

UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Analizador Sintático

Alunos:

Guilherme Brunassi Nogima (9771629)

João Pedro Silva Mambrini Ruiz (9771675)

Professor:

Thiago A. S. Pardo

São Carlos, SP

2020

1 Introdução

O objetivo deste trabalho é a implementação de um analisador sintático utilizando o método descendente preditivo recursivo e o modo pânico para tratamento de erros.

A análise sintática é a segunda fase do compilador, posterior à análise léxica. Sua função é verificar se o programa entrada está de acordo com a estrutura gramatical da linguagem, e relatar ao usuário os possíveis erros sintáticos.

Neste trabalho, serão discutidas as alterações realizadas no analisador léxico e os detalhes de implementação do analisador sintático e do tratamento de erros. Por fim, serão apresentadas as instruções de uso e um exemplo de execução do programa.

Link do projeto no GitHub: <https://github.com/gbnogima/compilador>

2 Alterações no Analisador Léxico

Com o objetivo de melhorar a usabilidade e eficiência do compilador, foram realizadas uma série de alterações no analisador léxico enviado na primeira entrega, com base nos resultados obtidos. As mudanças estão listadas a seguir.

- **Especificação do erro:** Para melhor identificação do erro, foram acrescentadas informações para diferenciação de cada erro léxico. E, também, especificando o que era esperado no lugar do erro.
- **Localização do erro:** De forma a facilitar a localização do erro no arquivo, foi adicionado às mensagens de erro a sua localização exata, contendo a linha do código e a posição na linha.
- **Linha de comando:** De modo a tornar mais versátil e prática a utilização do programa, foi acrescentada uma funcionalidade que permite ao usuário especificar arquivos de entrada e saída diretamente pela linha de comando. Além disso, foi acrescentada uma flag de ajuda.

3 O Analisador Sintático

O analisador sintático neste trabalho foi implementado utilizando o método descendente preditivo recursivo. Neste método, inicia-se a leitura do programa a partir do símbolo inicial da gramática.

A primeira parte desenvolvida no analisador sintático foi a sua integração com o analisador léxico. Esta funcionalidade é executada pela função “update_token”, que atualiza, a cada chamada, os valores de “token” e “string”, as quais correspondem ao token e ao valor atual da leitura. Essas são variáveis globais, podendo ser acessadas em qualquer lugar do código.

A funcionalidade da análise sintática foi desenvolvida por meio de diversas funções, que correspondem à implementação dos não-terminais presentes na gramática do P-. Para facilitar a interpretação do código, os nomes utilizados foram os mesmos presentes na gramática.

No trecho de código abaixo, é apresentada a função “corpo”, que assim como cada função, possui seus próprios valores de “max_error” e “follow”. Ela é responsável por chamar os não terminais que realizam a leitura das declaração de variáveis, constantes e funções. Posteriormente, ela consome “begin”, e chama outra função que consome os comandos. Por fim, ela recebe “end”.

```
def corpo(program):
    max_errors = 2
    follow = [ 'simb_pf' ]

    dc_c(program)
    dc_v(program)
    dc_p(program)

    if(token == 'simb_begin'):
        update_token(program)
    else:
        handle_error(program, string, "Esperava-se 'begin'",
                      [ 'id' ], follow, max_errors)

    comandos(program)

    if(token == 'simb_end'):
        update_token(program)
    else:
        handle_error(program, string, "Esperava-se 'end'",
                      [ ';' ], follow, max_errors)
```

Para informar ao usuário a saída produzida, foi utilizado um arquivo de saída, que por padrão será “saida.txt”, mas poderá ser especificado na linha de comando. Ele conterá todos os erros identificados, tanto no analisador léxico quanto no sintático. Para cada erro é especificada a sua localização, contendo a linha e a posição do erro.

4 Tratamento de Erros

O método utilizado para o tratamento de erros sintáticos foi o Modo Pânico, que tem por característica a utilização de tokens de sincronização para retomar a análise sintática a partir de um ponto seguro, tornando possível detectar os demais erros que possam existir no restante do programa.

