VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ


Dokumentace k projektu do IFJ/IAL

# Implementace překladače jazyka IFJ18

Tým 106, varianta II

**Členové týmu**:

| | | | |
|---|---|---|---|
| Lukáš Válek | – | xvalek15 | 25 % |
| Lukáš Piwowarský | – | xpiwow00 | 25 % |
| Adam Pankuch | – | xpanku00 | 25 % |
| Jindřích Šesták | – | xsesta05 | 25 % |

**Rozšíření**: BASE, IFTHEN

5. prosince 2018

# 1 Úvod

Cílem projektu bylo vytvoření překladače pro jazyk IFJ18, který je podmnožinou jazyka Ruby. Vybrali jsme si variantu II. - tabulka s rozptýlenými položkami.

# 2 Etapy projektu

1. Návrh struktury programu (konečný automat (viz obrázek 1), LL gramatika (viz gramatika 1), LL tabulka (viz gramatika 1), precedenční tabulka (viz tabulka 2))
2. Lexikální analýza
3. Prediktivní syntaktická analýza
4. Precedenční syntaktická analýza
5. Generování kódu
6. Testování
7. Tvorba dokumentace

# 3 Datové struktury

`token_t` – Struktura uchovávající informace o tokenech.
`stack_tkn_t` – Zásobník tokenů. Využívaný především v prediktivní analýze.
`stack_sa_t` – Zásobník realizovaný pomocí obousmerně vázaného seznamu.
`stack_str_t` – Zásobník řetězců. Využívaný především při generování kódu.
`queue_t` – Fronta tokenů a řetězců. Využívaný v lexikální analýze.
`list_t` – Fronta řetězců realizována pomocí jednosměrně vázaného seznamu.
`symtable_t` – Tabulka s rozptýlenými položkami.

# 4 Lexikální analyzátor (LA)

Lexikální analyzátor (dále jen LA) představoval první část projektu, kterou bylo nutno implementovat. Zavedli jsme obousměrnou komunikaci, mezi LA a syntaktickým analyzátorem (dále jen SA), kdy SA může číst token, ale také vrátit token zpět do LA. Při dalším volání LA není token čten ze standartního vstupu, ale z fronty, ve kterém jsou uchovávány dříve vrácené tokeny. Toto je realizováno pomocí datové struktury:`queue_t`. Dále lexikální analyzátor zajišťuje ignorování komentářů.

# 5 Syntaktický analyzátor (SA)

## 5.1 Prediktivní analýza

Rozhodli jsme se použít prediktivní analýzu (dále jen PA), protože nám přišla sofistikovanější a také větší výzva na implementaci. LL tabulka společně s pravidly LL gramatiky jsou umístěny přímo v kódě překladače. LL tabulka je implementována jako matice integerů. Každý integer prezentuje číslo pravidla, které se použije. Pravidla LL gramatiky jsou implentovány jako matice řetzců. Každý řádek matice reprezentuje jedno pravidlo. V LA používáme tyto datové struktury:
`stack_str_t` – Zásobník ukládající generovaný kód při if-statement, while-loop a definicích funkcí.
`stack_tkn_t` – Nutný pro implementaci algoritmu prediktivní analýzy.

## 5.2 Precedenční analýza

Výrazy jsou jsou zpracovávány metodou precedenční syntaktické analýzy pomocí algoritmu známého z přednášek. Analyzátor komunikuje se třemi hlavními moduly: scannerem, parserem a generátorem cílového kódu. Komunikace mezi těmito moduly probíhá převážně pomocí datového typu `token_t`.

Při implementaci jsou použity tyto pomocné datové struktury: `stack_sa_t` – Zásobník uchovávající příchozí tokeny ze scanneru, včetně následných neterminálů, které vzniknou při analyzování výrazu. `stack_tkn_t`– Zásobník uchovávající parametry, se kterými je volána funkce v zdrojovém kódu.

