Especificação da Etapa 2 do Projeto de Compilador

Análise Sintática

O trabalho consiste no projeto e implementação de um compilador funcional para uma linguagem de programação que a partir de agora chamaremos de **Linguagem Amarela**. Na segunda etapa do trabalho é preciso fazer um analisador sintático utilizando a ferramenta de geração de reconhecedores **Bison** e continuar o preenchimento da tabela de símbolos com outras informações encontradas, associando os valores corretos aos *tokens*. De uma forma geral, o analisador sintático deve verificar se a sentença fornecida (o programa de entrada) faz parte da linguagem ou não.

1 Funcionalidades Necessárias

A solução do grupo para a análise sintática deve ter as seguintes funcionalidades:

1.1 Definir a gramática da linguagem

A gramática da linguagem **Amarela** deve ser definida de acordo com a descrição geral apresentada na Seção 2. As regras gramaticais devem ser incorporadas ao arquivo que configura o código gerado pelo **Bison**, de forma que o analisador sintático gerado pela ferramenta possa realizar a análise sintática de forma apropriada.

1.2 Realizar a análise sintática e relatório de erros

Caso a análise sintática termine de forma correta, o programa deve retornar o valor da constante de pré-processamento identificada por `SINTATICA_SUCESSO`. Esse retorno deve obrigatoriamente ser feito pela função principal `main` (note que segundo a Seção 3, o arquivo `main.c` não pode ser alterado). Caso a entrada não seja reconhecida, deve-se imprimir uma mensagem informando a linha do código da entrada que gerou o erro sintático e informações adicionais que auxiliem o programador que está utilizando o compilador a identificar o erro sintático identificado. Na ocasião de uma mensagem de erro, o analisador sintático deve retornar o valor da constante de pré-processamento identificada por `SINTATICA_ERRO`.

- **Nota importante:** Deve-se obrigatoriamente utilizar estas duas constantes de pré-processamento, pois seus valores serão utilizados durante o processo de avaliação automática (incluindo os testes realizados com `ctest`).

1.3 Enriquecimento da tabela de símbolos

Uma vez que vários lexemas da entrada podem representar *tokens* de tipos diferentes, a tabela de símbolos deve ser alterada de forma que a chave de cada uma das entradas não seja mais simplesmente o lexema, mas a combinação entre o lexema e o tipo do *token*. O tipo de um determinado *token* pode ser somente um dentre as seguintes constantes. Elas estão definidas no arquivo `main.h` do repositório e podem ser livremente utilizadas em qualquer parte do código.

```
#define SIMBOLO_LITERAL_INT    1
#define SIMBOLO_LITERAL_FLOAT  2
#define SIMBOLO_LITERAL_CHAR   3
#define SIMBOLO_LITERAL_STRING 4
#define SIMBOLO_LITERAL_BOOL   5
#define SIMBOLO_IDENTIFICADOR  6
```

O conteúdo de cada entrada na tabela de símbolos deve ter pelo menos três campos: número da linha da última ocorrência do lexema, o tipo do *token* da última ocorrência, e o valor do token convertido para o tipo apropriado (inteiro, ponto-flutuante, char, booleano ou cadeia de caracteres). O segundo campo, representado pelo tipo do *token* é o mesmo utilizado na chave da entrada, mas desta vez com um campo específico para tal. O valor do token é um campo que pode assumir diferentes tipos: uma possibilidade é utilizar a construção `union` da linguagem C para conter os diferentes tipos possíveis para os símbolos. A conversão deve ser feita utilizando funções tais como `atoi`, no caso de números inteiros, e `atof`, no caso de ponto-flutuantes. Os tipos caractere e cadeia de caracteres não devem conter aspas no campo valor.

1.4 Associação da entrada na tabela ao *token* correspondente

O analisador léxico é o responsável pela criação da entrada na tabela de símbolos para um determinado *token* que acaba de ser reconhecido. Nesta etapa, deve-se associar um ponteiro para a estrutura de dados que representa o conteúdo da entrada na tabela de símbolos ao *token* correspondente. Esta associação deve ser feita pelo analisador léxico (ou seja, no arquivo `scanner.l`) de forma a melhorar o desempenho das demais fases do compilador.

