Gerador de Código Intermediário - Tradutores

Adelson Jhonata Silva de Sousa - 18/0114913

[1] Universidade de Brasília
[2] Departamento de Ciência da Computação
180114913@aluno.unb.br

1 Motivação

Esta parte do trabalho de tradutores que aborda a integração de todas as partes anteriores do trabalho e a geração de código intermediário e tem como objetivo aplicar na prática os conceitos teóricos vistos em [ALSU06] e estudados até o momento na matéria.

Utilizando um subconjunto da linguagem C e adicionando uma nova primitiva de dados do tipo *list* temos uma nova linguagem conforme descrição da linguagem em [Nal]. Essa nova linguagem visa facilitar o tratamento de listas com operações necessárias para essa nova primitiva, como por exemplo, operações de *filter, map, header e tail* da lista.

A lista é uma das estruturas de dados mais importantes que existem, pois serve para armazenar os dados em sequência. Em C não existe uma primitiva do tipo lista. Temos que manipular vários elementos para ter disponível essa estrutura de dados. A nova primitiva *list* adicionada a linguagem vem para facilitar a construção de programas que necessitam de listas, já que com as operações e os construtores disponíveis não precisamos utilizar ponteiros para o próximo elemento da lista e outras operações que fazemos em C.

2 Descrição da análise léxica

A análise léxica é feita utilizando o gerador léxico FLEX [Est]. As regras da análise léxica estão detalhadas no arquivo scr/lexico.l. Dentro desse mesmo arquivo também estão os contadores de linha e coluna para identificação de erro e uma função *main* para leitura do arquivo de teste. Todas as regras utilizam expressões regulares (regex) para identificar os padrões e sequências e todas as regras estão especificadas com seus respectivos nomes na Tabela 1. Os *tokens* encontrados são exibidos na tela e já estão estruturados visando a tabela de símbolos utilizada na segunda parte deste trabalho, o analisador Sintático, e podem ser verificados no apêndice C e no arquivo **tabelaSimbolo.h** dentro do diretório *lib*. Todos os *tokens* e erros são exibidos juntamente com a linha e a coluna correspondente. Caso ocorra erro na análise léxica é exibida uma mensagem falando qual foi o erro.

Cada *token* tem uma linha, uma coluna, um símbolo e um valor de escopo. O símbolo é o identificador do *token*. Os *tokens* são enviados para o analisador sintático e tratados para serem inseridos na tabela de símbolos.

O analisador léxico irá analisar a linguagem fornecida, utilizando regras que contém tipos básicos de variáveis (int e float) além de declaração da lista (int ou float list), estrutura condicional *if, else*, laço de repetição *for*, operações aritméticas, operações lógicas e relacionais, e as operações da lista *map, filter, header, tail* e o construtor da lista.

3 Descrição da análise sintática

A análise sintática foi feita utilizando um arquivo descrito em bison e o programa *bison* [Bis], na versão 3.7.4. o bison é um programa que gera um autômato de pilha a partir de uma gramática anotada dada como entrada, a linguagem da gramática é passível de análise LR1.

No topo do arquivo foi definida a regra **%define lr.type canonical-lr** para que a derivação seja feita utilizando o autômato LR(1) canônico. LR(1) é ascendente, ou seja, de baixo para cima, e a árvore é gerada do final até o início.

O desenvolvimento do projeto está sendo feito seguindo algumas instruções descritas em [Aab].

3.1 Tabela de Símbolos

A tabela de símbolos é responsável por armazenar todos os símbolos durante a leitura da entrada. São armazenadas em um vetor as informações/tokens necessários para a utilização na análise semântica, como por exemplo variáveis declaradas, chamadas de função, escopo das variáveis, entre outros. Uma parte do código da tabela de símbolos está descrita no Apêndice C.

Os *tokens* recebidos da análise léxica são tratados e inseridos na tabela de símbolos sempre que é declarada uma variável ou função.

Além do identificador de cada *token* também é armazenada a linha, coluna, o tipo e o escopo em que se encontra.

Na Figura 1 está a representação gráfica da tabela de símbolos do arquivo testeCerto.c.

