UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
Compiladores

Nome: Fernando Gomes
Nome: Isaque Barbosa Martins
Nome: Thuanny Carvalho Rolim de Albuquerque
Nome: Matheus Leão de Carvalho
Nome: Gabriel Henrique Campos Medeiros

Relatório sobre a Linguagem CMANTIC

Nome	Pontuação
Fernando Gomes	10
Gabriel Henrique	10
Isaque Barbosa	10
Thuanny Albuquerque	10
Matheus Leão	10

Tabela 1: Pontuações por participação dos alunos

Link para o repositório.

1 Introdução

Neste projeto, estamos implementando um compilador feito com a ferramenta Bison para a linguagem **CMANTIC** e com analixador léxico feito com a ferramenta Flex. O objetivo deste trabalho é explorar e aplicar conceitos de compiladores, como análise léxica e análise sintática. A linguagem que está sendo criada pelo nosso grupo possui como diferencial as estruturas de seleção "case & otherwise"e "unless & else". O analizador produzido é capaz de fazer análise de tipo baseado em um esquema de tradução dirigido pela sintaxe.

2 Design da Implementação

2.1 Remodelagem do analisador léxico da linguagem

Para a implementação do nosso compilador, é necessário a construção do analisador léxico da linguagem. Ele é responsável por ler os caracteres do código-fonte, agrupar os caracteres em lexemas, classificá-los.

Tomando como base o analisador léxico já construído no projeto anterior, o adatamos para que utilizasse os tokens gerados pelo nosso analisador sintático, implementado com o Bison. Dessa forma, temos uma dependência da compilação do código do Bison, mas temos um código resultante como o abaixo. Note, o uso do namespace *yy*, a classe *parser* e o enum *token*, gerados pelo bison.

```
%option c++ noyywrap

%option yyclass="CustomLexer"

%{
    #include "parser.tab.hh"
    #include "custom_lexer.hpp"
    #include "symbol_table.hpp"

    #include <iostream>
    #include <string>
    #include <cstring>

    #undef  YY_DECL
    #define YY_DECL int CustomLexer::yylex(yy::parser::semantic_type* yylval)

    extern int line_number;
```

```
18
19      void yyerror(CustomLexer& lexer, const char *message);
20  %}
21
22  NAME [a-zA-Z][a-zA-Z0-9_]*
23  SINGLE_COMMENT "//".*
24  MULTI_COMMENT "(\*"([^*]|\*+[^)*])*\*+")"
25  INT_LITERAL [0-9]+
26  STRING_LITERAL \"([^\\\"\n]|\\.)*\"
27  FLOAT_LITERAL [0-9]+"."[0-9]+([Ee][-+][0-9]{2})?
28
29  %%
30
31  {INT_LITERAL} {
32      yylval->ival = std::stoi(yytext);
33      return yy::parser::token::A_INT_LITERAL;
34  }
35
36  {FLOAT_LITERAL} {
37      yylval->fval = std::stof(yytext);
38      return yy::parser::token::A_FLOAT_LITERAL;
39  }
40
41  {STRING_LITERAL} {
42      yylval->sval = new std::string(yytext + 1, strlen(yytext) - 2);
43      return yy::parser::token::A_STRING_LITERAL;
44  }
45
46  {NAME} {
47      yylval->sval = new std::string(yytext);
48      return yy::parser::token::A_NAME;
49  }
50
51  {SINGLE_COMMENT} {
52      std::cout << "Single comment: " << yytext << std::endl;
53  }
54
55  {MULTI_COMMENT} {
56      std::cout << "Multi comment: " << yytext << std::endl;
57  }
58
59  "if" {
60      return yy::parser::token::A_IF;
61  }
62
63  "then" {
64      return yy::parser::token::A_THEN;
65  }
66
67  "else" {
68      return yy::parser::token::A_ELSE;
69  }
70
71  "fi" {
72      return yy::parser::token::A_FI;
73  }
74
75  ...
76
77  \n    {
78      line_number++;
79      }
80
81  [ \t\r]+  { /* Ignore whitespace */ }
82
```

```
.      { yyerror(*this, "Invalid character"); }

%%

void yyerror(CustomLexer& lexer, const char *message)
{
    std::cerr << "Error: \"" << message << "\" in line " << line_number
              << ". Token = " << lexer.YYText() << std::endl;
    exit(1);
}
```

Listing 1: "Dicionário léxico"

No inicio do analisador há algumas opções, macros e inclusões para o correto funcionamento na linguagem C++, logo em seguida há um contador de linhas para uso no caractere \n e, por fim a declaração da função *yyerror*, utilizado para lançamento de erros léxicos.

