Implementace překladače imperativního jazyka IFJ23

Tým xdubro01

Maksim Dubrovin	xdubro01
Anastasiia Samoilova	xsamoi00
Ilya Volkov	xvolk02
Anastasiia Mironova	xmiron05

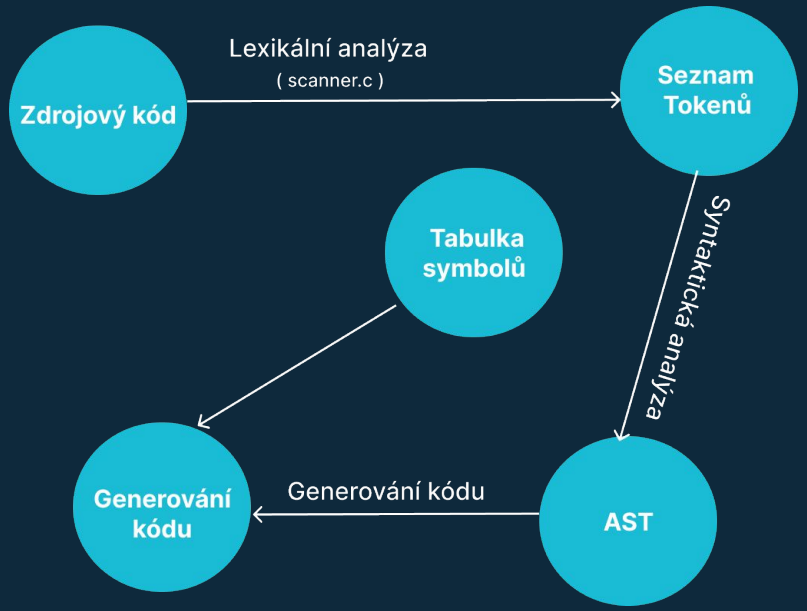
Obsah

- Úvod
  - Lexikální analýza
  - Syntaktická analýza
  - Generování kódu
- O práci v týmu

Jak to funguje v našem programu?

- Čtení zdrojového kódu
- Zahájení analýzy
- Generování kódu nebo návrat chyby

1

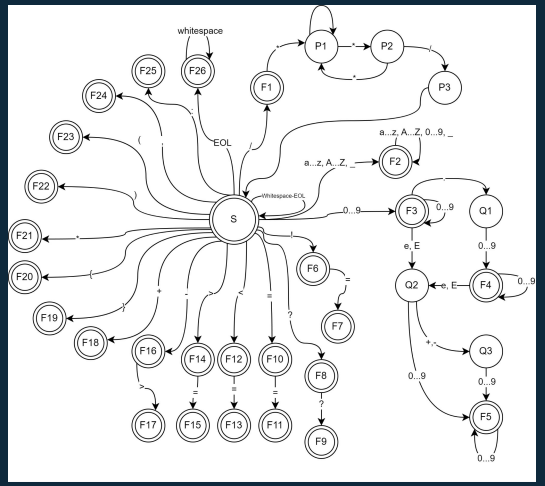



```
var i : Int = 2 * 2

while (i > 0)
{
    write("hello\n")
    i = i - 1
}
```

Příklad zdrojového kódu

Lexikální analýza



- Teoreticky lexikální analyzátor je konečný automat

- Implementováno v kódu pomocí "switch case"

