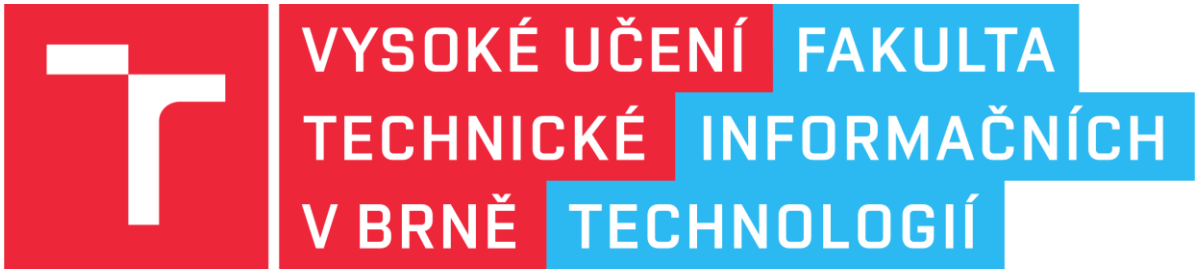



# Dokumentace projektu z předmětů IFJ a IAL:

*Implementace překladače imperativního jazyka IFJ23*

*xanton07, Varianta TRP-izp*

**Vedoucí týmu:**

| | | |
|---|---|---|
| Marek Antoňů | xanton07 | 25% |

**Členové týmu:**

| | | |
|---|---|---|
| Matěj Lepeška | xlepes00 | 25% |
| Petr Plíhal | xpliha02 | 25% |
| Tomáš Řezníček | xrezni33 | 25% |

# Obsah

1. Úvod ... 3
1.1. Cíl projektu ... 3
1.2. Rozdělení práce ... 3
1.3. Verzovací systém a komunikace v týmu ... 3
2. Lexikální analýza ... 4
2.1. Návrh FSM ... 4
2.2. Implementace ... 4
2.3. Úpravy a překážky ... 4
3. Syntaktická analýza ... 5
3.1. Rekurzivní syntaktická analýza ... 5
3.2. Precedenční syntaktická analýza ... 5
4. Sémantická analýza ... 6
5. Generování výstupu ... 6
6. Tabulka symbolů ... 7
7. Další abstraktní datové struktury ... 7
7.1. Zásobník tokenů ... 7
8. LL gramatika ... 8
9. LL tabulka ... 10
10. Precedenční tabulka ... 11
11. FSM ... 12

# 1. Úvod

## 1.1. Cíl projektu

Cílem našeho projektu je implementace překladače jazyka IFJ23, který vychází z jazyka Swift. Pro tento úkol jsme využili programovací jazyk C.

## 1.2. Rozdělení práce

Práce na projektu byla rozdělena následovně:

**Marek Antoňů:**
Syntaktická analýza pomocí LL gramatiky, generování kódu

**Tomáš Řezníček:**
Tabulka symbolů a další abstraktní datové struktury, výrazy, dokumentace

**Petr Plíhal:**
Lexikální analýza, testy, makefile, dokumentace

**Matěj Lepeška:**
Výrazy, dokumentace

## 1.3. Verzovací systém a komunikace v týmu

Na začátku spolupráce proběhlo osobní setkání, při kterém byla rozdělena práce mezi jednotlivé členy týmu. Poté se hlavními komunikačními kanály stal discord, a to jak ve variantě zpráv, tak i pravidelných hovorů, určených pro řešení aktuálně vzniklých problémů, a github, kde každý pracoval ve vlastní větvi, kterou posléze spojil s main větví, ve které se nacházel funkční a oddebugovaný kód.

# 2. Lexikální analýza

Lexikální analyzátor jsme implementovali ve čtyřech fázích – návrh konečného stavového automatu, implementace inspirována konečným automatem, úpravy a testování nedostatků.