Esta associação deve ser realizada através do uso da variável global `yylval`[1], que é usada pelo **Flex** para dar um "valor" ao *token*, além do identificador retornado imediatamente após o reconhecimento. Como esta variável global pode ser configurada com a diretiva `%union`, sugere-se o uso do campo `valor_simbolo_lexico` para a associação. Portanto, a associação deverá ser feita através de uma atribuição para a variável `yylval.valor_simbolo_lexico`.

1.5 Remoção de conflitos gramaticais

Deve-se realizar a remoção de conflitos *Reduce/Reduce*[2] e *Shift/Reduce*[3] de todas as regras gramáticas. Estes conflitos devem ser tratados através do uso de configurações para o bison (veja a documentação sobre `%left`, `%right` ou `%nonassoc`). Os mesmos podem ser observados através de uma análise cuidadosa do arquivo `parser.output` que será gerado automaticamente no momento da execução do make. Notem que a remoção de conflitos pode ser feita, em alguns casos, somente através da re-escrita da gramática.

2 Descrição Geral da Linguagem Amarela

Um programa na linguagem **Amarela** é composto por um conjunto opcional de declarações de variáveis globais e um conjunto opcional de funções, que podem aparecer intercaladamente e em qualquer ordem. Todas as declarações de variáveis globais são **terminadas** por ponto-e-vírgula. Cada função é descrita por um cabeçalho seguido do seu corpo, como descrito na Seção 2.2. Os comandos simples podem ser aqueles descritos na Seção 2.3.

2.1 Declarações de Variáveis Globais

As variáveis são declaradas pelo seu tipo, seguidas pelo seu nome. O tipo pode estar precedido opcionalmente pela palavra reservada `static`. A linguagem inclui também a declaração de vetores, feita pela definição de seu tamanho inteiro positivo entre colchetes, colocada à direita do nome, ou seja, ao final da declaração. Variáveis podem ser dos tipos `int`, `float`, `char`, `bool` e `string`.

2.2 Definição de Funções

Cada função é definida por um cabeçalho e um corpo. O **cabeçalho** consiste no tipo do valor de retorno, seguido pelo nome da função e terminado por uma lista. O tipo pode estar precedido opcionalmente pela palavra reservada `static`. A lista é dada entre parênteses e é composta por zero ou mais parâmetros de entrada, separados por vírgula. Cada parâmetro é definido pelo seu tipo e nome, e não pode ser do tipo vetor. O tipo de um parâmetro pode ser opcionalmente precedido da palavra reservada `const`. O **corpo** da função é um bloco de comandos, como definido a seguir na Seção 2.4. A função **não** deve ser terminada por ponto-e-vírgula.

2.3 Comandos Simples

Os comandos simples da linguagem podem ser: declaração de variável local, atribuição, construções de fluxo de controle, operações de entrada, de saída, e de retorno, um bloco de comandos, chamadas de função, e o comando vazio. O comando vazio só pode aparecer em um bloco de comandos. Eles são detalhados a seguir:

Declaração de variável Uma declaração de variável local consiste no tipo da variável precedido opcionalmente pela palavra reservada `static`, e o nome da variável. As declarações locais, ao contrário das globais, não permitem vetores e podem permitir o uso da palavra reservada `const` antes do tipo (após a palavra reservada `static` caso esta aparecer). Uma variável local pode ser opcionalmente inicializada com um valor válido caso sua declaração seja seguida do operador composto "`<=`" e de um identificador ou literal.

Atribuição Na atribuição, usa-se uma das seguintes formas:

[1] `http://www.gnu.org/software/bison/manual/html_node/Token-Values.html`
[2] `http://www.gnu.org/software/bison/manual/html_node/Reduce_002fReduce.html`
[3] `http://www.gnu.org/software/bison/manual/html_node/Shift_002fReduce.html`

```
identificador = expressão
identificador[expressão] = expressão
```

Os tipos corretos para o assinalamento e para o índice serão verificados somente na análise semântica.

