Trabalho Final - Tradutor

Heitor de Lima Belém - 160123950

Universidade de Brasília 160123950@aluno.unb.br

1 Motivação

Este trabalho tem como objetivo a apresentação das análises léxica, sintática, semântica e geração de código intermediário para uma linguagem específica. [Nal]. Para fixação do conteúdo obtido através do livro base da disciplina [ALSU07], foi proposto o desenvolvimento dos analisadores léxico, sintático, semântico e também do gerador de código intermediário para a linguagem *C-IPL* [Nal], baseada nos princípios da linguagem *C*. O objetivo dessa linguagem é acrescentar um tipo de dados não existente em *C*, a lista.

Listas implementam uma coleção organizada de valores, assim como os vetores, entretanto, elas possuem operações especiais de acesso, adição e remoção de itens, que podem variar de linguagem para linguagem. Para a *C-IPL* [Nal], foram apresentadas operações de acesso aos elementos através dos operadores '?' e '!', operações de remoção utilizando o operador '%' e de inserção por meio do ':'. Por fim, são descritas funções para operar sobre as listas, sendo elas: filter, representada pelo operador binário '<<' e map, representada por '>>'.

2 Descrição da análise léxica

Para implementar o analisador léxico da linguagem proposta, foi utilizado o programa *FLEX* (*Fast Lexical Analyzer*), uma ferramenta geradora de programas que reconhecem padrões léxicos em textos [Est]. A estrutura de um arquivo reconhecido pelo *FLEX*, que possui a extensão .l, é dividida em três partes:

1. Definições
   Definições de funções, constantes, variáveis globais e inclusão de bibliotecas. Para este trabalho, foram criadas três variáveis globais: *errors_count*, *line_idx* e *column_idx*, que representam, respectivamente, a quantidade de erros obtidos durante a análise e o número da linha e coluna atual.
2. Regras
   Aqui são escritas as expressões regulares que vão procurar padrões no arquivo juntamente com as ações a serem tomadas ao encontrar tais padrões.
3. Código
   Nesta seção, encontra-se o código da função principal do arquivo com extensão *.l*, é aqui que será colocado o código escrito pelo usuário, este código é transferido e incorporado para o programa que implementa o autômato.

É importante ressaltar que, na fase de análise léxica, além da geração dos tokens através do processo de escaneamento do texto, é feita também a determinação dos escopos existentes no programa. Para isso, foi utilizado um contador global responsável por identificar o contagem de escopos atual.

3 Descrição da análise sintática

Para o analisador sintático deste trabalho, foi utilizada a ferramenta *Bison* [CS21], que consiste em um programa que gera, a partir das regras de uma gramática livre do contexto, um analisador sintático LR(1) canônico.

A estrutura básica de um arquivo reconhecido pelo *Bison*, que tem a extensão *.y*, segue a mesma ideia do Flex, com as mesmas seções. Entretanto, na seção de regras do Bison é onde ficam as regras da gramática livre de contexto.

3.1 Tabela de Símbolos

A tabela de símbolos é uma estrutura auxiliar, utilizada pelo compilador, para localizar variáveis ou funções (símbolos) utilizados durante a execução de um programa.

Para este trabalho, a implementação desse componente foi realizada através da utilização de uma lista de estruturas do tipo *T_Symbol*, que armazena informações importantes para a próxima etapa do processo de compilação: a análise semântica.

As informações guardadas para cada símbolo são: lexema, linha, coluna, escopo do identificador e flag para indicar se o símbolo é variável ou função.

3.2 Árvore de sintática

Para a criação da árvore sintática, foi utilizada uma estrutura de dados composta por nós. O conjunto de nós da árvore representa a estrutura sintática do programa a ser traduzido. Cada nó contem informações como regra atual da gramática, campo para indicar se o nó é terminal ou não, caso não seja terminal, ponteiro para a entrada na tabela de símbolos, caso seja uma variável, função ou argumento e o ponteiro para os nós filhos.

