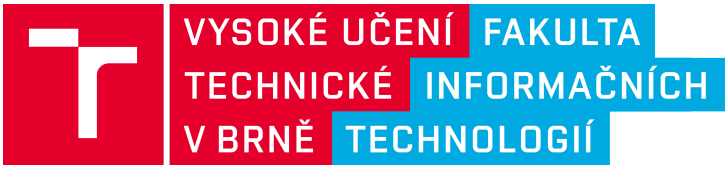



# IFJ projekt 2022

Tým xhucov00, Varianta TRP
Implementovaná rozšíření: FUNEXP, OPERATORS (částečně)

Vedoucí:
Vladimír Hucovič (xhucov00)

Ostatní členové týmu:
Petr Kolouch (xkolou05)
Ondřej Zobal (xzobal01)
Marek Havel (xhavel46)

7. prosince 2022

| Jméno | xlogin | bodový podíl |
|---|---|---|
| Vladimír Hucovič | xhucov00 | 25% |
| Petr Kolouch | xkolou05 | 25% |
| Ondřej Zobal | xzobal01 | 25% |
| Marek Havel | xhavel46 | 25% |

Tabulka 1: Bodové rozdělení

# Obsah



---

**Autoři**

- Vladimír Hucovič - Syntaktická analýza - rekurzivní sestup i precedenční analýza, sémantická analýza, generování kódu
- Petr Kolouch - Lexikální analýza, syntaktická analýza - rekurzivní sestup, Sémantická analýza
- Ondřej Zobal - Lexikální analýza, precedenční syntaktická analýza, sémantická analýza, generování kódu
- Marek Havel - Lexikální analýza, syntaktická analýza - rekurzivní sestup, QE pro generování kódu

## Lexikální analýza

Prvním problémem bylo postavit stavový automat a vytvořit rozumný výčtový typ pro všechny možné stavy. Všechny koncové stavy mají korespondující lexémy, ale mezistavy vlastní lexémy nemají. Proto jsme použili dva nezávislé výčty `ScannerState` a `lex`.

Lexikální analyzátor je implementován v souboru `scanner.c`, jeho hlavní funkcí je `scan_next_token`, ta přečte následující token v daném souboru a vrátí jej. `scan_next_token` čte soubor po jednotlivých znacích a společně s informací o předchozím stavu je posílá do funkce `get_next_state`, která implementuje celý stavový automat lexikální analýzy (viz obrázek č. 1 na straně 9). `get_next_state` pak vyhodnotí přechod mezi stavy a o výsledku informuje prostřednictvím struktury `StateInfo`. Ta obsahuje:

1. Výčet `result`, který sděluje funkci `scan_next_token`, jak má nakládat s právě přečteným bytem, jestli jej má uložit do znakového seznamu, vrátit jej zpět do souboru, přeskočit nebo zastavit čtení a oznámit chybu, zároveň informuje o tom, zda bylo dokončeno čtení lexému.

2. Unii proměnných `next_state` a `lex`. V případě že z `result` vyplývá, že se dokončilo čtení tokenu, `lex` obsahuje platnou hodnotu lexému tohoto tokenu. V opačném případě je definován `next_state`, který sděluje do kterého stavu automat přešel. `scan_next_token` si tuto hodnotu pamatuje do příštího volání `get_next_state`.

Podle návratové hodnoty z `get_next_state` funkce `scan_next_token` ukládá přečtené znaky do seznamu `buffer`, takto nasbírané znaky jsou po dokončení čtení uloženy do tokenu jako užitečná hodnota typu `string`, nebo v případě celých a desetinných čísel, jako `int` nebo `double`.

Funkce `scan_next_token` může začínat stavem `S_START`, který slouží k běžnému čtení tokenů, nebo stavem `S_PROLOG`, který je použit při čtení prvního tokenu v souboru a způsobí, že stavový automat nahlásí chybu při výskytu jakýchkoliv nepovolených znaků před prologem (viz obrázek č. 3 na straně 10).

