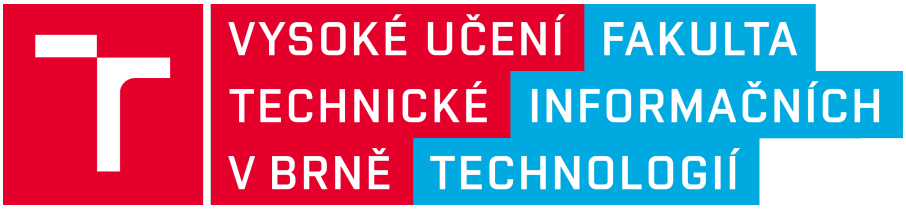
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



IFJ - Formálne jazyky a prekladače

Dokumentácia projektu - Prekladač jazyka IFJ24

30. november 2024 Rudolf Baumgartner (xbaumg01), Jakub Pogádl (xpogad00), Boris Semanco (xseman06), Igor Lacko (xlackoi00)

Obsah

1 Úvod

Tento dokument ako už názov napovedá je dokumentácia projektu z predmetu IFJ - Formálne jazyky a prekladače.Táto dokumentácia obsahuje popis implementácie jednotlivých častí prekladača, vrátane lexikálnej a syntaktickej analýzy, sémantickej analýzy a generovania kódu. Cieľom projektu je vytvoriť funkčný prekladač, ktorý dokáže preložiť zdrojový kód napísaný v jazyku IFJ24 do spustiteľného formátu.

2 Stručný popis zdrojových súborov

- `codegen.c, cocodegen.h`
- `conditionals.c, conditionals.h`
- `core-parser.c, core-parser.h`
- `embedded-functions.c, embedded-functions.h`
- `error.c, error.h`
- `expression-parser.c, expression-parser.h`
- `function-parser.c, function-parser.h`
- `loop.c, loop.h`
- `scanner.c, scanner.h`
- `shared.c, shared.h`
- `stack.c, stack.h`
- `symtable.c, symtable.h`
- `types.h`
- `vector.c, vector.h`

`codegen.h` a `codegen.c`: Tieto súbory obsahujú implementáciu generovania kódu pre jazyk IFJCode24.

`conditionals.h` a `conditionals.c`: Tieto súbory obsahujú implementáciu parsovania podmienok v programe. Obsahujú funkcie na spracovanie podmienkových výrazov a generovanie kódu pre podmienky.

`core-parser.h` a `core-parser.c`: Hlavný modul nášho prekladača. Obsahuje funkciu main, základ implementácie syntakctickej a sémantickej analýzy, spracovanie deklarácii a priradenia do premenných, či spracovanie argumentov funkcií.

`embedded-functions.h` a `embedded-functions.c`: Tieto súbory obsahujú implementáciu spracovania volaní vstavaných funkcií v jazyku IFJ. Obsahujú funkcie na identifikáciu, spracovanie

a generovanie kódu pre volania vstavaných funkcií, ako sú `readi32`, `write`, `concat`, `strcmp` a ďalšie.

`error.h` a `error.c`: Tieto súbory obsahujú implementáciu spracovania chýb v programe. Obsahujú funkcie na výpis chýb a definície makier, ktoré reprezentujú jednotlivé chybové kódy.

`expression-parser.h` a `expression-parser.c`: Tieto súbory obsahujú implementáciu precedenčnej syntaktickej analýzy zdola nahor, ktorá sa zameriava na spracovanie výrazov pomocou precedenčnej tabuľky. Taktiež sa tu nachádza následná sémantická analýza výrazov a následné generovanie zásobníkového kódu.

`function-parser.h` a `function-parser.c`: Tieto súbory obsahujú implementáciu prvého priechodu nášho prekladača. Ako názov napovedá, tento prístup sme zvolili hlavne kvôli funkciám, aby sme poznali dopredu ich návratové typy a argumenty. Tento modul však obsahuje aj isté predspracovanie premenných, o čom je napísané podrobnejšie v príslušnej sekcii.

`loop.h` a `loop.c`: Tieto súbory obsahujú implementáciu spracovania cyklov v programe, väčšinu syntakctickej a sémantickej analýzy pre cykly a generovanie kódu.

`scanner.h` a `scanner.c`: Tieto súbory obsahujú implementáciu lexikálnej analýzy. Taktiež ukladajú načítané tokeny do poľa tokenov ”stream”, z ktorého sa čerpá pri druhom priechode.

`shared.h` a `shared.c`: Tieto súbory obsahujú externé deklarácie premenných, ktoré sú zdieľané medzi rôznymi časťami programu. Obsahujú veci ako reťazce reprezentujúce rôzne kľúčové slová IFJ24, či pole tokenov ”stream”. Taktiež obsahujú jednotlivé ID náveští pre generovanie kódu.

`stack.h` a `stack.c`: Tieto súbory obsahujú implementáciu zásobníkových štruktúr, ktoré sa využívajú najmä pri výrazoch, ale aj pre kontrolu rozsahu (”scope”) programu pomocou zásobníka tabuliek symbolov.

`symtable.h` a `symtable.c`: Tieto súbory obsahujú implementáciu tabuľky symbolov, ktorá uchováva informácie o všetkých identifikátoroch v programe, vrátane (v prípade premenných) ich typov, názvov a prípadne aj hodnôt (ak sú známe pri preklade). V prípade funkcií príslušný symbol obsahuje parametre a ich typy, návratovú hodnotu, ale pre predídenie viacerých definícií pri generováni kódu aj názvy premenných ktoré sa v danej funkcii nachádzajú.