Implementace lexikálního analyzátoru je v souborech **`scanner.c`** a **`scanner.h`**

## 2.1. Návrh FSM

Prvotní návrh konečného automatu posloužil jako předloha pro samotnou implementaci. V pozdějších fázích byly některé stavy nahrazeny funkcemi, v grafu FSM jsme je ponechali pro přehlednost a tyto skutečnosti do něj vyznačili.

## 2.2. Implementace

Nejdůležitější implementovanou částí je funkce **`int get_token(data_type_token *token)`**, která vyplňuje strukturu tokenu podle následujícího lexému a jejíž návratová hodnota udává, jestli je načtený lexém korektní. Funkce také ignoruje komentáře a bílé znaky, pokud nejsou součástí některého tokenu.

Ostatní pomocné funkce slouží například pro zpracování částí lexémů, inicializaci scanneru, rušení tokenů a ladění chyb.

## 2.3. Úpravy a překážky

Úpravy byly časově nejnáročnější částí práce na lexikálním analyzátoru. V porovnání s prvotní verzí, jak FSM, tak implementace, došlo k značenému množství změn, například:

V původním návrhu mělo každé klíčové slovo svůj typ tokenu, což později stěžovalo práci s nimi, takže jsme **`attribute`** rozšířili z prvku typu **`char *`** na oddělenou datovou strukturu **`attribute`**, ve které bylo klíčové slovo prvkem struktury. Mimo to jsme dodali některá opomenutá klíčová slova.

Další změnou bylo doplnění informací k některým typům tokenů do struktury **`attribute`**. Původně byl **`attribute`** čistě polem znaků, ve kterém byly užitečné informace pro vybrané tokeny. Tento přístup se rychle ukázal jako nepraktický a přidali jsme tak do struktury **`attribute`** prvky **`*string`**, **`integer`**, **`decimal`**.

Jednou z posledních změn byla úprava prvku **`newline_before`**, který původně zaznamenával, že je token typu identifikátor prvním tokenem na novém řádku, později byla tato informace zaznamenávána i pro ostatní typy tokenů.

# 3. Syntaktická analýza

## 3.1. Rekurzivní syntaktická analýza

Pro syntaktickou analýzu programu používáme rekurzivní sestup, implementace se nachází v *analysis.c*. Zde je implementována funkce pro každý neterminál, kromě expr ten je řešen precedenční syntaktickou analýzou. Jednotlivá pravidla LL-gramatiky jsou vybírána podle aktuálně načteného tokenu, pro některé není pouze token dostatečný a je potřeba přistoupit do tabulky symbolů (např. přiřazení do proměnné a volání uživatelské funkce). Tento výběr se děje podle precedenční tabulky LL-gramatiky. Jediným rozdílem od LL-gramatiky a implementace je spojení neterminálů <command> a <command_func>, jelikož jediný rozdíl mezi těmito neterminály je pravidlo s RETURN. Proto byly spojeny a pomocí pomocné proměnné ve struktuře *parser_data*. Pomocí této struktury jsou také předávány data nutné pro syntaktickou a sémantickou analýzu.

## 3.2. Precedenční syntaktická analýza

Precedenční analyzátor využívá při analýze svoji tabulku (viz. Precedenční tabulka) a dále také pracuje se zásobníkem, který je definovaný v souborech *stack.c* a *stack.h.* Precedenční analýza je definována ve funkci *expr*(), kde se nejprve na vrchol zásobníku uloží $ a dále se pomocí cyklu načítá terminál na vrcholu zásobníku a token ze vstupu. Poté se pomocí funkce *preced_table_index*() zjistí index v precedenční tabulce. Pokud je vrácen index -1, tak výraz je celý zpracován, jinak se vybere znak (<, >, = , “ “), který se nachází na pozici indexů. Podle znaku z precedenční tabulky se poté provádí jednotlivé operace. Redukce se provádí pomocí funkce *reduce*(), která podle počtu operandů, které je potřeba zredukovat, vybere příslušnou funkci pro provedení redukce. Počet operandů se pro redukci zjišťuje ve funkci *amountToReduce*(). Při redukci jednoho operandu se volá funkce *reduceOneByRule*(). Při redukci dvou operandů se volá funkce *reduceTwoByRule*() a při redukci tří operandů se volá funkce *reduceThreeByRule*(). Ve funkci *reduceThreeByRule*() se podle druhého operandu vybírá příslušné pravidlo pro redukci. Dále se zde pomocí testovacích funkcí provádí kontrola prvního a třetího operandu. Funkce vrací použité pravidlo pro redukci. Činnost funkcí reduceTwoByRule a reduceOneByRule je podobná.