Oříškem byla implementace dekódování escape sekvencí řetězců. Ta probíhala až jako jedna z posledních prací na lexikálním analyzátoru a do dosavadního návrhu příliš neseděla. Nakonec jsme ji implementovali prostřednictvím pomocného stavového automatu `process_str_escape_sequence` (diagram viz obrázek č. 2 na straně 10), který má svoje vlastní stavy. Jelikož by bylo nepraktické používat výčtový typ pro zaznamenávání veškerých kombinací escape sekvencí, použili jsme seznam znaků a výčet použili pouze k rozlišení druhu escape sekvence.

Dalším problémem bylo zpracovávání komentářů. Rozhodli jsme se k nim přistupovat jako k ostatním lexémům, máme pro ně ve stavovém automatu speciální stav, a považujeme je jako speciální typ bílých znaků.

## Syntaktická analýza - Rekurzivní sestup

Pro zpracování všech příkazů programu kromě výrazů jsme použili metodu rekurzivního sestupu. Všechny funkce, které rekurzivní sestup implementují, se nachází v souboru `parser.c` a jejich název vždy obsahuje příponu `Expansion`. V místech, kde je v programu očekáván výraz, se předává kontrola precedenční syntaktické analýze, která je implementována ve funkci `precParser`. Precedenční analýza je dále popsána v následující kapitole. Mezi syntaktickými kontrolami se průběžně přímo generuje cílový kód a provádí se sémantické kontroly. LL-tabulka, která řídí rekurzivní sestup, je v příloze - tabulka 4 na straně 11.

Syntaktická analýza v našem překladači probíhá ve dvou průchodech: v prvním se pouze zkontrolují a posbírají hlavičky funkcí a přidají se do globální tabulky symbolů. V druhém průchodu se projde zbytek kódu. S naším řešením je tedy možné volat funkci dřív, než je definována.

## Syntaktická analýza - Precedenční syntaktická analýza

Syntaktická kontrola výrazů probíhá zdola-nahoru metodou precedenční syntaktické analýzy. Pro potřeby této metody jsme implementovali abstraktní datový typ zásobník, který kromě běžného zásobníku obsahuje některé speciální metody. Bližší popis zásobníku a dalších struktur využívaných v precedenční analýze se nachází v kapitole Použité datové struktury.

Zásobník může obsahovat 3 typy prvků: `Terminal`, `Handle` a `Nonterminal`. Typ `Terminal` obsahuje načtený token, obohacený o další informace. Typ `Handle` označuje místo na zásobníku, nad kterým se bude při příští redukci redukovat. Typ `Nonterminal` reprezentuje redukovaný výraz, kterým může být term nebo dva výrazy spojené aritmetickým, relačním či řetězcovým operátorem. Samotná analýza pak načítá tokeny na vstupu a podle jejich precedence provádí operace vkládání a redukce na zásobníku. Precedenční tabulka je implementována ve funkci `getPrecedence`. Při redukci se funkcí `precParseCheckRule` kontroluje, jestli redukovaný výraz odpovídá pravidlům výrazové gramatiky. Pokud redukce proběhné úspěšně, redukované prvky jsou odstraněný ze zásobníku a nahrazeny novým prvkem typu `Nonterminal`, který popisuje původní výraz. V případě neúspěchu je překlad ukončen se syntaktickou chybou. Výstupem precedenční analýzy je jeden prvek typu `Nonterminal`, který obsahuje veškeré informace o zpracovaném výrazu. Tento prvek tvoří kořen binárního stromu, jehož uzly obsahují informaci o konkrétním operátoru a synové uzlů jsou operandy. Precedenční tabulka je k dispozici v příloze - tabulka 5 na straně 11.

## Sémantická analýza

Protože je IFJ22 dynamicky typovaný jazyk, většina sémantické analýzy je prováděna za běhu přímo v interpretu pomocí interních funkcí, které jsou stručně popsány v následující kapitole

## Generování kódu

Jak už bylo uvedeno v kapitole o rekurzivním sestupu, výsledný kód v jazyce IFJcode22 se generuje průběžné mezi syntaktickými akcemi.

