



Universidade do Minho
Departamento de Informática

Engenharia de Linguagens

Projecto Integrado

Software para Análise e Avaliação de Programas

Grupo 2:

José Pedro Silva	pg17628
Mário Ulisses Costa	pg15817
Pedro Faria	pg17684

Braga, 27 de Junho de 2011

Resumo

Este relatório explica com bastante detalhe o desenvolvimento de um sistema de submissão de projectos, trabalhos ou respostas a problemas num concurso de programação. Este sistema deve ser capaz de aceitar registos de alunos e professores. Deve ainda permitir a submissão de código por parte dos alunos e a submissão de exercícios por parte dos professores. O sistema deve ainda ser capaz de fazer uma análise detalhada sobre o código submetido pelo aluno e gerar um relatório para o professor ler sobre esse mesmo código.

Aqui está documentada, de forma incremental todo o processo de desenvolvimento deste sistema, desde a tomada de decisões sobre a arquitectura, modelação de dados até às decisões de implementação da ferramenta.

Índice

Resumo	i
Índice	iv
Índice de Figuras	v
Índice de Tabelas	vi
1 Introdução	1
1.1 Motivação	1
1.2 Objectivos	1
1.2.1 Ferramentas	2
1.3 Descrição do Sistema	2
1.3.1 Utilizadores	2
1.3.2 Funcionalidades do sistema	2
1.4 Estrutura do Relatório	3
I Modelação e Análise	5
2 Modelação do Problema	6
2.1 Modelação Informal	6
2.2 Modelação Formal	8
2.3 Modelo de Dados	11
2.4 Importação de dados	12
3 Métricas	18
3.1 Análise Estática	18
3.2 Análise Dinâmica	19
3.3 Métricas de qualidade de <i>software</i>	20
3.3.1 Bug patterns	21
3.3.2 Source Lines of code (SLOC)	21
3.3.3 Métricas de Segurança	22
3.3.4 Cyclomatic complexity	23
II WebApp e interface pelo terminal & Scripts de Instalação e Avaliação	24
4 Web Application	25
4.1 Criação de grupos, docentes e concorrentes	25
4.2 Linguagens de programação	27
4.3 Submissão de programas	28
4.4 Compilação	28
4.5 Execução	29
4.6 Guardar resultados	29

5	Interface pelo terminal	31
5.1	Perl	31
5.2	Milestone II	32
5.3	Milestone IV	33
6	Scripts de avaliação	34
6.1	Geração de imagens	34
6.2	Makefile	36
6.3	Cloning	37
7	Detecção de clones	39
7.1	Modo de utilização	39
7.2	Processamento do código	39
7.3	Output	40
7.4	Integração com a aplicação	40
8	Instalação	41
8.1	User check	41
8.2	Identificação da máquina	42
8.3	Instalação MacOSX	43
8.3.1	Bibliotecas C	43
8.3.2	Módulos Perl	43
8.3.3	Instalação de software essencial para a PartellI	43
8.3.4	Ruby, Rails e Gems	44
8.4	Instalação Ubuntu	44
8.4.1	Bibliotecas C	45
8.5	Fase actual	45
III	Parser e Aplicação das métricas	46
9	Strafunski	47
9.1	Explicação	47
9.1.1	Estratégias	48
9.2	Exemplos	49
9.2.1	McCabe Index—	49
9.2.2	Extracção das assinaturas das funções	50
10	Front-end	51
10.1	Estudo do Front-End	51
10.2	Bug no Language.C em MacOSX	54
10.3	Exemplos da árvore gerada pelo Language.C	56
11	Métricas Implementadas	58
11.1	Métricas calculadas	58
11.2	Tipo de dados das métricas	59
11.3	API de métricas	59
11.4	Exemplos de cálculo de métricas	60
12	Conclusão e Trabalho Futuro	62
	Bibliografia	62

A	AST da Linguagem C99	64
A.1	Gramática Lexica	64
A.1.1	Elementos Lexicos	64
A.1.2	Keywords	64
A.1.3	Identificadores	64
A.1.4	Universal character names	64
A.1.5	Constantes	65
A.1.6	String literals	66
A.1.7	Header names	66
A.2	Phrase structure grammar	66
A.2.1	Expressions	66
A.2.2	Declarations	67
A.2.3	Statements	69
A.2.4	External definitions	69
A.3	Preprocessing directives	70

Índice de Figuras

2.1	Arquitectura do sistema	7
2.2	Modelo de dados - Grupo e Docente/Admin	12
2.3	Modelo de dados - Concurso, tentativa e enunciado	13
2.4	diagrama do schema para o enunciado	16
2.5	diagrama do schema para a tentativa	17
4.1	Página de registo	26
4.2	Dados de um grupo	26
4.3	Comandos necessários para tornar um utilizador sem privilégios num docente.	27
4.4	Configuração de uma nova linguagem de programação no sistema	28
4.5	Página de submissão de programas (vista de administrador)	28
4.6	Página onde pode consultar as tentativas (vista das tentativas de todos os utilizadores)	30
5.1	Menu Principal	32
5.2	Linguagens disponíveis	32
5.3	Procura e alteração do user Zella Douglas	33
6.1	Linhas por Linguagem	35
6.2	Ficheiros por Linguagem	35
6.3	Linhas por Linguagem	36
6.4	Visão global do projecto	37
7.1	Listagem dos warnings	40

Índice de Tabelas

3.1 Avaliação da cyclomatic complexity	23
--	----

1

Introdução

Conteúdo

1.1	Motivação	1
1.2	Objectivos	1
1.2.1	Ferramentas	2
1.3	Descrição do Sistema	2
1.3.1	Utilizadores	2
1.3.2	Funcionalidades do sistema	2
1.4	Estrutura do Relatório	3

Este relatório descreve todo o projecto desenvolvido para o módulo Projecto Integrado da UCE de Engenharia Linguagens do Mestrado em Engenharia Informática da Universidade do Minho.

