Engenharia Gramatical (4º ano de Curso)

Trabalho Prático 2

Relatório de Desenvolvimento

Henrique Costa
(pg50415)

José Pedro
(pg50525)

1 de maio de 2023

Resumo

Analisadores estáticos de código fonte são cada vez mais relevantes nas fases de desenvolvimento de *software*. Estas ferramentas avaliam os seus componentes e suas características (sem executar o *software* propriamente dito) detectando alguns erros que possam impedir a execução.

O propósito deste relatório é a descrição completa do analisador estático de código para a linguagem Hertz desenvolvida pelo grupo. Esta descrição ocorre em termos das estruturas de dados utilizadas, o seu funcionamento e como as análises são mostradas. Os objetivos deste analisador envolvem a detecção de erros à nível de variáveis (redeclarações, variáveis não declaradas, não utilizadas, etc.) e alguns sumários sobre as instruções, tipos de dados e variáveis utilizadas no programa.

O "Relatório hertz", nome dado ao resultado do analisador da linguagem Hertz, foi concebido pensando no programador Hertz. Tal relatório é apresentado através de uma página HTML elegante e de fácil visualização.

Conteúdo

Capítulo 1

Introdução

Enquadramento e contexto

A utilização de analisadores estáticos de código fonte tem, pelo menos duas vertentes. Por um lado, por motivos de segurança, através da deteção de vulnerabilidades. Por outro lado, por motivos de eficiência de desenvolvimento de *software*. A eficiência de desenvolvimento de *software* é um ponto que é bastante focado por **IDE**'s. Ao aumentar a eficiência do programador este consegue focar-se no raciocínio da resolução do problema e permite um desenvolvimento mais rápido e com menos erros. Sem estas ferramentas o programador iria perder mais tempo a compilar o programa e a fazer *debugging* de erros do compilador, aumentando o tempo em que não está a resolver o problema em questão.

Muitos problemas abordados por este último tipo de ferramentas envolve a deteção de erros de variáveis e recomendações de formatação do código. É nesse âmbito que este trabalho se foca, na criação de um analisador capaz de dar informação relevante a um programador da linguagem criada. É possível assim um desenvolvedor ser auxiliado por esta ferramenta antes de realizar compilações que envolvam um maior tempo de compilação.

De salientar, no entanto, que é usual muitas ferramentas fazerem uma análise mais profunda que costuma envolver a verificação de tipos e verificação de acessos de memória indevidos.

Objetivos

O objetivo principal deste trabalho é uma travessia correta e completa da linguagem de programação imperativa (LPI) criada. Uma vez que foram adicionadas algumas funcionalidades extra, em relação aos requisitos pedidos no enunciado, a travessia completa da LPI seria muito exaustiva. Desta forma, o objetivo principal era a travessia da LPI criada que cumprisse os seguintes requisitos: declarar variáveis; manipular valores dos tipos Inteiro, Booleano, Array, Tuplo, String, Lista; escrever instruções tais como Atribuição, Leitura, Escrita, Seleção (SE, CASO), Repetição (ENQ-FAZER, REPETIR-ATE, PARA-interv-FAZER)

Com a travessia garantida os restantes objetivos envolvem a resolução de problemas comuns na criação de uma LPI. Estes problemas envolvem a gestão de escopos de variáveis, identificação de aninhamento de estruturas de controlo e ainda problemas de otimização como a identificação de

estruturas de controlo redundantes.

Para a resolução destes problemas um aspeto muito importante é a minimização de travessias necessárias para resolver os objetivos. No entanto, é necessário manter um código que seja passível de ser mantido no futuro. A complexidade de um código não é exatamente uma métrica precisa, no entanto, pretende-se alertar para o facto de no caso de se duas soluções visitarem o mesmo número de nodos, então é melhor realizar mais uma travessia/análise se permitir um código que seja mais fácil de manter.

Estrutura do Relatório

Os resultados mostrados pelo Analisador Hertz têm como finalidade cumprir uma série de requisitos sobre o código fonte. Estes **requisitos** são inicialmente explicitados neste relatório, por forma a fundamentar a solução concebida. De seguida, é apresentado o **desenho da solução**. Este "desenho"explica uma visão geral do relacionamento dos módulos da aplicação, assim como as estruturas de dados mais relevantes para o problema. Para além disto, são também caracterizados todos os aspectos do resultado produzido pelo Analisador, isto é, como as análises realizadas são expostas num ficheiro HTML.

Algumas **decisões e problemas de implementação** são mencionados após o desenho da solução. Estas decisões possuíram uma enorme influência na codificação do Analisador, envolvendo, nomeadamente, questões relativas a travessia da árvore abstracta do código Hertz. Posteriormente são apresentados alguns **testes robustos** realizados sobre a aplicação, com o objetivo de explorar todos os recursos do Analisador.

Ao longo da construção do Analisador Hertz percebemos a possibilidade de desenvolver paralelamente um *Beautifier* para a linguagem Hertz. Esta *feature* extra é apresentada no capítulo Funcionalidades Extras. Por fim, é feita uma conclusão e análise crítica sobre o trabalho desenvolvido.

Capítulo 2

Análise e Especificação

2.1 Descrição informal do problema

O problema consiste na criação de um relatório que descreva pontos de interesse na LPI Hertz. Estes pontos de interesse devem envolver tarefas que explorem o conhecimento de travessias de árvores abstratas de gramáticas de LPI's, assim como a resolução de problemas encontrados na compilação de LPI's, como escopo de variáveis, aninhamento de estruturas de controlo e identificação de funções reservadas.

2.2 Especificação dos Requisitos

O analisador da linguagem Hertz deve extrair as seguintes características de um dado programa na linguagem Hertz:

1. Contagem do número total de variáveis utilizadas, assim como a listagem de todas as variáveis do programa indicando os casos de:
   i) redeclaração;
   ii) não declaração;
   iii) variáveis usadas mas não inicializadas;
   iv) variáveis declaradas e nunca mencionadas.
2. Contagem do número total de tipos de dados utilizados, assim como a listagem dos tipos de dados;
3. Contagem do número total de instruções que formam o corpo do programa, indicando o número de instruções dos seguintes tipos:
   i) atribuições;
   ii) condicionais;
   iii) cíclicas;

iv) leitura;

v) escrita.

4. Contagem do número total de estruturas de controlo que surgem aninhadas em outras estruturas de controlo do mesmo ou de tipos diferentes.

5. Indicar possíveis substituições de estruturas de controlo condicional aninhadas por uma única estrutura. Para além disto, deve ser também mostrada a nova expressão da instrução condicional que substitua as originais.

Capítulo 3

Desenho da Solução

3.1 Arquitetura do Analisador

O Analisador Hertz possui uma arquitetura modular em três etapas. Inicialmente é construída a árvore abstracta do código fonte através da biblioteca Lark do Python. Esta primeira etapa está concretizada no módulo *Hertz.py*. Quanto à segunda etapa, esta tem como objetivo realizar uma travessia na árvore abstracta e recolher algumas informações das estruturas condicionais existentes no código. Estas informações são armazenadas em *IFStructures* - estrutura auxiliar para armazenamento de instruções condicionais da linguagem Hertz. Esta etapa é concretizada no módulo *AnalisadorIF* que utiliza internamente o *Interpreter* da biblioteca Lark (de modo a ser realizada a tal travessia). O resultado desta etapa é então utilizado na terceira etapa: o Analisador geral.