Com essa estrutura, é possível percorrer a árvore utilizando um algoritmo de busca em profundidade, apresentando dados relevantes sobre cada nó visitado e, posteriormente, buscando informações necessárias para a análise semântica.

4 Descrição da análise semântica

A análise semântica é responsável por verificar as instruções existentes em um código e determinar se, combinadas, elas fornecem um programa que possua sentido. Por exemplo, avaliar se uma variável utilizada foi declarada e está

acessível em um determinado trecho de código, apontar problemas em chamadas de funções com número errado de parâmetros, entre outros erros descritos posteriormente nesta seção são responsabilidades do analisador semântico.

Para realizar a análise semântica, foi utilizada a seção de regras do Bison [CS21], em conjunto com a tabela de símbolos, árvore sintática gerada pela análise sintática e a pilha de escopos, de modo que toda a análise é realizada em apenas uma passagem.

4.1 Regras gerais

Conforme exposto na descrição do trabalho [Nal], é necessário que o programa contenha a definição de uma função com o identificador **main** como a principal do programa. Para realizar esta análise, foi feita uma busca na tabela de símbolos por uma entrada que seguisse as condições de ter o lexema correspondente ao desejado, o escopo de sua declaração deveria ser o 0 (zero), que idenfica o escopo mais geral do programa e também o campo que indica se é variável, função ou parâmetro deveria indicar que a entrada encontrada é do tipo função.

4.2 Declaração e utilização de variáveis e funções

Para esta análise, foi utilizada a pilha de escopos e a tabela de símbolos. As etapas para a verificação consistem em, para a declaração de variáveis e funções, identificar se a variável ou função já foi declarada no mesmo escopo, percorrendo a tabela de símbolos à procura de um símbolo com o mesmo identificador do atual, caso encontre uma entrada na tabela que corresponda a essa condição, é lançado um erro semântico.

Já para a utilização de variáveis e funções, foi utilizada a pilha de escopos e a entrada correspondente ao símbolo atual na tabela de símbolos, de modo que fosse possível verificar se o identificador do escopo em que o símbolo foi utilizado possui também uma entrada na pilha de escopos, o que indicaria que a variável ou função possui definição disponível naquele escopo.

4.3 Chamada de função

Para a análise semântica das chamadas de função foram considerados os seguintes aspectos:

- Número de parâmetros: é necessário verificar se a quantidade de argumentos passados para a função é o mesmo que o determinado na declaração dela e, caso não seja, é informado o erro semântico. Para esta verificação, uma função recursiva percorre a lista de nós que representa os argumentos na árvore sintática, incrementando um contador. Por fim, o valor deste contador é comparado ao valor anotado na tabela de símbolos que corresponde ao número de parâmetros necessários para cada função.

- Análise dos tipos dos parâmetros e retorno: para este aspecto, é avaliado o tipo de cada parâmetro passado e também do valor retornado, de forma que, se não for o mesmo, há uma tentativa de fazer a coerção de tipos e, caso não seja possível, um erro semântico é lançado para o usuário. Esta análise é feita de forma semelhante à análise descrita acima, porém em vez de incrementar um contador, é acessada a entrada correspondente ao argumento na tabela de símbolos para verificar se o tipo do argumento é compatível com o tipo do parâmetro esperado naquela posição.
  - Conversão entre tipo lista e tipos *int* ou *float* não são possíveis, gera um erro semântico.
  - Conversão entre tipo int e float, gera um nó intermediário na árvore representando uma operação de coerção de tipos, de forma que o tipo resultante seja o que foi descrito na definição da função.
  - Listas do tipo int não podem ser convertidas para listas do tipo float e vice versa. Portanto, qualquer tentativa de conversão implícita gera um erro semântico.
- Para a verificação do retorno de funções, a lógica utilizada é que o retorno deve receber o mesmo tipo da função que determina o seu escopo, e deve obedecer às mesmas regras de conversão citadas nos itens acima.

