Trabalho 01 - Compiladores

Bruno Rafael dos Santos, Gabriel Anselmo Ramos

15 de Maio de 2022

1 Definição da Gramática

Na forma de Backus-Naur temos a gramática descrita da seguinte forma:

```
<atribuição> ::= <variável> ":=" <expressão>
<bool-lit> ::= true | false
<chamada-de-função> ::= <id> "(" <lista-de-expressões> | <vazio> ")"
<chamada-de-procedimento> ::= <id> "(" <lista-de-expressões> | <vazio> ")"
<comando> ::= <atribuição> | <condicional> | <iterativo> | <chamada-de-procedimento> |
<comando-composto>
<comando-composto> ::= begin <lista-de-comandos> end
<condicional> ::= if <expressão> then <comando> ( else <comando> | <vazio> )
<corpo> ::= <declarações> <comando-composto>
<declaração> ::= <declaração-de-variável> | <declaração-de-função> | <declaração-de-
procedimento>
<declaração-de-função> ::= function <id> "(" <lista-de-parâmetros> | <vazio> ")" : <tipo-
simples> ; <corpo>
<declaração-de-procedimento> ::= procedure <id> "(" <lista-de-parâmetros> | <vazio> ")" ;
<corpo>
<declaração-de-variável> ::= var <lista-de-ids> : <tipo>
<declarações> ::= <declaração> ; | <declarações> <declaração> ; | <vazio>
<digito> ::= 0 | 1 | 2 | ... | 9
<expressão> ::= <expressão-simples> | <expressão-simples> <op-rel> <expressão-simples>
<expressão-simples> ::= <expressão-simples> <op-ad> <termo> | <termo>
<fator> ::= <variável> | <literal> | "(" <expressão> ")" | <chamada-de-função>
<float-lit> ::= <int-lit> . <int-lit> | <int-lit> . | . <int-lit>
<id> ::= <letra> | <id> <letra> | <id> <digito>
<int-lit> ::= <digito> | <int-lit> <digito>
<iterativo> ::= while <expressão> do <comando>
<letra> ::= a | b | c | ... | z
<lista-de-comandos> ::= <comando> ; | <lista-de-comandos> <comando> ; | <vazio>
<lista-de-expressões> ::= <lista-de-expressões> , <expressão> | <expressão>
<lista-de-ids> ::= <id> | <lista-de-ids> , <id>
<lista-de-parâmetros> ::= <parâmetros> | <lista-de-parâmetros> ; <parâmetros>
<literal> ::= <bool-lit> | <int-lit> | <float-lit>
<op-ad> ::= + | - | or
<op-mul> ::= * | / | and
<op-rel> ::= < | > | <= | >= | = | <>
<outros> ::= ! | @ | # | ...
<parâmetros> ::= ( var | <vazio> ) <lista-de-ids> : <tipo-simples>
<programa> ::= program <id> ; <corpo> .
<seletor> ::= <seletor> "[" <expressão> "]" | "[" <expressão> "]" | <vazio>
<termo> ::= <termo> <op-mul> <fator> | <fator>
<tipo> ::= <tipo-agregado> | <tipo-simples>
<tipo-agregado> ::= array ( "[" <literal> .. <literal> "]" ) of <tipo>
<tipo-simples> ::= integer | real | boolean
<variável> ::= <id> <seletor>
<vazio> ::= ε
```

Figure 1: Gramática na forma de Backus-Nor

2 Especificação Flex

2.1 Conjuntos de símbolos

Na especificação do arquivo flex temos na sessão de definições temos os seguintes conjuntos de símbolos descritos:

```
DIGITO  ([0-9])
BOOL-LIT "true"|"false"
OP-AD "+"|"-"|"or"
OP-MUL  "*"|"/"|"and"
OP-REL "<"|">"|"<="|">="|"="|"<>"
VAZIO ""
OUTROS "!"|"@"|"#"|"$"|"%"|"¨"|"&"|"?"|"."|"'"|"{"|"}"
TIPO "integer"|"real"|"boolean"
LETRA[a-z]
KEY-WORD "if"|"else"|"then"|"begin"|"end"|"function"|";"|":"|"while"|"do"|
          ","|"array"|"["|"]"|"var"|"procedure"|"of"|"("|")"|":="
```

2.2 Tokens

Para obtenção dos tokens temos as seguinte expressões regulares (seguidas de suas respectivas instruções entre "{}", quais serão explicadas na sessão de comentários sobre a implementação):

Token	
E.R.	**Tipo**
{OP-AD}	opAd
{OP-MUL}	opMul
{OP-REL}	opRel
{OUTRO}	outro
{TIPO}	tipo
{VAZIO}	vazio
{DIGITO}+	intlit
{DIGITO}+"."{DIGITO}*|{DIGITO}*"."{DIGITO}+	floatlit
{KEY-WORD}	keyWord
{LETRA}+({DIGITO}|{LETRA})*	ID
.	naoRec

Obs.: apesar de id, int-lit e float-lit não serem símbolos terminais para a gramática podemos reconhecê-los como tokens através do analisador léxico, o que pode então facilitar o trabalho do analisar do sintático.

3 Comentários sobre a implementação

No presente trabalho além da utilização das funcionalidades padrões da ferramenta "Flex" há a utilização de variáveis globais, estruturas e funções para montagem de uma tabela de símbolos que agregue com informações úteis para a análise sintática, detecção de erros e localização dos mesmos.