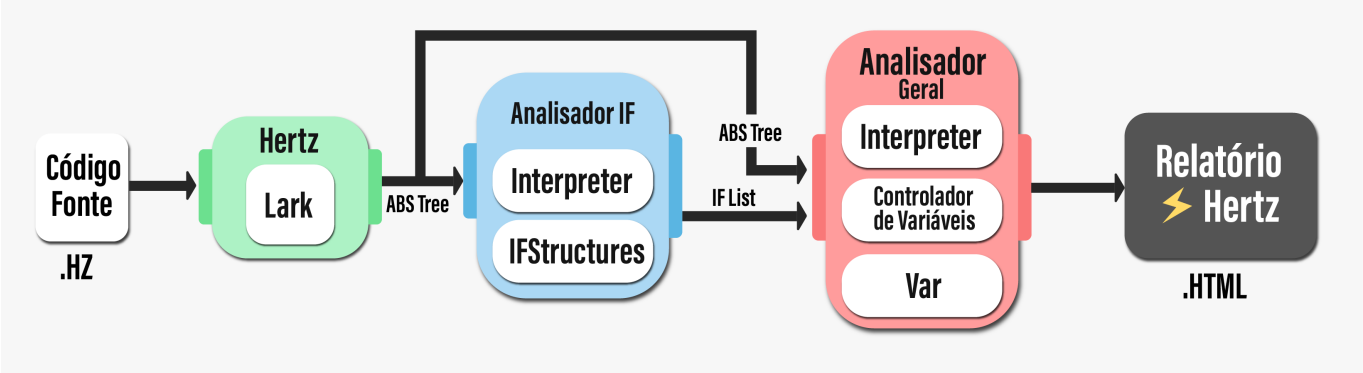

Figura 3.1: Arquitetura da Solução

A terceira etapa é concretizada no módulo *AnalisadorGeral*, cujo propósito é, nomeadamente, gerar o Relatório Hertz - ficheiro em HTML com a análise estática do código Hertz. Esta etapa realiza uma nova travessia na árvore abstracta do código, construindo granularmente o ficheiro HTML. As informações prévias conhecidas relativas às instruções condicionais (graças à etapa anterior) auxilia na criação do Relatório Hertz e, caso não existisse, acumularia (e muito) a complexidade deste módulo. Este módulo utiliza a classe *ControladorDeVariaveis* (estruturas suportes para a travessia) e a classe *Var* (abstração de uma variável da linguagem Hertz).

3.2 Estruturas de Dados

Stack de escopos de variávies

De modo ao Analisador cumprir o requisito 1, uma estrutura de dados adequada devia dar suporte às operações efetuadas aquando da detecção de uma variável na travessia da árvore. Esta estrutura devia ser capaz de cumprir os seguintes requisitos:

1. Seja fácil pesquisar se a variável foi declarada num escopo que permita o seu uso no escopo atual;
2. Seja possível etiquetar as variáveis como (1) já inicializadas e (2) já utilizadas.

A estrutura que melhor se adequava a estes requisitos era uma *stack*. Mas não uma *stack* de variáveis, e sim uma *stack* em que cada elemento seja uma abstração de um **escopo de variáveis**. Um **escopo de variáveis** são todas as variáveis presentes num determinado escopo do código que, a nível da gramática desenvolvida, corresponde a todas as variáveis contidas num **bloco_exp**, ou, no caso excecional, as variáveis que são argumentos numa declaração de função. Esta estrutura foi definida então na classe **ControladorDeVariaveis** como:

```
self.stackDeVariaveis: List[Dict[str,Var]]
```

Definição da *stackDeVariaveis*

Como pode ser visto, as variáveis de um determinado escopo são armazenadas num dicionário (por forma a ser de fácil acesso a verificação de sua existência), enquanto os escopos são armazenados numa lista que se comporta como uma *stack*. Este comportamento se resume às seguintes operações:

```
def initNovoEscopo(self) -> None:
        self.stackDeVariaveis.append({})

def popVariaveisDoEscopo(self) -> None:
    self.stackDeVariaveis.pop()
```

Push e *Pop* da *stackDeVariaveis*

Iniciar um novo escopo corresponde a preparar um dicionário vazio para ser preenchido pelas variáveis que se detectarem no tempo de vida de tal escopo. A utilização destes métodos se encontram no seguinte excerto da travessia da árvore:

```
def bloco_exp(self, production):
    self.controladorDeVariaveis.initNovoEscopo()
    for declaracao in production.children:
        self.visit(declaracao)
    self.controladorDeVariaveis.popVariaveisDoEscopo()
```

Regra de produção *bloco_exp* no interpretador

Como uma **declaração** dentro de um escopo pode por sua vez conter escopos aninhados, justifica-se então a tal *stack* utilizada: os escopos ancestrais continuam a existir e, para além disto, quando um escopo acaba, as suas variáveis são removidas da *stack*. Assim, o teste da existência e usabilidade

de uma variável aquando da sua detecção na travessia, corresponde a verificar a sua existência na *stackDeVariaveis* (uma vez que ela contém somente as variáveis utilizáveis até o escopo atual).

Alguns detalhes mais profundos relativos ao momento da inserção de uma variável na *stack* serão discutidos no capítulo a seguir, mas de forma resumida, quando uma variável é declarada pela primeira vez num determinado escopo do código e não foi declarada ainda nos escopos ancestrais, é então utilizado o seguinte método:

```
def pushVariavelDoEscopo(self,variavel:Var):
        # ... Caso a variável ainda não foi declarada nos escopos ancestrais
        self.stackDeVariaveis[-1][variavel._id] = variavel
```

Função *pushVariavelDoEscopo*

Como o último elemento da *stack* corresponde ao escopo atual em análise na travessia, inserir uma variável na *stack* corresponde a mapeá-la no dicionário presente no último elemento da *stack*. A verificação de se a variável já foi declarada (nalgum escopo que permita a sua utilização no escopo atual) é feita do seguinte modo:

```
def variavelNaoDeclarada(self,id) -> bool:
    for variaveisDoEscopo in self.stackDeVariaveis:
        if id in variaveisDoEscopo:
            return False
    return True
```

Função *variavelNaoDeclarada*

Pode-se observar que, para escopos não relacionados no código (por exemplo funções ao mesmo nível de aninhamento) é possível haver variáveis identificadas com o mesmo nome, uma vez que a *stack* apenas armazena as variáveis utilizáveis até o escopo atual em análise durante a travessia na árvore. O seguinte código ilustra como a *stackDeVariaveis* evolui ao longo da travessia:

```
integer  main(integer p){

    integer  i;
    integer  k;
    loop (i = 1; i < 6; i = i + 1){
        integer f;
        if (i % 2 == 0){
            integer  m = 0;
        } else {
            integer  h = 0;
        }
        integer  m;
    }
    integer  l;
    return 0;
}
```

Código exemplo

```
initNovoEscopo() // escopo dos argumentos
stackDeVariaveis = [{p}]
initNovoEscopo() // escopo do corpo da função
stackDeVariaveis = [{p},{i}]
stackDeVariaveis = [{p},{i,k}]
initNovoEscopo() // escopo do loop
stackDeVariaveis = [{p},{i,k},{f}]
initNovoEscopo() // escopo do if
stackDeVariaveis = [{p},{i,k},{f},{m}]
popVariaveisDoEscopo()
stackDeVariavies = [{p},{i,k},{f},{h}]
popVariaveisDoEscopo()
stackDeVariaveis = [{p},{i,k},{f,m}]
popVariaveisDoEscopo()
stackDeVariaveis = [{p},{i,k,l}]
popVariaveisDoEscopo()
stackDeVariaveis = [{p}]
popVariaveisDoEscopo()
```

Evolução da *stack* de variáveis

Representação de uma variável

A representação feita apenas abrange os problemas encontrados. Não foi adicionada informação relativa aos tipos da variável, por exemplo. Uma variável contém:

- uma *String* que identifica a variável dentro do escopo em que foi declarada (*id*);
- um *Bool* responsável por indicar se a variável já foi inicializada (*inicializada*);
- um *Bool* responsável por indicar se a variável já foi usada no seu escopo (*usada*);
- um inteiro que indica a linha em que a variável foi declarada (*linhaDeclaracao*).

Esta última instância é responsável por distinguir variáveis que têm o mesmo *id* e em que o escopo de uma das variáveis é subconjunto do escopo da outra variável, mas cujo o escopo de declaração é diferente. Por exemplo, se o argumento de uma função tiver o *id* argc deveria ser possível declarar uma variável com o mesmo *id* dentro do corpo da função. Nesse caso esta última variável iria fazer *shadow* ao argumento da função. Desta forma, para distinguir, é possível comparar a linha de declaração das duas variáveis.
Em suma, esta representação da variável é concretizada na classe *Var*, e são objetos deste tipo que são mapeados nas variáveis do escopo mencionadas anteriormente. Com isto, alcança-se o segundo *design goal* mencionado em 2.

