

Relatório - Tradutor de C*

Nicholas Nishimoto Marques - 15/0019343

14/09/2020

1 Introdução e Área da Computação

A linguagem proposta para o trabalho é a C*, consistindo em uma extensão da linguagem C com a adição de operações sobre tuplas, onde seria possível declarar tuplas com tipos diferentes de dados, somar as tuplas (posição à posição), subtrair, em geral realizar operações básicas nas tuplas. Com isso seria possível otimizar muitos projetos de código, possibilitando arquiteturas de software novas e mais organizadas para os desenvolvedores, otimizando construções em Engenharia de Software.

A linguagem C* destina-se ao interesse do aluno na sub-área da computação de Engenharia de Software. É uma área que atualmente é responsável por projetar a arquitetura dos softwares, o modo como ele será implementado, os testes à serem feitos e a manutenção do mesmo. Tais implementações de estrutura de dados permitem uma maior flexibilidade de operações, organizando o código gerado, que é a ideia por trás das linguagens de alto nível.

As dificuldades que podem ser encontradas são fazer as operações em tuplas somando cada componente de forma certa e tratar erros que possam acontecer em cada caso, já que elas podem ter tipos diferentes de dados a cada posição.

Muitas vezes quando estamos fazendo um software nos deparamos com modelagens de estruturas que possuem vários atributos, como por exemplo uma struct em C para representar uma pessoa:

```
1 struct pessoa {  
2     char* nome;  
3     int idade;  
4     float salario;  
5 }
```

Listing 1: Estrutura de dados para uma pessoa em C

Porém, nessa abordagem não conseguimos fazer operações básicas como atribuir à uma tupla diretamente outra tupla, por exemplo, nesse caso para criar uma pessoa com nome João, idade 21 e salario 3000.0 teríamos que atribuir pra cada parte da variável um valor separadamente:

```
1 struct pessoa joao;  
2 joao.name = "Joao";  
3 joao.idade = 21;  
4 joao.salario = 3000.0;
```

Listing 2: Atribuições em estrutura em C

Na modelagem da linguagem C* é possível fazer operações com tuplas de forma direto, nesse caso a tupla joao seria:

```
1 tuple char* name, int idade, float salario joao;  
2 joao = ("Joao", 21, 3000.0);
```

Listing 3: Tuplas em C*

Facilitando no longo prazo operações com tuplas. Se quiséssemos por exemplo somar tuplas e obter resultados para cada componente, teríamos:

```
1 tuple char* name, int idade, float salario joao;  
2 tuple char* name, int idade, float salario maria;  
3 joao = ("Joao", 21, 3000.0);  
4 maria = ("Maria", 22, 5300.0);  
5 tuple result = joao + maria // podemos declarar os tipos da  
   tupla implicitamente  
6  
7 result.salario = 8300.0 // soma dos salarios das tuplas  
   envolvidas  
8 result.idade = 43 // soma das idades
```

Listing 4: Operações em tuplas em C*

2 Motivação para a escolha da linguagem/área: que tipos de problemas resolve

A área escolhida é a da Engenharia de Software, área que vem se ampliando cada vez mais e especializada em arquiteturas de códigos, modos de automatizar integrações e melhores maneiras de se fazer testes para um projeto de software. O tipo de problema que a linguagem proposta resolve se dá nessa área, que consiste em otimizar interações de dados que necessitam dessa estrutura (como por exemplo operações de vetores físicos, ou mesmo para criação de objetos de uma classe, por exemplo, um aluno seria um tupla: ("João", 16, "masculino")).

Isso inclui também problemas de operações sobre essas tuplas, como por exemplo a soma de vetores (somar cada atributo da tupla), ou então fazer a média das notas por idade dos alunos (fazer média de todas as notas de alunos que tem o atributo idade da tupla iguais), etc.

Para isso, será facilitado conforme mostrado na listagem 4 o processo de somar tuplas, subtrair tuplas, fazer assign de tuplas, etc. Um exemplo claro disso é gerar um teste por exemplo que compare em uma query de banco de dados se o usuário encontrado é o usuário procurado. Suponha que eu faça uma query e encontre um usuário, representado pela seguinte tupla (após convertido da query pra linguagem C*):

```
1 joao = User.find(1)//(1,"Joao", 21, 3000.0);  
2 // id, nome, idade, salario
```

Listing 5: tupla de usuario

Nesse caso podemos fazer um teste hipotético de apenas comparar se a tupla que representa João é a tupla que foi retornada da query.

```
1 joao = User.find(1)//(1,"Joao", 21, 3000.0);
```

```
2 expect(joao).to equal((1, "Joao", 21, 3000.0))
```

Listing 6: teste para tupla de usuario

Onde o expect seria um comando de verificar se o argumento passado é igual ao argumento dentro do equal. Isso facilita bastante pois não temos que comparar posição a posição da tupla para saber se ambas as partes são iguais.