# 4. Sémantická analýza

Sémantická analýza je realizována zároveň s průběhem syntaktické analýzy. Za pomoci dat předávaných uvnitř struktury *parser_data* a tabulky symbolů. Pro zajištění sémantické analýzy rámce proměnné byla implementována struktura *var_list*, tato struktura se chová jako zásobník seznamů. Pro názvy parametrů uživatelských funkcí byla implementována struktura *arg_list*, tato struktura je seznam s ukazatelem na aktivní prvek. Tyto struktury jsou implementovány ve stejnojmenných souborech.

# 5. Generování výstupu

Generování je implementováno stejně jako sémantická analýza během syntaktické analýzy. Generování je implementováno v *generator.c*, generování je rozděleno na potřebné funkce. Vestavěné funkce jsou generovány najednou při volání začátku generátoru.

# 6. Tabulka symbolů

Tabulka symbolů je implementovaná pomocí hashovací tabulky s implicitním zřetězením synonym. Tabulka má pevně danou velikost, která je ale dostatečná pro naprostou většinu programů. Kvůli výběru varianty s implicitním zřetězením bylo nutné vedle funkce vypočítávající hash, neboli index, na kterém se v tabulce nachází požadovaný prvek nebo jeho synonymum, vytvořit i funkci, která počítá krok, po kterém program prochází prvky tabulky, než nalezne buďto hledaný prvek, nebo prázdné místo pro uložení nového. Pro tento krok bylo zvoleno, že se bude nacházet v intervalu <1, 16>. Dále bylo výběrem této varianty zapříčiněno, že prvky nejsou odstraňovány, ale pouze nastaveny jako neaktivní, v případě že již nejsou dále při běhu programu potřebné. Všechny prvky jsou odstraněny na konci programu, kdy se naráz uvolní celá tabulka a všechny její prvky.

# 7. Další abstraktní datové struktury

## 7.1. Zásobník tokenů

V průběhu tvorby projektu bylo nutné implementovat i zásobník pro ukládání tokenů. Tento zásobník je implementován tak, že lze využívat pouze nejaktuálněji vložený prvek. Práci se zásobníkem obstarávají funkce push, která uloží token na zásobník, pop, která vymaže token z vrcholu zásobníku a funkce top, která vrátí token nacházející se na vrcholu zásobníku.

## 8. LL gramatika

<prog> FUNC ID (<param>)<rtype>{<command_func>}<prog>
<prog> EOF
<prog> <command><prog>

<rtype> -><type>
<rtype> epsilon

<param> <param_val><param_more>
<param> epsilon

<param_more> ,<param_val><param_more>
<param_more> epsilon

<param_val> ID ID :<type>

<command> VAR ID <dtype><init><command>
<command> LET ID <dtype><init><command>
<command> IF<expr>{<command>}ELSE{<command>}<command>
<command> WHILE<expr>{<command>}<command>
<command> ID = <assign><command>
<command> <func_call><command>
<command> epsilon

