Universidade Federal de Goiás
Instituto de Informática
Compiladores
2025-1

Compilador para a Linguagem Goianinha
Trabalho 1 - Análises Léxica e Implementação da Tabela de Símbolos

Thierson Couto Rosa

1 Objetivo

Este projeto tem como objetivo a implementação de um compilador didático para a linguagem *Goianinha*. O presente texto descreve a Etapa I do projeto que consiste na especificação de funções que implementem uma tabela de símbolos de símbolos e do analisador léxico para a linguagem.

2 Gramática para a linguagem *Goianinha*

A seguir, é apresentada a gramática para a linguagem *Goianinha*. Os não-terminais da gramática iniciam-se com letra maiúscula e o símbolo inicial é *Programa*. Os terminais aparecem em negrito quando são formados por palavras ou por sequência de caracteres. Exemplo: **int, +, - (), <=**, etc. Os **terminais** da gramática correspondem aos *tokens* (linguagens regulares) da linguagem *Goianinha*.

$$
\begin{array}{lcl}
Programa & \rightarrow & DeclFuncVar\ DeclProg \\
\\
DeclFuncVar & \rightarrow & Tipo\ \textbf{id}\ DeclVar\ ;\ DeclFuncVar \\
 & & |Tipo\ \textbf{id}\ DeclFunc\ DeclFuncVar \\
 & & |\epsilon \\
\\
DeclProg & \rightarrow & \textbf{programa}\ Bloco \\
\\
DeclVar & \rightarrow & ,\ \textbf{id}\ DeclVar \\
 & & |\ \epsilon
\end{array}
$$

$$
\begin{array}{lll}
DeclFunc & \rightarrow & (\ ListaParametros\ )\ Bloco \\
\\
ListaParametros & \rightarrow & \epsilon \\
 & & |ListaParametrosCont \\
\\
ListaParametrosCont & \rightarrow & Tipo\ \textbf{id} \\
 & & |Tipo\ \textbf{id}\ ,\ ListaParametrosCont \\
\\
Bloco & \rightarrow & \{\ ListaDeclVar\ \ ListaComando\ \} \\
 & & |\{\ ListaDeclVar\ \} \\
\\
ListaDeclVar & \rightarrow & \epsilon \\
 & & |Tipo\ \textbf{id}\ DeclVar\ ;\ ListaDeclVar \\
\\
Tipo & \rightarrow & \textbf{int} \\
 & & |\textbf{car} \\
\\
ListaComando & \rightarrow & Comando \\
 & & |Comando\ \ ListaComando \\
\\
Comando & \rightarrow & ; \\
 & & |Expr\ ; \\
 & & |\textbf{retorne}\ \ Expr\ ; \\
 & & |\textbf{leia}\ LValueExpr\ ; \\
 & & |\textbf{escreva}\ \ Expr\ ; \\
 & & |\textbf{escreva}\ \ \text{“}\textbf{cadeiaCaracteres}\text{”}\ ; \\
 & & |\textbf{novalinha}\ ; \\
 & & |\textbf{se}\ (\ Expr\ )\ \textbf{entao}\ Comando \\
 & & |\textbf{se}\ (\ Expr\ )\ \textbf{entao}\ Comando\ \ \textbf{senao}\ Comando \\
 & & |\textbf{enquanto}\ (\ Expr)\ \textbf{execute}\ Comando \\
 & & |Bloco \\
\\
Expr & \rightarrow & OrExpr \\
 & & |LValueExpr{=}AssignExpr \\
\\
OrExpr & \rightarrow & OrExpr\ \textbf{ou}\ AndExpr \\
 & & |AndExpr
\end{array}
$$

