UNIVERSIDADE FEDERAL DE LAVRAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Curso: *Ciência da Computação*	*GCC130 – Compiladores*	Professor: *Rafael S. Durelli*
Pontuação: *40 pontos* (3 *questões)*	**TP**	Data: *Veja etapa*

INFORMAÇÕES SOBRE TP:

1. Atividades entregues **após o prazo** terão penalização na nota. Logo, fiquem atentos à data de entrega.
2. Cópias (total ou parcial) serão penalizadas com **nota zero** em todos os trabalhos.
3. A atividade é grupo de, *no máximo*, 3 (três) alunos.
4. O trabalho deve ser entregue pelo *Campus* e será avaliado junto com os alunos no laboratório em data estipulada.
5. Envie arquivos *somente* nos formatos *txt* e *pdf* (não enviar *doc*, *docx*, *odt*, etc.). Arquivos compactados somente *zip* e *tar.gz* (não enviar *rar*, *z*, etc.). Não use acentos e "ç" nos nomes de arquivo.

O objetivo do trabalho prático é desenvolver um analisador léxico, sintático e semântico para a linguagem Java-, conforme especificação em anexo. O trabalho está dividido em 3 etapas, conforme descrição a seguir.

1. **Etapa 1 – Análise Léxica** (Data Limite de Entrega: Ver Sig, 5 pontos)

   O objetivo desta etapa é implementar um analisador léxico para a linguagem C- (a documentação oficial do C- está no Campus)-.

   O analisador léxico deverá ser implementado usando uma linguagem especificada e PRÉ-DEFINIDA pelo grupo. Ele deverá retornar, a cada chamada, o token reconhecido.

   Além de reconhecer os tokens da linguagem, o analisador léxico deverá detectar possíveis erros e reportá-los ao usuário. O programa deverá informar o erro e seu local (linha e coluna).

   Lembre-se que espaços em branco (espaços, tabulações, quebras de linha, etc.) e comentários não são tokens. Portanto, devem ser descartados.

   Faça também um programa para testar o analisador léxico. Este programa deve imprimir a linha, a coluna, o lexema e o tipo do token que foi identificado.

   Lembre-se também de criar e mostrar a Tabela de Simbolos similar a tabela apresentada em aula.

   Você deverá entregar nesta etapa:

   - O autômato para reconhecimento dos tokens;
   - O programa com todos os arquivos-fonte;
   - Os arquivos de teste. Inclua testes com um programa correto contendo todos os tipos de tokens e um outro com um programa contendo pelo menos 5 erros distintos;
   - Um relatório (em forma de artigo) contendo introdução, referencial teórico, descrição do trabalho com suas estratégias de solução, testes executados e resultados obtidos, conclusão e referências bibliográficas.

2. **Etapa 2 – Análise Sintática** (Data Limite de Entrega: Ver Sig, 15 pontos)

   O objetivo desta etapa é implementar um analisador sintático para a linguagem C- -.

   O analisador sintático deverá ser implementado usando uma linguagem especificada e PRÉ-DEFINIDA pelo grupo. Ele deverá varrer todo o código-fonte fornecido como entrada e relatar possíveis erros de sintaxe. Ao encontrar um erro, o programa deverá exibir uma mensagem com informações sobre ele e o local onde ocorreu (linha e coluna do código-fonte). Após a exibição do erro, o programa deve continuar a análise sintática para o restante do código-fonte.

   Você deverá adaptar a gramática da linguagem escolhida pelo grupo para que ela possa ser implementada utilizando o conceito *recursive descent parser*.

   Seu programa deverá obter os tokens usando o analisador léxico desenvolvido na Etapa 1 do trabalho. Se for necessário, faça as devidas modificações no seu analisador léxico.

   Você deverá entregar nesta etapa:

   - O programa com todos os arquivos-fonte;
   - Os arquivos de teste. Inclua testes com um programa correto contendo todos os comandos da linguagem e um outro com um programa contendo pelo menos 10 erros distintos;

- Um relatório (em forma de artigo) contendo introdução, referencial teórico, descrição do trabalho com suas estratégias de solução, testes executados e resultados obtidos, conclusão e referências bibliográficas. Incremente o relatório entregue na Etapa 1.

Lembre-se também de atualizar (adicionar novos campos) agora na Tabela de Simbolos.

3. **Etapa 3 (Final) – Tabela de Símbolos, Análise Semântica e Geração de Código Intermediário** (Data Limite de Entrega: Ver Sig, 20 pontos)

   O objetivo desta etapa é implementar/atualizar a tabela de símbolos, o analisador semântico e o gerador de código intermediário para a linguagem C-.

   Seu programa deve:

   - Criar uma estrutura de dados, usando tabela hash, para implementar a tabela de símbolos, bem como operações para inserir e obter símbolos dessa tabela. A tabela deve armazenar todos os identificadores do programa. Para cada variável, armazene também seu tipo de dado e o endereço relativo de onde ela será armazenada. Gere um erro se um identificador não foi declarado ou está fora de escopo.
   - Gerar código de três endereços. Além disso deve-se também criar estruturas de dados, tais como quádruplas ou triplas, para o código de três endereços.

