

Implementace překladače

imperativního jazyka IFJ20

Projektová dokumentace

|  |  |  |
| --- | --- | --- |
|  | **Tým 053, varianta I** |  |
|  | Jana Stopková (xstopk01) – vedoucí | 40 % |
| 3. prosince 2020 | Filip Vágner (xvagne08) | 40 % |
|  | Tomáš Matušák (xmatus34) | 20 % |
|  | Kovář Jiří (xkovar90) | 0 % |

Obsah

[1 Úvod 3](#_Toc58270357)

[2 Práce v týmu 3](#_Toc58270358)

[2.1 Verzovací systém 3](#_Toc58270359)

[2.2 Komunikace 3](#_Toc58270360)

[2.3 Rozdělení práce mezi členy týmu 3](#_Toc58270361)

[2.4 Zdůvodnění odchylek od rovnoměrného rozdělení bodů 3](#_Toc58270362)

[3 Řešení projektu 4](#_Toc58270363)

[3.1 Lexikální analýza 4](#_Toc58270364)

[3.2 Syntaktická analýza 6](#_Toc58270365)

[3.2.1 Syntaktická analýza založená na rekurzivním sestupu 6](#_Toc58270366)

[3.2.2 Precedenční syntaktická analýza 6](#_Toc58270367)

[3.3 Sémantická analýza 7](#_Toc58270368)

[3.4 Generování kódu 7](#_Toc58270369)

[3.4.1 Generování jmen proměnných 7](#_Toc58270370)

[3.5 Použité datové struktury 7](#_Toc58270371)

[3.5.1 Token 7](#_Toc58270372)

[3.5.2 Tabulka symbolů 8](#_Toc58270373)

[3.5.3 Zásobník tabulek symbolů 8](#_Toc58270374)

[3.5.4 Lineární seznam 9](#_Toc58270375)

[4 Závěr 9](#_Toc58270376)

[5 Použitá literatura a reference 9](#_Toc58270377)

# Úvod

Cílem projektu bylo vytvořit program, který načítá vstupní kód v jazyce IFJ20 a překládá jej do mezikódu IFJcode20 s provedením syntaktických a sémantických kontrol.

# Práce v týmu

Po získání základních znalostí potřebných k práci na projektu jsme si rozdělili úkoly. Nejprve jsme navrhli a vytvořili tabulku symbolů a LL gramatiku. Následně jsme současně začali pracovat na syntaktické analýze založené na rekurzivním sestupu a precedenční syntaktické analýze. Scanner jsme začali navrhovat až později. Poslední částí bylo generování kódu.

## Verzovací systém

Jako verzovací systém jsme použili git, protože s ním máme nejvíce zkušeností. Za hosting jsme zvolili GitHub.

## Komunikace

Jako hlavní komunikační platforma našeho týmu byl zvolen Slack, jelikož se nám již dříve osvědčil při práci na projektu z předmětu IVS (Praktické aspekty vývoje software). K jednomu z kanálů jsme připojili aplikaci GitHub, která nás vždy informovala o tom, když někdo z týmu vytvořil nový commit. Osobní schůzky jsme vzhledem k distanční výuce nahradili hovory přes aplikaci Discord.

## Rozdělení práce mezi členy týmu

|  |  |
| --- | --- |
| Jana Stopková (xstopk01) | vedení týmu, tabulka symbolů, syntaktická a sémantická analýza bez výrazů, generování kódu |
| Filip Vágner (xvagne08) | lexikální analýza, syntaktická a sémantická analýza pro výrazy, generování kódu |
| Tomáš Matušák (xmatus34) | lexikální analýza, testování, dokumentace |

Tabulka 1 - Rozdělení práce mezi členy týmu

## Zdůvodnění odchylek od rovnoměrného rozdělení bodů

Posledního člena týmu jsme do tabulky Tabulka 1 nezahrnuli z toho důvodu, že se nám s ním po jeho přidání do týmu nepodařilo pomocí školního emailu i přes opakované pokusy spojit, tj. na žádný nám neodpověděl, ani nikomu z nás nenapsal a jiný prostředek komunikace jsme s ním neměli.