Pretende-se que este projecto seja um *Software* para Análise e Avaliação de Programas (SAAP), tendo este *software* como objectivo criar um ambiente de trabalho, com um interface Web, que permita a docentes/alunos avaliar/submeter programas automaticamente.

1.1 Motivação

Este trabalho é muito interessante do ponto de vista de produto final e como obra didáctica para a sua concretização. Para a sua feitura os autores terão de aplicar conhecimentos adquiridos na área de arquitectura de um sistema de informação, desenvolvimento para a web, linguagens de scripting, bases de dados, processamento de textos, etc. . .

Como se pode facilmente concluir, este é um projecto que do ponto de vista técnico é muito motivador devido à sua magnitude e inovação técnica que implica aos seus autores.

1.2 Objectivos

Este projecto tem como objectivos consolidar conhecimentos adquiridos nos diferentes módulos da UCE de Engenharia de Linguagens Assim sendo, vamos recorrer a tecnologia aprendida durante as aulas, bem como tecnologia já conhecida pelos autores, aprendida durante a formação académica ou à parte desta. Um dos grandes objectivos é consolidar e desenvolver ainda mais o uso de tecnologias variadas, para arranjar uma solução elegante, funcional e que cumpra os requisitos do sistema descrito. Assim, este projecto é mais do que um projecto de mestrado, passando a ser encarado como um desafio ao conhecimento e a pôr em prática conhecimentos adquiridos.

1.2.1 Ferramentas

As ferramentas/tecnologias que estamos a utilizar são as seguintes:

- DB2
- Perl
- Ruby on Rails (RoR)
- Haskell

Fazer-se-à uso de DB2 por suportar nativamente XML e ainda XPath e XQuery para fazer travessias no XML. O uso de Perl prende-se com o facto de ser incrivelmente fácil criar sem muito esforço pequenas ferramentas que acreditamos serem úteis para identificar padrões em texto ou cortar pequenos pedaços de texto.

O uso de RoR deve-se ao facto da simplicidade em criar ambientes web. Decidimos ainda utilizar Haskell eventualmente em tarefas mais complexas, por ser uma linguagem de rápida implementação e bastante segura.

1.3 Descrição do Sistema

O SAAP - Software para Análise e Avaliação de Programas é um sistema disponível através de uma interface web, que terá como principal função submeter, analisar e avaliar automaticamente programas. O sistema estará disponível em parte para utilizadores não registados, mas as suas principais funcionalidades estarão apenas disponíveis para docentes e grupos já registados no sistema.

O sistema poderá ser utilizado para várias finalidades, no entanto estará direccionado para ser utilizado em concursos de programação e em elementos de avaliação universitários.

1.3.1 Utilizadores

Existem três entidades que podem aceder ao sistema.

O administrador, que além de poder aceder às mesmas funcionalidades do docente, funcionalidades essas que já iremos descrever, é quem tem o poder de criar contas para os docentes.

O docente tem acesso a todo o tipo de funcionalidades relacionadas com a criação, edição e eliminação de concursos e enunciados, assim como consulta de resultados dos concursos e geração/consulta de métricas para os ficheiros submetidos no sistema.

O grupo, que pode ser constituído por um ou mais concorrentes, terá acesso aos concursos disponíveis, poderá tentar registar-se nos mesmos, e submeter tentativas de resposta para cada um dos seus enunciados.

Além do que já foi referido, todos os utilizadores podem editar os dados da sua conta.

Falta referir que um utilizador não registado (guest), pode criar uma conta para o seu grupo, de modo a poder entrar no sistema.

1.3.2 Funcionalidades do sistema

As várias funcionalidades do sistema já foram praticamente todas mencionadas, vamos no entanto tentar explicar a sua maioria, com um maior nível de detalhe.

Criação de contas de grupo e de docente

Criação de conta de grupo: Qualquer utilizador não registado poderá criar uma conta no sistema para o seu grupo, através da página principal do sistema. Terá de preencher dados referentes ao grupo e aos respectivos concorrentes.

Criação de conta de docente: Como já foi referido, o administrador terá acesso a uma página onde poderá criar contas para docentes.

Criação de concursos e enunciados

Tanto o administrador como o docente podem criar concursos. Depois de preencher todos os dados do concurso podem passar à criação de enunciados. Nesta altura caso prefiram, podem importar enunciados previamente criados em xml. O concurso só ficará disponível na data definida aquando da criação do concurso.

