Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação

Trabalho 02 - Analisador Sintático

SCC0605 Teoria da Computação e Compiladores
Prof. Dr. Thiago A. S. Pardo

Patrick Oliveira Feitosa 10276682
Pedro Henrique Rampim Natali 10310655
Rafael Farias Pinho 10276737

São Carlos, São Paulo 18/05/2020

Sumário

Introdução

Nesta etapa iremos incrementar o Trabalho 1, no qual desenvolvemos um analisador léxico. Agora implementamos o analisador sintático do compilador da linguagem P--. Após essa etapa, o programa deverá ler o código-fonte e retornar se a sintaxe dele está correta, se não os erros que ele apresentou.

Além disso, foi feita a correção do código do analisador léxico, que foi entregue no Trabalho 1, a partir dos seis casos de teste fornecidos na plataforma tidia para verificação de erros léxicos. As seguintes mudanças foram feitas no analisador léxico:

- Foram adicionados alguns símbolos na lista de palavras reservadas como simb_to para o casa da palavra reservada to e simb_const para constante.
- Foi consertado um erro causado ao se ler a função write pelo analisador léxico.
- Foi adicionado a possibilidade de ler e analisar a função procedure que não havia sido implementada.
- Foi consertado o erro causado quando o analisador lia o símbolo de maior que, o que gerava um problema de execução no código.

Grafos

Para facilitar o entendimento, foram desenvolvido os grafos das funções implementadas tendo como referência a gramática da linguagem. Com elas será possível entender tanto as funções são estruturadas quanto analisar a representação visual dos grafos de <*program*>, <*dc_c*>, <*dc_v*>, <*dc_p*>, <*corpo_p*>, <*cmd*>, <*condicao*>, <*expressao*>, <*termo*> e <*fator*>.

<program>

O grafo considerado como a estrutura do sintático é o que definimos como program, ele é responsável por verificar se foi iniciado o token program com seu id, após isso, verifica todas as declarações possíveis (variável, constante ou procedure). Com as verificações feitas, começa a estrutura do programa principal com begin e executa as funções do <cmd> até encontrar o fim do programa. Sua estrutura pode ser vista visualmente na Figura 1:

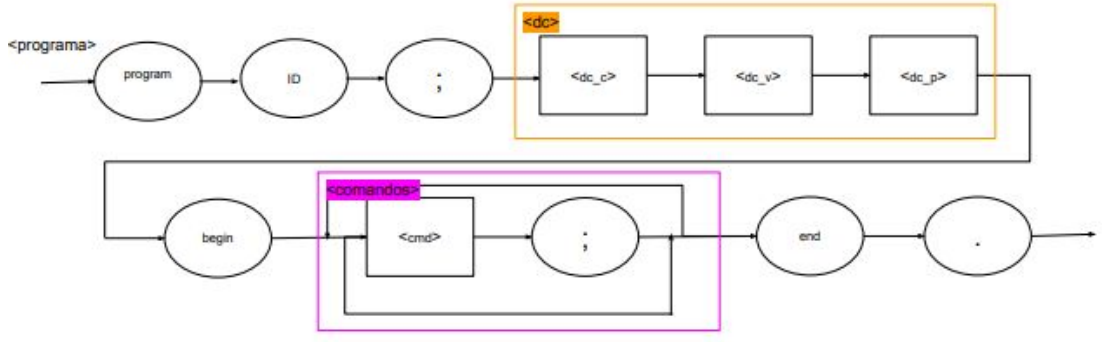

Figura 1: Representação do grafo <program>

<dc_c>

A declaração de uma constante é feita analisando o token simb_const, espera-se seguinte uma igualdade, seu tipo e o simb_pv. É possível após o simb ocorrer outra declaração de constante, visto que podem ser declaradas em mais de uma linha seguintes. Nota-se que quando não encontra-se algum dos tokens esperados, um erro é escrito. Sua estrutura está presente na Figura 2.

