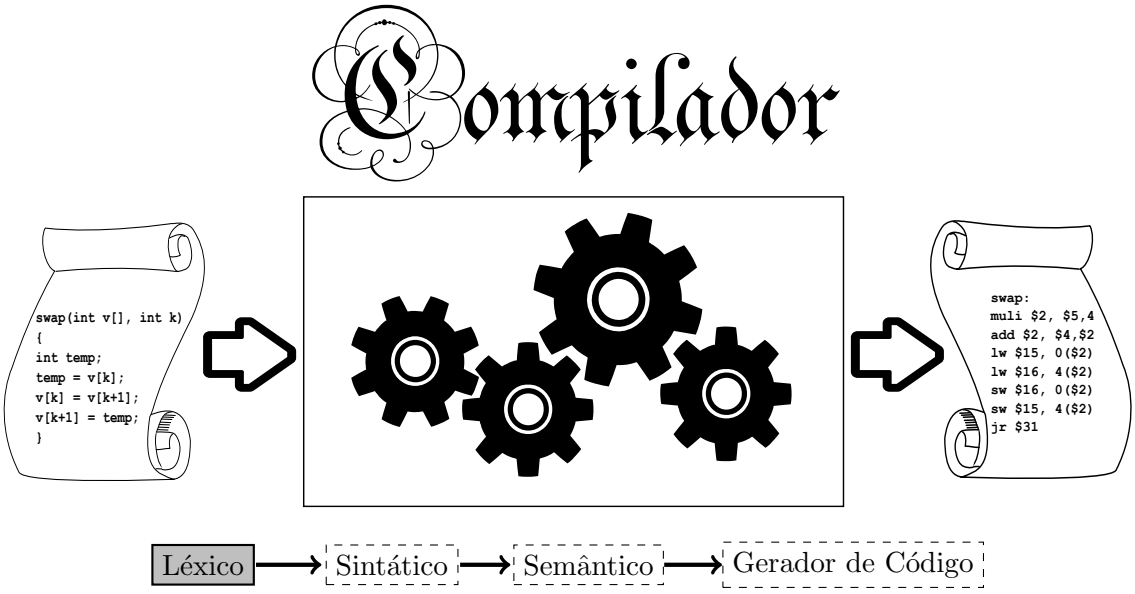
Universidade Estadual de Londrina

DC Departamento de Computação

1COP020 - Compilador C: Analisador Léxico



Utilizando a ferramenta Flex e a linguagem C, implemente um analisador léxico para um **subconjunto** de *tokens* da linguagem C. Para cada *token* reconhecido, você deverá imprimir a classificação do mesmo. A classificação dos *tokens* que você deve seguir encontra-se a seguir:

```
Identificadores: IDENTIFIER
Inteiros: NUM_INTEGER
Octais:   NUM_OCTAL
Hexadecimal: NUM_HEXA
String:  STRING
Caracter: CHARACTER

void     VOID
int      INT
char     CHAR
return   RETURN
break    BREAK
switch   SWITCH
case     CASE
default  DEFAULT
```

Versão deste texto: V. 2022.02

```
do        DO
while     WHILE
for       FOR
if        IF
else      ELSE
typedef   TYPEDEF
struct    STRUCT
+         PLUS
-         MINUS
*         MULTIPLY
/         DIV
%         REMAINDER
++        INC
--        DEC
&         BITWISE_AND
|         BITWISE_OR
~         BITWISE_NOT
^         BITWISE_XOR
!         NOT
&&        LOGICAL_AND
||        LOGICAL_OR
==        EQUAL
!=        NOT_EQUAL
<         LESS_THAN
>         GREATER_THAN
<=        LESS_EQUAL
>=        GREATER_EQUAL
>>        R_SHIFT
<<        L_SHIFT
=         ASSIGN
+=        ADD_ASSIGN
-=        MINUS_ASSIGN
;         SEMICOLON
,         COMMA
:         COLON
(         L_PAREN
)         R_PAREN
{         L_CURLY_BRACKET
}         R_CURLY_BRACKET
[         L_SQUARE_BRACKET
]         R_SQUARE_BRACKET
?         TERNARY_CONDITIONAL
#         NUMBER_SIGN
->        POINTER
printf    PRINTF
scanf     SCANF
define    DEFINE
exit      EXIT
```

Identificadores: os identificadores podem ser iniciados por letras maiúsculas ou minúsculas ou ainda o caractere *underscore*. A partir do segundo símbolo, números de 0 até 9 também podem aparecer na formação do identificador. Os identificadores podem possuir um tamanho máximo de 255 caracteres.