`types.h`: Tento súbor obsahuje definície dátových typov používaných v programe. Obsahuje definície štruktúr pre tokeny, symboly, zásobníky, štruktúru ”Parser”ktorá reprezentuje stav syntaktického analyzátora, precedenčnú tabuľku a ďalšie dátové typy potrebné pre implementáciu.

`vector.h` a `vector.c`: Tieto súbory obsahujú implementáciu dynamického poľa, ktoré sa používa na ukladanie a manipuláciu s dynamickými zoznamami prvkov. Vektor umožňuje dynamické pridávanie, odoberanie a prístup k prvkom, čo je užitočné pri správe tokenov, symbolov a ďalších dátových štruktúr.

3 Lexikálna analýza

Lexikálna analýza je časť prekladu, ktorá načítavá základné stavebné bloky programu nazývané tokeny. Tie ukladá pri prvom priechode do poľa tokenov ktoré je potom pri druhom "hlavnom"priechode pripravené pre syntaktickú/sémantickú analýzu. Taktiež rozpoznáva neznáme lexikálne jednotky a ukončuje program s vhodným chybovým kódom v takomto prípade.

3.1 Implementácia lexikálnej analýzy

Implementácia lexikálnej analýzy je obsiahnutá v súboroch `scanner.h` a `scanner.c`. Hlavné úlohy lexikálnej analýzy zahŕňajú:

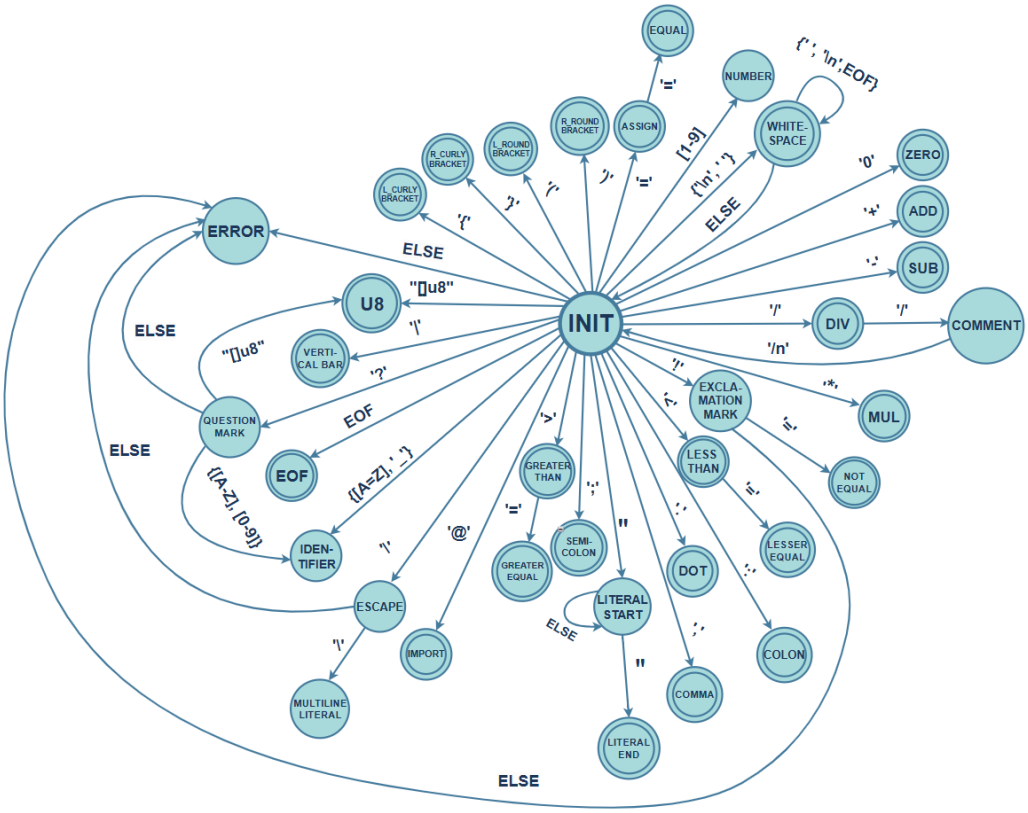
- **Rozpoznávanie tokenov:** Lexikálny analyzátor musí rozpoznať rôzne typy tokenov, ako sú kľúčové slová, identifikátory, operátory, literály a komentáre.
- **Rozlišovanie kľúčových slov od identifikátorov:** Jeden z problámov lexikálnej analýzy je rozdiel kľúčových slov a identifikátorov. Kľúčové slová sú rezervované slová jazyka, zatiaľ čo identifikátory sú názvy premenných, funkcií a iných entít definovaných používateľom. Tento problém sme vyriešili pomocou poľa, ktoré obsahuje názvy kľúčových slov jazyka IFJ24 a teda lexikálny analyzátor sa do neho pozerá stále po načítaní identifikátorov.
- **Rozpoznanie lexikálnych chýb**

Hlavné funkcie lexikálnej analýzy:

- `LoadTokenFromStream`: Načíta ďalší token zo vstupného prúdu, pričom preskakuje biele znaky (Teda zavolá funkciu ConsumeWhitespace(), ktorá ich preskočí)
- `InitToken`: Inicializuje nový token.
- `CopyToken`: Vráti kópiu tokenu, ktorý je odovzdaný ako parameter, aby sa predišlo dvojitému uvoľňovaniu pamäte pri vkladaní do tabuľky symbolov.
- `DestroyToken`: Zničí token a uvoľňuje pridelenú pamäť.
- `NextChar`: Vráti ďalší znak zo vstupu bez posunutia dopredu (vráti znak späť).
- `GetCharType`: Získa typ ďalšieho znaku (biely znak, písmeno, číslo), v prípade že automat sa nevie deterministicky rozhodnúť na jeho konkrétnej hodnote
- `ConsumeNumber`: Spracováva číselný literál a nastavuje atribút tokenu na hodnotu čísla (double/-float).
- `ConsumeExponent`: Pomocná funkcia pre `ConsumeNumber`, ktorá spracováva exponent čísla.
- `ConsumeLiteral`: Spracováva reťazcový literál.
- `DoesMultiLineLiteralContinue`: Preskakuje biele znaky, kým nenarazí na ďalší nebiely znak.

- `ConsumeHexadecimalEscapeSequence`: Spracováva a validuje escape sekvenciu vo forme `\xHH` v literáli.
- `ConsumeMultiLineLiteral`: Spracováva viacriadkový reťazcový literál.
- `ConsumeIdentifier`: Spracováva identifikátor alebo kľúčové slovo.
- `ConsumeImportToken`: Spracováva token pri narazení na znak `@`.
- `ConsumeComment`: Spracováva komentáre a zvyšuje číslo riadku, keď narazí na znak nového riadku.
- `ConsumeWhitespace`: Preskakuje biele znaky, zvyšuje číslo riadku pri narazení na znak nového riadku a vráti prvý ne-biely znak.
- `ConsumeU8Token`: Spracováva špeciálny prípad tokenu `[]u8` a kontroluje, či je `u8` kľúčové slovo.
- `IsValidPrefix`: Kontroluje, či je identifikátor s predponou '?' platný alebo nie (či je to kľúčové slovo).
- `IsKeyword`: Kontroluje, či je daný reťazec kľúčové slovo.
- `PrintToken`: Debugovacia funkcia na tlač tokenu.

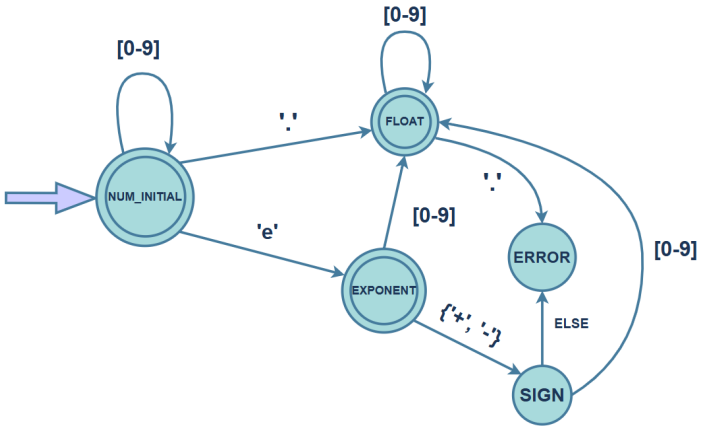
3.2 Diagram celkového konečného automatu pre lexikálnu analýzu



Automat celkovej lexikálnej analýzy.

3.3 Diagram pre čísla

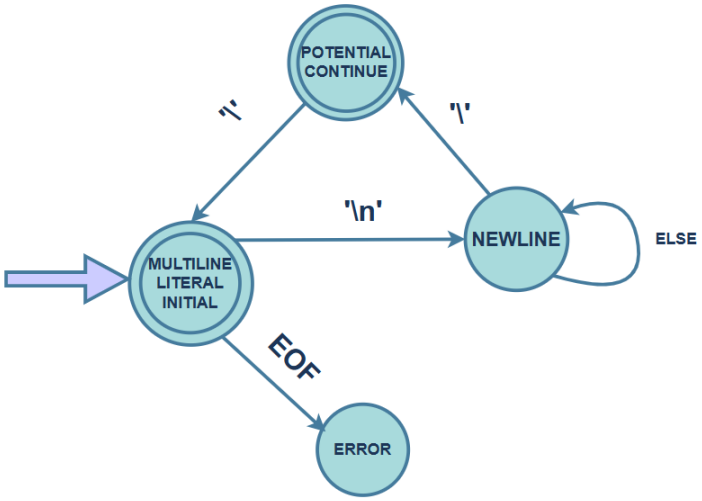
Diagram pod-automatu konečného automatu pre lexikálnu analýzu, do ktorého sa dostane priechodom do stavu ”NUMBER”. Tento automat spracováva číselné literály



Pod-automat lexikálnej analýzy, ktorý spracováva číselné literály (čísla)