<init> = <assign>
<init> epsilon

<assign> <expr>
<assign> <func_call>

<func_call> ID (<arg>)
<func_call> READSTRING (<arg>)
<func_call> READINT (<arg>)
<func_call> READDOUBLE (<arg>)
<func_call> WRITE (<arg>)
<func_call> INT2DOUBLE (<arg>)
<func_call> DOUBLE2INT (<arg>)
<func_call> LENGTH (<arg>)
<func_call> SUBSTRING (<arg>)
<func_call> ORD (<arg>)
<func_call> CHR (<arg>)

<dtype> :<type>
<dtype> epsilon

<arg> <arg_val><arg_more>
<arg> epsilon

<arg_more> ,<arg_val><arg_more>
<arg_more> epsilon

<arg_val> ID : <val>
<arg_val> <val>

<type> STRING
<type> INT
<type> DOUBLE
<type> STRING?
<type> INT?
<type> DOUBLE?

<val> STRING_VAL
<val> INT_VAL
<val> DOUBLE_VAL
<val> ID
<val> NIL

<command_func> VAR ID <dtype><init><command_func>
<command_func> LET ID <dtype><init><command_func>
<command_func> RETURN<expr><command_func>
<command_func> IF<expr>{<command_func>}ELSE{<command_func>}<command_func>
<command_func> WHILE<expr>{<command_func>}<command_func>
<command_func> ID = <assign><command_func>
<command_func> <func_call><command_func>
<command_func> epsilon