Figura 1. Tabela de Símbolos.

Tabela de Simbolos

Simbolo	Tipo Entrada	Escopo	Linha	Coluna
x	Variavel	1	2	9
y	Variavel	1	3	11
a	Variavel	1	4	11
m	Variavel	1	6	9
m	Variavel	2	8	13
m	Variavel	3	10	17
asd	Variavel	3	11	17
abc	Variavel	1	21	9
main	Funcao	0	1	5

3.2 Árvore Sintática Abstrata

A árvore sintática abstrata representa o fluxo do parser bottom-up da gramática descrita. Onde cada nó representa um não terminal e seus filhos representam os terminais, que são os símbolos. Cada nó tem um ponteiro apontando para o próximo nó e pode ter ou não filhos.

Assim podemos percorrer a árvore em profundidade e verificar se está tudo de acordo com o descrito no programa.

Ao finalizar o parser da gramática e do input, a árvore é exibida graficamente se não tiver nenhum erro, se tiver erro a árvore não é mostrada e mostra apenas a tabela de símbolos.

4 Descrição da análise semântica

Utilizando as partes anteriores desse projeto, como a análise léxica e sintática, entramos agora na análise semântica.

A análise semântica consiste em analisar e detectar todos os possíveis erros referentes à declaração e tipos de variáveis e funções, visando a perfeita execução do programa. Por exemplo um programa sem a função principal (função *main*) não deve ser executado pois é um requisito obrigatório ter a função principal. Isso é analisado e tratado na análise semântica.

Algumas dos problemas detectados na análise semântica são:

- Detecção da Função Principal (*main*);
- Detecção de variável e função duplicada, redeclarada ou não declarada;
- Conversão de tipos (Coerção/*Cast*) entre variáveis e/ou funções;
- Quantidade de parâmetros de uma função;
- Verificação do tipo de retorno de uma função;

Se um código passa com sucesso na análise semântica então ele está pronto para ser gerado o código intermediário e ser executado.

Uma das funções desenvolvidas é a responsável por verificar se existe uma função principal (*main*), percorrendo toda a tabela de símbolos e verificando se existe um símbolo *main* no escopo 0 e se o tipo de entrada é função. Se não atender esses requisitos é mostrado um erro semântico na tela informando que não existe uma função *main* no programa.

Foram desenvolvidas funções que são responsáveis por verificar se uma variável ou função está duplicada, redeclarada ou não foi declarada. As funções percorrem toda a tabela de símbolos e árvore sintática abstrata, comparando os símbolos e os escopo das variáveis/funções para fazer a verificação. Quando é encontrada uma variável/função duplicada, redeclarada ou não declarada é mostrado um erro semântico na tela informando a linha, coluna e identificador do erro.

A regra de escopo foi desenvolvida através da interação entre o escopo léxico e sintático. A alteração do escopo é feita toda vez que é encontrado um "{"ou "}". Quando encontrado um "{"o escopo aumenta, ou seja, é incrementado e todas as variáveis declaradas dentro vai receber esse novo escopo e só podem ser

usadas nesse escopo atual. Quando encontrado um "}"o escopo diminui e todas as variáveis declaradas dentro do escopo anterior não são mais utilizáveis, a não ser que sejam declaradas novamente neste escopo atual. A estrutura utilizada para controlar o escopo foi um vetor global para fazer a interação com os analisadores léxico e sintático e a cada novo escopo é a adicionado no vetor e a variável escopoAtual é atualizada.

Os tipos e conversão de tipos são todas anotadas na árvore informando o tipo e a conversão (*cast*) feito. Nas regras descritas no analisador sintático é chamada uma função responsável por fazer toda a conversão necessária, nas regras de expressão, exp, atribuição, opMultDiv, opSomaSub, entre outras a função castDeTudo é chamada e passado como parâmetros o tipo, o lado direito e o lado esquerdo. A função recebe os parâmetros e verifica os tipos de cada lado e o tipo passado. Se o tipo é "INT"e o tipo do lado direito é "FLOAT"então é feita uma conversão chamada *float_para_int*. Algumas das conversões feitas são:

- Int para Float;
- Float para Int;
- List para List, onde uma lista só pode unificar com outra lista;
- List para Nulo;

5 Geração de Código Intermediário

A geração de código intermediário é feita seguindo a documentação disponibilizada pelo autor do interpretador de código intermediário [San]. O código intermediário é um código de três endereços descritos na linguagem *assembly*.