3.1 Estruturas

3.1.1 Table

A estrutura *Table* tem como objetivo permitir a criação de uma lista encadeada (que neste código segue a política de uma fila quanto a remoção e inserção de elementos), tendo em cada nó (elemento) as seguintes variáveis:

1. **token:** armazena a string do token lido.
2. **type:** armazena o tipo (ou escopo) do token lido de acordo com o que foi definido para cada.
3. **lenght:** armazena o tamanho da string do token lido.
4. **line:** armazena a linha da qual o token foi lido.
5. **column:** armazena a coluna inicial a partir de onde o token foi lido.
6. **next:** aponta para o próximo nó da lista encadeada.

3.1.2 HeadTable

A estrutura *HeadTable* tem objetivo de servir apenas como "cabeça" e "cauda" da lista, visando o fato de que desejamos ao imprimir a tabela de símbolos ler a lista a partir do primeiro token armazenado na mesma (motivação da "cabeça") mas também queremos realizar a inserção do símbolos em complexidade temporal $O(1)$ utilizando um ponteiro que leva direto ao final da lista (motivação da "cauda"). Sendo assim, nesta estrutura temos as seguintes variáveis:

1. **first:** ponteiro para a "cabeça" da lista.
2. **last:** ponteiro para a "cauda" da lista.

3.2 Variáveis globais

As variáveis globais definidas no código são:

1. **num_line:** indica a linha atual sendo lida.
2. **num_column:** indica a coluna atual sendo lida.
3. **fila:** ponteiro para uma estrutura *Head* usado para referenciar a fila usada na tabela de símbolos.
4. **numberOfTokens:** contador do número de tokens.

3.3 Funções

3.3.1 Main

Para imprimir a tabela de símbolos tem-se na função *main* um for loop que utiliza como critério de parada a variável **numberOfTokens** e imprime as linhas da tabela de símbolo navegando pela lista a partir da "cabeça". Além de imprimir a tabela de símbolos no terminal, esta função cria um arquivo txt de nome "resultado.txt" com a tabela de símbolos; caso na pasta do código já haja um arquivo de mesmo nome este será apagado ao ser gerado um arquivo novo.

3.3.2 AddToken

Em todo momento que algum token é obtido, a string do mesmo é enviada para a função *addToken* que irá criar um nó para o novo token (preenchendo as variáveis **token**, **lenght**, **line** e **column**) e adicionar tal token na fila.

3.3.3 SetType

Para fins de melhor edição e leitura do código, foi definida a função *setType* para preencher a variável **type** do último token adicionado na fila. Tal função tem como argumento uma string (qual irá preencher a variável **type**).

3.3.4 Observação

Após toda definição de token há um comando *addToken* que recebe a string **yytext** (string do token obtido) e um comando *setType* que recebe o nome do tipo do token.

4 Compilação, ambiente e linguagem

O código flex foi compilado em um sistema operacional Windows 11 Pro 64 bits utilizando a versão 2.5.4 da ferramente Flex, o editor de código VS Code e a coleção de compiladores gcc versão (MinGW.org GCC Build-2) 9.2.0.

5 Análise do exemplo proposto

5.1 Análise léxica do exemplo

Quanto a esta parte da análise, foram encontrados diversos erros (caracteres não reconhecidos), mas vale destacar que maior parte destes é ocasionada pela presença de letras maiúsculas pois estas não foram incluídas na gramática proposta. Visto isso deve-se considerar a simples adição de tais símbolos terminais à gramática para se obter uma linguagem mais abrangente. Além disso deve-se pontuar também que não há como fazer comentário na linguagem definida a partir da gramática proposta, por isso também deve-se considerar a adição de alguma regra que possibilite isto. A tabela de símbolos resultante da leitura do arquivo exemplo.txt pode ser vista neste link.

5.2 Análise do exemplo quanto à gramática

Nesta parte se encontraram muito mais erros, quais estão destacados a seguir (supondo que o símbolo inicial da gramática seja $< programa >$):

1. O código não inicia com "program" seguido de um id e não finaliza com ".".

2. Nas primeiras linhas existem declarações de variáveis incorretas e após tais declarações não há um "begin" como descrito pela derivação a partir de $< comandocomposto >$.

3. As declarações de funções não tem o token "function" antes do nome (id) e o tipo esta declarado antes de qualquer outro token, mas deveria vir após ":", sendo seguido de ";" e então o corpo da função.

4. O comando "print" não existe na gramática, logo, por exemplo, em:

   ```
   print vetor[80]
   ```

   há dois id's seguidos não separados por qualquer símbolo (o que não pode ocorrer segundo as regras desta gramática).

5. O comando "for" não existe na gramática, logo, por exemplo, em:

   ```
   for (g: 1, 10, 1):
   ```

   há uma chamada de um procedimento inexistente seguida de ":" (o que não ocorre pelas regras de $< chamada - de - procedimento >$).

6. Há token "while" com ":" ao invés de "do" após os parênteses.

7. Há token "if" com ":" ao invés de "then" após a expressão.

8. O comando "return" não existe na gramática, logo, em:

   ```
   return x+n
   ```

   há um id seguido por uma expressão, algo que não pode ocorrer segundo as regras da gramática.