Tradutores - Geração de Código Intermediário

Matheus Breder Branquinho Nogueira - 17/0018997

Universidade de Brasília mtbreder@gmail.com

1 Motivação

A disciplina de Tradutores tem por objetivo estudar os requisitos teóricos e práticos para a construção de tradutores de linguagens de programação. Desse modo, o projeto dessa disciplina é focado na construção de um tradutor para a linguagem C-IPL.

A linguagem C-IPL foi desenvolvida com o objetivo de facilitar o tratamento de listas em programas escritos na linguagem C. Portanto, foi introduzida uma nova primitiva de dados para listas assim como operações utilizando essa nova primitiva. Listas são estruturas de dados muito utilizadas, de fácil entendimento e também possuem um nível de complexidade baixo. Portanto, ao adicionar uma nova primitiva a linguagem C para o tratamento de listas assim como operações com esse novo tipo de dado, a construção de programas que utilizam desse tipo de estrutura é facilitada e agilizada.

Esse relatório descreve a implementação da análise léxica, sintática, semântica e geração de código intermediário da linguagem C-IPL.

2 Análise Léxica

Para realizar a análise léxica da linguagem, foram utilizadas expressões regulares para identificar as palavras da linguagem, como: operadores (aritméticos, lógicos, relacionais e atribuições), dígitos, comentários e comandos de controle de fluxo. Além disso, outros tokens também fazem parte da composição da linguagem e foi necessário avaliá-los também, como: parênteses, chaves, ponto-e-vírgula.

A identificação de erros léxicos da linguagem foi feita por meio da identificação da linha e da coluna onde se encontra o erro. Além disso, o erro é identificado por qualquer expressão que não esteja determinada pelas expressões regulares das palavras da linguagem.

3 Análise Sintática

A análise sintática do projeto foi feita utilizando o Bison. E as regras da gramática estão definidas no fim do relatório na Seção 8.1.

O Bison, seguindo as especificações do projeto, gera um analisador sintático LR(1), segundo a diretiva %define lr.type canonical-lr.

3.1 Tabela de simbolos

Para a criação da tabela de símbolos foi utilizada a estrutura abaixo. Os dados armazenados nela fornecem a identificação do símbolo, seu tipo, sua declaração que será útil para a análise semântica, a posição tanto da linha quanto da coluna em que o símbolo se encontra no código, seu escopo, o número de parâmetros e uma variável para salvar o registrador que será utilizado na geração do código intermediário.

O cálculo do escopo não funcionaria se apenas fosse incrementado com a abertura de chaves e decrementado com fechamento de chaves, pois dois escopos diferentes seriam entendidos como o mesmo em locais distintos do código. Portanto, para realizar essa diferenciação do escopo, ele é incrementado a partir do último maior valor de escopo. Portanto, ele pode ser decrementado normalmente, porém, ao adicionar um novo escopo, o valor desse novo escopo é calculado de forma que sempre será um valor distinto.

```
typedef struct Symbol {
        char id[100];
        char type[100];
        char decl[100];
        int line;
        int column;
        int scope;
        int params;
        int reg;
} Symbol;
```

3.2 Árvore sintática

De modo semelhante, a estrutura abaixo foi utilizada para a criação da árvore de derivação. Os dados armazenados na struct informam a sua identificação, ou seja, qual regra da gramática é feita naquele nó, o seu tipo, o tipo do seu retorno, e um dado para salvar o a linha de código intermediário que será gerada durante a análise.

4 Análise Semântica

A análise semântica foi feita utilizando o Bison e as estruturas já criadas na fase da análise sintática, como a tabela de símbolos, a árvore de derivação.

A análise semântica é a análise lógica do código, sendo assim, erros como a falta da uma função main, quantidade errada de parâmetros de uma função e declaração duplicada de variáveis ou funções, devem ser percebidos durante essa fase da análise.

```
typedef struct TreeNode{
    struct TreeNode* t1;
    struct TreeNode* t2;
    struct TreeNode* t3;
    struct TreeNode* t4;
    char value[100];
    int type;
    int ret;
    char tac_code[1000];
} TreeNode;
```

4.1 Conversão de Tipos

Para realizar a conversão de tipos, toda vez que um operação ou uma atribuição é feita, dentro da própria regra é feita a verificação de tipos. Os tipos dos nós filhos são comparados, e caso uma conversão de tipos seja necessária, o novo tipo calculado é armazenado no nó pai.

4.2 Chamada de funções e variáveis

Ao ser feita uma chamada ou uma declaração de função ou variável, é feita uma busca na tabela de símbolos para verificar se o nome passado já foi declarado ou não.

4.3 Regras de Escopo

A pilha de escopo é a estrutura utilizada para avaliar quais escopos podem ser acessados. Ná prática, é apenas um vetor com os valores dos escopos,

Quando um token "{" é encontrado, o escopo é incrementado e inserido na pilha, e de modo semelhante, quando um token "}" é encontrado, o escopo é decrementado e seu valor no vetor é zerado. Para realizar esse processo, foi criada uma função que é chamada sempre que é necessário fazer uma troca no escopo, seja para incrementar ou decrementar. Desse modo, os valores dos escopos ficam armazenado no vetor e o valor é atualizado sempre a partir do último maior escopo.