Registo e participação nos concursos

Um grupo que esteja autenticado no sistema pode tentar registar-se num dos concursos disponíveis, e será bem sucedido se a chave que utilizar for a correcta.

Depois de se registar no concurso, o tempo para a participação no mesmo inicia a contagem decrescente e o grupo poderá começar a submeter tentativas para cada um dos enunciados. O sistema informará, pouco depois da submissão, se a resposta estaria correcta ou não. Depois do tempo esgotar o grupo não pode submeter mais tentativas.

Geração das métricas para os ficheiros submetidos

O docente ou o administrador a qualquer altura podem pedir ao sistema que gere as métricas para as tentativas submetidas.

Consulta dos ficheiros com as métricas e dos resultados do concurso

O docente pode aceder aos logs que contêm a informação sobre as tentativas submetidas, ou se desejar, apenas aceder a informações mais específicas, tal como qual exercício tentou submeter determinado grupo, e se teve sucesso ou não.

Pode também visualizar os ficheiros com as informações das métricas.

1.4 Estrutura do Relatório

Este documento está provisoriamente estruturado em duas partes referentes a cada uma das Milestones até à data alcançadas. Dentro de cada parte estará descrito de forma sucinta todo o trabalho realizado separado logicamente por secções.

Capítulo 1º O leitor é introduzido ao tema abordado, e é justificada a utilidade do projecto.

Parte I Nesta parte apresentamos a modelação do problema e a análise que foi feita para apresentar esta solução

Capítulo 2º Neste capítulo é explicada a modelação e todas as decisões tomadas para a modelação do sistema a que nos propomos a concretizar.

Capítulo 3º São discutidas os vários tipos de métricas que existem e quais as que interessam para este trabalho.

Parte II Nesta parte mostramos o interface web e pelo terminal que fizemos e ainda alguns scripts que foram desenvolvidos

Capítulo 4º É dada especial atenção ao sistema front-end Web que foi desenvolvido.

Capítulo 5º Descreve uma implementação de um sistema front-end pelo terminal que tem como finalidade ser usado por administradores.

Capítulo 6º São mostrados e explicados a criação e uso de alguns scripts desenvolvidos para fazerem pequenas tarefas para a aplicação.

Capítulo 7º É explicado o script de detecção de clones mais a fundo.

Capítulo 8º É explicado o script que foi criado para instalação do sistema.

Parte III Nesta parte explicamos o front-end C que usamos e as estratégias que usamos para extrair informação dos programas C

Capítulo 9º É explicado a livreria Strafunski e o que esta permite, ainda é mostrado alguns exemplos de como a usamos.

Capítulo 10º É explicado o sistema front-end C feito em Haskell que foi utilizado para avaliar o código submetido.

Capítulo 11º São explicadas as métricas implementadas, a API de métricas que foi criada e são mostradas alguns exemplos de cálculo de métricas.

Capítulo 12º No capítulo final são tecidos alguns comentários relativos ao trabalho efectuado e motivação para trabalho futuro.

Parte I

Modelação e Análise

2

Modelação do Problema

Conteúdo

2.1 Modelação Informal	6
2.2 Modelação Formal	8
2.3 Modelo de Dados	11
2.4 Importação de dados	12

Neste capítulo, pretende-se dar uma descrição textual da várias modelações que foram feitas ao sistema, mais precisamente, a modelação informal, a modelação formal, o modelo de dados e, por fim, a importação de dados.

2.1 Modelação Informal

Com o diagrama da arquitectura do sistema, figura 2.1, pretende-se mostrar as várias entidades que podem aceder ao sistema, assim como as várias actividades que cada uma pode realizar e tarefas para o sistema processar. Também é realçada a ideia de que alguns dos recursos do sistema só estão disponíveis ao utilizador depois de passar por outros passos, ou seja, o diagrama dá a entender a ordem pelas quais o utilizador e o sistema podem/devem executar as tarefas.

Para começar, como temos duas entidades diferentes que podem aceder ao sistema (o docente e o aluno/concorrente), dividiu-se o diagrama em duas partes distintas (uma para cada entidade referida), de modo a facilitar a leitura.

Em ambos os casos, o login é a primeira actividade que pode ser realizada. Caso o login não seja efectuado com sucesso, é adicionado ao log file uma entrada com a descrição do erro. Caso contrário, e o login ser efectuado com sucesso, consoante as permissões do utilizador em questão, tem diferentes opções ao seu dispôr.

No caso do login pertencer a um docente, este terá acesso aos dados de cada um dos grupos, podendo verificar os resultados que estes obtiveram na resolução das questões do(s) concurso(s), assim como ao ficheiro que contém a análise das métricas dos vários programas submetidos pelos mesmos. Poderá também criar novos concursos e os seus respectivos exercícios, assim como adicionar baterias de testes para os novos exercícios, ou para exercícios já existentes.

