Trabalho Prático 2
Avaliador de Expressões

Arthur Souto Lima

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

arthursl@ufmg.br

1. Introdução

O projeto consiste em implementar um avaliador de expressões, que deve receber um programa de entrada, na gramática definida na especificação, dividi-lo em um analisador léxico e analisar tais tokens em um analisador sintático. O método de análise sintática é o método ascendente SLR, que exige a computação de funções auxiliares como FIRST e FOLLOW, além de tabelas, como ACTION e GOTO. Após a análise, o programa deve imprimir se os programas passados estão de acordo ou não com a linguagem da gramática.

2. Instruções

2.1. Compilação

O programa foi desenvolvido em C++ e acompanha um makefile para sua compilação. Basta executar o comando *make* para compilar o programa. O executável final se chama *main* e estará na pasta raiz do programa (onde está também o makefile).

2.2. Execução

Compilado o programa, deve-se rodar o executável *main*, por exemplo, no terminal, via o comando

./main

Após iniciar, o programa requisitará que seja digitada a expressão (ou as expressões, separadas por vírgula) em seguida. Digite o que for desejado e aperte ENTER. O programa fará a análise léxica seguida da análise sintática para cada expressão passada. Se, durante alguma das análises, for detectado algum erro, o programa o acusará e encerrará a execução.

Caso algum identificador tenha sido utilizado, o programa requisitará o seu respectivo valor, que deve ser inteiro, real ou lógico. O programa insistirá até que o valor inserido seja algum desses três tipos.

Numa execução de um programa correto, serão impressos os tokens encontrados e, em seguida, a confirmação do analisador sintático, a impressão do ”Programa Aceito”. Processadas todas as expressões, o programa será encerrado automaticamente.

2.2.1. Sair

Se, ao invés de uma expressão, for inserido apenas *quit* no programa, ele encerrará sua execução.

2.2.2. Impressão das Tabelas

Caso se deseje, o programa pode imprimir as tabelas ACTION e GOTO que ele cria para a gramática do projeto. Basta inserir *print* ao invés de uma expressão. As tabelas, bem como cada um dos estados, estarão no arquivo *print.txt*, também na pasta raiz do programa (onde está o makefile). O programa já possui uma impressão no seu arquivo *print.txt* no momento da entrega.

3. Implementação

3.1. Visão Geral

O programa foi implementado em C++, compilado pelo compilador G++ da GNU Compiler Collection. Juntamente com os códigos-fonte, está incluso também um arquivo *makefile* que é utilizado para compilar o programa através do comando *make*.

Os códigos-fonte estão na pasta *src* e os arquivos-cabeçalho na pasta *include*. O projeto está separado em vários módulos para facilitar o desenvolvimento bem como entendimento de quem o analisa. Um módulo típico possui um arquivo *.cpp* e um arquivo *.h* ambos com o mesmo nome do módulo. A seguir estão os principais módulos, seus respectivos arquivos e funcionalidades:

- **Analisador Léxico**: módulo *lex*, inclui a função *findTokens* que divide o programa dado em tokens e uma série de funções auxiliares para detectar cada tipo de token definido na especificação;
- **Analisador Sintático**: módulo principal *parser*, basicamente a implementação do algoritmo de parsing visto nas aulas. Precisa de vários módulos auxiliares para tabelas (módulo *tabelas*), tratamento de estruturas da gramática (módulos *gram*, *prod*, *cadeia*, *symbol*), tratamento de itens (módulos *item* e *conj*) e gerenciamento das ações (módulo *action*);

- **Funções Auxiliares I**: as funções auxiliares como FIRST e FOLLOW estão implementadas como métodos da classe Gramática (módulo *gram*);
- **Funções Auxiliares II**: as funções auxiliares como CLOSURE, GOTO, ITENS e expandir a gramática estão implementadas no módulo *utils*
- **Gramática Proposta**: a gramática constante na especificação está implementada seguindo o framework desenvolvido no trabalho, ou seja, como um objeto da classe *Gramatica*, encontra-se no arquivo cabeçalho *tp.h*.
- **Estruturas de Gramática**: para modularizar e abstrair muitas das operações sobre estruturas menores das gramáticas, foram criadas classes para símbolos, terminais e não-terminais, (módulo *symbol*); classe para cadeias, sequências de símbolos, (módulo *cadeia*); classe para produções, lista de cadeias, com um não-terminal do lado esquerdo (módulo *prod*); classe para representar toda uma gramática e suas produções (módulo *gram*);
- **Estruturas de Item**: para lidar com os itens LR(0) e suas funcionalidades, há a classe Item (módulo *item*). Além disso, a máquina de estados (conjunto de conjuntos de item) está no módulo *conj*. A função que calcula todos os items (constrói a máquina) está no módulo *utils*.

3.2. Concepção

O programa foi baseado nos algoritmos vistos em aula, isto é, baseado naqueles constantes em [Aho et al. 2007]. Além deste, para o analisador léxico, também serviu de inspiração o algoritmo apresentado por [Khan 2017].

