

دانشگاه اصفهان
دانشکده مهندسی کامپیوتر
گروه نرم افزار

گزارش فاز اول پروژه درس طراحی کامپایلر

عنوان

طراحی و پیاده سازی **Lexical Analyzer** برای زبان برنامه نویسی مایک

اعضای گروه
نفیسه مومنی
نگار نادیان
پریسا شهابی نژاد

استاد درس
دکتر مریم اسدی

دی ۱۴۰۰

مقدمه

کامپایلر نرم افزار کامپیوتری است که کدهای کامپیوتری نوشته شده در یک زبان برنامه نویسی (زبان مبدأ) را به زبان برنامه نویسی دیگر (زبان مقصد) تبدیل می کند. نام کامپایلر در درجه اول برای برنامه هایی استفاده می شد که کد منبع را از یک زبان برنامه نویسی سطح بالا به زبان سطح پایین تر (مانند زبان اسمبلی، کد شیء یا کد ماشین) برای ایجاد یک برنامه اجرایی ترجمه می کنند.

مراحل کامپایلر

از نظر مفهومی، یک کامپایلر در فازهایی عمل می کند که هر یک از آنها برنامه منبع را از یک نمایش به دیگری تبدیل می کند. مراحل به شرح زیر است:

تحلیل و بررسی

۱. تحلیل لغوی
۲. تجزیه
۳. تحلیل معنایی
۴. تولید کد میانی

سنتز

۱. بهینه سازی کد
2. تولید کد

اهداف

هدف این پروژه طراحی و پیاده سازی مراحل مختلف یک کامپایلر برای زبان برنامه نویسی مایک است. ساختارهای زیر توسط این کامپایلر مدیریت می شوند:

- Data Types: انواع داده های char، int
- Comments: نظرات تک خطی و چند خطی
- کلمات کلیدی
- Looping Constructs: از حلقه های for و while پشتیبانی می کند.
- ساختارهای شرطی: عبارات if...else
- عملگرها
- جداکننده ها: (,), Dot (.)

پیاده‌سازی

تحلیلگر لغوی پیاده‌سازی شده از ۳ بخش زیر تشکیل شده است:

Definition section

%%

Rules section

%%

C code section

بخش تعریف، فایل‌های هدر نوشته شده به زبان C را وارد می‌کند. همچنین امکان نوشتن هر کد C در اینجا وجود دارد. بخش قوانین، الگوهای عبارت منظم را با دستورات C مرتبط می‌کند. وقتی lexer متنی را در ورودی می‌بیند که با الگوی داده شده مطابقت دارد، کد C مرتبط را اجرا می‌کند. بخش کد C شامل عبارات و توابع C است که حاوی کدهایست که توسط قوانین در بخش قوانین فراخوانی شده‌اند.

Definition section:

```
%option noyywrap

/* Definition Section */
%{
    #include <stdio.h>
    FILE *output;
    int line_count = 1;
}%

NEWLINE \r?\n
ANYTHING_EXCEPT_NEWLINE [^\r\n]

ANYTHING ({ANYTHING_EXCEPT_NEWLINE}|{NEWLINE})
BLOCK_COMMENT \#\[^\[|\r\n\|(\![^\[|\r\n\|)\)*\!+\#
SINGLE_LINE_COMMENT "\#"[^\].*
```

Rules section: Keywords

```
/* Rule Section */
%%
{NEWLINE} {line_count++;}
[ \f\v\r\t]+ {}

{BLOCK_COMMENT} |
{SINGLE_LINE_COMMENT} {}

"int" {
    fprintf(output, "TOKEN_INT\n");
}
"char" {
    fprintf(output, "TOKEN_CHAR\n");
}
"if" {
    fprintf(output, "TOKEN_IF\n");
}
"else" {
    fprintf(output, "TOKEN_ELSE\n");
}
"elseif" {
    fprintf(output, "TOKEN_ELSEIF\n");
}
"while" {
    fprintf(output, "TOKEN_WHILE\n");
}
"for" {
    fprintf(output, "TOKEN_FOR\n");
}
"return" {
    fprintf(output, "TOKEN_RETURN\n");
}
"void" {
    fprintf(output, "TOKEN_VOID\n");
}
"main" {
    fprintf(output, "TOKEN_MAIN\n");
}
"continue" {
    fprintf(output, "TOKEN_CONTINUE\n");
}
"break" {
    fprintf(output, "TOKEN_BREAK\n");
}
```