Logo em seguida, tem-se uma lista de *regex* utilizados na análises de valores literais e para a verificação de comentários. Por fim, temos a lista de análises léxicas, onde cada uma retorna um tipo enumerado (enum) e em alguns casos um valor adicional armazenado em uma union, ambos difinidos no arquivo *parser.y*.

2.2 Tabela de símbolos do compilador

A tabela de símbolos é responsável por armazenar os identificadores, e seus respectivos tipos, em escopos do programa, sendo capaz de armazenar os nomes e tipos de variáveis, structs e procedures e o escopo corrente onde cada identificador foi criado.

A tabela de símbolos, mostrada na listagem 2, irá armazenar uma lista de escopos e o escopo atual para coordenar a localidade das variáveis. Cada escopo, enfim, possui um *map* de símbolos, de forma que armazena-se seu nome e seu conteúdo (o tipo).

```
class SymbolTable {
    private:
        struct Scope {
            std::unordered_map<std::string, Symbol> symbols;
            Scope* parent;

            Scope() : parent(nullptr) {}
            Scope(Scope* p) : parent(p) {}
        };

        Scope* current_scope;
        std::vector<std::unique_ptr<Scope>> all_scopes;
}
```

Listagem 2: Estrutura da tabela de símbolos

Cada símbolo consiste em uma *struct* com um nome, uma categoria e seu conteúdo, como mostrado na listagem 3. A categoria indica se o símbolo é referente à um procedimento, registro ou variável, sendo utilizado para comparações.

```
struct Symbol {
    std::string name;
    SymbolCategory category;
    SymbolContent content;
};
```

Listagem 3: Estrutura dos símbolos

```
enum class SymbolCategory {
    PROCEDURE,
    RECORD,
    VARIABLE,
    UNDEFINED
};
```

Listagem 4: Caterogia dos símbolos

O conteúdo 5 de um símbolo é implementada por um *std::variant* que é análogo a uma *union*, onde pode armazenar os tipos *Procedure*, *Record* ou *Variable*.

O tipo ***Procedure*** consiste em um atributo do tipo *VarType* representando o tipo de retorno do procedimento e um vetor de *ParamField* que indicarão o nome e tipo dos parâmetros deste procedimento.

O tipo ***Record*** possui somente um vetor de *ParamField* que representa os atributos do registro.

Por fim, o tipo ***Variable*** representa uma variável em si e possui apenas um atributo *VarType* indicando o seu tipo.

```
struct ParamField {
    std::string name;
    VarType type;
};

struct Procedure {
    VarType return_type;
    std::vector<ParamField> params;
};

struct Variable {
    VarType type;
};

struct Record {
    std::vector<ParamField> fields;
};

using SymbolContent = std::variant<Procedure, Record, Variable>;
```

Listagem 5: Conteúdo dos símbolos

A estrutura *VarType* 6 foi implementada de forma flexível permitindo trabalhar com diferentes tipos da linguagem somente com este tipo. Ela possui:

- Um enum *PrimitiveType* que indica o tipo primitivo armazenado por esta variável, o tipo *NOT_PRIMITIVE* indicando que não se trata de um tipo primitivo, ou o tipo *UNDEFINED* sinalizando que o tipo pode ainda ser definido, ou indica um erro a depender da localidade analizada.
- Possui um atributo opcional indicando o nome do registro, isto por que, quando se cria uma variável do tipo de um registro criado pelo usuário, o nome do registro deve ser armazenado em algum lugar para poder ser usado como referência.
- Um atributo do tipo *std::unique_ptr<VarType>* que guarda uma referência para o tipo referenciado, caso seja uma referência .

```
enum class PrimitiveType {
    BOOL,
    FLOAT,
    INT,
    STRING,
    VOID,
    NOT_PRIMITIVE,
    REF,
    UNDEFINED
};

struct VarType {
    PrimitiveType p_type;
    std::optional<std::string> record_name;
    std::unique_ptr<VarType> referenced_type;
}
```

Listagem 6: Tipo dos símbolos

2.3 Gramática de atributos

Para compreender o que é validado por nosso programa, é necessário compreender os atributos de nossas produções. Dado o enfoque na verificação de tipos, temos uma gramática sumarizada em tipos e escopos, e validações dependentes dos tipos e escopos. Dessa forma, geramos a tabela 2 de atributos.