Tomáš Matušák má oproti zbytku bodů méně, protože se zapojil až v pozdější fázi a programování syntaktické ani sémantické analýzy se neúčastnil, také se nezapojoval do diskusí ohledně řešení různých problémů, na které jsme během projektu narazili.

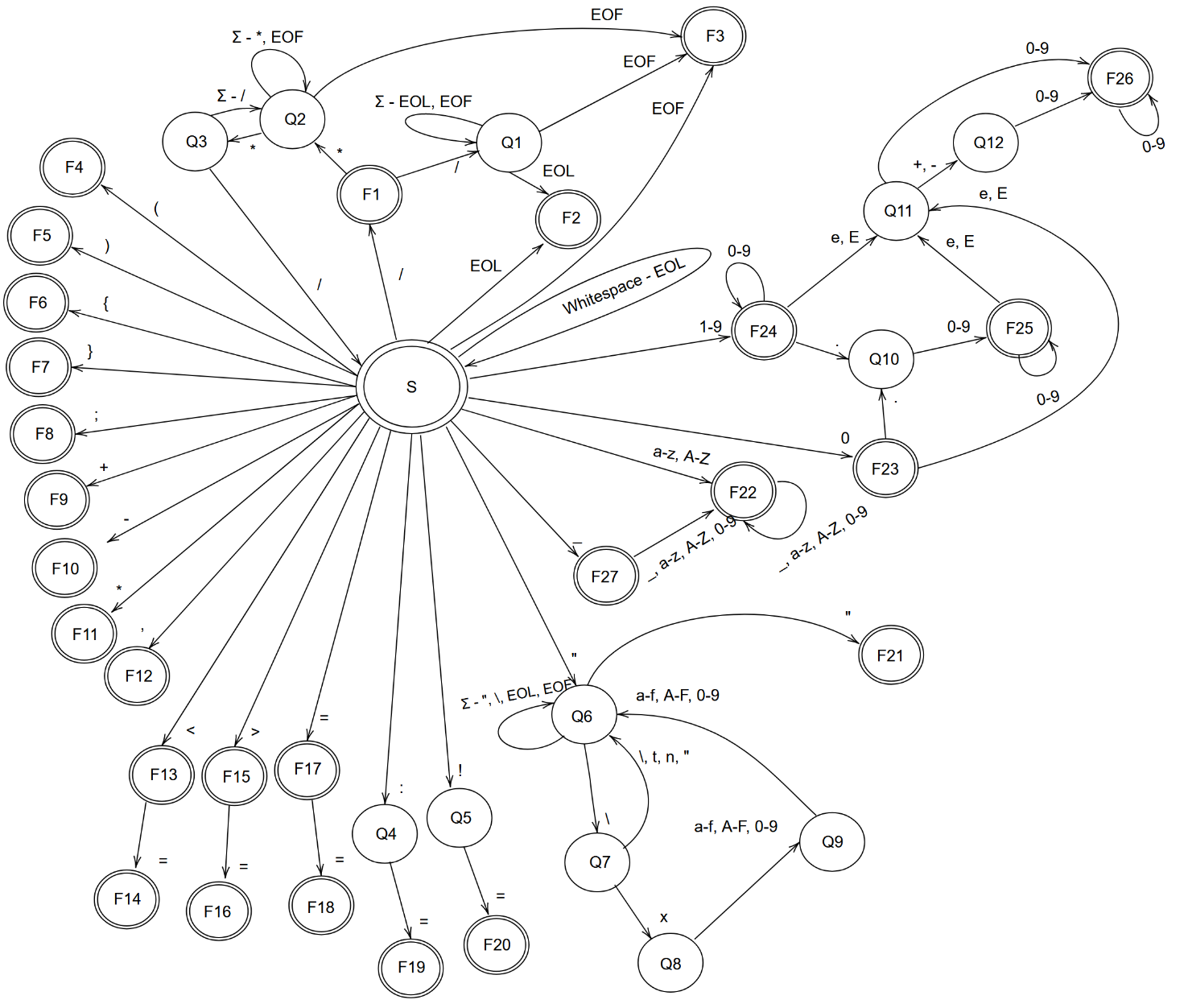
# Řešení projektu

## Lexikální analýza

Lexikální analyzátor je implementován jako deterministický konečný automat fungující dle předem vytvořeného diagramu (Obrázek 1). V jazyce C jsme použili switch, který se v cyklu opakuje tak dlouho, dokud nenačte celý token, nebo nenarazí na chybu. Ze vstupu se postupně načítají znaky, podle jejichž hodnoty se přechází mezi jednotlivými stavy. Při přechodu mezi stavy se lexikální analyzátor snaží načíst co nejdelší možný vstup, který je možný s aktuální konfigurací dosáhnout. Pokud je v nekoncovém stavu načten neočekávaný znak, jedná se o chybu, je vrácen příslušný chybový kód a program je ukončen. V případě, že je automat v koncovém stavu a načte neočekávaný znak, tak přejde do počátečního a jsou uloženy informace o načteném tokenu.

V případě načtení literálu dojde k přetypování na daný datový typ a následně je hodnota uložena do tokenu do proměnné pro tento datový typ. U literálu typu string je navíc jeho hodnota převedena do formátu, který vyhovuje jazyku IFJcode20.

Pro zpracování nekonečně dlouhých řetězců jsme využili soubory [str.h](#_Použitá_literatura_a) a [str.c](#_Použitá_literatura_a), které jsou dostupné na stránkách předmětu IFJ.



Obrázek 1: Diagram konečného stavového automatu

Legenda:

S START Q4 COLON

F1 DIVADE F19 ASSIGNMENT

Q1 ONE\_LINE\_COMMENT Q5 NEG\_STATE\_1

Q2 MULTI\_LINE\_COMMENT F20 NEG\_STATE\_2

Q3 MULTI\_LINE \_COMMENT\_END Q6 STRING\_START

F2 EOL Q7 ESCAPE\_STATE\_START