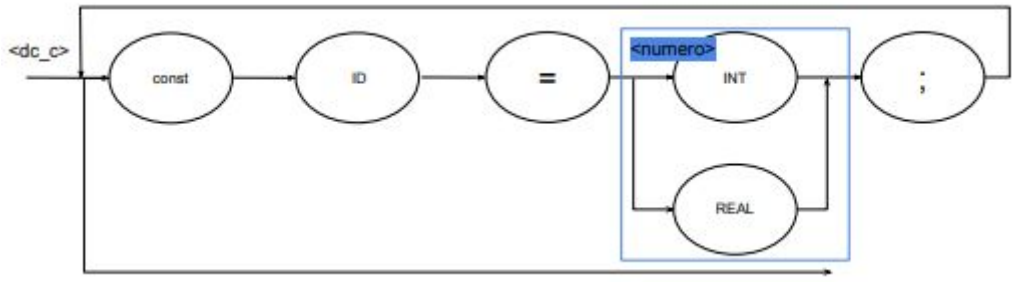

Figura 2: Representação do grafo <dc_c>

<dc_v>

Semelhante ao <dc_c>, o grafo para declaração de variáveis espera o token correspondente à uma variável, analisa seu id (ou seus id's - separados por vírgula), espera um tipo correspondente às variáveis e simb_pv para finalizar. No final, pode ocorrer outra declaração de variáveis, ou sai da função. Sua representação é apresentada na Figura 3.

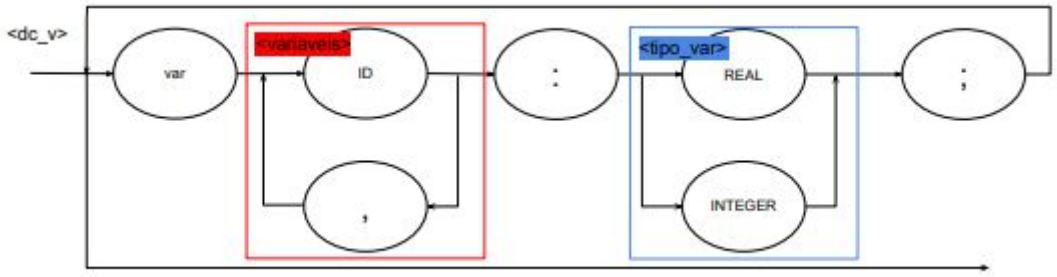

Figura 3: Representação do grafo <dc_v>

<dc_p>

Um procedure é uma função, então a estrutura apresentada aqui é semelhante ao <program>, onde variáveis podem ser declaradas e funções são executadas à partir do <cmd>. A diferença é que seu end termina em simb_pv e não simb_p como é feito na <program>

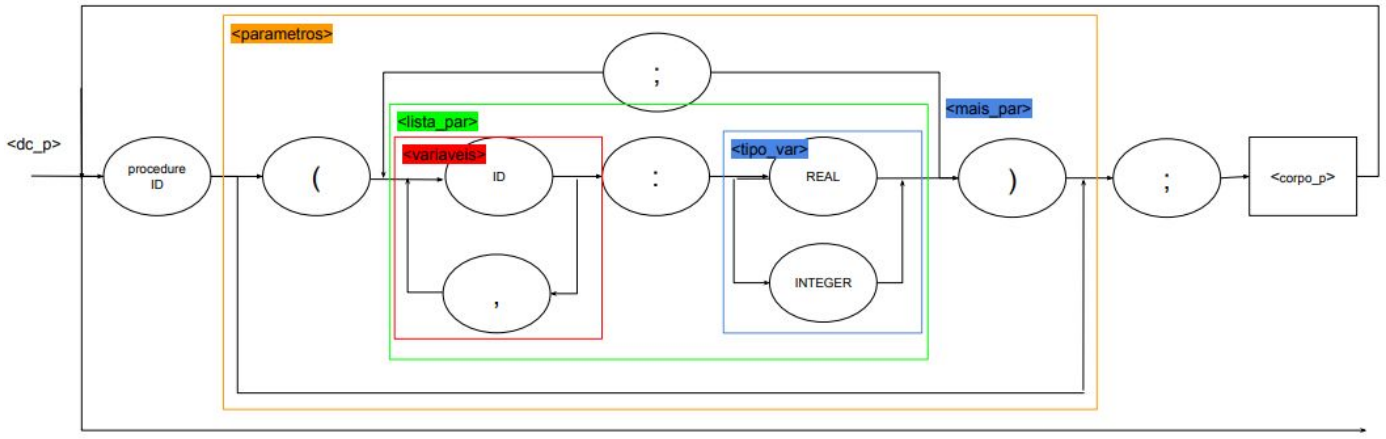

Figura 4: Representação do grafo <dc_p>

<corpo_p>

Essa função tem o intuito de gerar o corpo do programa após a variáveis e funções terem sido inicializadas.