Simbolo	Atributo	Descrição
PROGRAM	valid	Sintetizado: Programa correto
DECL_LIST	scope	Sintetizado: Conjunto de pares (name, tipo) de variáveis declaradas
DECL	value	Sintetizado: Pare (name, tipo) da variável declarada
VAR_DECL	value	Sintetizado: Par (name, tipo) referente à variável declarada
PROC_DECL	scope	Sintetizado: Conjunto de pares (name, tipo) de variáveis declaradas
REC_DECL	scope	Sintetizado: Conjunto de pares (name, tipo) de variáveis declaradas
PARAMFIELD_DECL	value	Sintetizado: Par (name, tipo) referente à variável declarada
STMT_LIST	scope	Sintetizado: Conjunto de pares (name, tipo) de variáveis declaradas
STMT	scope	Herdado: Conjunto de pares (name, tipo) de variáveis declaradas
ASSIGN_STMT	value	Sintetizado: Par (name, tipo) referente à variável declarada
IF_STMT	scope	Herdado: Conjunto de pares (name, tipo) de variáveis declaradas
WHILE_STMT	scope	Herdado: Conjunto de pares (name, tipo) de variáveis declaradas
RETURN_STMT	type	Sintetizado: Tipo da expressão retornada
CALL_STMT	valid	Sintetizado: Tipos das expressões correspondem aos tipos declarados no procedimento
CALL_STMT	type	Sintetizado: Tipo do procedimento
EXP	valid	Sintetizado: Tipos das sub-expressões são equivalentes
EXP	type	Sintetizado: Tipo resultante da expressão
VAR	value	Sintetizado: Par (name, tipo) referente à variável declarada
REF_VAR	type	Sintetizado: Tipo resultante
DEREF_VAR	value	Sintetizado: Par (name, tipo) referente à variável declarada
NEW_NAME	type	Sintetizado: Tipo respectivo ao identificador
TYPE	type	Sintetizado: Tipo descrito
ARITH_OP	valid	Sintetizado: Tipos das expressões utilizadas são equivalentes
LOG_OP	valid	Sintetizado: Tipos das expressões utilizadas são equivalentes
REL_OP	valid	Sintetizado: Tipos das expressões utilizadas são equivalentes

Tabela 2: Tabela de atríbutos para a gramática

Dado os atríbutos pensados para a nossa gramática, foi criado um gráfico de dependência indicando a forma com os quais os atributos são herdados ou sintetizados entre as produções da gramática, mostrado na figura 1.