4.4 Operações com listas

É possível dividir em operações unárias e binárias. Para o primeiro tipo de operação, é considerado que, ao encontrar um operador unário de listas, é necessário que exista, logo depois, um operando do tipo lista, e isso é verificado através da tabela de símbolos. Após essa verificação, é feita a anotação do tipo retornado por cada operação unária, seguindo a descrição da linguagem em [Nal]. Já para as operações binárias, como "map" e "filter", foi definido que o operando à esquerda do operador deve ser uma função unária que retorne um tipo simples e que o operando à direita seja do tipo lista. Caso essas especificações não sejam seguidas, é lançado um erro semântico. Para a verificação de tais condições, existe uma função que, dado o nó da árvore que corresponde à operação, verifica o tipo retornado por cada expressão que representa os operandos.

4.5 Conversão de tipos

Para realizar a conversão dos tipos, é preciso recuperar a informação do tipo da variável da árvore sintática, para então determinar para qual tipo ela deve ser convertida. Essa conversão é representada na árvore através da criação de um nó intermediário que indica qual foi a operação de coerção realizada e qual o tipo resultante dessa operação, como mostrado na figura 1

Para os tipos simples é possível fazer a coerção do tipo **int** para o tipo **float** e vice-versa. Para variáveis do tipo lista, não há como converter de **int list** para *float list*, isso gera um erro semântico. Outro ponto importante, relacionado às listas, é que constantes **NIL** só podem ser atribuídas a variáveis do tipo lista e devem receber o tipo da lista, por exemplo, se a lista for **int list**, então a constante será, também, do tipo **int list**.

```
|_ = [type: int]
  |_ identifier: <int, i>
  |_ cast [float->int] [type: int]
    |_ function_call [type: float]
      |_ identifier: <float, func>
      |_ function_args
        |_ function_args
          |_ function_args
            |_ function_args
            |_ cast [float->int] [type: int]
              |_ identifier: <float, t2>
          |_ cast [float->int] [type: int]
            |_ identifier: <float, t3>
        |_ cast [float->int] [type: int]
          |_ identifier: <float, t4>
```

Figura 1. Exemplo de coerção de tipos

5 Geração de código intermediário

O objetivo final desta etapa é gerar um código que funcione corretamente, com o comportamento esperado, ao ser executado. Para cumprir tal objetivo, foram seguidas as instruções disponíveis na documentação da ferramenta TAC.

Esta ferramenta é dividida em basicamente duas seções:

- **.table**
  Aqui são feitas as declarações de variáveis e constantes
- **.code**
  Nesta seção são armazenadas todas as funções e expressões contidas no programa.

Para construir o código de três endereços resultante desta etapa, foram feitas alterações na estrutura de cada nó da árvore para ser possível armazenar dados utilizados na construção do código de três endereços, como a operação feita, os parâmetros e o destino do resultado.

Para fazer o programa funcionar como esperado, algumas funções foram criadas e adicionadas no início da seção **.code**, para que possam ser utilizadas durante a execução do *TAC*, como por exemplo, funções para escrever uma cadeia de caracteres na tela, *write* e *writeln*.

5.1 Declarações de variáveis e conversões de tipos

Sempre que for encontrada uma declaração, ela é armazenada na tabela de símbolos do TAC (**.table**), juntamente com o seu escopo e lexema identificador, o que mantém a corretude do código em questão. Para variáveis criadas em tempo de execução, são utilizados os registradores disponibilizados na seção **.code**, em ordem crescente e de acordo com a necessidade.

Para as conversões de tipos, foram traduzidas as operações de coerção feitas durante a análise semântica (nós criados para identificar operações de conversão) para a linguagem reconhecida pelo TAC. Dessa forma, o código não possuirá erros de tipos incompatíveis.