4.4 Passagem de Parâmetros

Para fazer a análise correta da passagem de parâmetros, toda vez que uma função é chamada, a quantidade de argumentos da chamada é comparada com a quantidade de argumentos da função que está sendo chamada. Isso é feito fazendo a comparação com a função existente na tabéla de símbolos. Além disso, para verificar se os tipos dos parâmetros estão corretos, na tabela de símbolos, os parâmetros são armazenados de modo a identificar que são parâmetros, e desse modo, na chamada de função a tabela de símbolos é verificada analisando os parâmetros que estão no mesmo escopo que aquela função.

4.5 Verificação da Função Main

Para fazer a verificação da existência de uma função main, obrigatória para a linguagem, toda vez que uma função é declarada, seu nome é verificado. Se o nome da função for "main", uma variável chamada "main_error", que possui o valor 1 inicialmente, é decrementada para zero. Caso contrário, se no programa não existir uma função main, após a análise semântica é emitido um erro.

5 Geração de Código Intermediário

A geração do código intermediário feita pelo tradutor é baseado no TAC (Three Address Code). O código de três endereços gerado pelo tradutor, que é um código sasembly, será executado pela máquina virtual de TAC descrita na referência.

Inicialmente o tradutor fará a análise léxica, sintática e semântica, e caso não apresente erros, será gerado o código intermediário. Esse código será apresentado em um arquivo no formato .tac com o mesmo nome do arquivo de entrada para o tradutor. E esse arquivo gerado poderá ser utilizado pela máquina virtual.

Para realizar essa etapa da execução do tradutor, foram utilizadas duas variáveis para armazenar o código da parte .table e .code do código intermediário. Essas variáveis são strings em que o código intermediário é concatenado ao longo da criação da árvore de abstração. A parte .table do código é gerada quando há uma declaração de variável, e a parte do .code é gerada nas operações necessárias para que o código seja executado corretamente.

5.1 Regras de Escopo

No TAC, variáveis com mesmo no devem ser distinguidas, e isso é feito concatenando o escopo ao nome da variável. Desse modo, caso variáveis distintas possuam o mesmo nome, serão diferenciadas pelo valor do seu escopo, e caso sejam do mesmo escopo, a análise semântica já indica a presença de um erro.

5.2 Conversão de Tipos

A conversão de tipos é feita na análise semântica. Desse modo, quando for necessário realizar a operação de conversão de tipos no TAC, durante a análise semântica, a linha de código TAC já é criada quando ocorre uma conversão de tipos.

5.3 Chamada de Função e Retorno

Para garantir que toda função possui pelo menos um retorno, no final da execução de cada função é inserido a instrução return. Além disso, para garantir que uma função não utilize palavras reservadas do TAC, foi concatenado o caracter "_" antes do nome de cada função.

5.4 Laços de Repetição e Condicionais

Para a função if e else, inicialmente é feito um jump para a parte do código em que a condição é testada, e na sequência é feito um brnz que fará um branch para a parte do código correspondente ao condicional

Para um laço de repetição, no final de cada execução é feita a análise do condicional por meio de um brnz para verificar a condição de parada e assim sair do laço de repetição.

5.5 Nova Primitiva List

A primitiva de lista é feita por meio de duas variáveis, a primeira é a lista em si, que é inicializada com valores nulos, e outra variável que armazena o seu tamanho.

5.6 Operações com Listas

A realização das operações sobre listas é feita de modo que a função analisa o primeiro dado da lista, realiza a operação sobre ele, e acessa o endereço do próximo elemento da lista. Assim, toda a lista é percorrida até que seja encontrada o valor NIL, indicando o fim da lista.

6 Arquivos de Teste

Os arquivos de teste estão disponíveis na pasta tests. Dentro dessa pasta existem quatro programas de teste diferentes. Os arquivos de teste que têm a sua análise correta são:

1. teste-correto1.c
2. teste-correto2.c

Enquanto que os arquivos de teste que têm sua análise incorreta são:

1. teste-incorreto1.c.
2. teste-incorreto2.c.

O arquivo de teste incorreto teste-incorreto1.c apresenta erros sintáticos e semânticos. Apresenta um erro semântico na linha 14 e coluna 5 por chamar uma função não declarada, e outro erro semântico na linha 17 e coluna 10 por fazer comparação entre int e int list. O arquivo ainda apresenta um erro sintático, um na linha 23 e coluna 12 por ter ID não esperado.

Já o arquivo de teste incorreto teste-incorreto2.c apresenta quatro erros semânticos. Um erro na linha 14 e coluna 5 por chamar uma varíavel ainda não declarada. Outro erro na linha 19 e coluna 9 por chamar uma função não declarada. Outro erro na linha 19 e coluna 7 por tentar fazer um assign de uma função não declara, e por isso não tem tipo, a uma variável int. E um último erro na linha 27 e coluna 9 por chamar uma função com o número incorreto de argumentos.

