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 atribuit 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:

```
tuple char* name, int idade, float salario joao;
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:

```
tuple char* name, int idade, float salario joao;
tuple char* name, int idade, float salario maria;
joao = ("Joao", 21, 3000.0);
maria = ("Maria", 22, 5300.0);
tuple result = joao + maria // podemos declarar os tipos da
  tupla implicitamente

result.salario = 8300.0 // soma dos salarios das tuplas
  envolvidas
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.

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.

```
LETRA [A-Za-z]
DIGITO [0-9]
INT "-"?{DIGITO}+
FLOAT "-"?{DIGITO}+("."{DIGITO}+)*
```

```
ID {LETRA}({LETRA}|{DIGITO})*
BOOL "true"|"false"
TIPO "float"|"int"|"char"|"bool"|"void"
TUPLE "tuple"
CONDICOES "if"|"else"
LACOS "while"|"for"
RETORNO "return"
OP_ARITM "+"|"-"|"*"|"/"
OP_COMP "=="|"!="|"<="|"<"|">="|">"
OP_LOG "&&"|"||"
OP_ASSIGN "="

programa:
  declaracoes

declaracoes:
  declaracoes declaracao
| declaracao

declaracao:
  var_decl
| TUPLE declaracao_tupla
| func_decl

declaracao_tupla:
  TIPO ',' declaracao_tupla
| var_decl

var_decl:
  TIPO ID ';'

func_decl:
  TIPO ID '(' parm_tipos ')' ';
| TIPO ID '(' ')' ';'
| TIPO ID '(' parm_tipos ')' '{' cod_blocks '}' ';'
| TIPO ID '(' ')' '{' cod_blocks '}' ';'

parm_tipos:
  parm_tipos TIPO ID
| parm_tipos TIPO ID '[' ']'
| TIPO ID ','
| TIPO ID
| TIPO ID '[' ']'
| TUPLE ID

cod_blocks:
  cod_blocks cod_block
| cod_block

cod_block:
  "if" '(' expressao_logica ')' '{' cod_blocks '}'
| "if" '(' expressao_logica ')' '{' cod_blocks '}' "else" '{'
    cod_blocks '}'
```

```
57 | LACOS '(' expressao_logica ')' '{' cod_block '}'
58 | RETORNO ';'
59 | RETORNO termo ';'
60 | RETORNO '(' expressao ')' ';'
61 | assign ';'
62 | print
63 | ID '(' expressao ')' ';'
64 | ID '(' ')' ';'
65 | scan '(' ID ')' ';'
66 | declaracoes { $$ = $1 }
67
68 assign:
69   ID OP_ASSIGN expressao
70 | ID '[' INT ']' OP_ASSIGN expressao
71
72 expressao:
73   op_expressao
74 | '(' expressao ')'
75
76 expressao_logica:
77   OP_LOG op_expressao
78 | '!' op_expressao
79 | op_expressao OP_COMP op_expressao
80 | '(' op_expressao ')'
81 | op_expressao
82
83 op_expressao:
84   op_expressao OP_ARITM termo | termo
85
86 termo:
87   ID | INT | FLOAT | ID '[' INT ']'
88
89 scan:
90   SCAN '(' ID ')'
91
92 print:
93   PRINT '(' termo ')' ';' | PRINT '(' palavra ')' ';'
```

Listing 5: 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 à 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

A linguagem proposta para o trabalho é a C*, consistindo em uma extensão da linguagem C com a adição de operações sobre tuplas. Para isso 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].