3.4 Diagram pre viacriadkové reťazcové reťazcové

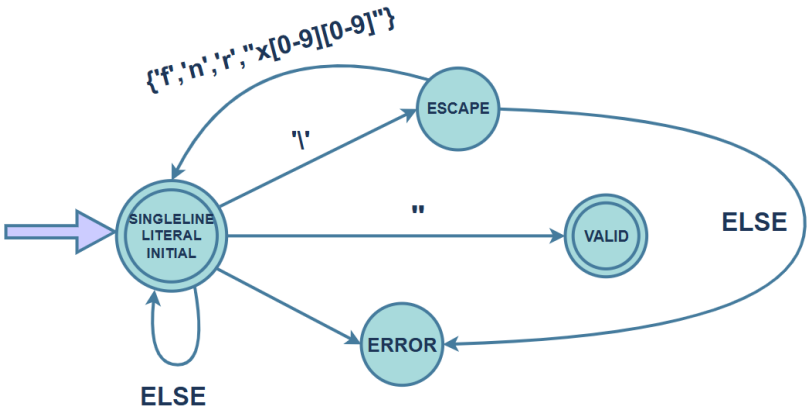
Diagram pod-automatu konečného automatu pre lexikálnu analýzu, do ktorého sa dostane priechodom do stavu ”MULTILINE LITERAL”. Tento automat spracováva viacriadkové reťazcové literály.



Pod-automat lexikálnej analýzy, ktorý spracováva viacriadkové reťazcové literály

3.5 Diagram pre jednoriadkové reťazcové literály

Diagram pod-automatu konečného automatu pre lexikálnu analýzu, do ktorého sa dostane priechodom do stavu ”SINGLELINE LITERAL”. Tento automat spracováva jednoriadkové reťazcové literály.



Pod-automat lexikálnej analýzy, ktorý spracováva jednoriadkové reťazcové literály

4 Syntaktická analýza

Syntaktická analýza je hlavná fáza prekladu, ktorá kontroluje či je vztupný reťazec generovaný danou LL-gramatikou. Pre implementáciu sme zvolili metódu rekurzívneho zostupu, ktorej "mozog"je funkcia `ProgramBody()`. Pri analýze výrazov sa tento prístup obmieňa a je využitá analýza zdola-nahor založená na precedenčnej tabuľke.

4.1 Popis implementácie syntaktickej analýzy zhora nadol

Implementácia syntaktickej analýzy zhora nadol je obsiahnutá v súboroch `core-parser.h`, `core-parser.c` `function-parser.h`, `function-parser.c`, `loop.h`, `loop.c` a `conditional.h`, `conditional.` Hlavné úlohy syntaktickej analýzy zahŕňajú:

- **Spracovanie definícií funkcií:** Syntaktický analyzátor musí rozpoznať a spracovať definície funkcií, vrátane ich názvov, návratových typov a parametrov. To môže byť problematické ak sú funckie definované neskôr v kóde ako sú volané, pre to sma zvlolili implementáciu dvojprie-chodového prekladača.
- **Spracovanie deklarácií či priradení do premenných:** Syntaktický analyzátor musí rozpoznať a spracovať deklarácie či priradenia do premenných v rámci funkcií a v prípade novej premennej vytvoriť položku v tabuľke symbolov.
- **Spracovanie cyklov či podmienok:** Syntaktický analyzátor musí rozpoznať a spracovať cykly `while` a podmienky `if else`.

Hlavné funkcie syntaktickej analýzy zhora nadol:

- `ParseFunctionDefinition`: Funkcia volaná pri prvom priechode prekladača po detekovaní kľúčového slova 'pub'. Funkcia dopredu zistí parametre (typy, názvy, počet) funkcie a uchová tieto informácie do globálnej (na funkcie) tabuľky symbolov. Prebieha tu aj syntaktická kontrola (teda prítomnosť zátvoriek, či parametre majú daný typ...).
- `ParseParameters`: Pomocná funkcia `ParseFunctionDefinition`. Prejde cez všetky parametre a hodí ich do štruktúry funkcie.
- `ParseVariableDeclaration`: Funkcia detekuje pri priechode deklaráciu novej premennej (prítomnosť tokenu var/const) a uchová názov tejto premennej do štruktúry funkcie, v ktorej rozsahu sa táto premenná nachádza. Tím sa predídu viacnásobné príkazy DEFVAR pre rovnakú premennú (napríklad pri cykloch), kedže všetky premenné sa definujú na začiatku funkcie.
- `ParseWhileLoop`: Spracuje cyklus typu while(pravdivostný výraz). Kopntroluej či výraz je sémanticky správny (volá expression parser) a pomocou rekurzívneho zostupu sa zanorí do jeho tela a vygeneruje kód pre podmienené skoky.
- `ParseNullableWhileLoop`: Spracuje typ while cyklu while(výraz s null) —ID—. Funguje veľmi podobne ako štandardný while cyklus, no generuje trochu iný kód a vkladá novú premennú do tabuľky symbolov.

- `ParseIfStatement`: Spracuje základnú konštrukciu typu if, teda typ if(pravdivostný výraz) else. Funguje podobne ako analýza cyklov, zanruje sa však dvakrát a generuje iný kód.

- `CheckTokenTypeVector`: Kontroluje, či ďalší token zodpovedá očakávanému typu tokenu, inak ukončí program. Táto varianta kontroluje ďalší token v poli stream (pomocou zdieľanej globálnej premennej `stream-index`, ktorá určuje aktuálnu pozíciu v poli a táto funkcia ju posunie na ďalšiu pozíciu.

- `CheckKeywordTypeVector`: Verzia `CheckTokenTypeVector`, ktorá kontroluje ale kľúčové slovo (napríklad po konci if bloku očakávame else, atď ...)

