Universidade do Minho

Mestrado em Engenharia Informática

2011

Análise e Conceção de Software
Análise e Transformação de Software
Portal para Certificação da Qualidade de Programas *Tiger*

Ano Lectivo de 2011/2012

PG20189 **André Pimenta Ribeiro**
PG19824 **Cedric Fernandes Pimenta**
PG20978 **Rafael Fernando Pereira Abreu**

9 de Julho de 2012

Resumo

Neste relatório, e no âmbito do módulo de Análise e Transformação de Software, iremos apresentar uma análise efetuada à linguagen *Tiger* e respetivos programas. Este estudo tem como pressuposto a criação de uma aplicação Web que seja capaz de efetuar uma avaliação objetiva a programas escritos na dita linguagem.

A realização desta análise passa por vários passos importantes, desde a criação de um *parser* para interpretar a linguagem *Tiger*, definição de métricas a utilizar nas avaliações, assim como o modelo de avaliação baseado na norma **ISO 9126**.

Posto isto, será apresentada a modelação do portal de certificação de qualidade que tem como objetivo permitir a visualização daquele que será o espaço onde o utilizador submeterá o seu programa (na linguagem *Tiger*) para avaliação.

Por fim, será documentado todo o desenvolvimento da mencionada aplicação Web e o modo de funcionamento desta, sendo que neste processo, foram implementadas, através da ferramenta *TOM*, as métricas referidas acima.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contextualização, Apresentação do Objeto de Estudo e Objetivos

Com o desenvolvimento da tecnologia e com a existência de cada vez mais "programadores", começa-se a necessitar de um maior controlo sobre o software desenvolvido, nomeadamente a avaliação da qualidade do código desenvolvido.

Porém, atribuir uma avaliação a um programa desenvolvido em qualquer linguagem não é um processo simples e linear, antes pelo contrário, trata-se de um processo muitas vezes subjetivo e alvo de discussões, sobre quais as melhores métricas, entre outros assuntos provenientes do processo de avaliação.

Neste trabalho prático é proposto a criação de um portal de certificação de qualidade de programas escritos na linguagem *Tiger* que tem como principal objetivo a atribuição de uma avaliação de qualidade a um programa que tenha sido submetido. Tal como já foi referido, o processo de avaliação nem sempre se mostra ser um processo pacífico. A avaliação automática, em certos pontos, é possível desde que se faça uma análise correta da linguagem e se destaquem características importantes destas, que deverão ser sempre cumpridas de forma a se poder atingir um grau de qualidade satisfatório em qualquer software.

Para atingir as características exatas, tal como foi referido, foi criado um catálogo de métricas que se distinguem nos programas *Tiger* e, posteriormente, foi efetuada uma análise baseada na norma **ISO 9126**, tendo como fundamentos as métricas especificadas. Esta mesma análise, ou melhor, este modelo de qualidade foi colocado em funcionamento numa aplicação Web, que tem como principal objetivo a certificação da qualidade dos programas *Tiger* que lhe serão submetidos.

Toda a avaliação encontra-se exposta à supervisão de um administrador que, tendo em conta as diferentes estatísticas apresentadas, pode definir, periodicamente ou quando assim o deseja, os parâmetros de qualidade sobre os quais os programas *Tiger* são avaliados.

1.2 Motivação

"... Um dos objetivos da engenharia de software é a transformação da criação de software de uma maneira artística, indisciplinada e pouco entendível para uma forma devidamente controlada, quantificável e previsível..."

Pretende-se, com este trabalho prático, a criação de um portal que seja capaz de avaliar programas escritos na linguagem *Tiger*, mas, sobretudo, o desenvolvimento de capacidades

que nos garantam uma boa prática no desenvolvimento e avaliação de software de forma genérica, e não apenas no caso de estudo.

Pesquisas realizadas em empresas de software indicam que mais de metade dos grandes projetos de software se deparam com algum tipo de atraso, excesso de custo/prazo ou algum fracasso na execução aquando da implementação por falta de controlo. As **métricas de software** são um assunto já discutido à muitos anos, mas poucas vezes utilizadas, o que provoca o acontecimento de situações como a referida anteriormente.

Uma métrica é uma medição de um atributo (propriedade ou característica) de uma determinada entidade (produto, processo ou recurso) que será utilizada, neste caso de estudo, para efetuar medidas ao código da linguagem em questão (neste caso, a linguagem *Tiger*). A utilização de métricas irá permitir o aperfeiçoamento dos processos de desenvolvimento de software mas, acima de tudo, permitir uma avaliação real e objetiva. A avaliação já referida tem como propósito permitir a identificação das melhores práticas no desenvolvimento de software, bem como a avaliação do impacto da variação de um ou mais atributos do produto ou processo na qualidade e/ou produtividade.

Assim, a nossa grande fonte de motivação surge por todos os fatos mencionados acima, ou seja, a obtenção de novos conhecimentos que nos tornam capazes de construir processos de avaliação de software ou de escolher os principais parâmetros a ter em conta nessa avaliação, sem deixar de tomar em consideração toda a informação adquirida no desenvolvimento de um *parser*, através de uma gramática independente de contexto, e da criação de estruturas de dados apropriadas, recorrendo à ferramenta *TOM*, para aplicação a uma qualquer linguagem.

1.3 Estrutura

O presente relatório é constituído por 6 capítulos: **Introdução** onde é feita a contextualização dos objetivos que nos foram propostos; **Domínio do problema** onde serão apresentados os conceitos necessários para um melhor entendimento do problema; **Qualidade de programas *Tiger*** onde é apresentada a solução desse mesmo problema e **Implementação** que possui o intuito de mostrar a aplicação Web desenvolvida e toda a tecnologia utilizada no processo de avaliação do código *Tiger*. Por fim, o capítulo **Conclusão** que possui um pequeno balanço sobre a realização deste trabalho e particularidades que possuam alguma relevância.

Capítulo 2

Domínio do problema

2.1 O que é a linguagem *Tiger*

A linguagem *Tiger* é uma linguagem de programação criada por AndrewAppelm, que é usada no livro ModernCompilerImplementationInML como exemplo.

Tiger é uma linguagem que segue o paradigma imperativo, e que apresenta as seguintes características:

- **Tipos de dados**
  - *Integers*
  - *Strings*
  - *Arrays*
  - *Records*
- **Declarações**
  - *Variáveis*
  - *Funções*
  - *Tipos*
- **Controlo de fluxo**
  - *If-Then-Else*
  - *If-Then*
  - *While*
  - *For*
  - *Let-In*
- **Atribuições**

2.2 Análise da linguagem *Tiger*

De modo a construir o parser para esta linguagem, foi necessário construir uma gramática que conseguisse interpretar qualquer tipo de código construído corretamente nesta linguagem. Assim, foi utilizado o software ANTLR para a definição da referida gramática sendo que esta passou por diversos níveis de refinamento até à obtenção do resultado final. De forma a

tornar a leitura desta gramática o mais compreensível possível, iremos dividi-la em pequenos excertos, explicando-os de forma clara. Consideramos importante referir que a gramática construída funciona apesar de existirem alguns *warnings*, acontecimento que se deve ao facto de a sintaxe desta linguagem permitir um grau de liberdade elevado (como é comum), o que torna possível construir-se uma mesma expressão pelo uso de diferentes conjuntos de regras. Uma vez referenciado isto, temos então:

```
options {
  k=2;
}
tokens {
        ARRAYOF = 'array of' ;
        IF = 'if' ;
        THEN = 'then' ;
        ELSE = 'else' ;
        WHILE = 'while' ;
        FOR = 'for' ;
        TO = 'to' ;
        DO = 'do' ;
        LET = 'let' ;
        IN = 'in' ;
        END = 'end' ;
        OF = 'of' ;
        BREAK = 'break' ;
        NIL = 'nil' ;
        FUNCTION = 'function' ;
        VAR = 'var' ;
        TYPE = 'type' ;
        IMPORT = 'import' ;
        PRIMITIVE = 'primitive' ;
        IGUAL = '=' ;
        MAIS = '+' ;
        MULT = '*' ;
        DIV = '/' ;
        AND = '&' ;
        OR = '|' ;
        DPIGUAL = ':=' ;
        AP = '(' ;
        FP = ')' ;
        AC = '{' ;
        FC = '}' ;
        APR = '[' ;
        FPR = ']' ;
        DP = ':' ;
        VIR = ',' ;
        PVIR = ';' ;
        PONTO = '.' ;
}
```

Tendo em conta que o ANTLR é uma ferramenta poderosa e que a linguagem Tiger permite aos seus utilizadores alguma liberdade na sintaxe a utilizar, decidimos aumentar o *lookahead* da nossa gramática para 2 (k=2). Para além disto, após uma análise cuidada à linguagem em questão, definimos as palavras e símbolos reservados desta em tokens.

```
prog            : exp ;
exp                     : expOR expORPr ;
expOR           : expAND expANDPr ;
expAND          : aritExp relExp ;
expORPr         : OR exp
                        | //vazio
                        ;

expANDPr        : AND exp
                        | // vazio
                        ;
```

Como é possível verificar neste excerto, temos definido diversas regras que surgem devido à necessidade da gramática ser LL(k) (gramática que lê o texto da esquerda para a direita e produz uma derivação mais à esquerda). Estas são as regras principais apartir das quais toda a árvore abstracta se desenvolverá.

```
aritExp         : term termPr ;
relExp          : RELOP aritExp
                        | //vazio
                        ;

term            : factor factorPr ;
termPr          : (MAIS|NEG|IGUAL) term termPr
                        | //vazio
                        ;

factorPr        : (MULT|DIV) factor factorPr
                        | //vazio
                        ;

factor          : NIL
                        | INT
                        | STRING
                        | AP expList FP
                        | NEG exp
                        | IF exp THEN exp (ELSE exp)?
                        | WHILE exp DO exp
                        | FOR ID DPIGUAL exp TO exp DO exp
                        | BREAK
                        | LET decList IN expList END
                        | lValue
                        ;
```