Sendo assim, o tratamento de erros foi implementado utilizando a função a seguir:

```

def handle_error(program, string, msg, sync=[], follow=[], max_errors=0):
    global parser_errors
    parser_errors.append(tuple((lexical.line_count,
                               lexical.i-lexical.chcnt-1, msg, string)))

    if(max_errors < 0):
        sync = follow

    if(token == 'id'):
        update_token(program)

    while(token not in sync+follow and lexical.i < len(program)):
        update_token(program)

    if(lexical.i < len(program)):
        if(max_errors < 0) or token in follow:
            return True
        else:
            return False
    else:
        print_errors()
        exit(1)

```

Os parâmetros da função são os seguintes:

- *program*: string que contém o programa;
- *string*: trecho onde foi identificado o erro;
- *msg*: mensagem de erro;
- *sync*: lista contendo os tokens de sincronização;
- *follow*: lista contendo os seguidores do pai do não terminal atual;
- *max_errors*: se for menor que 0, limite de erros foi excedido e a sincronização segue para o seguidor do pai.

A função procede da seguinte forma:

1. O erro, com sua mensagem e localização, é acrescentado à lista de erros;
2. Caso o limite de erros da leitura atual tenha excedido, os tokens de sincronização serão apenas os seguidores do pai da leitura atual;
3. É buscado um token que esteja na lista de tokens de sincronização;
4. Caso o token encontrado esteja em follow e o limite de erros não tenha excedido, retorna True, caso contrário retorna False;

5. Caso a leitura chegue ao fim do programa sem encontrar um token de sincronização, a execução é encerrada.

Nesta etapa do desenvolvimento, foram tomadas algumas decisões para que a retomada da leitura fosse eficiente. Para isso, é importante que sejam utilizados os tokens de sincronização adequados. Sendo assim, a estratégia geral utilizada foi, para cada erro em um símbolo A que é analisado, são utilizados os seguidores de A e os seguidores do pai de A como tokens de sincronização. Em alguns casos, são acrescentados, ainda, tokens extras, que impedirão, em caso de erro, que uma grande cadeia seja desconsiderada na análise.

Outro detalhe considerado nesta etapa foi a primeira atualização de token. Em alguns casos, é interessante considerar o token atual para encontrar um ponto de sincronização, por exemplo, em caso de algum trecho ausente. No entanto, isto pode se tornar um problema, por exemplo, no caso de um erro em uma palavra reservada, ela é interpretada como identificador, que muitas vezes é um seguidor da palavra reservada. Para evitar este tipo de situação, foi implementada uma condição, que atualiza, obrigatoriamente, o token antes de buscar pelo ponto de sincronização.

5 Instruções de Uso

Para execução do programa, basta possuir o Python3 instalado. Para rodar o código na linha de comando, utiliza-se:

```
python3 analisador_sintatico.py -i <programa_entrada> [-o <arquivo_saida>]
```

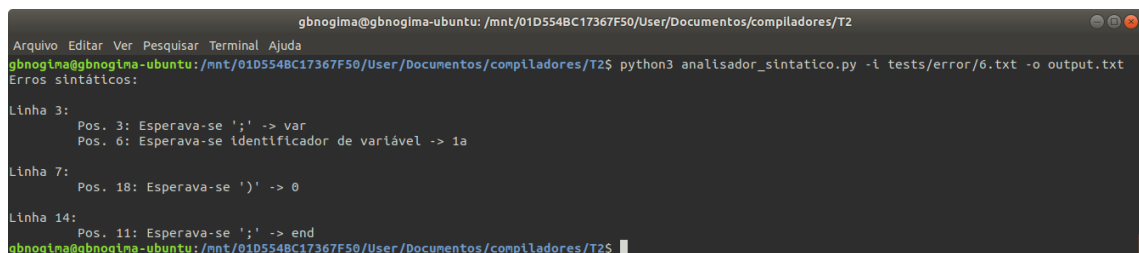
Se não especificado, o arquivo saída será 'saida.txt'.

6 Exemplo de Execução

Para exemplificar a execução do programa será utilizado o programa abaixo, que contém 3 erros sintáticos e 1 erro léxico.