3 Gramática da Linguagem

A gramática proposta se baseia na gramática do ANSI C [3], que consiste em gramáticas de termos e predicados [4] e dos tipos e expressões. A gramática incluiu as expressões a mais necessárias para as tuplas, representadas pelos predicados tuple.

Todos os terminais estão em maiúsculo, sequências entre chaves { } são indicativos de possíveis repetições, e sequências entre colchetes [] são indicativos de opcionalidade, adicionados à gramática a lógica de escrita e leitura da linguagem.

```
1 LETRA  [A-Za-z]
2 DIGITO [0-9]
3 INT    "-"?{DIGITO}+
4 FLOAT  "-"?{DIGITO}+("."{DIGITO}+)*
5 ID     {LETRA}({LETRA}|{DIGITO})*
6 BOOL   "true"|"false"
7 TIPO   "float"|"int"|"char"|"bool"|"void"
8 TUPLE  "tuple"
9 CONDICAOES "if"|"else"
10 LACOS   "while"|"for"
11 RETORNO "return"
12 OP_ARITM  "+"|"-"|"*"|"/"
13 OP_COMP   "=="|"!="|"<="|"<"|">="|>"
14 OP_LOG    "&&"|"||"
15 OP_ASSIGN "="
16
17 programa:
18     declaracoes
19
20 declaracoes:
21     declaracoes declaracao
22 | declaracao
23
24 declaracao:
25     var_decl
26 | TUPLE declaracao_tupla
27 | func_decl
28
29 declaracao_tupla:
30     TIPO ',' declaracao_tupla
31 | var_decl
32
33 var_decl:
34     TIPO ID ';'
35
```

```

36 func_decl:
37     TIPO ID '(' parm_tipos ')' ';
38 | TIPO ID '(' ')' ';
39 | TIPO ID '(' parm_tipos ')' '{ cod_blocks }' ';
40 | TIPO ID '(' ')' '{ cod_blocks }' ';
41
42 parm_tipos:
43     parm_tipos TIPO ID
44 | parm_tipos TIPO ID '[' ']'
45 | TIPO ID ','
46 | TIPO ID
47 | TIPO ID '[' ']'
48 | TUPLE ID
49
50 cod_blocks:
51     cod_blocks cod_block
52 | cod_block
53
54 cod_block:
55     "if" '(' expressao_logica ')' '{ cod_blocks }'
56 | "if" '(' expressao_logica ')' '{ cod_blocks }' "else" '{
57     cod_blocks }'
58 | LACOS '(' expressao_logica ')' '{ cod_block }'
59 | RETORNO ';
60 | RETORNO termo ';
61 | RETORNO '(' expressao ')' ';
62 | assign ';
63 | print
64 | ID '(' expressao ')' ';
65 | ID '(' ')' ';
66 | scan '(' ID ')' ';
67 | declaracoes { $$ = $1 }
68
69 assign:
70     ID OP_ASSIGN expressao
71 | ID '[' INT ']' OP_ASSIGN expressao
72
73 expressao:
74     op_expressao
75 | '(' expressao ')'
76
77 expressao_logica:
78     OP_LOG op_expressao
79 | '!' op_expressao
80 | op_expressao OP_COMP op_expressao
81 | '(' op_expressao ')'
82 | op_expressao
83
84 op_expressao:
85     op_expressao OP_ARITH termo | termo
86
87 termo:
88     ID | INT | FLOAT | ID '[' INT ']'

```

```

88
89 scan:
90     SCAN '(' ID ')',
91
92 print:
93     PRINT '(' termo ')' ';' | PRINT '(' palavra ')' ';'

```

Listing 7: Gramática da Linguagem

4 Descrição breve da semântica da linguagem.

Palavras reservadas da linguagem: int float void tuple if else while return

Símbolos: + - * / >= < <= != () { } [] ;

A palavra reservada int remete à declaração de uma variável inteira.

A palavra reservada float remete à declaração de uma variável ponto flutuante.

A palavra reservada void remete à declaração de uma variável sem tipo.

A palavra reservada tuple remete à declaração de uma tupla, seguida pelos tipos das variáveis da tupla (exemplo: tuple int,float remete à uma dupla de inteiro com ponto flutuante).

A palavra reservada if seguida de uma expressão booleana remete à uma condição em que, se a expressão booleana que segue o if for verdadeira, então todo bloco de código seguinte ao mesmo será executado.