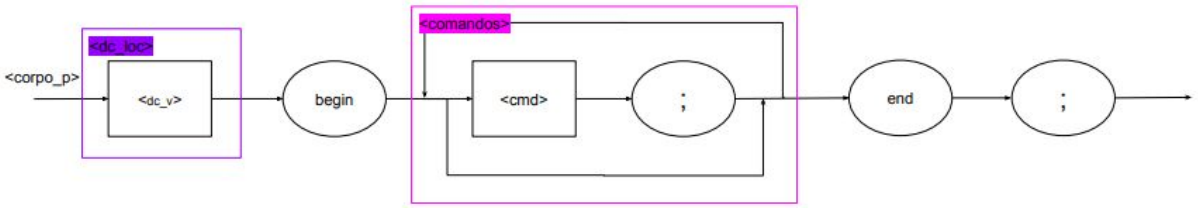

Figura 5: Representação do grafo da <*corpo_p*>.

<cmd>

Essa função irá ler todos os comandos do código, os quais podem ser de read, write, while, if id ou begin. essa função é chamada pelo corpo_p, onde irá ler o primeiro token do grafo e então verificar para onde prosseguir. Tendo lido o primeiro token o grafo continuará para um dos 6 comandos diferentes possíveis e seguir o grafo até o final.

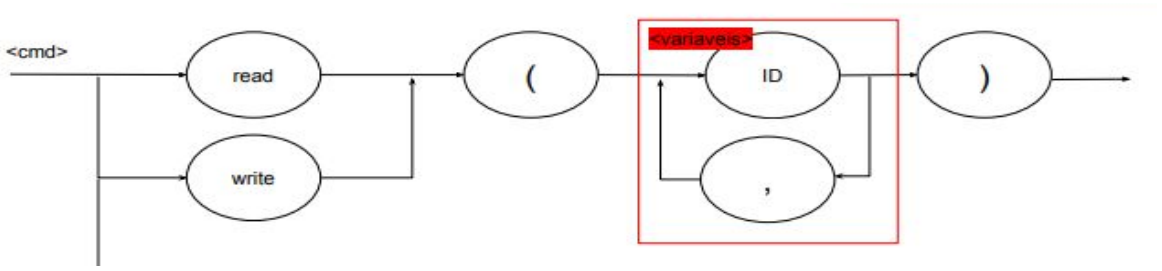

Figura 6: Representação do grafo da leitura e escrita de uma variável da <cmd>.

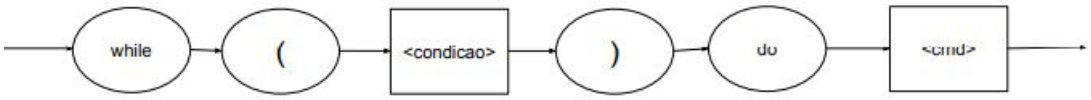

Figura 7: Representação do grafo da estrutura de um *while* da <cmd>.

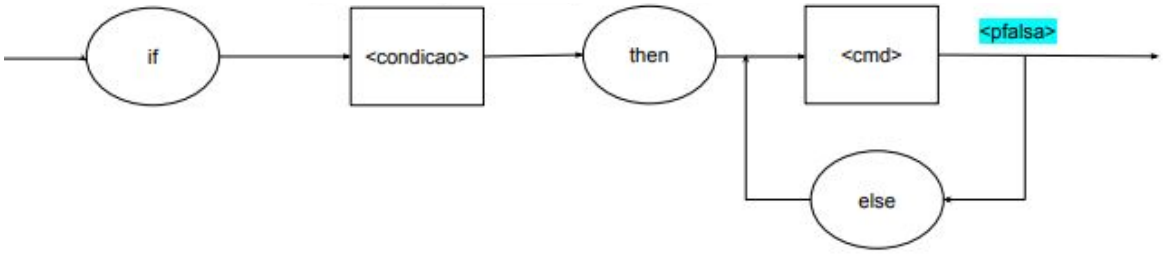

Figura 8: Representação do grafo da estrutura de um *if* da <cmd>.