F3 EOF Q8 ESCAPE\_STATE\_HEX\_1

F4 LEFT\_BRACKET Q9 ESCAPE\_STATE\_HEX\_2

F5 RIGHT\_BRACKET F21 STRING\_END

F6 LEFT\_CURLY\_BRACKET F27 UNDERSCORE

F7 RIGHT\_ CURLY\_BRACKET F22 TEXT\_STATE

F8 SEMICOLON F23 ZERO\_STATE

F9 PLUS F24 NUMBER\_STATE

F10 MINUS Q10 DOT\_STATE

F11 TIMES F25 DECIMAL\_STATE

F12 COMMA Q11 EXPONENT\_STATE

F13 LESS\_THAN Q12 EXPONENT\_SIGN\_STATE

F14 LESS\_OR\_EQUAL\_THAN F26 FLOAT\_STATE

F15 MORE\_THAN

F16 MORE\_OR\_EQUAL\_THAN

F17 EQUALS

F18 EQUAL\_TO

## Syntaktická analýza

Syntaktický analyzátor je hlavní částí celého programu. Po jeho spuštění žádá lexikální analyzátor o token. V případě načtení tokenu, který nevyhovuje syntaktickým pravidlům jazyka IFJ20, je vrácena chyba a program je ukončen.

Při zpracovávání projektu jsme zvolili přístup, kdy se syntaktická analýza provádí společně se sémantickou analýzou a generováním kódu.

### Syntaktická analýza založená na rekurzivním sestupu

Syntaktická analýza se řídí LL gramatikou (odkaz). K většině neterminálů patří funkce, která kontroluje syntaxi a sémantiku a dle gramatiky postupně volá další funkce. Jako první je ze syntaktického analyzátoru volána funkce program. V místech, kde se očekává výraz, je zavolána funkce psa, která provádí syntaktickou analýzu pro výrazy.

Kvůli ne zcela správnému původnímu návrhu LL gramatiky neodpovídají gramatická pravidla 22 a 23 našemu řešení. Zpracování neterminálu <assign\_r> označujícího pravou stranu přiřazení máme provedeno následovně: Na začátku funkce assign\_r je volána funkce psa, která zpracuje případný výraz, literál, či identifikátor a v případě volání funkce nechá zpracovat neterminál <func> zavoláním funkce func. Po skončení funkce psa se dále pokračuje v rekurzivním sestupu.

