SCC0605 Teoria da Computação e Compiladores

Prof. Dr. Thiago A.S Pardo

Trabalho 2: Analisador Sintático

Nome:	*Número USP:*
Carla Nunes	8479343
Leandro A. Silva	9805341
Marilene A. Garcia	10276974

1 Introdução

Dando continuidade ao trabalho 1, desenvolvido anteriormente na disciplina de Teoria da Compilação e Compiladores (SCC0605), o trabalho 2 teve como objetivo principal a implementação do analisador sintático do compilador. Bem como, o tratamento de erros necessários para essa análise sintática. Além disso, foram feitos alguns ajustes referentes ao analisador léxico.

A implementação do trabalho 1 e o relatório desta parte podem ser encontrados no seguinte repositório do GitHub:
github.com/CarlaNunes/Analisador-L-xico.git

O grupo, optou por continuar o desenvolvimento do projeto na linguagem de programação C++ por questão de compatibilidade com a parte anterior. Assim como na parte léxica, foi criada uma biblioteca própria das funções da análise sintática. Essa biblioteca está nos arquivos **"sintatico.h"** e **"sintatico.cpp"**. O código **"main.cpp"** faz a chamada da função de análise sintática após o término da análise léxica, gerando a saída, **"saida.txt"**, indicando os erros léxicos e sintáticos encontrados na compilação e suas respectivas linhas.

Toda implementação do analisador sintático encontra-se no seguinte repositório do GitHub:
github.com/leandroS08/SCC0605-Compilador._P–

Para compilar e executar o projeto, foi usado um ambiente *Linux*, diretamente no terminal. Os comandos necessários estão listados abaixo. Sendo o *pasta* substituído pela pasta na qual estão os arquivos. A pasta *errados* contém exemplos de códigos com erros, já a *certos*, exemplos de códigos sem erros. E o $<X>$ indica o índice do arquivo de entrada teste desejado e $<pasta>$ deve ser "certo"ou "errado"para inficar códigos sem ou com erro, respectivamente.

```
g++ main.cpp lexico.cpp sintatico.cpp -o exe
./exe <pasta>/<X>_input.txt
```

Além do arquivo de saída, no formato exigido pela proposta do trabalho, no terminal também é mostrada uma mensagem final de compilação. Indicando, se o compilador conseguiu consumir todas as cadeias e o número de erros encontrados. Um exemplo, de mensagem de compilação no terminal:

********* COMPILAÇÃO FINALIZADA *********
- Sobraram cadeias não processadas
- Número de erros mapeados: 7

2 Correções léxicas e ajustes no projeto 1

Na parte das modificações do analisador léxico, só foi realizada a detecção da finalização correta do comentário. Ou seja, se ao final do comentário teríamos um fecha parênteses. O outro erro que seria corrigido, era do tamanho das variáveis, só que, segundo orientações seria desnecessário.

Além das correções de erros, também foram feitos algumas mudanças no código principal, dentre elas vale ressaltar:

- Transferência da função **token()** para os códigos da biblioteca de análise léxica.
- Criação de uma classe chamada **Node** em um código-cabeçalho separado chamado **"elemento.h"**, que, para essa parte do projeto, representa cada token, sua classificação léxica e a linha em que se encontra.
- Criação de uma fila - estrutura **queue** do C++ - a partir dos tokens lidos, que é o primórdio da tabela de símbolos que será implementada em partes futuras do compilador.
- Transferência da impressão dos erros no arquivo **saida.txt** para os códigos léxicos e sintáticos, em vez de serem impressos na função principal.
- Implementação de uma variável contadora de erros, léxicos e sintáticos.

3 Análise Sintática Descendente Preditiva Recursiva - ASDPR

3.1 Contextualização teórica

O analisador sintático (ou parser) é a etapa principal de um compilador, pois ele coordena as demais etapas. Utilizando-se da gramática da linguagem, ele é responsável por verificar a boa formatação do programa, isto é, se as cadeias fornecidas pelo analisador léxico estão corretas.