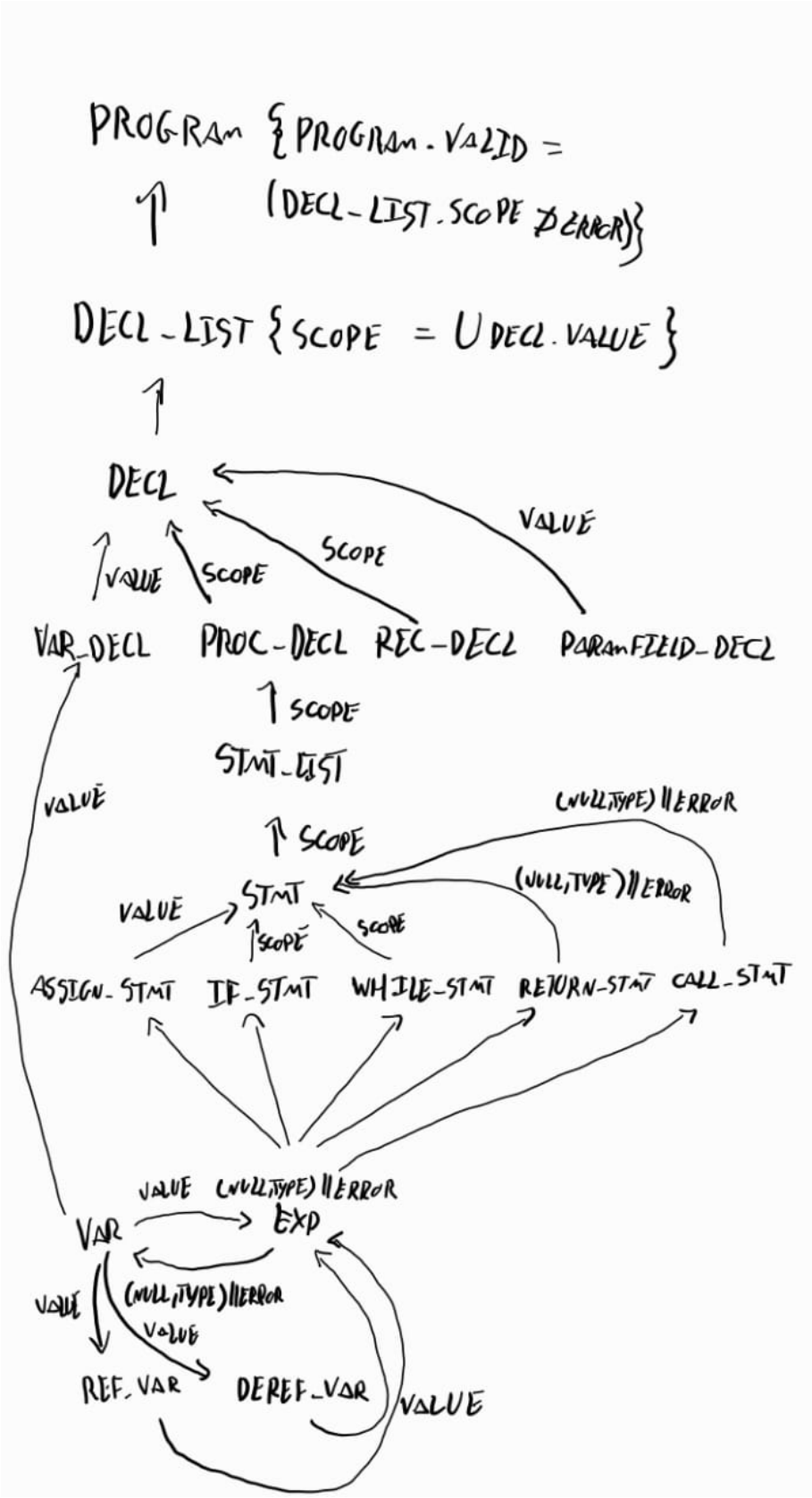

Figura 1: Grafo de dependência dos atributos

2.4 Analisador sintático

O analisador sintático consiste no programa Bison/Yacc que implementa os tokens utilizados no analisador léxico e cria as produções do programa. Como auxílio, também é criado uma union que é utilizado para

armazenar variáveis a serem guardadas na tabela de símbolo.

Para cada produção, o Bison permite a implementação de uma lógica customizada a ser utilizada com os valores das produções internas. Dessa forma, é possível implementar aquilo que é idealizado na gramática de atríbutos, ou seja, é possível gerar, armazenar e comparar os valores das produções e, em específico, os nomes e tipos de variáveis, registros e procedimentos. Sendo assim, é detalhado abaixo a estrutura básica de nosso analisador sintático e também é disponibilizado o código implementado neste link.

Nas listagens 7 e 8 têm-se a definição da união que armazena valores durante a análise; além da definição dos tokens, os mesmos utilizados no programa flex, e, para os tokens respectivos à operadores, a associatividade. O próprio Bison utilizará a associatividade definida para realizar a análise sintática corretamente das expressões.

```
%union {
    int ival;
    float fval;
    std::string* sval;
    VarType* type_val;
    std::vector<ParamField>* param_vec;
    std::vector<VarType*>* type_vec;
}

%token<ival> A_INT_LITERAL
%token<fval> A_FLOAT_LITERAL
%token<sval> A_NAME A_STRING_LITERAL

%type<type_val> optional_assign_exp exp type_spec literal bool_literal
    var_access
%type<type_val> ref_var deref_var call_stmt_as_exp optional_exp_val
    optional_return_type
%type<param_vec> optional_param_list param_list optional_rec_field_list
    rec_field_list paramfield_decl
%type<type_vec> optional_arg_list arg_list

%token A_LINE

%token A_PROGRAM A_BEGIN A_END A_PROCEDURE A_VAR
%token A_IF A_THEN A_ELSE A_FI A_WHILE A_DO A_OD A_RETURN A_UNLESS A_CASE A_OF
    A_ESAC A_OTHERWISE
%token A_TRUE A_FALSE A_FLOAT A_INT A_STRING A_BOOL A_NULL A_STRUCT
%token A_IN A_NOT A_NEW A_REF A_DEREF A_U_MINUS A_U_PLUS
%token ';' ':' ',' '[' ']' '{' '}' '(' ')' '<' '>'  '=' '+' '-' '*' '/' '^' '.'
    '|'
%token A_ASSIGN A_LESS_THAN_EQUAL A_GREATER_THAN_EQUAL A_DIFFERENT A_EQUAL
    A_OR_LOGIC A_AND_LOGIC A_RANGE
```

