Relatório Parcial

Analisador Léxico

Renan Martins Zomignani Mendes
Tiago Schelp Lopes

1. **Quais são as funções do analisador léxico nos compiladores/ interpretadores?**

Extração e classificação de átomos: associa o texto-fonte à outro texto formado pelos átomos que os símbolos componentes do texto-fonte representam. O analisador léxico também classifica esses átomos em *identificadores, palavras reservadas, números inteiros sem sinal, números reais, cadeias de caracteres, sinais de pontuação e de operação, caracteres especiais, símbolos compostos de dois ou mais caracteres, comentários, etc.*

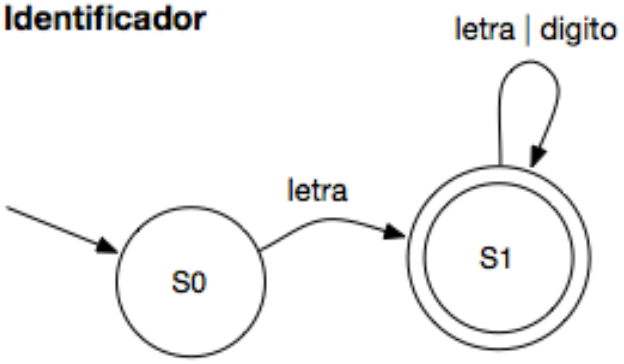
2. **Quais as vantagens e desvantagens da implementação do analisador léxico como uma fase separada do processamento da linguagem de programação em relação à sua implementação como sub-rotina que vai extraindo um átomo a cada chamada?**

Implementação	Vantagens	Desvantagens
Fase separada do processamento	Simplicidade do compilador final: funções nitidamente separadas entre as diversas etapas	A comunicação entre o analisador léxico deverá ser feita por meio de arquivos intermediários ou ainda com todos os tokens carregados em memória
Sub-rotina	Carrega em memória somente a quantidade necessária de tokens à análise sintática.	
	Lê cada token somente uma vez	

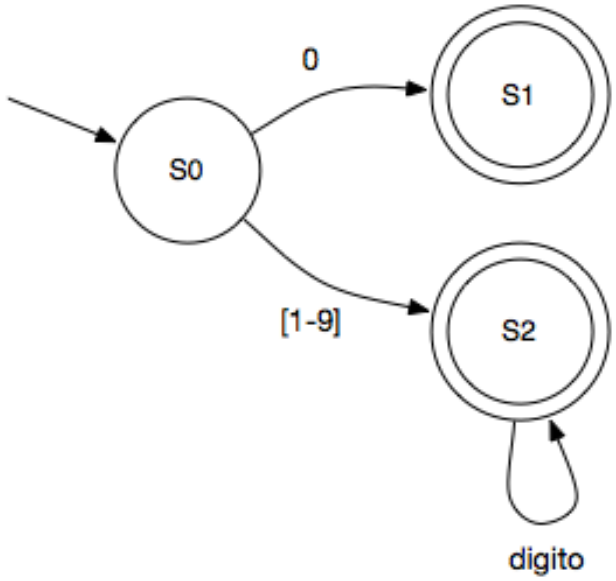
3. **Defina formalmente, através de expressões regulares sobre o conjunto de caracteres ASCII, a sintaxe de cada um dos tipos de átomos a serem extraídos do texto-fonte pelo analisador léxico, bem como de cada um dos espaçadores e comentários.**

token	Expressão Regular
ID	[a-zA-Z]([a-zA-Z]|[0-9])*
PALAVRA_RESERVADA	(if|else|end|while|int|float|bool|char|true|false)
NUM_INT	(0|[1-9][0-9]*)
NUM_REAL	NUM_INT*\.NUM_INT+
STRING	“({\”}^{c})*”
PONTUACAO	\n
CARACTERE_ESPECIAL	(\+|\-|*|/|%|=|\(|\)|>|<|>=|<=|!=|==|&|||!)
ESPACADOR	(|\t)
COMENTARIO	#{\n}^{c}\n