Entrada O comando é identificado pela palavra reservada `input`, seguida de uma expressão, seguida pelo operador composto "`=>`", seguido por outra expressão.

Saída O comando é identificado pela palavra reservada `output`, seguida de uma lista de expressões separadas por vírgulas.

Retorno O comando de retorno é identificado pela palavra reservada `return` seguida de uma expressão.

Fluxo de controle Os comandos de controle de fluxo são descritos a seguir, na Seção 2.6.

Comando vazio Para facilitar a escrita de programas aceitando o caractere de ponto-e-vírgula como terminador, e não apenas separador, a linguagem deve aceitar também o comando vazio.

Chamada de função Uma chamada de função é um comando, identificado pelo nome da função, seguido de argumentos entre parênteses e separados por vírgula. Cada argumento pode ser uma expressão.

Shift Um comando de shift aparece nas seguintes formas:

```
identificador << numero
identificador >> numero
```

onde número é um literal inteiro.

2.4 Bloco de Comandos

Um bloco de comandos é definido entre chaves, e consiste em uma sequência, possivelmente vazia, de comandos simples, **separados** por ponto-e-vírgula. Um bloco de comandos é considerado como um comando único simples, recursivamente, e pode ser utilizado em qualquer construção que aceite um comando simples.

2.5 Expressões Aritméticas e Lógicas

As expressões aritméticas têm como folhas identificadores, opcionalmente seguidos de expressão inteira entre colchetes, para acesso a vetores, ou podem ser literais numéricos e em código `ASCII`. As expressões aritméticas podem ser formadas recursivamente com operadores aritméticos, assim como permitem o uso de parênteses para associatividade. Expressões lógicas podem ser formadas através dos operadores relacionais aplicados a expressões aritméticas, ou de operadores lógicos aplicados a expressões lógicas, recursivamente. Outras expressões podem ser formadas considerando variáveis lógicas do tipo `bool`. Nesta etapa do trabalho, porém, não haverá distinção alguma entre expressões aritméticas, inteiras, de caracteres ou lógicas. A descrição sintática deve aceitar qualquer operadores e subexpressão de um desses tipos como válidos, deixando para a análise semântica das próximas etapas do projeto a tarefa de verificar a validade dos operandos e operadores. Finalmente, um operando possível de expressão é uma chamada de função, como descrito na Seção 2.3 acima. O caractere especial "`-`" (veja especificação da etapa 1) indica subtração.

2.6 Comandos de Fluxo de Controle

Para o controle de fluxo, a linguagem **Amarela** possui duas construções condicionais e duas construções de repetição, descritas informalmente como segue:

```
if (expressão) then comando
if (expressão) then comando else comando
while (expressão) do comando
do comando while (expressão)
```

3 Requisitos Obrigatórios

A função `main` deve estar em um arquivo chamado `main.c`, e 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`. A entrada para o **Bison** deve estar em um arquivo com o nome `parser.y`. 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 desta etapa do projeto é o código obtido como resultado na etapa anterior, mais o código atualizado do repositório git do professor. Para realizar um *merge* das modificações e ativar o funcionamento da função `main` correspondente a etapa 2, deve-se executar os seguintes comandos, configurando também o processo de compilação com a seguinte sequência de comandos (supondo que o diretório corrente é `build` e que o diretório superior contém os fontes do programa, e que o nome `origin` é o nome do repositório git do professor):

```
$ git pull origin master
$ cmake -DETAPA_1=OFF -DETAPA_2=ON ..
$ make
```

O arquivo `parser.y` deverá ser fortemente modificado para atender aos requisitos deste trabalho. A solução do aluno deve partir disto 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 e2
```

Caso os testes falhem, utilize a opção `-V` com o comando `ctest`. Informações adicionais serão apresentadas, e sua interpretação poderá auxiliá-los a resolver os problemas que geraram as falhas detectadas.

4 Dúvidas

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

Bom trabalho!