O programa irá iniciar a análise léxica, análise sintática, logo após a análise semântica e, se nenhum ocorrer durante as análises, por fim a geração de código intermediário. A saída do programa é um arquivo gerado com extensão .tac pronto para ser executado e interpretado pelo interpretador de código intermediário.

O TAC possui duas seções extremamentes importantes que é a *.table* e a *.code*, onde na *.table* é onde basicamente fica a tabela de símbolos, ou seja, todas as variáveis e constantes declaradas no programa ficam declaradas nesta parte do código. A *.code* é onde estão declaradas as expressões e funções do programa.

5.1 Escopo, cast e variáveis

Para diferenciar as variáveis e seus escopos, durante a geração do código intermediário cada variável e função será nomeada com seu ID e Escopo correspondente, como por exemplo x0, x1, x2, etc.

Variáveis temporárias utilizadas durante a execução do código são utilizadas em ordem crescente e estão todas presentes na seção *.code*

A conversão de tipos *(cast)* é feita durante a análise semântica e anotada na árvore abstrata quando necessário. Durante a geração do código intermediário a verificada na árvore se é necessária a conversão de tipos e se necessária é executada traduzindo para a linguagem reconhecida do TAC, isso tudo é feito antes da operação ocorrer.

5.2 Entrada e Saída

Para a entrada de dados foram as usadas as instruções *scani* para variáveis do tipo int e scanf para variáveis do tipo *float*. O programa verifica a cada chamada de entrada o tipo da variável que irá salvar a entrada.

Para saída de dados foram definidas dentro da seção .code uma série de instruções e funções em assembly responsável por imprimir na tela a saída do programa. As instruções e funções estão descritas no apêndice D.

6 Descrição dos arquivos testes

Os arquivos de testes desenvolvidos para a aplicação estão dentro da pasta *tests*.

Os arquivos testeCerto.c e testeCerto2.c são os testes que não contém nenhum erro léxico, sintático ou semântico.

Os arquivos testeErrado.c e testeErrado2.c, contém erros sintáticos e semânticos em sua execução.

testeErrado.c

- Linha: 5 - Coluna: 5 - Identificador: f - Erro Semantico - Funcao ja declarada!!!
- Linha: 23 - Coluna: 13 - Identificador: r - Erro Semantico - Variavel nao declarada!!!
- Linha: 26 - Coluna: 18 - Identificador: asdasd - Erro Semantico - Variavel nao declarada!!!
- Linha: 26 - Coluna: 9 - Identificador: asdasd - Erro Semantico - Variavel nao declarada!!!
- Linha: 32 - Coluna: 9 - Identificador: faaaa - Erro Semantico - Funcao nao declarada!!!
- Linha: 33 - Coluna: 5 - Identificador: c - Erro Semantico - Variavel nao declarada!!!
- Linha: 35 - Coluna: 12 - Token: : - Erro: syntax error, unexpected CONSTRUTOR_LISTA, expecting ID
- Linha: 37 - Coluna: 13 - Token:) - Erro: syntax error, unexpected FECHA_PARENTESES

testeErrado2.c

- Linha: 7 - Coluna: 8 - Token: ; - Erro: syntax error, unexpected PONTOVIRGULA
- Linha: 5 - Coluna: 5 - Identificador: f - Erro Semantico - Funcao ja declarada!!!
- Linha: 11 - Coluna: 8 - Token: a - Erro: syntax error, unexpected ID, expecting PONTOVIRGULA
- Linha: 12 - Coluna: 8 - Token:) - Erro: syntax error, unexpected FECHA_PARENTESES
- Erro Semantico - Funcao Main nao encontrada!!!