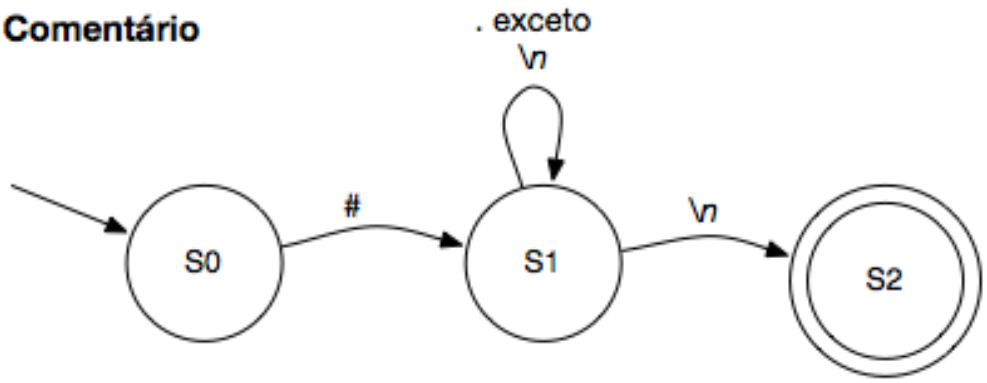
4. Converta cada uma das expressões regulares, assim obtidas, em autômatos finitos equivalentes que reconheçam as correspondentes linguagens por elas definidas.



