TR: Primeira Etapa - Análise Léxica

Júlia Llorente e Vinícios Bidin Santos

1. Definição da Gramática

A seguir, temos a definição da gramática proposta na forma de Backus-Naur:

programa → *declaração-lista*
declaração-lista → *declaração-lista declaração* | *declaração*
declaração → *var-declaração* | *fun-declaração*
var-declaração → *tipo-especificador* **ID** **;** | *tipo-especificador* **ID** **[** **NUM** **]** **;**
tipo-especificador → **int** | **void**
fun-declaração → *tipo-especificador* **ID** **(** *params* **)** *composto-decl*
params → *param-lista* | **void**
param-lista → *param-lista* **,** *param* | *param*
param → *tipo-especificador* **ID** | *tipo-especificador* **ID** **[]**
composto-decl → **{** *local-declarações statement-lista* **}**
local-declarações → *local-declarações var-declaração* | *vazio*
statement-lista → *statement-lista statement* | *vazio*
statement → *expressão-decl* | *composto-decl* | *seleção-decl*
| *iteração-decl* | *retorno-decl*
expressão-decl → *expressão* **;** | **;**
seleção-decl → **if** **(** *expressão* **)** *statement*
| **if** **(** *expressão* **)** *statement* **else** *statement*
iteração-decl → **while** **(** *expressão* **)** *statement*
retorno-decl → **return** **;** | **return** *expressão* **;**
expressão → *var* **=** *expressão* | *simples-expressão*
var → **ID** | **ID** **[** *expressão* **]**
simples-expressão → *soma-expressão relacional soma-expressão*
| *soma-expressão*
relacional → **<=** | **<** | **>** | **>=** | **==** | **!=**
soma-expressão → *soma-expressão soma termo* | *termo*
soma → **+** | **-**
termo → *termo mult fator* | *fator*
mult → ***** | **/**
fator → **(** *expressão* **)** | *var* | *ativação* | **NUM**
ativação → **ID** **(** *args* **)**
args → *arg-lista* | *vazio*
arg-lista → *arg-lista* **,** *expressão* | *expressão*

Figura 1: Gramática.

2. Especificação Flex

2.1. Definições

Na seção de definições da gramática, temos os seguintes conjuntos de símbolos descritos:

NUM [0-9]
CHAR [a-zA-Z]
ID {CHAR}({CHAR}|{NUM})*
COMPARISONS "<=" | "<" | ">" | ">=" | "==" | "!="
SUM "+" | "-"
MULT "*" | "/"
BOOL "true" | "false" | "TRUE" | "FALSE"
TYPES "int" | "float" | "bool" | "void"
KEYWORDS "if" | "else" | "while" | "return" | ";" | "[" | "]" | "(" | ")" | "{" | "}" | "," | "="

2.2. Tokens

Na tabela a seguir, temos a classificação das definições de um lexema para seu respectivo token, utilizado para inserção dos tokens na tabela de símbolos, que será discutida posteriormente.

Lexema	**Token**
{ID}	ID
{COMPARISONS}	CompOP
{SUM}	SumOP
{MULT}	MultOP
{BOOL}	Bool
{TYPES}	Type
{KEYWORDS}	Keyword

Tabela 1: Lexemas e seus tokens.

3. Comentários sobre a implementação

3.1. Estruturas de dados

3.1.1. Coord

A estrutura *Coord* atua como as ocorrências de um token na fita lida. Nela, tem-se a posição (linha e coluna) em que o token foi encontrado, e um ponteiro para a próxima ocorrência, caso ela exista.

3.1.2. Row

Para a estrutura *Row*, temos uma lista encadeada simples tendo uma política de acesso de uma fila, em relação a inserção de elementos.

Cada nó da lista, tem as seguintes informações:

- **lexema:** armazena o conteúdo (string) do token lido.
- **token:** armazena o escopo do token lido.
- **coords:** aponta para a lista de posições em que o token aparece.
- **next:** aponta para o próximo nó da lista.

3.1.3. Table

A estrutura *Table* serve apenas o propósito de servir como “cabeça” e “cauda” da lista, para que ao utilizar a função de adicionar um token na lista, que será discutida posteriormente, tenha uma complexidade temporal constante, sendo adicionado sempre ao final desta, com ajuda do ponteiro da cauda da lista. Já para funções como a de imprimir a tabela, em que desejamos partir do início dela, tem-se uma variável apontando para o início desta lista.

3.2. Funções

3.2.1. newTable

Na função *newTable* temos a alocação dinâmica de uma instância da estrutura *Table*, onde além de ser alocado espaço na memória para tal, tem-se os ponteiros para o início e fim da tabela iniciados e apontando para *NULL*.

3.2.2. addToken

Sempre que a regra da expressão regular de algum token é validada, ou seja, quando um token é encontrado, este é adicionado na tabela de símbolos, contendo as informações que contém em um nó da estrutura *table*, sendo estes, o lexema, o token e a posição que este foi encontrado.