- `CheckAndReturnTokenVector`: Verzia `CheckTokenTypeVector`, ktorá daný token vráti.

- `CheckTokenTypeStream`: Funguje podobne ako `CheckTokenTypeVector`, ale vstupný token berie z vstupného programu. Zároveň v prípade úspechu vloží vstupný token do poľa `stream`. Obdobne existujú funkcie `CheckKeywordTypeStream` a `CheckAndReturnTokenStrea`

4.2 Popis implementácie syntaktickej analýzy zdola nahor

Implementácia syntaktickej analýzy zdola nahor je obsiahnutá v súboroch `expressionparser.h` a `expressionparser.c` . Táto analýza sa zameriava na spracovanie výrazov pomocou precedenčnej tabuľky. Hlavné úlohy syntaktickej analýzy zdola nahor zahŕňajú:

- **Konverzia infixovej notácie na postfixovú notáciu:** Pomocou precedenčnej analýzy a vkladaním neterminálov pravej strany redukovaných pravidiel na koniec výstupného reťazca sa výraz prevedie na postfixový reťazec, ktorý je jednodušší na vyhodnotenie (v tomto prípade na generovanie kódu).

- **Nahrádzanie premenných hodnotami známe počas kompilácie:** Premenné sú nahradené ich hodnotami, ak sú tieto hodnoty známe počas kompilácie.

- **Kontrola kompatibility operandov:** Syntaktický analyzátor kontroluje, či sú operandy kompatibilné pre dané operácie.

- **Generovanie kódu pre aritmetické a logické operácie:** Syntaktický analyzátor generuje kód pre aritmetické a logické operácie medzi operandmi.

Hlavné funkcie syntaktickej analýzy zdola nahor

- `InfixToPostfix`: Hlavná funkcia precedenčnej analýzy. Zároveň, ako názov napovedá, konvertuje výraz v infixovej notácii na postfixovú notáciu. V prípade že vstupný výraz není valídny, ukončí program s vhodnou chybovou hláškou.

- `ReplaceConstants`: Nahrádza premenné hodnotami, ktoré sú známe počas kompilácie.

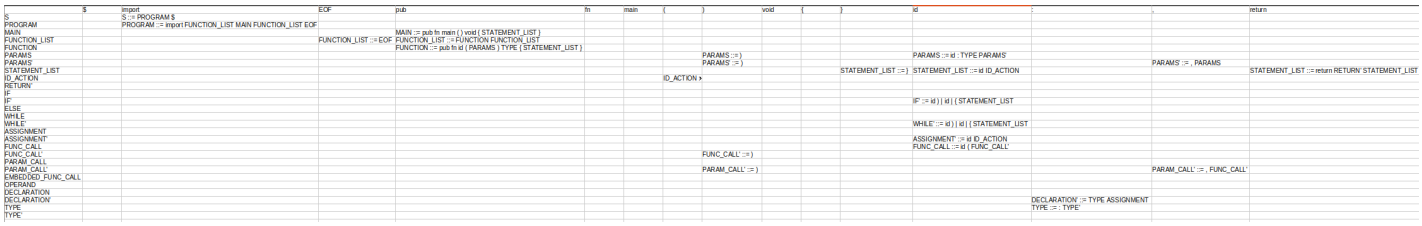
4.3 LL gramatika

```
<program> -> import <function-list> <main> <function-list>
<main> -> pub fn main ( ) void { <statement-list>
<function-list> -> <function> <function-list>
<function-list> -> $
<function> -> pub fn id ( <params> <type> { <statement-list>
<params> -> id : <type> <params'>
<params> -> )
<params'> -> , <params>
<params'> -> )
<statement-list> -> <if> <statement-list> <else> <statement-list>
<statement-list> -> <while> <statement-list>
<statement-list> -> <declaration> <statement-list>
<statement-list> -> id <id-action>
<statement-list> -> return <return'> <statement-list>
<statement-list> -> <embedded-func-call> <statement-list>
<statement-list> -> }
<id-action> -> <assignment>
<id-action> -> ( <func-call'>
<return'> -> <expression>
<return'> -> ;
<if> -> if(<if'>
<if'> -> id) |id| { <statement-list>
<if'> -> <expression>){ <statement-list>
<else> -> else { <statement-list>
<while> -> while( <while'>
<while'> -> <expression> ) { <statement-list>
<while'> -> id) |id| { <statement-list>
<assignment> -> = <assignment'>
<assignment'> -> <operand> ;
<assignment'> -> id;
<assignment'> -> <expression>
<assignment'> -> <embedded-func-call>
<func-call> -> id( <func-call'>
<func-call'> -> )
<func-call'> -> <param-call>
<param-call> -> <operand> <param-call'>
<param-call'> -> )
<param-call'> -> ,<func-call'>
<embedded-func-call> -> ifj.<func-call>
<operand> -> int
<operand> -> float
<operand> -> literal
<operand> -> null
<declaration> -> const id <declaration'>
<declaration> -> var id <declaration'>
```