Aqui temos mais uma vez algumas regras auxiliares (pelo motivo acima referido). Para além destas regras relevantes, importa destacar a regra **factor** que como é facilmente observável, se trata da regra usada na determinação do tipo de controlo de fluxo ou expressão detetada no código a analisar.

```
decList         : dec* ;
dec                     : tyDec
                        | varDec
                        | funDec
                        ;

tyDec           : TYPE typeId IGUAL ty ;
ty                      : AC fieldList FC
                        | ARRAYOF typeId
                        | typeId
                        ;

fieldList       : (ID (DP|IGUAL) typeId (intFieldList)*)? ;

intFieldList    : VIR ID (DP|IGUAL) typeId
                                | APR exp FPR (OF exp)?
                                ;

typeId          : ID
                        | STRING
                        | INT
                        | NIL
                        ;

varDec          : VAR ID (DP typeId)? (DPIGUAL|IGUAL) exp ;
funDec          : FUNCTION ID AP fieldList FP (DP typeId)? IGUAL exp ;
lValue          : ID (functionRecordArray);

functionRecordArray
                : AP argList FP
```

```
                | AC fieldList FC
                | (APR exp FPR)? (OF exp|(PONTO ID |APR exp FPR)* (DPIGUAL exp))?
                ;

expList         : (exp (PVIR exp)*)? ;
argList         : (exp (VIR exp)*)? ;
```

Neste conjunto de regras, definimos as várias formas de declarações que existem na linguagem *Tiger* bem como os diferentes tipos que as variáveis podem ter. Deste modo é, ainda, possível efetuar corretamente o parser das mais variadas formas de escrita de código que é permitido.

```
RELOP           : '<'|'>'|'<>'|'<='|'>=' ;

NEG                     : '-' ;

ID                      : (LETTER (LETTER|DIGIT|'_')*|'_main') ;
INT                     : DIGIT+ ;
STRING          : '"'~('"')*'"' ;
fragment DIGIT  : '0'..'9' ;
fragment LETTER
                        : ('a'..'z'|'A'..'Z') ;
NESTED_ML_COMMENT
                :   '/*'
                    (options { greedy=false; } : (NESTED_ML_COMMENT|.))*
                    '*/' { $channel=HIDDEN; }
                ;
WHITESPACE      : ('\t'|' '|'\r'|'\n'|'\u000C')+ { $channel = HIDDEN; } ;
```

Por fim, temos o excerto que contém as várias regras terminais. Definimos as diferentes possibilidades de uma operação de relação e a negação bem como é composto um **Id**, um **Int** e uma **String** (esta definição baseia-se na já mencionada análise da sintaxe da linguagem *Tiger*). O uso de *fragments* prende-se com a ferramenta usada e é uma simples particularidade; por fim, temos as regras que permitem ignorar os comentários (neste caso, tem em conta a possibilidade de comentários aninhados) e os espaços em branco.

Capítulo 3

Qualidade de programas *Tiger*

3.1 Métricas de Análise

O processo de validação e certificação de software é cada vez mais importante, assim como necessário, para a construção do mesmo. Mas como podemos afirmar que um programa é bom ou mau? Só existe bons programas se existirem maus programas, e é com os conhecimentos, com a experiência da interpretação e com o estudo de vários programas, que atingimos a noção de bom ou mau programa.

No entanto, é importante saber quais as caraterísticas que podem ser utilizadas e que, simultâneamente possuem uma posição relevante na decisão de um bom ou mau software. Estas caraterísticas serão as métricas que utilizaremos para a análise de programas escritos em *Tiger*.

... "uma métrica de qualidade de software é uma função cujos inputs são os dados do software e cujo a saída é um único valor numérico que pode ser interpretado como o grau que determina a qualidade do software..."

Existem, então, diversas métricas, tanto diretas como indiretas, capazes de contribuir para um feedback à cerca da qualidade de um programa, no entanto nem todas são objetivas ou podem ser usadas sem a experiência e conhecimento humano, isto é, de forma automática. Por isto, decidimos apresentar um catálogo de métricas, que contém um conjunto que reúnam o mínimo de consenso e que sejam passíveis de utilização na linguagem *Tiger*. Estamos, por isso, a falar de métricas que devem ser calculáveis, facilmente entendidas e testadas, que sejam passíveis de estudos estatísticos e que, consequentemente, possam ser expressas numa determinada unidade. Por último, e para o caso de estudo em concreto, deverão ser repetíveis e independentes do observador.

1. **Linhas de Código**: O número de linhas de código é, possivelmente, a métrica mais comum entre as diferentes linguagens, assim como a mais evidente. Com o número de linhas de código pretende-se, sobretudo, analisar a dimensão do código, pois como é do senso comum da engenharia de software *"...Um bom código não é um código grande..."*.

2. **Métricas de Complexidade Ciclomática de McCabe**. Métricas que se baseiam na medida do número de caminhos linearmente independentes de um módulo ou função. Esta métrica conta o número de caminhos independentes que se pode tomar apartir do código fonte, e atribui uma pontuação numérica única para cada método.
   A complexidade ciclomática começa com um valor inicial de um e a incrementa em um sempre que existe a ocorrência da declaração de um *if*, *for* e *while* e das operações

boleanas & e |.

3. **Métricas Halstead**: São métricas que se baseia na complexidade do problema e das funções. Estas métricas têm em conta o número de operadores distintos ou o número de operandos distintos mas também o número total de operadores e operandos. Após uma breve pesquisa, descobrimos que através dos valores agregados, seria possível aplicar um conjunto de fórmulas de modo a obter determinados valores, sendo que a nossa opção recaiu sobre o cálculo da dificuldade em escrever ou perceber o código submetido pela seguinte fórmula:

$$D = (n1/2) * (N2/n2)$$

onde *n1* corresponde ao número de operadores distintos, *n2* ao número de operandos distintos e *N2* ao número total de operandos.

4. **Número de Comentários**: A existência de comentários ao longo do código é sempre positiva, pois deve ter como objetivo auxiliar a interpretação deste. Pode-se até ser mais exigente e afirmar que a existência de um comentário (contendo uma descrição de utilidade, bem como os seus princípios de funcionamento) antecedendo uma função/método deveria ser obrigatório.

5. **Número de Erros**: A existência de erros em qualquer programa é um indicador por si só de má qualidade, ou melhor, de que no exato momento a que se submeteu à avaliação, se trata de um software inválido. Pode-se, então, considerar o número de erros como uma métrica com utilidade para o caso de estudo.

6. **Número de Funções**: A contagem do número de funções presente em determinado código pode não ser um fator importante por si só mas é importante para relativizar as outras métricas. Para além disto, quanto mais funções o código apresentar, possivelmente melhor documentado e menos complexo ele se encontra. Por exemplo, um programa com apenas uma função e muitas linhas de código não é positivo mas se tiver muitas linhas de código mas também usar muitas funções, pode ser compreensível essa opção.

7. **Tempo**: Esta é uma métrica que se encontra indiretamente relacionada com a complexidade do problema; no entanto, iremos apresentar o tempo de análise como uma métrica a ter em consideração no processo de benchmarking.

3.2 Modelo de Qualidade

Será criado um modelo de qualidade baseado na norma **ISO 9126** que se apresenta na figura a baixo, e que será utilizado para avaliar os programas *Tiger*.

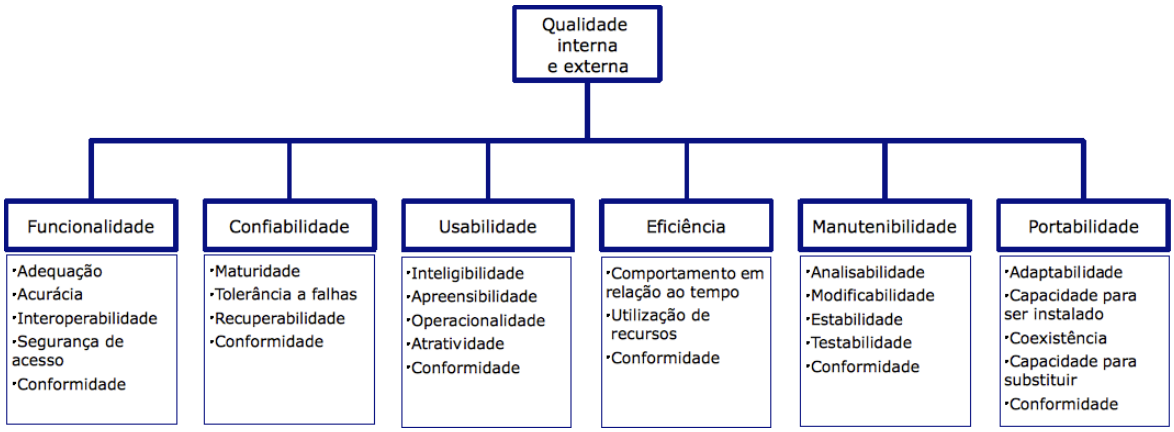

Figura 3.1: Modelo de avaliação **ISO 9126**

3.2.1 Modelo de avaliação ISO 9126

Funcionalidade

- *Confiança*: pretende-se analisar a confiança do código através do **número de linhas**, pois funções com um número de linhas pequeno sujeita-se a um menor número de erros. **Número de erros** existentes ou não num programa *Tiger* ditam de forma objectiva se este consegue transmitir uma sensação de confiança ao utilizador.

- *Adequação*: pretende-se avalizar a adequação, através da existência de **comentários**, pois permitem explicar qual o propósito de cada função e verificar se cumpre ou não os objetivos da mesma.

Segurança

- *Maturidade*: mais uma vez o **número de erros** vai ter influência, pois um código com erros tem uma maturidade bastante baixa, ou seja, ainda está propício a muitos desenvolvimentos.
  A existência de diferentes operadores e a correspondente complexidade do problema pode ser sinal de maturidade, sendo a **métrica Halstead** uma boa solução para avalizar.
  A análise da **complexidade ciclomática** também será uma boa opção pois permite analisar a simplicidade do programa que muitas vezes se encontra ligada diretamente/indiretamente à maturidade desta e o **número de funções** pode ser um bom indicador do nível de desenvolvimento de determinado código.

- *Tolerância a falhas*: o **número de erros** define automaticamente se este tem falhas ou não, no entanto, métricas como a **complexidade ciclomática** e as de **Halstead**) também podem determinar a tolerância a falhas do programa, pois quanto mais complexo for o código, maior a probabilidade de existirem erros.

Da mesma forma, o **o número de linhas** pode determinar a tolerância a falhas do problema até porque quanto maior for o programa, maior pode ser a dificuldade de corrigir determinada falha.

Usabilidade

- *Inteligibilidade*: para esta característica podemos, desde já, realçar o **número de linhas** e o **número de funções**, pois um código pequeno e modelar será mais fácil de analisar.
  Mais uma vez, a complexidade (**Complexidade Ciclomática** e de **Halstead**) está diretamente ligada à inteligibilidade, já que quanto menos complexo for um programa, maior será a facilidade de o perceber.
  Por fim, e não menos importante, a existência de **comentários** é essencial para um melhor entendimento de cada função.

- *Apreensibilidade*: mais uma vez, o **número de linhas**, a **Complexidade Ciclomática**, o uso de **comentários** e o **número de funções** serão as métricas mais adequadas para a avaliação desta característica.