Rules section: Operators

```
"=" {  
    fprintf(output, "TOKEN_ASSIGN\n");  
}  
"<" {  
    fprintf(output, "TOKEN_LESS\n");  
}  
"<=" {  
    fprintf(output, "TOKEN_LESSEQUAL\n");  
}  
"==" {  
    fprintf(output, "TOKEN_EQUAL\n");  
}  
"!=" {  
    fprintf(output, "TOKEN_NOTEQUAL\n");  
}  
">" {  
    fprintf(output, "TOKEN_GREATER\n");  
}  
">=" {  
    fprintf(output, "TOKEN_GREATEREQUAL\n");  
}  
"|" {  
    fprintf(output, "TOKEN_OR\n");  
}  
"&" {  
    fprintf(output, "TOKEN_AND\n");  
}  
"^" {  
    fprintf(output, "TOKEN_XOR\n");  
}  
"||" {  
    fprintf(output, "TOKEN_OR_OP\n");  
}  
"&&" {  
    fprintf(output, "TOKEN_AND_OP\n");  
}
```

Rules section: Operators

```
    "!" {
        fprintf(output, "TOKEN_NOT\n");
    }
    "+" {
        fprintf(output, "TOKEN_ADD\n");
    }
    "-" {
        fprintf(output, "TOKEN_SUB\n");
    }
    "*" {
        fprintf(output, "TOKEN_MUL\n");
    }
    "/" {
        fprintf(output, "TOKEN_DIV\n");
    }
    "++" {
        fprintf(output, "TOKEN_INC_OP\n");
    }
    "--" {
        fprintf(output, "TOKEN_DEC_OP\n");
    }
    "+=" {
        fprintf(output, "TOKEN_ADD_ASSIGN\n");
    }
    "-=" {
        fprintf(output, "TOKEN_SUB_ASSIGN\n");
    }
    "*=" {
        fprintf(output, "TOKEN_MUL_ASSIGN\n");
    }
    "/=" {
        fprintf(output, "TOKEN_DIV_ASSIGN\n");
    }
    "&=" {
        fprintf(output, "TOKEN_AND_ASSIGN\n");
    }
    "|=" {
        fprintf(output, "TOKEN_OR_ASSIGN\n");
    }
    "^=" {
        fprintf(output, "TOKEN_XOR_ASSIGN\n");
    }
}
```

Rules section: Delimiters

```

" ." {
    fprintf(output, "TOKEN_DOT\n");
}
" (" {
    fprintf(output, "TOKEN_LEFTPAREN\n");
}
")" {
    fprintf(output, "TOKEN_RIGHTPAREN\n");
}
"{" {
    fprintf(output, "TOKEN_LBRACE\n");
}
"}" {
    fprintf(output, "TOKEN_RBRACE\n");
}
"[" {
    fprintf(output, "TOKEN_LBRACK\n");
}
"]" {
    fprintf(output, "TOKEN_RBRACK\n");
}
", " {
    fprintf(output, "TOKEN_COMMA\n");
}

```


Rules section: Patterns

```
\"([^\\"\\r\\n]|[\\\"]{ANYTHING})*\" {
    fprintf(output, "TOKEN_CHAR_CONST\\n");
}
[-|+]?([1-9][0-9]*|0) {
    fprintf(output, "TOKEN_INT_CONST\\n");
}
[a-zA-Z_][a-zA-Z0-9_]* {
    fprintf(output, "TOKEN_IDENTIFIER\\n");
}
[0-9]([a-zA-Z_]|[0-9])* {
    fprintf(output, "Error: Invalid numeric constant or identifier.\\n");
}

{ANYTHING} {
    if(yytext[0]=='#')
    {
        fprintf(output,"Error: Unmached comment at line no. %d\\n",line_count);
    }
    else if(yytext[0]=='"')
    {
        fprintf(output,"Error: Incomplete character at line no. %d\\n",line_count);
    }
    else
    {
        fprintf(output,"Error: Unrecognized character at lone no. %d.\\n", line_count);
    }
    return 0;
}

%%
```

Code section:

```
/* Code Section */
int main (){
    FILE* input = fopen("./Test Cases/test1.txt", "r");
    yyin = input;
    output = fopen("Phase1_Tokens.txt", "w");
    fprintf(output, "The resulted tokens are:\n");
    yylex();
    fclose(output);
    fclose(input);
    return 0;
}
```

اسکریپت flex پیاده‌سازی شده موارد زیر را از ورودی تشخیص می‌دهد:

- Single-line comments
 - Statements processed: #.....
- Multi-line comments
 - Statements processed: #!.....!#, #!...#!...!#
- Errors for unmatched comments
 - Statements processed : #!.....
- Parentheses (all types)
 - Statements processed : (.), {..}, [..]
- Operators
- Errors for incomplete characters
 - Statements processed : char a = "abcd
- Literals (integer, character)
 - Statements processed : int, char
- Keywords
 - Statements processed : if, else, void, while, int, break and so on.
- Identifiers
 - Statements processed : a, abc, a_b, a12b4
- Errors for any invalid character used that is not in C character set

