Especificação da Etapa 1 do Projeto de Compilador

Análise Léxica e Tabela de Símbolos

O trabalho como um todo consiste no projeto e implementação de um compilador funcional para uma determinada gramática de linguagem de programação. Esta primeira etapa do trabalho consiste em fazer um analisador léxico utilizando a ferramenta de geração de reconhecedores **Flex** e inicializar uma tabela com os símbolos relevantes encontrados, incluindo os atributos destes símbolos tais como qual a linha no arquivo do lexema correspondente.

Este texto está organizado da seguinte forma. A Seção 1 apresenta as funcionalidades necessárias na implementação desta etapa. A descrição dos *tokens* do analisador léxico está presente na Seção 2. Por fim, a Seção 3 traz requerimentos adicionais e obrigatórios na realização desta etapa, inclusive o requerimento de que a função `main` fornecida não deve ser alterada, pois é utilizada na forma como está pela avaliação automática.

1 Funcionalidades Necessárias

A solução do grupo para a análise léxica deve ter as seguintes funcionalidades.

1.1 Definir expressões regulares

Reconhecimento dos lexemas correspondentes aos *tokens* descritos na Seção 2, unicamente através da definição de expressões regulares no arquivo da ferramenta **Flex**. Cada expressão regular deve estar associada a pelo menos um tipo de *token*, ou a mais de um *token* quando necessário. O tipo de um *token* é definido através das constantes definidas no arquivo de configuração do **Bison** que é fornecido ou através de códigos *ASCII* para caracteres simples.

1.2 Implementar uma tabela de símbolos

Implementar uma estrutura de dados que será a tabela de símbolos do compilador. Esta tabela deve ser implementada como uma estrutura na forma de um dicionário onde cada entrada é representada por uma chave e um conteúdo. A chave, única no dicionário, deve ser uma cadeia de caracteres do tipo `char*` enquanto que o conteúdo correspondente deve ser uma `struct` com diferentes campos que mudam ao longo das etapas do projeto de compilador. Na etapa um, o conteúdo das entradas na tabela de símbolos está especificado na Subseção 1.4. Para facilitar a codificação da tabela de símbolos, o nome do tipo de dado do dicionário deve ser `comp_dict_t`, enquanto que as entradas no dicionário devem ser do tipo cujo nome é `comp_dict_item_t`. Esses novos tipos de dados devem vir acompanhados de funções para gerenciá-los, tais como funções de criação, alteração, adição de uma nova entrada, etc. **Deve-se prever a existência de várias tabelas de símbolos no projeto de compilador**.

1.3 Controlar o número da linha da entrada

Controlar o número de linha do arquivo fonte, implementando uma função que deve ser obrigatoriamente implementada com o protótipo `int getLineNumber(void)` (no arquivo `misc.c`) e será usada nos testes automáticos.

1.4 Preencher a tabela de símbolos

A tabela de símbolos deve ser preenchida somente com os *tokens* identificadores e literais (inteiros, ponto flutuantes, caracteres e cadeia de caracteres). Os outros tipos de *tokens*, tais como palavras reservadas, caracteres especiais e operadores compostos devem estar ausentes da tabela de símbolos. A chave de cada entrada na tabela de símbolos deve ser o **lexema** correspondente ao *token* encontrado. No caso dos identificadores, deve ser o nome do identificador. No caso dos literais, deve ser a cadeia de caractere correspondente ao literal. O conteúdo de cada entrada na tabela de símbolos deve ser o número da linha onde o último lexema correspondente foi encontrado. Isto implica que, nesta etapa, o único campo da `struct comp_dict_item_t` é um valor inteiro que contém o número da linha onde o lexema foi encontrado. Na ocorrência de múltiplos lexemas idênticos na entrada, como é o caso quando um identificador aparece várias vezes no código fonte, somente o número da linha da última ocorrência deve estar registrado na entrada correspondente. É importante notar que os arquivos começam pela linha número 1 (um).

1.5 Ignorar comentários

Ignorar comentários no formato C99 de única linha e múltiplas linhas, incluindo espaços em branco. Exemplos de comentários válidos de acordo com a especificação C99 e que devem ser ignorados através de expressões regulares:

```
/* este
   //
     é um comentário multi-linha */
// este é um comentário de uma linha
```

1.6 Lançar erros léxicos

Lançar erros léxicos ao encontrar caracteres inválidos na entrada, retornando o *token* de erro correspondente.

2 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, utiliza-se uma constante, com recursos do **Bison**, com um código maior do que 255 para representá-los.