Resultados e estatísticas gerais

Listas e conjuntos foram guardados de forma a que, no final da travessia, seja possível disponibilizar estáticas ao utilizador, estatísticas essas como:

- Total de variáveis;
- Tipos de dados usados;
- Tipos de instruções usadas;
- Total de variáveis não usadas;
- Total de aninhamentos de controlo;

Variáveis para determinar aninhamentos e recursividade

Durante as travessias foram utilizadas variáveis cujo seu propósito era indicar quantas visitas recursivas estavam a ocorrer num nodo. Estas variáveis auxiliavam a gestão de variáveis de estado que tinham que ser reinicializadas quando não se visitava um nodo de forma recursiva. Foram também utilizadas para determinar o número de aninhamentos de ciclos e detetar se um ciclo era um potenciador de gasto de energia. Por exemplo, se um ciclo tivesse aninhado noutros dois ciclos então havia a possibilidade da complexidade energética do código ser elevada. Para além de auxiliar na gestão de estado e de determinar o aninhamento atual de estruturas de controlo, foi imperativo para visitar a estrutura de **SE**'s, criada numa travessia anterior.

Lista de estruturas de controlo SE

Na segunda travessia é guardada uma lista com os **SE**'s que não estão aninhados noutro **SE**.

Estrutura de controlo SE

Cada estrutura de controlo **SE** contém:

- um *Bool* que indica se o bloco de expressão do **SE** contém apenas um item, sendo que esse item é também uma estrutura **SE**;
- uma lista de **SE**'s que estão aninhados nesta estrutura de controlo;
- uma expressão que é a condição do **SE**;
- uma expressão opcional, utilizada quando é possível juntar **SE**'s, que representa a condição que é recomendada para juntar os **SE**'s.

3.3 Representação em HTML dos Resultados da Análise

A execução do Analisador Hertz sobre um código em Hertz gera um relatório Hertz, através de um ficheiro HTML. O desenho da solução foi concebido com base na busca por mostrar os resultados da análise de maneira **(1)** fácil e agradável de se ler, **(2)** intuitiva para navegação e **(3)** dispondo os elementos essenciais para o utilizador.
A estrutura deste documento baseia-se em três secções:

1. Secção de estatísticas gerais sobre o código;
2. Secção de visualização "aumentada"do código;
3. Secção de estatísticas sobre estruturas de controlo.

A seguir serão caracterizadas com mais pormenor cada uma destas partes do ficheiro HTML.

Secção de estatísticas gerais sobre o código

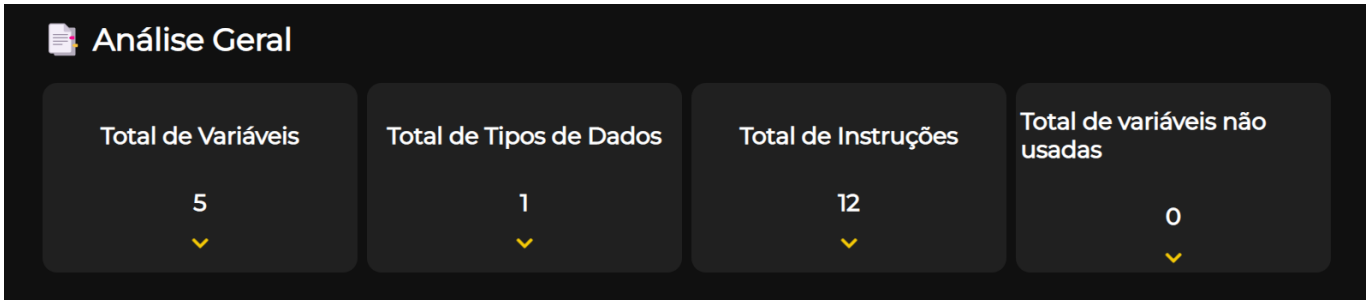

Figura 3.2: Secção 1 (não expandida) do ficheiro HTML

A figura 3.3 mostra um *draft* da secção inicial (não expandida) do ficheiro HTML. Esta secção é composta de quatro divisões. Cada divisão possui o número total de alguma estatística (no caso da primeira divisão essa estatística é o número total de variáveis) e, ao ser pressionada a seta amarela, a secção é expandida e são listados os itens individuais da estatística correspondente (no caso da primeira divisão são listadas as variáveis).

1. **Total de variáveis:** número total de variáveis declaradas no programa. A listagem das variáveis indica o seu nome e a linha em que foi declarada (que se justificará nos próximos capítulos);
2. **Total de Tipos de dados:** número total de tipos de dados distintos utilizados no programa;
3. **Total de instruções**: número total de instruções utilizadas no programa. As instruções consideradas são:
    - Atribuições;
    - Condicionais;

- Cíclicas;
- Leitura;
- Escrita.

Ao ser expandida esta divisão, são identificadas individualmente as quantidades de cada tipo de instrução presente.

4. **Total de variáveis não usadas**: número total de variáveis declaradas mas não utilizadas noutro momento sequer o da declaração. A sua listagem é feita de maneira semelhante à divisão 1.

A imagem 3.3 mostra um *draft* daquilo que seria o resultado de expandir todas as divisões desta secção do ficheiro.

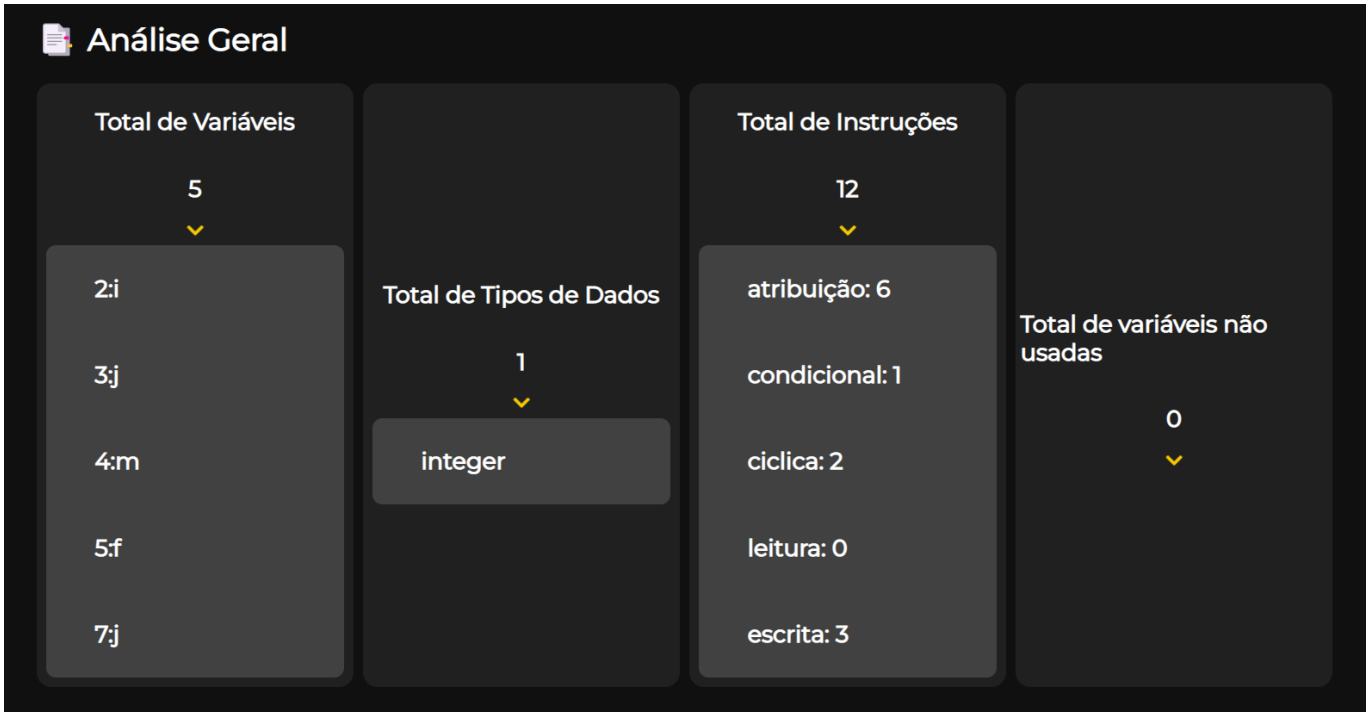

Figura 3.3: Secção 1 (expandida) do ficheiro HTML

Secção de visualização "aumentada"do código

A parte intermédia do ficheiro HTML mostra o código em análise com diversos recursos de estilização. É utilizada uma coloração individualizada para diferentes componentes do código (tipos de dados, nome de funções, *keywords* de instruções, entre outras), como também são identificadas as linhas do código e é realizada uma indentação própria do código. Todas as *features* foram concretizadas de modo a cumprir o *design goal* 3.3 - ser um relatório de leitura fácil e agradável.