7 Instruções para compilação e execução do programa

Dentro do diretório principal tem um arquivo *makefile* gera o autômata (em C) a partir do arquivo de entrada do flex e compila o .c, bastando apenas entrar com

o comando *make* no terminal. Para facilitar eu também coloquei dentro desse mesmo arquivo *make* todos os testes disponíveis. Para executar todos os testes basta entrar com o comando *make run* no terminal. Caso não queira utilizar o make poderá usar os seguintes comandos no terminal para executar o programa:

1. cd src/
2. bison -d sintatico.y
3. flex lexico.l
4. gcc-11 -g -c arvore.c -o arvore.o
5. gcc-11 -g -c tabelaSimbolo.c -o tabelaSimbolo.o
6. gcc-11 sintatico.tab.c lex.yy.c arvore.o tabelaSimbolo.o -g -Wall -o ../tradutor
7. cd ../
8. ./tradutor tests/nome_do_teste.c

O programa foi desenvolvido no ambiente Linux Ubuntu 20.04. Tentei me aproximar mais das versões descritas pela professora, mas teve algumas que não consegui. Então segue a configuração:

1. Kernel: 5.11.0-37-generic
2. gcc-11: gcc version 11.1.0
3. flex: 2.6.4
4. bison 3.7.4

Referências

[Aab] Anthony A. Aaby. Compiler construction using Flex and bison. https://dlsiisv.fi.upm.es/traductores/Software/Flex-Bison.pdf. Acessado em 06 de Setembro de 2021.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools.* Pearson/Addison Wesley, second edition, 2006.

[Bis] GNU Bison. The yacc-compatible parser generator. manual of bison. https://www.gnu.org/software/bison/manual/bison.html. Acessado em 28 de Agosto de 2021.

[Est] Will Estes. Lexical analysis with flex, for flex 2.6.2. https://westes.github.io/flex/manual/. Acessado em 09 de Agosto de 2021.

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

[San] Luciano Santos. Interpretador de código de três endereços. https://github.com/lhsantos/tac. Acessado pela última vez em 21/10/2021.

A Gramática

⟨*inicio*⟩ ::= ⟨*programa*⟩

⟨*programa*⟩ ::= ⟨*Lista de Declaração*⟩

⟨*Lista de Declaração*⟩ ::= ⟨*Lista de Declaração*⟩⟨*declaração*⟩ | ⟨*declaração*⟩

⟨*TIPO*⟩ ::= ⟨*TIPO_INT*⟩ | ⟨*TIPO_FLOAT*⟩ | ⟨*TIPO_LIST_INT*⟩ | ⟨*TIPO_LIST_INT*⟩

⟨*declaração*⟩ ::= ⟨*declaração de variável*⟩ | ⟨*declaração de função*⟩

⟨*declaração de variável*⟩ ::= ⟨*TIPO*⟩ ⟨*ID*⟩;

⟨*declaração de função*⟩ ::= ⟨*TIPO*⟩ ⟨*ID*⟩ (lista de parâmetros) { declarações }

⟨*lista de parâmetros*⟩ ::= ⟨*TIPO*⟩ ⟨*ID*⟩ , ⟨*lista de parâmetros*⟩ | ⟨*TIPO*⟩ ⟨*ID*⟩ |
vazio

⟨*declarações*⟩ ::= { ⟨*declarações*⟩ }
| ⟨*condicional*⟩
| ⟨*entrada*⟩
| ⟨*saída*⟩
| ⟨*for*⟩
| ⟨*return*⟩
| ⟨*expressão*⟩
| ⟨*expressão List*⟩

⟨*expressão*⟩ ::= ⟨*exp*⟩ ; | ⟨*ID*⟩ = ⟨*expressão*⟩ | ⟨*ID*⟩ = ⟨*NIL*⟩;

⟨*exp*⟩ ::= ⟨*expressao logica*⟩ | !exp ;

⟨*declaração list*⟩ ::= ⟨*TIPO*⟩ ⟨*ID*⟩;

⟨*return*⟩ :: = return expressão; | ;