Kód pro funkce je vygenerován mezi příkazy hlavního těla programu, proto je před návěštím funkce vygenerován kód pro skok na návěští označující konec funkce, aby se předešlo nechtěnému vykonávání příkazů v těle funkce bez jejího volání. Návěští začátku funkce je tvořeno podtržítkem a jménem funkce. Ještě před generováním příkazů v těle funkce se provádí volání funkce, která definuje lokální proměnné. Poté se provede přesun argumentů ze zásobníku do lokálních proměnných a zároveň jejich typová kontrola. Na konci funkcí, které mají návratový typ jiný než `void`, se generuje skok na návěští `ERROR_4`, který proběhne v případě, že se v těle funkce nenarazí na příkaz návratu.

Návěští generovaná pro řídící příkazy podmínky a cyklu jsou očíslována podle jejich pořadí ve zdrojovém programu, číslováno od 0. Tímto očíslováním je zajištěna jejich jedinečnost v rámci přeloženého programu.

Kód výrazů se generuje ze stromu neterminálů, do kterého se zanořujeme dvěma různými způsoby:

1. reverzním postorderem v případě že je aktuální uzel neterminál přiřazení. Toto je nutné protože v naší implementaci kompilátoru považujeme znak přiřazení za operátor a podporujeme zřetězené přiřazení.

2. běžným postorderem v případě ostatních operátorů.

Při výpočtu výrazů používáme zásobníkový kód všude, kde je to možné.

### Interní funkce

- `%ENFORCE_TYPES` - Ověří, že hodnota ve druhém argumentu je rovna řetězci v prvním argumetu. Ze zásobníku uklidí pouze první argument. Funkce nic nevrací, ale v případě, že typy nesedí, ukončí program.
- `%NORMALIZE_TYPES` - Pokusí se převést své dva arumenty na stejný datový typ, v případě že to není možné, program ukončí, jinak oba své argumenty vrací.
- `%NORMALIZE_NUMERIC_TYPES` - Pokusí se převést své dva argumenty na stejný číselný datový typ, v případě, že to není možné, program ukončí, jinak oba své argumenty vrací.
- `%NULL_TO_INT` - Funkce otestuje jestli je některý ze dvou argumentů `nil@nil` a v případě že ano, nahradí jej za `int@0`. Oba argumenty vrací.
- `%EMPTY_STRING_TO_INT` - Jesliže je argument typu `str` a je prázdný, vrací `int@0`, jinak vrací argument.
- `%NULL_TO_STR` - Jestliže je argument typu `nil@nil`, vrací `string@`, jinak vrací argument.
- `%TO_BOOL` - Zkonvertuje daný argument na boolean a vrátí ho.
- `%RELATION_TYPECAST` - Jestliže je první argument typu `nil@nil`, převede jej na typ druhého argumentu a oba vrátí.
- `%STACK_SWAP` - Vrací své dva argumenty v opačném pořadí.
- `%COMPARE_DTYPES` - Vrací pravdu pokud jsou datové typy hodnot dvou argumentů identické, jinak nepravdu.
- `%EQUALITY` - Vrací pravdu pokud jsou dva dané argumenty stejné, jinak vrací nepravdu.
- `%NONEQUALITY` - Vrací nepravdu pokud jsou dva dané argumenty stejné, jinak vrací pravdu.
- `%LESS` - Porovná dva argumenty znamínkem `<`.
- `%GREAT` - Porovná dva argumenty znamínkem `<=`.
- `%GREAT` - Porovná dva argumenty znamínkem `>`.
- `%GREAT_EQUAL` - Porovná dva argumenty znamínkem `>=`.
- `%CHECK_IF_IS_TYPE` - Jako argumenty dostane typ a hodnotu, vrátí true jestli je hodnota stejného typu.
- `%CHECK_IF_IS_TYPE_OR_NULL` - To samé, jen vrátí true i v případě že typ hodnoty je NULL.

## Použité datové struktury

### ADT seznam

V projektu na mnoha místech využíváme datový typ dvousměrně vázaný seznam. Je implementován genericky pomocí makra a v projektu jsme ho využili pro ukládání typů `Nonterminal`, `char`, `Token` a `variable`. Implementace seznamu je v souborech `list.h` a `list.c`

#### listElem

Typ prvku uložený v seznamu. Obsahuje užitečná data a odkazy na levého a pravého souseda.

## Lexikální analýza

### Lex

Lex je výčtový typ který slouží pro určení typu lexému, který se předává prostřednictvím `tokenu` do syntaktické analýzy.

