Universidade Federal de Sergipe

Relatório
Elipses ...

Matéria: Compiladores
Turma: 01
Docente: Beatriz Trinchão Andrade de Carvalho

Discentes:Felipe Santos Rocha,
Joao Paulo Feitosa Secundo,
Vitoria Maria Meneses Mota Teixeira,
Everton Santos de Andrade Júnior

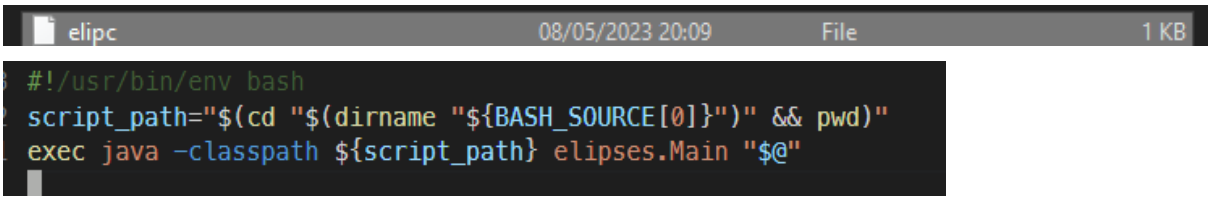
Sumário

1. Introdução

Implementamos a Tabela de Símbolos, Analisador Semântico e geração de código em C (versão C99) e geração do executável caso *gcc* esteja no *path*.

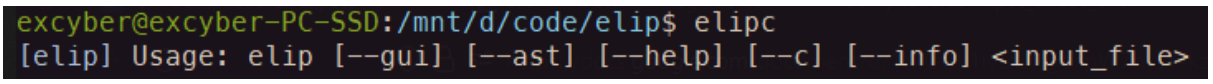
Após compilar o projeto inteiro criamos dois arquivos de scripting que simulam um executável do nosso compilador um versão para Windows e outra para Linux. Isso foi feito para não precisar chamar java para compilar um arquivo *.elip*. Para ver as intruções de compilação e uso veja **readme.md** na raíz do projeto, para mairo facilidade ver : https://github.com/evertonse/elip

No caso de linux temos um arquivo bash chamado elipc que faz a passagem dos argumentos para o compilador usando java.

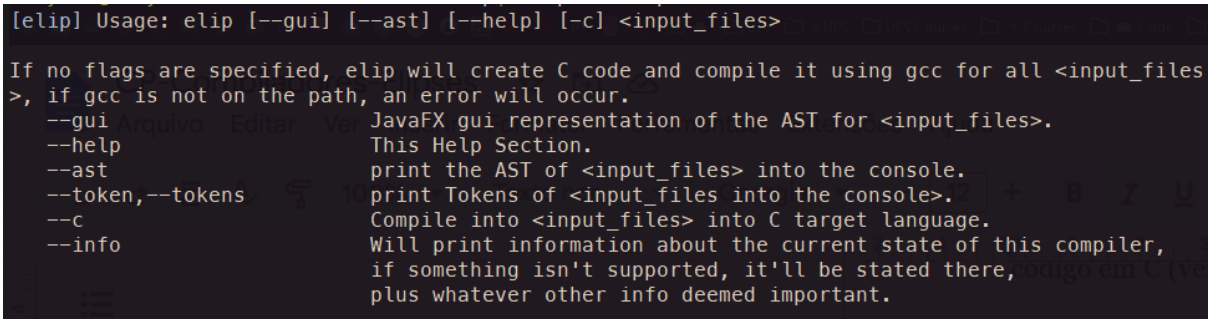


Demonstração:

Uma vez configurado no path podemos chamar o compilador pelo terminal usando o comando **elipc**:



Uma sessão de ajuda ao usar **elipc --help:**



Compilando um arquivo .elip, considere fibonacci.elip

```
entrada (inteiro fibonacci(inteiro n):(
    se (n < 2) entao
        n
    senao
        (fibonacci(n-1) + fibonacci(n-2))
))
```

Usamos o comando **elipc** nome_do_arquivo.elip:

```
excyber@excyber-PC-SSD:/mnt/d/code/elip/test/examples$ elipc fibonacci.elip
[elip] INFO: input filefibonacci.elip
[elip] Tokenizing file: fibonacci.elip
[elip] success:  Tokenized with success.
[elip] success:  Parsed into AST with success.
[elip] success:  semantic analysis checked.
[elip] Tokenizing file: fibonacci.elip
[elip] success: C code generated at fibonacci.elip.c
[elip] C code fibonacci.elip.c is compilable
```

Por *default* Elipses Compiler (**elipc**) irá *tokenizar*, fazer o *parsing*, fazer *validação semântica* e *gerar código* em C. Note as informações passadas pelo comando, indicando que todas as etapas foram um sucesso.
Vamos visualizar o código gerado, mais detalhes sobre esse processo nas sessões a seguir.

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <ctype.h>
#include <math.h>
// Define an enum for boolean values
typedef enum {
    false = 0,
    true = 1
} bool;

float elip_cosseno(float x);
float elip_seno(float x);
float elip_logaritmo(float x);
float elip_logaritmo(float x);
float elip_potencia(float x, float y);

int elip_fibonacci( int elip_n);
float elip_cosseno(float x) {
    return cosf(x);
}

float elip_seno(float x) {
    return sinf(x);
}

float elip_tangente(float x) {
    return tanf(x);
}

float elip_logaritmo(float x) {
    return log10f(x);
}

float elip_potencia(float x, float y) {
    return powf(x,y) ;
}
```

Note que importamos bibliotecas comuns que serão usadas para informar algumas coisas para o usuário de elipses. Além de definir funções padrões de matemática que são podem ser utilizadas pelo programador de elipses.

Mais embaixo, tem a função main de C e a função de entrada de elipses que é chamada por pelo main do C, mais explicações nas próximas seções.

Percebe-se como um bloco é implementado em C e como a expressão **se-senao** é traduzida, são todos blocos de C.

Todo nome que pertence à elipses têm o prefixo “**elip_**”, para higienizar as variáveis e não haver colisão com algo definido em C.

```c
int elip_fibonacci( int elip_n)  {

    int if_0;
    {
        if ((elip_n<2)) {
            if_0 = elip_n;
        } else {

            int block_0;
            {
                block_0 = (elip_fibonacci((elip_n-1)) + elip_fibonacci((elip_n-2)));
            }
            if_0 = block_0;
        }
    }
    int return_data = if_0;
    return (return_data);
}
```

```c
int main(int argc, char *argv[]) {
    setlocale(LC_ALL, "fr_FR.UTF-8");
    char *arg;
    size_t arg_len;
   if (argc <2) {
        printf("%s expects 1 arguments from cmd line\n", argv[0]);
        printf("The expected types are inteiro n ");
         return 1;
    }
    arg = argv[1];

    arg_len = strlen(arg);
    int n = atoi(arg);

    printf("%d\n",((int)elip_fibonacci(n)));
}
```

Compilando um executável:
Basta adicionar uma flag **elipc** --exe

```
[elip] INFO: elip is gonna generate C code as Target Language.and create a executable using gcc.
[elip] INFO: input filefibonacci.elip
[elip] Tokenizing file: fibonacci.elip
[elip] success:  Tokenized with success.
[elip] success:  Parsed into AST with success.
[elip] success:  semantic analysis checked.
[elip] Tokenizing file: fibonacci.elip
[elip] success: C code generated at fibonacci.elip.c
[elip] C code fibonacci.elip.c is compilable
[elip] success: fibonacci.elip.c C code compiled into executable fibonacci successfully
```

Vamos executar fibonacci