Dentre os aspectos de estilização, aquele que mais se destaca são as informações extras dispostas sobre o código. Isto é, os requisitos de identificação de se uma variável foi redeclarada, não inicializada ou não declarada, são feitos dentro do próprio local do código que a identificação ocorre! Daí a designação de "visualização aumentada"do código. É utilizada coloração e animações para distinguir as variáveis que são identificadas segundo os critérios citados. Para além disto, ao utilizador passar o *mouse* sobre estas variáveis, surge na tela uma mensagem indicativa sobre aquilo que foi identificado sobre tal variável.

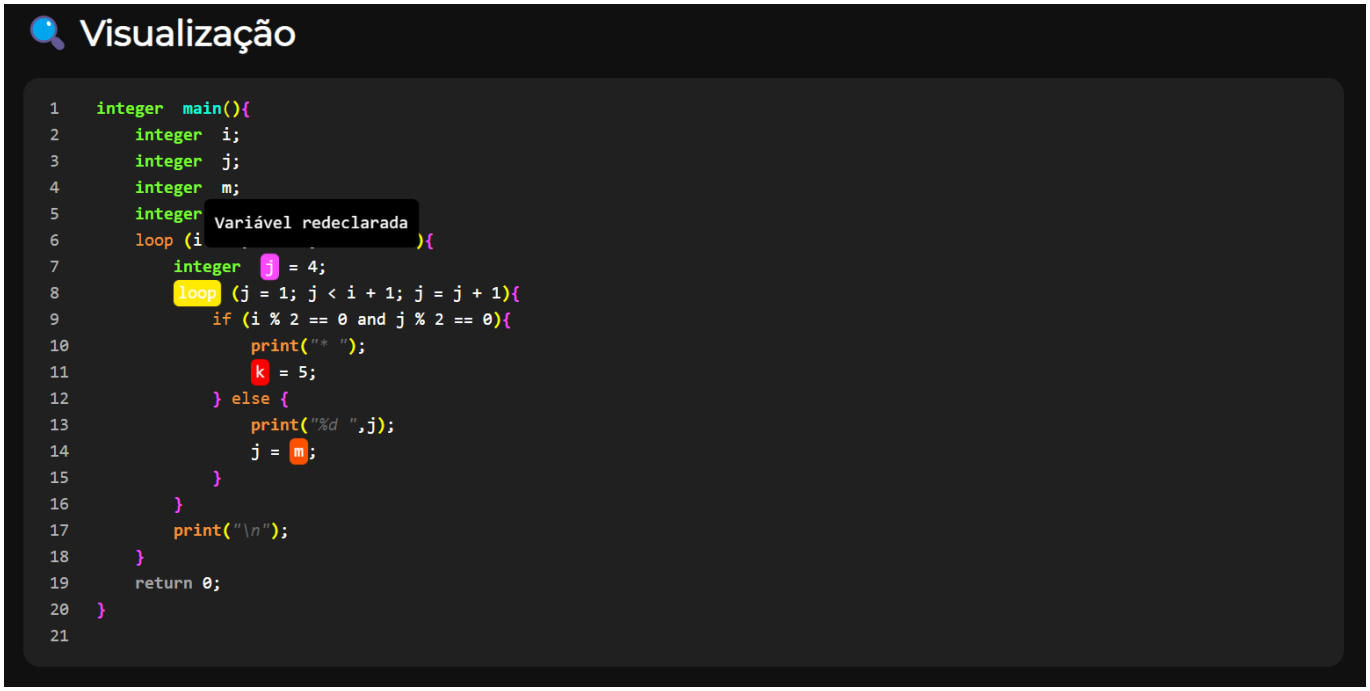

Figura 3.4: Secção 2 do ficheiro HTML

A figura 3.4 mostra um *draft* de um possível código Hertz em visualização aumentada, onde o utilizador está com o *mouse* sobre a variável **j** (e por isto existe a mensagem sobre ela). A coloração utilizada distingue diferentes aspectos identificados, não se reduzindo apenas em identificações sobre variáveis:

- <variavel> identifica uma variável redeclarada;
- <variavel> identifica uma variável como não inicializada;
- <variavel> identifica uma variável não declarada;
- <Instrucao> identifica uma instrução potencial de grande uso energético (comumente encontradas em *loops* aninhados);
- if identifica um "*if*"que pode ter a condição substituída por conta de "*ifs*"aninhados.

Secção de estatísticas sobre estruturas de controlo

Figura 3.5: Secção 3 do ficheiro HTML

A terceira (e última) secção do ficheiro HTML produzido é destinada a estatística geral de quantas estruturas de controlo aninhadas existem no código. Como trabalho futuro, desejamos adicionar mais informações nesta secção, como por exemplo uma visualização do código já com a simplificação dos "*ifs*"identificados com condições potenciais de serem substituídas.

Por fim, a imagem 3.3 revela aquilo que o utilizador vê quando abre a página HTML do relatório Hertz produzido sobre o seu código:

Figura 3.6: Ficheiro HTML produzido pela Análise

Capítulo 4

Codificação e Testes

4.1 Alternativas, Decisões e Problemas de Implementação

Construção do ficheiro HTML

Como foi visto na secção 3.2, o ficheiro HTML gerado pelo analisador possui três secções. Como a segunda secção corresponde a visualização de dados encima do próprio código, era imperativa a construção desta parte do ficheiro HTML ao longo da travessia da árvore. Por outro lado, sendo a primeira e terceira secção destinadas à informações estatísticas globais do código (como por exemplo o número total de variáveis utilizadas), o excerto HTML destas secções deveria ser formado apenas no fim da travessia da árvore. Deste modo, a primeira decisão tomada foi esta separação da construção do ficheiro HTML em três partes, sendo realizada a concatenação das secções no fim da travessia.

```
def start(self,bloco):
    self.visit(bloco.children[0])
    estatisticasStr = self.estatisticasHTML()
    self.pageHTML += estatisticasStr
    self.pageHTML += self.codeHTML
    estatisticasStr = self.estatisticasAninhamento()
    self.pageHTML += estatisticasStr
```

Excerto da regra de produção *start* do interpretador

No excerto 4.1 é mostrada a concatenação das três secções do ficheiro HTML, sendo a primeira secção construída ao fim da travessia pela função *estatisticasHTML()*, a segunda secção contida em *codeHTML* previamente formada ao longo da travessia e por fim a terceira secção construída através da função *estatisticasAninhamento()*. Para além desta decisão de estruturação, um dos problemas de implementação mais relevantes envolvia a construção da segunda secção do ficheiro HTML - a visualização do código. Este problema pode ser resumido a seguinte pergunta: **"Em qual momento da travessia deve ser preenchida a string do código em HTML?"**

Tomemos como exemplo a escrita das variáveis. Esta escrita depende de uma série de factores, como (1) se a variável foi detectada aquando de uma declaração, (2) se a variável foi detectada aquando de uma atribuição, (3) se a variável foi detectada como parâmetro de uma função, entre outros. Cada um desses factores pode influenciar na maneira como uma variável é mostrada no ficheiro HTML.