Eficiência

- *Utilização de Recursos*: esta característica pode ser avalizada através das **métricas de Halstead** e da **Complexidade Ciclomática**, pois uma menor complexidade significará uma menor utilização dos recursos disponíveis.

Manutenção

- *Analisabilidade*: uma vez mais, o **número de erros** é o indicador mais direto da capacidade de identificar erros no código.
  A **Complexidade Ciclomática e as métricas de Halstead** também vão estar indiretamente ligada a capacidade de analisabilidade dos programas *Tiger*, assim como o **número de linhas** e o **número de funções** pelos motivos já referidos relativos à extensão e à modularidade de determinado código.

Portabilidade

- *Adaptabilidade*: a **Complexidade Ciclomática** é uma métrica que pode definir de forma clara a capacidade de adaptabilidade da mesma forma que o **número de funções** também pode.

Modelo de qualidade baseado na norma **ISO 9126**		
Característica	Sub-Característica	Métricas a usar
Funcionalidade	Confiança	**1 e 5**
	Adequação	**4**
	Precisão	Não definido
	Segurança	Não definido
	Interoperabilidade	Não definido
Segurança	Maturidade	**2, 3, 5, 6**
	Tolerância a falhas	**1, 2, 3, 5**
	Recuperabilidade	Não definido
Usabilidade	Inteligibilidade	**1, 2, 3, 4, 6**
	Apreensibilidade	**1, 2, 4, 6**
	Operabilidade	Não definido
Eficiência	Comportamento em relação ao tempo	Não definido
	Utilização de recursos	**2 e 3**
Manutenção	Analisabilidade	**1, 2, 3, 5, 6**
	Modificabilidade	Não definido
	Estabilidade	Não definido
	Testabilidade	Não definido
Portabilidade	Adaptabilidade	**2 e 6**
	Capacidade para ser instalada	Não definido
	Capacidade para substituir	Não definido
	Coexistência	Não definido

Tabela 3.1: Modelo de avaliação *Tiger*

Capítulo 4

Modelação do Portal em *WebRatio*

O portal de certificação de programas *Tiger* irá funcionar na base de dois princípios iniciais. O primeiro princípio baseia-se na funcionalidade que permitirá a qualquer pessoa que pretenda avaliar um programa desenvolvido em *Tiger*, a capacidade de submeter esse programa, para que este possa ser avaliado pelo portal, oferecendo, assim, uma capacidade de avaliação automática do código submetido ao utilizador. Já o segundo princípio baseia-se na existência de uma entidade denominada de administrador, que irá periodicamente, ou quando assim achar necessário, ajustar os valores e critérios de avaliação dos programas *Tiger*.

Estamos, então, a falar de uma aplicação Web que contém essencialmente dois tipos de utilizadores: o utilizador normal que irá submeter os programas *Tiger*, e o administrador que irá definir os pesos de cada métrica e atribuir um determinado valor a cada um destes, baseando-se nas estatísticas obtidas sobre as métricas de avaliação ao longo do tempo.

Será uma aplicação simples, que se irá basear num conjunto de métricas de avaliação, tendo como principal pressuposto a estatística e a classificação do código submetido.

Modelo de Dados

O modelo de dados implementado no WebRatio consiste em 8 entidades que permitem guardar todas as informações importantes e relevantes sobre o Portal:

- *User*
- *Group*
- *Module*
- *ProgramaTiger*
- *CaracPrograma*
- *PesoMetricas*
- *BomPrograma*
- *MauPrograma*
- *RepositorioBase*

Na Entidade *User* é guardada a informação sobre os utilizadores tal como nome de utilizador, e-mail e senha para acesso à págna pessoal no portal.

Como já foi referido anteriormente só existem dois tipos de utilizadores, o utilizador normal e o administrador, por isso, precisamos de ter a entidade *Group*. Cada grupo tem acesso a *SiteViews* diferentes por isso é necessária a entidade *Module* que identifica as *SiteViews* disponíveis para cada grupo de utilizadores.

Como o objetivo do Portal é certificar a qualidade dos programas *Tiger* submetidos pelos utilizadores, criámos a entidade *ProgramaTiger*, que guarda o nome do programa, uma pequena descrição das funcionalidades, a data da sua submissão, o nome do ficheiro submetido e a sua avaliação em estrelas.

Cada ficheiro submetido será automaticamente avaliado segundo as métricas definidas pelo administrador, por isso, para cada programa Tiger, é relevante guardar as características desse mesmo programa (*CaracPrograma*). Para cada métrica especificada, existe um peso/avaliação associado (PesoMetricas) e que é inserido pelo administrador, permitindo avaliar o programa e concluir sobre a qualidade do mesmo.

Consoante a avaliação atribuída a cada programa submetido, será armazenada uma referência para este, ou na entidade *BomPrograma* quando têm avaliação acima de 3 estrelas ou na entidade *MauPrograma* caso tenha 3 ou menos estrelas. Esta divisão dos programas permite uma melhor gestão e certificação da qualidade sobre os programas submetidos à medida que o tempo passa e o número de programas submetidos aumenta.

Em último, temos a tabela *RepositorioBase* que será responsável por guardar a informação relativa aos programas que são tomados como referência na classificação das diferentes métricas, aquando da análise dos programas submetidos.

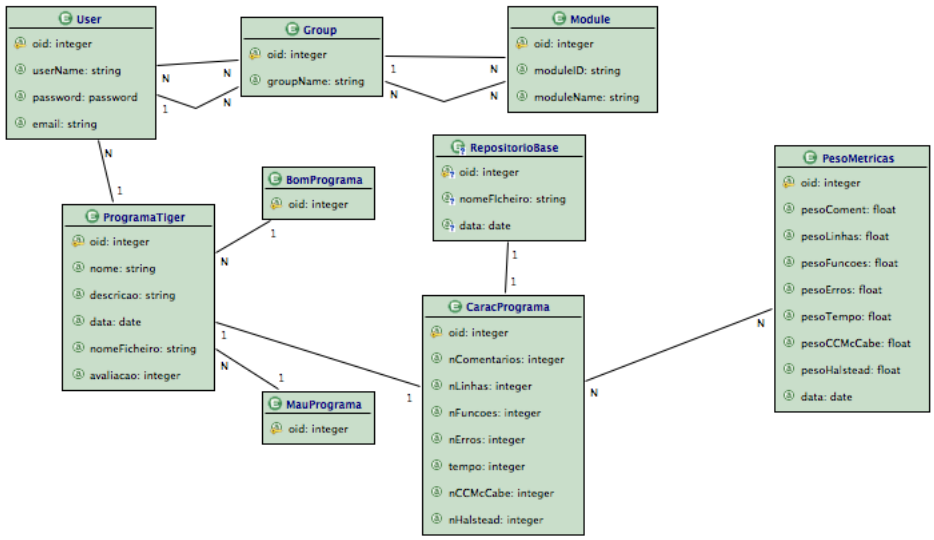

Figura 4.1: Modelo de Dados

Para além deste modelo de dados, através da ferramenta WebRatio, foi modelado o futuro portal de certificação da linguagem *Tiger* que demonstraremos, em seguida, com as *SiteViews* dos diferentes utilizadores acima identificados (de modo a tornar esta documentação o mais abrangente possível, optamos por apresentar a página web gerada pela ferramenta para caso do utilizador usufruir do estatuto de administrador e mostrámos os elementos que constituem a página web que é gerada caso se trate de um utilizador regular). Resta apenas mencionar que nesta modelação, já prevemos a autenticação de um utilizador (um utilizador passa a ter acesso a todos os programas que já submeteu) e o registo de novos utilizadores.

Figura 4.2: Página Inicial de Autenticação

Como podemos ver, a primeira página que um utilizador visita é a página de autenticação onde ele deve efetuar o login ou, caso não possua uma conta, registar-se e criar uma conta.

Figura 4.3: SiteView para um Administrador

Tal como foi dito acima, um administrador irá possuir um conjunto de ações que os utilizadores comuns não possuem. Assim, e para além das funcionalidades de editar a sua conta ou ele próprio submeter programas, poderá visualizar todas as submissões que foram efetuadas bem como as caraterísticas de cada código submetido, terá acesso a um conjunto de estatísticas sobre as avaliações efetuadas e poderá definir os pesos que cada métrica possui na avaliação final do código submetido, consoante aquilo que considere ser o mais correto. Terá, ainda, acesso à lista de programas que foram submetidos e que foram considerados bons/maus por este portal (dois repositório distintos). Por fim, existe o repositório base de programas *Tiger* que serve de referência para determinar os valores padrões de cada métrica e ao qual o administrador pode adicionar novos programas caso assim o entenda.

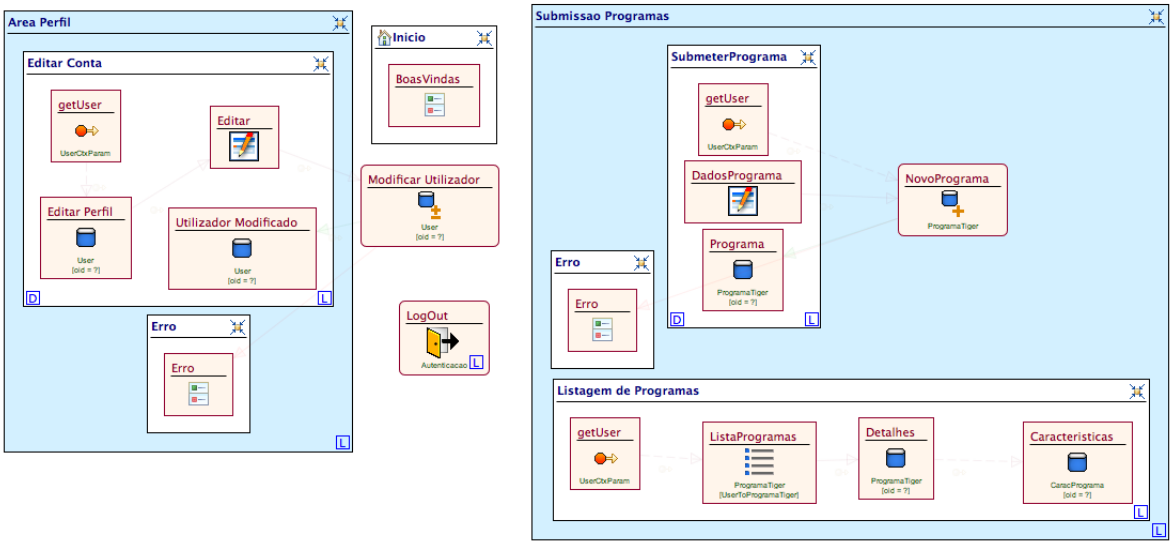

Figura 4.4: Componentes WebRatio para SiteView de um Utilizador

Como é possível verificar pelos componentes usados no WebRatio, a siteview dos utilizadores pretende ser o mais simplificada possível de modo a tornar todo o processo fluído. Assim, temos a àrea em que permitimos ao utilizador alterar os seus dados pessoais e, depois, tem a àrea para submissão dos programas que consiste na inserção do programa e dos seus atributos na nossa base de dados, sendo que em caso de sucesso, é demonstrada a informação inserida mais a avaliação que foi efetuada. Uma vez que os utilizadores possuem uma conta, pensamos que seria atraente que o utilizador pudesse ver a lista de todas as suas submissões e que nos detalhes fosse demonstrado os atributos referidos e as caraterísticas que o nosso portal retirou do código fornecido.