```
excyber@excyber-PC-SSD:/mnt/d/code/elip/test/examples$ ls
fibonacci  fibonacci.elip  fibonacci.elip.c
```

Ao executar sem argumentos somos avisados, que fibonacci espera um *inteiro n*

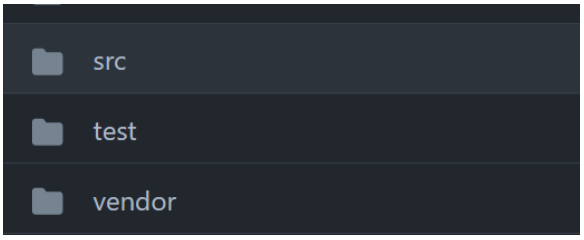
```
excyber@excyber-PC-SSD:/mnt/d/code/elip/test/examples$ ./fibonacci
./fibonacci expects 1 arguments from cmd line
The expected types are inteiro n
```

Todo parâmetro definido na entrada do *.elip* compilado deve ser passado pelo terminal (*cmd line*), por exemplo, ./fibonacci 3 retorna o terceiro número de fibonacci, calculado recursivamente, como vimos no *.elip*

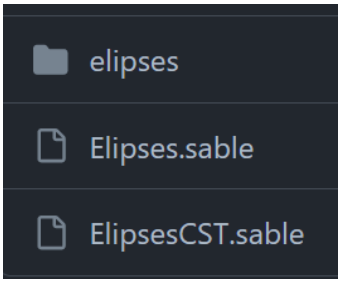
```
excyber@excyber-PC-SSD:/mnt/d/code/elip/test/examples$ ./fibonacci 3
2
```

Outras opções são válidas para debugging como *--token* para ver os tokens quando compilar, ou *--gui* para visualizar a árvore abstrata (AST) usando JavaFX

2. Estrutura das pastas

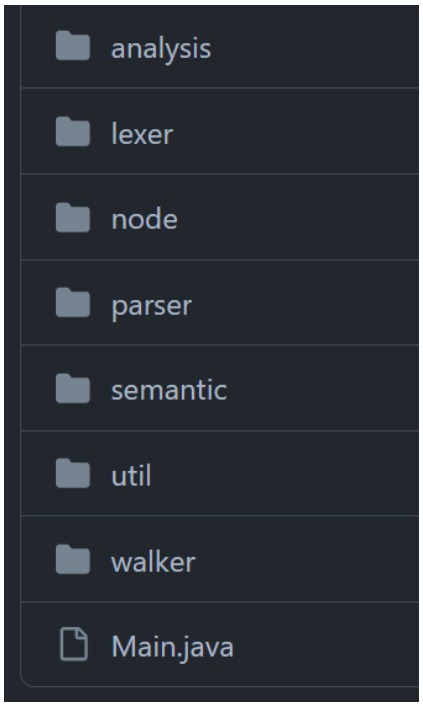


As três pastas da raiz: **src**, contém todo o código para compilar *Elipses Compiler* (elipc), isto é, o projeto inteiro; **vendor**, são as dependências que nesse caso é sablecc.jar; **test**, são arquivo *.elip* para compilar e testar elipc;



Dentro de **src** temos o arquivo *.sable,* o arquivo *ElipseCST.sable* gera a **árvore concreta** que pode ser usada para debugging, a **árvore abstrata (AST)** se encontra em "*Elipses.sable*".

Ademais, "src" contém pacote **elipses** onde todas as classes geradas pelo sablecc, adicionadas das classes implementadas no projeto, se encontram.



No pacote elipses temos os pacotes implementados por nós que são o pacote "***semantic***", "***util***" e "***walker***".

Main é o começo do compilador.

- "util", contém a Logger de elipses e um String Builder porém com indentação e outras capacidades para implementação do gerador de código C.
- "semantic", todas as classes para fazer inferência de tipos, símbolos, classe de erros e warnings, flags.
- "walker" todos os classes que são filhas de Depth First Adapter

3. Símbolo

Inicialmente criamos uma classe para o **Symbol**, todo símbolo pode possuir no máximo 5 informações que são, o tipo, a assinatura, o nome, possivelmente o token e o valor. O token serve para informação de linha e posição dado um erro.

```
private String name;
private Signature signature;
private Type type;
private Token token;
```

Um tipo é um enum java, que pode ser , *unkown*. *Unkown* ocorre quando fazemos uma operação errada, deixamos como unkown para propagar erros e poder encontrar mais erros de uma vez só, pode ser também *int, bool,real* ou *function*.

```
static public enum Type {
    UNKOWN, INT,BOOL,REAL, FUNCTION, COUNT;
```

Se for *function*, isso quer dizer que esse símbolo representa uma função e sua assinatura *Signature* não deve ser nula.

Assim como na especificação, a classe Signature é um tipo recursivo, que contém uma lista de **SignatureParam** que é uma classe que pode ser um Type ou outra Signature .

```
static public class Signature {
    Type return_type;
    List<SignatureParam> parameters = new ArrayList<>();
```

```
static public class SignatureParam {
    Type type = null;
    Signature sig = null;
```

Na classe **Signature** definimos uma função recursiva que testa igualdade entre duas Signature, essa função vai servir para validar chamadas de funções, e parâmetros. Outros detalhes se encontram no código fonte.

4. Tabela de Símbolos

Uma vez com a definição de símbolos em mão, procuramos criar uma tabela de símbolos utilizados na estrutura de dados padrão de Java, como **Stack** e **Hashmap**.

Além disso, a tabela de símbolos deve ter capacidade de entrar em um escopo, sair de um escopo, registrar uma variável e seu tipo/symbolo, checar de uma variável

existe em qualquer escopo válido, ou seja checar o escopo atual e seus escopos país, checar se está no escopo global ou atual.

A implementação foi feita usando um Stack (pilha) de HashMaps, e cada HashMap associa uma chave string para um valor Symbolo dessa maneira:

```
public class SymbolTable {
    private Stack<Map<String, Symbol>> stack;
    private Map<String, Symbol> global;
```

Para entrar e sair de um escopo, basta dar um push ou pop no stack assim:

```
public void enterScope() {
    stack.push(new Hashtable<>());
}

public void exitScope() {
    stack.pop();
}
```

Para adicionar, deve-se olhar no topo do stack e adicionar lá:

```
public boolean add(String name, Symbol symbol) {
    Map<String, Symbol> current_scope = stack.peek();
    if (current_scope.containsKey(name)) {
        return false; // Symbol with the same name already exists
    } else {
        current_scope.put(name, symbol);
        return true; // Symbol added successfully
    }
}
```

Para checar se existe simplesmente olhamos todo os escopos do atual até o global e vemos se encontramos esse nome:

```
// If exists in any scope
public boolean exists(String name) {
    for (int i = stack.size() - 1; i >= 0; i--) {
        if (stack.get(i).containsKey(name)) {
            return true;
        }
    }
    return false;
}
```

De maneira similar podemos pegar um símbolo, dado um nome. Precisamos checar todo o stack de hashmaps:

```
public Symbol get(String name) {
    for (int i = stack.size() - 1; i >= 0; i--) {
        Map<String, Symbol> current_scope = stack.get(i);
        if (current_scope.containsKey(name)) {
            return current_scope.get(name);
        }
    }
    return null;
}
```

O mesmo é aplicado para outras funções da tabela de símbolo que são simplesmente utilitárias para auxiliar na etapa semântica.

5. Utils

Foram desenvolvidas duas classes utilitárias uma é a **ElipLogger**, uma classe estática que printa com cores como vimos na introdução, e também printa formatando e indicado linha e posição.

E a outra classe é **IndentedStringBuilder**, que é usada extensivamente na geração de código C. Para manter legibilidades criamos um nível de indentação.

Nessa mesma classe, houve a implementação que possibilitou salvar o estado de um index de uma string e adicionar nesse index mais tarde (mais tarde aqui quer dizer em outro nó da árvore posterior ao atual). Isso é muito bom para inferência de tipos quando o tipo só será resolvido em um momento posterior em um outro nó da AST.

```
public class IndentedStringBuilder {

    private StringBuilder sb;
    private String indentation;
    private int indentation_level;
    Stack<Integer> index_stack = new Stack<>();
```

A função de salvar um index é com a interface

```
public IndentedStringBuilder pushIndex() {
    this.index_stack.push(this.size());
    return this;
}

public IndentedStringBuilder popIndex() {
    this.index_stack.pop();
    return this;
}
```

Dado um index atual podemos inserir nesse local salvo com a função:

```
public IndentedStringBuilder insertAtCurrentIndex(String str) {
    int i = index_stack.peek();
    this.insert(i,str);
    return this;
}
```

Algo similar acontece em indentação, neste é usado também na busca na AST, se estamos dentro de um bloco é esperado que aumente-se o indentação e em seguida quando sair do bloco diminua a indentação

```
public IndentedStringBuilder pushIndent() {
    this.indentation_level++;
    this.indentation = " ".repeat(this.indentation_level * 4);
    return this;
}

public IndentedStringBuilder popIndent() {
    if (this.indentation_level > 0) {
            this.indentation_level--;
            this.indentation = " ".repeat(this.indentation_level * 4);
        }
    return this;
}
```

6. Inferência de Tipos

Criamos uma classe que faz a inferência de tipo, ela é totalmente recursiva. Nela temos duas funções importantes, **getSymbolOrNull** leva em consideração todas as possíveis expressões que podem ser avaliadas em um símbolo. Não é só um ID que pode gerar um símbolo, uma expressão *se-senao*, pode gerar um símbolo, e este símbolo pode até ser uma função.