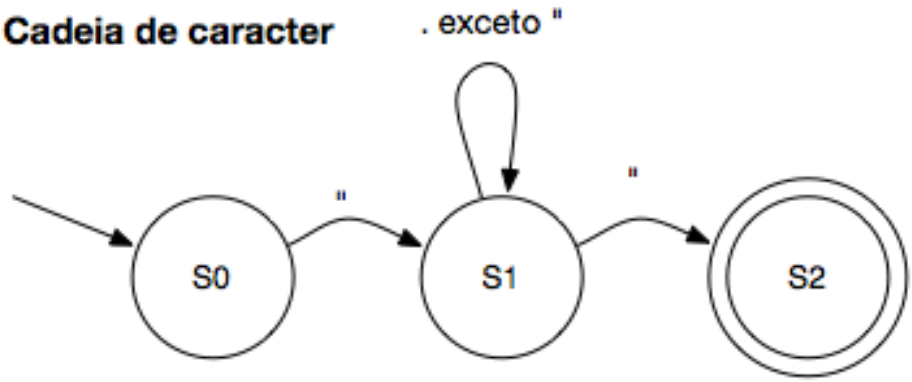
Número Inteiro



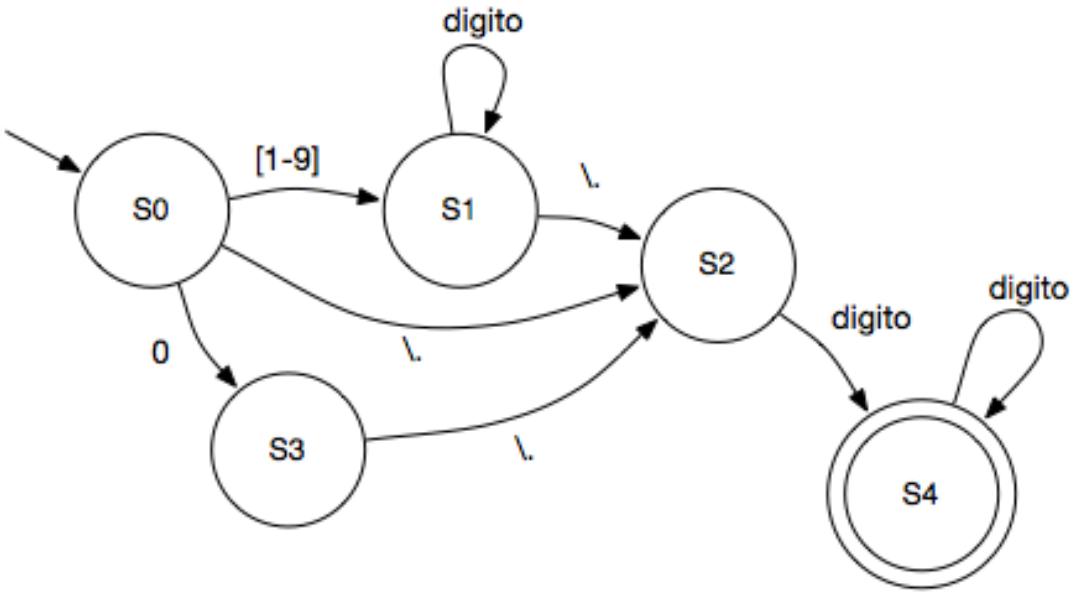
Comentário



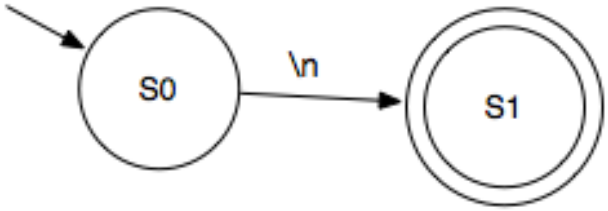
Cadeia de caracter



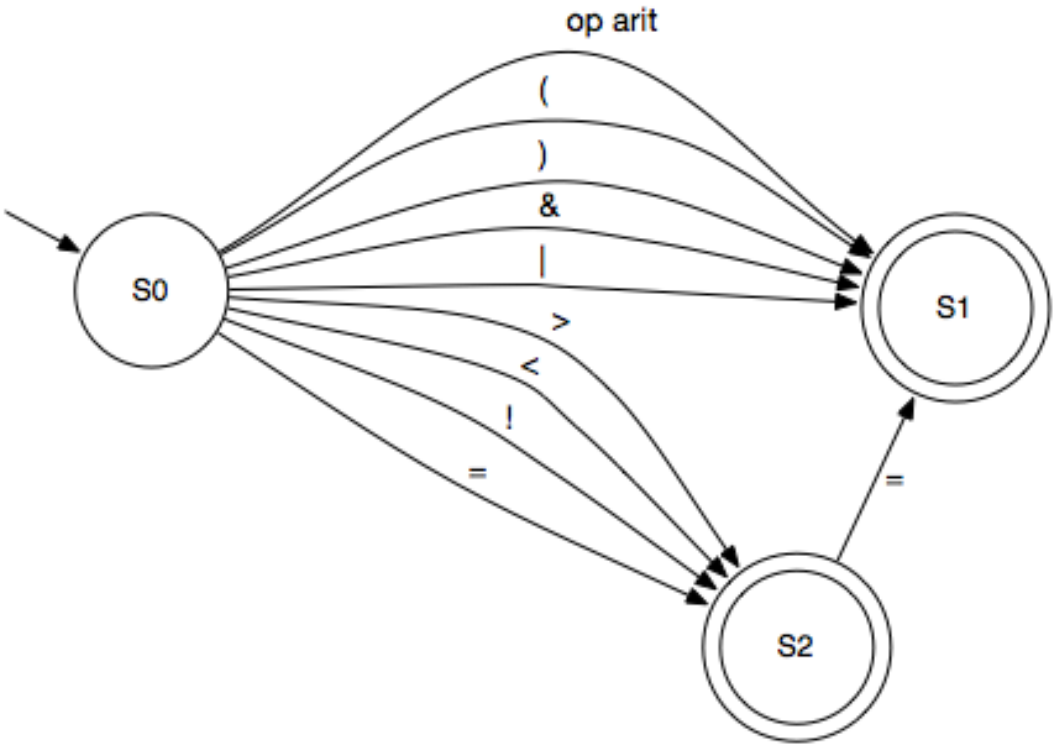
Número Real



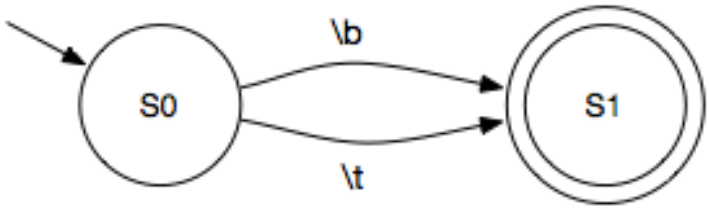
Pontuação



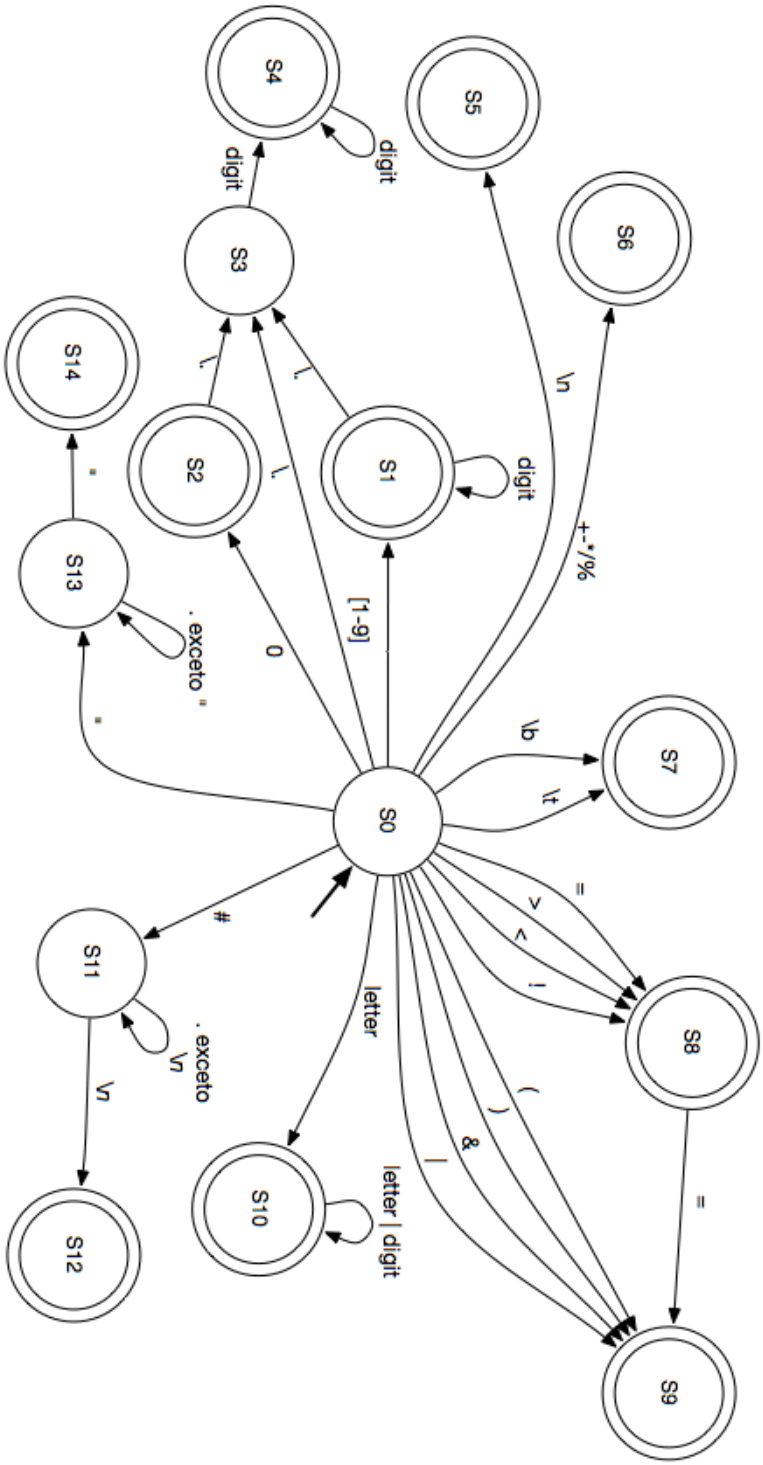
Caracteres Especiais



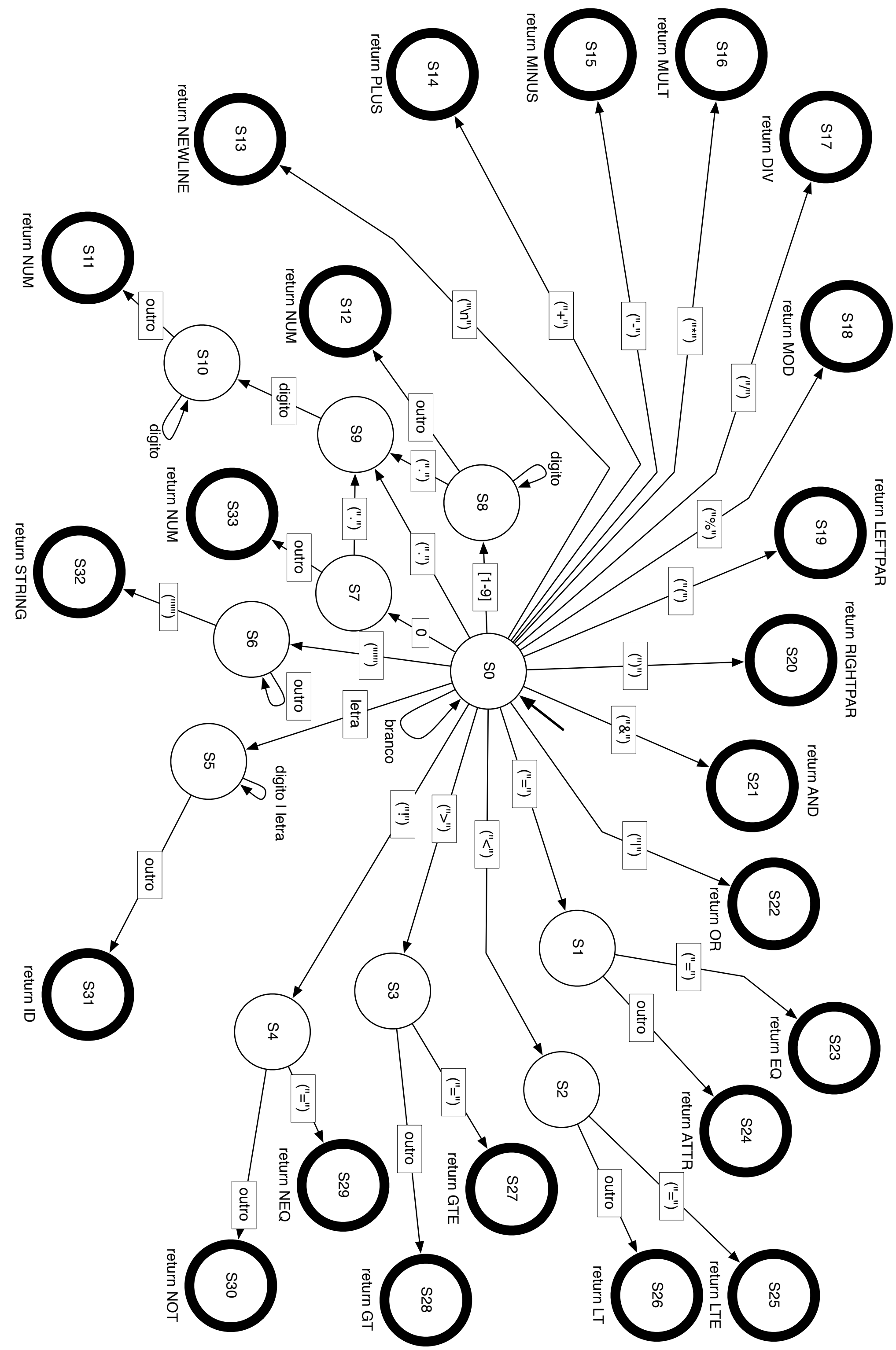
Espaçador



5. **Crie um autômato único que aceite todas essas linguagens a partir de um mesmo estado inicial, mas que apresente um estado final diferenciado para cada uma delas.**



6. **Transforme o autômato assim obtido em um transdutor, que emita como saída o átomo encontrado ao abandonar cada um dos estados finais para iniciar o reconhecimento de mais um átomo do texto.**



7. **Converta o transdutor assim obtido em uma sub-rotina, escrita na linguagem de programação de sua preferência. Não se esqueça que o final de cada átomo é determinado ao ser encontrado o primeiro símbolo do átomo ou do espaçador seguinte. Esse símbolo não pode ser perdido, devendo-se, portanto, tomar os cuidados de programação que forem necessários para reprocessá-los, apesar de já terem sido lidos pelo autômato.**