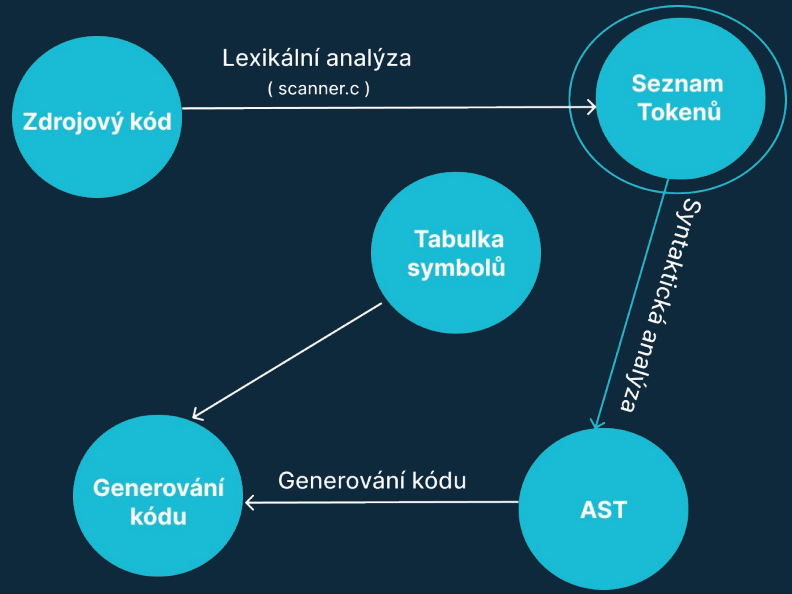
```
char ch = advance();
switch (ch)
{
    case '*': return makeFromType(TOKEN_STAR);
    case '+': return makeFromType(TOKEN_PLUS);
    case '-':
        return match('>')? makeFromType(TOKEN_ARROW): makeFromType
        (TOKEN_MINUS);
    case '(': return makeFromType(TOKEN_LEFT_PAREN);
    case ')': return makeFromType(TOKEN_RIGHT_PAREN);
    case '{': return makeFromType(TOKEN_LEFT_BRACE);
    case '}': return makeFromType(TOKEN_RIGHT_BRACE);
    case '/': return makeFromType(TOKEN_SLASH);
    case ',': return makeFromType(TOKEN_COMMA);
    case '.': return makeFromType(TOKEN_DOT);
    case ':': return makeFromType(TOKEN_COLON);
    case '\n': return makeFromType(TOKEN_EOL);
    case '?':