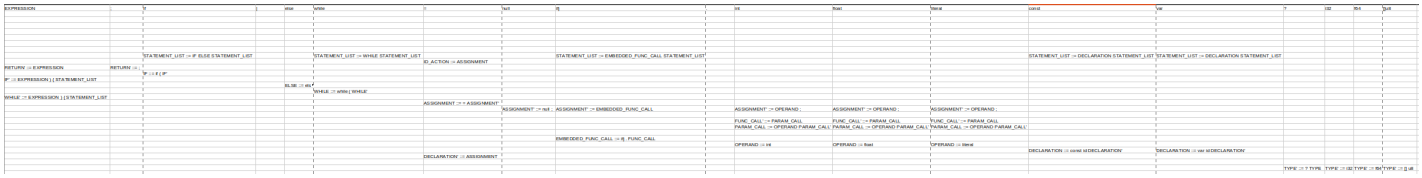
```
<declaration’> -> <assignment>
<declaration’> -> <type> <assignment>
<type> -> : <type’>
<type’> -> i32
<type’> -> f64
<type’> -> []u8
<type’> -> ?<type’>
```

4.4 LL tabuľka

odkaz na online tabuľku



Popis pre LL-tabuľku, časť 1.



Popis pre LL-tabuľku, časť 2.

4.5 Precedenčná tabuľka

1/2	id	$*$	$/$	$+$	$-$	$==$	$!=$	$<$	$>$	$<=$	$>=$	$($	$)$	$
id	NULL	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	NULL	$>$	$>$
$*$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
$/$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
$+$	$<$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
$-$	$<$	$<$	$<$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$<$	$>$	$>$
$==$	$<$	$<$	$<$	$<$	$<$	NULL	NULL	NULL	NULL	NULL	NULL	$<$	$>$	$>$
$!=$	$<$	$<$	$<$	$<$	$<$	NULL	NULL	NULL	NULL	NULL	NULL	$<$	$>$	$>$
$<$	$<$	$<$	$<$	$<$	$<$	NULL	NULL	NULL	NULL	NULL	NULL	$<$	$>$	$>$
$>$	$<$	$<$	$<$	$<$	$<$	NULL	NULL	NULL	NULL	NULL	NULL	$<$	$>$	$>$
$<=$	$<$	$<$	$<$	$<$	$<$	NULL	NULL	NULL	NULL	NULL	NULL	$<$	$>$	$>$
$>=$	$<$	$<$	$<$	$<$	$<$	NULL	NULL	NULL	NULL	NULL	NULL	$<$	$>$	$>$
$($	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$=$	NULL
$)$	NULL	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	$>$	NULL	$>$	$>$
$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	$<$	NULL	success

5 Sémantická analýza

Sémantická analýza v našom projekte prebieha súčasne so syntaktickou analýzou a zameriava sa na kontrolu správnosti významu programu. Cieľom je overiť správnosť deklarácií, definícií a kompatibilitu typov v celom programe.

5.1 Implementácia sémantickej analýzy

V rámci sémantickej analýzy vykonávame kontroly deklarovaných symbolov a výrazov. Symboly (premenné a funkcie) sú uchovávané v tabuľke symbolov, ktorá umožňuje efektívne vyhľadávanie a správu symbolov počas analýzy. Pri spracovávaní kódu kontrolujeme, či sú symboly správne deklarované a definované pred ich použitím.

Výrazy sú spracovávané generovaním zásobníkového kódu pomocou postfixového zápisu a algoritmom na vyhodnocovanie postfixového zápisu [1]. Tento spôsob nám umožňuje efektívne vykonávať operácie nad výrazmi a zároveň kontrolovať typy operátorov a argumentov. Ak sa vyskytne nesúlad v typoch (napr. pokus o operáciu s nekompatibilnými typmi), generujeme sémantickú chybu.

Okrem toho sa kontrolujú aj parametre funkcií, aby boli v súlade s ich deklaráciami. Kontrola zahŕňa overenie počtu a typu parametrov, ako aj návratového typu funkcie.

6 Generovanie kódu

Generovanie kódu je fáza kompilácie, ktorá prekladá syntaktický strom alebo iné medzireprezentácie programu do cieľového jazyka, v tomto prípade do jazyka IFJCode24.

6.1 Implementácia generovanie kódu

Implementácia generovania kódu je hlavne obsiahnutá v súboroch codegen.h a codegen.c, ale nejaký kód je generovaný aj v ostatných súboroch (napríklad skoky pri podmienkach a cykloch)

- **Generovanie hlavičky programu:** Vytvorenie hlavičky programu pre IFJCode24.
- **Definícia premenných:** Generovanie kódu pre definíciu premenných v rôznych rámcoch (globálny, lokálny, dočasný).
- **Generovanie kódu pre funkcie:** Generovanie kódu pre definície a volania funkcií.
- **Generovanie aritmetických a logických operácií:** Generovanie kódu pre aritmetické a logické operácie.
- **Generovanie kódu pre vstupno-výstupné operácie:** Generovanie kódu pre operácie čítania a zápisu.
- **Generovanie podmienok a cyklov:** Generovanie kódu pre podmienky a cykly.

Hlavné funkcie generovania kódu:

- `GeneratePostfixExpression`: Generuje zásobníkový IFJ24 kód pre daný výraz v postfixovej forme. Zároveň.