Devem ser removidos:

```
Espaços em branco
Comentários de uma linha:  (//)
Comentários de múltiplas linhas:  (/* ...  */)
```

Também devem ser detectados comentários de múltiplas linhas que são iniciados e não estão finalizados.
Os *tokens* reconhecidos devem ser impressos um por linha. Quando um erro for detectado, deve-se mostrar a linha e a coluna onde o erro ocorreu. Considere o seguinte exemplo:

Exemplo de entrada:

```
if(@)
{
   a
}
```

Saída esperada:

```
IF
L_PAREN
error:lexical:1:4: @
R_PAREN
L_CURLY_BRACKET
IDENTIFIER(a)
R_CURLY_BRACKET
```

Observe que na saída esperada, a mensagem de erro apresentada foi:

```
error:lexical:1:4: @
```

onde o primeiro número indica a linha onde o erro ocorreu e o segundo número indica a coluna onde o erro ocorreu. No exemplo dado, o erro ocorreu na linha 1, coluna 4. Observe também que deve ser impresso o caractere que causou o erro léxico. Mesmo quando ocorrerem erros, o processo de reconhecimento dos *tokens* não para, continuando até que se atinja o fim do arquivo de entrada.

Observe o exemplo a seguir:

```
if(1)
{
/* isto eh um
comentario iniciado
e nao terminado
```

Saída esperada:

```
IF
L_PAREN
NUM_INTEGER(1)
R_PAREN
L_CURLY_BRACKET
error:lexical:3:1: unterminated comment
```

No exemplo apresentado, o arquivo termina com um comentário de bloco que não foi fechado. Tal tipo de erro deve ser apresentado com a mensagem padrão do exemplo. Observe também que mesmo o arquivo contendo 3 linhas de comentário de bloco não terminado, a mensagem de erro irá mostrar a linha onde o comentário se inicia. No caso do exemplo, o comentário se inicia na linha 3. Mesmo que o comentário possua inúmeras linhas, o erro deve apontar a linha onde o comentário é iniciado. Tal como outros erros léxicos, também deve-se informar a coluna onde o erro ocorreu. No exemplo apresentado, o erro ocorreu na coluna 1. A seguir é apresentado um exemplo maior de arquivo de entrada, bem como sua respectiva saída.

Entrada (Composta por 17 linhas):

```
if(@)
{
   a;
   printf("Adeus mundo \"cruel!\"")
   //Oi, eu sou um comentario de linha
   a->[666] += 0x34 + 07 << !2;
     "cadeia" != '@' + variavel_nao_declarada;
   /* oi, eu sou um
      comentario de bloco */
   for(;;){$} +45=-78,0X78+08;
   07;  // sou octal
   007; // sou inteiro
   0xa; // sou hexa
   0XA; // tambem sou hexa
}
comment++;/*Isto eh um
comentario sem fim...
```

Saída esperada (Composta por 61 linhas):

```
IF
L_PAREN
error:lexical:1:4: @
R_PAREN
L_CURLY_BRACKET
IDENTIFIER(a)
SEMICOLON
PRINTF
L_PAREN
```

```
STRING(Adeus mundo \"cruel!\")
R_PAREN
IDENTIFIER(a)
POINTER
L_SQUARE_BRACKET
NUM_INTEGER(666)
R_SQUARE_BRACKET
ADD_ASSIGN
NUM_HEXA(0x34)
PLUS
NUM_OCTAL(07)
L_SHIFT
NOT
NUM_INTEGER(2)
SEMICOLON
STRING(cadeia)
NOT_EQUAL
CHARACTER(@)
PLUS
IDENTIFIER(variavel_nao_declarada)
SEMICOLON
FOR
L_PAREN
SEMICOLON
SEMICOLON
R_PAREN
L_CURLY_BRACKET
error:lexical:10:12: $
R_CURLY_BRACKET
PLUS
NUM_INTEGER(45)
ASSIGN
MINUS
NUM_INTEGER(78)
COMMA
NUM_HEXA(0X78)
PLUS
NUM_INTEGER(08)
SEMICOLON
NUM_OCTAL(07)
SEMICOLON
NUM_INTEGER(007)
SEMICOLON
NUM_HEXA(0xa)
SEMICOLON
NUM_HEXA(0XA)
SEMICOLON
R_CURLY_BRACKET
```

```
IDENTIFIER(comment)
INC
SEMICOLON
error:lexical:16:11: unterminated comment
```

Como já dito, os identificadores possuem um tamanho máximo de 255 caracteres. Identificadores que possuirem mais que 255 caracteres, devem causar um erro léxico. Considere o exemplo a seguir:

Entrada (Composta por 6 linhas numeradas, onde a numeração NÃO faz parte do código):

```
1: int main()
2: {
3:     int identificador_muito_grande_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
4:
5:    return 0;
6: }
```

Saída esperada (Composta por 12 linhas):

```
INT
IDENTIFIER(main)
L_PAREN
R_PAREN
L_CURLY_BRACKET
INT
error:lexical:3:9: identifier too long
SEMICOLON
RETURN
NUM_INTEGER(0)
SEMICOLON
R_CURLY_BRACKET
```