        return match('?')? makeFromType(TOKEN_DOUBLE_QUESTIONS):
        makeFromType(TOKEN_QUESTION);
    case '!':
        return match('=')? makeFromType(TOKEN_BANG_EQUAL):makeFromType
        (TOKEN_BANG);
    case '=':
        return match('=')? makeFromType(TOKEN_EQUAL_EQUAL): makeFromType
        (TOKEN_EQUAL);
    case '<':
```

Lexikální analýza

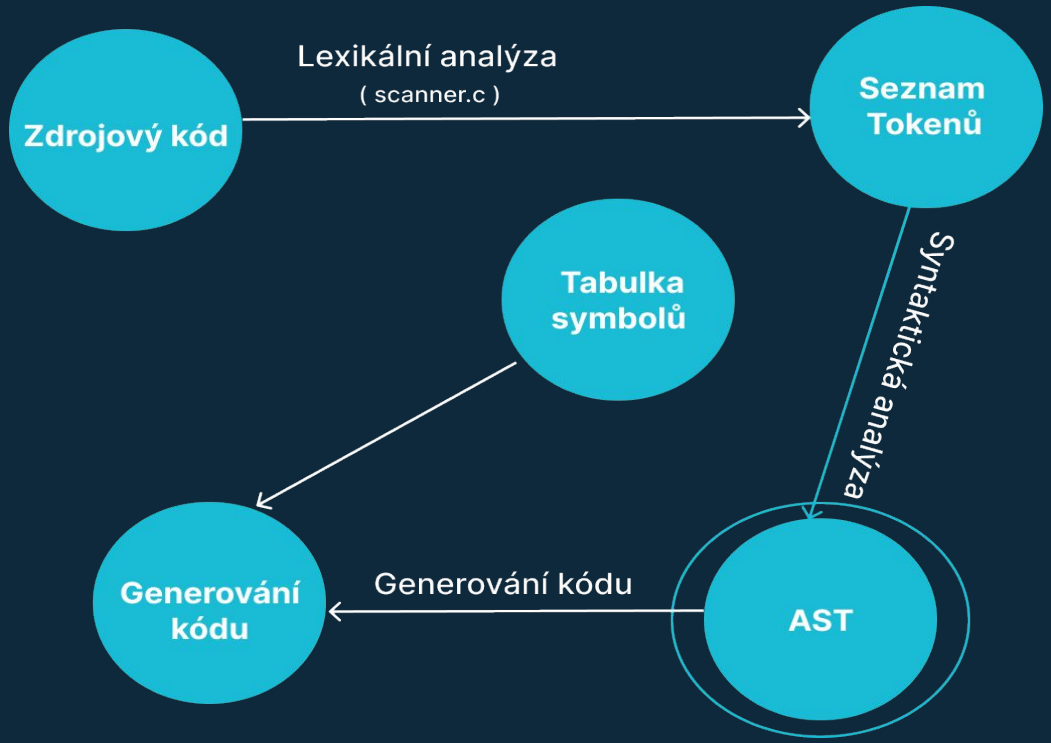
```
 1, TOKEN_VAR, 'var'
 2, TOKEN_IDENTIFIER, 'i'
 3, TOKEN_COLON, ':'
 4, TOKEN_TYPE_INT, 'Int'
 5, TOKEN_EQUAL, '='
 6, TOKEN_INTEGER, '2'
 7, TOKEN_STAR, '*'
 8, TOKEN_INTEGER, '2'
 9, TOKEN_PLUS, '+'
10, TOKEN_INTEGER, '2'
11, TOKEN_EOL
12, TOKEN_EOL
13, TOKEN_WHILE, 'while'
14, TOKEN_LEFT_PAREN, '('
15, TOKEN_IDENTIFIER, 'i'
16, TOKEN_GREATER, '>'
17, TOKEN_INTEGER, '0'
```