Capítulo 5

Implementação

Neste capítulo, será feita uma descrição do desenvolvimento efetuado nas diferentes fases da implementação do projeto. Uma vez que algumas destas fases já foram explicadas de modo detalhado anteriormente neste relatório, apenas iremos mencionar como elas são integradas na aplicação Web desenvolvida. As seguintes secções estão ordenadas segundos a ordem pela qual foram desenvolvidas.

5.1 *Parsing*

Esta foi a fase inicial do projeto e que consistiu na análise da sintaxe da linguagem *Tiger* e no desenvolvimento da gramática independente de contexto com capacidade para percorrer um programa da mencionada linguagem, desde que este esteja correto. Esta fase já foi explicada ao pormenor no segundo capítulo deste relatório, restando apenas mencionar que a gramática desenvolvida é um passo importantíssimo no uso da ferramenta TOM (fase seguinte).

5.2 Ferramenta TOM

Como referido na página oficial do *Tom* (*tom.loria.fr*), trata-se de uma extensão da linguagem projetada para manipular estruturas em árvore e documentos XML, permitindo fazer reconhecimento de padrões, facilitando a inspeção de objetos e na recuperação de valores.

Outras das grandes vantagens de usar *Tom* é que pode ser usado em conjunto com muitas linguagens desde *C*, *Java*, *Python*, *C++*, *C#* entre outras.

O *Tom* cria a árvore sintática abstrata (AST) da linguagem *Tiger* permitindo analisar se determinado ficheiro submetido no portal está bem estruturado nesta linguagem.

Para isso tivemos escrever um compilador e interpretador de programas em *Tiger* através do *parsing* já referido anteriormente.

De seguida mostramos os passos que seguimos para criar o compilador e interpretador.

5.2.1 Árvore de Sintaxe Abstrata (AST)

O ficheiro *Gom* fornece uma sintaxe para definir de forma concisa a árvore de sintaxe abstrata, cada ficheiro permite definir os módulos da linguagem e em cada módulo é identificado os operadores e os seus tipos.

A seguir podemos ver o exemplo da árvore abstrata (*Rec.gom*) de um *Factor* sendo que o ficheiro completo se encontra em anexo.

```
module parser.Rec
        imports int String
        abstract syntax

        (...)

        Factor = FNil()
                   | FInteger(i:int)
                   | FString(s:String)
                   | FExpList(el:ExpList)
                   | FNeg(e1:Exp)
                   | FWhile(e1:Exp, e2:Exp)
                   | FFor(id:String, e1:Exp, e2:Exp, e3:Exp)
                   | FBreak()
                   | FLet(dl:DecList, el:ExpList)
                   | FLvalue(lv:LValue)
                   | FIfThen(e1:Exp, e2:Exp)
                   | FIfThenElse(e1:Exp, e2:Exp, e3:Exp)

        (...)
```

5.2.2 *Parsing*

Para fazer *parsing* de um ficheiro usamos a gramática já definida em *ANTLR* e associamos a cada regra da gramática o módulo que a define na árvore abstrata. Com isso conseguimos definir a sintaxe concreta dos programas em *Tiger*.

Para implementar o *parser* criamos um ficheiro (*Rec.g*) que pode ser visto em anexo. De seguida apenas apresentamos um exemplo:

```
factor : 'nil'
                    -> ^(FNil)
           | INT
                           -> ^(FInteger INT)
           | STRING
                           -> ^(FString STRING)
           | '(' expList ')'
            -> ^(FExpList expList)
           | '-' e1=exp
                    -> ^(FNeg $e1)
           | 'while' e1=exp 'do' e2=exp                                          -> ^(
    FWhile $e1 $e2)
           | 'for' ID ':=' e1=exp 'to' e2=exp 'do' e3=exp       -> ^(FFor ID $e1 $e2 $
    e3)
           | 'break'
                    -> ^(FBreak)
           | 'let' decList 'in' expList 'end'                              -> ^(FLet
    decList expList)
           | lValue
                           -> ^(FLvalue lValue)
           | ('if' e1=exp 'then' e2=exp ('else' e3=exp)?        -> {e3==null}? ^(
    FIfThen $e1 $e2)

     ^(FIfThenElse $e1 $e2 $e3))
```

5.2.3 Interpretador

Depois de fazer o *parsing* de um ficheiro submetido temos que interpretar os dados extraídos. O tom permite facilmente utilizando um estilo de programação funcional armazenar e analisar e valores capturados.

Para implementar o interpretador criamos um ficheiro (*Rec.t*) que também pode ser visto na integra em anexo. Mais uma vez de seguida demonstramos o exemplo do interpretador de um *Factor* em *Tiger*:

```
private void interpFactor(Factor f) {
    %match(f) {
        FNil() -> {
                if(!operandos.contains("nil")) operandos.add("nil");
                nOperandos++;
        }

        FInteger(i) -> {
                if(!operandos.contains(Integer.toString(`i))) operandos.add(Integer.
   toString(`i));
                nOperandos++;
        }

        FString(s) -> {
                if(!operandos.contains(`s)) operandos.add(`s);
                nOperandos++;
        }

        FExpList(el) -> {
                if(!operadores.contains("(")) operadores.add("(");
                `interpExpList(el);
                if(!operadores.contains(")")) operadores.add(")");
        }

        FNeg(e1) ->  {
                if(!operadores.contains("-")) operadores.add("-");
                `interpExp(e1);
        }

        FWhile(e1, e2) ->  {
                if(!operadores.contains("while")) operadores.add("while");
                `interpExp(e1);
                if(!operadores.contains("do")) operadores.add("do");
                `interpExp(e2);
                nMcCabe++;
        }

        FFor(id, e1, e2, e3) ->  {
                if(!operadores.contains("for")) operadores.add("for");
                if(!operandos.contains(`id)) operandos.add(`id);
                nOperandos++;
                if(!operadores.contains(":=")) operadores.add(":=");
                `interpExp(e1);
                if(!operadores.contains("to")) operadores.add("to");
                `interpExp(e2);
                if(!operadores.contains("do")) operadores.add("do");
                `interpExp(e3);
                nMcCabe++;
        }

        FBreak() ->  {
                if(!operadores.contains("break")) operadores.add("break");
                //System.out.println("Break");
        }

        FLet(dl, el) ->  {
                if(!operadores.contains("let")) operadores.add("let");
                `interpDecList(dl);
                if(!operadores.contains("in")) operadores.add("in");
                `interpExpList(el);
                if(!operadores.contains("end")) operadores.add("end");
        }

        FLvalue(lv) -> {
                `interpLValue(lv);
        }

        FIfThen(e1, e2) -> {
                if(!operadores.contains("if")) operadores.add("if");
                `interpExp(e1);
                if(!operadores.contains("then")) operadores.add("then");
```

```
                'interpExp(e2);
                nMcCabe++;
        }

        FIfThenElse(e1, e2, e3) -> {
                if(!operadores.contains("if")) operadores.add("if");
                'interpExp(e1);
                if(!operadores.contains("then")) operadores.add("then");
                'interpExp(e2);
                if(!operadores.contains("else")) operadores.add("else");
                'interpExp(e3);
                nMcCabe++;
        }
    }
}
```

5.2.4 Compilador de *Tiger*

Por fim para criar o compilador utilizando os três ficheiros referidos anteriormente (*Rec.gom*, *Rec.g* e *Rec.t*) utilizamos o *Ant* que:

1. Gera as classes e métodos em *JAVA* da estrutura de dados para implementar a árvore de sintaxe abstrata (AST).
2. Gera o conversor das regras em *ANTLR* para *GOM*.
3. Gera o *parser* utilizando o *ANTLR*.
4. Compila o programa *Tom*.
5. Compila o código *JAVA* gerado.

5.3 Aplicação Web

Por último, foi desenvolvida a aplicação Web onde todas as fases anteriores foram integradas. A tecnologia escolhida recaiu sobre o uso de *Servlets* juntamente com *JSP* pois consideramos tratar-se de uma tecnologia com futuro e que estamos a aprender de momento, e a acessibilidade de integração com as outras tecnologias que necessitamos de utilizar também foi considerada como um ponto a favor.

Toda a implementação foi desenvolvida conforme aquilo que tinha sido anteriormente modelado e tanto o administrador como o utilizador dispõem de todas as ações a que nos propusemos a integrar. De modo a tornar mais perceptível o trabalho de implementação desenvolvido, efetuaremos uma breve explicação sobre a arquitetura do projeto e sobre a função de avaliação criada e apresentaremos um manual de utilizador.

Para o nosso projeto, foi aplicado o padrão MVC (Model-View-Controller) em que os Models são Java Beans (classes com um construtor vazio e com os métodos *get* e *set* definidos), os Controllers são os Servlets criados consoante a necessidade de executar determinada tarefa e as Views são os ficheiros *.html* (estáticos) e *.jsp* (dinâmicos) que representam as diferentes páginas que podem ser visualizadas conforme necessário. É importante destacar o fato de termos criado um servlet para cada função que considerámos ser fundamental (usando, por diversas vezes, o conceito de *forwarding* e *chaining*).