A palavra reservada else remete à um trecho de código que será executado caso o trecho de código presente num bloco if anterior não seja executado.

A palavra reservada while seguida de uma expressão booleana remete à uma condição em que, enquanto a expressão booleana for verdadeira, o bloco de código seguinte continuará sendo executado.

A palavra reservada return remete à um retorno da chamada de uma função, que é o parâmetro seguinte à palavra return.

Regras de Escopo:

Os escopos na linguagem são definidos a partir da delimitação por chaves após palavras reservadas que indicam um bloco (todas as da sintaxe de cod block ou func decl). Por exemplo:

```
void id { // escopo de bloco (função) }
```

```
if (expressão) { // escopo de bloco }
```

Sobre tuplas podem ser feitas todas as operações básicas que os tipos que ela contém permitem. Por exemplo: tuple int, float person = (20, 500.50)

```
tuple int, float person2 = (30, 150.10)
```

```
permitiria person[0] + person2[0] retorna 50 ou ainda
```

```
person + person2 retorna (50, 650.60)
```

Todas as outras operações e desvios condicionais da linguagem se assemelham ao da linguagem

C.

5 Análise Léxica

5.1 Implementação e Funcionamento do programa tradutor

Será feito um programa tradutor para tal linguagem. O Programa tradutor irá funcionar seguindo as etapas de tradução de Análise Léxica, Análise Sintática, Análise Semântica e Gerador de Código intermediário, como citado em Compilers: Principles, Techniques and Tools [1].

Na primeira etapa será implementado o analisador léxico. Para essa parte, será utilizado o programa Flex [5] para gerar um programa analisador léxico na linguagem C, a partir da escrita do arquivo .l (lex language) onde é descrita a gramática de C* e os padrões léxicos.

A seguir podem ser visto algumas definições importantes que serão importantes na construção dos tokens no processo de análise léxica, como as definições de tipo (float, int, char, bool, void, tuple) e operações, por exemplo.

```
1      %x COMENTARIO
2      %x INTEIRO
3      DELIM [ \t]
4      ENTER [\n]
5      ESPACO {DELIM}+
6      SEPARADOR [;{}() ,\[\]]
7      ASPAS [ '\" ]
8      LETRA [A-Za-z]
9      DIGITO [0-9]
10     INT  "-"?{DIGITO}+
11     FLOAT  "-"?{DIGITO}+("."{DIGITO}+)*
12     ID {LETRA}({LETRA}|{DIGITO})*
13     VAR_DECL {ID}[''({DIGITO})+'']
14
15     /* keywords */
16     BOOL "true"|"false"
17     TIPO "float"|"int"|"char"|"bool"|"void"
18     TUPLE "tuple"
19     CONDICoes "if"|"else"
20     LACOS "while"|"for"
21     RETORNO "return"
22     /* COD_BLOCK IF '(' EXPRESSION ')' COD_BLOCK [ else COD_BLOCK] */
23     OP_ARITM  "+"|"-"|"*"|"/"
24     OP_COMP  "=="|"!="|"<="|"<"|">="|>"
25     OP_LOG  "&&"|"||"
26     OP_ASSIGN  "="
27     SCAN "scan"
28     PRINT "print"
```

Listing 8: Definições dos lexemas

A partir desse código em Lex, com a execução do programa flex um analisador léxico em C é gerado, capaz de identificar os lexemas da linguagem C* e converter nos tokens.

Ele identifica o texto escrito na linguagem C* e separa essas entradas em tokens, podendo ser tipos (declarações de tipos, float, int), ID (caso de nome de funções e variáveis, operadores (por exemplo o =, operador de atribuição).

5.2 Tratamento de Erros

Para o tratamento de erros, em um primeiro momento, será implementado um identificador de erros léxicos (padrões que não pertencem à gramática de linguagem, símbolos como @, , etc.)

Com a linguagem lex isso fica bem fácil de ser implementado, dado que se nos padrões de formações dos tokens o lexema não se encaixar em nenhum, ele será um erro léxico, que pode ser traduzido pela regra "." do arquivo lex:

```
1  . {
2      errors++;
3      strcpy(err[error_index].type, "nsym");
4      strcpy(err[error_index].msg, "Símbolo n o pertence
gram tica");
5      strcpy(err[error_index].sym, yytext);
6      err[error_index].line = lin;
7      error_index++;
8  }
```

Listing 9: Regra .

Esse padrão basicamente, definido pela expressão regular "." [7], significa qualquer lexema. Como ela se encontra no final do arquivo, terá a menor prioridade possível, logo tudo que não se encaixar em uma regra da linguagem cairá nessa regra, onde serão feitos os tratamentos de erro.