```
  (inteiro one ():(1))
1 (inteiro two ():(2))
2
3 (inteiro surprise(inteiro fn()):(fn()))
4
5 (inteiro if_function_reference():(
6     surprise(
7         se (
8             se (verdadeiro) entao verdadeiro senao falso
9         ) entao
0             one
1         senao
2             two
3     )
4 ))
```

Na imagem acima, há um se-senão que retorna *one* ou *two* que são funções, então dessa maneira, existem vários casos em que é necessário checar se um expressão gera ou não um símbolo.

Para validar, tipos de várias expressões devem ser consideradas, por exemplo nesse caso que mostrei acima, temos que *surprise* recebe uma função de alta ordem chamada *fn*, então temos que avaliar se a expressão *se-senao* gera símbolo, em seguida pegamos esse símbolo e validamos que tem a assinatura que precisamos para passar para a função surprise

Sendo assim, para implementar *getSymbolOrNull* precisamos considerar todos os casos onde uma expressão pode gerar símbolo. Esses casos são, *lambda expressões, expressões de bloco, expressão se-senao* e expressão *identifier*.

Todas as outras devem ter tipos comuns, o retorno de uma função, não pode retornar outra função, pois em na especificação de Elipses um função deve retornar inteiro, booleano, ou real.

Já bloco, se senão e lambda podem retornar um identifier, e portanto podem ser avaliados em um símbolo;

A implementação encontra-se abaixo, note como caso seja encontrar um node que não seja um ID, então chamamos getSymbolOrNull recursivamente, o caso base é o AIdExp

```java
public Symbol getSymbolOrNull(PExp node) {
    Symbol return_symbol = null;
    if (node instanceof ALambdaExp) {
        ALambdaExp e = (ALambdaExp)node;
        return_symbol = getSymbolOrNull(e.getBody());
    }
    else if (node instanceof ABlockExp) {
        ABlockExp e = (ABlockExp)node;
        return_symbol = getSymbolOrNull(e.getExp());
    }
    else if (node instanceof AIdExp) {
        AIdExp e = (AIdExp)node;
        String id = e.getIdentifier().getText();
        return_symbol = table.get(id);
    }
    else if (node instanceof AIfExp) {
        AIfExp e = (AIfExp)node;
        Symbol truthy = getSymbolOrNull(e.getTruthy());
        Symbol falsy  = getSymbolOrNull(e.getFalsy());
        // Both truthy and Falsy MUST have the same type
        // and there fore if any of them are not null
        // hey must have the same signtura, but not necessarily the same
        // symbol, but for all purposes we only call the functin to get
        // the Signature / type not the id itsself
        if (truthy != null ) {
            return_symbol = truthy;
        }
        else if (falsy != null ) {
            return_symbol = falsy;
        }
    }
    // Cant do node instanceof ACallExp because a call expr
    // can't ever return a ID and therefore can't have symbol

    // Any binary operator return an R-Value and therefore
    // can't have a symbol storage
    // Unary operation as well, negate/not an id returns an R-Value

    return return_symbol;
```

Algo similar foi implementado para fazer a inferência de tipos com a função *getType*, caso um tipo seja composto, chamamos getType recursivamente até cair num caso base que seria um número literal, ou booleano, se for uma funçao, retornamos o valor de retorno dessa função. Para mais detalhes ver 'getType' no arquivo TypeInference.java. Mais será comentado na sessão Análise Semântica.

7. Erros e Warnings

As classes que encapsulam possíveis tipos de *erros* e *warnings* são apenas enum de Java com uma mensagem padrão. Um erro específico é um tipo de erro somado a uma mensagem customizada para ser mais especializado.

Por exemplos um construtor de erro Semântico seria:

```
public SemanticError(SemanticErrorType error_type, String custom_msg) {
    this.type = error_type;
    this.custom_msg = custom_msg;
}
```

Já os tipos de erros são este enum:

```
public enum SemanticErrorType {
    NONE(""),
    DUPLICATE_ENTRY("Duplicate entry"),
    NO_ENTRY("No entry point"),
    ALREADY_DECLARED("Identifier already declared in this scope"),
    UNDECLARED("Use of undeclared indentifier"),
    UNDEFINED_FUNCTION("Undefined function"),
    DUPLICATE_VARIABLE("Duplicate variable"),
    INCOMPATIBLE_TYPES("Incompatible types"),
    INCOMPATIBLE_RETURN_TYPE("Incompatible Return Type"),
    DUPLICATE_FUNCTION("Duplicate function"),
    INCORRECT_NUMBER_OF_ARGUMENTS("Incorrect number of arguments"),
    INCORRECT_USE_OF_ARGUMENTS("Incorrect use of arguments in a function call"),
    SIGANTURE_ON_ENTRY("Higher order function on entry"),
    INVALID_OPERATION("Invalid operation"),
    INVALID_OPERATION_ON_FN_REFENRECE("Invalid operation with function reference operand"),
    INVALID_MODULUS_OPERATION("Invalid use of '%' modulus operator");

    private String message;
```

8. Análise Semântica

Na análise semântica, utilizamos todas as classes citadas, e mais uma uma classe filha *DepthFirstApdater* onde definimos diversos *overrides out* e *in* de nodes.

Nesses *overrides in* fazemos checagem ou entramos em escopos e no *override out* saímos do escopo ou algum outro check de algo. Tudo depende de qual tipo de nó estamos. Houve casos de precisar mudar a ordem de busca, então houve *override case* além dos *in* and *out*.

8.1 Funções

No *in* da declaração de uma função, precisamos adicionar o seu identifier com sua assinatura para a tabela de símbolos e em seguida entrar no novo escopo

```
// << [EXP]
@Override
public void inADeclFunc(ADeclFunc node) {
    String name = node.getIdentifier().getText();

    boolean ok = table.add(name, new Symbol(
        name, node.getType().toString(), node.getParam()
    ));

    if (!ok) {
        errors.add(new SemanticError(SemanticErrorType.ALREADY_DECLARED
    }
    assert table.existsInGlobalScope(name);

    Symbol s = table.get(name);

    ElipLogger.debug( name + " signature -> "  + s.getSignature());

    table.enterScope();
```

Também checamos se já existe uma função como entrada, se sim, adicionamos o erro de duplicação de funções de entrada;

```
TKwEntry entry = node.getKwEntry();
if (entry != null) {
    if (flags.entry_found == false) {
        flags.entry_found = true;

        first_entry_msg =
            "\n\t First defined in line "
            + entry.getLine()
            + " position " + entry.getPos()
            + " on function '" + node.getIdentifier() + "' with params '" + node.getParam() + " '"
        ;
```

Também proibimos o uso de funções de alto nível na função de **entrada**, pois não há como passar um função para a entrada através do *command line*. Então é um erro semântico, mas só caso seja a função de entrada. Para isso fazemos um pequeno teste de todos os parâmetros e vemos se é do tipo **ASignatureParam**, se sim reportamos erro “SIGNATURE_ON_ENTRY”.

```
  for (PParam param : node.getParam()) {
      if (param instanceof ASignatureParam) {
          errors.add(
              new SemanticError( SemanticErrorType.SIGANTURE_ON_ENTRY, node.getIdentifier(), " , ther
          );
      }
  }
```

O erro gerado nesse caso é algo assim

```
[elip] error: test/codegen/ir.elip[12,13]:  Higher order function on entry   , th
eres no way to expect a higher order function given from cmd line! real multiplic
arPorPi inteiro random real
```

No *out* da declaração de uma função, fazemos inferência de tipo no retorno dessa função e checamos se os dois tipos são diferentes iguais, ou se o tipo retornado é uma signature, reportamos errors nesses dois casos:

```
if (identifier_symbol.isFunction()) {
    errors.add(new SemanticError(
        SemanticErrorType.INCOMPATIBLE_RETURN_TYPE, node.getIdentifier(),
        " on '" + func_id + "' expected: '" + expect
        + "' got function signature '" + identifier_symbol.getSignature() + "'"
    ));
}
```

Mas no caso do tipo retornado poder sofrer coerção, então reportamos um warning, caso contrário é um erro. Por exemplo, se a função espera um **inteiro** mas fora retornado um **real**, permitimos isso, e geramos código fazendo *cast* de *float* para *int*, porém isso gera um *warning* de elip.