No exemplo apresentado, existe um identificador declarado, cuja extensão é de 256 caracteres. Como o seu tamanho é maior que o limite determinado, a mensagem de erro `identifier too long` deve ser exibida, indicando a linha onde o identificador foi declarado e a coluna onde o mesmo se inicia. No exemplo apresentado, o identificador foi declarado na linha 3 e começa na coluna 9. Neste texto, o identificador do exemplo aparece em várias linhas de forma a caber na página, mas ele está totalmente escrito na linha 3;

Observações

- O caractere de tabulação não irá aparecer nos arquivos de entrada e não precisa ser tratado.
- Números hexadecimais são iniciados por `0x` ou `0X`.
- Números octais sempre são iniciados pelo número `0` seguido pelos números de `1` até `7` e depois pelos números de `0` até `7`. Um número iniciado por dois zeros, como `007`, deve ser tratado como inteiro.

- Os seguintes caracteres de barra invertida devem ser reconhecidos:

Código	Significado
\a	Alerta (beep)
\b	Retrocesso (BS)
\f	Alimentação de formulário (FF)
\n	Nova linha (LF)
\r	Retorno de carro (CR)
\t	Tabulação horizontal
\v	Tabulação vertical
\\	Barra invertida
\'	Aspas simples
\"	Aspas duplas
\?	Interrogação
\0	Null

Warnings

Além dos avisos de erro léxico, um outro tipo de mensagem que pode ser gerada são os **warnings**, que nada mais são do que avisos sobre alguma condição não usual que foi detectada no código, que embora não seja um erro, pode indicar alguma condição quer gere resultados imprevistos durante a execução do programa. Quando o compilador detectar alguma condição que gera um **warning**, ele deve emitir a mensagem apropriada e continuar com o processo de compilação.

Neste momento da implementação do compilador (análise léxica), somente uma condição irá gerar um **warning**. Considere o exemplo a seguir:

Entrada

```
int main()
{
    /* comentario
    /* de bloco
    */
    return 0;
}
```

Saída esperada:

```
INT
IDENTIFIER(main)
L_PAREN
R_PAREN
L_CURLY_BRACKET
warning:4:5: '/*' within block comment
RETURN
NUM_INTEGER(0)
SEMICOLON
R_CURLY_BRACKET
```

Comentários de bloco são iniciados pelos caracteres `/*`

Observe que no código apresentado, o padrão `/*` aparece novamente já dentro de um comentário de bloco. A cada vez que isso ocorrer, deverá ser emitida uma mensagem de aviso. No exemplo apresentado, foi gerada a seguinte mensagem: `warning:4:5:  '/*' within block comment`

A mensagem corresponde a um **warning** (aviso), que embora não seja um erro léxico, corresponde a uma condição não usual, no caso, um novo início de comentário de bloco dentro de um comentário de bloco. A mensagem de **warning** deve indicar a linha e a coluna onde o padrão `/*` foi detectado dentro do comentário de bloco. No exemplo apresentado, isso ocorreu na linha 4, coluna 5.

IMPORTANTE: Se ficou com alguma dúvida em relação a qualquer item deste texto, não hesite em falar com o professor da disciplina, pois ele está à disposição para sanar eventuais dúvidas, além do que, isso faz parte do trabalho dele.

Especificações de Entrega

O trabalho deve ser entregue no *AVA* em um arquivo com o nome `lexico.l`.

A entrega deve ser feita exclusivamente no *AVA* até a data/hora especificada. Não serão aceitas entregas atrasadas ou por outro meio que não seja o *AVA*.

Os arquivos entregues serão compilados da seguinte forma:

```
$flex ./lexico.l
$gcc lex.yy.c -o lexico
```

Desta forma, certifique-se que o seu código pode ser compilado/executado corretamente com os comandos apresentados.

O programa gerado deve ler as suas entradas da entrada padrão do sistema e imprimir as saídas na saída padrão do sistema. Um exemplo de execução para uma entrada chamada `teste.c` seria a seguinte:

```
$./lexico < teste.c
```

IMPORTANTE: Arquivos ou programas entregues fora do padrão receberão nota **ZERO**. Entende-se como arquivo fora do padrão aquele que tenha um nome diferente de `lexico.l`. Entende-se como programa fora do padrão aquele que apresentar erro de compilação, que não ler da entrada padrão, não imprimir na saída padrão, por exemplo. Uma forma de verificar se seu arquivo ou programa está dentro das especificações é testar o mesmo com o `script` de testes que é fornecido no *AVA*. Se o seu arquivo/programa **não** funcionar com o `script`, significa que ele está **fora** das especificações e, portanto, receberá nota **ZERO**.