Como explicitado na seção anterior, há diversos módulos no programa. Isso foi feito para facilitar o desenvolvimento, propiciando que cada módulo fosse criado e testado separadamente. Além disso, isso abstrai peculiaridades de estruturas maiores nas maiores, isto é, por exemplo, a gramática não precisa saber como estão guardadas as regras de uma de suas produções, mas consegue acessá-las pelo operador [] de *Producao*.

3.3. Estruturas de Dados

3.3.1. Estruturas Gerais

Muitas das bibliotecas-padrão de C++ foram utilizadas para auxiliar no TP. A biblioteca de string foi extensivamente usada em praticamente todos os módulos, já que ela representava o núcleo de um terminal e também de um não-terminal. Os *smart pointers* de C++ também foram utilizados, principalmente o *shared pointer*, para podermos utilizar os ponteiros de C, mas com funcionalidades extras.

Além disso, os contêineres da biblioteca padrão (STL): *vector*, *set* e *stack* foram utilizados em vários módulos do projeto, princpalmente o *vector*. Além destes, o *pair* foi muito utilizado na construção das tabelas.

3.3.2. Estruturas de Gramática

As classes *Terminal* e *NaoTerminal* (derivadas de *Symbol*) são a unidade monomérica das representações de gramática. Subindo a hierarquia, as estruturas de dados escolhidas são listadas a seguir:

- ***Cadeia***: *vector* de símbolos, já que é uma sequência ordenada;
- ***Producao***: *vector* de cadeias, representando o lado direito da produção e um não-terminal representando o lado esquerdo;
- ***Gramatica***: *vector* de produções

3.3.3. Estruturas de Item

Os itens possuem a mesma organização que uma produção, tanto que o construtor de *Item* copia uma cadeia específica de uma produção. As únicas diferenças são que, como comentado, um item só pode possuir uma cadeia (uma regra) da produção e deve possuir um ponto em alguma posição dessa cadeia.

A máquina de estados é um *set* de *sets* de *Item*, ou seja, um conjunto de conjuntos de item.

3.3.4. Analisador Sintático

O analisador sintático deve usar pilhas para utilizar a máquina de estados. Por conveniência, são usadas duas pilhas (*stack* da STL), uma de inteiros, para guardar o índice dos estados, e uma de símbolos (classe *Symbol*), para guardar os terminais e não terminais que estão sendo processados ou que estão sendo processados. Utilizar duas pilhas facilita a implementação, já que uma pilha só pode conter elementos de um tipo.

As tabelas são *vectors* de *vectors* (na verdade *vectors* de ponteiros de *vectors*, para evitar cópias desnecessárias de objetos complexos). A tabela ACTION guarda pares (*pair* da STL) de *Terminal* e uma *Acao*. A tabela GOTO guarda pares de *NaoTerminal* e um inteiro (índice de estado).

A entrada do analisador sintático é um *vector* de *Symbol* (na verdade de ponteiros de *Symbol*).

3.3.5. Analisador Léxico

A entrada do Analisador Léxico é uma string de C++, que será processada caractere a caractere, produzindo o *vector* de (ponteiros de) *Symbol*. Este será redirecionado pelo *main* para o analisador sintático.

3.4. Desafios e Decisões

A parte mais desafiadora, bem como mais trabalhosa, deste projeto foi conceber e implementar os diferentes níveis de abstração para cada hierarquia da representação da gramática, para que fosse possível tanto a criação quanto a manipulação de forma eficaz e simples de uma gramática.

As principais decisões quanto às estruturas de dados estão descritas nas subseções anteriores. Adotou-se um paradigma de Orientação à Objetos, propiciada por C++, para separar interesses, criando diferentes classes para objetos com diferentes funcionalidades e responsabilidades.

Dada a complexidade de muitos objetos e a intensa passagem de parâmetros para diversas funções e métodos de variados módulos, teve-se o cuidado de evitar passar objetos por cópia, mas sim por referência (seja referência "&" de C++ ou por ponteiros), evitando que sejam criados muito mais objetos além dos necessários.

4. Exemplos

A seguir estão várias entradas no programa, com o "hardcopy" do terminal. Normalmente, fazemos um teste que possui erro e, logo em seguida, o corrigimos para que seja aceito.