$$
\begin{array}{lcl}
AndExpr & \rightarrow & AndExpr \ \textbf{e} \ EqExpr \\
 & & |EqExpr \\
 & & \\
EqExpr & \rightarrow & EqExpr == DesigExpr \\
 & & |EqExpr \ \textbf{!=} \ DesigExpr \\
 & & |DesigExpr \\
 & & \\
DesigExpr & \rightarrow & DesigExpr < AddExpr \\
 & & | \ DesigExpr > AddExpr \\
 & & | \ DesigExpr >= AddExpr \\
 & & | \ DesigExpr <= AddExpr \\
 & & |AddExpr \\
 & & \\
AddExpr & \rightarrow & AddExpr + MulExpr \\
 & & |AddExpr - MulExpr \\
 & & |MulExpr \\
 & & \\
MulExpr & \rightarrow & MulExpr * UnExpr \\
 & & |MulExpr \ / \ UnExpr \\
 & & | \ UnExpr \\
 & & \\
UnExpr & \rightarrow & -PrimExpr \\
 & & |! \ PrimExpr \\
 & & |PrimExpr \\
 & & \\
LValueExpr & \rightarrow & \textbf{id} \\
 & & \\
PrimExpr & \rightarrow & \textbf{id} \ (\ ListExpr \ ) \\
 & & | \ \textbf{id} \ (\ ) \\
 & & | \ \textbf{id} \\
 & & | \ \textbf{carconst} \\
 & & | \ \textbf{intconst} \\
 & & | \ (\ Expr \ ) \\
 & & \\
ListExpr & \rightarrow & Expr \\
 & & | \ ListExpr \ , \ Expr
\end{array}
$$

3 Tabela de Símbolos para a linguagem Goianinha

Conforme mostra a descrição da gramática, um bloco pode conter uma lista de declaração de variáveis locais (que pode ser vazia) e pode ou não ter uma lista de comandos. Porém, um comando da lista de comandos pode ser um outro bloco. Ou seja, os blocos aninham-se uns dentro dos outros. Além disso, cada bloco que possui uma declaração de variáveis locais inicia um escopo novo, exceto no caso de funções onde os identificadores que formam os parâmetros da função estão no mesmo escopo do bloco da função. Essa situação permite a coexistência de variáveis/parâmetros distintos que possuem o mesmo lexema de identificador (mesmo nome), por exemplo um mesmo nome x declarado em um escopo mais externo e outro objeto de

nome x declarado em um escopo mais interno. Veja explicações mais detalhadas no slide sobre tabelas de símbolos na Plataforma Turing.

Nesta parte do trabalho deve ser implementada uma pilha de tabela de símbolos (ou tabela de nomes) capaz de armazenar lexemas de identificador. Essa estrutura deve ter associada a ela o seguinte conjunto de operações:

a) iniciar a pilha de tabela de símbolos - operação que cria uma estrutura de dados inicialmente vazia para representar a pilha de tabelas de símbolos (pilha de escopos);

b) criar uma nova tabela de símbolos (novo escopo) e empilhá-la na pilha de tabela de símbolos.

c) pesquisar por um nome (lexema de identificador) na pilha de tabelas de símbolos. Essa função deve iniciar a pesquisa na tabela que está no topo da pilha (tabela de símbolos correspondente ao último escopo criado) e enquanto não encontrar, procurar nas tabelas seguintes, do topo em direção à base da pilha de tabelas de símbolos. Deve retornar um ponteiro para a entrada da tabela de símbolos onde o nome foi encontrado ou um ponteiro vazio, caso o nome procurado não se encontre na pilha de tabelas.

d) remover escopo atual - essa função elimina a tabela de símbolos que está no topo da pilha de tabelas.

e) inserir um nome de função na tabela de símbolos atual (que está no topo da pilha de tabelas). Essa função deve conter como parâmetros: a string com o nome da função, um número inteiro correspondente ao número de parâmetros da função e o tipo retornado pela função (os dois últimos parâmetros serão úteis para as fases de análise semântica e/ou geração de código do compilador);

f) inserir um nome de variável na tabela de símbolos atual. Essa função deve ter como parâmetros: a string com o nome da variável, um inteiro indicando o tipo da variável (informação a ser utilizada futuramente, na análise semântica) e um inteiro positivo, indicando em qual posição da lista de declaração de variáveis a variável ocorreu (essa informação será útil posteriormente, para a geração de código).

g) inserir o nome de um parâmetro na tabela de símbolos atual. Essa função deve ter como parâmetros: a string com o nome do parâmetro, um inteiro indicando o tipo do parâmetro (informação a ser utilizada futuramente, na análise semântica), um inteiro positivo, indicando em qual posição da lista de declaração de parâmetro o parâmetro ocorreu (essa informação será útil posteriormente, para a geração de código) e um ponteiro para a entrada da tabela de símbolos atual onde se encontra o nome da função a qual esse parâmetro pertence.

h) eliminar a pilha de tabelas de símbolos.