```
if ( !inference.canApplyCoercion(expect, got)) {
    errors.add(new SemanticError(
        SemanticErrorType.INCOMPATIBLE_RETURN_TYPE, node.getIdentifier(),
        " on '" + func_id + "' expected: '" + expect + "' got: '" + got + "'"
    ));
}
else if (expect != got) {
    warnings.add( new SemanticWarning(node.getIdentifier(),
        "on '" + func_id+  "' Coercion from given '" + got +"' to " + " '" + expect + "'"
    ));

}
```

e por fim saímos do escopo

```
table.exitScope();
```

8.2 Bloco

Entramos em escopo no *in* e saímos no *out*

```
@Override
public void inABlockExp(ABlockExp node) {
    // Is the resposability of whoever need blockexp to
    // make shure it's the correct type
    ElipLogger.debug("enteredd scope in a block");
    table.enterScope();
}
```

```
@Override
public void outABlockExp(ABlockExp node) {
    //table.printAllScopes();
    table.exitScope();
    ElipLogger.debug("exited scope in a block");
}
```

8.3 Declaração de Constante

No *in* adicionamos ao escopo atual o identificador dessa constante, e checamos se já existe nesse escopo, se sim, isso é um erro de "já declarado"

```
@Override
public void inADeclConst(ADeclConst node) {
    assert node.getIdentifier() != null;

    String name = node.getIdentifier().getText().strip();

    boolean exist = table.existsInCurrentScope(name);

    ElipLogger.debug(name + exist );
    if (exist) {
        errors.add(new SemanticError(
            SemanticErrorType.ALREADY_DECLARED, node.getIdentifier(),
            "on '" + node.toString() + "'"
            ));
        return;
    }

}
```

No *out* fazemos inferência de tipo novamente para checar se a constante passado é do tipo declarado. Checamos caso a expressão dessa constante tenha sido uma função, por exemplo, e reportamos erros.

```
if (identifier_symbol.isFunction()) {
    errors.add(new SemanticError(
        SemanticErrorType.INCOMPATIBLE_TYPES, node.getIdentifier(),
        " on '" + name + "' declaration const: expected: '" + expect + "' got
    ));
}
```

Da mesma maneira na função verificamos se pode haver coerção, se sim, permitimos com um warning:

```
if ( !inference.canApplyCoercion(expect, got)  ) {
    errors.add(new SemanticError(
        SemanticErrorType.INCOMPATIBLE_TYPES, node.getIdentifier(),
        " on '" + name + "' declaration const: expected: '" + expect + "' got: " + got
    ));
}
else if (expect != got) {
    warnings.add( new SemanticWarning(
        node.getIdentifier(),
        "on '" + name+  "' Coercion from given '" + got +"' to " + " '" + expect + "'"
    ));
}
```

8.4 Parâmetros

No *override in* dos parâmetros, adicionamos a tabela de símbolos e caso já exista reportamos erro que variável duplicada. Esse exemplo abaixo é para parâmetro simples.

O mesmo foi feito para parâmetros de *assinatura*, ou seja para funções de alto nível.

```java
@Override
public void inATypeParam(ATypeParam node) {
    if (node.getIdentifier() == null) {
        return;
    }


    Token t = node.getIdentifier();
    String name = node.getIdentifier().getText();

    assert !table.existsInCurrentScope(name);

    // If the current parameters already exists in current scope
    // i.e. the function scope, then it's a error, it is an already
    // declared parameter
    if (table.existsInCurrentScope(name)) {
       errors.add(new SemanticError(
            SemanticErrorType.ALREADY_DECLARED, t, " on parameter " + node.toString()
        ));
    }
    else {
        table.add(name, new Symbol(name, node.getType().toString()));

    }
}
```

8.4 Lambda

Para expressões *lambda* a inferência de tipo é obrigatóri, e a ordem de inferência é diferente da DepthFirst, pois devemos considerar o resultado do lambda por último, antes devemos saber os tipos de expressões passadas, para depois adicionar os identifier a tabela de símbolo.

Por exemplo esse lambda abaixo:

```
(lambda(r|p|q):( se(r) entao (p) senao --q )[verdadeiro|2,1|3,1])
```

Primeiro devemos avaliar os argumentos à direita, depois que sabemos que o primeiro argumento é verdadeiro e portanto é **booleano**, o segundo é 2,1 e portanto **real**, e o terceiro é 3,1 e também é **real**.

Associamos cada valor e tipo com seu identificador e adicionamos a tabela de símbolo e só depois avaliamos a expressão interna.

Deve ser essa ordem pois pode ser que os argumentos sejam outra lambda ou algo mais complexo e então haverá outra caminhada na árvore antes de retornar e saber o tipo.

No *in* lambda checamos se a quantidade de argumentos é o mesmo da quantidade de identificados e dizemos qual é maior ou menos caso seja diferente. E por fim entramos no novo escopo

```java
@Override
public void inALambdaExp(ALambdaExp node) {
    List<PExp> args = node.getArgs();
    List<TIdentifier> ids  = node.getId();
    int ids_count = ids.size();
    int args_count = args.size();

    if (args_count != ids_count) {
        if (ids_count > 0) {
            errors.add(new SemanticError(
                SemanticErrorType.INCORRECT_NUMBER_OF_ARGUMENTS, ids.get(0),
                "on lambda expectd: " + ids_count + ", got: " + args_count
            ));
        }
        else {
            errors.add(new SemanticError(
                SemanticErrorType.INCORRECT_NUMBER_OF_ARGUMENTS,
                "on lambda expectd: " + ids_count + ", got: " + args_count
            ));
        }
    }

    table.enterScope();
    ElipLogger.debug("lambda entered scope");

}
```

No *override case* lambda, fazemos o que foi dito antes, caminhamos na árvore pelos identificados seguido dos argumentos e adicionamos a tabela, checando por repetidos. No override out, saímos do escopo.

8.5 Chamada de funções

No *override in*, checamos se a função existe no escopo global, se não existir é erro de uso de função não declarada, e depois usamos aquelas funções da classe *Signature* para testar se o uso dos argumentos condiz com a assinatura da função sendo chamada.

```java
// >> [EXP]
 @Override
 public void inACallExp(ACallExp node) {
     String name = node.getId().getText();
     Token t = node.getId();


     if (!table.exists(name)) {
         errors.add(new SemanticError(SemanticErrorType.UNDECLA
         // need to end right here;
         // can't stablish anything else
         caseEOF(null);
     }

     // we can be sue it exists because we've checked it
     table.get(t.getText()).setToken(t);

     SemanticError error = inference.isCorrectUseOfArgs(name, n
     if ( error != null) {
         errors.add(error);
     }
     // the function is defined
 }
```

8.6 Default

Todo o resto fazemos a inferência de tipo, para todo tipo de operação, *, > , % etc.... Essa checagem foi implementada na classe TypeInference que a medida que infere, ele tem capacidade de gerar erros e warnings à medida que recursivamente checa os nós. Por exemplo, para todo identificador usado, checa-se se ele está na tabela, senão é um uso de identificador não declarado.

```
else if(node instanceof AIdExp) {
    String name = ((AIdExp)node).getIdentifier().getText();
    if(!table.exists(name)) {
        Token t = ((AIdExp)node).getIdentifier();
        errors.add(new SemanticError(
            SemanticErrorType.UNDECLARED,t, " " + t ));
        return_symbol  = Symbol.Type.UNKOWN;
    }
    else {
        return_symbol  = table.get(name).getType();
    }
}
```

Na expressão se-senao, permitimos na expressão de condição apenas booleanos, então reportamos erro se qualquer outro tipo de expressão seja utilizado ali, usando inferência.

Informamos erros customizados variando de acordo com o tipo de operação e sobre quais tipos estão operando. Para o caso da expressão de condição nao existe coerção possível, só pode ser booleano.

Depois checamos se o tipo da *senao-branch* é igual *entao-branch*, todo possível caminho deve retornar um tipo igual, ou seja, não podemos ter o caso se uma branca retornar um real e outro retornar um função.

Retornar um ID para uma função em um expressao se-senao é permitido, mas só se ambas branches retornem uma função com a mesma assinatura. Então vários tipos de erros de incompatibilidades podem ocorrer.

Toda operação binária também é checado os tipos, o operador %, assim como no C, só é definido se ambas expressões, esquerda e direita, forem inteiro. Caso não seja é um erro.