3.2.3. pointError

Quando um erro é encontrado, informações como a posição encontrada e o erro em si são fatos relevantes para que o desenvolvedor possa encontrar o erro com facilidade, como neste trabalho está sendo feita a abordagem do analisador léxico, os erros encontrados nesta etapa de compilação serão léxicos. Desta maneira a função recebe como parâmetros o lexema do erro, o tipo de erro, a linha e coluna encontradas.

3.2.4. printTable

A função responsável por imprimir a tabela de símbolos geradas a partir da fita é *printTable*, ela vai receber o ponteiro da tabela e o número de tokens lidos, que é uma informação obtida ao adicionar novos tokens na tabela, e é utilizado como critério de parada para o laço de repetição. A tabela é impressa em um arquivo chamado “output.txt”.

3.3. Segmentação (arquivos)

3.3.1. compile.sh

Neste arquivo *sh*, temos um script com os comandos que seriam executados para a compilação do programa. Sendo estes, o flex, gcc e a própria execução do arquivo de saída.

3.3.2. helpers.h

Neste arquivo de cabeçalho, encontram-se as estruturas de dados utilizadas, e ainda as definições das funções auxiliares utilizadas. Sendo optado por fazer desta maneira para que no arquivo *.lex* fique não somente com as definições das gramáticas e especificações, mas para que fique centralizada as funções.

3.3.3. input-1.txt

Exemplo 1 de entrada de dados.

3.3.4. input-2.txt

Exemplo 2 de entrada de dados.

3.3.5. output.txt

Após ler o código de entrada de dados, o processo de análise léxica for efetuada e a tabela de símbolos gerada, esta é impressa neste arquivo de saída.

3.3.6. tr-parte1.lex

As definições e especificações (expressões regulares) da gramática, os terminais aceitos, e ainda a função *main*, função principal e responsável pela execução do código se encontram neste arquivo.

4. Compilação / Ambiente / Linguagem

Foram utilizadas as ferramentas Flex, em sua versão 2.6.4. Quanto à ferramenta de compilação, o gcc, a coleção de compiladores, trata-se da versão 11.2.0 (Ubuntu 11.2.0-19ubuntu11).

Para compilação e execução, temos um script em shell, que limpa o terminal, gera o código em c a partir do arquivo lex, e compila este arquivo gerado em c. Para efetuar a compilação basta executar o arquivo sh, para tal, navegue primeiramente para o diretório “COM/” e em seguida no diretório “trabalho1/”, utilize o comando “cd COM/trabalho1/”. Após estar dentro da pasta, execute o script passando como argumento o nome do arquivo que será feita a análise léxica utilizando “./compile.sh input-1.txt” ou “./compile.sh input-2.txt”. Caso não esteja liberado para execução, primeiro dê esta permissão para o arquivo, rodando o comando “chmod +x compile.sh”.

O código de entrada será lido, os tokens serão separados e inseridos na tabela de símbolos e esta é impressa no arquivo de saída, no final, apresentará os erros léxicos caso haja algum e o arquivo de saída “output.txt” será impresso no terminal utilizando o comando “cat output.txt”.

5. Análise de exemplos

5.1. Exemplo 1

5.1.1. Gramática

A gramática foi criada para apresentar erros quando analisada lexicalmente, nela contém lexemas identificadores, números inteiros, floats, palavras reservada na linguagem e dois tipos de comentários: “//” e “/* */”. Se houverem caracteres não reconhecidos na análise léxica porque não pertencem a nenhum Token pré-definido. No caso do “4.2.1” o analisador léxico identifica o “4.2” como um lexema do tipo float mas como contém um “.1” ele encontra um erro léxico.

5.1.2. Análise Léxica

Após a análise léxica obtivemos o output dos seguintes erros esperados da gramática:

- Erro léxico em linha 1, coluna 1: <3
- Erro léxico em linha 9, coluna 15: 4.5a
- Erro léxico em linha 10, coluna 2: 4.2.1

5.2. Exemplo 2

5.2.1. Gramática

Considerando erros léxicos, podemos considerar caracteres não reconhecidos, sendo estes terminais não adicionados nas definições da gramática, ou ainda uma produção mal formada, uma expressão que não siga nenhum dos padrões definidos que a linguagem deveria aceitar, como por exemplo um dígito seguido de um caractere, como “9x”. Apesar de tanto o elemento individual “9” quanto o elemento individual “x” serem elementos válidos, estes juntos, não fazem parte de nenhum conjunto gerado, apontando assim para um erro léxico.

5.2.2. Análise Léxica

Após efetuada a análise léxica do exemplo dado, os erros esperados, que foram discutidos acima, são os seguintes:

- Erro léxico na linha 6, coluna 1: !
- Erro léxico em linha 7, coluna 1: 9x
- Erro léxico em linha 8, coluna 1: 88dao7