Por fim, iremos descrever o processo que descreve as diferentes fases desde a submissão de um ficheiro até à visualização dos respetivos resultados (optámos por descrever este processo e mais nenhum outro pois concordámos que este processo é onde grande parte do trabalho desenvolvido anteriormente é aplicado); assim sendo, após o utilizador submeter o

ficheiro, os dados fornecidos são guardados na nossa base de dados e o ficheiro submetido é lido e guardado localmente. Após isto, é executado um forward para um novo *servlet* onde será invocado o TOM (isto é, o projecto desenvolvido nesta ferramenta) e é dado o caminho onde o ficheiro foi guardado, retornado os respectivos valores das características que nos propusemos a avaliar. Estas características são guardadas na base de dados e é executada a função avaliação sobre esses valores obtidos, retornado um valor entre um e cinco, consoante a qualidade do código avaliado (segundo os pesos atuais das métricas) e é atualizado o campo da avaliação na tabela onde foram guardados os dados do ficheiro submetido. Nesta mesma altura, é verificada a avaliação do código em questão e este é colocado no repositório dos bons ou maus programas (no primeiro, caso possua uma classificação de 4 ou 5 e no segundo, caso contrário). Por fim, é executado um redirect para um novo *servlet* que se limitará a buscar, à base de dados, os dados relativos ao ficheiro submetido e às suas características e colocá-los acessíveis para que a *view* criada os possa aceder e mostrá-los.

5.3.1 Função de Avaliação

Outra funcionalidade importante desenvolvida durante a implementação foi a função de avaliação que é responsável por receber as métricas calculadas, aquando da submissão do código, o conjunto atual dos pesos das métricas e os valores padrão atuais e devolver a classificação do código avaliado entre 1 e 5. O modo de avaliação pelo qual optámos foi de comparar os valores das métricas com o respetivo valor padrão mas, para além disto e de modo a conseguir contextualizar a diversidade de código que pode existir, decidimos também usar outras métricas que possam influenciar a métrica em questão (em alguns casos, o número de funções, noutros as linhas de código). Também decidimos que para obter uma classificação de 5 em qualquer métrica, o valor desta deve ser melhor que o valor padrão sendo que algo na ordem do valor padrão apenas garante uma classificação de 4 estrelas.

Por fim, após termos uma classficação (de 1 a 5) para cada métrica que pretendemos avaliar, aplicámos o respetivo peso dessa métrica, definido pelo administrador, obtendo a classificação final do código submetido segundo todas as métricas.

```
private int avaliacao(ResultSet result, int nComentarios, float pesoComent, int
    nLinhas, float pesoLinhas, int nFuncoes, float pesoFuncoes, int nErros, float
    pesoErros, int ms, float pesoTempo, int nCCMcCabe, float pesoCCMcCabe, int
    nHalstead, float pesoHalstead) throws SQLException {

    float nComentariosVP = 0, nLinhasVP = 0, nFuncoesVP = 0, nErrosVP = 0, nTempoVP =
    0, nCCMcCabeVP = 0, nHalsteadVP = 0;
    //Recebe o conjunto de valores padrao e retira o respetivo valor de cada metrica
    while(result.next()){
      nComentariosVP = result.getFloat(1);
      nLinhasVP = result.getFloat(2);
      nFuncoesVP = result.getFloat(3);
      nErrosVP = result.getFloat(4);
      nTempoVP = result.getFloat(5);
      nCCMcCabeVP = result.getFloat(6);
      nHalsteadVP = result.getFloat(7);
    }

    //Comentarios
    int estrelasComentarios = 0;
    if (nComentarios > nComentariosVP + nFuncoes) estrelasComentarios = 5;
    else if ((nComentarios >= nComentariosVP - nFuncoes) && (nComentarios <=
    nComentariosVP + nFuncoes)) estrelasComentarios = 4;
    else if (nComentarios > nComentariosVP - nFuncoes * 2) estrelasComentarios = 3;
    else if (nComentarios > nComentariosVP - nFuncoes * 3) estrelasComentarios = 2;
    else estrelasComentarios = 1;

    //Linhas
    int estrelasLinhas = 0;
    if (nLinhas < nLinhasVP - nFuncoes) estrelasLinhas = 5;
```

```
    else if ((nLinhas <= nLinhasVP + nFuncoes) && (nLinhas >= nLinhasVP - nFuncoes))
    estrelasLinhas = 4;
    else if (nLinhas < nLinhasVP + nFuncoes * 2) estrelasLinhas = 3;
    else if (nLinhas < nLinhasVP + nFuncoes * 4) estrelasLinhas = 2;
    else estrelasLinhas = 1;

    //Funcoes
    int estrelasFuncoes = 0;
    if (nFuncoes < nFuncoesVP + nLinhas / 45) estrelasFuncoes = 1;
    else if (nFuncoes < nFuncoesVP + nLinhas / 30) estrelasFuncoes = 2;
    else if (nFuncoes < nFuncoesVP + nLinhas / 20) estrelasFuncoes = 3;
    else if (nFuncoes < nFuncoesVP + nLinhas / 15) estrelasFuncoes = 4;
    else estrelasFuncoes = 5;

    //Erros
    int estrelasErros = 0;
    if (nErros == 0) estrelasErros = 5;
    else estrelasErros = 1;

    //Tempo
    int estrelasTempo = 0;
    if (ms < nTempoVP - nLinhas * 0.5) estrelasTempo = 5;
    else if ((ms >= nTempoVP - nLinhas * 0.5) && (ms <= nTempoVP + nLinhas * 0.5))
    estrelasTempo = 4;
    else if (ms < nTempoVP + nLinhas) estrelasTempo = 3;
    else if (ms < nTempoVP + nLinhas * 1.5) estrelasTempo = 2;
    else estrelasTempo = 1;

    //CCMcCabe
    int estrelasCCMcCabe = 0;
    if (nCCMcCabe > nCCMcCabeVP + nFuncoes * 8) estrelasCCMcCabe = 1;
    else if (nCCMcCabe > nCCMcCabeVP + nFuncoes * 4) estrelasCCMcCabe = 2;
    else if (nCCMcCabe > nCCMcCabeVP + nFuncoes * 2) estrelasCCMcCabe = 3;
    else if (nCCMcCabe > nCCMcCabeVP) estrelasCCMcCabe = 4;
    else estrelasCCMcCabe = 5;

    //Halstead
    int estrelasHalstead = 0;
    if (nHalstead > nHalsteadVP + nFuncoes * 5) estrelasHalstead = 1;
    else if (nHalstead > nHalsteadVP + nFuncoes * 2.5) estrelasHalstead = 2;
    else if (nHalstead > nHalsteadVP + nFuncoes) estrelasHalstead = 3;
    else if (nHalstead > nHalsteadVP) estrelasHalstead = 4;
    else estrelasHalstead = 5;

    float resultado = 0;
    resultado = estrelasComentarios * pesoComent + estrelasLinhas * pesoLinhas +
    estrelasFuncoes * pesoFuncoes + estrelasErros * pesoErros + estrelasTempo *
    pesoTempo + estrelasCCMcCabe * pesoCCMcCabe + estrelasHalstead * pesoHalstead;
    return Math.round(resultado);
}
```

5.3.2 Guia de Utilizador

Nesta secção irá ser mostrado um pequeno guia para a utilização da aplicação Web, demonstrando as diferentes capacidades do utilizador e do administrador. Assim, temos a página inicial onde o utilizador pode-se registar ou efetuar a autenticação caso já possua uma conta.

Figura 5.1: Registo de um utilizador

Figura 5.2: Autenticação

Caso a autenticação tenha sido feita por um utilizador comum, a imagem seguinte será a sua vista da aplicação. A ação mostrada é a da submissão de código, podendo ainda obter uma lista de todas as suas submissões ou efetuar logout.

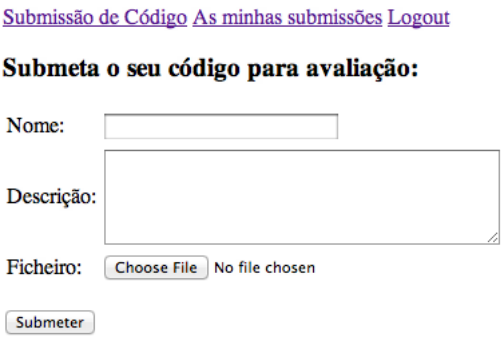

Figura 5.3: Vista inicial do utilizador - Submissão de Ficheiro

Uma pequena referência para o facto de, na imagem seguinte, o utilizador possuir a habilidade de aceder aos dados que a nossa aplicação retirou do ficheiro submetido, descarregar o ficheiro ou, ainda, aceder a uma análise detalhada do ficheiro.

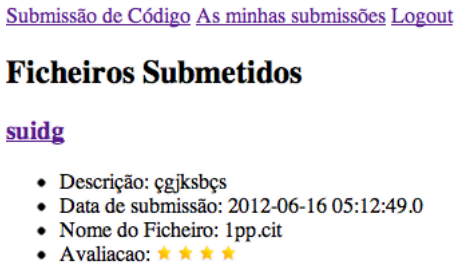

Figura 5.4: Lista das submissões efetuadas

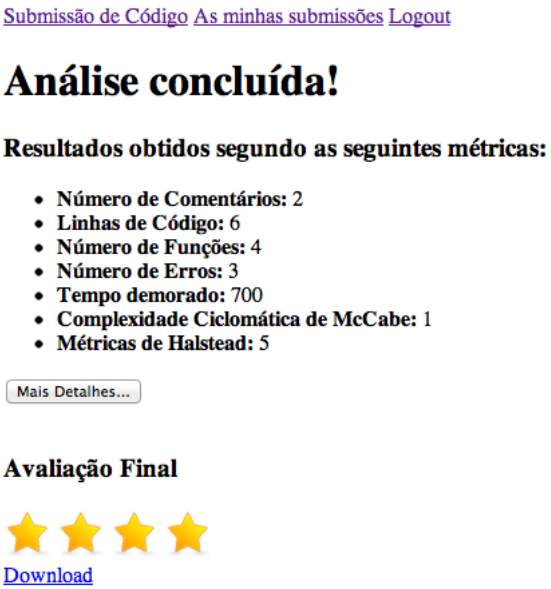

Figura 5.5: Classificação do ficheiro escolhido

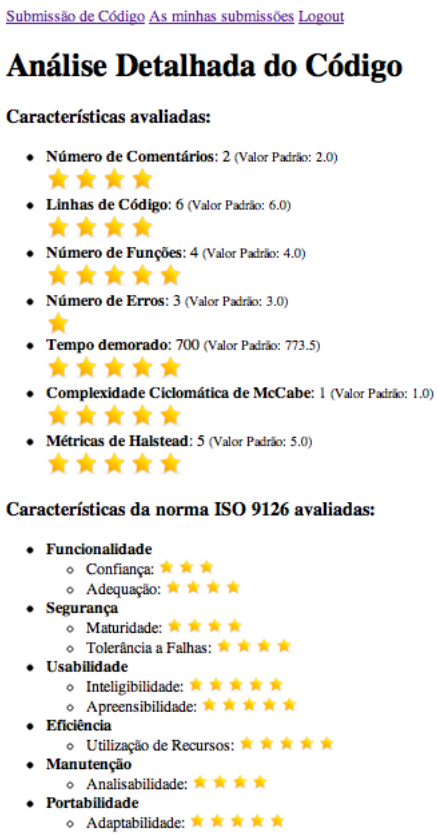

Figura 5.6: Análise detalhada do ficheiro escolhido

Por outro lado, caso a autenticação tenha sido feita por um administrador, ele é redirecionado para a mesma tarefa mas possui muitas mais opções disponíveis. De seguida será mostrado as diferentes vistas consoante a opção escolhida, entre as quais: listagem de todas as submissões, listagem de todos os pesos já criados para as métricas definidas, criação de um novo conjunto de pesos das métricas, conjunto de programas considerado bons ou maus (consoante a opção escolhida) e visualizar todos os ficheiros que pertencem ao repositório base. O administrador poderá ainda efetuar logout.