```
if( node instanceof AModExp
&& ((left !=  Type.INT) || (right != Type.INT))
) {
    errors.add(new SemanticError(SemanticErrorType.INVALID_MODULUS_OPERATION, ". Modulus
        + " left: '"  + left_node   + "'"
        + " right: '" + right_node  + "'"
    ));
    return Symbol.Type.UNKOWN;
}
```

Todo operador de relação, ex: 2>3 gera um booleano, mas se por acaso na direita ou esquerda um tipo que não seja inteiro ou real, um erro ocorre.

Da mesma maneira, ‘e’ ‘ou’, operadores de lógica, só podem ser usados em booleano, qualquer outra tentativa é um erro

Para todos esses casos de operadores, mesmo os unários, checamos se estamos operando em um função e retornamos um erro personalizado

9. Análise Semântica na Prática

Muitos detalhes foram omitidos por brevidade , então agora apresento vários código fonte de elipses e os respectivos erros que geram.

9.1 Entrada Duplicada

Arquivo fonte : "test/semantic/incorrect_duplicate_entry.elip"

```
> semantic > = incorrect_duplicate_entry.elip
(inteiro baskara(real a | real b):(
    2
))

entrada ( inteiro principal () :
    (2)
)

entrada (inteiro mais_principal(real x | real y):(
   baskara(2,3 | 3,2) +
   baskara(0b100011 | 3,2)
))
```

Esse programa possui duas entradas

```
[elip] success:  Parsed into AST with success.
[elip] error: test/semantic/incorrect_duplicate_entry.elip[9,1]:  Duplicat
e entry
        First defined in line 5 position 1 on function 'principal ' with
params '[] '
[elip] warning: test/semantic/incorrect_duplicate_entry.elip[11,4]:  on '0
b100011 ' the 1th argument ' was coerced from 'inteiro' to 'real'
```

Note o Compilador detecta a linha e posição que a função duplicada está, além de quando foi definido primeiro, e qual a função que foi definida.
Ainda o **elipc** gera um warning no primeiro argumento passado para bhaskara, pois transformamos o inteiro em real implicitamente.

9.2 Já Definido

Arquivo fonte : "test/semantic/incorrect_already_defined.elip",

```
1   entrada ( inteiro already_defined_parameters(inteiro a | inteiro a) :
  1     (2)
  2 )
  3
  4 (inteiro already_defined_const_declaration() : (
  5       ( (const inteiro x (2)) (const real x(2,4)) x + x)
  6 ))
  7
  8
  9 # Function already defined
 10 ( inteiro already_defined_func() :(1))
 11 ( inteiro already_defined_func() :(1))
 12
 13 (inteiro already_defined_nested_const_declaration() : (
 14       (
 15         (const inteiro x (
 16             (
 17                 # No problem here, real x shadow the earlier int x
 18                 (const real x(2,4))
 19                 # Problem occur here, should get an error on booleano x
 20                 (const booleano x( verdadeiro ))
 21                 x
 22             )
 23         ))
 24
 25         x
 26       )
 27 ))
 28
```

O **elipc** detecta vários erros de já declarado, na linha 1, temos dois inteiros com identificador repetidos no parâmetro.

Na linha 5 temos que x inteiro e x real repetido. Depois na linha 12 duas funções com mesmo nome.

Na linha 21, temos redefinição de x para booleano, note que “variable shadowing” é permitido, ou seja, definir identificador com mesmo nome, desde que esteja em outro escopo.

Na última função, x é definido como inteiro e depois em outro escopo é definido como real, isso não há problema, o problema ocorre quando no mesmo escopo tentamos declara outro x booleano.

```
[elip] error: test/semantic/incorrect_already_defined.elip[1,66]:  Identifier already
declared in this scope   on parameter inteiro a
[elip] error: test/semantic/incorrect_already_defined.elip[6,43]:  Identifier already
declared in this scope  on 'real x 2,4 '
[elip] error: test/semantic/incorrect_already_defined.elip[12,11]:  Identifier already
 declared in this scope   error on inteiro already_defined_func 1
[elip] error: test/semantic/incorrect_already_defined.elip[21,33]:  Identifier already
 declared in this scope  on 'booleano x verdadeiro '
```

9.3 Não declarado

Arquivo fonte: "test/semantic/incorrect_undeclared.elip",

```
27
26 (inteiro use_of_undeclared_identifier_x(real b) : (
25     ( b % x)
24 ))
23
22 (inteiro existent_function() : (
21     2
20 ))
19
18 (inteiro undeclared_decl_const_x_y() : (
17       (
16         (const inteiro z (
15             (
14                 (const real w(2,4))
13                 x + y
12             )
11         ))
10         x
 9       )
 8 ))
 7
 6 entrada ( inteiro use_of_undeclared_func() : (
 5     # No problem here
 4     existent_function()
 3     # Problem here
 2     + non_existent_function(2)
 1 ))
29
```

Primeiros erros informa que ta usando x não declarado
e depois informa que o módulo operador está sendo usado em tipo não integral.

```
[elip] success:  Parsed into AST with success.
[elip] error: test/semantic/incorrect_undeclared.elip[4,11]:  Use of undeclared
 indentifier   x
[elip] error: test/semantic/incorrect_undeclared.elip Invalid use of '%' modulu
s operator  . Modulus operator of non-integral types is undefined left: 'b ' ri
ght: 'x '
```

E como não se sabe o tipo de retorno dessa primeira função, também recebemos o erro de tipo incompatível entre o retorno dessa função e o tipo esperado

```
[elip] error: test/semantic/incorrect_undeclared.elip[3,10]:  Incompatible Retu
rn Type   on 'use_of_undeclared_identifier_x'' expected: 'inteiro' got: 'unkown
'
```

Na segunda função “undeclared_decl_const_x_y temos uso de x e y não declarados

```
[elip] error: test/semantic/incorrect_undeclared.elip[16,17]:  Use of undeclare
d indentifier   x
[elip] error: test/semantic/incorrect_undeclared.elip[16,21]:  Use of undeclare
d indentifier   y
```

Depois o uso de função não declarada e outros erros de operação com ‘unkown’

```
[elip] error: test/semantic/incorrect_undeclared.elip Incompatible types   bina
ry Operation with Unkown Type  left: 'x ' right: 'y '
[elip] error: test/semantic/incorrect_undeclared.elip[13,24]:  Incompatible typ
es   on 'z' declaration const: expected: 'inteiro' got: unkown
[elip] error: test/semantic/incorrect_undeclared.elip[19,9]:  Use of undeclared
 indentifier   x
[elip] error: test/semantic/incorrect_undeclared.elip[11,10]:  Incompatible Ret
urn Type   on 'undeclared_decl_const_x_y'' expected: 'inteiro' got: 'unkown'
[elip] error: test/semantic/incorrect_undeclared.elip[27,7]:  Use of undeclared
 indentifier   trying to call undeclared function non_existent_function
```

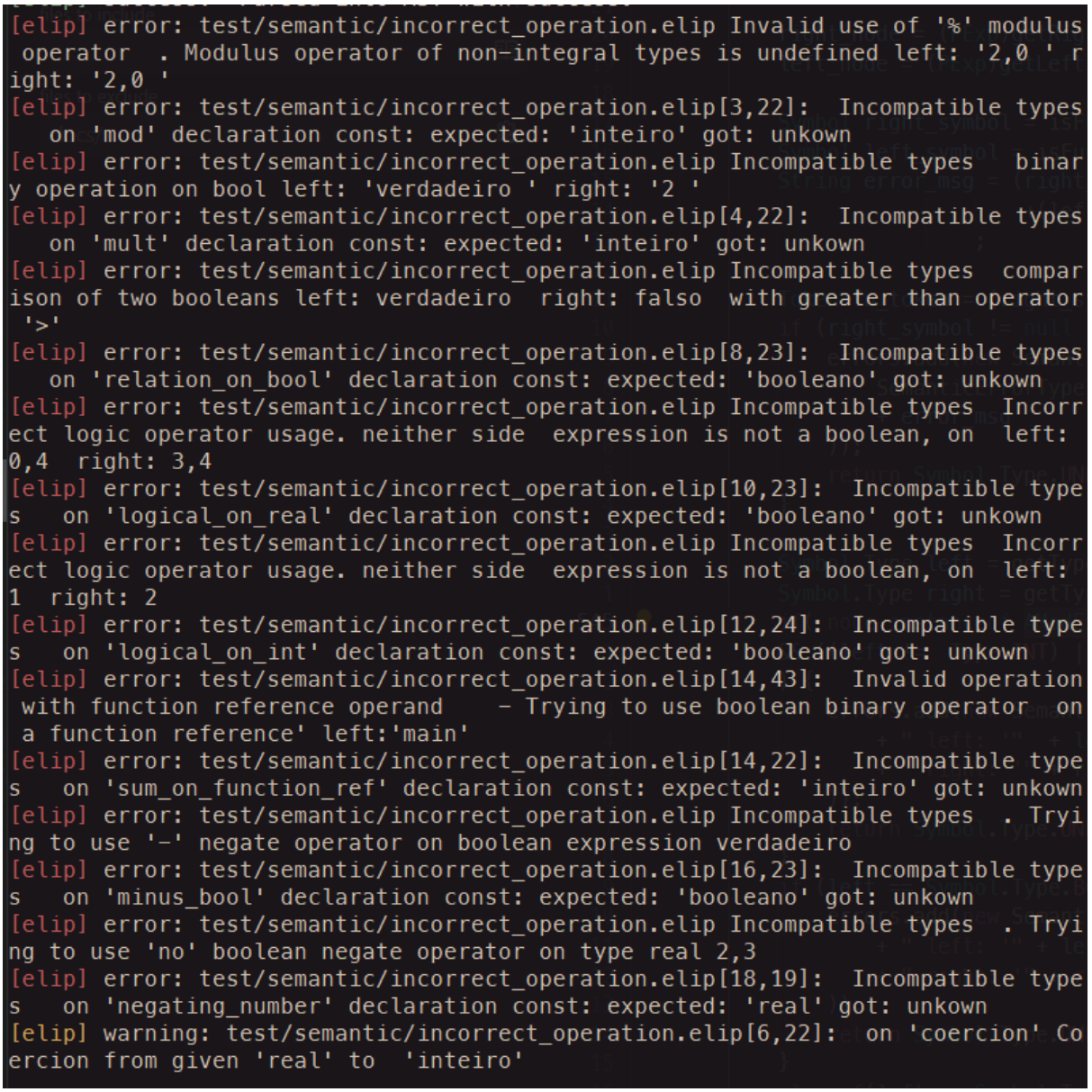
Como disse antes o sistema para permitir mais erros acumularem é propagar o tipo ‘unkown’ qualquer operação com uma operação entre tipos diferentes que não seja definida, gera unkown e qualquer coisa com unkown é unkown também.

9.4 Operações Incorretas

Arquivo fonte: "test/semantic/incorrect_operation.elip"
Uma série de operações com seus erros comentados