V našem řešení se vyskytují ještě další neterminály, například <literal> nebo <id\_literal>, které nemají pro své zpracování vlastní funkci. Podrobněji v kapitole 3.2.2.

### Precedenční syntaktická analýza

Pravidla pro zpracování výrazů jsou uložena v precedenční tabulce (odkaz). Řádky tabulky označují terminál, který je nejblíže vrcholu zásobníku. Sloupce označují aktuální token. Každá buňka tabulky může obsahovat jednu ze čtyř následujících hodnot: S (<), R (>), X (chyba), E (=). Pro zjištění pravidla z precedenční tabulky se používá funkce table\_value.

* Pravidlo S (<) vezme token na vstupu, přidá jej na zásobník a označí jako začátek redukce.
* Pravidlo R (>) prochází zásobníkem k místu, které je označeno jako začátek redukce. Část od vrcholu zásobníku až k označenému místu je zredukována na neterminál a jsou provedeny sémantické akce pro dané **pravidlo??**.
* Pravidlo X značí syntaktickou chybu.
* Pravidlo E (=) pouze přidává aktuální token na zásobník.

V souboru psa.c kromě samotného provádění precedenční syntaktické analýzy probíhá i rozlišení, zda je v načítaném vstupním kódu použit identifikátor, anebo se jedná o volání funkce (jméno funkce následované znakem ‘(‘ ).

Funkce psa dále zpracovává literály, výrazy a identifikátory, které jsou zahrnuty v neterminálech, které nejsou zpracovávány rekurzivním sestupem. Jmenovitě se jedná o neterminály <literal>, <expr\_literal>, <id\_literal>, <id\_expr>. Je tomu tak opět z důvodu pozdních změn gramatiky, které jsme již neměli čas přímo reflektovat v kódu.

## Sémantická analýza

V průběhu rekurzivního sestupu se kontroluje, zdali jsou používané funkce proměnné definované. Tato kontrola probíhá za pomoci tabulky symbolů. Pokud se narazí na funkci, která doposud nebyla definována, tak se do tabulky symbolů uloží její název, očekávané typy parametrů a návratové typy, které jsou zjištěny z levé části přiřazení. Zároveň je nastaven atribut isDefined na hodnotu 0, která značí, že funkce zatím nebyla definována.

Po skončení syntaktické analýzy se provede kontrola, jestli jsou všechny volané funkce definované a jestli byla definována funkce main, jejíž použití je v jazyce IFJ20 povinné.

Dále se kontroluje například zda počet parametrů v levé části přiřazení odpovídá počtu parametrů v pravé části a zda odpovídají i datové typy. K této kontrole využíváme strukturu IsUsedList.

## Generování kódu

Před započetím syntaktické analýzy se vygenerují vestavěné funkce a dále se již potřebný kód generuje současně se syntaktickou a sémantickou analýzou.

Pravděpodobně nejobtížnější částí generování kódu byl for cyklus, kde jsme při skončení jedné iterace a začátku druhé, naráželi na problém opětovné definice proměnných v hlavičce a těle cyklu. Tento problém jsme vyřešili vytvořením dočasného rámce, ve kterém jsme znovu definovali všechny dříve definované proměnné z lokálního rámce. Pomocí instrukce PUSHFRAME jsme je pak přesunuli do nového lokálního rámce. Po každé iteraci jsme z lokálního rámce opět vytvořili dočasný rámec instrukcí POPFRAME a následně jsme aktualizovali hodnotu všech dříve definovaných proměnných.

### Generování jmen proměnných

Vzhledem k tomu, že se proměnné mohou překrývat, jsme se rozhodli jejich názvy generovat s příponou čísla rámce, ve kterém je definována.