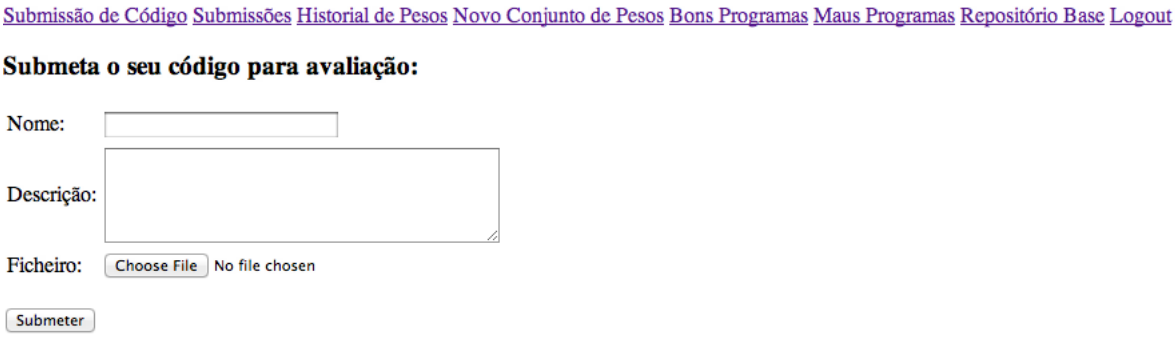

Figura 5.7: Vista inicial do administrador - Submissão de Ficheiro

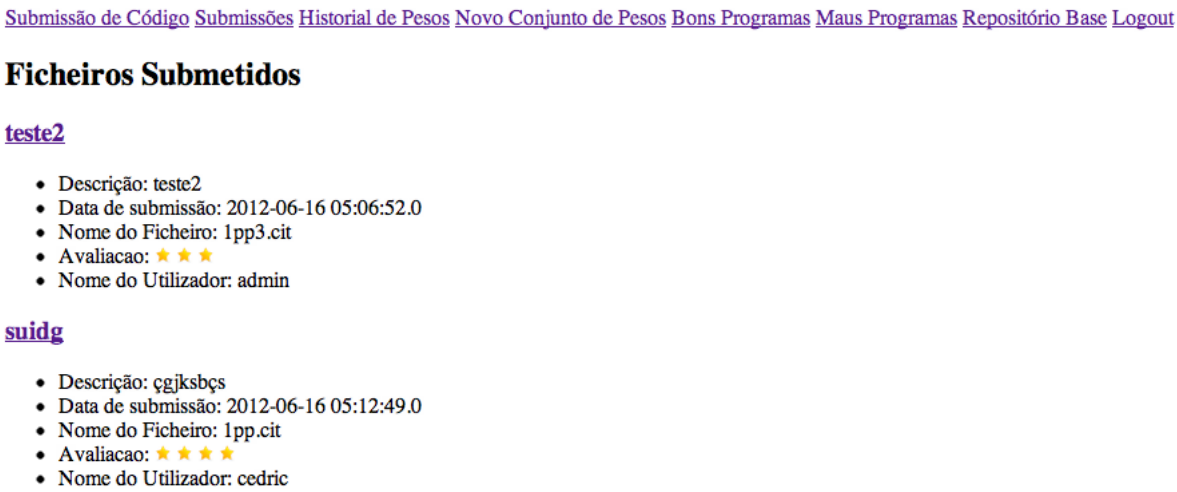

Figura 5.8: Lista de todas as submissões

Tal como acima, após listar todas as submissões, o administrador pode ver a classificação de cada submissão, efetuar o seu descarregamento ou aceder à análise detalhada.

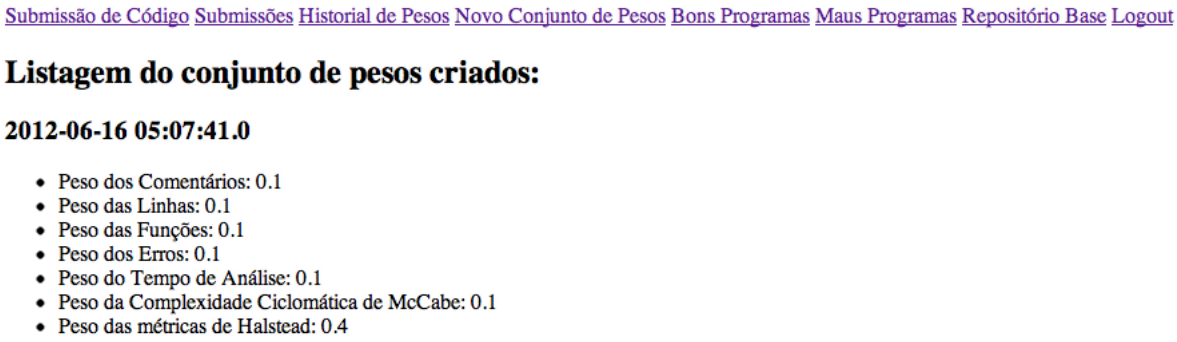

Figura 5.9: Lista de todas os conjuntos de pesos que já foram definidos

Importante destacar nesta lista o fato de a lista apresentar-se ordenada segundo a data de submissão (da mais recente para mais antiga), ou seja, o primeiro conjunto de pesos é aquele que é aplicado de momento.

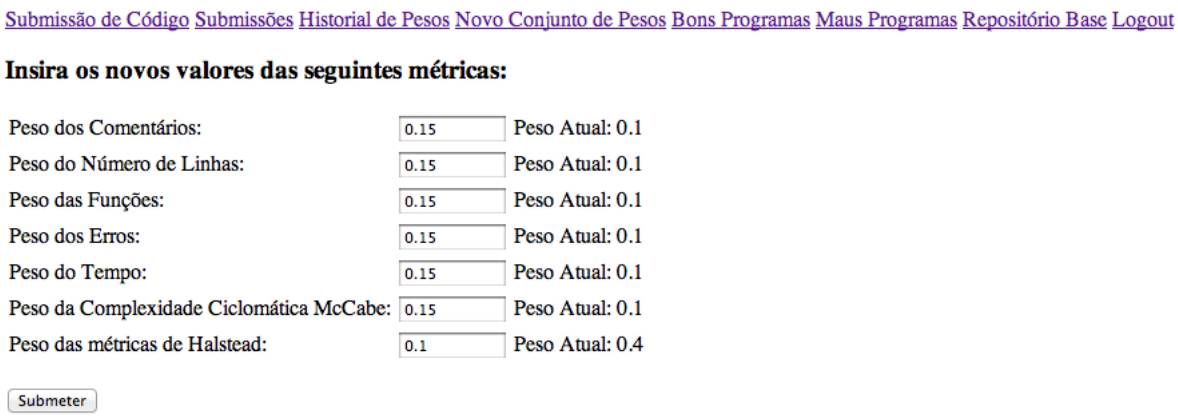

Figura 5.10: Criar novo conjunto de pesos para as métricas

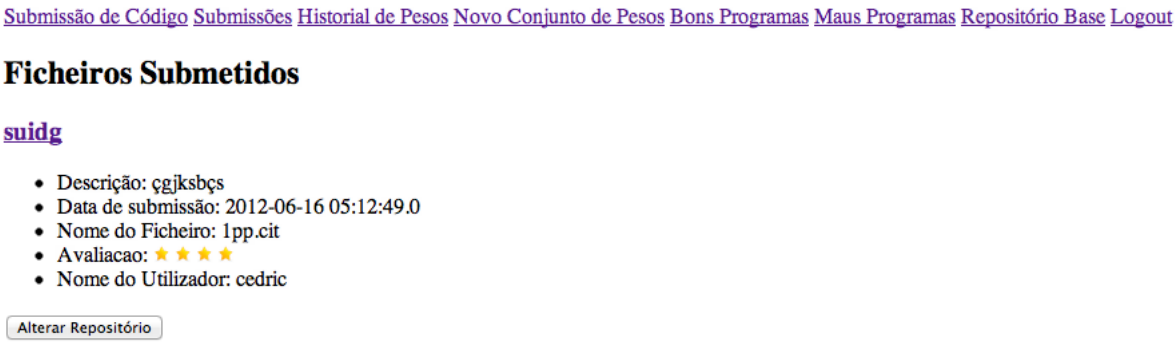

Figura 5.11: Lista de todos os programas classificados como bons

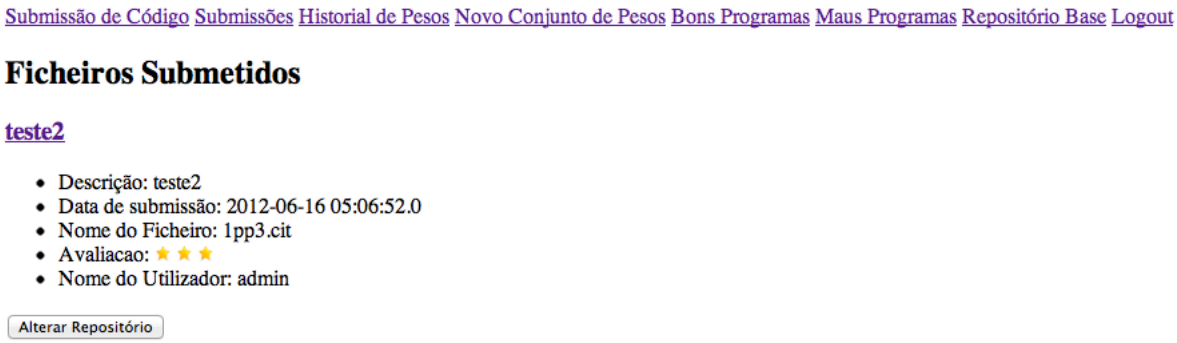

Figura 5.12: Lista de todos os programas classificados como maus

Tanto na lista de programas bons como de programas maus, o administrador tem a possibilidade de aceder aos detalhes da submissão (tal como foi explicado acima). Para além disto, o administrador também pode adicionar determinado programa ao repositório base ou alterar o repositório em que a submissão se encontra (passar a submissão para o repositório dos bons programas caso ela se encontre no outro repositório contrário, ou vice-versa).