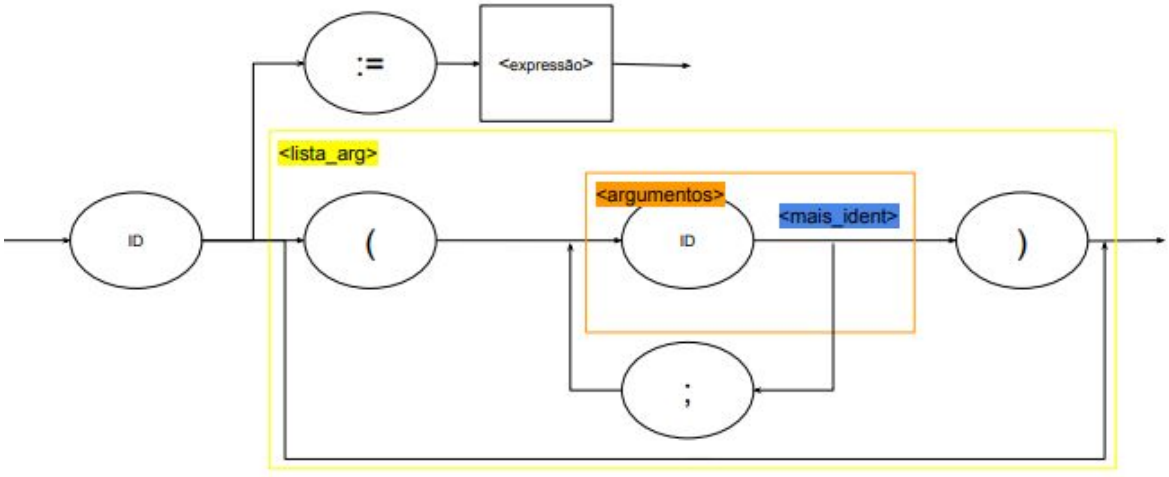

Figura 9: Representação do grafo da estrutura de atribuicao ou chamada de funcao na <cmd>.

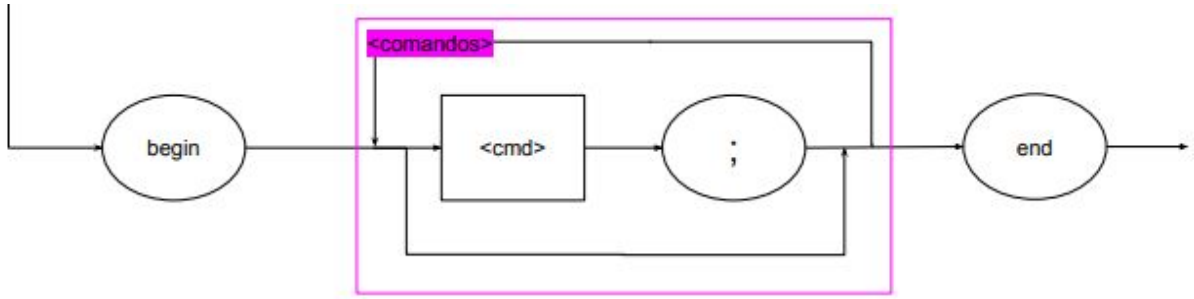

Figura 10: Representação do grafo da estutura de um *begin* da <cmd>.

<condicao>

Essa função tem o intuito de verificar se a sintaxe de uma condição obdece a linguagem. Ela é dividida em duas expressões intermediada por uma relação. As relações podem ser de igualdade, diferente, maior ou igual, menor e igual, maior ou menor.

