

## Trabalho Prático - Especificação da Etapa 1: Análise Léxica e Inicialização de Tabela de Símbolos

### Resumo:

O trabalho consiste na implementação de um compilador funcional. Esta primeira etapa do trabalho consiste em fazer um analisador léxico utilizando a ferramenta de geração de reconhecedores *lex* (ou *flex*) e inicializar uma tabela de símbolos encontrados.

### Funcionalidades necessárias:

A sua análise léxica deve fazer as seguintes tarefas:

- reconhecer as expressões regulares que descrevem cada tipo de lexema;
- classificar os lexemas reconhecidos em *tokens* retornando as constantes definidas no arquivo `tokens.h` fornecido ou codigos *ascii* para caracteres simples;
- incluir os identificadores e os literais (inteiros, caracteres e *strings*) em uma tabela de símbolos implementada com estrutura *hash*;
- 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;
- ignorar comentários no formato C99 de única linha e múltiplas linhas;
- informar erro léxico ao encontrar caracteres inválidos na entrada, retornando o *token* de erro;
- 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;

### Descrição dos 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 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 são: `word`, `bool`, `byte`, `if`, `then`, `else`, `loop`, `input`, `output` e `return`. Para cada uma deve ser retornado o *token* correspondente.

## Caracteres especiais

Os caracteres simples especiais empregados pela linguagem são listados abaixo 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 que retorna `yytext[0]`. São eles:

, ; : ( ) [ ] { } + - \* / < > = ! & \$

## Operadores Compostos

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

Source Representation	Returned Token
<code>&lt;=</code>	<code>OPERATOR_LE</code>
<code>&gt;=</code>	<code>OPERATOR_GE</code>
<code>==</code>	<code>OPERATOR_EQ</code>
<code>!=</code>	<code>OPERATOR_NE</code>
<code>&amp;&amp;</code>	<code>OPERATOR_AND</code>
<code>  </code>	<code>OPERATOR_OR</code>

## Identificadores

Os identificadores da linguagem são usados para designar variáveis, vetores e nomes de funções, e são formados por uma sequência de caracteres alfabéticos ou dígitos, sendo obrigatoriamente iniciados por um caractere alfabético, onde considera-se caractere alfabético como letras maiúsculas ou minúsculas ou o caractere *underline* (`'_'`), e onde são dígitos: `'0'-'9'`.

## Literais

Literais são formas de descrever constantes no código fonte. Literais inteiros são representados como repetições de um ou mais dígitos decimais. Literais do tipo caractere são representados por um único caractere entre aspas simples, como por exemplo: `'a'`, `'x'`, `'-'`. Literais do tipo *string* são qualquer sequência de caracteres entre aspas duplas, como por exemplo `"meu nome"` ou `"Mensagem!"`, e servem apenas para imprimir mensagens com o comando `'output'`. Os literais do tipo caractere e *string* podem ser inseridos na tabela de símbolos já sem as aspas que os identificam no código fonte. Existem ainda dois literais especiais, designados pelas palavras `TRUE` e `FALSE`, escritas em maiúsculo, que são valores válidos para inicialização de variáveis ou expressões booleanas.

## 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`, já que a função `main` não pode estar contida no código de `scanner.l`. 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` e `scanner.l`. Instruções mais detalhadas sobre o formato de submissão do trabalho e cuidados com detalhes específicos serão disponibilizadas em outro documento separado, em seguida.

## Atualizações e Dicas

Verifique regularmente a página da disciplina e o final desse documento 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.

Porto Alegre, 10 de Março de 2015

## Anexo - Código tokens.h

```
/*
 * Lista dos tokens, com valores constantes associados.
 * Este arquivo serah posteriormente substituido, nao acrescente nada.
 * Os valores das constantes sao arbitrarios, mas nao podem ser alterados.
 * Cada valor deve ser distinto e fora da escala ascii.
 * Assim, nao conflitam entre si e com os tokens representados pelo proprio
 * valor ascii de caracteres isolados.
 */

#define KW_WORD          256
#define KW_BOOL          258
#define KW_BYTE          259
#define KW_IF            261
#define KW_THEN          262
#define KW_ELSE          263
#define KW_LOOP          264
#define KW_INPUT         266
#define KW_RETURN        267
#define KW_OUTPUT        268

#define OPERATOR_LE      270
#define OPERATOR_GE      271
#define OPERATOR_EQ      272
#define OPERATOR_NE      273
#define OPERATOR_AND     274
#define OPERATOR_OR      275

#define TK_IDENTIFIER    280
#define LIT_INTEGER      281
#define LIT_FALSE        283
#define LIT_TRUE         284
#define LIT_CHAR          285
#define LIT_STRING        286

#define TOKEN_ERROR      290

/* END OF FILE */
```