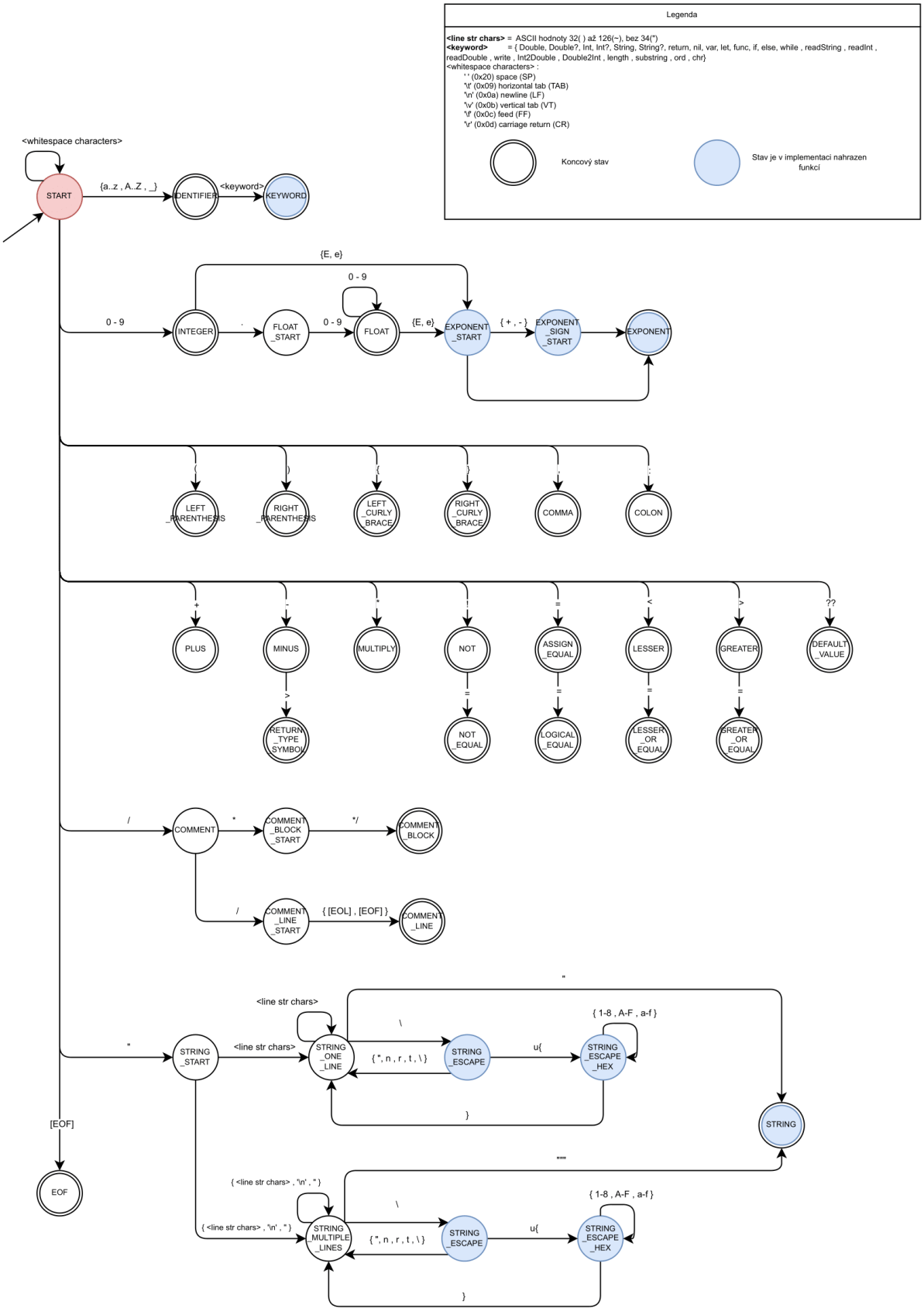
# 9. LL tabulka

| | <prog> | <rtype> | <param> | <param_more> | <param_val> | <command> | <assign> | <func_call> | <dtype> | <arg> | <arg_more> | <arg_val> | <type> | <val> | <init> | <command_func> |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FUNC | 1 | | | | | 17 | | | 34 | | | | | | 19 | |
| EOF | 2 | | | | | 17 | | | 34 | | | | | | 19 | |
| VAR | 3 | | | | | 11 | | | 34 | | | | | | 19 | 52 |
| LET | 3 | | | | | 12 | | | 34 | | | | | | 19 | 53 |
| RETURN | | | | | | | | | 34 | | | | | | 19 | 54 |
| IF | 3 | | | | | 13 | | | 34 | | | | | | 19 | 55 |
| WHILE | 3 | | | | | 14 | | | 34 | | | | | | 19 | 56 |
| READSTRING | 3 | | | | | 16 | 21 | 23 | 34 | | | | | | 19 | 58 |
| READINT | 3 | | | | | 16 | 21 | 24 | 34 | | | | | | 19 | 58 |
| READDOUBLE | 3 | | | | | 16 | 21 | 25 | 34 | | | | | | 19 | 58 |
| WRITE | 3 | | | | | 16 | 21 | 26 | 34 | | | | | | 19 | 58 |
| INT2DOUBLE | 3 | | | | | 16 | 21 | 27 | 34 | | | | | | 19 | 58 |
| DOUBLE2INT | 3 | | | | | 16 | 21 | 28 | 34 | | | | | | 19 | 58 |
| LENGTH | 3 | | | | | 16 | 21 | 29 | 34 | | | | | | 19 | 58 |
| SUBSTRING | 3 | | | | | 16 | 21 | 30 | 34 | | | | | | 19 | 58 |
| ORD | 3 | | | | | 16 | 21 | 31 | 34 | | | | | | 19 | 58 |
| CHR | 3 | | | | | 16 | 21 | 32 | 34 | | | | | | 19 | 58 |
| STRING | | | | | | | | | | | | | 41 | | | |
| INT | | | | | | | | | | | | | 42 | | | |
| DOUBLE | | | | | | | | | | | | | 43 | | | |
| STRING? | | | | | | | | | | | | | 44 | | | |
| INT? | | | | | | | | | | | | | 45 | | | |
| DOUBLE? | | | | | | | | | | | | | 46 | | | |
| STRING_VAL | | | | | | | 20 | | | 35 | | 40 | | 47 | | |
| INT_VAL | | | | | | | 20 | | | 35 | | 40 | | 48 | | |
| DOUBLE_VAL | | | | | | 15, 16 | 20 | | | 35 | | 40 | 39, 40 | 49 | | 57, 58 |
| ID | 3 | | 6 | | 10 | | 20, 21 | 22 | | 35 | | | | 50 | 19 | |
| ( | | 5 | | | | | | | | | | | | | | |
| ) | | | 7 | 9 | | 17 | | | | | 38 | | | | 19 | 59 |
| , | | | | | | | | | | 36 | | | | | | |
| : | | | | | | | | | 33 | | | | | | | |
| = | | | | | | | | | 34 | | | | | | 18 | |
| -> | | 4 | | | | | | | | | | | | | | |
| ; | | | | 8 | | | | | | | 37 | | | | | |

