Universidade Federal do Amazonas
Instituto de Computação
Compiladores
Prof. Edson Nascimento Silva Júnior

Trabalho 01 – Análise Léxica

Descrição geral

O objetivo deste trabalho consiste na implementação de um compilador funcional. Esta primeira etapa do trabalho consiste em fazer um analisador léxico e inicializar uma tabela global de símbolos encontrados para uma linguagem de programação de alto nível.

Análise Léxica

A sua análise léxica deve fazer as seguintes tarefas:

A. reconhecer as expressões regulares que descrevem cada tipo de lexema;
B. classificar os lexemas reconhecidos em tokens retornando as constantes definidas no arquivo tokens.h fornecido ou códigos ascii para caracteres simples;
C. incluir os identificadores e os literais (inteiros, reais, caracteres e strings) em uma tabela de símbolos global implementada com estrutura hash;
D. controlar o número de linha do arquivo fonte, e fornecer uma função declarada como int getLineNumber(void) a ser usada nos testes e pela futura análise sintática;
E. ignorar comentários de única linha e múltiplas linhas;
F. informar erro léxico ao encontrar caracteres inválidos na entrada, retornando o token de erro;
G. definir e atualizar uma variável global, e, uma função int isRunning(void), que mantém e retorna valor true (diferente de 0) durante a análise e muda para false (igual a 0) ao encontrar a marca de fim de arquivo;

Alfabeto

O Compilador deve ler um arquivo de entrada que contém símbolos válidos da linguagem. No caso do C--, estes símbolos são:

- Letras: **ab ... zAB ... Z**
- Dígitos: **0123456789**
- Símbolos Especiais: **, ; ( ) = < > + - * / % [ ] “ ‘ _ $ { } ? : ! . etc**
- Separadores**:** espaço, enter, tab

Tokens

Existem tokens que correspondem a caracteres particulares, como vírgula, ponto-e-vírgula, parênteses, para os quais é mais conveniente usar seu próprio código ascii, convertido para inteiro, como valor de retorno que os identifica. Para os tokens compostos, como palavras reservadas e identificadores, cria-se uma constante (#define em C ANSI) com um código maior do que 255 para representá-los.

Os tokens representam algumas categorias diferentes, como palavras reservadas, operadores de mais de um caractere e literais, e as constantes definidas no código do trabalho são precedidas por um prefixo para melhor identificar sua função, separando-as de outras constantes que serão usadas no compilador.

Palavras reservadas

As palavras reservadas da linguagem neste semestre são:

char, int, real, bool, if, then, else, while, input, output, return.

Para cada uma deve ser retornado o token correspondente.

Caracteres especiais

Os caracteres simples especiais empregados pela linguagem são listados abaixo (estão separados apenas por espaços), e devem ser retornados com o próprio código ascii convertido para inteiro.

Você pode fazer isso em uma única regra léxica. São eles:

, ; ( ) [ ] { } = + - * / % < > & | ~

Operadores Compostos

A linguagem possui, além dos operadores representados por alguns dos caracteres acima, operadores compostos, que necessitam mais de um caractere (somente dois) para serem representados no código fonte. São somente quatro operadores relacionais, conforme a tabela:

Representação original	Token retornado
<=	**OPERATOR_LE**
>=	**OPERATOR_GE**
==	**OPERATOR_EQ**
!=	**OPERATOR_DIF**

Identificadores

Os identificadores da linguagem são usados para designar variáveis, vetores e nomes de funções, são formados por uma sequência de um ou mais caracteres alfabéticos minúsculos ou maiúsculos e também os caracteres **'ponto'('.')** e **underline (´_´)**, e não podem conter dígitos em nenhuma posição;

Literais

- Literais são formas de descrever constantes no código fonte.
- Literais inteiros são formados por uma sequência de um ou mais dígitos decimais.
- Literais do tipo caractere são representados por um único caractere entre ***aspas simples*** (mais precisamente apóstrofo, ASCII decimal 39), como por exemplo: **'a', 'X', '-'**.
- Literais do tipo de dado real são definidos por dois literais decimais separados pelo caractere **'.'** (sem espaços, como, por exemplo, **"2.5"**).
- Literais do tipo string são quaisquer sequências de caracteres entre aspas duplas, como por exemplo **"meu nome"** ou **"Mensagem!"**, e servem apenas para imprimir mensagens com o comando **"output"**. Strings consecutivas não podem ser consideradas como apenas uma, o que significa que o caractere de aspas duplas não pode fazer parte de uma string. Para incluir os caracteres de aspas duplas e final de linha, devem ser usadas sequências de escape, como **"\""** e **"\n"**.

Comentários

Comentários de uma única linha começam em qualquer ponto com a sequência **"\\"** e terminam na próxima marca de final de linha, representada pelo caractere **'\n'**.

Comentários de múltiplas linhas iniciam pela sequência **"\\\"** e terminam pela sequência **"///"**, sendo que podem conter quaisquer caracteres, que serão todos ignorados, incluindo uma ou mais quebras de linha, as quais, entretanto, devem ser contabilizadas para controle do número de linha.

Caracteres em branco

Os caracteres de espaço, tabulação e nova linha são considerados como **"caracteres em branco"** e serão ignorados pelo analisador léxico da linguagem. Portanto, eles podem ocorrer entre quaisquer outros lexemas, e serão usados apenas para definir a disposição visual no editor e para separar diferentes lexemas entre si.