Na figura 4 vemos que, ao cair nessa regra ".", aumentamos o contador de erros e o índice do erro, atualizando também a linha em que ele ocorre (que é identificada pois a cada quebra de linha na linguagem aumentamos o índice da linha, visto na regra ENTER). Uma estrutura denominada lerror (lexical error) grava o tipo do erro (nesse caso nsym, ou no symbol, representando que não há esse lexema na linguagem), a mensagem de erro a ser exibida e qual o caracter, salvo na variável yytext, nativa do programa flex. Esse armazenamento será feito (ao invés do report imediato) para se organizar melhor os erros (por tipos) na hora de printar os erros.

Um vetor de erros (lerror) é gerado, e a cada erro identificado adiciona-se ele ao vetor de erros, exibido no final da análise léxica do arquivo da linguagem.

5.3 Funções Introduzidas

As funções introduzidas no código do analisador léxico gerado são mostradas a seguir. Basicamente temos a estrutura de erro e funções para o tratamento e para mostrar tais erros.

```
1  struct lerror {
2      char type[50];
3      char msg[100];
4      char sym[SYM_SIZE];
5      int line;
6  };
7
8  struct lerror err[SYM_TABLE_SIZE];
9
10 struct symbol_item {
11     char sym[SYM_SIZE];
12     int position;
13     int scope;
14 };
```

```

15
16     struct symbol_item symbol_table[SYM_TABLE_SIZE];
17
18     void printErrors(){
19         printf("Erros encontrados:\n");
20         int i;
21         for(i=0; i<errors; i++){
22             printf("\nErro numero %d\n", i+1);
23             printf("Simbolo: %s\n", err[i].sym);
24             printf("%s\n", err[i].type);
25             printf("%s\n", err[i].msg);
26             printf("Linha: %d\n", err[i].line);
27             printf("\n");
28         }
29     };

```

Listing 10: Funções introduzidas no analisador léxico

A variável `errors` denota a quantidade de erros encontrada na análise léxica, identificados como descrito na seção anterior. A variável `error_index` denota o índice do erro encontrado, utilizado para facilitar a inserção dos erros no vetor de erros e a identificação de cada um. As variáveis `col` e `lin` são variáveis que irão ser iteradas durante a análise léxica, para caso um erro seja encontrado seja possível retornar a linha e coluna onde esse erro foi encontrado, para facilitar o tratamento dos erros.

A struct `error` é a estrutura de dados utilizada para salvar os erros, contendo o tipo do erro (`type`), a mensagem de erro a ser exibida (`msg`), o símbolo encontrado no erro (`sym`) e a linha onde o erro foi encontrado (`line`). A função `printErrors` é chamada no final da análise léxica, onde o vetor denominado `err` (vetor de `errors`) é iterado e são mostrados todos os erros encontrados.

5.4 Descrição dos Arquivos de Teste

Alguns arquivos de teste foram gerados, denominados `teste.cstar`, `teste_error.cstar` e `teste2_error.cstar`. Os arquivos com `error` no nome contém alguns erros léxicos identificados na execução do programa: No arquivo `teste_error.cstar` temos:

```

1 int 5a;

```

Listing 11: `teste_error.cstar`

Um erro de padrão, onde uma variável está sendo declarada com número no começo. No arquivo `teste2_error.cstar` temos:

```

1 int ab@cax# = 0;
2
3 int returnAbacaxi() {
4     return abacaxi;
5 }

```

Listing 12: `teste_error2.cstar`

Onde temos um erro de símbolos que não pertencem à gramática.

No arquivo `teste.cstar` temos um programa que funciona normalmente, tendo os padrões encontrados na gramática:


```
1 tuple int, float alface;  
2 alface = (20, 500.40);  
3 float b[10];  
4 print(alface);
```

Listing 13: teste.cstar

Onde quando executado gera os tokens mostrados na figura 3.

6 Análise Sintática

6.1 Implementação e Funcionamento do Programa

A ideia geral da análise sintática é utilizar os tokens obtidos pelo analisador léxico (flex) anteriormente e a partir deles obter a árvore abstrata e a tabela de símbolos.

Para isso foi utilizado o Bison, que é uma implementação de parsers YACC, para adicionarmos as regras de formação sintática da gramática e com base nisso gerar a árvore.