## 10. Precedenční tabulka

| | ! | * / | + - | == != < > <= >= | ?? | ( | ) | i | $ |
|---|---|---|---|---|---|---|---|---|---|
| ! | | < | > | > | > | < | > | < | > |
| * / | < | > | > | > | > | < | > | < | > |
| + - | < | < | > | > | > | < | > | < | > |
| == != < > <= >= | < | < | < | | > | < | > | < | > |
| ?? | < | < | < | < | < | < | > | < | > |
| ( | < | < | < | < | < | > | = | < | |
| ) | > | > | > | > | > | | < | | > |
| i | > | > | > | > | > | | > | | > |
| $ | < | < | < | < | < | < | | < | |

# 11. FSM


Legenda
<line str chars> = ASCII hodnoty 32( ) až 126(~), bez 34(")
<keyword> = { Double, Double?, Int, Int?, String, String?, return, nil, var, let, func, if, else, while , readString , readInt , readDouble , write , Int2Double , Double2Int , length , substring , ord , chr}
<whitespace characters> :
' ' (0x20) space (SP)
'\t' (0x09) horizontal tab (TAB)
'\n' (0x0a) newline (LF)
'\v' (0x0b) vertical tab (VT)
'\f' (0x0c) feed (FF)
'\r' (0x0d) carriage return (CR)
Koncový stav
Stav je v implementaci nahrazen funkcí
<whitespace characters>
START
{a..z , A..Z , _}
IDENTIFIER
<keyword>
KEYWORD
{E, e}
0 - 9
0 - 9
0 - 9
INTEGER
.
FLOAT _START
FLOAT
{E, e}
EXPONENT _START
{ + , - }
EXPONENT _SIGN _START
EXPONENT
(
)
{
}
,
:
LEFT _PARENTHESIS
RIGHT _PARENTHESIS
LEFT _CURLY _BRACE
RIGHT _CURLY _BRACE
COMMA
COLON
+
-
*
!
=
<
>
??
PLUS
MINUS
MULTIPLY
NOT
ASSIGN _EQUAL
LESSER
GREATER
DEFAULT _VALUE
>
=
=
=
=
RETURN _TYPE _SYMBOL
NOT _EQUAL
LOGICAL _EQUAL
LESSER _OR _EQUAL
GREATER _OR _EQUAL
/
COMMENT
*
COMMENT _BLOCK _START
*/
COMMENT _BLOCK
/
COMMENT _LINE _START
{ [EOL] , [EOF] }
COMMENT _LINE
"
STRING _START
<line str chars>
STRING _ONE _LINE
\
{ ", n , r , t , \ }
STRING _ESCAPE
u{
{ 1-8 , A-F , a-f }
STRING _ESCAPE _HEX
}
"
STRING
[EOF]
EOF
"""
{ <line str chars> , "\n" , " }
STRING _MULTIPLE _LINES
{ <line str chars> , "\n" , " }
\
{ ", n , r , t , \ }
STRING _ESCAPE
u{
{ 1-8 , A-F , a-f }
STRING _ESCAPE _HEX
}