No caso do login pertencer a um aluno/concorrente, o utilizador terá a opção de se registar num concurso ou de seleccionar um no qual já esteja registado. Já depois de seleccionar o concurso, pode



ainda escolher o exercício que pretende submeter. Depois de submeter o código fonte do programa correspondente ao exercício escolhido, e já sem a interação do utilizador, o sistema compilará e tentará executar os diferentes inputs da bateria de testes do exercício, e comparar os resultados obtidos com os esperados. No fim de cada um destes procedimentos, serão guardados os resultados / erros. Para terminar, será feito um estudo das métricas do ficheiro submetido, tendo como resultado a criação um ficheiro com os dados relativos a essa avaliação.

2.2 Modelação Formal

Afim de haver algum rigor na definição do sistema decidiu-se fazer um modelo mais formal do ponto de vista dos dados e das funcionalidades que o sistema apresenta. A ideia desta modelação é ainda ser um modelo formal da especificação atrás descrita. Para isso utilizou-se uma notação orientada aos contratos (design by contract), com a riqueza que as pré e pós condições de funções oferecem.

Assim sendo, os contratos estão descritos da seguinte forma:

$$\begin{array}{c} \{P\} \\ C \\ \{R\} \end{array}$$

em que P define uma pré condição, C uma assinatura de uma função e R uma pós condição. De notar que tanto a pré como a pós condição têm de ser elementos booleanos e devem-se referir à assinatura da função. Assim sendo definiu-se que apenas o contrato C é válido se a sua pré e pós condições devolverem *true*. A pré e pós condição podem ser vazias.

Neste sistema, definiu-se que a assinatura C da função pode ter uma particularidade, que é a instânciação de um elemento que pertença a um determinado tipo. Ou seja, pode-se dizer

$$soma :: a \sim Int \rightarrow b \sim Int \rightarrow Int$$

para expressar que a função *soma* recebe dois parametros a e b que são inteiros e devolve um elemento do tipo inteiro.

Decidiu-se também, que por motivos de facilidade de leitura e afim de evitar a complexidade formal, não especificar o estado interno do sistema, como por exemplo o estado do Apache, da base de dados entre outros componentes do sistema. Especificar isso não iria trazer nada de interessante ao que se pretende mostrar aqui. Assim, neste modelo formal apenas se vê de forma clara, os contratos que queremos que o nosso sistema tenha.

Começou-se então por definir o contrato da função *login* que permite a um determinado utilizador entrar no sistema. Esse contrato estipula que recebendo um par $Username \times Hash$ e um *SessionID* devolve ou um *Error* ou um novo *SessionID* que associa o utilizador à sua sessão no sistema. Afim de haver privacidade sobre os dados críticos do utilizador, como a password, decidiu-se apenas receber do lado do servidor a *Hash* respectiva da sua palavra-passe, sendo esta hash gerada do lado do cliente. Esta técnica não tem nada de novo, mas por incrível que pareça ainda há sistemas online que não usam este tipo de mecanismos.

$$\begin{array}{l} \{existsInDatabase(u)\} \\ login :: u \sim Username \times Hash \rightarrow SessionID \rightarrow Error + SessionID \\ \{\} \end{array}$$

Apresenta-se de seguida o modelo de dados formal que o sistema usa. Considerou-se que um *Exercicio* tem um *Enunciado* e um dicionário a relacionar *Input's* com *Output's*, um concurso tem um nome, um tipo e um conjunto de exercícios.

$data\ Dict\ a\ b = (a \times b)^*$
 $data\ Exercicio = Exercicio\ Enunciado\ (Dict\ Input\ Output)$
 $data\ Contest = Contest\ Nome\ Tipo\ Exercicio^*$

Para criar um novo concurso, é necessário assegurar que o utilizador que requisita este serviço é um professor, visto não sendo requisito alunos criarem concursos. É necessário ainda assegurar que o concurso que se vai criar tem no mínimo um exercicio.

$\{existeSession(s) \wedge isProf(s) \wedge (notEmpty \circ getExercice)\ c\}$
 $createContest :: s \sim SessionID \rightarrow c \sim Contest \rightarrow 1$
 $\{(notEmpty \circ getDict)\ c\}$

Para criar um novo exercicio, o utilizador tem de ser um professor e o exercicio em questão não pode ser repetido no sistema. Decidiu-se desse modo para evitar redundância na informação que se tem armazenada.

$\{existeSession(s) \wedge isProf(s) \wedge (not \circ exist)(Exercicioed)\}$
 $createExercice :: s \sim SessionID \rightarrow e \sim Enunciado \rightarrow d \sim (Dictab) \rightarrow 1$
 $\{exerciceCreated(Exercicioed)\}$

Pode-se ainda consultar os logs de um concurso específico. Aqui apresenta-se como argumento todo o concurso - *Contest* para apenas evitar a definição evidente de identificadores. É claro que a implementação desta acção, irá receber como parâmetro o identificador do concurso, como em outros parâmetros deste modelo formal.

$\{existeSession(s) \wedge isProf(s) \wedge contestIsClosed(c)\}$
 $consultarLogsContest :: s \sim SessionID \rightarrow c \sim Contest \rightarrow LogsContest$
 $\{\}$

De seguida mostra-se a especificação de efectuar um registo no concurso, pretende-se apenas que o concurso não esteja cheio.