4.1. Teste Simples

Uma expressão simples, somando 1 e 1. Inicialmente fazemos um teste que falha, pois o operador " ^ " não está definido na gramática.

```
==========AVALIADOR DE EXPRESSÕES==========
Digite a expressão: 1 ^ 1
Valid Unsigned Constant : 1
Erro: identificador inválido: ^
```

Em seguida o corrigimos para o sinal ”+” e o programa é aceito com sucesso.

```
==========AVALIADOR DE EXPRESSÕES==========
Digite a expressão: 1 + 1
Valid Unsigned Constant : 1
Valid operator (1-char): +
Valid Unsigned Constant : 1
(constant, 1)
(ADDOP, +)
(constant, 1)
Programa Aceito
```

4.2. Expressão mais complicada

Com uma expressão um pouco mais complicada agora e um erro sutil: a chamada de função deve possuir parênteses.

```
==========AVALIADOR DE EXPRESSÕES==========
Digite a expressão:3.02 - (2*cos 45)
Valid Unsigned Constant : 3.02
Valid operator (1-char): -
Valid Unsigned Constant : 2
Valid operator (1-char): *
Valid operator (keyword) : cos
Valid Unsigned Constant : 45
(constant, 3.02)
(ADDOP, -)
((, ()
(constant, 2)
(MULOP, *)
(SPECOP, cos)
(constant, 45)
(), ))
Erro de Sintaxe Geral
```

Colocando os parênteses para que o teste seja aceito.

```
==========AVALIADOR DE EXPRESSÕES==========
Digite a expressão: 3.02 - (2*cos(45))
Valid Unsigned Constant : 3.02
Valid operator (1-char): -
Valid Unsigned Constant : 2
Valid operator (1-char): *
Valid operator (keyword) : cos
Valid Unsigned Constant : 45
(constant, 3.02)
(ADDOP, -)
((, ()
(constant, 2)
(MULOP, *)
(SPECOP, cos)
((, ()
```

```
(constant, 45)
(), ))
(), ))
Programa Aceito
```

4.3. Várias expressões

Por fim, um teste que possui várias expressões. A tentativa inicial possui um erro na segunda expressão, o que interromperá a avaliação das expressões. Nesse teste também observamos a entrada de variáveis pelo usuário, observe que o usuário deve inserir um valor inteiro, real ou lógico para o programa continuar.

```
==========AVALIADOR DE EXPRESSÕES==========
Digite a expressão: (var1 + 2) * 7, var2 + 12.003e -5, (5 >= log (2*2))
Valid Identifier : var1
        Valor de <var1>: var1
        Valor de <var1>: T
        Valor de <var1>: V
        Valor de <var1>: 10
Valid operator (1-char): +
Valid Unsigned Constant : 2
Valid operator (1-char): *
Valid Unsigned Constant : 7
((, ()
(id, 10)
(ADDOP, +)
(constant, 2)
(), ))
(MULOP, *)
(constant, 7)
Programa Aceito

Valid Identifier : var2
        Valor de <var2>: var
        Valor de <var2>: var2
        Valor de <var2>: var1
        Valor de <var2>: -35e-2
Valid operator (1-char): +
Valid Unsigned Constant : 12.003e
Valid Signed Constant : -5
(id, -35e-2)
(ADDOP, +)
(constant, 12.003e )
(constant, -5)
Erro de Sintaxe Geral
```

Corrigindo, temos um espaço entre o ”E” do expoente da constante da segunda expressão.

```
==========AVALIADOR DE EXPRESSÕES==========
Digite a expressão: (var1 + 2) * 7, var2 + 12.003e-5, (5 >= log (2*2))
Valid Identifier : var1
        Valor de <var1>: 10
Valid operator (1-char): +
Valid Unsigned Constant : 2
Valid operator (1-char): *
Valid Unsigned Constant : 7
((, ()
(id, 10)
(ADDOP, +)
(constant, 2)
(), ))
(MULOP, *)
(constant, 7)
Programa Aceito

Valid Identifier : var2
        Valor de <var2>: -35E-2
Valid operator (1-char): +
Valid Unsigned Constant : 12.003e
(id, -35E-2)
(ADDOP, +)
(constant, 12.003e-5)
Programa Aceito

Valid Unsigned Constant : 5
Valid operator (2-char): >=
Valid operator (keyword) : log
Valid Unsigned Constant : 2
Valid operator (1-char): *
Valid Unsigned Constant : 2
((, ()
(constant, 5)
(RELOP, >=)
(SPECOP, log)
((, ()
(constant, 2)
(MULOP, *)
(constant, 2)
(), ))
(), ))
Programa Aceito
```

5. Conclusão

Ao final deste trabalho prático foi possível experienciar o papel de um designer de compilador. Apresentaram-se diversos desafios no que tange ao analisador léxico e sintático. Felizmente, com o auxílio do livro-texto [Aho et al. 2007], bem como das indicações e direcionamentos proporcionados pela professora, que foram de ímpar valor, foi possível desenvolver o projeto com sucesso. Certamente os aprendizados foram muito proveitosos, principalmente para aplicar na prática os conceitos e algoritmos vistos durante as aulas.

References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. (2007). *Compilers Principles,Techniques, Tools*. Addison Wesley.

Khan, M. I. (2017). C program to detect tokens in a c program. [1]. Acesso em: 08 jan 2021.

[1] `https://www.geeksforgeeks.org/c-program-detect-tokens-c-program/`