### Token

Struktura, která ukládá informace o vstupních tokenech. Atribut `lex` určuje lexém. Dále obsahuje unii, která ukládá další informace o tokenu v závislosti na tom, o jaký lexém se jedná (například název proměnné či hodnotu celočíselného literálu)

### Scanner state

Scanner state je výčtový typ, pomocí kterého probíhá přechod mezi jednotlivými stavy stavového automatu skeneru. Jeho obsah reflektuje všechny stavy schématu výjma konečných stavů.

## Syntaktická analýza

### ADT zásobník

Zásobník, který v projektu používáme v precedenční syntaktické analýze, je rozšířenou verzi klasického zásobníku. Konkrétně je například možné získat ze zásobníku i data prvku, který není přímo na vrcholu (např. funkcí `findHandle`). Navíc jsou prvky v něm dvousměrně vázány, v tomto ohledu má tedy spíše více společného s datovým typem seznam. Implementace zásobníku se nachází v souborech `stack.h` a `stack.c`.

### stackData

struktura stackData představuje datovou část prvků na zásobníku. Obsahuje unii, ve které je buď ukazatel na terminál nebo neterminál, případně nedefinovaná hodnota. Jaký z těchto ukazatelů je platný, případně jestli vůbec některý, lze zjistit z výčtu `elemType`, který se tu rovněž nachází.

### stackElement

Struktura prvku na zásobníku, skládá se z datové části a ukazatelů na následující a předchozí prvek.

### elemType

elemType určuje typ prvku na zásobníku. Může se jednat o `Terminal`, `Nonterminal` nebo `Handle`.

### dataType

dataType uchovává informaci o datovém typu hodnoty uložené ve strukturách `Terminal` a `Nonterminal`

### Terminal

Struktura obsahující odkaz na token, výčet terminalType a v případě, že se jedná o volání funkce, argumenty tohoto volání.

### terminalType

Určuje, o jaký typ terminálu se jedná pro potřeby precedenční analýzy. Existuje zde překrytí s výčtem Lex. Některé typy jsou sloučeny, kdežto jiné jsou odstraněny.

#### Nonterminal

Tato struktura ukládá informace o výrazu potom, co byl na zásobníku redukován. Tvoří pak binární strom, který se velmi příjemně zpracovává při generování kódu, díky přítomnosti zásobníkových instrukcích v jazyce IFJcode22.

#### Funcall

Struktura `funcall` slouží pro popis volání funkce. Parametr `funId` odpovídá jménu funkce, `args` seznamu argumentů volané funkce.

### Sémantická analýza

### Symtable

V naší variantě zadání jsme tabulku symbolů implementovali jako tabulku s rozptýlenými položkami. Použili jsme algoritmus `djb2`, jeho implementaci jsme převzali z webu `http://www.cse.yorku.ca/~oz/hash.html`.

#### symtable_item

Záznam v tabulce symbolů. Skládá se z klíče, hodnoty (tvořené strukturou symtable elem) a ukazatele na další synonymum.

#### symtable_elem

Záznam v tabulce symbolů. Výčet `type` určuje, zda se jedná o typ `function` nebo `variable`. Unie pak obsahuje ukazatel na tento typ.

#### function

Struktura vytvořena pro funkce. Obsahuje název funkce, návratový datový typ, příznak zda může být `null`, seznam argumentů funkce a lokální tabulku symbolů.

#### variable

Struktura proměnné složena z jejího jména a datového typu s příznakem, jestli může být jeho hodnota nulová.

## Makefile

Pro automatizaci a sestavení jednotlivých částí programu jsme vytvořili univerzální Makefile. Součástí tohoto souboru je nastavení překladače včetně parametrů pro dodatečnou detekci chyb a varování, inkrementální sestavování. Program je možné sestavit s ladícími symboly pomocí příkazu `make`, nebo v optimalizované verzi pomocí `make release=1`.

