Trabalho 3 da disciplina de Tradutores*

Ana Paula Martins Tarchetti[1] - 17/0056082

Departamento de Ciência da Computação, Universidade de Brasília (UnB) - Brasília, DF, 70910-900
aptarchetti@gmail.com

1 Introdução e motivação

Um compilador consiste em traduzir um programa em uma linguagem de programação fonte para uma linguagem de programação alvo. Além disso, um compilador também relata possíveis erros que possam estar presentes no código-fonte. A compreensão do processo de compilação/tradução permite a utilização das técnicas aprendidas em aplicações futuras. Exemplos desse uso são: editores de texto, sistemas de recuperação de informações, reconhecimento de padrões, construção de novas linguagem de programação, desenho digital e verificadores [ALSU03].

Este trabalho tem como objetivo fixar de forma prática o entendimento desse processo de construção de compiladores computacionais. Dessa forma, este documento escrito traz os detalhes da implementação que foi feita com base nos processos envolvidos na compilação/tradução de programas. A linguagem de programação fonte que será utilizada é a C-IPL, que se baseia em um subconjunto da linguagem C com algumas alterações e uma nova primitiva que foi criada com o intuito de facilitar a manipulação de listas. Em adição a isso, a linguagem de programação alvo da compilação é a *Three Address Code* [San15].

1.1 Analisador léxico

A primeira etapa que foi implementada é o analisador léxico, também chamado de analisador linear. A função dessa etapa é identificar em um código as sequências de caracteres que formam *tokens*, sendo esse a unidade mínima que compõe o vocabulário de um programa e é composto da seguinte forma: <nome do token, atributos do token>.

Para realizar a implementação dessa etapa foi utilizada a ferramenta *Flex*, [PEM07], cujo intuito é gerar programas que realizam correspondência de padrões em um dado texto. Então, por meio dessa ferramenta, foram declaradas expressões regulares (que foram detalhadas no **Apêndice A**) para indicar as seguintes definições:

- LETRA: qualquer caractere alfabético;
- DIGITO: qualquer algarismo;
- ID: identificadores de funções e variáveis;

* Professora: Cláudia Nalon

- INTEGER: reconhece números inteiros;
- FLOAT: reconhece números de ponto flutuante;
- STRING_LITERAL: cadeias de texto delimitadas por aspas duplas;
- ESPAÇO: qualquer espaço em branco;
- Outras expressões regulares para reconhecimento de erros específicos.

Após isso, foram definidas regras de identificação de *tokens*. As regras declaradas são divididas em:

- Palavras-chave: **int**, **float**, **list**, **read**, **write**, **writeln**, **main**, **return**, **if**, **else**, **for**, **NIL**;
- Novas operações primitivas: "**?**", "**%**", "**»**", "**«**", ":", "**!**" (sobrecarregado, podendo também denotar negação);
- Operadores: **||**, **&&**, >, <, >=, <=, ==, **!**=, +, **-**, *****, /, =;
- Delimitadores e outros: **vírgula**, **ponto e vírgula**, **parênteses** e **chaves**;
- Regras auxiliares: Ignorar espaços, indicar comentários e indicar erros léxicos e suas linhas e colunas correspondentes.

1.2 Analisador sintático

O analisador sintático, também chamado de analisador gramatical, corresponde à segunda etapa da compilação. Essa etapa se baseia em uma Gramática Livre de Contexto (GLC) para expressar regras recursivas ou não, que indicam as estruturas de agrupamento de *tokens* que podem ser aceitas pelo compilador. A GLC utilizada neste trabalho está descrita no **Apêndice B** deste documento. Ademais, é nessa etapa em que se cria/popula a árvore sintática abstrata, essa estrutura servirá de auxílio para as próximas etapas do compilador/tradutor.