7 Compilação e Execução

O projeto do analisador léxico foi executado e compilado nas seguintes configurações de sistema:

1. GCC version 11
2. Flex version 2.6.4
3. Ubuntu 20.04 LTS
4. Kernel : 4.4.0-19041-Microsoft

Para compilar o programa digite a seguinte instrução no terminal:

```
$ flex -o src/lex.yy.c src/flex.l
$ bison -o src/syntax.tab.c -d src/syntax.y
$ gcc -o tradutor -g src/syntax.tab.c src/lex.yy.c
lib/structures.c -Wall -Wpedantic
```

Para executar os casos de teste digite as seguintes instruções no terminal:

```
$ ./tradutor tests/teste-correto1.c
$ ./tradutor tests/teste-correto2.c
$ ./tradutor tests/teste-incorreto1.c
$ ./tradutor tests/teste-incorreto2.c
```

Para executar o código TAC é necessário instalar o executável TAC presente em https://www.github.com/lhsantos/tac, e após executar um teste correto, utilizar o arquivo gerado no formato .tac como entrada para o TAC:

```
$ ./tac nome_do_arquivo.tac
```

References

1. A.V.Aho, M.S. Lam, R. Sethi, nd J.D. Ullman. Compilers: Principles, Techniques, & Tools. Pearson/Addison Wesley, 2nd edition, 2007.
2. Claudia Nalon. Trabalho Prático - Descrição da Linguagem (C-IPL) https://aprender3.unb.br/mod/page/view.php?id=464034. Acessado em 10/08/2021.
3. W. Estes. Lexical Analysis With Flex, for Flex 2.6.2 https://westes.github.io/flex/manual. Acessado em 19/08/2021.
4. R. Corbett and GNU Project Team. Manual GNU Bison https://www.gnu.org/software/bison/manual. Acessado em 16/09/2021.
5. Luciano Santos. Simples three address code virtual machine. https://www.github.com/lhsantos. Acessado em 25/10/2021.

8 Anexo

8.1 Gramática da linguagem

start → prog
prog → prog prog_block | prog_block
prog_block → decl | funct_decl
decl → TYPE ID ';' | TYPE LIST ID ';'
funct_decl → funct '(' params ')' '{' block '}'
funct_decl → funct '(' ')' '{' block '}'
funct → TYPE ID | TYPE LIST ID
params → params ',' param | param
param → TYPE ID | TYPE LIST ID
block → block statement | block decl
block → statement | decl
statement → expression ';' | ass_op ';'
statement → '{' block '} | flow_control
ass_op → id ASSIGN_OP operation
expression → operation | read | write
operation → log_op
log_op → log_op LOG_OP rel_op | rel_op
rel_op → rel_op REL_OP list_op | list_op
list_op → add_op LIST_BIN_OP list_op
list_op → add_op LIST_CONSTRUC list_op
list_op → add_op
add_op → add_op ADD_OP mul_op | mul_op
add_op → ADD_OP mul_op
mul_op → mul_op MUL_OP op_un | op_un
op_un → LIST_UN_OP op_un | value
write → WRITE '(' operation ')'
read → READ '(' id ')'
flow_control → if_else_statement | for_statement
flow_control → return_statement
if_else_statement → IF_RW '(' operation ')' statement
if_else_statement → IF_RW '(' operation ')' statement ELSE_RW statement
for_statement → FOR_RW '(' ass_op ';' operation ';' ass_op ')' statement
return_statement → RETURN_RW operation
value → id | function_call | '(' operation ')'
value → INT | FLOAT | NIL_RW | STRING
function_call → ID '(' function_params ')' | ID '(' ')'
id → ID
function_params → function_params ',' operation | operation

8.2 Léxico da linguagem

Table 1. Léxico da linguagem

Padrão	Descrição	Exemplo
String	Caracteres entre ” ”	”hello”
Add_op	Operadores aritméticos de soma	+, -
Mult_op	Operadores aritméticos de multiplicaçã	*, /
Log_op	Operadores lógicos	&&, ||
Rel_op	Operadores relacionais	>, >=, <
Ass_op	Operadores de atribuição	=
List_bin_op	Operadores binários sobre listas	: , >>
List_un_op	Operadores unários sobre listas	%, :, !, ?
Type	Tipos de dados	int, float list
Integer	Tipo de número inteiro	10, 20, 30
Float	Tipo de número real	3.14
Digit	Caracteres de dígitos	”1”, ”2”, ”7”
Letter	Caracteres de letras	”a”, ”b”
Id	Identificadores iniciados com letras, seguidos ou não de dígitos ou underscores	”a1”, ”ab1”, ”ab1_”
Read	Função de entrada	read
Write	Função de saída	write, writeln
New_line	Quebra de linha	”\r”,”\n”
Nil_rw	Constante para listas	NIL
Reserved_words	Palavras reservadas da linguagem	if, else, for, return
End	Caracter que indica fim do comando	”;”
Bracket	Caracteres de parênteses, colchetes ou chaves	”(”, ”[”, ”{”