   Não é necessário fazer a análise semântica e a geração de código de três endereços para funções/procedimentos nesta etapa. No entanto, a análise sintática e a criação da tabela de símbolos deve ser efetuada.

   Você deverá entregar nesta etapa:

   - O programa com todos os arquivos-fonte;
   - Os arquivos de teste. Inclua testes com um programa correto contendo todos os comandos da linguagem e um outro com um programa contendo pelo menos 10 erros distintos (ex.: variável não declarada ou fora de escopo, expressão com tipos incompatíveis);
   - Um relatório (em forma de artigo) contendo introdução, referencial teórico, descrição do trabalho com suas estratégias de solução, testes executados e resultados obtidos, conclusão e referências bibliográficas. Incremente os relatórios entregues nas Etapas 1 e 2.

Observações Importantes:

- O programa deverá ser modularizado; não use variáveis globais nos módulos; utilize a passagem de parâmetros;
- Toda e qualquer mensagem de orientação e de erro deve ser adequadamente tratada;
- Use comentários para documentar o seu código.

ANEXO: DESCRIÇÃO DA LINGUAGEM C-

- Tipos de dados: inteiro, real, caractere, arranjo e registro.
- Funções: recursão, parâmetros passados por valor.
- Comandos: Atribuição, if/else, while, E/S simples (tratados como funções).
- Comentários: texto entre /* e */ (sem comentários aninhados).
- Palavras reservadas: int, float, struct, if, else, while, void, return (caixa baixa)
- Símbolo Inicial: <programa>

1. <programa> ::= <declaração-lista>
2. <declaração-lista> ::= <declaração> {<declaração>}
3. <declaração> ::= <var-declaração> | <fun-declaração>
4. <var-declaração> ::= <tipo-especificador> <ident> **;** | <tipo-especificador> <ident> <abre-colchete> <num-int> <fecha-colchete> {<abre-colchete> <num-int> <fecha-colchete>} **;**
5. <tipo-especificador> ::= **int** | **float** | **char** | **void** | **struct** <ident> <abre-chave> <atributos-declaração> <fecha-chave>
6. <atributos-declaração> ::= <var-declaração> {<var-declaração>}
7. <fun-declaração> ::= <tipo-especificador> <ident> **(** <params> **)** <composto-decl>
8. <params> ::= <param-lista> | **void**
9. <param-lista> ::= <param> {, <param>}
10. <param> ::= <tipo-especificador> <ident> | <tipo-especificador> <ident> <abre-colchete> <fecha-colchete>
11. <composto-decl> ::= <abre-chave> <local-declarações> <comando-lista> <fecha-chave>
12. <local-declarações> ::= {<var-declaração>}
13. <comando-lista> ::= { <comando> }
14. <comando> ::= <expressão-decl> | <composto-decl> | <seleção-decl> | <iteração-decl> | <retorno-decl>
15. <expressão-decl> ::= <expressão> **;** | **;**
16. <seleção-decl> ::= **if (** <expressão> **)** <comando> |
17. **if (** <expressão> **)** <comando> **else** <comando>
18. <iteração-decl> ::= **while (** <expressão> **)** <comando>
19. <retorno-decl> ::= **return ;** | **return** <expressão> **;**
20. <expressão> ::= <var> = <expressão> | <expressão-simples>
21. <var> ::= <ident> | <ident> <abre-colchete> <expressão> <fecha-colchete> {<abre-colchete> <expressão> <fecha-colchete>}
22. <expressão-simples> ::= <expressão-soma> <relacional> <expressão-soma> |
23. <expressão-soma>
24. <relacional> ::= **<=** | **<** | **>** | **>=** | **==** | **!=**
25. <expressão-soma> ::= <termo> {<soma> <termo>}
26. <soma> ::= **+** | **-**
27. <termo> ::= <fator> {<mult> <fator>}
28. <mult> ::= ***** | **/**
29. <fator> ::= **(** <expressão> **)** | <var> | <ativação> | <num> | <num-int>
30. <ativação> ::= <ident> **(** <args> **)**
31. <args> ::= [<arg-lista>]
32. <arg-lista> ::= <expressão> {**,** <expressão>}
33. <num> ::= [**+** | **-**] <dígito> {<dígito>} [**.** <dígito> {<dígito>}] [**E** [**+** | **-**] <dígito> {<dígito>}]
34. <num-int> ::= <dígito> {<dígito>}
35. <digito> ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
36. <ident> ::= <letra> {<letra> | <dígito>}
37. <letra> ::= **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z**
38. <abre-chave> ::= **{**
39. <fecha-chave> ::= **}**
40. <abre-colchete> ::= [
41. <fecha-colchete> ::=]

Forma Normal de Backus (BNF)

BNF é uma notação alternativa para representar as regras de produção de uma gramática, onde:

1. $\rightarrow$ é substituído por ::=
   Exemplo: $w \rightarrow s$ se escreve como $w ::= s$
2. os símbolos não terminais (variáveis) estão entre $< >$
3. os símbolos terminais estão em negrito.

A notação BNF é usada para definir gramáticas com as características de que o lado esquerdo de cada regra é composta por um único símbolo não terminal. EBNF é a notação BNF estendida, onde:

1. símbolos entre [e] são opcionais;
2. símbolos entre { e } são repetições (zero ou mais).