مثال:

Input 1

```
int main(){
    int n,i.
    char ch.##Character Datatype

    for (i=0,i<n,i++){
        if(i<10){
            int x.
            while(x<10){
                x+=5.
            }
        }
    }
    #!
    This File Contains Test cases about Datatypes,Keyword,Identifier,Nested For and while loop,
    Conditional Statement,Single line Comment,MultiLine Comment etc.!!
}
```

Output 1

```
The resulted tokens are:
TOKEN_INT
TOKEN_MAIN
TOKEN_LEFTPAREN
TOKEN_RIGHTPAREN
TOKEN_LBRACE
TOKEN_INT
TOKEN_IDENTIFIER
TOKEN_COMMA
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_CHAR
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_FOR
TOKEN_LEFTPAREN
TOKEN_IDENTIFIER
TOKEN_ASSIGN
TOKEN_INT_CONST
TOKEN_COMMA
TOKEN_IDENTIFIER
TOKEN_LESS
TOKEN_IDENTIFIER
TOKEN_COMMA
TOKEN_IDENTIFIER
TOKEN_INC_OP
TOKEN_RIGHTPAREN
TOKEN_LBRACE
TOKEN_IF
TOKEN_LEFTPAREN
TOKEN_IDENTIFIER
TOKEN_LESS
TOKEN_INT_CONST
TOKEN_RIGHTPAREN
TOKEN_LBRACE
TOKEN_INT
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_WHILE
TOKEN_LEFTPAREN
TOKEN_IDENTIFIER
TOKEN_LESS
TOKEN_INT_CONST
TOKEN_RIGHTPAREN
TOKEN_LBRACE
TOKEN_IDENTIFIER
TOKEN_ADD_ASSIGN
TOKEN_INT_CONST
TOKEN_DOT
TOKEN_RBRACE
TOKEN_RBRACE
TOKEN_RBRACE
Error: Unmached comment at line no. 14
```

Input 1

```
#!/ struct pair{
    int a;
    int b;
}.!#

int fun(int x){
    return x*x.
}

int main(){
    int a=2,b,c,d,e,f,g,h.

    c=a+b.
    d=a*b.
    e=a/b.

    g=a&&b.
    h=a||b.
    h=a*(a+b).
    h=a*a+b*b.
    h=fun(b).

    #This Test case contains operator,structure,delimeters,Function.
}
```

Output 2

The resulted tokens are:

```
TOKEN_INT
TOKEN_IDENTIFIER
TOKEN_LEFTPAREN
TOKEN_INT
TOKEN_IDENTIFIER
TOKEN_RIGHTPAREN
TOKEN_LBRACE
TOKEN_RETURN
TOKEN_IDENTIFIER
TOKEN_MUL
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_RBRACE
TOKEN_INT
TOKEN_MAIN
TOKEN_LEFTPAREN
TOKEN_RIGHTPAREN
TOKEN_LBRACE
TOKEN_INT
TOKEN_IDENTIFIER
TOKEN_ASSIGN
TOKEN_INT_CONST
TOKEN_COMMA
TOKEN_IDENTIFIER
TOKEN_COMMA
TOKEN_IDENTIFIER
TOKEN_COMMA
TOKEN_IDENTIFIER
TOKEN_COMMA
TOKEN_IDENTIFIER
TOKEN_COMMA
TOKEN_IDENTIFIER
TOKEN_COMMA
TOKEN_IDENTIFIER
TOKEN_COMMA
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_IDENTIFIER
TOKEN_ASSIGN
TOKEN_IDENTIFIER
TOKEN_ADD
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_IDENTIFIER
TOKEN_ASSIGN
TOKEN_IDENTIFIER
TOKEN_MUL
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_IDENTIFIER
TOKEN_ASSIGN
TOKEN_IDENTIFIER
TOKEN_DIV
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_IDENTIFIER
TOKEN_ASSIGN
TOKEN_IDENTIFIER
TOKEN_AND_OP
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_IDENTIFIER
TOKEN_ASSIGN
TOKEN_IDENTIFIER
TOKEN_OR_OP
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_IDENTIFIER
TOKEN_ASSIGN
TOKEN_IDENTIFIER
TOKEN_MUL
TOKEN_LEFTPAREN
TOKEN_IDENTIFIER
TOKEN_ADD
TOKEN_IDENTIFIER
TOKEN_RIGHTPAREN
TOKEN_DOT
TOKEN_IDENTIFIER
TOKEN_ASSIGN
TOKEN_IDENTIFIER
TOKEN_MUL
TOKEN_ADD
TOKEN_IDENTIFIER
TOKEN_MUL
TOKEN_IDENTIFIER
TOKEN_DOT
TOKEN_IDENTIFIER
TOKEN_ASSIGN
TOKEN_IDENTIFIER
TOKEN_LEFTPAREN
TOKEN_IDENTIFIER
TOKEN_RIGHTPAREN
TOKEN_DOT
TOKEN_RBRACE
```