Foram declarados os tokens na linguagem do bison [2] para utilização dos mesmos nas regras da gramática posteriormente. Alguns tokens de operações foram identificados com sua associatividade respectiva, right para associatividade à direita e left para à esquerda:

```
1  %token  BOOL
2  %token <tipo> TIPO
3  %token <str> IF ELSE
4  %token  LACOS
5  %token <str> RETORNO
6  %token <str> INT FLOAT TUPLE
7  %token <id> ID
8  %token <str> DIGITO LETRA
9  %token  SEPARADOR
10 %token  PRINT SCAN
11
12 %right <operador> OP_ASSIGN
13 %left <operador> OP_ARITHM
14 %left <operador> OP_LOG
15 %left <operador> OP_COMP
```

Listing 14: Tokens da linguagem

Para que o parser reconhecesse possíveis erros sintáticos foram definidas as regras da gramática para o Bison. As regras do Bison se assemelham à gramática descrita na seção 3.

6.2 Política de Erros

A ideia da política de erros nessa parte sintática é identificar regras que não existem na gramática, ou seja, regras que não podem ser derivadas de nenhuma forma na gramática.

Para isso faz-se o uso de todas as regras declaradas para identificar se eventualmente os tokens encontrados podem ser derivados. Caso isso não ocorra um erro sintático é gerado e adicionado ao vetor de erros sintáticos, assim como a linha que ele ocorre, que são mostrados no final da compilação do programa.

Para identificar a linha (principal política de tratamento de erros) basta pegar do analisador léxico a variável definida previamente `lin`, que já era incrementada na identificação de caracteres ENTER, que estará com a linha atual do token onde o erro ocorre.

Para mostrar esse erro basta sobrescrever a função `yyerror` do bison para mostrar o erro e onde ele ocorre. Essa função é chamada por outras funções auxiliares para fazer os tratamentos de erros.

6.3 Funções adicionadas

A função `yyerror` foi adicionada (sobrescrita) para mostrar o erro sintático e a linha onde ele ocorre, como mostrado na figura 9.

Funções para gerar a árvore e os nós da árvore sintática foram adicionadas. As definições dessas funções são mostradas abaixo:

```
1 node* ins_node(char* var_type, int node_type, char node_kind,
  node *left, node *right, char* node_val);
2 node* ins_node_symbol(char* var_type, int node_type, char
  node_kind, char* id);
3 void print_tree(node * tree, int prof);
```

Listing 15: Funções relacionadas à árvore sintática

A função `ins_node` recebe como parâmetros o tipo da variável (caso o nó seja uma declaração de variável ou função, podendo ser inteiro, float, char ou tuple), o tipo do nó (se é um símbolo e deve ser guardado na tabela de símbolos ou se é um nó regular), o kind do nó (que pode ser F para função, V para uma variável, C para um bloco de código, etc.), o valor do nó (no caso de variáveis e funções, o ID dela, e em outros casos algum identificador que seja necessário) e também ponteiros para o nó da direita e da esquerda, caso sejam regras que tenham descendentes. Essa função é a responsável por adicionar nós nas árvores.

A função `ins_node_symbol` se assemelha a `ins_node`, porém formatada especificamente para se o nó é um símbolo, chamando também a função de adicionar o nó na tabela de símbolos da forma correta.

A função `print_tree` é uma função que percorre a árvore em altura e printa todos os nós na formatação correta.

Além dessas funções referentes à árvore algumas funções referentes a tabela de símbolos foram adicionadas:

```
1 void add_to_s_table(char* id, char* var_type, int s_node_type,
  int scope);
2 void print_s_table();
```

Listing 16: Funções relacionadas à tabela de símbolos

A função `add_to_s_table` adiciona um símbolo à tabela, recebendo o id (que seria um identificador do símbolo, como o nome da variável), o tipo da variável (para conter na tabela se símbolos o tipo do retorno da função, ou o tipo da variável por exemplo, inteiro, float, char ou tuple), o tipo do nó (se é função, se é variável, etc.) e o escopo em que o símbolo foi encontrado (sendo um inteiro que representa esse escopo, com 0 sendo o escopo global).

A função `print_s_table` é a função que mostra a tabela de símbolos no final.

6.4 Construção da árvore sintática

A árvore sintática foi gerada a partir das definições das regras sintáticas:

Em cada identificação de regras temos 2 tipos, regras que levam em uma variável ou em um terminal.

Para a construção dessa árvore será utilizado uma estrutura de dados com 3 principais atributos (cada uma dessa estrutura será um nó da árvore). Cada regra de derivação será levada num nó da árvore.

Cada nó da árvore irá conter os terminais que são utilizados naquele nível da árvore e referências para os outros nós em outros níveis, que podem ser gerados a partir de argumentos que esses tokens no nó atual podem ter.