## 5.3 Sémantická analýza

Sémantická analýza probíhá ve třech modulech, a to v: predektivním analyzátoru, precedenčním analyzátoru, generátoru vnitřního kódu.

Prediktivní analyzátor společně s precedenčním analyzátorem je zodpovědný za ověřování správného počtu parametrů při volání funkce, za ověřování redefinice funkcí a kolize jmen proměnných a funkcí.

Generátor vnitřního kódu kontroluje datové typy operandů, pokud je to možné. V případě, že alespoň jeden operand je proměnná, dochází k sémantické kontrole za běhu. Dále se generuje běhová kontrola dělení nulou.

## 5.4 Implementace TS

Tabulka symbolů je implementována jako tabulka s rozptýlenými položkaci s explicitně zřetězenými synonymy. Synonyma jsou svázána do jednosměrného seznamu. Položky jsou vkládány na začátek seznamu. Jako mapovací funkci jsme použili funkci z předmětu IJC.

## 5.5 Generování kódu

Pro potřeby tohoto projektu jsme zvolili přímé generování tří adresného kódu, protože jiné metody neposkytovaly žádnou výhodu jen časové zdržení. Instrukce generujeme do abstraktního datového typu List, protože je potřeba zamezit vícenásobné deklaraci v cyklech. Pokud se analýza nachazí v cyklu, budou veškeré instrukce deklarace proměnných vytisknuty před začátkem cyklu.

Generování kódu probíhá v těchto modulech: prediktivní analyzátor a precedenční analyzátor.

Prediktivní analyzátor generuje cílový kód pro definice funkcí, if-statement, while-loop a definování proměnných.

Precedenční analyzátor generuje cílový kód pro volání funkcí, volání vestavěných funkcí, výrazy.

### 5.5.1 Generování výrazů

V rámci generování instrukcí pro výrazy se musela provádět sémantická analýza. Tu jsem provedli použitím instrukcí pro zjištění typu proměnných, jejich přetypování a následných skoků. Při generování kódu jsme se rozhodli pro lehkou optimalizaci. Ta spočívá v tom, že na základě kombinace datových typů operandů, získaných z precedenční analýzy, se vygenerují jen ty nejnutnější instrukce. Tím se omezí množství skoků a testů ve výsledném programu. Vygenerovaný kód je přehlednější a dosáhne se zrychlení běhu programu.

# 6 Rozdělení práce

Lukáš Válek
- Tabulka symbolů
- Implementace stacku pro řetězce
- Generování cílového kódu pro výrazy
- Generování cílového kódu pro vestavěné funkce
- Implementace dynamického listu parametrů

Lukáš Piwowarski
- Návrh a implementace precedenční analýzy
- Implementace stacku pro tokeny
- Generování cílového kódu pro volání funkcí
- Implementace stacku pro precedenční analýzu

Adam Pankuch
- Návrh a implementace prediktivní analýzy
- Implementace datové struktury list, queue
- Generování cílového kódu pro if-statement, while-loop, definice funkcí, definice proměnných
- Implementace testovacího skriptu

Jindřich Šesták
- Návrh a implementace lexikálního analyzátoru
- Generování cílového kódu pro if-statement, while-loop
- Generování vestavěných funkcí
- Implementace dynamického řetězce

Dále se každý podílel na tvorbě dokumentace a tvorbě testovacích zdrojových kódů.

# 7 Rozšíření

## 7.1 BASE

Toto rozšíření jsme řešili malou pouze malou úpravou v LA.

## 7.2 IFTHEN

Už v návrhu LL gramatiky jsme počítali s tímto rozšířením, proto nebyly potřebné pozdější úpravy.

# 8 Komunikace v týmu

Jako hlavní komunikační kanál sloužil Messenger, popř. Discord (videohovory). Dále probíhaly pravidelné schůzky, na kterých jsme prodiskutovávali udělenou práci a rozdělovali jsme novou.

Jako verzovací systém jsme použili Git hostovaný na službě Github.