6 Descrição dos arquivos de teste

Os arquivos utilizados para verificar o funcionamento do analisador semântico desenvolvido no trabalho estão no subdiretório */tests*. Arquivos com o prefixo **success_**, representam os casos em que a análise semântica não identifica nenhum erro durante o processo. Já os arquivos com o prefixo **wrong_** englobam os casos em que o analisador identifica erros semânticos.

Os erros apresentados para cada arquivo estão identificados abaixo:

- *wrong_ex1.c*

```
[SEMANTIC ERROR] Line: 17 | Column: 9 - Undefined reference to 'a'
[SEMANTIC ERROR] Line: 31 | Column: 9 - Invalid unary operation '-' for operand type
[SEMANTIC ERROR] Line: 32 | Column: 9 - Function or variable 'a' already declared
[SEMANTIC ERROR] Line: 33 | Column: 11 - Undefined reference to 'asd'
```

- *wrong_ex2.c*

```
[SEMANTIC ERROR] Line: 2 | Column: 12 - Undefined reference to 'a'
[SEMANTIC ERROR] Line: 2 | Column: 5 - Cannot cast from 'int' to ''
[SEMANTIC ERROR] Line: 23 | Column: 15 - Invalid '>>' binary operation
[SEMANTIC ERROR] Line: 24 | Column: 15 - Invalid '<<' binary operation
[SEMANTIC ERROR] Line: 31 | Column: 11 - Wrong number of arguments passed to 'mul'
[SEMANTIC ERROR] No 'main' function found
```

7 Compilação e execução do programa.

Requisitos para compilação: os programas *FLEX* e *BISON* (versão 2.6.4 e 3.7.6, respectivamente), o compilador *GCC* (versão 11.1.0), GNU Make (versão 4.3). Com isso devidamente instalado, é possível prosseguir para os seguintes passos.

- No diretório raiz do projeto:
- Verificar se existe a pasta */src/obj/*, caso não exista, é necessário criar antes de executar os comandos abaixo.

```
$ make all
$ ./tradutor ./tests/<nome_do_arquivo>.c
```

- Se desejar executar o valgrind para verificar possíveis vazamentos de memória, basta executar os seguintes comandos:

```
$ make all
$ make valgrind ./tradutor ARGS="<caminho_arquivo_teste>"
```

- Os comandos executados na compilação, com suas respectivas flags, são os seguintes:

```
bison -d ./src/parser.y -v
flex ./src/scanner.l
gcc -c -o src/obj/parser.tab.o src/parser.tab.c -ll -Wall -g -I ./lib
gcc -c -o src/obj/symbol_table.o src/symbol_table.c -ll -Wall -g -I ./lib
gcc -c -o src/obj/syntatic_tree.o src/syntatic_tree.c
-ll -Wall -g -I ./lib
gcc -c -o src/obj/lex.yy.o src/lex.yy.c -ll -Wall -g -I ./lib
gcc -o tradutor src/obj/parser.tab.o src/obj/symbol_table.o
src/obj/syntatic_tree.o src/obj/lex.yy.o -ll -Wall -g -I ./lib
```

Referências

[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, Tools*. Pearson/Addison Wesley, 2nd edition, 2007.

[CS21] R. Corbett and R. Stallman. Bison. https://www.gnu.org/software/bison/manual/bison.pdf, Online; acessado 08 de Agosto de 2021.

[Est] W. Estes. Flex: Fast lexical analyser generator. https://github.com/westes/flex. Online; acessado 08 de Agosto de 2021.

[Gup21] A. Gupta. The syntax of C in Backus-Naur form. https://tinyurl.com/max5eep, Online; acessado 08 de Agosto de 2021.

[Nal] C. Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034. Acessado pela última vez em 10/08/2021.