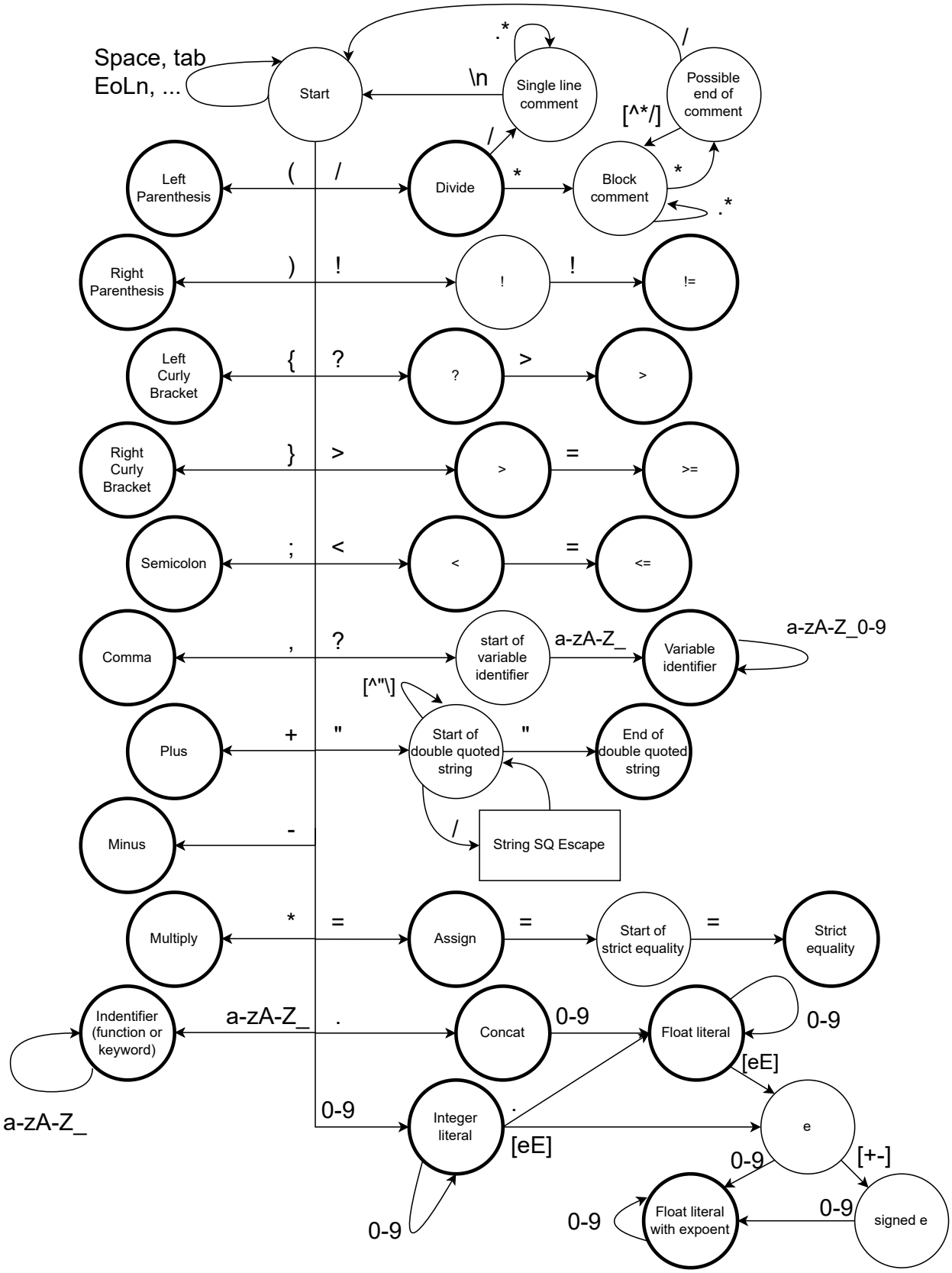
## Seznam příloh

| Název | Typ | Číslo |
|---|---|---|
| LL Gramatiky | Tabulka | 3 |
| Diagram stavového automatu | Obrázek | 1 |
| Diagram automatu pro escape sekvence v řetěžcích | Obrázek | 2 |
| Diagram automatu pro načítání prologu | Obrázek | 3 |
| LL tabulka | Tabulka | 4 |
| LL Precedens | Tabulka | 5 |