No momento atual da disciplina não fica clara a importância da pilha de tabela de símbolos. Ela será utilizada pelo analisador semântico e pelo gerador de código. Portanto, nesta primeira fase, o(a) aluno(a) deve implementar um programa de teste, apenas para demonstrar que as funções descritas acima funcionam corretamente.

4 Analisador Léxico

A função que implementa o analisador léxico poderá ser gerada automaticamente, utilizando-se um gerador de analisadores léxicos, por exemplo, Lex, ou Flex. O Lex e o Flex geram um analisador léxico escrito na linguagem C. O Flex possui, ainda, a opção de gerar um analisador léxico na linguagem C++.

A função gerada deve ser capaz de reconhecer os *tokens* da gramática da linguagem *Goianinha* especificada na Seção 2. Os tokens correspondem aos símbolos e palavras que aparecem em negrito na gramática. As palavras reservadas de *Goianinha* são: **programa**, **car**, **int**, **retorne**, **leia**, **escreva**, **novalinha**, **se**, **entao**, **senao**, **enquanto**, **execute**. No caso em que o *token* é uma constante *string* (**constString**), ou um identificador (**id**), a função deve gerar como lexema, o texto que forma a *string* ou o identificador (ex.: "x1", "cont2").

Um identificador em *Goianinha* deve iniciar com letra ou "_", seguido de zero ou mais letras, dígitos ou "_".

O analisador léxico deverá também processar e descartar comentários. Os comentários podem ocupar mais de uma linha do programa fonte. Um comentário em *Goianinha* inicia com o par de símbolos "/*"e termina com o par de símbolos "*/". Um erro deve ser reportado, caso um comentário não termine.

O analisador léxico deve reportar, através de mensagens, erros léxicos encontrados no arquivo de entrada. São exemplos de erros léxicos: caracteres inválidos na linguagem, comentários que não terminam, cadeia de caracteres que não terminam ou que ocupam mais de uma linha no arquivo de entrada.

Caso o arquivo contenha um erro léxico, o programa deverá imprimir uma linha contendo exatamente o seguinte: **ERRO:**, seguido por um espaço em branco e, por uma das seguintes mensagens de erro: **CARACTERE INVÁLIDO** ou **COMENTÁRIO NAO TERMINA** ou **CADEIA DE CARACTERES OCUPA MAIS DE UMA LINHA**. Após a mensagem de erro o programa deve imprimir, na mesma linha, o número da linha do programa fonte onde o erro foi encontrado.

Geralmente, o código obtido por uma ferramenta geradora de analisador léxico corresponde a uma função na linguagem para a qual o analisador é gerado. Por exemplo, o Flex e o Lex geram uma função denominada `yylex()`. essa função é do tipo `int` e precisa retornar ao programa que a chama uma constante inteira que indica o token encontrado. A função deve retornar -1 (EOF) quando o fim de arquivo (padrão ¡¡EOF¿¿, veja manual da ferramenta) é encontrado. A função `yylex()` armazena em uma variável global yytext o lexema encontrado na entrada. Essa variável é um ponteiro para `char` e deve ser uma variável global, pois seu conteúdo será acessado pelo código gerado pela ferramenta e pelo programa que irá chamar a função `yytext()`. Descrevemos esse programa como *Falso Analisador Sintático* na seção seguinte.

5 Falso Analisador Sintático

Para ser possível testar o código do analisador léxico gerado, deve ser implementado, em um arquivo separado, um programa `main.c` que estamos chamando aqui de "Falso sintático" porque ele não faz análise sintática. Ele apenas fica chamando recursivamente a função `yylex()` e imprimindo o *token* e o lexema encontrados pela função. Para cada token encontrado, o programa imprime ainda a linha onde o token ocorreu, através da variável global `yylineno`.