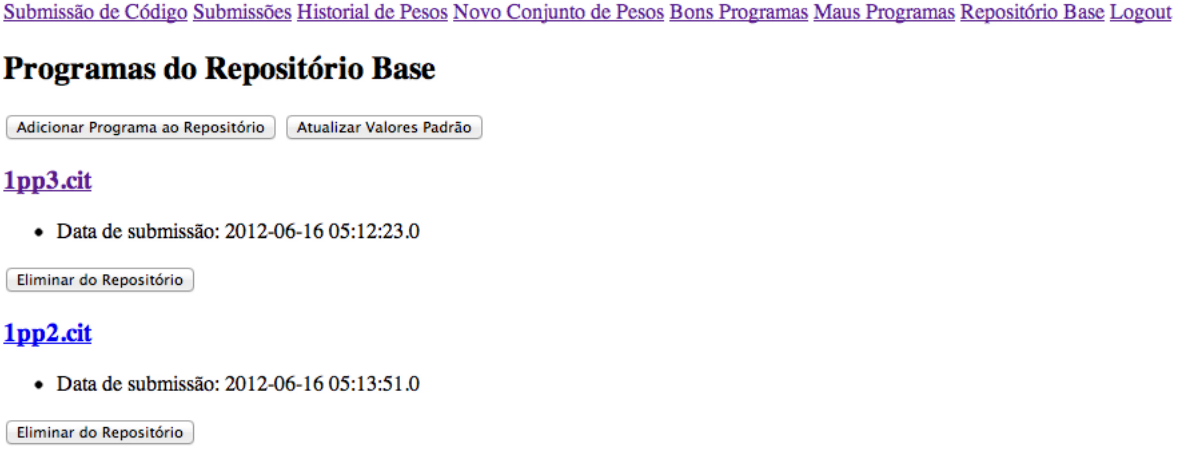

Figura 5.13: Lista de todos os programas do repositório base

Por fim, nesta lista, o administrador poderá remover determinado programa do repositório, adicionar um novo programa ao repositório ou, ainda, decidir que a aplicação deve correr todo o repositório base e actualizar os valores padrão para cada métrica.

Capítulo 6

Conclusão

Neste trabalho grande parte da fase inicial foi investida em analisar a linguagem *Tiger* e a sua sintaxe. Como referido, a linguagem *Tiger* foi criada como caso de estudo apresentado no livro **ModernCompilerImplementationInML**, sendo uma linguagem relativamente simples e sem grande complexidade mas que no entanto, causou a necessidade de uma análise mais demorada, pois tratava-se uma linguagem desconhecida.

Após a análise da linguagem *Tiger*, passámos a uma análise das métricas que se podiam aplicar nesta, assim como a ligação que se poderia fazer com as normas de qualidade **ISO 9126**. Esta análise encontra-se completa, considerando que desprezamos alguns parâmetros que achamos menos importantes, dando maior importância aqueles que escolhemos para representar no programa de avaliação; para estas escolhas, pesou muito a nossa falta de experiência no que toca à manipulação de programas *Tiger*, ainda que tenha sido desenvolvida (a experiência) ao longo do projeto, e o fato de nunca termos realizado análises aprofundadas sobre as métricas a ter em consideração aquando da escrita de código.

Após a visualização dos resultados obtidos, somos da opinião que a aplicação Web desenvolvida se encontra num nível bastante satisfatório já que todos os objetivos a que nos propusemos foram alcançados com sucesso. Após os mais variados testes efetuados sobre o portal, os resultados retornados foram os esperados e as capacidades do administrador encontra-se bem implementadas (a criação tanto de um repositório dos programas bem classificados como outro repositório para os piores, a manutenção do repositório base de programas *Tiger* e a possibilidade de manipular os pesos e os valores de referência das métricas) bem como as ações do utilizador regular (submissão de ficheiros para avaliação e listagem dos seus ficheiros submetidos).

Como trabalho futuro, somos da opinião que a aplicação Web desenvolvida poderia evoluir no que diz respeito à obtenção de dados estatísticos e à função utilizada na avaliação do código. Para além disto, com o desenvolver da nossa experiência, o mapeamento das métricas e das normas de qualidade **ISO 9126** utilizadas sofreriam, provavelmente, diversas alterações de modo a adequarem-se àquilo que nos era proposto.

Bibliografia

[1] Prof. Stephen A. Edwards *Programming Languages and Translators Programming Assignment 1: Scanner, Parser, and AST.*
Columbia University, 2002.

[2] Z. Benjamin,V. Diana,S. Ina ,N. Helmut Neukirchen,G Jens .: *Applying the ISO 9126 Quality Model to Test Specifications.*

[3] Andrew W. Appel.: *Modern Compiler Implementation in ML.* .
Cambridge University Press, 2004

[4] Kaner C., Bond W.: *Software Engineering Metrics: What Do They Measure and How Do We Know?.*
International Software metrics Symposium, 2004

Capítulo 7

Anexos

.1 Linguagem de Domínio Específico

```
grammar Tiger;

options {
  k=2;
}

tokens {
        ARRAYOF = 'array of' ;
        IF = 'if' ;
        THEN = 'then' ;
        ELSE = 'else' ;
        WHILE = 'while' ;
        FOR = 'for' ;
        TO = 'to' ;
        DO = 'do' ;
        LET = 'let' ;
        IN = 'in' ;
        END = 'end' ;
        OF = 'of' ;
        BREAK = 'break' ;
        NIL = 'nil' ;
        FUNCTION = 'function' ;
        VAR = 'var' ;
        TYPE = 'type' ;
        IMPORT = 'import' ;
        PRIMITIVE = 'primitive' ;
        IGUAL = '=' ;
        MAIS = '+' ;
        MULT = '*' ;
        DIV = '/' ;
        AND = '&' ;
        OR = '|' ;
        DPIGUAL = ':=' ;
        AP = '(' ;
        FP = ')' ;
        AC = '{' ;
        FC = '}' ;
        APR = '[' ;
        FPR = ']' ;
        DP = ':' ;
        VIR = ',' ;
        PVIR = ';' ;
        PONTO = '.' ;
}

prog            : exp ;
exp             : expOR expORPr ;
expOR           : expAND expANDPr ;
expAND          : aritExp relExp ;
```

```
expORPr         : OR exp
                | //vazio
                ;

expANDPr        : AND exp
                | // vazio
                ;

aritExp         : term termPr ;
relExp          : RELOP aritExp
                | //vazio
                ;

term            : factor factorPr ;
termPr          : (MAIS|NEG|IGUAL) term termPr
                | //vazio
                ;

factorPr        : (MULT|DIV) factor factorPr
                | //vazio
                ;

factor          : NIL
                | INT
                | STRING
                | AP expList FP
                | NEG exp
                | IF exp THEN exp (ELSE exp)?
                | WHILE exp DO exp
                | FOR ID DPIGUAL exp TO exp DO exp
                | BREAK
                | LET decList IN expList END
                | lValue
                ;

decList         : dec* ;
dec             : tyDec
                | varDec
                | funDec
                ;

tyDec           : TYPE typeId IGUAL ty ;
ty              : AC fieldList FC
                | ARRAYOF typeId
                | typeId
                ;

fieldList       : (ID (DP|IGUAL) typeId (intFieldList)*)? ;

intFieldList    : VIR ID (DP|IGUAL) typeId
                | APR exp FPR (OF exp)?
                ;

typeId          : ID
                | STRING
                | INT
                | NIL
                ;

varDec          : VAR ID (DP typeId)? (DPIGUAL|IGUAL) exp ;
funDec          : FUNCTION ID AP fieldList FP (DP typeId)? IGUAL exp ;
lValue          : ID (functionRecordArray);

functionRecordArray
                : AP argList FP
                | AC fieldList FC
                | (APR exp FPR)? (OF exp|(PONTO ID |APR exp FPR)* (DPIGUAL exp))?
                ;

expList         : (exp (PVIR exp)*)? ;
argList         : (exp (VIR exp)*)? ;
```

```
RELOP           : '<'|'>'|'<>'|'<='|'>=' ;

NEG             : '-' ;

ID              : (LETTER (LETTER|DIGIT|'_')*|'_main') ;
INT             : DIGIT+ ;
STRING          : '"'~('"')*'"' ;
fragment DIGIT  : '0'..'9' ;
fragment LETTER
                : ('a'..'z'|'A'..'Z') ;
NESTED_ML_COMMENT
    :   '/*'
        (options { greedy=false; } : (NESTED_ML_COMMENT|.))*
        '*/' { $channel=HIDDEN; }
    ;
WHITESPACE      : ('\t'|' '|'\r'|'\n'|'\u000C')+ { $channel = HIDDEN; } ;
```