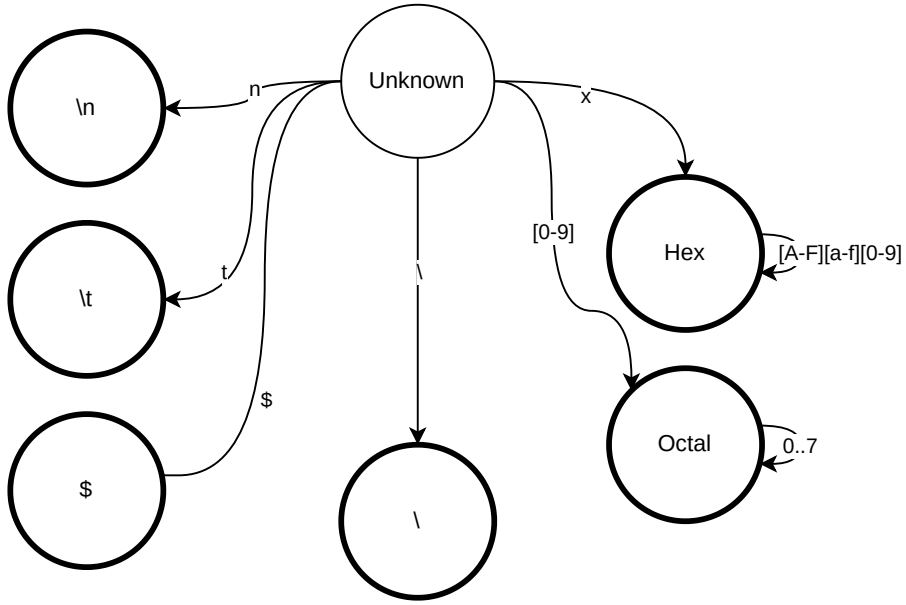
Tabulka 2: Přílohy

01 <PROG> ➔ ‘<?php’ <DECLARE_ST> <ST_LIST> <END_TOKEN>
02 <DECLARE_ST> ➔ ‘declare’ ‘(‘ ‘strict_types’ ‘=’ ‘1’ ‘)’ ’;’
03 <ST_LIST> ➔ <ST> <ST_LIST>
04 <ST_LIST> ➔ eps
05 <ST> ➔ ‘function’ <ID> ‘(‘ <PARAMS> ‘)’ ‘:’ <TYPE> <BLOCK>
06 <ST> ➔ ‘if’ ‘(‘ <EXPR> ‘)’ <BLOCK> ‘else’ <BLOCK>
07 <ST> ➔ ‘while’ ‘(‘ <EXPR> ‘)’ <BLOCK>
08 <ST> ➔ ‘return’ <EXPR> ‘;’
09 <ST> ➔ <EXPR> ‘;’
10 <PARAMS> ➔ <TYPE_NAME_NO_VOID> <VAR> <PARAM_LIST>
11 <PARAMS> ➔ eps
12 <PARAM_LIST> ➔ ‘,’ <?> <TYPE_NAME_NO_VOID> <VAR> <PARAM_LIST>
13 <PARAM_LIST> ➔ eps
14 <?> ➔‘?’
15 <?> ➔eps
16 <TYPE> ➔ <TYPE_NAME>
17 <TYPE_NAME> ➔ ’?’ <TYPE_NAME_NO_VOID>
18 <TYPE_NAME> ➔ ’int’ | ’float’ | ’void’ | ’string’
19 <TYPE_NAME_NO_VOID> ➔ ’int’ | ’float’ | ’string’
20 <BLOCK> ➔ ‘{‘ <BLOCK_STLIST> ‘}’
21 <EXPR> ➔ <SWITCH TO PREC. PARSING>
22 <END_TOKEN> ➔ ‘?>’
23 <END_TOKEN> ➔ eps
24 <BLOCK_STLIST> ➔ eps
25 <BLOCK_STLIST> ➔ <BLOCK_ST> <BLOCK_STLIST>
26 <BLOCK_ST> ➔ ‘if’ ‘(‘ <EXPR> ‘)’ <BLOCK> ‘else’ <BLOCK>
27 <BLOCK_ST> ➔ ‘while’ ‘(‘ <EXPR> ‘)’ <BLOCK>
28 <BLOCK_ST> ➔ ‘return’ <EXPR> ‘;’
29 <BLOCK_ST> ➔ <EXPR> ‘;’