Por exemplo, se uma variável foi detectada no contexto de uma declaração, pode acontecer dela estar sendo redeclarada e então ocasionar numa estilização diferenciada. A solução mais intuitiva seria preencher o HTML ao nível das regras de produção que dessem um "contexto"às variáveis. Tomando como exemplo novamente uma variável num contexto de declaração, seria agradável escrever tal variável aquando da regra de produção *dec_var*, uma vez que se teria o controlo sobre o contexto em que a variável foi detectada. Porém, não foi seguido este caminho.

A **primeira razão** pela estratégia mencionada ser falha envolve a ausência de filosofia na codificação - o código ficaria desorganizado, sendo difícil gerir a travessia na árvore face aos aumentos dos recursos da linguagem. A **segunda razão** envolve o aumento sem necessidade da complexidade do código, que pode ser ilustrado no cenário de declaração de uma variável: caso a escrita fosse realizada na regra *dec_var*, teria de ser possível obter a variável para depois escrevê-la. À primeira vista isto parece não ser um problema, até lembrarmos das variáveis complexas, como é o caso da *var_subscrito* (que além de um *id*, pode possuir expressões complexas). Deste modo, algumas perguntas emergiam, como: "Será necessária alguma estrutura para armazenar variáveis complexas e depois passá-las para *string*?" "A visita na regra *var_subscrito* deve retornar apenas o ID da variável?". A resposta para estas perguntas envolvia adotar uma abordagem altamente granular, onde a escrita de qualquer coisa na *string* HTML deveria procurar ser o mais perto possível do nível dos *tokens* da linguagem.

Todavia, existe um *trade-off* nesta escolha. Quanto mais granular for a escrita do HTML, mais se perde o "contexto"de escrita. Neste cenário, a escrita duma variável seria na própria regra de produção da variável, regra a qual pode ser utilizada tanto em declarações, atribuições, entre outros contextos. Deste modo, surgiu a necessidade da criação de algumas instâncias de estado no analisador, como é o caso das variáveis *isDeclaracao*, *isAtriv* e *varInExp* que indicam o tal contexto em que a variável é analisada na travessia. Assim, consegue-se ter o controlo sobre as operações a serem realizadas, como pode ser visto no excerto a seguir que mostra alguns contextos em que uma variável é encontrada:

```
def var_unica(self, production):
    self.varID = production.children[0].value
    if self.isDeclaracao:
        self.controladorDeVariaveis.pushVariavelDoEscopo(
            variavel=Var(self.varID,self.varInicializada, self.lineNo)
        )
    if self.isAtrib:
        self.controladorDeVariaveis.initVariavel(self.varID)
    if self.varInExp:
        self.controladorDeVariaveis.usarVariavel(self.varID)
    self.writeVar()
```

Excerto da regra de produção *var_unica* do interpretador

Com esta abordagem, encontramos uma filosofia de preenchimento do ficheiro HTML de maneira granular. O que essa filosofia diz é que, enquanto existir itens não terminais nas regras de produções, deve-se abstrair a preocupação de escrita destes itens nas visitas destas regras. Isto garante que, desde que a visita seja ordenada de acordo com o código fornecido, o ficheiro HTML será construído corretamente e, para além disto, será possível ter um controlo de estilização da linguagem ao nível dos *tokens* (garantindo assim o aspecto colorizado das *keywords* da linguagem).

Formatação e linhas do código

Como foi visto na secção 1, não só o número total de variáveis utilizadas seria calculado, como também a listagem de todas as variáveis do programa era um requisito. E isto poderia ocasionar num problema de identificação das variáveis, uma vez que é possível existir variáveis com o mesmo nome em escopos diferentes.

Por forma a solucionar este problema, optamos por diferenciar as variáveis através da sua linha de declaração. Deste modo, justifica-se a necessidade do número da linha da variável mencionado em 3.2, como também do número das linhas presente na representação em HTML do código 3.2. Por esta mesma razão existe a instância *lineNo* no Analisador, destinada a manter o estado atual relativo a linha atual em análise do código. O detalhe especial aqui é que a tal "linha atual"é baseada na **formatação induzida por nós** face ao código sujeito a análise. Por outras palavras, realizamos uma **formatação** do código, para não só melhorá-lo esteticamente, como também contribuir para a diferenciabilidade de identificação das variáveis (caso contrário, imagine o código todo numa linha, tanto a identificação das variáveis como o relatório Hertz seriam comprometidos).

Travessia adicional

Como mencionado na secção 4.1, o ficheiro HTML é criado à medida que se percorre a árvore. Para resolver o problema de descobrir se um **SE** podia ser simplificado era necessário visitar primeiro as estruturas **SE**. Estas visitas não poderiam escrever no ficheiro HTML, senão a representação HTML estaria errada. Se fossem feitas na mesma travessia que a construção do HTML a complexidade do código para realizar esta travessia iria ser insuportável e contraprodutiva.

A nossa solução foi a criação de uma estrutura intermédia que contém a informação necessária para que quando fosse a feita a construção do HTML fosse possível recomendar se **SE**'s podiam ser juntados. A criação desta estrutura intermédia foi feita numa travessia simples que apenas visita nodos em que há a possibilidade de aparecerem **SE**'s. Por exemplo, não é necessário percorrer uma regra de produção *expr* porque se sabe que não se vai encontrar um **SE** nessa visita. No entanto, numa regra de produção *dec_loop*, no corpo do ciclo é possível encontrar um **SE**.

4.2 Testes realizados e Resultados

Teste Geral

O primeiro teste realizado teve como propósito abrangir todas as possíveis 5 identificações sobre o código mencionadas em 3.3. Para além disto, procuramos testar estas identificações em **escopos aninhados**, por forma a verificar as funcionalidades da *stackDeVariaveis*. A seguir serão comparados lado a lado o código fonte Hertz com o resultado da "visualização aumentada"do relatório Hertz.

```
integer main() {
   integer i;
   integer j;
   integer i = 9 + j;
   loop(i=1; i<6; i=i+1) {
        integer c = i + b + j;
        integer j;
        bool v = true;
        if(i % 2 == 0 and j < 3){
            string nome = "Levi";
            string sobrenome = "Ackerman";
            do {
                integer m = 6;
                j = 9;
                string sobrenome = "Jaegar";
                integer a;
                printf("Hello World");
            } again if (j > 5)
        }else{
            integer f;
            string nome = "Eren";
            a = 8 + f + i;
            if(i > 10){
                if(a < 15){
                    loop while (v and i < f){
                        integer f = 5;
                    }
                }
            }
        }
    }
    integer f = 5;
    return 0;
}
```

Código Teste Geral

```
1  integer main(){
2      integer i;
3      integer j;
4      integer i = 9 + j;
5      loop (i = 1; i < 6; i = i + 1){
6          integer c = i + b + j;
7          integer j;
8          bool v = true;
9          if (i % 2 == 0 and j < 3){
10             string nome = "Levi";
11             string sobrenome = "Ackerman";
12             do {
13                 integer m = 6;
14                 j = 9;
15                 string sobrenome = "Jaegar";
16                 integer a;
17                 printf("Hello World");
18             } again if (j > 5)
19         } else {
20             integer f;
21             string nome = "Eren";
22             a = 8 + f + i;
23             if i > 10){
24                 if (a < 15){
25                     loop while(v and i < f){
26                         integer f = 5;
27                     }
28                 }
29             }
30         }
31     }
32     integer f = 5;
33     return 0;
34 }
```

Figura 4.1: Resultado Teste Geral

O primeiro notável resultado é a detecção da variável i como redeclarada na linha 4. Isto faz sentido, uma vez que esta mesma variável havia sido declarada na linha 2 do mesmo escopo. Para além disto, a variável j foi detectada como não inicializada (aquando da sua utilização também na linha 4). Apenas com estes resultados, conseguimos explorar duas detecções sobre variáveis, num contexto de utilização de uma expressão complexa.