⟨*for*⟩ ::= for (⟨*expressão*⟩; ⟨*expressão*⟩; ⟨*expressão*⟩) ⟨*declarações*⟩

⟨*condicional*⟩ ::= if(⟨*expressão*⟩) ⟨*declarações*⟩
| if(⟨*expressão*⟩) declarações else ⟨*declarações*⟩

⟨*entrada*⟩ ::= read(⟨*ID*⟩);

⟨*saída*⟩ ::= write(⟨*string*⟩ | ⟨*expressão*⟩); | writeln(⟨*string*⟩ | ⟨*expressão*⟩);

⟨*expressão list*⟩ ::= ?⟨*ID*⟩ | !⟨*ID*⟩ | ⟨*ID*⟩ = ⟨*ID*⟩ >> ⟨*ID*⟩ | ⟨*ID*⟩ = ⟨*ID*⟩ << ⟨*ID*⟩

⟨*NIL*⟩ ::= ⟨*NIL*⟩

⟨*expressao logica*⟩ :: = expressao logica OR exp
| expressao logica AND exp
| exp

⟨*expressao relacional*⟩ ::= ⟨*expressao relacional*⟩ ⟨*operacoes relacionais*⟩ ⟨*exp*⟩

⟨*operaçao soma_sub*⟩ ::= ⟨*operaçao soma_sub*⟩ OP_SOMA_SUB ⟨*exp*⟩;

⟨*operaçao mult_div*⟩ ::= ⟨*operaçao mult_div*⟩ OP_MULT_DIV ⟨*argumento*⟩;

⟨*argumento*⟩ ::= ⟨*ID*⟩
| ⟨*NUMERO*⟩
| (exp)
| ⟨*Chamada de Função*⟩

⟨*Chamada de Função*⟩ ::= ID (exp)

⟨*NUMERO*⟩ :: ⟨*INT*⟩ | ⟨*FLOAT*⟩

B Tabela de expressões Regulares

Tabela 1. Regras e expressões regulares utilizadas.

Nome	Expressão Regular
COMENTARIO	"//".*|"/*".*|"*/".*
ID	[a-zA-Z_][_a-z0-9A-Z]*
INT	[0-9]+
FLOAT	[0-9]+"."[0-9]+
TIPOS	int|float|"int list"|"float list"
OP_B_ARITMETICAS	"+"|-"|"*"|"/"
OP_LOGICAS	"&&"|"!"|"||"
OP_B_RELACIONAIS	"<"|"<="|">"|">="|"=="|"!="
CONTROLEFLUXO	if|else|for|return
ENTRADA	read
SAIDA	write|writeln
CONSTRUTOR_LISTA	":"
OP_LISTA	"?"|"%"
FUNCOES_LISTA	">>"|"<<"
STRING	\"[^\"]*\"|'[^']*'
NIL	NIL
PARENTESES	"("|")"
CHAVES	"{"|"}"
ATRIBUICAO	"="
VIRGULA	","
PONTOVIRGULA	";"
COLCHETES	"["|"]"
QUEBRALINHA	\n|\r
TABESPACO	[\t]+|[ ]+

C Estrutura da tabela de símbolos

```
#ifndef TABELA_SIMBOLO_H
#define TABELA_SIMBOLO_H

typedef struct TabelaSimbolo{
    int escopo;
    char simbolo[100];
    int linha;
    int coluna;
    char tipoEntrada[50];
    struct TabelaSimbolo* proximo;
} TabelaSimbolo;

TabelaSimbolo* insereSimbolo(TabelaSimbolo* id,
int escopo, char* simbolo, char* tipoEntrada,
int linha, int coluna, int parametros);

void mostraTabela(TabelaSimbolo* id);

void limpaTabela(TabelaSimbolo* id);

#endif
```

D Instruções de saída em assembly

```
write_fn:
        mov $500, 0
proximo:
        mov $499, #1
        mov $499, $499[$500]
        add $500, $500, 1
        print $499
        sub $499, $500, #0
        brnz proximo, $499
        return
writeln_fn:
        call write_fn, 2
        println
        return
```