# 9 Shrnutí

Projekt byl pro nás zcela novou zkušeností. Nikdo z nás nikdy nepracoval na větším týmovém projektu, proto nám proces tvoření překladače přinesl spoustu nových znalostí, nejen z oblasti formálních jazyků, programování v jazyce C, návrhu komplexnějšího programu, ale také z oblasti týmové práce a uskalými s ní spojenými, jako je například komunikace. Určitě byl pro nás projekt přínosem.

| | **if** | **else** | **elseif** | **end** | **while** | **def** | **EOL** | **EOF** | **=** | **(** | **)** | **,** | **ID** | **FUNC** | **"expr"** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **[st-list]** | 1 | 0 | 0 | 0 | 1 | 1 | 2 | 3 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| **[EOL-EOF]** | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **[stat]** | 7 | 0 | 0 | 0 | 7 | 6 | 0 | 0 | 0 | 7 | 0 | 0 | 7 | 7 | 7 |
| **[command]** | 9 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 |
| **[func-assign-expr]** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 11 | 0 | 0 | 0 | 0 | 0 | 0 |
| **[end-list]** | 12 | 0 | 0 | 14 | 12 | 0 | 13 | 0 | 0 | 12 | 0 | 0 | 12 | 12 | 12 |
| **[if-list]** | 15 | 18 | 17 | 19 | 15 | 0 | 16 | 0 | 0 | 15 | 0 | 0 | 15 | 15 | 15 |
| **[id-func]** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 20 | 21 | 0 |
| **[params-gen]** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 22 | 0 | 0 | 0 | 0 | 0 |
| **[p-brackets]** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 24 | 0 | 23 | 0 | 0 |
| **[p-brackets-cont]** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 26 | 25 | 0 | 0 | 0 |

