UFRGS – Informática - INF01147 - Compiladores - 2017/2
Turma A - Prof. Marcelo Johann

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 necesárias:
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 codigos *ascii* para caracteres simples;
c. incluir os identificadores e os literais (inteiros, caracteres e *strings*) em uma tabela de símbolos 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 no formato C99 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;

Descrição dos tokens
Existem tokens que correspondem a caracteres particulares, como vírgula, ponto-e-vírgula, parênteses, para os quais é mais conveninente 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: `byte`, `short`, `long`, `float`, `double`, `if`, `then`, `else`, `while`, `for`, `read`, `print` e `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 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
<=	OPERATOR_LE
>=	OPERATOR_GE
==	OPERATOR_EQ
!=	OPERATOR_NE
&&	OPERATOR_AND
||	OPERATOR_OR

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, iniciando por um caractere alfabético, incluindo *underline* (´_´), e permitindo dígitos (´0´-´9´) a partir da segunda posição. Os identificadores também permitem o uso de espaço em branco, desde que não seja na primeira e nem na última posição. Assim, "var 1" é um identificador válido.

Literais

Literais são formas de descrever constantes no código fonte. Literais inteiros são representados como repetições de um ou mais dígitos decimais. Literais reais são formados por duas sequências de um ou mais dígitos decimais, contendo um ponto decimal entre elas, sem espaços. Literais do tipo caracter 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 ´`print`´. *Strings* consecutivas não podem ser consideradas como apenas uma, o que

significa que o caracatere de aspas duplas não pode fazer parte de uma *string*. Para incluir o caracter de aspas duplas e final de linha, devem ser usadas sequências de escape, como "\"" e "\n".

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 necesário 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 orgnização, 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.

Atualizações e Dicas

Verifique regularmente a página da discplina 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, 14 de Setembro de 2017

Anexo - Código tokens.h

```
/*
 * Lista dos tokens, com valores constantes associados.
 * Este arquivo serah posterioremente 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_INT         256
#define KW_BYTE        256
#define KW_SHORT       257
#define KW_LONG        258
#define KW_FLOAT       259
#define KW_DOUBLE      260
#define KW_IF        261
#define KW_THEN        262
#define KW_ELSE        263
#define KW_WHILE       264
#define KW_FOR         265
#define KW_READ        266
#define KW_RETURN      267
#define KW_PRINT       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_REAL       282
#define LIT_CHAR       285
#define LIT_STRING     286

#define TOKEN_ERROR    290

/* END OF FILE */
```