$\{existSession(s) \wedge contestNotFull(c)\}$
 $registerOnContest :: s \sim SessionID \rightarrow c \sim Contest \rightarrow Credenciais$
 $\{\}$

Pode-se ainda submeter um exercicio, o que implica que esta acção tenha como consequência devolver um relatório com os resultados interessantes para mostrar ao participante de um concurso.

$\{existeSession(s) \wedge exerciceExist(e)\}$
 $submitExercice :: s \sim SessionID \rightarrow e \sim Exercicio \rightarrow res \sim Resolucao \rightarrow rep \sim Report$
 $\{rep = geraReportseres\}$

Modelação de acções de selecção

Existem acções do sistema proposto que envolvem a selecção de items nos forms ou então métodos que geram efeitos secundários, assim decidiu-se explicar aqui esse conjunto de contractos. Estes métodos são interessantes de modelar porque, assim fica mais claro ver os parâmetros que recebemos para os concretizar.

Tem-se então a acção de escolha de um exercicio numa panóplia de exercicios disponíveis no concurso que o utilizador está actualmente inscrito e a participar.

De relembrar que o uso do tipo 1 significa o tipo unitário, ou seja, do ponto de vista formal esta

operação não devolve nada, apenas altera o sistema. Sistema esse que no início explicou-se que por motivos de complexidade não tinha interesse especificar formalmente.

```
{existeSession(s) ∧ exerciceExist(e)}
escolheExercicio :: s ~ SessionID → e ~ Exercicio → 1
{}
```

Tem-se ainda a escolha do concurso para um grupo que está já registado.

```
{existSession(s) ∧ existContest(c) ∧ userRegistadoNoContest(s, c)}
escolheConcursoJaRegistado :: s ~ SessionID → c ~ Contest → 1
{}
```

Apresenta-se de seguida o contracto da função que gera um relatório ao concorrente.

```
{ }
geraReport :: e ~ Exercicio → res ~ Resolucao → Report
{ }
```

A título de exemplo, da potencialidade deste tipo de modelação, serviu-se agora da linguagem do Haskell para especificar a definição desta operação em maior detalhe, como se pode ver a partir das seguintes definições:

```
geraReportBugCompile :: Exercicio → Error → Report
geraReportBugCompare :: Exercicio → Errado → Report
geraReportNoBug :: Exercicio → Resolucao → Report

execute :: Program → Exercicio → ResolucaoProposta
```

A *ResolucaoProposta* é a resolução submetida pelos concorrentes e apresenta o seguinte tipo:
data ResolucaoProposta = Dict Input Output

Tem-se ainda a função que recebe uma resolução como input e devolve ora um programa pronto já compilado, ora um erro no caso de surgir algum problema na compilação.

```
{ }
compile :: Resolucao → Error + Program
{ }
```

E ainda uma função de comparação que recebe uma resolução e um exercicio e devolve se o Output pretendido é o mesmo que o que o programa submetido origina.

```
{length(Exercicio) == length(ResolucaoProposta)}
compare :: Exercicio → ResolucaoProposta → Certo + Errado
{ }
```

```
geraReport :: Exercicio -> Resolucao -> Report
geraReport exer res = do
  case compile res of
    (Left error) -> geraReportBugCompile error res
    (Right p) ->
      let resProps = execute p exer
      in case (compare exer resProps) of
        (Left certo) -> geraReportNoBug e res
        (Right errado) -> geraReportBugCompare errado res
```

Se fosse pretendido este código de um ponto de vista de composição de funções a sua conversão poderia ser facilmente atingida pelas seguintes definições:

```
geraReport exer res =
  compile res >>= \p -> compare exer (execute p exer)
  >>= \c -> geraReportNoBug exer res
```

O contrato da função que gera o relatório final baseando-se na participação de uma equipa num determinado concurso:

$$\{existSession(s) \wedge existContest(c)\}$$

$$geraFinalReport :: s \sim SessionID \rightarrow c \sim Contest \rightarrow DictExercicioResolucao \rightarrow Report$$

$$\{\}$$

2.3 Modelo de Dados

Definiu-se que existirão três tipos de utilizadores: o administrador, o docente e o grupo.

- Administrador - é a entidade com mais poder no sistema. É o único que pode criar contas do tipo docente. É caracterizado por:
 - Nome de utilizador;
 - Nome completo;
 - Password
 - e-Mail
- Docente - entidade que tem permissões para criar concursos, exercícios, aceder aos resultados das submissões, (...). Os seus atributos coincidem com os do Administrador.
- Grupo - entidade que pode registar-se em concursos e submeter tentativas para os seus diferentes enunciados.

Decidiu-se que o sistema terá a noção de grupo, e um grupo não é mais do que um conjunto de concorrentes. No entanto, o grupo é que possui as credenciais para entrar no sistema (nome de utilizador e password). Além disso também tem um nome pelo qual é identificado, um e-mail que será usado no caso de haver necessidade de se entrar em contacto com o grupo, e um conjunto de .

É importante incluir a informação de cada concorrente no grupo para, se possível, automatizar várias tarefas tais como lançamento de notas. Cada concorrente é caracterizado pelo seu nome completo, número de aluno (se for o caso), e e-mail.

Para finalizar, pretende-se explicar em que consistem os concursos, enunciados e tentativas.