- `IntExpression`: Generuje kód pre celočíselný výraz daný operátorom.
- `InitRegisters`: Inicializuje registre a vytlačí IFJCode24 pre definovanie niektorých globálnych registrov.
- `DefineVariable`: Definuje premennú v IFJCode24, v podstate len pekne pomenovaný wrapper pre `fprintf`.
- `IfLabel`: Generuje štítok pre `if` podmienku.
- `ElseLabel`: Generuje štítok pre `else` vetvu.
- `EndIfLabel`: Generuje štítok pre koniec `if` podmienky.
- `WhileLabel`: Generuje štítok pre začiatok `while` cyklu.
- `EndWhileLabel`: Generuje štítok pre koniec `while` cyklu.
- `MOVE`: Generuje kód pre presun hodnoty z `src` do `dst`.
- `PUSHS`: Generuje kód pre vloženie symbolu na dátový zásobník.
- `SETPARAM`: Generuje kód pre presun hodnoty do parametra funkcie.
- `READ`: Volá inštrukciu `READ` na čítanie symbolu určitého typu do premennej.
- `WRITEINSTRUCTION`: Volá inštrukciu `WRITE` na zápis symbolu do výstupu.
- `INT2FLOAT`: Konverzia z celého čísla na desatinné číslo.
- `FLOAT2INT`: Konverzia z desatinného čísla na celé číslo.
- `STRLEN`: Volá inštrukciu `STRLEN` na získanie dĺžky reťazca.
- `CONCAT`: Volá inštrukciu `CONCAT` na zreťazenie dvoch reťazcov.
- `STRI2INT`: Volá inštrukciu `STRI2INT` na konverziu reťazca na celé číslo.
- `INT2CHAR`: Volá inštrukciu `INT2CHAR` na konverziu celého čísla na znak.
- `STRCMP`: Generuje kód pre vstavanú funkciu `ifj.strcmp`.
- `STRING`: Generuje kód pre vstavanú funkciu `ifj.string`.
- `ORD`: Generuje kód pre vstavanú funkciu `ifj.ord`.
- `SUBSTRING`: Generuje kód pre vstavanú funkciu `ifj.substring`.
- `WriteStringLiteral`: Zapíše reťazcový literál v kompatibilnom formáte pre IFJCode24.
- `PushRetValue`: Vloží návratovú hodnotu funkcie na dátový zásobník podľa typu návratovej hodnoty.

7 Popis vývoja

V tejto sekcii bude popísaná naša implementácia projektu.

7.1 Rozdelenie práce v tíme

Rozdeľovanie práce sme rozdelili tak, aby každému členovi tímu bola priradená časť, ktorú vedel najlepšie. Ostatné, ktoré zostali, sme si buď rozdelili sami, alebo sme na tom pracovali všetci zároveň. V nasledújucej tabulke bude naše rozdelenie práce:

Meno	Práca
Rudolf Baumgartner (xbaumg01)	testy, dokumentácia, generovanie kódu, štruktúry
Igor Lacko (xlackoi00)	lexikálna analýza, sémantická analýza, syntaktická analýza, generovanie kódu
Jakub Pogádl (xpogad00)	prvý priechod prekladača, štruktúry, veducko
Boris Semanco (xseman06)	syntaktická analýza, tabuľka symbolov, errors

7.2 Komunikácia v tíme

Náš tím pozostáva zo skupiny kamarátov, čo významne prispelo k dobrej spolupráci a príjemnej atmosfére počas celého projektu. Preferovali sme osobné stretnutia, kde sme mohli na projekte pracovať spoločne a efektívnejšie. Takéto stretnutia nám umožnili rýchlo riešiť nejasnosti a navzájom si pomáhať.

V prípadoch, keď nebolo možné stretnúť sa osobne, sme komunikovali prostredníctvom Discordu alebo Instagram-u. Platformy nám umožnili udržať kontakt, koordinovať prácu a diskutovať nejasnostio. Ako už vyššie spomenuté, sme uprednostňovali osobné stretnutia, pretože priame kódovanie naživo podporovalo lepšie pochopenie problému a rýchlejšiu implementáciu riešení.

8 Použité datové štruktúry

8.1 Parser

Popis: Štruktúra Parser je kľúčová pre správu stavu počas parsovania zdrojového kódu. Obsahuje informácie o aktuálnej pozícii v kóde, aktuálnej funkcii, tabuľkách symbolov a ďalšie dôležité údaje potrebné na správne spracovanie a analýzu kódu.

Atribúty:

- **nestedlevel**: Aktuálna úroveň vnorených blokov kódu. Pomáha sledovať, koľko úrovní vnorenia sa v kóde nachádza.
- **linenumber**: Aktuálne číslo riadku, ktorý sa práve spracováva. Umožňuje sledovať pozíciu v zdrojovom kóde.
- **hasmain**: Indikuje, či bola nájdená hlavná funkcia (main). Používa sa na overenie, či program obsahuje vstupný bod.

- **currentfunction**: Ukazaateľ na aktuálnu funkciu, ktorá sa práve parsuje. Umožňuje prístup k informáciám o funkcii počas jej spracovávania, napríklad pre sémantické kontrolz pri príkaze return.

- **symtable**: Aktuálna tabuľka symbolov (vrchná na zásobníku), ktorá obsahuje lokálne premenné. Používa sa na ukladanie a vyhľadávanie symbolov počas spracovávania, taktiež napríklad pre sémantické kontroly (či je premenná redefinovaná, atď ...)