Controle e organização do seu código fonte

Você deve manter o arquivo tokens.h intacto, e separar a sua função main em um arquivo especial chamado main.c. Isso é necessário para facilitar a automação dos testes, que utilizará uma função main especial escrita pelo professor, substituindo a que você escreveu para teste e desenvolvimento. Você deve usar essa estrutura de organização, manter os nomes tokens.h.

Saída do programa

Gerar um arquivo no qual serão gravados cada token reconhecido e o seu respectivo lexema, e em caso de erro, a mensagem correspondente (escrever estas ações de forma clara).

Exemplos:

Entrada:	**Saída:**	
a = 2;	Tokens	Lexemas
	TK_IDENTIFIER OPERATOR_ATRIB KW_INT SG_SEMICOLON	a = 2 ;
‘a’ “palavra * a9 987.5	Tokens	Lexemas
	KW_CHAR TOKEN_ERROR	‘a’ “palavra

Principais funções que devem ser implementadas

- **analex():** função que implementa o diagrama de transição do analisador léxico.
- **prox_char():** esta função deve retornar apenas um caractere do arquivo de entrada por vez quando for chamada. Ela faz a interface entre o arquivo de entrada e o programa.
- **grava_token():** faz a gravação do token e do seu lexema no arquivo de saída

Estrutura básica para o programa principal

```
void main(){
     ch = prox_char();
     while(não fim) {
          (token, lexema) = analex();
          ...
          grava_token(token, lexema);
```

```
        }
}
```

Outras funções que precisam ser criadas

- **erro():** função que deve ser chamada para imprimir uma mensagem de erro e finalizar o analisador. Esta mensagem deve ficar gravada no arquivo de saída.

Equipes

O trabalho pode ser feito em grupos com no máximo 2 alunos.

O que deve ser entregue:

Além da entrega do código fonte, na plataforma do colabweb, o aluno deverá produzir:

1. **Manual do usuário** (uma página) Num arquivo chamado **mu.txt** ou **mu.doc**, contendo uma explicação de como se utilizar o analisador (explicar o formato da entrada e da saída do programa).
2. **um vídeo (ou mais)** explicando como as funções foram pensadas. As principais funções devem também ser explicadas. No vídeo, há a necessidade de apresentar o código fonte, a compilação e a execução do programa. Não há necessidade do aluno aparecer no vídeo.

Os vídeos devem ser postados em uma plataforma de vídeo (youtube, por exemplo), de modo que o professor possa acessar, e fazer parte da avaliação do trabalho. Para gravar pode usar serviços gratuitos e online. Por exemplo:

https://online-screen-recorder.com/pt

https://www.veed.io/pt-BR

https://streamyard.com/

ou se quiser, há aplicativos que podem ser baixados:

https://www.movavi.com/pt/learning-portal/gravadores-de-video.html

Atenção: Os vídeos não precisam ter alta produção. Para subir os vídeos para o Youtube, há uma necessidade de ajuste no seu perfil do youtube, com pelo menos 24h de antecedência. Portanto, sugiro que esse ajuste seja feito brevemente.

A nota será composta assim

- De 0 a 0,5 pontos pelo estilo de programação (nomes bem definidos, lugares de declarações, comentários).
- De 0 a 0,5 ponto pela compilação.
- De 0 a 0,5 ponto pelo formato de apresentação dos resultados
- De 0 a 7 pontos pela solução apresentada.
- De 0 a 1,5 pontos pelas informações dos vídeos.

Comentários Gerais

1. Comece a fazer este trabalho logo, enquanto o problema está fresco na memória e o prazo para terminá-lo está tão longe quanto jamais poderá estar.
2. Clareza, indentação e comentários no programa também vão valer pontos.
3. Trabalhos copiados serão penalizados conforme anunciado

Verifique regularmente os documentos e mensagens da disciplina para informar-se de alguma eventual atualização que se faça necessária ou dicas sobre estratégias que o ajudem a resolver problemas particulares. Em caso de dúvida, consulte o professor.

Dicas

- os tokens podem ser criados como constantes (#define)
- Indicar os erros possíveis (caracteres inválidos, literais inconsistentes, ...)
- Parar a execução após encontrar o primeiro erro

Gramática

Declarações

1. programa → lista-decl lista-com
2. lista-decl → lista-decl decl | decl
3. decl → decl-var | decl-func
4. decl-var → espec-tipo var ;
5. espec-tipo → INT | VOID | FLOAT
6. decl-func → espec-tipo ID (params) com-comp
7. params → lista-param | void | ε
8. lista-param → lista-param , param | param
9. param → espec-tipo var
10. decl-locais → decl-locais decl-var | ε

Comandos

11. lista-com → comando lista-com | ε
12. comando → com-expr | com-atrib | com-comp | com-selecao | com-repeticao | com-retorno
13. com-expr → exp ; | ;
14. com-atrib → var = exp ;
15. com-comp → { decl-locais lista-com }
16. com-selecao → IF (exp) comando | IF (exp) com-comp ELSE comando
17. com-repeticao → WHILE (exp) comando ;
18. com-retorno → RETURN ; | RETURN exp ;

Expressões

19. exp → exp-soma op-relac exp-soma | exp-soma
20. op-relac → <= | < | > | >= | == | !=
21. exp-soma → exp-soma op-soma exp-mult | exp-mult
22. op-soma → + | -
23. exp-mult → exp-mult op-mult exp-simples | exp-simples

24. op-mult → * | / | %
25. exp-simples → (exp) | var | cham-func | literais
26. literais → NUM | NUM.NUM
27. cham-func → ID (args)
28. var → ID | ID [NUM]
29. args → lista-arg | ε
30. lista-arg → lista-arg , exp | exp