Um concurso, resumidamente, é um agregado de enunciados. Contém outras propriedades, tais como um título, data de início e data de fim (período em que o concurso está disponível para que os grupos se registem), chave de acesso (necessária para o registo dos grupos), duração do concurso (tempo que o grupo tem para resolver os exercícios do concurso, a partir do momento que dá início à prova), e por fim, regras de classificação.

Um enunciado é um exercício que o concorrente tenta solucionar. Como seria de esperar, cada exercício pode ter uma cotação diferente, logo o peso do enunciado é guardado no mesmo. Para cada enunciado existe também um conjunto de inputs e outputs, que servirão para verificar se o programa submetido está correcto. Contém ainda uma descrição do problema que o concorrente deve resolver, assim como uma função de avaliação, função esta que define como se verifica se o output obtido está de acordo com o esperado.

Uma tentativa é a informação que é gerada pelo sistema, para cada vez que o grupo submete um

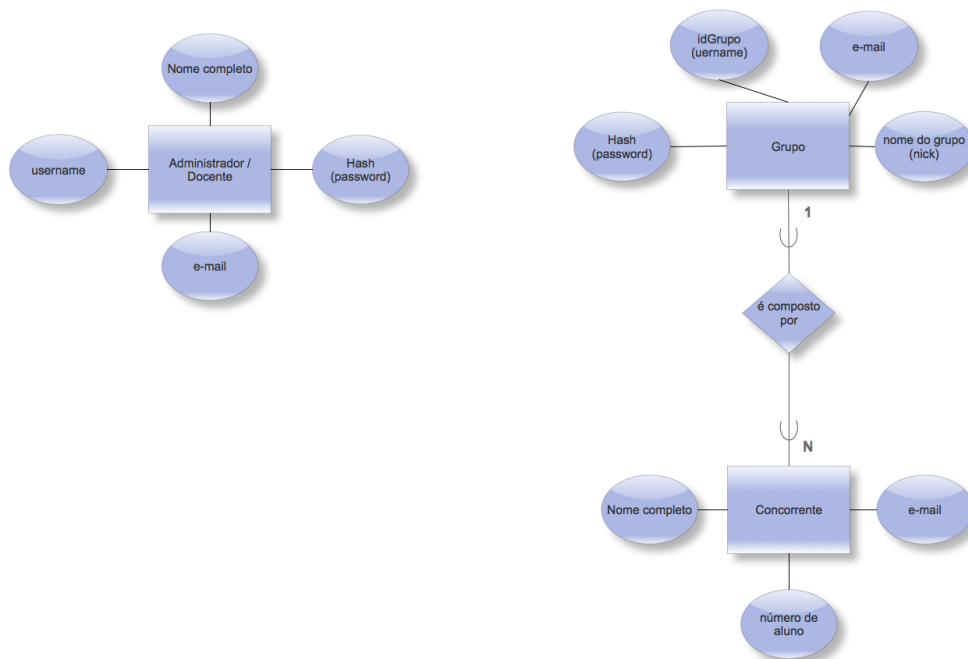


Figura 2.2: Modelo de dados - Grupo e Docente/Admin

ficheiro.

Além de conter dados sobre o concurso, enunciado e grupo a que pertence, a tentativa também contém o caminho para o código fonte do programa submetido, data e hora da tentativa, dados referentes à compilação e um dicionário com os inputs esperados e os outputs gerados pelo programa submetido.

2.4 Importação de dados

Uma das funcionalidades requeridas ao sistema proposto é a importação de enunciados e tentativas no formato xml. Esta funcionalidade será bastante útil para demonstrar e testar o sistema, sem que se tenha de criar manualmente os enunciados usando a interface gráfica, ou se tenha que submeter ficheiros com código fonte, de modo a serem geradas tentativas. Os campos presentes no xml de cada uma das entidades, *enunciado* e *tentativa*, são praticamente os mesmos que estão descritos no modelo de dados das respectivas entidades.