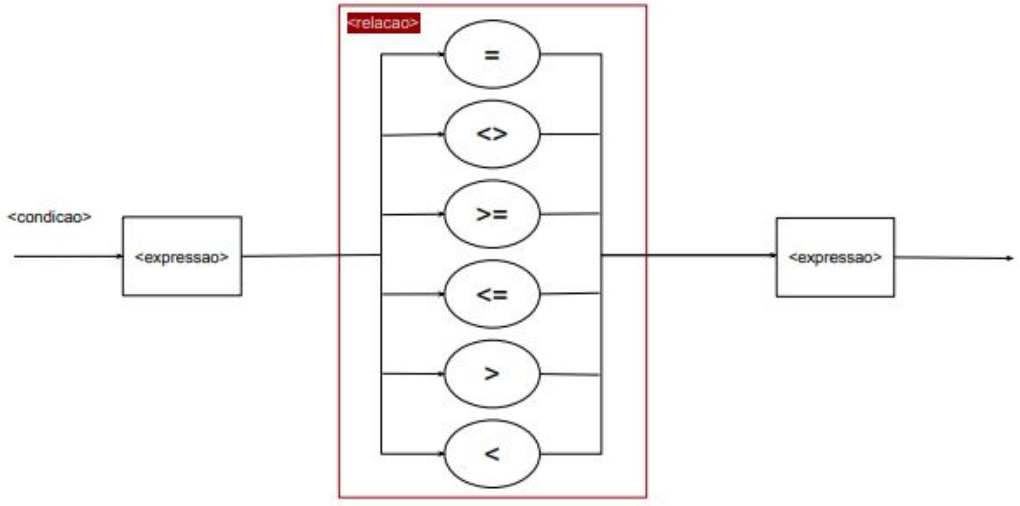

Figura 11: Representação do grafo da leitura de uma condição.

<expressao>

Cada expressão é dada pela soma ou subtração de termos, como se pode ver abaixo:

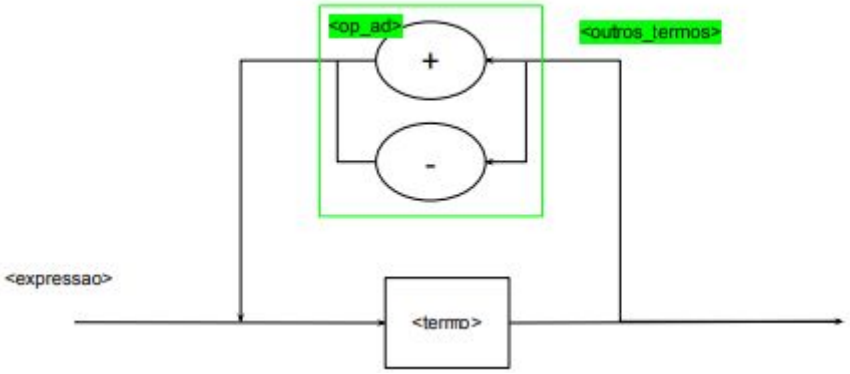

Figura 12: Representação do grafo da leitura de uma expressão.

<termo>

Já o termo é formado por uma multiplicação ou divisão de fatores, podendo ser acrescido de um sinal.

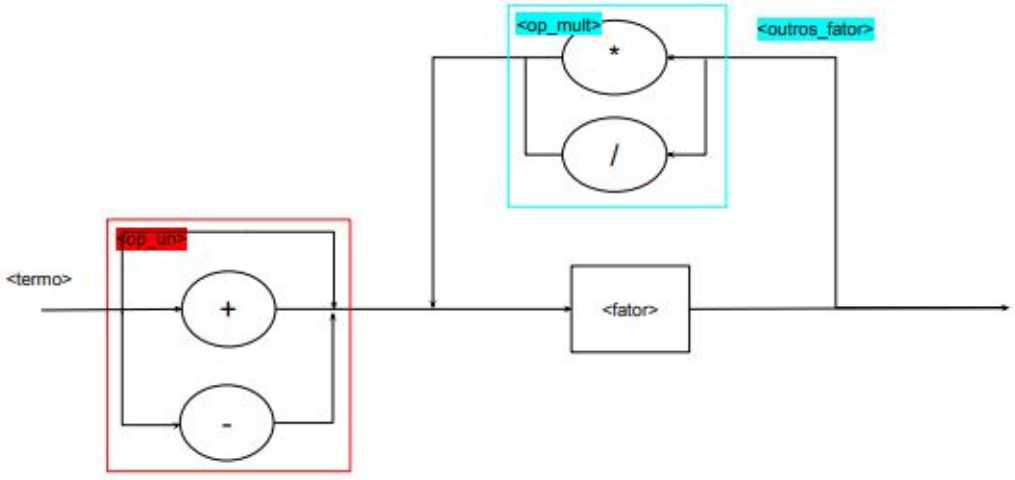

Figura 13: Representação do grafo da leitura de um termo.