8. **Crie um programa principal que chame repetidamente a sub-rotina assim construída, e a aplique sobre um arquivo do tipo texto contendo o texto-fonte a ser analisado. Após cada chamada, esse programa principal deve imprimir as duas componentes do átomo extraído (o tipo e o valor do átomo encontrado). Faça o programa parar quando o programa principal receber do analisador léxico um átomo especial indicativo da ausência de novos átomos no texto de entrada.**

9. **Relate detalhadamente o funcionamento do analisador léxico assim construído, incluindo no relatório: descrição teórica do programa; descrição da sua estrutura; descrição de seu funcionamento; descrição dos testes realizados e das saídas obtidas.**

Estrutura do token:
O token possui um valor (lexema), um tipo (de acordo com o seu conteúdo, definido no transdutor), uma linha e uma coluna (correspondente à sua posição no arquivo de entrada).

```
typedef struct token {
    tipoToken tipo;
    char *valor;
    int linha;
    int coluna;
} Token;
```

Os tipo que podem ser atribuídos aos tokens estão definidos no tipo tiposToken, detalhado abaixo.

```
typedef enum tiposToken { DIV, MULT, MINUS, PLUS, NEWLINE,
NUM, STRING, ID, NOT, NEQ, GT,  GTE, LT, LTE, ATTR, EQ,
OR, AND,  RIGHTPAR, LEFTPAR, MOD, EoF, ERRO} tipoToken;
```

Identificando os tokens:

```
Token* getNextToken(FILE* file, char* ch,
                    int* linha, int* coluna);
```

Essa função retorna o próximo token do arquivo indicado em FILE. Ela implementa a máquina de estados explicitada no transdutor da questão 6, com algumas generalizações para simplificar a quantidade de estados.

Os estados finais foram condensados em um único estado final chamado FIM_Token, esse estado chama a função `Token* criarToken(char *Lexema, tipoToken tipo, int linha, int coluna)`, que por sua vez cria um novo token, atribui os valores passados como parâmetros da função para os atributos correspondentes do novo Token criado e então retorna esse Token.

A máquina de estados implementada trata o tipo de token antes de ir para o estado final, dessa maneira, sem perda de funcionalidade foi possível condensar 24 estados finais em 1, proporcionando uma mais fácil leitura e implementação do código.

Nos casos em que os lexemas podem conter mais de um caractere foi preciso atenção especial com a leitura dos próximos caracteres. Por exemplo, só percebemos que um token do tipo número chega ao fim depois de ler o próximo caractere do arquivo, podendo ser não somente um espaçador como também um caractere especial. Não podemos então descartar o novo caractere lido, já que corresponde a um elemento do próximo token. Esse tratamento é feito em determinados estados dependendo da necessidade.

Outro ponto que é importante lembrar é o fato que, ao invés de indicarmos o caractere de quebra de linha (\n) no final da linha, preferimos indicá-lo na posição -1 da linha seguinte. Isso se fez visando uma mais fácil e limpa implementação do analisador léxico. De fato, não pudemos constatar empecilho algum decorrente dessa escolha.

DECORATIVOS:
Os decorativos são descartados no início da função getNextToken, até o momento são tratados como decorativos os comentários (#), espaços em branco (“ ”) e tabulações (\t).

ERROS:
Os caracteres não identificados no estado S0 são armazenados em um Token com o tipoToken `ERRO`, este erro pode ser localizado através da indicação de linha e coluna.

10. Explique como enriquecer esse analisador léxico com um expansor de macros do tipo #DEFINE, não paramétrico nem recursivo, mas que permita a qualquer macro chamar outras macros, de forma não cíclica. (O expansor de macros não precisa ser implementado).

Para implementar um expansor de macros, deve-se primeiro poder reconhecer as macros a partir dos comandos #DEFINE. A cada vez que um símbolo não previsto na gramática for lido, deve-se verificar se este constitui uma macro. Se for esse o caso, deve-se substitui-la por seu equivalente na gramática original. Para isso, implementa-se uma recursão que realiza os passos descritos anteriormente até que um conjunto de tokens que não sejam novas macros sejam reconhecidos.