A Gramática

Gramática que descreve o compilador da linguagem C-IPL [Nal], corrigida do trabalho anterior, seguindo as observações da professora. A estrutura do [Gup21] foi utilizada como base.

⟨*program*⟩ ::= ⟨*declarations*⟩

⟨*declarations*⟩ ::= ⟨*declarations*⟩ ⟨*declaration*⟩
| ⟨*declaration*⟩

⟨*declaration*⟩ ::= ⟨*function_declaration_statement*⟩
| ⟨*variable_declaration_statement*⟩

⟨*block*⟩ ::= '{' ⟨*statements*⟩ '}'

⟨*statements*⟩ ::= ⟨*statements*⟩ ',' ⟨*statement*⟩
| ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*expression_statement*⟩
| ⟨*io_statement*⟩
| ⟨*return_statement*⟩
| ⟨*variable_declaration_statement*⟩
| ⟨*for_statement*⟩
| ⟨*if_else_statement*⟩
| ⟨*block*⟩

⟨*function_declaration_statement*⟩ ::= ⟨*SIMPLE_TYPE*⟩ ⟨*IDENTIFIER*⟩ '(' ⟨*parameters_optative*⟩ ')' ⟨*statement*⟩
| ⟨*SIMPLE_TYPE*⟩ ⟨*LIST_TYPE*⟩ ⟨*IDENTIFIER*⟩ '(' ⟨*parameters_optative*⟩ ')' ⟨*statement*⟩

⟨*parameters_optative*⟩ ::= ⟨*empty*⟩
| ⟨*parameters*⟩

⟨*parameters*⟩ ::= ⟨*parameters*⟩ ',' ⟨*parameter*⟩
| ⟨*parameter*⟩

⟨*parameter*⟩ ::= ⟨*SIMPLE_TYPE*⟩ ⟨*IDENTIFIER*⟩
| ⟨*SIMPLE_TYPE*⟩ ⟨*LIST_TYPE*⟩ ⟨*IDENTIFIER*⟩

⟨*for_statement*⟩ ::= ⟨*RW_FOR*⟩ '(' ⟨*expression_optative*⟩ ';' ⟨*or_expression_optative*⟩ ';' ⟨*expression_optative*⟩ ')' ⟨*statement*⟩

⟨*if_else_statement*⟩ ::= ⟨*RW_IF*⟩ '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
| ⟨*RW_IF*⟩ '(' ⟨*expression*⟩ ')' ⟨*statement*⟩ ⟨*RW_ELSE*⟩ '(' ⟨*expression*⟩ ')' ⟨*statement*⟩

⟨*expression_statement*⟩ ::= ⟨*expression*⟩ ';'

⟨*io_statement*⟩ ::= ⟨*input_statment*⟩
| ⟨*output_statement*⟩

⟨*input_statement*⟩ ::= ⟨*IO_READ*⟩ '(' ⟨*IDENTIFIER*⟩ ';'

⟨*output_statement*⟩ ::= ⟨*IO_WRITE*⟩ '(' ⟨*expression*⟩ ')' ';'
| ⟨*IO_WRITE*⟩ '(' ⟨*LIT_STRING*⟩ ')' ';'

⟨*return_statement*⟩ ::= 'return' ⟨*expression*⟩ ';'

⟨*expression*⟩ ::= ⟨*IDENTIFIER*⟩ '=' ⟨*expression*⟩
| ⟨*or_expression*⟩
| ⟨*function_call_expression*⟩

⟨*function_call_expression*⟩ ::= ⟨*IDENTIFIER*⟩ '(' ⟨*function_arguments_optative*⟩ ')'

⟨*function_arguments_optative*⟩ ::= ⟨*empty*⟩
| ⟨*function_arguments*⟩

⟨*function_arguments*⟩ ::= ⟨*function_arguments*⟩ ',' ⟨*function_argument*⟩
| ⟨*function_argument*⟩