Listagem 7: Definição dos tokens

```
%left  A_OR_LOGIC
%left  A_AND_LOGIC
%nonassoc '<' '>' A_LESS_THAN_EQUAL A_GREATER_THAN_EQUAL A_EQUAL A_DIFFERENT
%left '+' '-'
%left '*' '/'
%right '^'
%right A_NOT A_U_MINUS A_U_PLUS
%left '.'
```

Listagem 8: Associatividade dos operadores

Por fim, na listagem 9, temos um exemplo de uma produção que utiliza os valores de suas produções internas para gerar um símbolo e armazena-lo na tabela de símbolos. A produção em questão é a da declaração de variáveis e, nela, é possível verificar a comparação de tipos da expressão e do tipo especificado ou a geração de erros, além do armazenamento de símbolos no caso correto.

Vale notar que a idealização de que a validez do programa se torna um atributo virtual a partir do momento em que lançamos o erro na própria produção VAR_DECL, entre outras. O lançamento do

erro localmente foi feito pois é possível, futuramente, criar erros mais detalhados com valores locais que seriam perdidos ao implementar a escalada de verificações para produções pai.

```
var_declaration:
    A_VAR A_NAME ':' type_spec optional_assign_exp // VAR name ":" TYPE [ ":=" EXP]
    {
        // Verifica se a variável já foi declarada no escopo atual
        if (symbol_table.lookup_current_scope_only(*$2)) {
            error("Variavel '" + *$2 + "' ja declarada neste escopo.");
        } else {
            bool types_are_ok = true;
            if ($5) {
                // Se os tipos de type_spec e optional_assign_exp não forem compatíveis, gera um erro
                if (!are_types_compatible(*$4, *$5)) {
                    std::string declared_type = type_to_string(*$4);
                    std::string assigned_type = type_to_string(*$5);
                    error("Incompatibilidade de tipos para a variável '" + *$2 +
                            "'. Tipo declarado: " + declared_type +
                            ", mas o tipo da expressão atribuída é:: " + assigned_type + ".");
                    types_are_ok = false;
                }
            }

            if (types_are_ok) {
                // Se os tipos forem compatíveis, cria a variável e insere na tabela de símbolos
                Variable var_content{*$4};
                Symbol new_symbol{*$2, SymbolCategory::VARIABLE, var_content};
                symbol_table.insert_symbol(*$2, new_symbol);
            }
        }
        delete $2;
        delete $4;
        if ($5) delete $5;
    }
    | A_VAR A_NAME A_ASSIGN exp // var NAME ":=" EXP
      {
        // Verifica se a variável já foi declarada no escopo atual
        if (symbol_table.lookup_current_scope_only(*$2)) {
          error("Variavel '" + *$2 + "' ja declarada neste escopo.");
          delete $4;
        } else {
            // Se a variável não foi declarada, declara-a com o tipo da expressão
          Variable var_content{*$4};
          Symbol new_symbol{*$2, SymbolCategory::VARIABLE, var_content};
          symbol_table.insert_symbol(*$2, new_symbol);
        }
        delete $2;
        delete $4;
      }
    ;
```

Listagem 9: Produção de declaração de variáveis

2.5 Validação

Para verificar que nosso programa funcionou corretamente, fizemos casos de testes para validar as diferentes características da linguagem, onde foi dividido em casos válidos, ou seja, aqueles que o programa deve rodar e finalizar sem erros; e os casos inválidos, aqueles que aguardamos que um erro ocorra no decorrer do programa. Abaixo são listados os casos pensados para validação da linguagem:

1. **Válidos**:
    - Declaração de variáveis: Foi testado a declaração com inferência de tipos, com a especificação de tipos, além da atribuição e reatribuição de variáveis;
    - Declaração de procedimentos: Testes para validar a declaração e chamamento de procedimentos no decorrer do programa;
    - Declaração de registros: Testes para validar a declaração de registros com suas variáveis, e a atribuição de um registro à uma variável;
    - Expressões: Foram testados as diversas formas de expressões que a gramática oferece;
    - Blocos: Por fim, foram testados os blocos de código, como If, while e outros.
2. **Inválidos**:
    - Declaração de variáveis: Foram idealizados testes para validar que não é permitido declarar uma variável mais de uma vez e que não é permitido modificar um tipo de uma variável;
    - Expressões: Foi validado que não é possível uma expressão acessar uma variável que não foi declarada.