Ela é considerada descendente quando parte do símbolo inicial da gramática e tenta chegar aos símbolos folha. Ela é considerada preditiva quando sempre "puxa"símbolos seguintes ao seu símbolo atual de análise para conferir se o seu seguidor está correta. E ela é recursiva pois as chamadas de procedimento são feitas umas sobre as outras, ou seja, uma principal chama outros procedimentos, que por sua vez chamam outros e assim por diante.

O analisador sintático implementado neste projeto é descendente preditivo recursivo.

3.2 Gramática P–

Como dito, toda análise sintática é feita em volta da gramática da linguagem trabalhada. Para o projeto, a gramática de P– fornecida foi incrementada com a regra do uso da diretiva "for":

```
1. <programa> ::= program ident ; <corpo> .
2. <corpo> ::= <dc> begin <comandos> end
3. <dc> ::= <dc_c> <dc_v> <dc_p>
4. <dc_c> ::= const ident = <numero> ; <dc_c> | \lambda
5. <dc_v> ::= var <variaveis> : <tipo_var> ; <dc_v> | \lambda
6. <tipo_var> ::= real | integer
7. <variaveis> ::= ident <mais_var>
8. <mais_var> ::= , <variaveis> | \lambda
9. <dc_p> ::= procedure ident <parametros> ; <corpo_p> <dc_p> | \
   lambda
10. <parametros> ::= ( <lista_par> ) | \lambda
11. <lista_par> ::= <variaveis> : <tipo_var> <mais_par>
12. <mais_par> ::= ; <lista_par> | \lambda
13. <corpo_p> ::= <dc_loc> begin <comandos> end ;
14. <dc_loc> ::= <dc_v>
15. <lista_arg> ::= ( <argumentos> ) | \lambda
16. <argumentos> ::= ident <mais_ident>
17. <mais_ident> ::= ; <argumentos> | \lambda
18. <pfalsa> ::= else <cmd> | \lambda
19. <comandos> ::= <cmd> ; <comandos> | \lambda
20. <cmd> ::=    read ( <variaveis> ) |
                 write ( <variaveis> ) |
                 while ( <condicao> ) do <cmd> |
                 if <condicao> then <cmd> <pfalsa> |
                 ident := <expressao> |
                 ident <lista_arg> |
                 begin <comandos> end |
                 for <atribuicao> to <numero> do <cmd>
21. <condicao> ::= <expressao> <relacao> <expressao>
22. <relacao> ::= = | <> | >= | <= | > | <
23. <expressao> ::= <termo> <outros_termos>
24. <op_un> ::= + | - | \lambda
25. <outros_termos> ::= <op_ad> <termo> <outros_termos> | \lambda
26. <op_ad> ::= + | -
27. <termo> ::= <op_un> <fator> <mais_fatores>
28. <mais_fatores> ::= <op_mul> <fator> <mais_fatores> | \lambda
29. <op_mul> ::= * | /
30. <fator> ::= ident | <numero> | ( <expressao> )
31. <numero> ::= numero_int | numero_real
32. <atribuicao> ::= <variavel> := <numero>
33. <variavel> ::= ident
```

3.3 Tratamento de erro e modo pânico

Existem diversos métodos de tratamento de erro em uma analisador sintático, dentre eles modo de pânico, recuperação de frases, produções de erro e correção global. Por ser mais simples de implementação, foi orientado o uso do modo de pânico. Ele funciona de uma forma bem simples: quando o compilador encontra uma cadeia incompatível com a linguagem, isto é, inesperada para o estado atual, ele relata uma mensagem de erro e sua respectiva linha e então consome os tokens problemáticos até um ponto de restauração, que pode ser um identificador, um delimitador ou uma palavra-chave. Desta forma, os erros não travam a compilação.