Para realizar a implementação dessa etapa foi utilizada a ferramenta Bison, [DS15], cujo intuito é gerar programas que realizam essa análise supracitada a partir da declaração de uma GLC em um arquivo de extensão "y". Em adição a isso, foram implementadas, em linguagem C, funções e estruturas auxiliares para que fosse possível a construção da tabela de símbolos e da árvore sintática. Ambas estruturas foram feitas por meio do uso de *struct's* para representar nós de uma árvore e ponteiros para *struct's* afim de poder relacionar esses nós.

A seguir, é possível compreender como se dá a organização das estruturas declaradas:

1. A tabela de símbolos é uma lista encadeada, aonde os nós possuem os seguintes campos:
   - char *nome;
   - char *tipo;
   - int escopo;
   - char *var_ou_func;
   - struct t_simbolo * proximo;
2. A árvore sintática abstrata é uma estrutura mista de árvore e lista encadeada, aonde os nós possuem os seguintes campos:
   - char *nome;

   - int linha;
   - int coluna;
   - struct t_node *primeiro_filho;
   - struct t_node *proximo_irmao;;
3. Dessa forma a profundidade da árvore é definida pelos ponteiros de filho e a largura da árvore é definida pelos ponteiros de irmão.

1.3 Analisador semântico

Nesta etapa, foi implementada a análise semântica. Nessa parte, além do uso da árvore sintática abstrata e da tabela de símbolos (lista encadeada), também foi feito o uso da árvore de escopo. A árvore de escopo se dá da seguinte forma:

1. Árvore de escopo: composta por *struct* de nós de árvore de escopo com os seguintes campos:
   - Número do escopo;
   - Ponteiro para o escopo pai;
2. Além disso, foram utilizadas variáveis auxiliares para armazenar o escopo atual e o contador de escopos existentes.
3. Por fim, as funções 'incrementa_escopo' e 'decrementa_escopo' são chamadas ao início de funções/blocos e ao final de funções/blocos respectivamente para o gerenciamento da árvore de escopo e das variáveis auxiliares.

Com base nisso, esta parte da implementação realiza a verificação dos seguintes pontos:

1. A existência da declaração da função 'main'.
   - Para isso foi criada a função 'existe_main' que verifica isso de acordo com a tabela de símbolos.
2. A redeclaração de variáveis e funções no mesmo escopo devem gerar erro.
   - Para isso foi criada a função 'incrementa_tabela' que verifica isso de acordo com a tabela de símbolos e apenas adiciona o símbolo caso ele não tenha sido declarado no escopo em questão, caso o contrário um erro é gerado.
3. O uso de variáveis e funções que não tenha sido declaradas em seu contexto de uso (escopo de uso ou escopo pai do escopo de uso) deve gerar erro.
   - Para isso foi criada a função 'verifica_contexto' que verifica isso de acordo com a tabela de símbolos e também de acordo com a árvore de escopo, supracitada.
4. A quantidade de parâmetros passados na chama de uma função deve ser a mesma quantidade de parâmetros definidos na declaração da função.
5. O tipo da expressão de retorno, caso exista chamada de retorno, deve ter o mesmo tipo de retorno do tipo da função, ou deve poder ser convertido implicitamente para o tipo declarado na função.
6. Para todos os casos em que se apliquem deve ser possível fazer a conversão implícita de tipos, caso o contrário, um erro deve ser gerado.

2 Compilação, Execução e Testes

Os passos para compilação e execução do analisador podem ser encontrados no arquivo **README.txt** do diretório principal do projeto, ou, a partir do diretório principal, siga os seguintes comandos:

```
Compilar:
$ bison -v --defines=lib/sintatico.tab.h src/sintatico.y
  -o src/sintatico.tab.c
$ flex -o src/lex.yy.c src/lexico.l
$ gcc-11 -std=c18 -std=c99 -Wall -Wextra -Wpedantic -g -I lib/ -Wall
  -o tradutor src/sintatico.tab.c src/lex.yy.c src/tradutor_utils.c -lfl

Compilar e executar:
$ make
$ ./tradutor < tests/lexico/<nome_do_teste>
$ ./tradutor < tests/sintatico/<nome_do_teste>
$ ./tradutor < tests/semantico/<nome_do_teste>
```