- Uložení Lexem do dynamického pole
- Zároveň vyhledává lexikální chyby

- Pole se seznamem tokenů se přenáší do další fáze

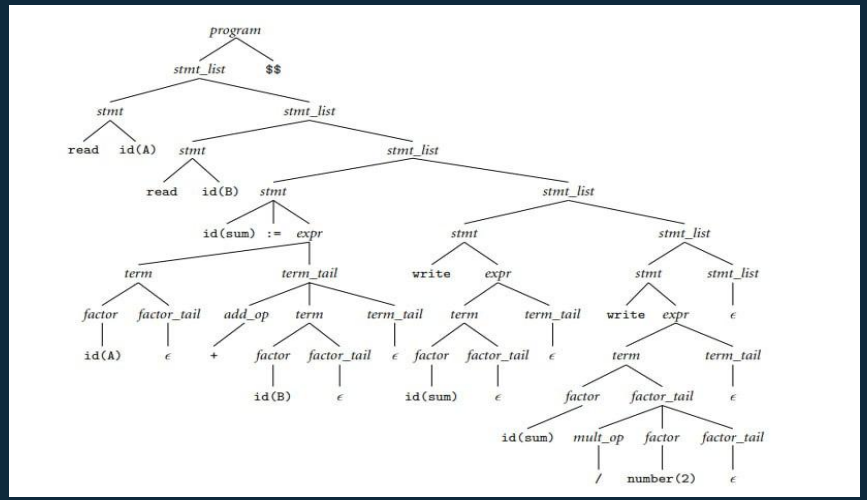


Syntaktická analýza



Syntaktická analýza

Z nalezených tokenů získáme pohled jako Abstraktní Syntaktický Strom

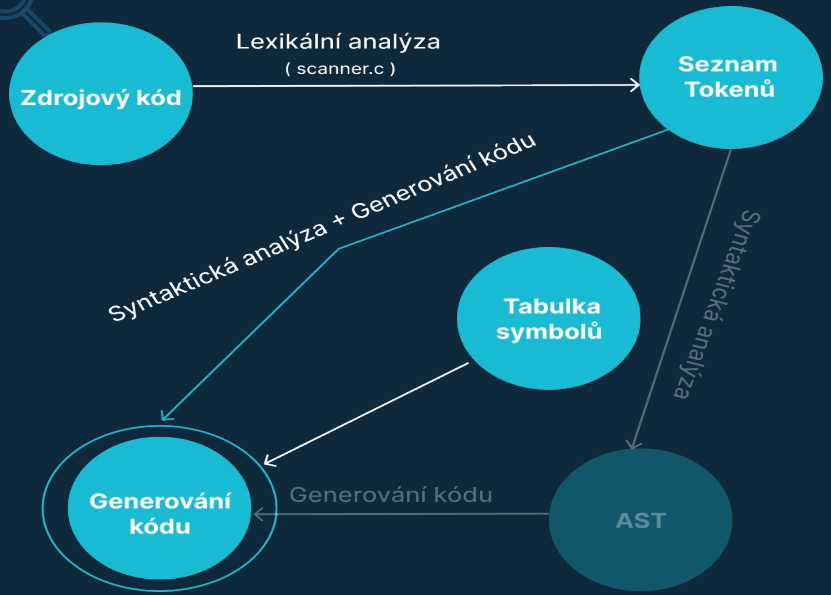


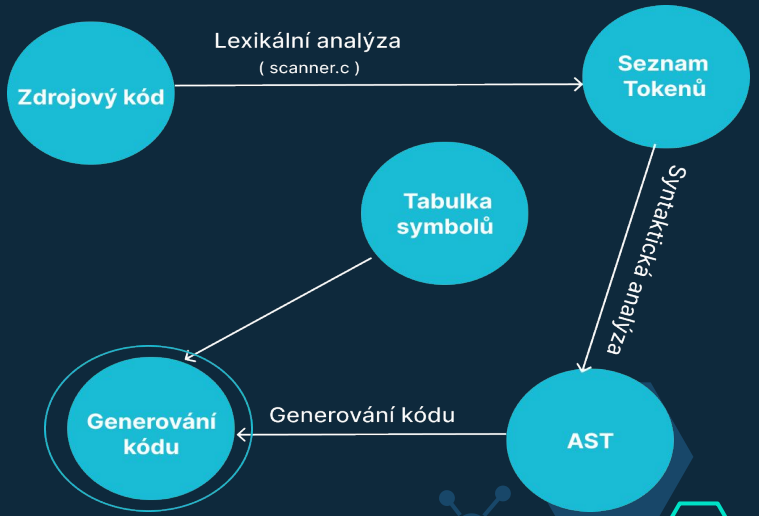
- Samotná datová struktura pro strom není vytvořena - používá se rekurzivní volání funkcí
- Zde také-práce se syntaktickými chybami

```
static void statement(bool isFirstFrame, bool isInWhile, Token* funcName) {
    if (match(TOKEN_IF)) {
        ifStatement(funcName, isInWhile);
    } else if (match(TOKEN_RETURN)) {
        returnStatement(funcName);
    } else if (match(TOKEN_WHILE)) {
        whileStatement(funcName, isInWhile);
    } else if (match(TOKEN_LEFT_BRACE)) {
        block(funcName, false);
    } else {
        expressionStatement();
    }
}
```

Generování kódu

- Generování kódu probíhá současně s průchodem stromu (AST)





Generování kódu

Získáme cílový kód, kde je virtuální stroj vnímán jako Stack machine

```
.IFJcode23

DEFVAR GF@i
PUSHS int@2
PUSHS int@2
MULS
PUSHS int@2
ADDS
POPS GF@i
JUMP $$main
LABEL $$main
CREATEFRAME
PUSHFRAME
LABEL whileStart0
PUSHS GF@i
```

- K nalezení sémantických chyb a generování kódu byly použity tabulky symbolů.
- V naší implementaci - této tabulky proměnných a funkcí, ve kterých jsou uloženy potřebné informace.

O práci v týmu

Unit testy

Pro testování byly použity-Unit testy

GitHub

Pro spravy verzí jsme použili Git a GitHub.

Komunikace

Osobní setkání a diskuse o malých otázkách / úlohách prostřednictvím sociální sítě v naší skupině

Děkujeme za pozornost