A segunda observação a ser feita é sobre a detecção da variável *b* na linha 6 como não declarada. Adicionalmente foi utilizada a variável *i* nessa mesma linha, por forma a verificar que nada é detectado sobre ela. Isto porque, apesar de estar num escopo diferente de quando foi declarada, tal escopo é **alcançável** a partir do escopo anterior. Ou seja, o escopo em que a variável *i* foi declarada é um escopo ancestral do escopo atual, o que permite a sua utilização. Esta variável *i* também não é detectada como não inicializada, uma vez que ocorre a sua inicialização no corpo do *loop*. Por fim, novamente foi utilizada a variável *j*, a fim de verificar que ela ainda é detectada como não inicializada.

A coloração a rosa na linha 7 para a variável *j* justifica-se dado que ela já foi declarada num escopo ancestral. A sua coloração a laranja na linha 9 ocorre novamente por ela ainda não ter sido inicializada. Este teste foi realizado para explorar a identificação destes aspectos da variável no contexto de uma expressão condicional complexa.

A próxima reflexão relevante é sobre a coloração amarela sobre a declaração dos ciclos *do-again-if* (linha 12) e *loop while* (linha 25). Como estes ciclos encontram-se aninhados sobre o ciclo *loop* (linha 5), estas instruções acabam por ser detectadas como **potenciais de grande gasto energético**. Conseguimos assim explorar mais um aspecto de identificação sobre o código, testando a utilização de três formas distintas de concretizar ciclos na linguagem Hertz.

Da mesma maneira que a variável *j* é identificada como redeclarada na linha 7, a variável *sobrenome* também é identificada como tal na linha 15. Isto acontece devido à sua declaração no escopo pai (escopo ancestral de grau 1) na linha 11. Todavia, reparemos no que se sucede quanto à variável *nome* declarada na linha 10. Esta variável é declarada novamente na linha 21, porém num escopo que não se relaciona hereditariamente com o escopo da linha 10. Por conta disto, essa variável não é identificada como redeclarada, sendo consideradas distintas as variáveis *nome* da linha 10 e da linha 21. Isto verifica o correto funcionamento da *stackDeVariaveis*.

Um comportamento semelhante ao anterior acontece com a variável *a* identificada como **não declarada** na linha 22. O escopo da linha 22 não é alcançável a partir do escopo da linha 16. E por isso, apesar de declarada na linha 16, tal variável não é considerada como declarada na linha 22. Para colocar ainda mais complexidade no teste, é verificada a não inicialização da variável *f* utilizada nessa mesma linha 16.

A coloração roxa no *if* da linha 23 identifica uma condição que pode ser simplificada devido a *ifs* aninhados. Ao passar o *mouse* por cima desta instrução, o utilizador pode ver a expressão recomendada para simplificar o código.

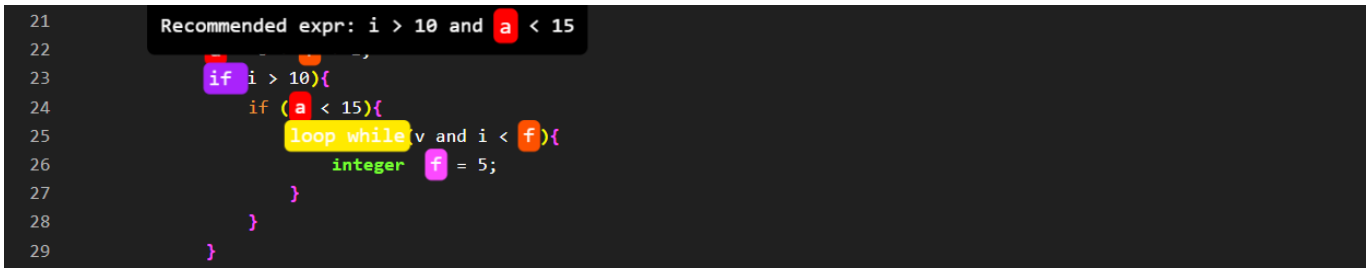

Figura 4.2: *if* detectado com condição passível de ser simplificada

Finalmente, o último resultado notável neste teste é a detecção de redeclaração da variável *f* na linha 26, e a não detecção de nada da variável *f* declarada na linha 20 e 32. O problema de tal variável declarada na linha 26 é o seguinte: o escopo em que ela se insere é um escopo filho do escopo em

que a variável *f* também é declarada na linha 20. E por isto, a sua declaração na linha 26 é na verdade considerada como uma "redeclaração". Em contrapartida, a variável *f* declarada na linha 20 não é considerada como redeclarada. Apesar dela estar num escopo filho do escopo da variável *f* declarada na linha 32, pelo simples facto da declaração da linha 32 ocorrer num momento **posterior** ao tempo de vida do escopo da variável *f* da linha 20, não há problemas. Caso a declaração da linha 32 ocorre-se, por exemplo, na linha 3, então aí sim tanto a variável *f* da linha 20 e da linha 26 seriam consideradas redeclaradas. Este notável exemplo mostra que a utilização de uma *stack* de escopos é ideal para esta tarefa, conseguindo tal estrutura abstrair aquilo que é o **tempo de vida** de uma variável.

A imagem 4.2 mostra o que o utilizador vê quando abre a página HTML gerada pelo Analisador. As estatísticas gerais sobre o código mostradas na secção 1 reúnem os resultados esperados, mostrando a variedade de variáveis e instruções utilizadas no teste. Com isto, foi explorado diversos recursos e aspectos da linguagem Hertz.

Figura 4.3: Teste Geral - Visão Geral ficheiro HTML

A figura seguinte revela a primeira secção do ficheiro HTML de maneira expandida. Merece comentário o facto da variável *j* aparecer duas vezes na listagem do total de variáveis, enquanto a variável *i* aparecer somente uma vez. Como a variável *i* foi redeclarada no mesmo escopo, ela não foi contabilizada como "nova variável". Em contrapartida, a redeclaração de *j* é contabilizada como nova variável por ocorrer **num escopo aninhado**. Este conceito é chamado de *variable shadowing*. No que respeita as instruções de **atribuição**, foi contabilizada também as atribuições nas expressões das instruções cíclicas. Por fim, outro detalhe relevante, é que atribuições de variáveis no momento de sua declaração não foram contabilizadas para a marcação destas variáveis como "utilizadas". Por este motivo, a variável *nome*, por exemplo, é listada como **não usada**.