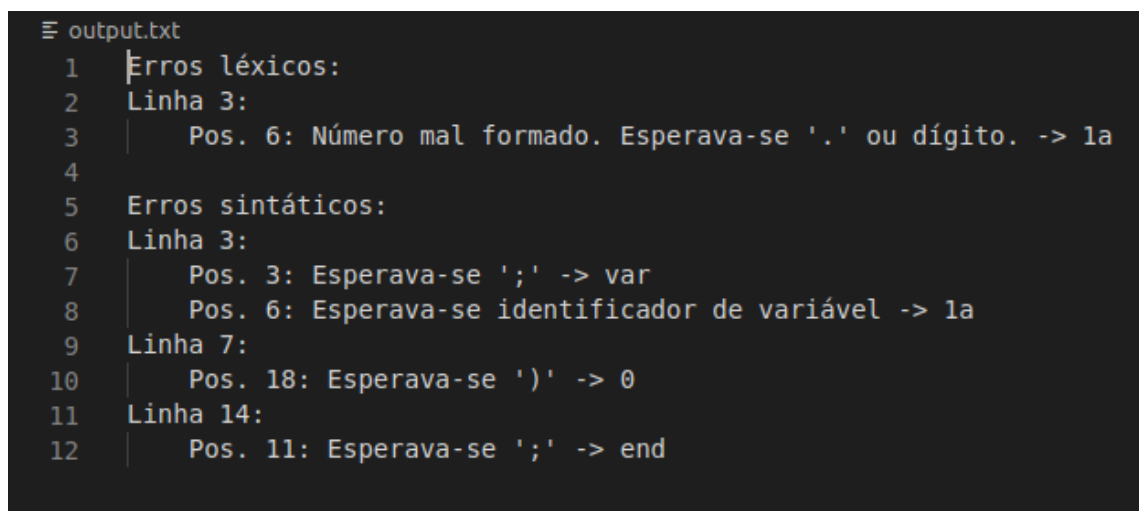
```
program foo
var 1a, b, c : integer;
begin
  c := 0;
  a := 2;
  while(a <= 9 0 do
  begin
    b := 1;
    while(b <= 9) do
    begin
      c := c+(a*b);
      b := b+1
    end;
    a := a+1;
  end;
end.
```

O terminal terá a seguinte saída:



```
gbnogima@gbnogima-ubuntu: /mnt/01D554BC17367F50/User/Documentos/compiladores/T2
Arquivo Editar Ver Pesquisar Terminal Ajuda
gbnogima@gbnogima-ubuntu: /mnt/01D554BC17367F50/User/Documentos/compiladores/T2$ python3 analisador_sintatico.py -i tests/error/6.txt -o output.txt
Erros sintáticos:
Linha 3:
  Pos. 3: Esperava-se ';' -> var
  Pos. 6: Esperava-se identificador de variável -> 1a
Linha 7:
  Pos. 18: Esperava-se ')' -> 0
Linha 14:
  Pos. 11: Esperava-se ';' -> end
gbnogima@gbnogima-ubuntu: /mnt/01D554BC17367F50/User/Documentos/compiladores/T2$
```

E o arquivo de saída será:



```
output.txt
1 Erros léxicos:
2 Linha 3:
3 | Pos. 6: Número mal formado. Esperava-se '.' ou dígito. -> 1a
4
5 Erros sintáticos:
6 Linha 3:
7 | Pos. 3: Esperava-se ';' -> var
8 | Pos. 6: Esperava-se identificador de variável -> 1a
9 Linha 7:
10 | Pos. 18: Esperava-se ')' -> 0
11 Linha 14:
12 | Pos. 11: Esperava-se ';' -> end
```

7 Conclusão

A etapa de análise sintática é uma etapa crucial da compilação, uma vez que determina se o programa é válido para a gramática da linguagem, ou seja, o analisador sintático determina se programa irá gerar um código interpretável.

A implementação de um analisador sintático é uma tarefa extensa, pois é necessário considerar diversas peculiaridades da linguagem. Além disso, o tratamento de erros exige atenção em cada uma das possibilidades, para que haja maior eficiência nas sincronizações.

Por fim, pode-se concluir que a eficiência de um analisador sintático depende muito de uma boa estruturação da linguagem, e, principalmente, das decisões do projetista.