No xml do enunciado, não há nada muito relevante a acrescentar, além de que não contém um id para o enunciado, pois este será gerada automaticamente pelo sistema. Passamos agora a apresentar um exemplo do mesmo.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <Enunciado xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:noNamespaceSchemaLocation="enunciado.xsd">
5   <idConcurso> 1 </idConcurso>
6   <Peso>20</Peso>
7   <Titulo> Exercicio 1 </Titulo>
8   <Descricao> Some os numeros que lhe sao passados como argumento, e apresente o resultado.
9     </Descricao>
10  <Exemplo>Input: 1 1 1 1 1   Output: 5</Exemplo>
11  <Docente> PRH </Docente>
12  <FuncAval>Diff</FuncAval>
13  <Linguagens>
14    <Linguagem>C</Linguagem>
15  </Linguagens>

```

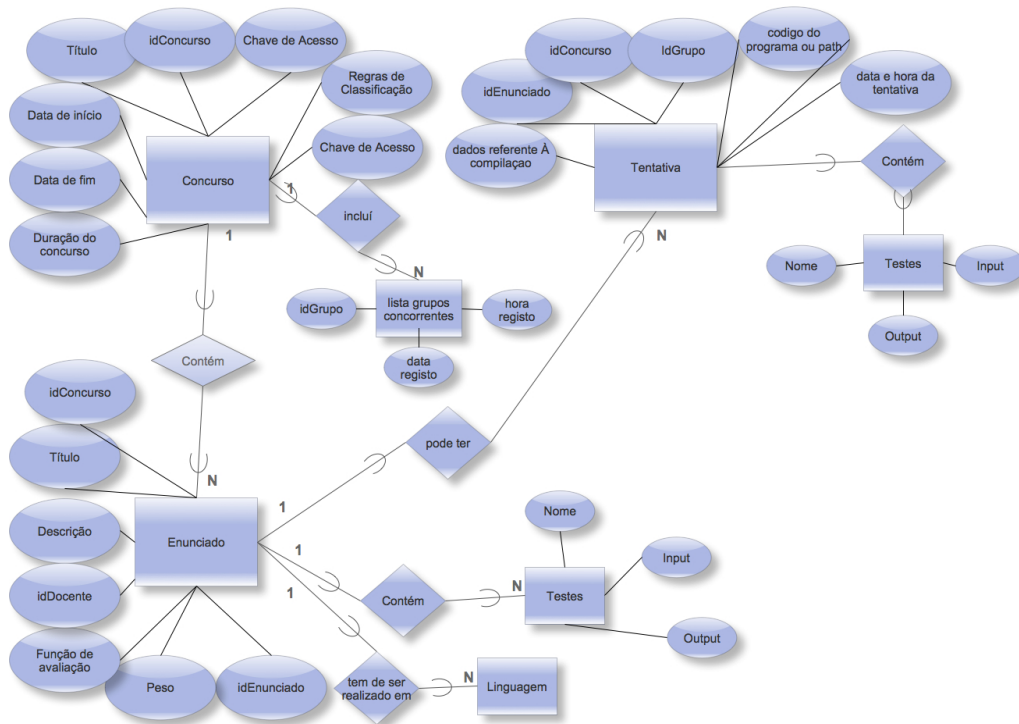


Figura 2.3: Modelo de dados - Concurso, tentativa e enunciado

```

15 <Dict>
16   <Teste>
17     <Nome>Lista vazia</Nome>
18     <Input></Input>
19     <Output>0</Output>
20   </Teste>
21   <Teste>
22     <Nome>Lista c/ 1 elem</Nome>
23     <Input>1</Input>
24     <Output>1</Output>
25   </Teste>
26   <Teste>
27     <Nome>Lista c/ varios elem</Nome>
28     <Input> 2 3 4 5 </Input>
29     <Output>14</Output>
30   </Teste>
31 </Dict>
32 </Enunciado>

```

Quanto ao xml para a *tentativa* há que realçar o facto de que o código fonte do programa vai dentro de uma tag xml. Além da tag xml, o código fonte terá de ir cercado de uma secção CDATA. Isto acontece para que o que o código fonte não seja processado com o restante xml que o contém.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <Enunciado xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:noNamespaceSchemaLocation="tentativa.xsd">
5   <idConcurso>1</idConcurso>
6   <idEnunciado>1</idEnunciado>
7   <idGrupo>36</idGrupo>
8
9   <data>2010-12-08</data>
10  <hora>16:33:00</hora>
11
12  <compilou>1</compilou>
13
14  <Dict>
15    <Teste>
16      <Nome>Lista vazia</Nome>

```

```

17         <Input></Input>
18         <Output>0</Output>
19     </Teste>
20     <Teste>
21         <Nome>Lista c/ 1 elem</Nome>
22         <Input>1</Input>
23         <Output>1</Output>
24     </Teste>
25     <Teste>
26         <Nome>Lista c/ varios elem</Nome>
27         <Input> 2 3 4 5 </Input>
28         <Output>14</Output>
29     </Teste>
30 </Dict>
31
32 <pathMetricas>sasas</pathMetricas>
33
34 <codigoFonte>
35     <nome>prog.c</nome>
36     <codigo>
37         <![CDATA[
38             #include <stdio.h>
39
40             ... restante codigo...
41         ]]>
42     </codigo>
43 </codigoFonte>
44 </Enunciado>

```

Para que todos os dados contidos nos ficheiros xml possam ser facilmente validados, foram criados dois *XML schema*. Neste schema, definiu-se quais as tags que devem existir em cada xml, o tipo de dados e até a gama de valores que serão contido por cada tag e a multiplicidade das tags .

Nesta fase inicial do projecto, ainda não foram sempre especificados os tipos de dados que serão contidos por cada tag.

No entanto, para alguns dos casos em que tal aconteceu apresenta-se alguns exemplos e explicações.