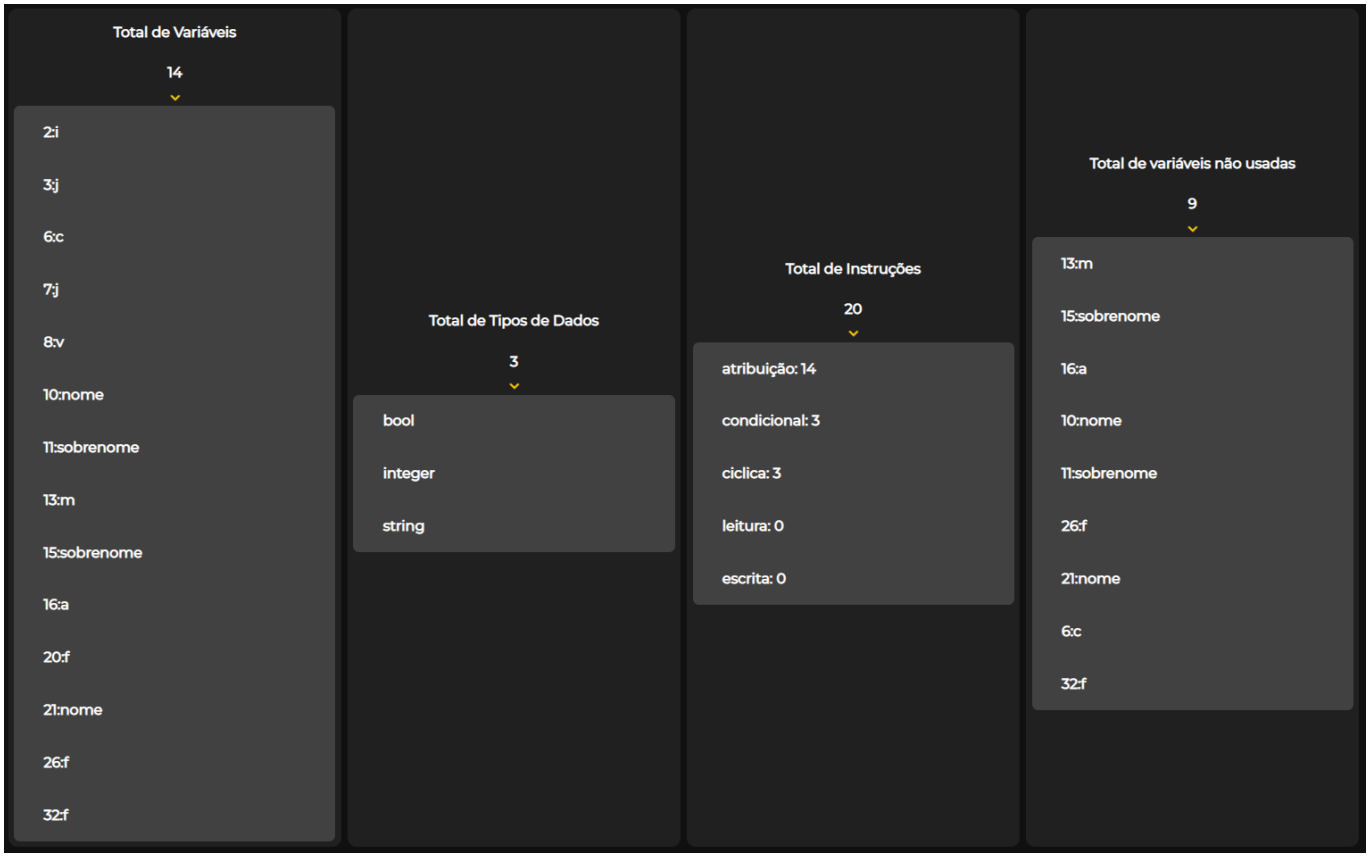

Figura 4.4: Teste Geral - Secção 1 expandida

Como existe uma estrutura cíclica *do-again-if* (linha 12) aninhada no bloco *if* (linha 9), que por sua vez está aninhado numa instrução cíclica *loop* (linha 5), é contabilizado um aninhamento de **3 níveis** para essas estruturas de controlo. Porém, dentro do bloco *else* (da mesma instrução condicional citada) existe um aninhamento de mais duas instruções condicionais (linha 23 e 24) e, no nível mais profundo, uma instrução cíclica *loop while* (linha 25). Logo, existem mais 3 estruturas de controlo aninhadas, totalizando 6 no geral.

Figura 4.5: Teste geral - Secção 3

Teste de *Closure* de Função

O segundo teste proposto tem como objetivo explorar o conceito de *closure* de funções: acesso ao escopo de uma função externa a partir de uma função interna. Deste modo, focamo-nos em definir funções dentro de outras funções e identificar as variáveis que usufruam do *closure* das funções.

```
integer func2(integer x, integer y){
    integer f = 5;
    integer g;
    integer func3(){
        b = 5 + g;
        f = 8;
        integer func4(){
            f = 9;
            integer g = 5;
            integer n = 2;
            integer h;
            return f + 5;
        }
        return x+x+n + f+func4();
    }
    n = 8;
    integer h = 9;
    integer a = func3();
    return a + y;
}

integer main() {
    integer b = 2;
    integer d = func2(x,b);
    print("Result %d",d);
    return 0;
}
```

Código Teste de *Closure* de Função

```
1   integer  func2(integer  x,integer  y){
2       integer  f = 5;
3       integer  g;
4       integer  func3(){
5           b = 5 + g;
6           f = 8;
7           integer  func4(){
8               f = 9;
9               integer  g = 5;
10              integer  n = 2;
11              integer  h;
12              return f + 5;
13          }
14
15          return x + x + n + f + func4();
16      }
17
18      n = 8;
19      integer  h = 9;
20      integer  a = func3();
21      return a + y;
22  }
23  integer  main(){
24      integer  b = 2;
25      integer  d = func2(x,b);
26      print("Result %d",d);
27      return 0;
28  }
```

Figura 4.6: Resultado Teste *Closure*

A variável *b* na linha 5 foi identificada corretamente como **não declarada**, enquanto a variável *g* como **não inicializada**. E aqui vemos a ideia de *closure* em ação: apesar de *g* não ter sido declarada na função *func3*, tal função é definida dentro da função *func2*. Logo, a função *func3* pode referenciar variáveis da função *func2*. Consequentemente, a variável *g* na linha 5 não é identificada como **não declarada** e sim somente como **não inicializada**. Complementarmente, não é detectado nada em relação à variável *f* na linha 6, que foi corretamente declarada na linha 2 da função exterior.

A função *func4* está definida dentro da função *func3* e, pelos motivos explicados no parágrafo anterior, tal função pode referenciar variáveis locais da função *func3*. E, como a função *func3* pode referenciar variáveis locais da função *func2*, por transitividade a função *func4* pode referenciar variáveis locais da função *func2*. Deste modo, novamente a variável *f* pode ser utilizada, como acontece na linha 8. E, pela variável *g* ter sido declarada na função *func2*, a sua declaração na linha 9 é na verdade considerada uma **redeclaração**.

É interessante de se analisar a detecção da variável *n* na linha 15 e na linha 18 como **não declarada**. Ora, apesar da variável *n* ter sido declarada dentro da função *func4* na linha 10, o conceito de *closure*

não funciona no sentindo inverso: funções externas não tem acesso as variáveis locais definidas em funções internas. Desse modo, a variável *n* só tem existência no contexto da função *func4*. Por outro lado, a variável *x* presente na linha 15 é um parâmetro da função *func2* e por isso não há nenhum problema relacionada a ela. O mesmo se sucede com a variável *f* dessa linha.

Por fim, reparemos na declaração da variável *h* na linha 19. Nada é identificado sobre ela, mesmo já havendo a declaração de uma variável *h* na linha 11. E para além disto, nada é detectado também sobre esta tal variável *h* da linha 11. Isto acontece porque, no momento da declaração de *h* na linha 11, não existia ainda uma variável *h* na função *func2*. E, por conta da natureza sequencial do código, ambas as variáveis não se interferem existencialmente. Uma situação similar ocorreu no Teste Geral com a variável *f*.

O mais interessante disto tudo, é que a implementação do conceito de *closure* das funções se encaixa perfeitamente com a *stack* de escopos de variáveis previamente explicada. Seja a definição de uma função, seja o bloco de um *loop* ou o bloco de uma instrução condicional, a gestão dos escopos é a mesma.