```
entrada (inteiro main(): (
    # modulus operation on non-integral types
    ( (const inteiro mod  (2,0 % 2,0))
      (const inteiro mult  (verdadeiro * 2))
      # warning should occur here
      (const inteiro coercion (2,3 * 4))
      # relational operador on boolean type
      (const booleano relation_on_bool (verdadeiro > falso))
      # logic operator on real type
      (const booleano logical_on_real (0,4 e 3,4))
      # logic operator on integer type
      (const booleano  logical_on_int (1 ou  2))
      # use of binary operator with function reference
      (const inteiro sum_on_function_ref (main +  2))
      # minus a boolean
      (const booleano minus_bool (-verdadeiro))
      # logic negating a numbers
      (const real negating_number ( nao 2,3))
        2
    )
))
```

Nada muito diferente do comentado no código fonte, cada erro está presente, com sua linha e posição e outras informações interessantes.



9.5 Erros na expressão se-senao

Arquivo fonte:"test/semantic/incorrect_if_expr.elip"

```
(booleano func_ref_as_condition(): (
    se (func_ref_as_condition) entao verdadeiro senao (2 >3)
))

(booleano truthy_as_func_ref(): (
    se (falso) entao truthy_as_func_ref senao (verdadeiro)
))

(inteiro falsy_as_func_ref(): (
    se (falso) entao 2 senao (lambda ():(falsy_as_func_ref)[])
))

(inteiro cond_as_real(): (
    se (1,0) entao 2 senao (3)
))

# if  expr must have the same type on all branches
(real different_type_on_branch(): (
    se (falso) entao 2 senao (3,3)
))

entrada (inteiro main(): (
   (2)
))
```

Primeira função, tem um erro dizendo que está usando uma referência a uma função como booleano na condição do se-senao.

Depois temos tipos diferentes nas branches na segunda função.

Na terceira função temos duas branches um que é lambda onde seu tipo é inferido para “signatura booleano ()” que é diferente da outra branch que é booleano.

Depois usando real na condição e outros erros:

```
[elip] success.  Parsed into AST with success.
[elip] error: test/semantic/incorrect_if_expr.elip[2,9]:  Incompatible types  .
 Trying to use function reference as a boolean on if-expression func: 'func_ref
_as_condition' with signature booleano ()
[elip] error: test/semantic/incorrect_if_expr.elip[1,11]:  Incompatible Return
Type   on 'func_ref_as_condition'' expected: 'booleano' got: 'unkown'
[elip] error: test/semantic/incorrect_if_expr.elip[6,22]:  Incompatible types
. both branches of the the if-expression has to have the same type  then-branch
: 'truthy_as_func_ref' with signature booleano ()  else-branch: with type 'bool
eano'
[elip] error: test/semantic/incorrect_if_expr.elip[5,11]:  Incompatible Return
Type   on 'truthy_as_func_ref' expected: 'booleano' got function signature 'boo
leano ()'
[elip] error: test/semantic/incorrect_if_expr.elip[9,10]:  Incompatible Return
Type   on 'falsy_as_func_ref' expected: 'inteiro' got function signature 'intei
ro ()'
[elip] error: test/semantic/incorrect_if_expr.elip Incompatible types  . Trying
 to use non boolean expression of if condition  on '1,0 2 3 'when type is 'real
''
[elip] error: test/semantic/incorrect_if_expr.elip[13,10]:  Incompatible Return
 Type   on 'cond_as_real'' expected: 'inteiro' got: 'unkown'
[elip] error: test/semantic/incorrect_if_expr.elip Incompatible types   in a if
 expression both possible values must have the same type  on 'falso 2 3,3 '  th
en-branch: with type 'inteiro'  else-branch: with type 'real'
[elip] error: test/semantic/incorrect_if_expr.elip[18,7]:  Incompatible Return
Type   on 'different_type_on_branch'' expected: 'real' got: 'unkown'
```

9.6 Uso incorreto de Argumentos

Arquivo fonte: "test/semantic/incorrect_args.elip"

Na primeira função deste arquivo, temos

```
(inteiro exemploAltaOrdem (inteiro digito | real multiplicarPorPi ( inteiro random() | real)):
    (
        #bloco_exp
        (
            (const real pi (3,1415))
            multiplicarPorPi(2| pi)
        )
    )
)
```

multiplicarPorPi espera uma função como primeiro argumento, mas passamos um inteiro, gera o seguinte erro :

```
[elip] success:  Parsed into AST with success.
[elip] error: test/semantic/correct_signature_check.elip[7,13]:  Incorrect use of argument
 in a function call  on function 'multiplicarPorPi' the 1th arguments '2 ' is not a funct
ion which is is different from expected: inteiro ()
```

Na função high_order deste arquivo, passamos um número incorreto de argumentos

```
(inteiro high_order(inteiro func(inteiro|inteiro)) : (
    # correct
    func(2|3)
    *
    # incorrect
    func(2)
))
```

O compilador detecta e informa o número de argumentos que esperavam e quantos foram passados

```
[elip] error: test/semantic/incorrect_args.elip[25,5]:  Incorrect use of a
rguments in a function call  on function 'func': missing arguments, expect
ed: 2 arguments  got: 1
```

Nessa função *high_order_complex*, vemos ela chamar a função f com dois argumentos, a função *add* e *add_real*, porém *add_real* tem uma assinatura diferente do esperado, **elipc** gera o erro correctamente

```
(inteiro add(inteiro x | inteiro y) : (
    x+y
))

(real add_real(real x | real y) : (
    x+y
))
```

```
(inteiro high_order_complex( inteiro f(inteiro g(inteiro|inteiro) )) : (

    # correct
    f(add)
    *
    # incorrect
    f(add_real)
))
```

Erro diz que la linha 46 esperava *inteiro(inteiro(inteiro|inteiro))* como argumento, mas recebeu *real (inteiro(inteiro|inteiro))*

```
[elip] error: test/semantic/incorrect_args.elip[46,5]:  Incorrect use of
 arguments in a function call  on function 'high_order_complex' the 1th
argument:  real (inteiro (inteiro|inteiro)) is different from expected :
 inteiro (inteiro (inteiro|inteiro))
```

9.6 Tipos Incompatíveis

Arquivo fonte "test/semantic/incorrect_return_type.elip".

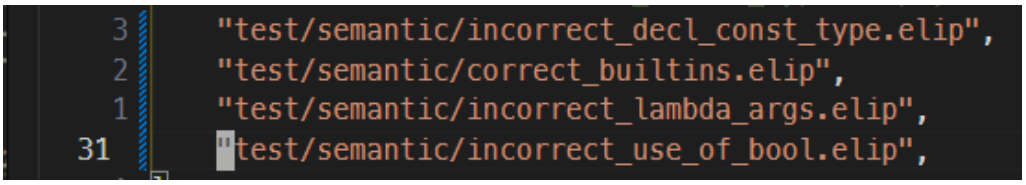
Enfim, não precisa ler todos os erros, mas basicamente será comentado abaixo oque esperado de erro ou não

```
entrada (inteiro incorrect_return_type() : (
    # returning bool instead of int
    verdadeiro
))