Além disso em cada nó iremos ter uma referência para tabela de símbolos (nula caso o token encontrado não tenha relação com a tabela de símbolos). A estrutura de dados que representa o nó pode ser vista a seguir:

```
1  typedef struct node {
2      char* var_type; //
3      char node_kind; // 'F' function, 'V' var, 'C' code_block..., '
4      D' declaration, 'E' expression
5      int node_type; // 'S' for symbol | 'R' for regular
6      struct node *left;
7      struct node *right;
8      struct node *middle;
9      char *val;
10 } node;
```

Listing 17: Estrutura de dados que representa um nó da árvore sintática

Nessa estrutura de dados vemos os ponteiros que apontam para os 2 principais atributos (nó da esquerda (left) e da direita (right)), um valor possível do nó (val), como por exemplo o ID da variável caso seja uma variável, o node_type, indicando se é um símbolo ou se é um nó regular, o node_kind, indicando mais especificamente o tipo de regra encontrado e o var_type, que indica se uma variável é inteira, float, char ou tuple, e caso seja função se a função retorna um desses tipos.

6.5 Tabela de Símbolos

A tabela de símbolos tem sua implementação dada por um hash map, onde cada chave do hash é uma entrada na tabela de símbolos, representada por um identificador que consiste num string que tem o nome do nó (no caso de uma variável, seria o ID da variável) e o escopo (nome da função), para identificar diferentes variáveis em diferentes escopos mas com mesmo nome. Um exemplo de entrada na tabela de símbolos pode ser vista a seguir:

```
1  # TABELA DE SÍMBOLOS
2  {
3      "person::main": {id: "person", var_type: "int", s_node_type: '
4      S', scope: 0}
5  }
```

Listing 18: Exemplo de uma linha na tabela de símbolos

onde vemos que a chave pra entrada na tabela é o identificador concatenado com o escopo da mesma.

A estrutura de dados que representa a tabela é mostrada a seguir:

```
1  typedef struct s_node {
2      char* id;
3      char* var_type;
4      int s_node_type;
5      int scope; // 0 = global
6      UT_hash_handle hh; // faz a estrutura ser um hash
```

```
7 } s_node;
```

Listing 19: Estrutura de dados de um componente da tabela de símbolos

6.6 Arquivos de teste

Foram utilizados mais 4 arquivos de teste, sendo 2 com sintaxes corretas na gramática e 2 com erros. Os arquivos corretos foram os seguintes:

teste_sintatico.cstar:

```
1 int i;  
2  
3 int sum(int a, int b){  
4     return (a + b);  
5 };
```

Listing 20: teste_sintatico.cstar

teste_sintatico2.cstar:

```
1     tuple int, char person;  
2  
3     int age(tuple p){  
4         return p[0];  
5     };  
6  
7     int gender(tuple p){  
8         return p[1];  
9     };  
10  
11     void printall(){  
12         print(person);  
13     };
```

Listing 21: teste_sintatico2.cstar

Os quais são interpretados corretamente pelo analisador sintático como sem erros. Além destes testes, ainda há outros testes para checar operações aritméticas, operações lógicas, blocos de código, entre outros, presentes na pasta de testes.

Os testes com erros foram os seguintes:

teste_sintatico_error.cstar:

```
1     int main(){  
2         person = 43;  
3     };  
4     tuple variavel;
```

Listing 22: teste_sintatico_error.cstar

Onde o erro consiste na não declaração dos tipos da tupla.

teste_sintatico_error2.cstar:

```
1     global = 0  
2     int main(){  
3         return 0;
```

```
4 }
```

Listing 23: teste_sintatico_error2.cstar

Onde os erro consiste na não declaração de uma função para inicializar o código.
teste_sintatico_error3.cstar:

```
1 int main(){  
2     while(){  
3         print a;  
4     };  
5 };
```

Listing 24: teste_sintatico_error3.cstar

Onde o erro consiste num while sem a expressão lógica.

7 Análise Semântica

A análise semântica consiste basicamente de 3 etapas:

- Verificar o uso de variáveis (se elas foram declaradas previamente)
- Verificar a redeclaração de variáveis (se existem variáveis no mesmo escopo com mesmo identificador sendo declaradas novamente)
- Verificar se os tipos dos operandos em operações aritméticas, de atribuição, de retorno de funções, parâmetros de funções e com tuplas são compatíveis (por exemplo, se estão sendo somados int com int, int com float, tuplas com mesmos tipos, etc.)

7.1 Implementação e Funcionamento do Programa

Para esta parte usou-se basicamente a implementação da árvore na etapa de análise sintática, utilizando-se algumas regras para determinar especificamente os 3 pontos citados anteriormente.

7.2 Política de Erros

A partir do explicitado, podemos inferir 3 tipos de erros que serão analisados nessa etapa: - Erros de utilização de variáveis não declaradas (denominados no código NO_DECLARATION_ERROR) - Erros de redeclaração de variável (denominadas no código REDECLARATION_ERROR) - Erro de tipos diferentes para operações (denominados no código TYPES_MISMATCH_ERROR)

Para tratamento desses erros, conforme eles ocorrerem serão reportados junto com a linha em que o erro acontece.