## Použité datové struktury

### Token

|  |  |
| --- | --- |
| Token je struktura, která slouží k uchovávání informací o aktuálním načteném tokenu.  Nezmínit tady něco k tomu tokenType? | typedef struct token {  tokenType type;  string string;  int64\_t intNumber;  double floatNumber;  } Token; |

### Tabulka symbolů

|  |  |
| --- | --- |
| Dle zvolené varianty zadání jsme implementovali tabulku symbolů jako binární vyhledávací strom. Struktura TNode reprezentuje uzel binárního stromu. Kromě základních atributů jako je klíč, dle kterého se vyhledává a ukazatelů na levý a pravý podstrom struktura obsahuje ještě další atributy, které byly třeba pro syntaktickou a sémantickou analýzu a generování kódu. | typedef struct tNode {  char\* key;  struct tNode\* lptr;  struct tNode\* rptr;  nodeType type;  bool isDefined;  int param;  struct tNode\* localTS;  struct retType\* retTypes;  } TNode; |

nodeType type: Atribut označující, zda se jedná o funkci, anebo o identifikátor. V případě identifikátoru je zde uložen jeho datový typ.

bool isDefined: Atribut označující, zda byla funkce/proměnná definována.

int param: V případě, že se jedná o funkci, určuje tento atribut počet parametrů. Pokud se jedná o vestavěnou funkci print, která nemá specifikovaný počet parametrů, je zde uložena hodnota -1. V případě, že se jedná o identifikátor, který je formálním parametrem funkce, označuje tento atribut pořadí tohoto parametru (začíná od 0).

struct tNode\* localTS: Tento atribut je využit pouze pokud se jedná o funkci. Ukazuje na tabulku symbolů, která obsahuje formální parametry.

struct retType\* retTypes: Lineární seznam obsahující návratové typy.

Většina operací nad tabulkou symbolů je řešena pomocí rekurze. Vycházeli jsme z přednášek předmětu IAL (odkaz na citaci). Dále jsme implementovali několik pomocných funkcí pro snazší práci s tabulkou symbolů.

### Zásobník tabulek symbolů

Pro sémantické akce a generování kódu jsme potřebovali znát úroveň aktuálního rámce. Nový rámec – prvek zásobníku je vytvořen vždy při vstupu do těla funkce/if/else/for cyklu a na začátku hlavičky for cyklu. Tímto způsobem máme informace o platnosti jednotlivých proměnných.

|  |  |
| --- | --- |
| Struktura TStack reprezentuje zásobník. | typedef struct tStack {  struct tStack\_elem\* top;  struct tStack\_elem\* bottom;  } TStack; |

struct tStack\_elem\* top: Ukazatel na vrchol zásobníku (aktuální rámec).

struct tStack\_elem\* bottom: Ukazatel na dno zásobníku. Přidali jsme jej z důvodu, abychom si usnadnili přístup k funkcím, které jsou uloženy v tabulce symbolů právě na dně zásobníku.

|  |  |
| --- | --- |
| Struktura TStack\_Elem reprezentuje prvek zásobníku. | typedef struct tStack\_elem {  int scope;  struct tNode\* node;  struct tStack\_elem\* next;  } TStack\_Elem; |

int scope: Číslo rámce (0 pro první rámec – dno zásobníku).

struct tNode\* node: Ukazatel na tabulku symbolů daného rámce.

struct tStack\_elem\* next: Ukazatel na předcházející rámec.

### Lineární seznam

Lineární seznam jsme využili pro dvě struktury. První z nich je struktura RetType, která slouží k uložení návratových typů funkce. Druhá struktura IsUsedList slouží k určení přiřazovaných hodnot, které se mají zahodit (při použití \_ na levé straně přiřazení) a také ke kontrole datových typů při přiřazování hodnot.

# Závěr

Projekt byl pro nás ve výsledku zajímavý a přínosný, ačkoli jsme ze začátku měli potíž na něm začít pracovat, kvůli jeho velkému rozsahu. Ještě něco dopsat

# Použitá literatura a reference

MAREŠ, Martin a Tomáš VALLA. *Průvodce labyrintem algoritmů*. Praha: CZ.NIC, z.s.p.o., 2017. CZ.NIC. ISBN 978-80-88168-19-5.

Jednoduchý interpret: str.c, str.h. In: *https://www.fit.vutbr.cz/study/courses/IFJ/private/projekt/* [online]. 2009, 10.11.2009 [cit. 2020-12-06]. Dostupné z: <https://www.fit.vutbr.cz/study/courses/IFJ/private/projekt/jednoduchy_interpret.zip>

IAL, něco z ifj, a str.h/str.c