(inteiro correct_return_type() : (
    # returning correct type, should not have problems
    2
))


(real expects_real() : (
    # incorrect returning integer
    2
))

(real expects_real_() : (
    # correct returning integer + real which results in real
    2 + 0,1
))

(real expects_real_but_returning_function() : (
    # correct returning integer + real which results in real
    expects_real
))

(real expects_real_but_returning_function_inside_a_lambda() : (
    # correct returning integer + real which results in real
    (lambda ():(expects_real)[])
))
```

```
[elip] error: test/semantic/incorrect_return_type.elip[2,18]:  Incompati
ble Return Type   on 'incorrect_return_type'' expected: 'inteiro' got: '
booleano'
[elip] error: test/semantic/incorrect_return_type.elip[23,7]:  Incompati
ble Return Type   on 'expects_real_but_returning_function' expected: 're
al' got function signature 'real ()'
[elip] error: test/semantic/incorrect_return_type.elip[28,7]:  Incompati
ble Return Type   on 'expects_real_but_returning_function_inside_a_lambd
a' expected: 'real' got function signature 'real ()'
[elip] warning: test/semantic/incorrect_return_type.elip[13,7]:  on 'exp
ects_real' Coercion from given 'inteiro' to  'real'
```

Temos outros arquivos de teste e erros que foram checados extensivamente, como de declaração de constante, possíveis erros ou uso correto de função que já vem na linguagem como potência, seno e cosseno, mais testes com uso incorreto em lambdas com exemplos bem complexos e mesmo assim o sistema não quebrou ainda, ou seja, detecta os erros sem falha.

O outros arquivos para verificação de erros são :

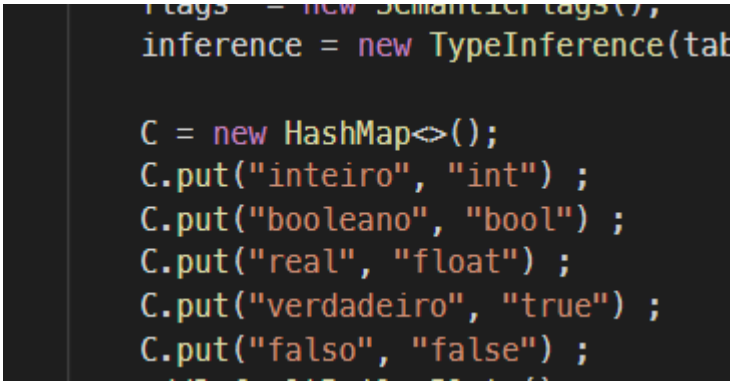


10. Geração de Código

O gerador de código também é uma classe filha de *DepthFirstAdapter* e ela faz o mesmo que a classe de análise semântica em termos da tabela de símbolos.

Entramos em escopos corretamente e marcamos tipos, e ID's em forma de símbolos para o programa. Da mesma maneira, fazemos interferência de tipo. A diferença é que agora precisamos traduzir os tipos para os tipos do C.

O tipos simples são traduzidos com um hashmap simples



O tipos assinatura são recursivos então também criamos uma função recursiva que traduz uma assinatura para um tipo em C que é um ponteiro para uma função

```
String fromSignatureToCType(Symbol.Signature s,String func_name) {
    Symbol.Signature signature = s;
    StringBuilder sb = new StringBuilder();
    sb.append( C.get(signature.getReturnType().toString()));
    if(func_name != null){
        sb.append("(*" + func_name + ")");
    }
    else {
        sb.append("(*)");
    }
    sb.append("(");
    List<Symbol.SignatureParam> parameters = signature.getParameters();
    for (int i = 0; i < parameters.size(); i++) {
        Symbol.SignatureParam param = parameters.get(i);
        if (i > 0) {
            sb.append(", ");
        }
        if (param.isSignature()) {
            Symbol.Signature param_sig = param.getSignature();
            sb.append(fromSignatureToCType(param_sig,null));
        }
        else {
            sb.append(C.get(param.getType().toString()));
        }
    }
    sb.append(")");
    return sb.toString();
}
```

nome da função no ponteiro

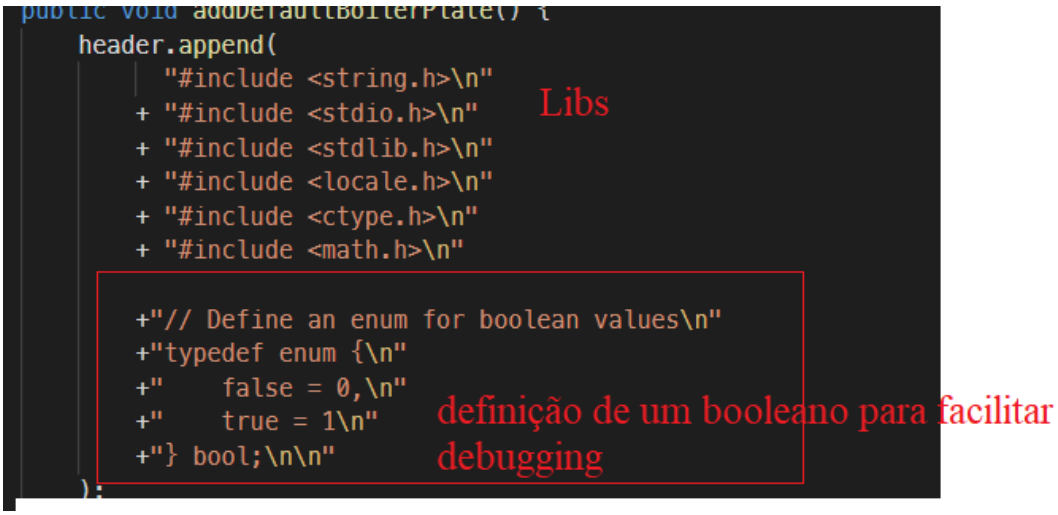
Chamada recursiva caso o parametro seja uma assinatura

No caso dessa classe "*walker*" chamada **CCodeGenerator**, é necessário fazer *override* do *case*, pois precisamos adicionar texto na metade do código. A todo momento estamos escrevendo em três pedaços principais de uma IndetendStringBuilder.
O body, header e footer:

```
    public IndentedStringBuilder header   = new IndentedStringBuilder();
    public IndentedStringBuilder body     = new IndentedStringBuilder();
    public IndentedStringBuilder footer   = new IndentedStringBuilder();
```

No header definimos funções padrões, e #include de libraries.

```
public void setUpBuiltins(IndentedStringBuilder header,
    header.append(
        "float elip_cosseno(float x);\n"
        +"float elip_seno(float x);\n"
        +"float elip_logaritmo(float x);\n"
        +"float elip_logaritmo(float x);\n"
        +"float elip_potencia(float x, float y);\n"
    );

    buffer.append(
```



No footer, definimos os códigos que precisam vir por último, como função main de C.

Na função main chamamos a função de entrada do código fonte elipses e passamos os argumentos passados no terminal para a entrada de elipses:

```
footer.append(
    "int main(int argc, char *argv[]) {\n"
+ "    setlocale(LC_ALL, \"fr_FR.UTF-8\");\n"
+ "    char *arg;\n"
+ "    size_t arg_len;\n"
);
```

Fazemos também a transformação de string dado no como argv no terminal para o tipo equivalente de C para passar pra entrada de elipses.

A partir de agora vou dar alguns exemplos de *cases* e na próxima seção vamos ver a prática. Antes devo informar, os *cases* mais simples, todo identificador, type, número, booleano e até assinatura, é um simples tradução e depois escrever no String Builder indentada, no body.

Por exemplo, um tipo real, simplesmente transformamos de ‘,’ para ‘.’ e escrito no *body*.