Tendo isso, foram criados os seguintes **arquivos de teste** para validar o funcionamento esperado nesta etapa da implementação:

- (i) **semantico/correto_teste_1.c** e (ii) **semantico/correto_teste_2.c** que não apresentam erros;
- (iii) **semantico/errado_teste_1.c**: Esse teste possui dois erros sintáticos e dois erros semânticos, o primeiro erro semântico é a falta da declaração da função 'main' no escopo global, o sengundo erro semântico é o uso da variável 'b' (linha 4, coluna 5) sem ter declarado a variável no escopo do uso ou em algum escopo pai do escopo do uso. Já o primeiro erro sintático é um ponto-e-vírgula inesperado (linha 6, coluna 8) e o segundo erro sintático é a vírgula inesperada (linha 7, coluna 21).
- (iv) **semantico/errado_teste_2.c**: Esse teste possui dois erros sintáticos e dois erros semânticos, o primeiro erro semântico é a redeclaração da variável 'a' no mesmo escopo (linha 3, coluna 11), inicialmente declarada na linha 2 e coluna 9, o sengundo erro semântico é a atribuição de uma constante do tipo INT a uma variável declarada com o tipo LIST (INT), sem que seja possível a conversão implícita de tipos. Já o primeiro erro sintático é um FECHA_PARENTESES esperado (linha 9, coluna 11) e o segundo erro sintático é uma CONTANTE (INT) inesperada (linha 10, coluna 11).

2.1 Ferramenta extra para a visualização da árvore sintática abstrata

Caso seja necessário, a implementação provê uma opção de uso de uma ferramenta para gerar um visualização da árvore sintática abstrata de cada teste, para isso siga os seguintes passos:

1. Execute o teste escolhido;
2. Pegue o conteúdo do arquivo "tree_output_file.txt";
3. Use o texto desse arquivo como entrada para a seguinte ferramenta [EE20]: https://ironcreek.net/syntaxtree/

Um exemplo de uso dessa ferramenta pode ser encontrado na **Figura 1**.

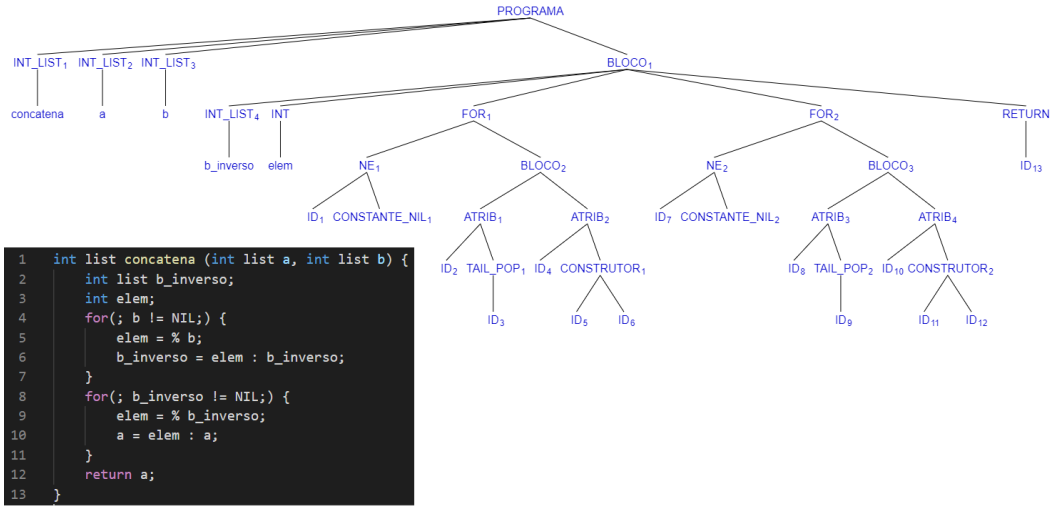

Figura 1. Exemplo de uso da ferramenta de visualização da árvore sintática abstrata, de acordo com o teste feito no programa mostrado acima que implementa a concatenação de duas listas.