Teste da simplificação das condições de instruções condicionais

```
integer main() {
   integer i = 5;
   bool check = true;
   integer j = 9;

   if(i < 10){
        if(check == true){
            if(i == 9){
                print("Cheguei!\n");
            }
        }
   }
   return 0;
}
```

Código Teste de Condicionais

```
1   integer  main(){
2       integer  i = 5;
3       bool  check = true;
4       integer  j = 9;
5       if i < 10){
6           if check == true){
7               if (i == 9){
8                   print("Cheguei!\n");
9               }
10          }
11      }
12      return 0;
13  }
```

Figura 4.7: Resultado Teste Condicionais

Apesar de já se ter visto a identificação de expressões condicionais a serem simplificadas no Teste Geral, este teste aprofunda tal *feature*. O resultado da análise em 4.2 mostra a identificação de duas instruções condicionais que podem ser simplificadas (identificação através da coloração roxa nas linhas 5 e 6). Ao passar o *mouse* por cima da instrução da linha 6, o utilizador vê a seguinte mensagem:

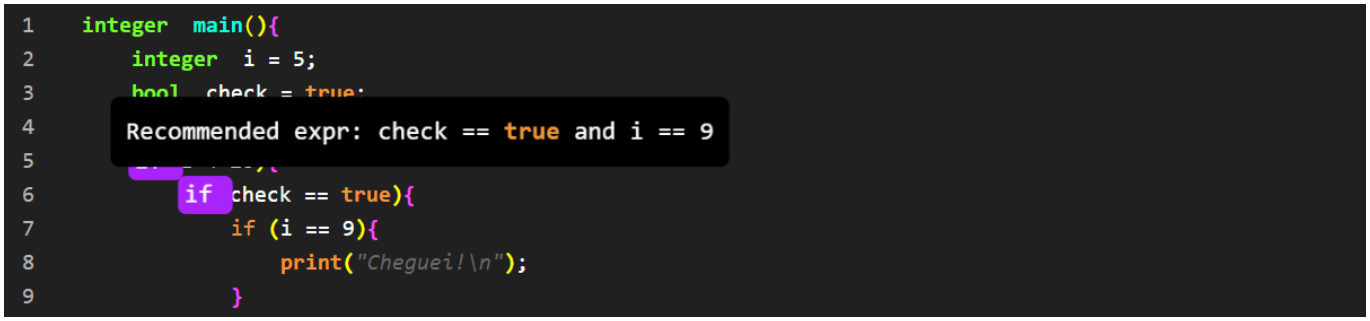

Figura 4.8: Resultado Teste Condicionais (2)

O que essa mensagem significa é que, face à instrução condicional aninhada na linha 7, a condição da instrução condicional da linha 6 pode ser substituída para a condição que aparece na mensagem. E, desta maneira, a instrução condicional da linha 7 torna-se desnecessária. Por outro lado, a própria instrução condicional da linha 6 também é desnecessária, uma vez que essa instrução também está aninhada com uma instrução condicional na linha 5. Logo, o utilizador vê a seguinte mensagem se passa o *mouse* por cima da instrução da linha 5:

Figura 4.9: Resultado Teste Condicionais (3)

A mensagem mostrada na figura 4.2 exprime uma condição que reúne as condições das instruções condicionais das linha 6 e 7. A utilização desta condição recomendada torna desnecessária a utilização das instruções condicionais das linha 6 e 7.

Capítulo 5

Funcionalidades extras

O processo de construção da **visualização aumentada** do código em HTML exigiu um importante detalhe: a indentação e formatação do código fonte. A utilização de uma abstração de uma *stack* tornou isto possível. Desta vez, a *stack* seria composta de caracteres de **espaço em branco**. As operações de *push* e *pop* correspondiam à inserção/remoção destes caracteres de espaço.

Em posse da *string* representativa do código indentado, bastava utilizar a *tag* <***pre***> em HTML para inserir tal *string*. Esta *tag* define um bloco de texto pré-formatado, ou seja, um texto em que a formatação é mantida exatamente como foi escrita no código fonte. Percebemos então que poderíamos utilizar esta *string* do código indentado noutros contextos, sem restringir apenas para construção do ficheiro HTML. Caso esta *string* fosse isolada do Analisador, sem ser preenchida com as *tags* HTML, poderíamos utilizá-la como resultado de um *Beautifier*. Isto é, para além do Analisador Hertz, facilmente criamos um programa extra, cujo propósito é apenas formatar e indentar o código fonte Hertz.

Em suma, o módulo Beautifier concretiza esta nova funcionalidade. Onde, dado um código Hertz, é gerado um novo código fonte Hertz, mas com a formatação induzida por nós. A seguir são dados três exemplos de utilização deste módulo:

```
integer main(
) {
   integer i;integer j;loop(i=1;
i<6; i=i+1) {j = 0;
   do
{j = i + 1;
         if (i % 2 == 0 and j % 2 == 0) {
      print("* ");
   } else {
 print("%d ", j);}} again if(

   j
   <
   5)
print("\n");
   }return 0;
}
```

Código Hertz mal formatado (1)

```
integer main(){
    integer i;
    integer j;
    loop(i = 1; i < 6; i = i + 1){
        j = 0;
        do{
            j = i + 1;
            if(i % 2 == 0 and j % 2 == 0){
                print("* ");
            }else {
                print("%d ",j);
            }
        } again if(j < 5)
        print("\n");
    }
    return 0;
}
```

Código Resultado do *Beautifer* (1)

```
integer func2(
    integer x,
integer y){
integer func3(){integer func4(){
return x + y;}return x + y + func4();
    }



    return x
    +
    y + func3();
} integer main() {

    integer array listIntegers = [     1,
    2,3,4,5];integer r = 0;loop
(integer i of listIntegers){
    if(i % 2 == 0){r = func2(i,5);
}else{loop while(i < 10){
print("Hello!\n");
                i =
        i +
1;}}}return r;
}
```

Código Hertz mal formatado (2)

```
integer func2(integer x,integer y){
    integer func3(){
        integer func4(){
            return x + y;
        }
        return x + y + func4();
    }
    return x + y + func3();
}
integer main(){
    integer array listIntegers = [1,2,3,4,5];
    integer r = 0;
    loop(integer i of listIntegers){
        if(i % 2 == 0){
            r = func2(i,5);
        }else {
            loop while(i < 10){
                print("Hello!\n");
                i = i + 1;
            }
        }
    }
    return r;
}
```

Código Resultado do *Beautifer* (2)

```
integer main() {integer r = 5;
if (r == 4) {
print("Hello World 1");
}
else if (r == 5) {
print("Hello World 2");
}
else if (r == 1) {
print("Hello World 3");
}
else if (r == 2) {
print("Hello World 4");
}
else {
print("Hello World 5");
}return 0;}
```

Código Hertz mal formatado (3)

```
integer main(){
    integer r = 5;
    if(r == 4){
        print("Hello World 1");
    }else if(r == 5){
        print("Hello World 2");
    }else if(r == 1){
        print("Hello World 3");
    }else if(r == 2){
        print("Hello World 4");
    }else {
        print("Hello World 5");
    }
    return 0;
}
```

Código Resultado do *Beautifer* (3)

Capítulo 6

Conclusão

As decisões relativas aos momentos de preenchimento do ficheiro HTML foram as mais determinantes para o projeto. A escolha de uma travessia adicional (para recolhas de informações sobre estruturas condicionais) foi também uma decisão importante para a manutenção e gestão do código. Aliás, é comum que compiladores de linguagem de programação tenham várias fases correspondentes a várias travessias (ou passagens) pelo código fonte. Geralmente estas fases correspondem a etapas distintas do processo de compilação e realizam diferentes tarefas. Por analogia, seguimos esta mesma metodologia de divisão de tarefas.

Para trabalho futuro, planeamos uma **refactorização** do código a nível da etapa da Travessia Geral. Esta refactorização encapsulará ainda mais as operações dentro da classe `ControladorDeVariaveis`, por forma a diminuir a complexidade do *AnalisadorGeral* ao nível das variáveis de estados. Para além disto, uma classe *ControladorDeInstrucoes* poderá vir a ser útil, uma vez que o módulo *ControladorDeVariaveis* passou a operar sobre conceitos para além de variáveis. Ora, como o seu propósito inicial era somente a gestão da *stackDeVariaveis*, faz sentido uma separação do código não destinado a tal *stack*.

O estado final do projeto consiste no cumprimento de todos os requisitos enunciados para o Analisador, assim como a adição do projeto paralelo de *Beautifiers* da linguagem Hertz. Como requisito futuro para adição de novos recursos, será considerado a visualização de alternativas de refactorização do código fonte mediante as tais estruturas de controlo condicionais passivas de serem substituídas. Adicionalmente, será expandida a abstração de uma variável para o armazenamento de seu **tipo** e assim, poder ser incluída informação relativa aos **tipos** das variáveis no relatório Hertz. Esta informação pode ser, por exemplo, a **verificação de tipos** no código. Isto pode garantir que as operações e expressões que estão sendo executadas envolvam tipos de dados compatíveis.