.2 Árvore de Sintaxe Abstrata (AST)

```
module parser.Rec
        imports int String
        abstract syntax
        Prog = Prog(e1:Exp)
        Exp = Exp(eop:ExpOR, eorp:ExpORPr)
        ExpOR = ExpOR(ea:ExpAND, eap:ExpANDPr)
        ExpAND = ExpAND(ae:AritExp, re:RelExp)
        ExpORPr = ExpORPr(e1:Exp)
                        | ExpORPrEmpty()
        ExpANDPr = ExpANDPr(e1:Exp)
                        | ExpANDPrEmpty()
        AritExp = AritExp(t:Term, tp:TermPr)
        RelExp = RelExpLess(a1:AritExp)
               | RelExpMore(a2:AritExp)
               | RelExpDiferent(a3:AritExp)
           | RelExpLessEqual(a4:AritExp)
                    | RelExpMoreEqual(a5:AritExp)
                    | RelExpEmpty()
        Term = Term(f:Factor, fp:FactorPr)
        TermPr = TermPrPlus(t:Term, tp:TermPr)
                    | TermPrNeg(t:Term, tp:TermPr)
                    | TermPrEqual(t:Term, tp:TermPr)
                    | TermPrEmpty()
        FactorPr = Times(f:Factor, fp:FactorPr)
                         | Div(f:Factor, fp:FactorPr)
                         | FactorPrEmpty()
        Factor = FNil()
                    | FInteger(i:int)
                    | FString(s:String)
                    | FExpList(el:ExpList)
                    | FNeg(e1:Exp)
                    | FWhile(e1:Exp, e2:Exp)
                    | FFor(id:String, e1:Exp, e2:Exp, e3:Exp)
                    | FBreak()
                    | FLet(dl:DecList, el:ExpList)
                    | FLvalue(lv:LValue)
                    | FIfThen(e1:Exp, e2:Exp)
                    | FIfThenElse(e1:Exp, e2:Exp, e3:Exp)
    DecList = DecList(Dec*)
        Dec = DTyDec(td:TyDec)
                | DVarDec(vd:VarDec)
                | DFunDec(fd:FunDec)
        TyDec = TyDec(ti:TypeId, t:Ty)
        Ty = TFieldList(fl:FieldList)
           | TArrayOf(ti:TypeId)
           | TTypeId(ti:TypeId)
        FieldList = FieldListDP(d:String, ti:TypeId, a:AuxFieldList)
                  | FieldListEqual(id:String, ti:TypeId, a:AuxFieldList)
                  | FieldListEmpty()
```

```
        AuxFieldList = AuxFieldList(IntFieldList*)
        IntFieldList = IntFieldList1(id:String, a:Aux1)
                     | IntFieldList2(e1:Exp)
                       | IntFieldList3(e1:Exp, e2:Exp)
        Aux1 = Aux11(ti:TypeId)
             | Aux12(ti:TypeId)
        TypeId = TypeId(id:String)
               | TypeIdString(s:String)
           | TypeIdInteger(i:int)
           | TypeIdNil()
        VarDec = VarDec1(id:String, a:Aux2)
           | VarDec2(id:String, ti:TypeId, a:Aux2)
        Aux2 = Aux21(e:Exp)
             | Aux22(e:Exp)
        FunDec = FunDec1(id:String, fl:FieldList, e1:Exp)
               | FunDec2(id:String, fl:FieldList, ti:TypeId, e1:Exp)
        LValue = LValue(id:String, fra:FunctionRecordArray)
        FunctionRecordArray = FRAArgList(al:ArgList)
                                         | FRAFieldList(fl:FieldList)
                                         | FRA1(a1:AuxFRA1)
                                         | FRA2(a2:AuxFRA2)
                                         | FRA3(a1:AuxFRA1, a2:AuxFRA2)
                                         | FunctionRecordArrayEmpty()
        AuxFRA1 = AuxFRA1(e1:Exp)
        AuxFRA2 = AuxFRA21(e1:Exp)
             | AuxFRA22(a3:Aux3, e2:Exp)
             | AuxFRA23(a3:Aux3, a4:Aux4, e2:Exp)
                     | AuxFRA24(a4:Aux4, e2:Exp)
        Aux3 = Aux3(String*)
        Aux4 = Aux4(Exp*)
        ExpList = ExpList(e1:Exp, a:AuxExpList)
                     | ExpListEmpty()
        AuxExpList = AuxExpList(Exp*)
        ArgList = ArgList(e1:Exp, a:AuxArgList)
                     | ArgListEmpty()
        AuxArgList = AuxArgList(Exp*)
```

.3 Parsing

```
grammar Rec;
options {
  output=AST;
  ASTLabelType=Tree;
  tokenVocab=RecTokens;
  k=2;
}

@header {
  package parser;
}
@lexer::header {
  package parser;
}
prog            : exp   -> ^(Prog exp)
                        ;
exp                     : e1=expOR e2=expORPr   -> ^(Exp $e1 $e2)
                        ;
expOR           : e1=expAND e2=expANDPr         -> ^(ExpOR $e1 $e2)
                        ;
expAND          : aritExp relExp        -> ^(ExpAND aritExp relExp)
                        ;
expORPr         : '|' exp               -> ^(ExpORPr exp)
                        |                                       -> ^(ExpORPrEmpty)
                        ;
expANDPr        : '&' exp                       -> ^(ExpANDPr exp)
                        |                                       -> ^(ExpANDPrEmpty)
                        ;
aritExp         : term termPr           -> ^(AritExp term termPr)
```

```
                    ;
relExp          : '<' aritExp           -> ^(RelExpLess aritExp)
                    | '>' aritExp           -> ^(RelExpMore aritExp)
                    | '<>' aritExp          -> ^(RelExpDiferent aritExp)
                    | '<=' aritExp          -> ^(RelExpLessEqual aritExp)
                    | '>=' aritExp          -> ^(RelExpMoreEqual aritExp)
                    |                                                   -> ^(
    RelExpEmpty)
                    ;
term            : factor factorPr -> ^(Term factor factorPr)
                    ;
termPr          : '+' term termPr       -> ^(TermPrPlus term termPr)
                    | '-' term termPr       -> ^(TermPrNeg term termPr)
                    | '=' term termPr       -> ^(TermPrEqual term termPr)
                    |                                       -> ^(TermPrEmpty)
                    ;
factorPr        : '*' factor factorPr   -> ^(Times factor factorPr)
                    | '/' factor factorPr   -> ^(Div factor factorPr)
                    |                                                   -> ^(
    FactorPrEmpty)
                    ;
factor          : 'nil'
                                -> ^(FNil)
                    | INT
                                    -> ^(FInteger INT)
                    | STRING
                                    -> ^(FString STRING)
                    | '(' expList ')'
                                -> ^(FExpList expList)
                    | '-' exp
                                    -> ^(FNeg exp)
                    | 'while' e1=exp 'do' e2=exp
                  -> ^(FWhile $e1 $e2)
                    | 'for' ID ':=' e1=exp 'to' e2=exp 'do' e3=exp          -> ^(
    FFor ID $e1 $e2 $e3)
                    | 'break'
                                    -> ^(FBreak)
                    | 'let' decList 'in' expList 'end'
                  -> ^(FLet decList expList)
                    | lv=lValue
                                    -> ^(FLvalue $lv)
                    | ('if' e1=exp 'then' e2=exp ('else' e3=exp)?           -> {e3
    ==null}? ^(FIfThen $e1 $e2)

     ^(FIfThenElse $e1 $e2 $e3))
                    ;
decList         : dec*                          -> ^(DecList dec*)
                    ;
dec                 : tyDec                         -> ^(DTyDec tyDec)
                    | varDec                        -> ^(DVarDec varDec)
                    | funDec                        -> ^(DFunDec funDec)
                    ;
tyDec           : 'type' typeId '=' ty  -> ^(TyDec typeId ty)
                    ;
ty                  : '{' fieldList '}'             -> ^(TFieldList fieldList)
                    | 'array of' ti=typeId          -> ^(TArrayOf $ti)
                    | ti=typeId                             -> ^(TTypeId $ti)
                    ;
fieldList       : ID ':' ti=typeId auxFieldList -> ^(FieldListDP ID $ti auxFieldList)
                    | ID '=' ti=typeId auxFieldList -> ^(FieldListEqual ID $ti
    auxFieldList)
                    |
            -> ^(FieldListEmpty)
                    ;
auxFieldList: (intFieldList)*   -> ^(AuxFieldList intFieldList*)
                    ;
intFieldList
                    : ',' ID aux1                                               -> ^(
    IntFieldList1 ID aux1)
                    | ('[' e1=exp ']' ('of' e2=exp)?        -> {e2==null}? ^(
    IntFieldList2 $e1)
```

```
                                                                                                                    ->
    ^(IntFieldList3 $e1 $e2))
                    ;
aux1            : ':' ti=typeId         -> ^(Aux11 $ti)
                | '=' ti=typeId         -> ^(Aux12 $ti)
                ;
typeId          : ID                    -> ^(TypeId ID)
                | STRING                -> ^(TypeIdString STRING)
                | INT                   -> ^(TypeIdInteger INT)
                | 'nil'                 -> ^(TypeIdNil)
                ;
varDec          : ('var' ID (':' ti=typeId)? aux2       -> {ti==null}? ^(VarDec1 ID
    aux2)
                                                                                                                    ->
    ^(VarDec2 ID $ti aux2))
                ;
aux2            : ':=' exp      -> ^(Aux21 exp)
                | '=' exp               -> ^(Aux22 exp)
                ;
funDec          : 'function' ID '(' fieldList ')' (':' ti=typeId)? '=' exp      -> {ti
    ==null}? ^(FunDec1 ID fieldList exp)

    ^(FunDec2 ID fieldList $ti exp)
                ;
lValue          : ID fra=functionRecordArray    -> ^(LValue ID $fra)
                ;

functionRecordArray
                : '(' argList ')'                       -> ^(FRAArgList
    argList)
                | '{' fieldList '}'                     -> ^(FRAFieldList
    fieldList)
                | ((a1=auxFRA1)? (a2=auxFRA2)?  -> {a1!=null && a2==null}? ^(
    FRA1 $a1)
                                                                                                                    ->
     {a1==null && a2!=null}? ^(FRA2 $a2)
                                                                                                                    ->
     {a1!=null && a2!=null}? ^(FRA3 $a1 $a2)
                                                                                                                    ->
     ^(FunctionRecordArrayEmpty))
                ;
auxFRA1         : '[' exp ']'           -> ^(AuxFRA1 exp)
                ;
auxFRA2         : 'of' exp                                                      -> ^(
    AuxFRA21 exp)
                | (a3=aux3 (a4=aux4)? ':=' e2=exp       -> {a4==null}? ^(
    AuxFRA22 $a3 $e2)
                                                                                                                    ->
     ^(AuxFRA23 $a3 $a4 $e2))
                | a4=aux4       ':=' e2=exp                                     -> ^(
    AuxFRA24 $a4 $e2)
                ;
aux3            : ('.' ID)*                     -> ^(Aux3 ID*)
                ;
aux4            : ('[' exp ']')*        -> ^(Aux4 exp*)
                ;
expList         : exp auxExpList        -> ^(ExpList exp auxExpList)
                |                                       -> ^(ExpListEmpty)
                ;
auxExpList  : (';' exp)*                -> ^(AuxExpList exp*)
                ;
argList         : exp auxArgList        -> ^(ArgList exp auxArgList)
                |                                               -> ^(
    ArgListEmpty)
                ;
auxArgList      : (',' exp)*    -> ^(AuxArgList exp*)
                ;

ID                      : (LETTER (LETTER|DIGIT|'_')*|'_main') ;
INT                     : DIGIT+ ;
STRING          : '"'~('"')*'"' ;
fragment DIGIT
```

```
                        : '0'..'9' ;
fragment LETTER
                        : ('a'..'z'|'A'..'Z') ;
NESTED_ML_COMMENT
    :   '/*'
        (options {greedy=false;} : (NESTED_ML_COMMENT | .))*
        '*/' {$channel = HIDDEN;}
    ;
WHITESPACE      : ('\t'|' '|'\r'|'\n' |'\u000C')+
        {
                $channel = HIDDEN;
        };
```