Referências

[ALSU03] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[DS15] Charles Donnelly and Richard Stallman. Gnu bison–the yacc-compatible parser generator. *Free Software Foundation, Cambridge*, 2015.

[EE20] Mei Eisenbach and André Eisenbach. jssyntaxtree. https://ironcreek.net/syntaxtree/, 2003-2020. (Último acesso em 09/17/2021).

[PEM07] Vern Paxson, Will Estes, and John Millaway. Lexical analysis with flex. *University of California*, page 28, 2007.

[San15] Luciano Santos. Tac - interpretador de código de três endereços - manual de referência, ref. versão 0.11. https://github.com/lhsantos/tac/blob/master/doc/tac.pdf, Março 2015. (Último acesso em 10/03/2021).

Apêndice A Tabela de definições regulares

Nome	Expressões Regulares (Regex)	Padrão
LETRA	[a-zA-Z]	letras (maiúsculas e minúsculas)
DIGITO	[0-9]	dígitos de 0 a 9
ID	[_a-zA-Z][_a-zA-Z0-9]*	identificadores
INTEGER	{DIGITO}+	número inteiro
FLOAT	({DIGITO}*\.?{DIGITO}+ | $\{DIGITO\} + \backslash.$)	número com ponto flutuante
STRING_LITERAL	\"(\\.[^"\\])*\"	expressões entre aspas duplas
ESPACO	[\t\v\f\r]	todos os tipos de espaço em branco

Apêndice B Gramática

⟨*programa*⟩ ::= ⟨*lista_de_declaracoes*⟩

⟨*lista_de_declaracoes*⟩ ::= ⟨*lista_de_declaracoes*⟩ ⟨*declaracao*⟩
| ε

⟨*declaracao*⟩ ::= ⟨*declaracao_de_variavel*⟩
| ⟨*declaracao_de_funcao*⟩

⟨*declaracao_de_variavel*⟩ ::= ⟨*tipo_de_variavel_id*⟩ **PONTO_VIRGULA**

⟨*tipo_de_variavel_id*⟩ ::= ⟨*tipo_de_variavel*⟩ ⟨*id*⟩

⟨*id*⟩ ::= **ID**

⟨*declaracao_de_funcao*⟩ ::= ⟨*tipo_de_variavel_id*⟩ **ABRE_PARENTESES**
⟨*parametros*⟩ **FECHA_PARENTESES** ⟨*definicao_de_funcao*⟩

⟨*definicao_de_funcao*⟩ ::= ⟨*bloco_de_comando*⟩

⟨*parametros*⟩ ::= ⟨*lista_de_parametros*⟩
| ε

⟨*lista_de_parametros*⟩ ::= ⟨*lista_de_parametros*⟩ **VIRGULA** ⟨*parametro*⟩
| ⟨*parametro*⟩

⟨*parametro*⟩ ::= ⟨*tipo_de_variavel_id*⟩

⟨*comando*⟩ ::= ⟨*bloco_de_comando*⟩
| ⟨*comando_unico*⟩

⟨*comandos*⟩ ::= ⟨*comandos*⟩ ⟨*comando*⟩
| ε

⟨*bloco_de_comando*⟩ ::= **ABRE_CHAVES** ⟨*comandos*⟩ **FECHA_CHAVES**

⟨*comando_ unico*⟩ ::= ⟨*comando_ condicional*⟩
| ⟨*comando_ iterativo*⟩
| ⟨*declaracao_ de_ variavel*⟩
| ⟨*chamada_ de_ retorno*⟩
| ⟨*comando_ de_ atribuicao*⟩
| ⟨*expressao*⟩ **PONTO_VIRGULA**

⟨*comando_ condicional*⟩ ::= **IF ABRE_PARENTESES** ⟨*expressao*⟩ **FECHA_PARENTESES** ⟨*comando*⟩
| **IF ABRE_PARENTESES** ⟨*expressao*⟩ **FECHA_PARENTESES** ⟨*comando*⟩ **ELSE** ⟨*comando*⟩

