RELATÓRIO

ESTRUTURA DO PROJETO

root
- **PasC**
  - **Documentation** # Documentação do projeto
    - **Automata** # Desenho do autômato e projeto do JFLAP
    - **Grammar** # Tabela Preditiva e gramática corrigida
    - **Reports** # Relatórios
    - **Tests** # Testes realizados
  - **Models** # Modelos e estruturas de dados
  - **Modules** # Módulos do compilador
  - **Properties** # Informações da solução
  - **Program.cs** # Classe principal
  - **pasc_test.pc** # Arquivo de teste geral

1. Models

O *namespace* Models guarda todos os modelos e estruturas de dados do compilador, aqui é onde ficam as classes para controle da tabela de símbolos, tokens, tags e identificadores.

1.1 Grammar.cs

Regex **DIGIT:** Expressão regular para verificação de dígitos.
Regex **LETTER:** Expressão regular para verificação de letras.
Regex **NUMCONST:** Expressão regular para verificar decimais e não decimais.
Regex **CHARCONST:** Expressão regular para verificar char entre aspas simples.
Regex **LITERAL:** Expressão regular para verificar string entre aspas duplas.
Regex **ID:** Expressão regular para verificar um identificador.

Dictionary **SYMBOL_TABLE<Token, Identifier>:** Estrutura hash que guarda a tabela de símbolos. O Token é a *key* e o Identifier é o *value*, todos os símbolos que serem adicionados terão que possuir estes dois parâmetros.

Método **Show()**

Imprime a SYMBOL_TABLE no console.

Método **Add(Token key, Identifier value)**

Adiciona um novo símbolo na SYMBOL_TABLE de acordo com os parâmetros *key* e *value*.

Método **Identifier GetID(Token key)**

Retorna a ID do identificador na SYMBOL_TABLE de acordo com a *key* passada por um token.

Método **Token GetToken(String lexeme)**

Retorna um token da SYMBOL_TABLE de acordo com um lexema e se o mesmo já está contido na tabela.

1.2 Identifier.cs

int **Tag:** Recebe o valor do identificador.
String **Tag:** Recebe o tipo do identificador.

1.3 Tag.cs

enum **Tag:** Identificadores enumerados para serem usados em conjunto com a tabela de símbolos.

1.4 Token.cs

int **Row:** Propriedade da linha.
Tag **Type:** Propriedade do tipo de acordo com o **enum Tag**.
int **Column:** Propriedade da coluna.
String **Lexeme:** Propriedade do lexema.

Método **Token(Tag Type, String Lexeme, int Row, int Column)**

Construtor do token.

Método Override **ToString()**

Sobrescreve o ToString() padrão para retornar a saída do token formatada.

2. Modules

O *namespace* Modules guarda todos os módulos do compilador, como o Analisador Léxico, Sintático e Semântico.

2.1 Global.cs

char **NEW_LINE:** Caractere de nova linha.
FileStream **SOURCE:** Arquivo fonte lido pelo compilador.

Método **SetEnvironment(string sourceFile)**

Abre e transforma o arquivo *sourceFile* passado como parâmetro e verifica o tipo de caractere de nova linha de acordo com o sistema operacional, adiciona uma nova linha em branco no arquivo fonte caso não haja por segurança para a leitura sempre correr de forma correta e abre o arquivo fonte na memória.

Método **EndParsing()**

Limpa o arquivo fonte da memória, exibe a tabela de símbolos e finaliza o programa.

2.1 Lexer.cs

char **CURRENT_CHAR:** Caractere atual que está sendo passado para o autômato.
StringBuilder **LEXEME:** Lexema atual que está sendo moldado pelo autômato.
int **ROW:** Linha atual.
int **COLUMN:** Coluna atual.
int **LAST_CHAR:** Caractere antecessor do CURRENT_CHAR.
readonly int **EOF:** Constante para verificação de fim de arquivo.
int **STATE:** Estado atual do autômato.
bool **QUOTES_ERROR:** Verificador de erros do fecha aspas.

Método **Restart()**

Volta um caractere e reinicia o autômato a partir do estado inicial.

Método Token **AddToken(Tag tag)**

Adiciona um novo token na tabela de símbolos e o retorna para o método que é chamado.

Método bool **IsASCII(char c)**

Verifica se um char está contido na tabela ASCII.

Método bool **IsNewLine()**

Verifica o se o CURRENT_CHAR é um char especial de quebra de linha.

Método **LexicalError(string message)**

Escreve no console uma mensagem de erro léxico formatada a partir do parâmetro *message*.

Método **MultilineCommentErrorCheck()**

Verifica se o comentário de várias linhas não foi fechado ao chegar ao EOF, essa verificação é chamada no método NextToken().

Método Token **NextToken()**

Avança um caractere do arquivo, conta linhas e colunas e verifica fim de arquivo. Controla as condições de estados do autômato a partir do CURRENT_CHAR lido, monta o lexema atual e retorna tokens encontrados em estados finais.

Método **SetState(int currentState, bool appendLexeme)**

Seta o estado atual e adiciona o CURRENT_CHAR ao LEXEME se solicitado pelo parâmetro *appendLexeme*.

Método string **GetLexeme()**

Retorna o lexema atual.

2.1 Parser.cs

int **ERROR_COUNT:** Contador de erros para auxiliar no modo pânico.
Token **Token:** Token atual que está sendo lido pelo Parser.

Método **Set()**

Inicia o Parser e ao final finaliza o programa chamando o método Global.EndParsing().

Método **SyntacticError(String lexeme)**

Exibe o erro sintático atual de acordo com o lexema e faz uma contagem. Se chegar a 5 mostra uma mensagem de muitos erros sintáticos encontrados e finaliza o programa chamando o método Global.EndParsing().

Método Tag **GetTag()**

Retorna a Tag do token atual.

Método **Advance()**

Pede o próximo token para o método NextToken() assim avançando a entrada.

Método **Eat()**

Verifica se a entrada é o que o Analisador Sintático está esperando e avança a entrada.

O resto dos métodos da classe fazem parte da construção do Parser e são ligados diretamente à gramática.

3. Program.cs

Classe principal do projeto, onde é realizado o controle do parâmetro do arquivo fonte passado pelo usuário, para iniciar o processo de compilação.