```
440
439     @Override
438     public void inARealExp(ARealExp node) {
437         body.append(
436             node.getNumberFrac()
435                 .getText()
434                 .replace(',', '.')
433         );
432     }
431
```

O outros também são definidos de maneira similar

```java
@Override
public void caseATrueExp(ATrueExp node)
{
    body.append("true");
}

@Override
public void caseAFalseExp(AFalseExp node)
{
    body.append("false");
}
```

10.1 Operações binárias

Toda operação binária segue a mesma fórmula de geração de código ,que é a seguinte:

```java
    @Override
    public void caseAModExp(AModExp node) {
        inAModExp(node);
        body.append("(");
        if(node.getLeft() != null)
        {
            node.getLeft().apply(this);
        }
        body.append("%");
        if(node.getRight() != null)
        {
            node.getRight().apply(this);
        }
        body.append(")");
        outAModExp(node);
    }
```

Abrimos parênteses, navegamos recursivamente a expressão à esquerda, depois escrevemos o operador, navegamos a expressão a direta e em seguida fechamos parêntesis.

Os parênteses é para caso a especificação da linguagem mude alguma prioridade de operador, não deveríamos que se importar com ajeitar a geração de código

10.2 Operações Unários

Quase da mesma maneira, porém só temos uma expressão. OBS: sempre estamos assumindo que o código está semanticamente correto, pois já passou pelo analisador semântico

```
@Override
public void caseANegativeExp(ANegativeExp node) {
        inANegativeExp(node);
        body.append("(-(");
        if(node.getExp() != null)
        {
            node.getExp().apply(this);
        }
        body.append("))");
        outANegativeExp(node);
}
```

10.2 Chamada de Funções

Navega para o identificador que irá escrever no body o nome do ID, depois abre parênteses, seguido do *apply* nos parametros com vírgulas separando e por fim o fecha parêntesis

```
    @Override
    public void caseACallExp(ACallExp node) {
        inACallExp(node);
        if(node.getId() != null)
        {
            node.getId().apply(this);
        }
        body.append("(");
        {
            List<PExp> copy = new ArrayList<PExp>(node.getArgs());
            int len = copy.size();

            for (int i = 0; i <  len; i++) {
                PExp e = copy.get(i);

                e.apply(this);

                if (i != len -1) {
                    body.append(", ");
                }
            }
        }
        body.append(")");
        outACallExp(node);
    }
```

10.3 Blocos, Lambdas e Ifs

Todos eles são feitos de maneira muito parecida, como bloco em C.

O código é um pouco grande, mas a ideia é simples, em vez de escrever no *body*, escrevemos no bloco que estamos, ou que é '{"}' ou if {} else{}.

À medida que vamos entrando em um nó que é um bloco, lambda, ou if, aumentamos a indentação do IndentedStringBuilder que comentamos, salvamos um index de onde paramos para adicionar a nova string gerada por outros nós nesse local novamente.

No exemplo de fibonacci que mostramos antes, a expressão se-senao é traduzida para C como um bloco, considerando que colisões já estão tratadas no semântico, não há problema em fazer em bloco. Fazemos a inferência do tipo do if e fazemos o mesmo para lambda e bloco.

```
int if_0;
{
    if (((elip_x<1)||(elip_x == 1))) {
        if_0 = elip_x;
    } else {

        int block_0;
        {
            block_0 = (elip_fibonacci((elip_x-1)) + elip_fibonacci((elip_x-2)));
        }
        if_0 = block_0;
    }
}
```

No caso do lambda o tipo é bem definido apesar de ser implícito, então há inferência de tipo a parte dos argumentos passados.

11. Geração de Código na prática, exemplos mais complexos

Nessa sessão apenas exemplifico os casos mais complexos que a geração de código consegue lidar.

11.1 Lambda passado como argumento para outro lambda.

```
entrada (inteiro other(inteiro x): (
    (lambda(x|y|z):( se(x) entao (z) senao --3,1 )
    [
        verdadeiro
        |2
        |(lambda(r|p|q):( se(r) entao (p) senao --q )[verdadeiro|2,1|3,1])
    ])
)
)
```

```
float lambda_1;
{
    bool elip_x = true ;
    int elip_y = 2 ;

    float lambda_3;
    {
        bool elip_r = true ;
        float elip_p = 2.1 ;
        float elip_q = 3.1 ;

        float if_1;
        {
            if (elip_r) {

                float block_1;
                {
                    block_1 = elip_p;
                }
                if_1 = block_1;
            } else {
                if_1 = (-((-(elip_q))));
            }
        }
        lambda_3 = if_1;
    }
    float elip_z = lambda_3 ;

    float if_2;
    {
        if (elip_x) {

            float block_2;
            {
                block_2 = elip_z;
            }
            if_2 = block_2;
        } else {
            if_2 = (-((-(3.1))));
        }
    }
    lambda_1 = if_2;
}
int return_data = lambda_1;
```

Código gerado. Note que o tipo é inferido para ser float. e ainda resolvemos o lambda do argumento PRIMEIRO, para depois resolver o primeiro lambda o que define os identificadores x,y,z;

Note também como os se-senao também são blocos com if e blocos também são blocos com nome block_n.

Considere a função de auto nível “home”

```
( inteiro home(inteiro mdc(inteiro md(inteiro| inteiro| inteiro))) :(
    2
))
( inteiro higher_order_lambda_wrapped_in_block():(
   (lambda (wraped) : (2) [(home)])
))
```

Ela é passada como argumento para uma lambda, e ainda esse argumento está dentro de um bloco. Mesmo assim a geração de código consegue deduzir o tipo do lambda tranquilamente

```
    return (return_data);
}
int elip_higher_order_lambda_wrapped_in_block()  {

    int lambda_0;
    {

        int(*block_1)(int(*)(int(*)(int, int, int)));
        {
            block_1 = elip_home;
        }
        int(*elip_wraped)(int(*)(int(*)(int, int, int)))  = block_1 ;
        lambda_0 = 2;
    }
    int return_data = lambda_0;
    return (return_data);
}
```

Nesse caso é um ponteiro para função com uma certa assinatura block_1;

Lambda dentro do lambda também é feito corretamente, mas o código acaba ficando grande, mas pode olhar esse código e ele compila. A versão compilada está em “test\codegen\lambda.elip.c”

```
( inteiro complex_lambda():(
    (2)*(2)
    + (lambda () : (2) [])
    + (lambda (x | y) : ( (lambda () : ((lambda () : ((lambda (x | mdc) : (x / mdc(x | 2) *2) [15 | mdc])) [])) [])+(
    + (lambda (x | mdc) : (x / mdc(x | 2) *2) [(lambda () : (2) []) | mdc])  # passing a func make problems arise
))
{--
```

Da mesma maneira conseguimos lidar com se-senao dentro do outro e retornando funções. Considere esse caso:

surprise recebe como argumento uma função, que nesse caso, pode ser *one* ou *two*

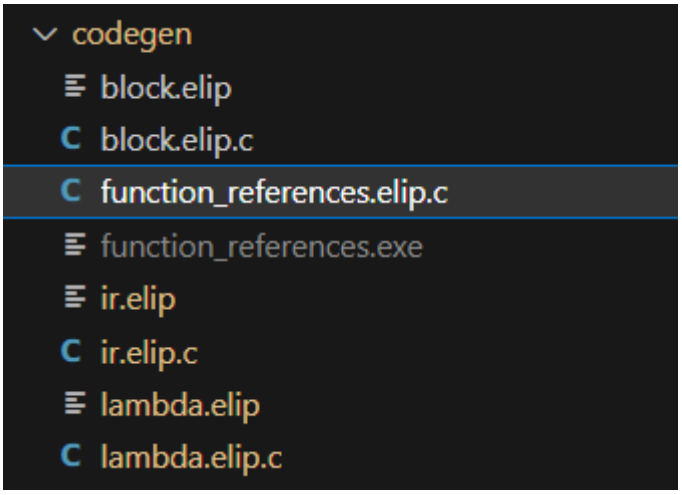
```
(inteiro one ():(1))
(inteiro two ():(2))

(inteiro surprise(inteiro fn()):(fn()))

(inteiro if_function_reference():(
    surprise(
        se (
            se (verdadeiro) entao verdadeiro senao falso
        ) entao
            one
        senao
            two
    )
))
```

Mas essa função está dentro de um se, a condição desse se-senao é outro se-senao;

Código gerado, o *if zero (if_0)* é corretamente identificado como ponteiro para função e o proximo if é identificado como booleano. Por fim o if_0 pode receber um ponteiro para *one* ou para *two*

```
int elip_if_function_reference()  {

    int(*if_0)();
    {

        bool if_1;
        {
            if (true) {
                if_1 = true;
            } else {
                if_1 = false;
            }
        }
        if (if_1) {
            if_0 = elip_one;
        } else {
            if_0 = elip_two;
        }
    }
    int return_data = elip_surprise(if_0);
    return (return_data);
}
```

Mais exemplos podem ser encontrados na pasta “test\codegen\”



12. Conclusão

Eu, Everton, amei a disciplina, quero fazer mais compiladores, computação gráfica. A parte teórica me deu saudades com Teoria da Computação, foi muito bom ver os autómato e aprender a usar essa ferramenta incrível, sablecc.

Peço encarecidamente que lembre, por favor, sobre o que conversamos das faltas, porque está marcada como falta apesar de termos conversado. E sobre o highlighting de Elipses que implementei para complementar 2 décimos na primeira unidade.

Foi um prazer trabalhar neste momento com a Sra. professora Beatriz.

Espero pegar processamento de imagens com a sra. em breve. Obrigado.