<fator>

Cada fator podem ter as seguintes estruturas:

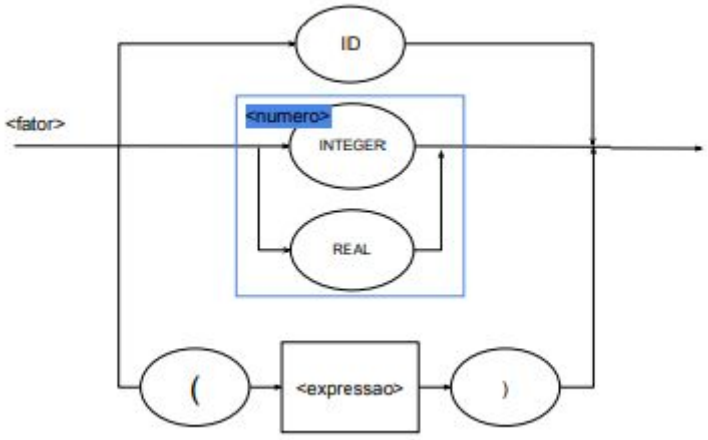

Figura 14: Representação do grafo da leitura de um fator.

Decisões de Projeto

- Grande parte das decisões de projeto foram tomadas acerca das criações de grafos e mudanças no léxico. Então ambas serão descritas.
    - **Criação de grafos:**
        - O grupo optou por desenvolver toda a estrutura das regras à partir de grafos de maneira mais simplificada possível para a implementação e correção de erros. Sendo assim, vários módulos (ou funções) foram criados para corresponder aos grafos descritos neste relatório.
    - **Mudanças no léxico:**
        - Para conseguir implementar o analisador sintático, foi necessário criar uma nova função chamada nextToken no léxico que funciona parecido como foi implementado para o trabalho passado só que antes era retornado uma tabela com todos os tokens do código e agora é retornado um token por vez, além da posição que está sendo lido, a linha que está sendo lida no código e a tabela com o último token adicionado.
        - Foi adicionado a verificação de linhas do código no analisador léxico que serão enviadas para o sintático sempre que for pedido um novo token para poder fazer a verificação de onde está o erro no código.
        - Foram adicionadas mudanças no autômato de comentaria para que ele pule as linhas quando houver erro de comentário de várias linhas, dessa forma erros no resto do código serão indicados nas linhas corretas.
        - Agora, erros comunicados pelo analisador léxico são adicionados a tabela de tokens como sendo um token de erro e pulados pelo

sintático para depois serem impressos no arquivo de saída, diferente de como era feito anteriormente que já eram impressos pelo próprio analisador.

- Foi adicionada também uma variável de controle nas funções dos autômatos, para garantir que a cada chamada de função nextToken só entre em um autômato, de forma que a tabela com todos os tokens só seja adicionado um por vez. Impedindo que sejam algum token não seja lido pelo sintático.

Execução

A seguir iremos demonstrar as especificações do sistema onde a aplicação foi desenvolvida e demonstrar um exemplo da mesma para execução.

Especificações

Este programa foi desenvolvido em uma máquina com as seguintes configurações:

Desktop: **Unity 7.4.5** Distro: **Ubuntu 16.04** xenial Dual core **Intel Core i7**-7500U

E é necessário a presença de **Python3** para execução. Caso não seja o caso, execute o seguinte comando abaixo para instalação do mesmo.

```
sudo apt-get install python3
```

Para executar o código desenvolvido pelo grupo, é necessário que na pasta do mesmo tenha um arquivo nomeado 'programa.txt'. Na pasta anexa está presente um exemplo (com resposta esperada para comparação), porém é possível alterar o mesmo que a execução segue normalmente.

No terminal, deve-se entrar no diretório da pasta (exemplo para Linux):

```
cd ~/SCC0605-Compiladores
```

Então deve-se executar o programa da maneira descrita a seguir:

```
python3 main.py
```

O programa então irá executar sobre o documento 'programa.txt' e irá devolver uma resposta 'saida.txt'. Nota-se que tanto a entrada quanto a saída estão em pastas separadas intituladas [entrada] e [saída] para melhor organização.

Exemplo de execução

O exemplo a seguir está presente em entrada/programa.txt, além do exemplo em questão, foi também elaborado uma entrada alternativa na mesma pasta com o nome programa2.txt

program meuprograma;
const b = int;
var a@: integer;
procedure nomep(x:real);
var e,z: integer;
begin
read(b);
write(c);
for b:=1 to 10 do
begin
b:=b+2;
a:=a-1;
;
end
begin
read($a,b);
while (b > a do
begin
write(a);
a := a + 1;
end;
end.
	Erro léxico na linha 3: caractere @ inválido em id
	Erro sintático na linha 13: end esperado
	Erro sintático na linha 14: ; esperado
	Erro léxico na linha 16: caractere inválido
	Erro sintático na linha 17:) esperado

programa.txt	**saida.txt**