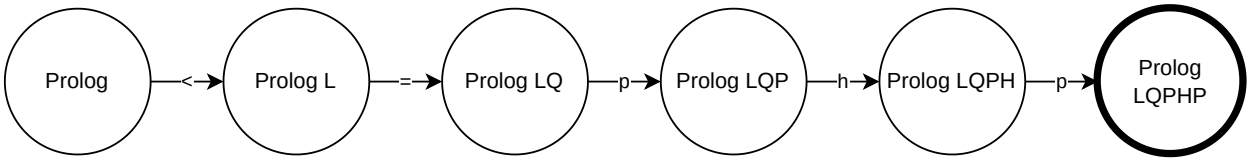
Tabulka 3: Gramatika




Obrázek 1: Diagram stavového automatu




Obrázek 2: Diagram automatu pro escape sekvence v řetězcích




Obrázek 3: Diagram automatu pro načítání prologu

| NONTERM \TERM | $ | <?php | declare | ?> | function | if | while | int | float | void | string | return | VAR | ? | , | { | FUNCALL | LITERAL | ) | } |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| <PROG> | | 01 | | | | | | | | | | | | | | | | | | |
| <DECLARE_ST> | | | 02 | | | | | | | | | | | | | | | | | |
| <ST_LIST> | 04 | | | 04 | 03 | 03 | 03 | | | | | 03 | 03 | | | | 03 | 03 | | |
| <BLOCK_STLIST> | | | | | | 25 | 25 | | | | | 25 | 25 | | | | 25 | 25 | | 24 |
| <ST> | | | | | 05 | 06 | 07 | | | | | 08 | 09 | | | | 09 | 09 | | |
| <BLOCK_ST> | | | | | | 26 | 27 | | | | | 28 | 29 | | | | 29 | 29 | | |
| <PARAMS> | | | | | | | | | 10 | | 10 | 10 | | | | | | | 11 | |
| <PARAM_LIST> | | | | | | | | | | | | | | | 12 | | | | 13 | |
| <TYPE> | | | | | | | | | 14 | 14 | 14 | 14 | | | 14 | | | | | |
| <?> | | | | | | | | 15 | 15 | 15 | 15 | | | 14 | | | | | | |
| <BLOCK> | | | | | | | | | | | | | | | | 20 | | | | |
| <END_TOKEN> | 23 | | | 22 | | | | | | | | | | | | | | | | |

Tabulka 4: LL tabulka

| PREC_TABLE | "+" | "-" | "/" | "*" | "=" | "===" | "!==" | ">=" | "<=" | ">" | "<" | ( | ) | operand | "$" | "." |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| "+" | > | > | < | < | > | > | > | > | > | > | > | < | > | < | > | > |
| "-" | > | > | < | < | > | > | > | > | > | > | > | < | > | < | > | > |
| "/" | > | > | > | > | > | > | > | > | > | > | > | < | > | < | > | > |
| "*" | > | > | > | > | > | > | > | > | > | > | > | < | > | < | > | > |
| "=" | < | < | < | < | < | < | < | < | < | < | < | < | > | < | > | < |
| "===" | < | < | < | < | > | > | > | < | < | < | < | < | > | < | > | < |
| "!==" | < | < | < | < | > | > | > | < | < | < | < | < | > | < | > | < |
| ">=" | < | < | < | < | > | > | > | > | > | > | > | < | > | < | > | < |
| "<=" | < | < | < | < | > | > | > | > | > | > | > | < | > | < | > | < |
| ">" | < | < | < | < | > | > | > | > | > | > | > | < | > | < | > | < |
| "<" | < | < | < | < | > | > | > | > | > | > | > | < | > | < | > | < |
| ( | < | < | < | < | < | < | < | < | < | < | < | < | = | < | | < |
| ) | > | > | > | > | > | > | > | > | > | > | > | | > | | > | > |
| operand | > | > | > | > | > | > | > | > | > | > | > | | > | | > | > |
| "$" | < | < | < | < | < | < | < | < | < | < | < | < | | < | | < |
| "." | > | > | < | < | > | > | > | > | > | > | > | < | > | < | > | > |

Tabulka 5: Precedenční tabulka