⟨*function_argument*⟩ ::= ⟨*expression*⟩

⟨*expression_optative*⟩ ::= ⟨*empty*⟩
| ⟨*expression*⟩

⟨*or_expression_optative*⟩ ::= ⟨*empty*⟩
| ⟨*or_expression*⟩

⟨*or_expression*⟩ ::= ⟨*or_expression*⟩ ⟨*LOGICAL_OP_OR*⟩ ⟨*and_expression*⟩
| ⟨*and_expression*⟩

⟨*and_expression*⟩ ::= ⟨*and_expression*⟩ ⟨*LOGICAL_OP_AND*⟩ ⟨*equality_expression*⟩
| ⟨*equality_expression*⟩

⟨*equality_expression*⟩ ::= ⟨*equality_expression*⟩ ⟨*EQUALITY_OP*⟩ ⟨*relational_expression*⟩
| ⟨*relational_expression*⟩

⟨*relational_expression*⟩ ::= ⟨*relational_expression*⟩ ⟨*RELATIONAL_OP*⟩ ⟨*list_expression*⟩
| ⟨*list_expression*⟩

⟨*list_expression*⟩ ::= ⟨*list_expression*⟩ ⟨*BINARY_LIST_OP*⟩ ⟨*addition_expression*⟩
| ⟨*addition_expression*⟩

⟨*addition_expression*⟩ ::= ⟨*addition_expression*⟩ ⟨*ARITMETIC_OP_ADDITIVE*⟩
⟨*multiplication_expression*⟩
| ⟨*multiplication_expression*⟩

⟨*multiplication_expression*⟩ ::= ⟨*multiplication_expression*⟩ ⟨*ARITMETIC_OP_MULTIPLICATIVE*⟩
⟨*simple_value*⟩
| ⟨*simple_value*⟩

⟨*simple_value*⟩ ::= ⟨*constant*⟩
| ⟨*IDENTIFIER*⟩
| ⟨*ARITMETIC_OP_ADDITIVE*⟩ ⟨*simple_value*⟩
| '!' ⟨*simple_value*⟩
| ⟨*UNARY_LIST_OP*⟩ ⟨*simple_value*⟩
| '(' ⟨*expression*⟩ ')'
| ⟨*function_call_expression*⟩

⟨*variable_declaration_statement*⟩ ::= ⟨*SIMPLE_TYPE*⟩ ⟨*IDENTIFIER*⟩ ';'
| ⟨*SIMPLE_TYPE*⟩ ⟨*LIST_TYPE*⟩ ⟨*IDENTIFIER*⟩ ';'

⟨*constant*⟩ ::= ⟨*C_INTEGER*⟩ | ⟨*C_FLOAT*⟩ | ⟨*C_NIL*⟩

B Definições Regulares

Padrões	**Expressões Regulares**
DIGIT	[0-9]
CHARACTER	[a-zA-Z]
C_INTEGER	{digit}+
C_FLOAT	{digit}*.{digit}+
C_NIL	'NIL'
T_SIMPLE	int | float
T_LIST	list
IDENTIFIER	{character}|[_]({character}|{digit}|[_])*
UNARY_LIST_OP	'?' | '%'
BINARY_LIST_OP	':' | '<<' | '>>'
EQUALITY_OP	'==' | '!='
RELATIONAL_OP	'>' | '<' | '>=' | '<='
ARITMETIC_OP_-ADDITIVE	'+' | '-'
ARITMETIC_OP_-MULTIPLICATIVE	'*' | '/'
ASSIGN_OP	'='
LOGICAL_OP_AND	'&&'
LOGICAL_OP_OR	'||'
RW_FOR	'for'
RW_IF	'if' | 'else'
RW_RETURN	'return'
IO_READ	'read'
IO_WRITE	'write' | 'writeln'
DELIMITERS	'(' | ')' | '{' | '}' | ';' | ','
LIT_STRING	" .* "
EXCLAMATION_OP	'!'