Na implementação, a biblioteca do analisador sintático tem uma função chamada **ErroSintatico(int erro, int line, bool &flag)**, que lida com todos os erros possíveis da análise. Os parâmetros dessa função são:

- **erro**: índice do erro, que determinará a mensagem de erro que o usuário receberá
- **line**: a linha onde o erro se encontra
- **flag**: variável de controle para saber se o procedimento tem erro

Para cada índice de erro passado para a função, uma mensagem é associada. Os erros mapeados no projeto foram:

- -1: modo pânico não conseguiu recuperar a compilação
- 0: faltando -> ; (aluno usp não faz isso...)
- 1: estrutura de inicialização do programa com erros
- 2: faltando . na finalização do programa"
- 4: faltando -> begin
- 5: faltando -> end
- 6: estrutura de declaração de constante inválida
- 7: operação inválida em área de declaração
- 8: estrutura de declaração de variável inválida
- 9: estrutura declaração de procedimento inválida
- 10: variáveis inseridas de forma inválida nesse comando"
- 11: comando inválido

- 12: corpo de procedimento inválido
- 13: estrutura de repetição inválida
- 14: estrutura condicional inválida
- 15: condição inválida
- 16: estrutura de atribuição inválida
- 17: faltando ->)
- 18: fator inválido
- 19: lista de argumentos inválida

3.4 Implementação e decisões de projeto

Como mencionado anteriormente, uma biblioteca própria foi criada para as funções do analisador sintático. Os códigos dessa biblioteca estão nos arquivos **"sintatico.h"** e **"sintatico.cpp"**. A chamada inicial do analisador sintático ocorre no **"main.cpp"**, que é o código principal do projeto.

Inicialmente, o analisador léxico gera uma fila com todos os elementos lidos do arquivo de entrada, em uma classe chamada *Node* descrita no header **elemento.h** e que contém, por enquanto, o conteúdo do*token*, sua classificação léxica e a linha dentro do código em que ele se encontra. O objetivo do analisador sintático, é consumir elemento por elemento dessa fila até que ela fique vazia.

Em geral, as funções do analisador sintático que realizam o consumo das cadeias são apenas implementações das regras da gramática. Para cada uma, um tratamento de erro foi inserido em conjunto, por meio de chamadas das funções **toEmPanico()** e **newToEmPanico()**. Essas funções consomem cadeias até chegarem em pontos de restauração. Para cada chamada de erro foi dado um índice como parâmetro que determina quais serão os termos de restauração aceitos.

Se a compilação, consumir todas as cadeias e não tiver nenhum erro, o programa está correto e pertence à linguagem P–. Em caso de existência de erros, o analisador é programado para também consumir os elementos com erro no modo pânico, para não travar a execução do compilador. Nesse caso, apesar da compilação consumir todos os elementos, o programa indica os erros no arquivo de saída.

4 Exemplo de execução

4.1 Input Correto 1

4.1.1 Arquivo de entrada

```
program nome2;
const B = 5;
var a: real;
var b: integer;

procedure nomep(x : real);
    begin
        read(c, a);

        if a<x+c then
            begin
                a:= c+x;
                write(a);
            end
        else
            begin
                c:= a+x;
            end;

        for i:=1 to 10 do
            begin
                x:=x+i;
                write(x);
            end;
    end;

begin {programa principal}
    read(b);
    nomep(b);
end.
```

4.1.2 Arquivo de saída

O arquivo de saída fica vazio em códigos sem erros.