- **globalsymtable**: Globálna tabuľka symbolov, ktorá obsahuje funkcie. Používa sa na ukladanie a vyhľadávanie globálnych symbolov (ktorými sú len funkcie, kedže IFJ24 obsahuje len lokálne premenné)

- **symtablestack**: Zásobník aktuálnych tabuliek symbolov. V zásade reprezentuje aktuálny rozsah programu, čo je taktiež užitočné pre sémantické kontroly (napríklad či je premenná definovaná, atď ...)

8.2 Symtable

Popis: Tabuľka symbolov, ktorá uchováva symboly (premenné a funkcie) v hashovacej tabuľke. Na implementáciu sme použili tabuľku s rozptýlenými položkami a implicitným zreťazením. Pre vyhľadávanie a vkladanie symbolov sa používa hashovací algoritmus (variant sdbm) [2], ktorý vypočíta index do tabuľky na základe mena symbolu. Pri používaní hashovacej tabuľky môže nastať situácia, kedy dva symboly majú rovnaký hash a teda aj rovnaký index. Pre tento prípad sa používa otvorené adresovanie, kde sa prehľadávajú nasledujúce pozície v tabuľke (lineárne), kým sa nenájde voľné miesto alebo sa nenájde existujúci symbol.

Atribúty:

- `capacity`: Kapacita tabuľky (maximálny počet položiek). Použili sme číslo 5009 (prvočíslo)

- `size`: Aktuálny počet obsadených položiek.

- `table`: Pole hashovacích položiek (`HashEntry`).

8.3 HashEntry

Popis: Použitý hashovací algoritmus je variant sdbm, ktorý sa ukázal ako efektívny na vytváranie distribúcií hodnôt hashov. Tento algoritmus pre každý znak mena symbolu vypočíta nový hash, ktorý sa nakoniec moduluje veľkosťou tabuľky, aby sa získal platný index. Každý symbol v tabuľke je reprezentovaný ako záznam (entry) v poli, kde každý záznam obsahuje:

Atribúty:

- `symbol-type`: Typ symbolu (funkcia alebo premenná).

- `symbol`: Ukazovateľ na symbol (premenná alebo funkcia).

- `is-occupied`: Boolean hodnota, ktorá indikuje, či je položka obsadená.

8.4 FunctionSymbol

Popis: Štruktúra reprezentujúca funkciu. Uchovaná v globálnej tabuľke symbolov `Parser` štruktúry.
Atribúty:

- `name`: Názov funkcie.
- `num-of-parameters`: Počet parametrov funkcie.
- `parameters`: Pole ukazovateľov na symboly premenných (parametre funkcie).
- `return-type`: Návratový typ funkcie.
- `has-return`: Boolean hodnota, ktorá indikuje, či funkcia má návratovú hodnotu.
- `variables`: Pole reťazcov (premenné vo funkcii).

8.5 VariableSymbol

Popis: Štruktúra reprezentujúca premennú.
Atribúty:

- `name`: Názov premennej.
- `type`: Typ premennej.
- `is-const`: Índikuje, či je premenná konštanta.
- `nullable`: Indikuje, či je premenná nullable (môže byť null).

8.6 SymtableStack

Popis: Zásobník tabuliek symbolov.
Atribúty:

- `size`: Veľkosť zásobníka.
- `top`: Ukazovateľ na vrchol zásobníka (najvyššia symbolická tabuľka).

8.7 Token

Popis: Štruktúra reprezentujúca token (lexikálnu jednotku).
Atribúty:

- `token-type`: Typ tokenu.
- `keyword-type`: Typ kľúčového slova, ktoré token reprezentuje. Ak neprezentuje žiadne kľúčové slovo, poožije sa hodnota NONE (pomocný dátový typ enum)
- `attribute`: Textová/reťazcová reprezentácia tokenu
- `line-number`: Číslo riadku, kde sa token nachádza.

8.8 TokenVector

Popis: Dynamické pole tokenov. Inicializované pri prvom priechode,
Atribúty:

- `token-string`: Pole ukazovateľov na tokeny.
- `length`: Aktuálna dĺžka poľa.
- `capacity`: Kapacita poľa.

8.9 StringArray

Popis: Dynamické pole reťazcov.
Atribúty:

- `strings`: Pole reťazcov.
- `count`: Počet reťazcov v poli.
- `capacity`: Kapacita poľa.

Tieto dátové štruktúry sú základom pre správu symbolov, tokenov a ďalších prvkov.

9 Záver

V tomto projekte sme vytvorili funkčný prekladač jazyka IFJ24, ktorý zahŕňa fázy lexikálnej, syntaktickej a sémantickej analýzy, ako aj generovanie medzikódu. Lexikálna analýza identifikuje tokeny, syntaktická analýza overuje gramatiku a sémantická analýza zabezpečuje konzistenciu typov a deklarácií. Po spracovaní kódu sme generovali medzikód, ktorý je spustiteľný interpretom. Tento prekladač správne identifikuje chyby a zabezpečuje efektívne spracovanie kódu jazyka IFJ24, čo nám poskytlo cenné skúsenosti v tvorbe prekladačov.

10 Použité zdroje

[1] VUT FIT. *Abstraktní datové typy II. Vyčíslenie postfixového výrazu*. citované: 2024-12-02. 2024. URL: `http://www.vut.cz`.

[2] Unknown. *sdbm Hash Function*. Web. citované: 2024-12-02. URL: `http://www.cse.yorku.ca/~oz/hash.html`.