Input 3



```
int main()  
{  
    char ch= "z" .  
    ch="\t".  
    char c="ab  
d".  
    int var = 0.  
    for(int i = 0, i < 5, --i)  
    {  
        var = var * 10.  
    }  
    return 0.  
}
```

Output 3

The resulted tokens are:

TOKEN_INT

TOKEN_MAIN

TOKEN_LEFTPAREN

TOKEN_RIGHTPAREN

TOKEN_LBRACE

TOKEN_CHAR

TOKEN_IDENTIFIER

TOKEN_ASSIGN

TOKEN_CHAR_CONST

TOKEN_DOT

TOKEN_IDENTIFIER

TOKEN_ASSIGN

TOKEN_CHAR_CONST

TOKEN_DOT

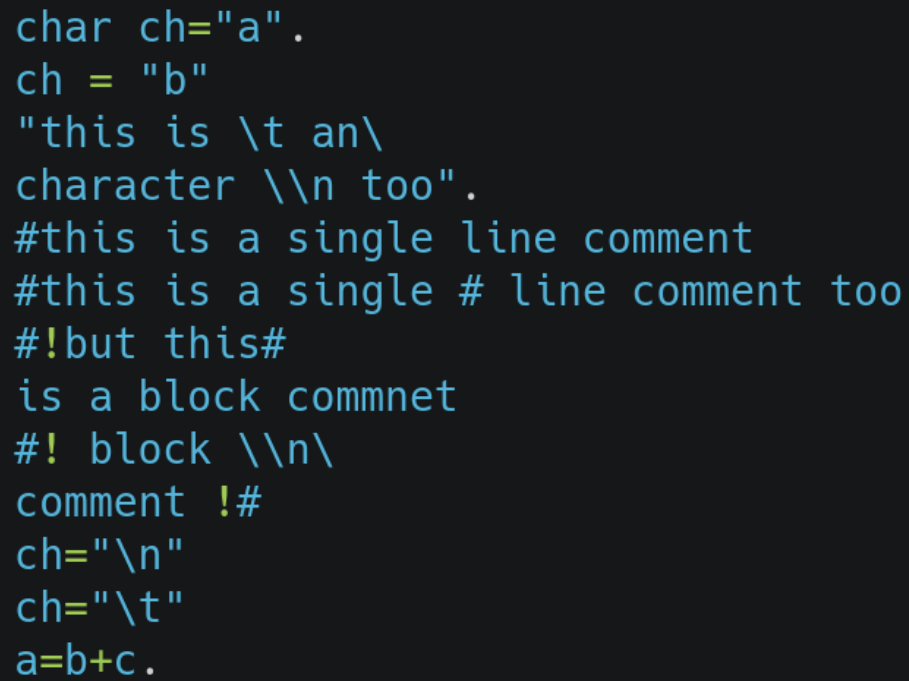
TOKEN_CHAR

TOKEN_IDENTIFIER

TOKEN_ASSIGN

Error: Incomplete character at line no. 5

Input 4



```
char ch="a".
ch = "b"
"this is \t an\
character \\n too".
#this is a single line comment
#this is a single # line comment too
#!but this#
is a block commnet
#! block \\n\
comment !#
ch="\n"
ch="\t"
a=b+c.
```

Output 4



The resulted tokens are:

```
TOKEN_CHAR  
TOKEN_IDENTIFIER  
TOKEN_ASSIGN  
TOKEN_CHAR_CONST  
TOKEN_DOT  
TOKEN_IDENTIFIER  
TOKEN_ASSIGN  
TOKEN_CHAR_CONST  
TOKEN_CHAR_CONST  
TOKEN_DOT  
TOKEN_IDENTIFIER  
TOKEN_ASSIGN  
TOKEN_CHAR_CONST  
TOKEN_IDENTIFIER  
TOKEN_ASSIGN  
TOKEN_CHAR_CONST  
TOKEN_IDENTIFIER  
TOKEN_ASSIGN  
TOKEN_IDENTIFIER  
TOKEN_ADD  
TOKEN_IDENTIFIER  
TOKEN_DOT
```

Input 5



Implicit Error that our Language doesn't support

```
int main() {  
    char @hello;  
    @hello = 'c';  
}
```

Output 5



The resulted tokens are:

TOKEN_INT

TOKEN_MAIN

TOKEN_LEFTPAREN

TOKEN_RIGHTPAREN

TOKEN_LBRACE

TOKEN_CHAR

Error: Unrecognized character at lone no. 3.