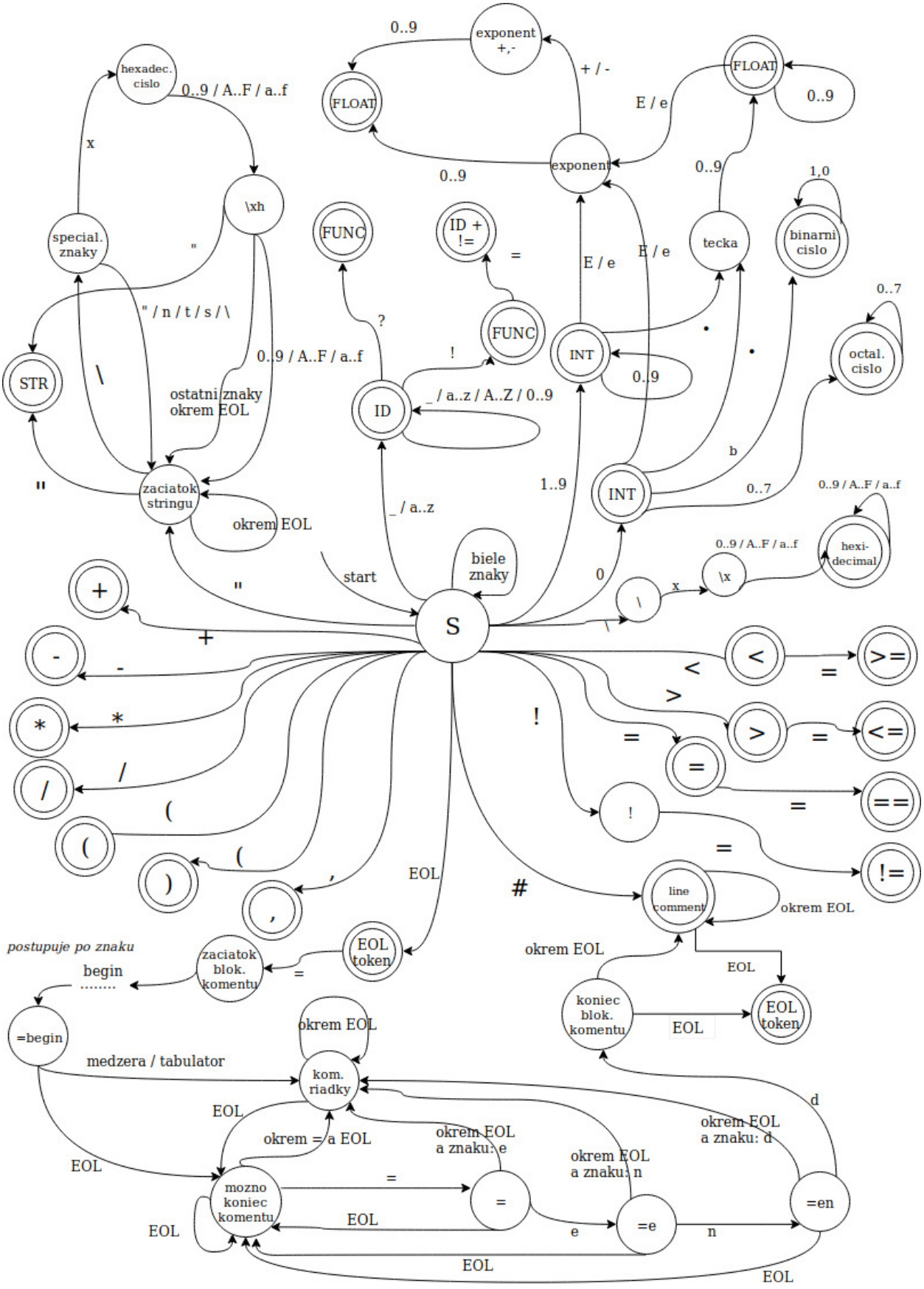
Tabulka 1: LL tabulka

**Gramatika 1:** LL GRAMATIKA

1: [st-list] → [stat] [EOL-EOF]
2: [st-list] → EOL [st-list]
3: [st-list] → $\varepsilon$
4: [EOL-EOF] → EOL [st-list]
5: [EOL-EOF] → $\varepsilon$
6: [stat] → def [id-func] [params-gen] [end-list]
7: [stat] → [command]
8: [command] → while do EOL [end-list]
9: [command] → if then EOL [if-list]
10: [command] → ID [func-assign-expr]
11: [func-assign-expr] → =
12: [end-list] → [command] EOL [end-list]
13: [end-list] → EOL [end-list]
14: [end-list] → end
15: [if-list] → [command] EOL [if-list]
16: [if-list] → EOL [if-list]
17: [if-list] → elif then EOL [if-list]
18: [if-list] → else EOL [end-list]
19: [if-list] → end
20: [id-func] → ID
21: [id-func] → FUNC
22: [params-gen] → ( [p-brackets]
23: [p-brackets] → ID [p-brackets-cont]
24: [p-brackets] → ) EOL
25: [p-brackets-cont] → , ID [p-brackets-cont]
26: [p-brackets-cont] → ) EOL

| OP | + | * | ( | ) | i | - | / | rel | f | , | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| + | > | < | < | > | < | > | < | > | < | > | > |
| * | > | > | < | > | < | > | > | > | < | > | > |
| ( | < | < | < | = | < | < | < | < | < | = | X |
| ) | > | > | X | > | X | > | > | > | X | > | > |
| i | > | > | X | > | X | > | > | > | X | > | > |
| - | > | < | < | > | < | > | < | > | X | > | > |
| / | > | > | < | > | < | > | > | > | X | > | > |
| rel | < | < | < | > | < | < | < | < | X | > | > |
| f | X | X | = | X | < | X | X | X | X | < | > |
| , | < | < | < | = | < | < | < | < | < | = | > |
| $ | < | < | < | X | < | < | < | < | < | X | X |

Tabulka 2: Precedenční tabulka




Obrázek 1: Konečný automat