Nos casos de variáveis não declaradas sendo usadas e variáveis redeclaradas, junto com a linha do erro será mostrado o identificador da variável em que o erro ocorre.

No caso do erro de tipos, os tipos envolvidos na operação serão mostrados, além da linha em que o erro ocorre. Se o erro for um retorno com tipo diferente da função que ele se encontra, será dito especificamente que o erro de tipos é em um retorno.

Para chamadas de funções é verificado se os tipos condizem com os parâmetros da função e se o retorno é associado corretamente à retornos e variáveis. Esses parâmetros são associados quando uma função é adicionada à tabela de símbolos, de modo que a passagem de parâmetros é checada conforme a ordem que os parâmetros aparecem e seus tipos.

7.3 Funções adicionadas

Para essa etapa algumas funções relacionadas aos tratamentos de erros foram adicionadas: A primeira função adicionada foi a

```
1 void semantic_error(int error_type, char *msg)
```

Listing 25: Função para tratar erros semânticos

, que recebe como parâmetro o tipo do erro (um inteiro referente aos 3 tipos de erros) e uma mensagem adicional à ser mostrada referente ao erro, e tem como saída uma mensagem relatando o erro.

Outra função adicionada foi a

```
1 int types_match(char* t1, char* t2)
```

Listing 26: Função para tratar erros de tipos

, que recebe duas strings representando os tipos das variáveis encontradas e retorna se operações entre eles são válidas (por exemplo, se chamarmos `types_match("int", "float")`, ele retornará 1 (verdadeiro), pois podemos fazer operações entre esses 2 tipos).

Uma função auxiliar para busca de símbolos na tabela teve que ser adicionada, para que se encontrassem as variáveis, tipos delas, se haviam sido declaradas já ou não, para tratamento de todos os tipos de erro dessa etapa:

```
1 s_node* find_in_s_table(char* id)
```

Listing 27: Função para encontrar um nó na tabela de símbolos

Onde essa função retorna um nó que representa o símbolo da tabela, e nulo caso a variável não seja encontrada.

Para verificar erros na passagem de parâmetros foi criada a função:

```
1 int check_params(node *nd)
```

Listing 28: Função para verificar tipos passados

Ela verifica parâmetro a parâmetro da chamada da função se os tipos correspondem. Caso sejam diferentes ela levanta um erro de `TYPES_MISMATCH`, apontando a função que o erro ocorre. Caso o número de argumentos não corresponda também ao número de argumentos da função é apontado um erro de `WRONG_ARGUMENTS_NUMBER`, indicando que há essa discrepância.

7.4 Arquivos de Teste

Os arquivos atualizados se encontram na pasta `/tests`.

Os arquivos `certo1.cstar` e `certo2.cstar` estão corretos, testando as principais funcionalidades da linguagem.

Os arquivos de erro são os denominados `errado[n]`, onde `n` é um número de 1 a 5. Em cada arquivo de erro, como comentário, estão os erros e as linhas que eles ocorrem.

O arquivo `errado1.cstar` possui um erro de redeclaração de variável, onde a variável `b` no escopo da função `main` é redeclarada.

```
1 int a;
2
```

```

3   int main(){
4       int b;
5       float b; /* redeclaracao de variavel */
6       int a;
7
8       return 0;
9   }

```

Listing 29: errado1.cstar

O arquivo errado2.cstar possui um erro de utilização de uma variável não declarada (a variável naodeclarada).

```

1   int declarada;
2
3   int main(){
4       naodeclarada = 0; /* variavel nao declarada */
5       return 0;
6   }

```

Listing 30: errado2.cstar

O arquivo errado3.cstar possui um erro de tipos incompatíveis, na tentativa de se somar float e int com variáveis do tipo char.

```

1   int main(){
2       int a;
3       int b;
4       int c;
5       char d;
6       float e;
7       c = a + b;
8       c = a - b;
9       c = a * b;
10      c = a / b;
11      c = a + d; /* int e char mismatch */
12      c = a * e; /* conversao implicita funcionando */
13      b = e - a; /* conversao implicita funcionando */
14      return 0;
15  }

```

Listing 31: errado3.cstar

O arquivo errado4.cstar possui um erro de retorno com tipo incompatível, sendo verificado para o tipo tuple.

```

1   tuple int age, char name person;
2
3   int age(tuple p){
4       return p; /* return type mismatch */
5   }
6
7   int gender(tuple p){
8       return p; /* return type mismatch */
9   }
10
11  int main(){

```



```

12         return 0;
13     }

```

Listing 32: errado4.cstar

Observações: para cada etapa há testes adicionais testando diversas funcionalidades, cada um nas pastas /tests/lexicos, /tests/semanticos e /tests/sintaticos, para erros lexicos, semanticos e sintaticos respectivamente.