Esse programa será substituído posteriormente pelo código do analisador sintático obtido por outra ferramenta, geradora de analisadores sintáticos (ex. Bison, Yacc). O programa tem as seguintes características:

- Deve ter as seguintes declarações globais:
  - `FILE *yyin;`
  - `extern int yylineno;`
  - `extern char* yytext;`
  - `extern int yylex();`

  A variável global `yyin` é utilizada pela função `yylex()` gerada pela ferramenta para processar o arquivo de entrada, lendo caractere por caractere do mesmo. A variável global `yylineno;` contém a linha do arquivo de entrada onde o token/lexema foi encontrado. Como ela é definida no código em C gerado pela ferramenta, é necessário usar a diretiva `extern` para indicar ao *linker* do gcc que ela já foi declarada em outro arquivo externo ao programa `main.c`. A função `yylex()` é a função na linguagem C que corresponde ao analisador léxico produzido pelo ferramenta geradora (Lex, Flex). Essa função também é definida globalmente no arquivo em C gerado pela ferramenta geradora de analisador léxico utilizada. Logo, no arquivo `main.c` ela deve ser declarada como `extern`, do contrário, o compilador cria outra variável com esse nome, enquanto queremos que os dois códigos usem a mesma variável.

- O programa recebe como parâmetro (`argv[1]`) o nome do arquivo fonte contendo um programa escrito na linguagem *Goianinha*.

- O programa deve abrir o arquivo com o nome em `argv[1]` armazenando o ponteiro para o arquivo aberto em `yyin` (descrito acima).

- O programa deve ter um loop controlado por fim de arquivo: `while(token=yylex()!=EOF){` que fica chamando a função `yylex()` que a cada novo token encontrado na entrada, retorna o código do *token* (constante inteira presente em `goianinha.c`).

- Após cada chamada o programa deve emitir uma mensagem: `"Encontrado o lexema %s pertencente ao token de codigo %d linha %d"`, onde o padrão %s deve ser substituído pelo valor do lexema que está na variável `yytext`, o valor do primeiro "%d"deve ser substituído pelo código do *token* encontrado e o segundo "%d"deve ser substituído pelo número da linha onde o *token* foi encontrado.

O programa principal que chama a função analisador sintático deve receber o nome do arquivo a ser compilado como parâmetro de entrada. Especificamente, dado que o programa executável do analisador sintático tenha o nome `goianinha`, e supondo que o arquivo de entrada seja `teste.g`, deve ser possível executar o falso sintático sobre o arquivo através do seguinte comando em uma *shell* do Linux: `./goianinha teste.g`.

6 Informações Sobre a Implementação e a Entrega

O trabalho é individual e deve ser entregue até o dia **14/04/2025** via tarefa criada no site da disciplina na plataforma Turing. Deve ser entregue uma pasta contendo duas

sub-pastas, uma com os códigos relacionados a implementação da pilha de tabelas de símbolos, e a outra, contendo os códigos relacionados ao analisador léxico. A sub-pasta contendo os códigos relacionados à tabela de deve conter:

- um arquivo header (.h) contendo a descrição das estruturas de dados utilizadas e protótipos das funções que manipulam a pilha de tabela de símbolos.
- um arquivo com as implementações das funções (arquivo .c ou .c++)
- um arquivo com a função `main()` que realiza os testes das funções implementadas.
- um arquivo makefile que realiza a compilação e a ligação dos módulos.

A sub-pasta contendo os códigos relacionados ao analisador léxico deve conter o seguinte:

- O código de entrada para o gerador de analisador léxico utilizado (arquivo denominado `goianinha.l`).
- O código do falso sintático (arquivo denominado `main.c`.
- O arquivo `Makefile` contendo:
  - Comando de execução do gerador de analisador léxico (Flex ou Lex) para conversão do arquivo de entrada `goianinha.l` em um programa em C (`goianinha.c` se a linguagem escolhida for C).
  - Comandos de compilação e link-edição para compilar e ligar o programa principal com o código do analisador léxico gerado pela ferramenta utilizada, para gerar o executável `goianinha`.

A pasta contendo as duas sub-pastas deve ser compactada com o utilitário gzip e submetida como uma tarefa a ser criada no site da disciplina. Os códigos gerados devem estar preparados para executarem no sistema operacional Linux (ambiente onde os trabalhos serão avaliados). No caso do analisador léxico, o programa executável deve ter o nome `goianinha`.

Importante:

Cópias idênticas ou modificadas dos códigos ou de partes dos códigos são consideradas plágios. **Plágio é crime**. O aluno que cometer plágio em seu trabalho receberá nota zero no mesmo.