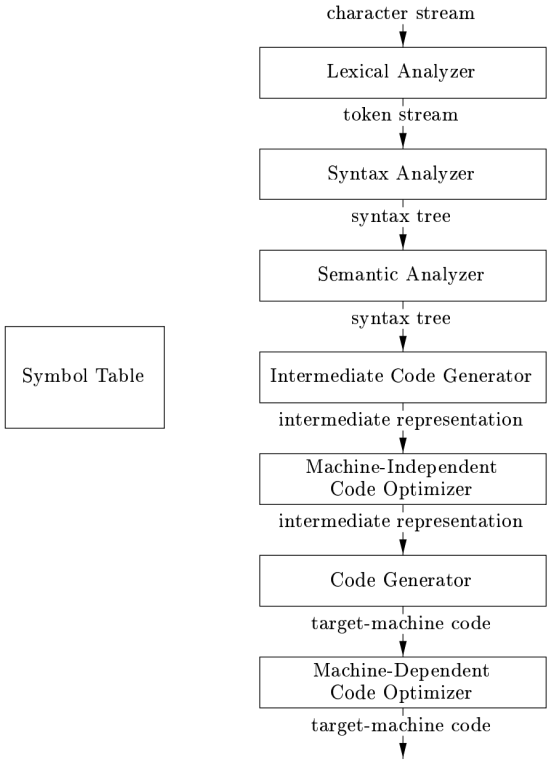

Figura 1: Etapas de Tradução

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 6: 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, "Simbolo n o pertence
    gram tica");
5           strcpy(err[error_index].sym, yytext);
6           err[error_index].line = lin;
7           error_index++;
8       }
```

Listing 7: Regra .

Esse padrão basicamente, definido pela expressão regular "."[6], 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.

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 l xicos:\n");
20      int i;
21      for(i=0; i<errors; i++){
22          printf("\nErro n mero %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 8: 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 lerror é a estrutura de dados utilizada para salvar os erros, contendo o tipo do erro (type), a mensagem de erro à ser exibida (msg), o símbolo encontrado no erro (sym) e a linha onde

o erro foi encontrado (line). A função printErrros é chamada no final da análise léxica, onde o vetor denominado err (vetor de lerrors) 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:

```
int 5a;
```

Listing 9: teste_error.cstar

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

```
int ab@cax# = 0;

int returnAbacaxi() {
  return abacaxi;
 }
```

Listing 10: 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:

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

Listing 11: 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 sintática 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 fora identificados com sua associatividade respectiva, right para associativiadade 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_ARITM
14  %left <operador> OP_LOG
15  %left <operador> OP_COMP
```

Listing 12: 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 13: 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 14: 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:

```
typedef struct node {
  char* var_type; //
  char node_kind; // 'F' function, 'V' var, 'C' code_block..., 'D' declaration, 'E' expression
  int node_type; // 'S' for symbol | 'R' for regular
  struct node *left;
  struct node *right;
  struct node *middle;
  char *val;
} node;
```

Listing 15: 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:

```
# TABELA DE S MBOLOS
{
  "person::main": {id: "person", var_type: "int", s_node_type: 'S', scope: 0}
}
```

Listing 16: 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:

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

```
} s_node;
```

Listing 17: 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:

```
int i;

int sum(int a, int b){
  return (a + b);
};
```

Listing 18: teste_sintatico.cstar

teste_sintatico2.cstar:

```
  tuple int, char person;

  int age(tuple p){
    return p[0];
  };

  int gender(tuple p){
    return p[1];
  };

  void printall(){
    print(person);
  };
```

Listing 19: 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:

```
  int main(){
    person = 43;
  };
  tuple variavel;
```

Listing 20: teste_sintatico_error.cstar

Onde o erro consiste na não declaração dos tipos da tupla.
teste_sintatico_error2.cstar:

```
  global = 0
  int main(){
    return 0;
```

```
4  }
```

Listing 21: 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 22: 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_MISSMATCH_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 23: 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

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

Listing 24: 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:

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

Listing 25: 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:

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

Listing 26: 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_MISSMATCH, 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 referentes aos testes semânticos se encontram na pasta tests/semanticos.

O arquivo error1.cstar possui um erro de redeclaração de variável, onde a variável b no escopo da função main é redeclarada.

```
int a;

int main(){
    int b;
    float b;
    int a;

    return 0;
};
```

Listing 27: error1.cstar

O arquivo error2.cstar possui um erro de utilização de uma variável não declarada (a variável naodeclarada).

```
int declarada;

int main(){
    naodeclarada = 0;
    return 0;
};
```

Listing 28: error2.cstar

O arquivo aritm_error.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;
12       c = a * e;
13       b = e - a;
14       return 0;
15   };
```

Listing 29: aritm_error.cstar

7.5 Dificuldades dessa etapa

A principal dificultade 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.

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] Ray Toal. *Regular Expressions*. URL: `https://cs.lmu.edu/~ray/notes/regex/`. (acesso em: 05.09.2020).