Há um teste denominado /tests/erradotuple.cstar que possui um erro de atribuição a uma tupla, sendo testado uma atribuição com tipos de dados que não são compatíveis.

7.5 Dificuldades dessa etapa

A principal dificuldade desta etapa foi a checagem de tipos, já que foi necessário atribuir casts implícitos (por exemplo, permitir soma de float com int) e teve-se que fazer toda análise na árvore de cada operação, verificando onde faria sentido os tipos das variáveis envolvidas em cada operação.

Além disso, para se encontrar o escopo do retorno (a função a qual ele se refere no retorno) teve-se que adaptar umas partes globais do código onde se armazenavam os escopos para que o escopo correto estivesse definido na regra de retorno.

8 Código de Três Endereços

Esta etapa consiste, a partir da árvore sintática abstrata, gerar os códigos de três endereços para serem executados pelo TAC - Three Address Code Interpreter [6].

Para isso, a partir da árvore sintática gerada na etapa 3, e após serem feitas todas as análises semânticas da etapa 4, uma função para gerar esses trechos de códigos de três endereços será feita, a partir do percorrimento da árvore:

```

1 void generateTacFile(node *tree, s_node *s_table);

```

Listing 33: Função para percorrer árvore e gerar códigos de 3 endereços

A ideia dessa função é que ela gera um arquivo .tac, e retorna o ponteiro para tal arquivo (para caso seja necessário seu uso). Esse arquivo .tac servirá como entrada para o Three Address Code Interpreter.

O arquivo .tac será formatado conforme o manual descrito em [6]. Para gerar a tabela de símbolos, o parâmetro s_table da função que irá gerar o arquivo .tac será percorrido e formatado de acordo com a especificação, na seção .table do arquivo .tac. A ideia aqui é percorrer cada linha da tabela e apenas formatar na formatação necessária.

Para a seção .code do TAC, uma formatação um pouco mais sofisticada será necessária, uma vez que é preciso gerar códigos em assembly para serem rodados pelo interpretador. Para isso, funções que traduzem partes dos nós na árvore sintática serão criadas para gerar os códigos necessários. Um exemplo disso é o seguinte trecho da árvore, que será traduzido para o código assembly correspondente:

```

1 | val: + | kind: E | type: Regular Node | var_type: int |
2   | val: a | kind: E | type: Regular Node | var_type: int |
3   | val: b | kind: E | type: Regular Node | var_type: int |

```

```
4 // vai gerar
5 add $3, $2, $1
```

Listing 34: Exemplo de trecho da árvore sendo traduzida para código .tac

Podemos inferir os argumentos e a expressão a ser gerada a partir dos parâmetros `val` e `var_type` presentes em cada linha da tabela para montar os códigos de 3 endereços.

Os arquivos de teste a serem utilizados serão os mesmos arquivos de teste corretos utilizados em etapas anteriores (`certo1.cstar` e `certo2.cstar`, presentes na raiz da pasta `tests` do programa).

8.1 Dificuldades

As principais dificuldades pra essa etapa consistem em mapear corretamente as operações para serem traduzidas para os códigos de três endereços, como por exemplo, mapear um laço para labels e códigos de jump na versão traduzida.

Referências

- [1] Alfred V. Aho et al. *Compilers: Principles, Techniques and Tools*. Vol. 7. Addison Wesley, 1986.
- [2] Aquamentus. *Flex and Bison*. URL: https://aquamentus.com/flex_bison.html. (acesso em: 29.09.2020).
- [3] P. Baudin et al. *ACSL: ANSI C Specification Language*. Vol. 2. Domaine de Voluceau, 78150 Rocquencourt, France: Institut National de Recherche en Informatique et en Automatique, 2008.
- [4] Arizona Edu. «C– Language Specification». Em: (abr. de 2005). URL: <http://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/cminusminusspec.html>.
- [5] Shivani Mittal. *Fast Lexical Analyser Generator*. URL: <https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/>. (acesso em: 05.09.2020).
- [6] Luciano Santos. *TAC - the Three Address Code interpreter*. Versão 1.0. Out. de 2014. URL: <https://github.com/lhsantos/tac>.
- [7] Ray Toal. *Regular Expressions*. URL: <https://cs.lmu.edu/~ray/notes/regex/>. (acesso em: 05.09.2020).