Os *tokens* se enquadram em diferentes categorias: palavras reservadas da linguagem, caracteres especiais, operadores compostos, identificadores e literais. O analisador léxico deve, para as categorias de palavras reservadas, operadores compostos, identificadores e literais, retornar o *token* correspondente de acordo o que está definido no arquivo `parser.y` (veja as linhas que começam por `%token`). Para a categoria de caracteres especiais, o analisador léxico deve retornar o código *ASCII* através de uma única regra que retorna `yytext[0]`.

2.1 Palavras Reservadas da Linguagem

As palavras reservadas da linguagem são:

```
int float bool char string if then else while do input output return const static
```

2.2 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. São eles:

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

2.3 Operadores Compostos

A linguagem possui operadores compostos, além dos operadores representados por alguns dos caracteres da seção anterior. Os operadores compostos são:

```
<=      >=      ==      !=      &&      ||      >>      <<
```

2.4 Identificadores

Os identificadores da linguagem são usados para designar nomes de variáveis e funções. Eles são formados por um caractere alfabético seguido de zero ou mais caracteres alfanuméricos, onde considera-se caractere alfabético como letras maiúsculas ou minúsculas ou o caractere sublinhado `_` e onde são dígitos: `0`, `1`, `2`, ..., `9`.

2.5 Literais

Literais são formas de descrever constantes no código fonte. Literais do tipo `int` são representados como repetições de um ou mais dígitos. Literais em `float` são formados como um inteiro seguido de ponto decimal e uma sequência de dígitos. Literais do tipo `bool` podem ser `false` ou `true`. Literais do tipo `char` são representados por um único caractere entre entre aspas simples como por exemplo o `'a'`, `'='` e `'+'`. Literais do tipo `string` são qualquer sequência

de caracteres entre aspas duplas, como por exemplo `"meu nome"` ou `"x = 3;"`. Os literais do tipo `char` e `string` devem ser inseridos na tabela de símbolos sem as aspas que os identificam no código fonte.

3 Requisitos Obrigatórios

A função `main` deve estar em um arquivo chamado `main.c`. Ela não deve ser alterada sob qualquer hipótese. Outros arquivos fontes são encorajados de forma a manter a modularidade do código fonte. Sugere-se e edição do arquivo `misc.c`, com suas funções `main_init` e `main_finalize` para a eventual necessidade de alocar e liberar estruturas de dados globais. A entrada para o *flex* deve estar em um arquivo com o nome `scanner.l`. As subseções seguintes apresentam os requisitos técnicos obrigatórios nesta etapa do projeto de compiladores. Elas serão consideradas na avaliação subjetiva da etapa.

3.1 Git e Cmake

A solução desta etapa do projeto de compiladores deve vir acompanhada de um repositório git que manteve o histórico de desenvolvimento do projeto. Cada commit deve ser o menor possível (utilize a ferramenta `git gui` para comitar apenas parte do arquivo modificado). Cada ação de commit deve vir com mensagens significativas explicando a mudança feita. Todos os membros do grupo devem ter feito ações de commit, pelo fato deste trabalho ser colaborativo. Estas duas ações – mensagens de commit e quem fez o commit – serão obtidas pelo professor através do comando `git log` na raiz do repositório solução do grupo. O comando `git blame` também será utilizado para verificar a participação de todos os membros do grupo na construção da etapa.

- **Nota importante:** O repositório git utilizado pelo grupo deve ser **privado aos membros do grupo**. O endereço do repositório deve ser informado ao professor para leitura e, no caso de necessidade, para escrita.

3.2 Código Inicial

O código inicial do projeto encontra-se no `bitbucket.org`, e pode ser clonado e inicialmente compilado (supondo que as bibliotecas necessárias para compilação já estão instaladas) assim:

```
$ git clone https://bitbucket.org/schnorr/compil-2015-2.git
$ cd compil-2015-2
$ mkdir build
$ cd build
$ cmake ..
$ make
```

Note que o arquivo `scanner.l`, que deverá ser fortemente modificado para atender aos requisitos deste trabalho, está praticamente vazio. A solução do aluno deve partir deste código inicial e utilizar a mesma estrutura de diretórios. Se novos arquivos de código fonte devem ser adicionados, modifique o arquivo `CMakeLists.txt` apropriadamente para que o novo arquivo seja incluído no processo de compilação.

3.3 Avaliação automática

Um conjunto de testes já estão disponíveis no repositório para que o grupo possa se autoavaliar. Para lançar estes testes, depois de ter compilado o programa utilizando as instruções acima e com a ferramenta `valgrind` instalada, execute o seguinte comando no diretório onde encontra-se os arquivos compilados:

```
ctest -R e1
```

4 Dúvidas

Crie um novo tópico no fórum de discussão do projeto de compilador no moodle.

Bom trabalho!