Trabalho 2: Analisador Sintático

SSC0605 - Teoria da Computação e Compiladores

Prof. Dr. Thiago Alexandre S. Pardo

Alunos:

Che Fan Pan	11200421
Eduardo Cavalari Valença	11234381
Marcos Vinicius Firmino Pietrucci	10770072
Murilo Mussatto	11234245

Engenharia da Computação
São Carlos - SP

2022

1 - Relato de decisões de projeto

O nosso projeto de analisador sintático de P-- foi implementado utilizando como base o nosso código de analisador léxico. O trabalho anterior gerava um vetor com todos os tokens lidos incluindo todos os erros léxicos encontrados (caracteres desconhecidos, comentários não fechados e etc).

De posse desse código, nossa implementação do analisador sintático foi construída separadamente. Decidimos por implementar todas as regras da linguagem disponibilizadas pelo professor, além de modificar o terminal **ident** da regra **<cmd>** para transformar em uma LL(1). Ademais, foi adicionado o comando **for** à regra **<cmd>** na forma:

```
<cmd> ::= for ident := <expressao> to <numero> do <cmd>
```

Cada regra (em linhas gerais) contém sua própria função, consumindo os caracteres e verificando a conformidade dos vetores lidos com a regra.

Caso um erro nestas regras for encontrado, decidimos por criar um vetor de "erros do sintático", o qual armazena a mensagem de erro e a linha na qual este ocorreu. Então, a decisão tomada no projeto foi encontrar o seguidor do token em que houve o erro para retomar a análise do compilador. Esse método é o chamado Modo Pânico.

Quando o programa lido atinge o EOF, é iniciado o processo de escrita dos erros em um arquivo de erro chamado "error_file.txt". Neste método, o programa inclui não só os erros do sintático, como também os erros do léxico (mapeados no programa do trabalho 1) e, para garantir que a impressão siga a ordem das linhas, é feito um *sort* com todos os erros (sintáticos e léxicos).

2 - Instruções de compilação e execução

O procedimento de compilação e execução é simples. É necessário ter os arquivos em uma máquina Linux ou Windows com WSL com GCC e make instalados. Junto ao código, enviamos um makefile para que o código seja compilado e executado.

Para compilar o código basta iniciar um terminal no diretório do trabalho e executar o comando:

```
~$ make
```

Nesse momento, todos os arquivos de compilação serão gerados. Configuramos o makefile para exibir saída apenas em caso de erro de compilação, se nenhuma mensagem aparecer significa que tudo ocorreu bem.

Feito isto, a execução é feita pelo comando:

```
~$ make run
```

O terminal pedirá o nome do arquivo de texto com o programa a ser analisado.

```
~$ make run
~$ Insira o nome do arquivo (com extensao):
```

Após a inserção, a execução começará normalmente e será gerado o arquivo "output.txt", contendo todos os tokens lidos, suas classes. Além disso, será criado o arquivo "error_file.txt", o qual contém tanto os erros do analisador léxico quanto os erros do analisador sintático, ordenados por linha.

3 - Modo pânico

A implementação do modo pânico no trabalho foi feito da seguinte forma: caso um erro sintático seja encontrado, os tokens seguintes ao token que houve erro serão consumidos até que se encontre o seguidor do token errado. Um exemplo seria a seguinte linha de código escrita de modo incorreto para a declaração do identificador:

```
const #x = 4;
```

A regra de declaração de variável é descrita abaixo:

```
<dc_c> ::= const ident = <numero> ; <dc_c> | λ
```

Como a declaração do identificador foi feita de maneira errada (`#x`), o compilador então identifica este token e o adiciona no vetor de erros sintáticos. Após isso, o algoritmo irá pular todos os tokens seguintes até encontrar o seguidor de `ident`, que no caso é o `=`, e a partir deste token é retomado a análise do compilador. Para o exemplo, após ter encontrado o seguidor é verificado se um número foi atribuído ao `=` e se há o `;` no final da declaração da variável.

Seguindo esse modo de operação, foi encontrado os seguidores para cada uma das regras da linguagem P–, representados na tabela 1.

Tabela 1 - Seguidor(es) de cada regra da linguagem P–.

Regra	**Seguidor**
programa	λ
corpo	.
dc	begin
dc_c	begin, var, procedure
dc_v	begin, procedure
tipo_var	;,)

variaveis	:,)
mais_var	:,)
dc_p	begin
parametros	;
lista_par	)
mais_par	)
corpo_p	begin, procedure
dc_loc	begin
lista_arg	;
argumentos	)
mais_ident	)
pfalsa	;
comandos	end
cmd	;
pos_indet	;
condicao	), then
relacao	+, -, ident, (, numero_int, numero_real
expressao	;, =, <>, >=, <=, >, <,), then, to, do
op_un	ident, (, numero_int, numero_real
outros_termos	;, =, <>, >=, <=, >, <,), then, to, do
op_ad	+, -, ident, (, numero_int, numero_real
termo	+, -
mais_fatores	+, -
op_mul	ident, (, numero_int, numero_real
fator	*, /
numero	;, *, /

4 - Exemplo de execução

Considere por exemplo o seguinte programa em P–:

meu_programa.txt

```
program p_;
var x: integer;
begin
    x:=-1;
    while (x<3) do
            x:=x+1;
end.
```

Este fragmento não contém erros, ou seja, é esperado que o analisador léxico e sintático não apontem erros. Ao executarmos o programa, conforme descrito na seção anterior, obtemos a seguinte saída.

error_file.txt

```
Compilation successful.
I can finally rest and watch the sun rise on a grateful universe.
```

Agora vamos adicionar alguns erros ao programa previamente apresentado. O novo arquivo de teste está disposto a seguir:

meu_programa.txt

```
program p_;
var x: integer;
begin
      x:= ;
      @while (x<3) do
            x:=x+1
end.
```

Este fragmento contém erros propositais, como a presença de símbolos inválidos (@, linha 5), a falta de um termo na atribuição (linha 4) a falta de um ponto e vírgula esperado (; , linha 6). Ao executarmos o programa, conforme descrito na seção anterior, obtemos a seguinte saída.

error_file.txt

```
Syntax Error: Unrecognized term, 4
Lexical Error: Invalid Character, 5
Syntax Error: Missing expected semicolon, 7
```

Ao analisarmos o resultado, podemos concluir que nosso programa funciona com sucesso. Conseguimos detectar ambos os tipos de erros, diferenciá-los, e ainda ordená-los conforme sua linha de aparição.