifikátory. Tyto znaky určují, zda může hodnota být nilová nebo nikoliv, a v atributu je to určeno hodnotou nillable.

Třídy tokenů jsou rozděleny do kategorií závorek, operátorů, logických operátorů, interpunkce, klíčových slov, typů, identifikátorů, hodnot a chyb. Kategorie slouží k rychlému porovnání v dalších analýzách.

Více informací o lexikální analýze je k dispozici v příloze o stavovém automatu.

## Syntaktická analýza

## Sémantická analýza

## Tabulka symbolů

Tabulku symbolů jsme implementovali v souborech symtable.c a symtable.h, a to jako tabulku s rozptýlenými položkami s implicitním zřetězením. K rozptylovací funkci jsme využili násobení prvočíselnou konstantou 79 a operaci XOR s ASCII hodnotou každého znaku klíče. Samotná struktura symbolu obsahuje klíč, hodnotu enum, abychom věděli, zda se jedná o funkci, či proměnnou a dvě struktury, s tím že do jedné ukládáme atributy funkce a do druhé atributy proměnné. Každá tabulka je dynamicky alokovaná, to stejné platí o všech vložených symbolech. Aby se kód zbytečně neopakoval, tak existuje funkce Searching, která je téměř v každé další funkci.

### Práce s tabulkou symbolů

Na začátek se musí každá tabulka neinicializovat pomocí funkce SymTableInit, pro vkládání/odstranění symbolů se používá InsertSymbol/RemoveSymbol. Na vkládání atributů do symbolů se používá AddFuncitonDetails, respektive AddVarDetails.

### Kolize a odstranění symbolu

Jelikož se jedná o implicitní zřetězení, tak jsme vyřešili funkčnost tabulky, tak že odstraněnému symbolu se uvolní alokované zdroje, ale dojde pouze ke změně klíče na hodnotu „if“. Tuhle hodnotu nemůže mít jak proměnná, tak funkce. V případě více kolizí se tak lze dostat i přes odstraněný symbol k tomu hledanému.

### Parametry funkce

Pro práci s parametry funkcí jsme vytvořili zvlášť soubory parametrs.c a parametrs.h. S parametry se pracuje jako se zřetězeným seznamem. Samotná struktura obsahuje jméno, identifikátor a datový typ parametru.

## Generování Kódu

Kód je generován na základě Tokenů, jenž jsou mi předány z Parseru v podobě Kořenu spojového seznamu. Při zavoláni funkce start\_code\_generation dostane generátor kódu Kořen spojového seznamu a začne jej procházet. Pro každý validní Token vygeneruje jemu příslušící kód.

### Rozhrání generátoru kódu

Generování kódu je implementováno v souboru code\_generator.c a jeho rozhraní se nachází v souboru code\_generator.h. Rozhraní nabízí spoustu funkcí pro generování jednotlivých části programu, např.: deklarace proměnné, generování funkce, podmínek a cyklů atd. Všechny tyto se funkce se volají ve správnou chvíli po dokončení všech analýz.

### Generování výrazů

Funkce generate\_expresion\_code a generate\_binary\_arithmetic\_code jsou zodpovědné za generování kódu pro výrazy, včetně aritmetických operací. Výsledky těchto operací jsou ukládány do globálních proměnných na globálním rámci.

### Generování přiřazení a definic proměnných

Funkce generate\_assignment\_code a generate\_variable\_definition\_code zajišťují generování kódu pro přiřazení hodnoty proměnné a definici nových proměnných. Umožňují pracovat s datovými typy, včetně celých čísel, desetinných čísel a řetězců.

### Podmínky a cykly

Pro podmínky a cykly jsou implementované funkce generate\_if\_statement\_code, generate\_while\_loop\_code a další. Za zmínku stojí generování cyklu while. Tato funkce zajištuje kód pro inicializaci cyklu, vyhodnocení podmínky a provedení bloku kódu v rámci cyklu. Tím umožňuje správné opakování kódu, dokud je podmínka cyklu splněna.

### Vestavěné funkce

Generátor obsahuje implementace vestavěných funkcí, jako například konverze mezi datovými typy, načítání ze vstupu a další. Tyto funkce generují odpovídající části mezikódu IFJ23.

## Příloha

### Diagram stavového automatu

### Precedenční tabulka

### LL gramatika

1. <Prog> -> func <funcDef> <prog>
2. <Prog> -> <stmntList> <prog>
3. <Prog> -> EOF
4. <stmntList> -> e
5. <stmntList> -> <stmnt> <stmntList>
6. <stmnt> -> let/var <varDef>
7. <stmnt> -> id <FuncCallOps>
8. <stmnt> -> if <Condition>
9. <stmnt> -> while <Loop>
10. <stmnt> -> <return>
11. <paramListFirst> -> e
12. <paramListFirst> -> <param> <paramList>
13. <paramList> -> e
14. <paramList> -> , <param> <paramList>
15. <param> -> id id : <Type>
16. <param> -> \_ id : <Type>
17. <varDef> -> id <varDefOptional>
18. <varDefOptional> -> : <Type> <varDefAssTyped>
19. <varDefOptional> -> <varDefAss>
20. <varDefAssTyped> -> e
21. <varDefAssTyped> -> <varDefAss>
22. <varDefAss> -> = <AssignOps>
23. <funcDef> -> id ( <paramListFirst> ) <funcRetType> { <stmntList> <return> }
24. <funcRetType> -> -> <Type>
25. <funcRetType> -> e
26. <FuncCallOps> -> = <AssignOps>
27. <FuncCallOps> -> ( <InParamListFirst> )
28. <AssignOps> -> <expr>
29. <AssignOps> -> id ( <InParamListFirst> )
30. <Condition> -> <condOps> { <stmntList> } else { <stmntList> }
31. <condOps> -> <expr>
32. <condOps> -> let id
33. <Loop> -> <expr> { <stmntList>}
34. <InParamListFirst> -> e
35. <InParamListFirst> -> <InParam> <InParamList>
36. <InParamList> -> , <InParam> <InParamList>
37. <InParamList> -> e
38. <InParam> -> id <opID>
39. <InParam> -> literal
40. <opID> -> : id
41. <return> -> <expr>
42. <expre> -> give away to expressions parser to handle
43. <Type> -> Int/Int?
44. <Type> -> Double/Double?
45. <Type> -> String/String?

### LL tabulka

## Členění implementačního řešení

**parser.{c, h}**

* Lexikální analyzátor

**scaner.{c, h}**

* Syntaktický a sémantický analyzátor

**expressions.{c, h}**

* Zpracování výrazů

**symtable.{c, h}**

* Tabulka symbolů

**parametrs.{c, h}**

* Pomocné funkce na parametry funkcí k tabulce symbolů

**code\_generator.{c, h}**

* Generování kódu

**error.{c, h}**

* Hlášení chyb

**main.c**

* Hlavní program