⟨*comando_ iterativo*⟩ ::= **FOR ABRE_PARENTESES** ⟨*expressao_for*⟩ **PONTO_VIRGULA** ⟨*expressao_for*⟩ **PONTO_VIRGULA** ⟨*expressao_for*⟩ **FECHA_PARENTESES** ⟨*comando*⟩

⟨*expressao_for*⟩ ::= ⟨*id*⟩ **ATRIB** ⟨*expressao*⟩
| ⟨*expressao*⟩

⟨*expressao*⟩ ::= ⟨*exp*⟩
| ε

⟨*exp*⟩ ::= ⟨*exp*⟩ **GT** ⟨*exp*⟩
| ⟨*exp*⟩ **LT** ⟨*exp*⟩
| ⟨*exp*⟩ **EQ** ⟨*exp*⟩
| ⟨*exp*⟩ **NE** ⟨*exp*⟩
| ⟨*exp*⟩ **LE** ⟨*exp*⟩
| ⟨*exp*⟩ **GE** ⟨*exp*⟩
| ⟨*exp*⟩ **AND** ⟨*exp*⟩
| ⟨*exp*⟩ **OR** ⟨*exp*⟩
| ⟨*exp_ list*⟩

⟨*exp_ list*⟩ ::= ⟨*exp_ list*⟩ **CONSTRUTOR** exp_l⟨*exp_ list*⟩ist
| ⟨*exp_ list*⟩ **FILTER** ⟨*exp_ list*⟩
| ⟨*exp_ list*⟩ **MAP** ⟨*exp_ list*⟩
| ⟨*exp_ aritmetica*⟩
| ε

⟨*exp_ aritmetica*⟩ ::= ⟨*termo*⟩
| ⟨*exp_ aritmetica*⟩ **SOMA** ⟨*termo*⟩
| ⟨*exp_ aritmetica*⟩ **SUB** ⟨*termo*⟩

⟨*termo*⟩ ::= ⟨*fator*⟩
| ⟨*termo*⟩ **MULT** ⟨*fator*⟩
| ⟨*termo*⟩ **DIV** ⟨*fator*⟩

⟨*fator*⟩ ::= ⟨*constante*⟩
| ⟨*func_ call_ exp*⟩
| **SUB** ⟨*fator*⟩
| **SOMA** ⟨*fator*⟩

| **TAIL_OR_NOT** ⟨*fator*⟩
| **TAIL_POP** ⟨*fator*⟩
| **HEADER** ⟨*fator*⟩
| **ID**
| **ABRE_PARENTESES** ⟨*exp*⟩ **FECHA_PARENTESES**

⟨*comando_de_atribuicao*⟩ ::= ⟨*id*⟩ **ATRIB** ⟨*expressao*⟩ **PONTO_VIRGULA**

⟨*func_call_exp*⟩ ::= ⟨*id*⟩ **ABRE_PARENTESES** ⟨*func_call_parameters*⟩ **FECHA_PARENTESES**
| **READ ABRE_PARENTESES** ⟨*expressao*⟩ **FECHA_PARENTESES**
| **WRITE ABRE_PARENTESES** ⟨*expressao*⟩ **FECHA_PARENTESES**
| **WRITELN ABRE_PARENTESES** ⟨*expressao*⟩ **FECHA_PARENTESES**

⟨*func_call_parameters*⟩ ::= ⟨*func_call_parameters*⟩ **VIRGULA** ⟨*expressao*⟩
| ⟨*expressao*⟩

⟨*chamada_de_retorno*⟩ ::= **RETURN** ⟨*expressao*⟩ **PONTO_VIRGULA**

⟨*tipo_de_variavel*⟩ ::= **INT**
| **FLOAT**
| **INT LIST**
| **FLOAT LIST**

⟨*constante*⟩ ::= **INTEGER_CONST**
| **FLOAT_CONST**
| **CONSTANTE_NIL**
| **STRING_LITERAL**