4.1.3 Resposta no terminal

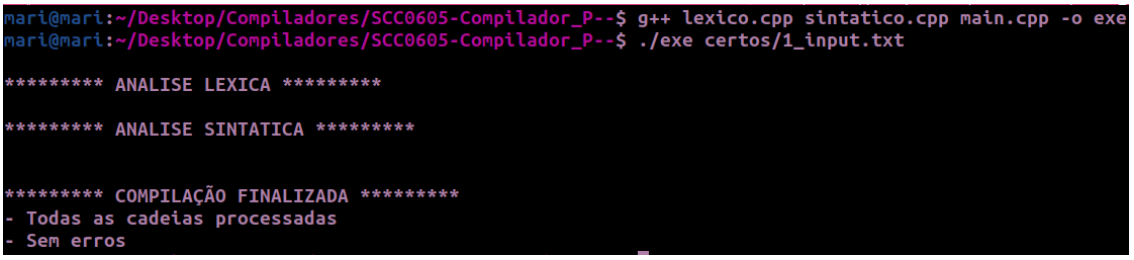

Figura 1: Resposta no terminal ao código sem erros

4.2 Input Incorreto 8

4.2.1 Arquivo de entrada

```
program nome2;
{exemplo 2}
var @a real;
var b integer;
procedure (x: real);
    var a, c: integer;
    begin
    read(c, a)
    if a<x+c then
        begin
        a:= c+x
        write( );
        end
    elsa c:= a + x;
    end;
begin {programa principal}
red(b);
nomep( );
end.
```

4.2.2 Arquivo de saída

```
Erro lexico na linha 3: caractere nao permitido
Erro sintatico na linha 3: variaveis inseridas de forma invalida
   nesse comando
Erro sintatico na linha 3: estrutura de declaracao de variavel
   invalida
Erro sintatico na linha 4: estrutura de declaracao de variavel
   invalida
Erro sintatico na linha 5: estrutura declaracao de procedimento
   invalida
```

```
6 Erro sintatico na linha 12: variaveis inseridas de forma invalida
     nesse comando
7 Erro sintatico na linha 14: comando invalido
8 Erro sintatico na linha 18: lista de argumentos invalida
```

4.2.3 Resposta no terminal

```
mari@mari:~/Desktop/Compiladores/SCC0605-Compilador_P--$ g++ lexico.cpp sintatico.cpp main.cpp -o exe
mari@mari:~/Desktop/Compiladores/SCC0605-Compilador_P--$ ./exe errados/8_input.txt

********* ANALISE LEXICA *********
Erro léxico na linha 3: caractere não permitido

********* ANALISE SINTATICA *********
Erro sintatico na linha 3: variáveis inseridas de forma inválida nesse comando
Erro sintatico na linha 3: estrutura de declaração de variável inválida
Erro sintatico na linha 4: estrutura de declaração de variável inválida
Erro sintatico na linha 5: estrutura declaração de procedimento inválida
Erro sintatico na linha 12: variáveis inseridas de forma inválida nesse comando
Erro sintatico na linha 14: comando inválido
Erro sintatico na linha 18: lista de argumentos inválida


********* COMPILAÇÃO FINALIZADA *********
- Todas as cadeias processadas
- Número de erros mapeados: 8
```

Figura 2: Resposta no terminal ao código com erros

5 Conclusão

O analisador sintático, funcionou muito próximo do esperado. Como todos os procedimentos foram mapeados, foi julgado desnecessário a construção de grafos sintáticos. A estrutura dos programas estão sendo verificadas de acordo com a gramatica P−− formulada, na seção 3.2.

Para programas corretos, o compilador funciona perfeitamente, todas as cadeias são consumidas e o arquivo de saída fica vazio. Para casos de erros, várias entradas foram testadas e para muitos casos a saída é bem intuitiva, entretanto, é válido ressaltar que para alguns casos as mensagens de erros apresentam um pouco de incoerência. Em alguns casos, erros podem provocar uma cascata de outros erros. Essa cascata não trava a compilação, mas é provável que alguns mensagens de erro sejam incoerentes.

Podemos concluir, que todo o trabalho seguiu as expectativas de operação e foram no caminho certo do aprendizado obtido na disciplina. A implementação da tabela de símbolos ajudará no desenvolvimento do analisador semântico.

Referências

1º) Repositório do trabalho: Disponível em:
`https://github.com/leandroS08/SCC0605-Compilador_P--`
Último acesso em 18 de Junho de 2020

2º) Notas de aula SCC0605- Teoria de Computação e Compiladores: Disponível em:
`https://ae4.tidia-ae.usp.br/portal/site/0dd38d17-9cb7-413c-a860-2c158b8ebaad/page/d3ff2027-a933-4750-b90e-6ba56e2f591f`

Último acesso em 18 de Junho de 2020