No xsd referente ao *enunciado* encontra-se o elemento *Peso*, que é um exemplo de uma tag que contém restrições. O *Peso* terá de ser um inteiro e terá um valor entre 0 e 100.

```

1 <ed:element name="Peso" default="25">
2     <ed:simpleType>
3         <ed:restriction base="ed:integer">
4             <ed:minInclusive value="0"/>
5             <ed:maxInclusive value="100"/>
6         </ed:restriction>
7     </ed:simpleType>
8 </ed:element>

```

Já o elemento *Linguagem* é também restringido, mas de uma forma ligeiramente diferente. A *Linguagem* será uma string, mas apenas poderá tomar um dos valores enumerados no xsd.

```

1 <ed:element name="Linguagem" maxOccurs="unbounded">
2     <ed:simpleType>
3         <ed:restriction base="ed:string">
4             <ed:enumeration value="C"/>
5             <ed:enumeration value="Java"/>
6             <ed:enumeration value="Haskell"/>
7         </ed:restriction>
8     </ed:simpleType>
9 </ed:element>

```

No xsd para a *tentativa* podemos evidenciar a multiplicidade das tags, ou seja, quantas vezes algumas delas se podem repetir. Na *tentativa*, existe um *Dict*, que contém uma ou mais tags *Teste*. Para definir que possam existir mais de que uma tag *Teste* dentro de *Dict*, adicionou-se o atributo *maxOccurs*, na entidade *Teste*, com o valor *"unbounded"*. O valor mínimo não é necessário definir, porque é um por default.

```

1 <tt:element name="Dict">
2     <tt:complexType>
3         <tt:sequence>
4             <tt:element name="Teste" maxOccurs="unbounded">

```

```
5         <tt:complexType>
6             <tt:sequence>
7                 <tt:element name="Nome" type="tt:string"/>
8                 <tt:element name="Input" type="tt:string"/>
9                 <tt:element name="Output" type="tt:string"/>
10            </tt:sequence>
11        </tt:complexType>
12    </tt:element>
13</tt:sequence>
14</tt:complexType>
15</tt:element>
```

Para dar uma ideia mais geral sobre ambos os *xml schema* criados, em vez de se apresentar aqui ambos os ficheiros integralmente, expôr-se-á os diagramas que o programa *oxygen* constrói e coloca ao nosso dispor, pois decidiu-se que torna o entendimento do schema muito mais simples.

Desta forma, de seguida apresentam-se os diagramas para o enunciado e para a tentativa:

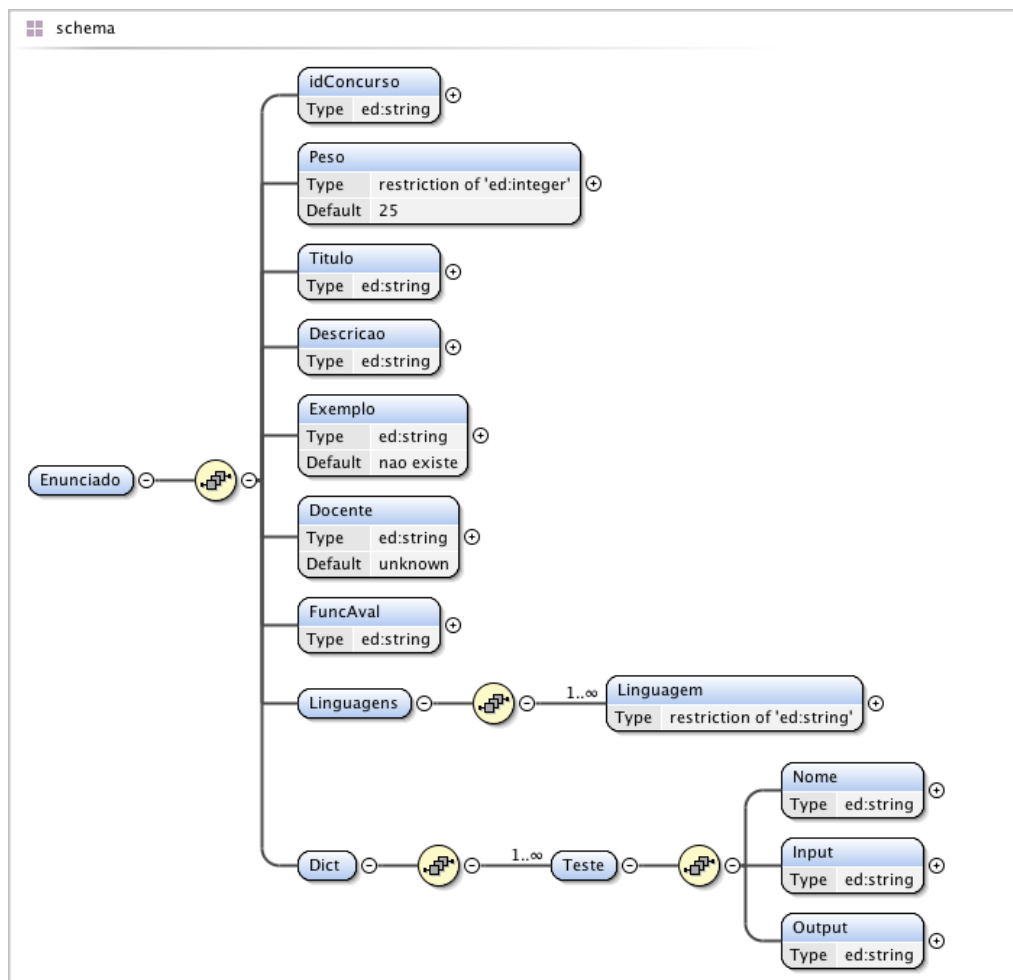


Figura 2.4: diagrama do schema para o enunciado

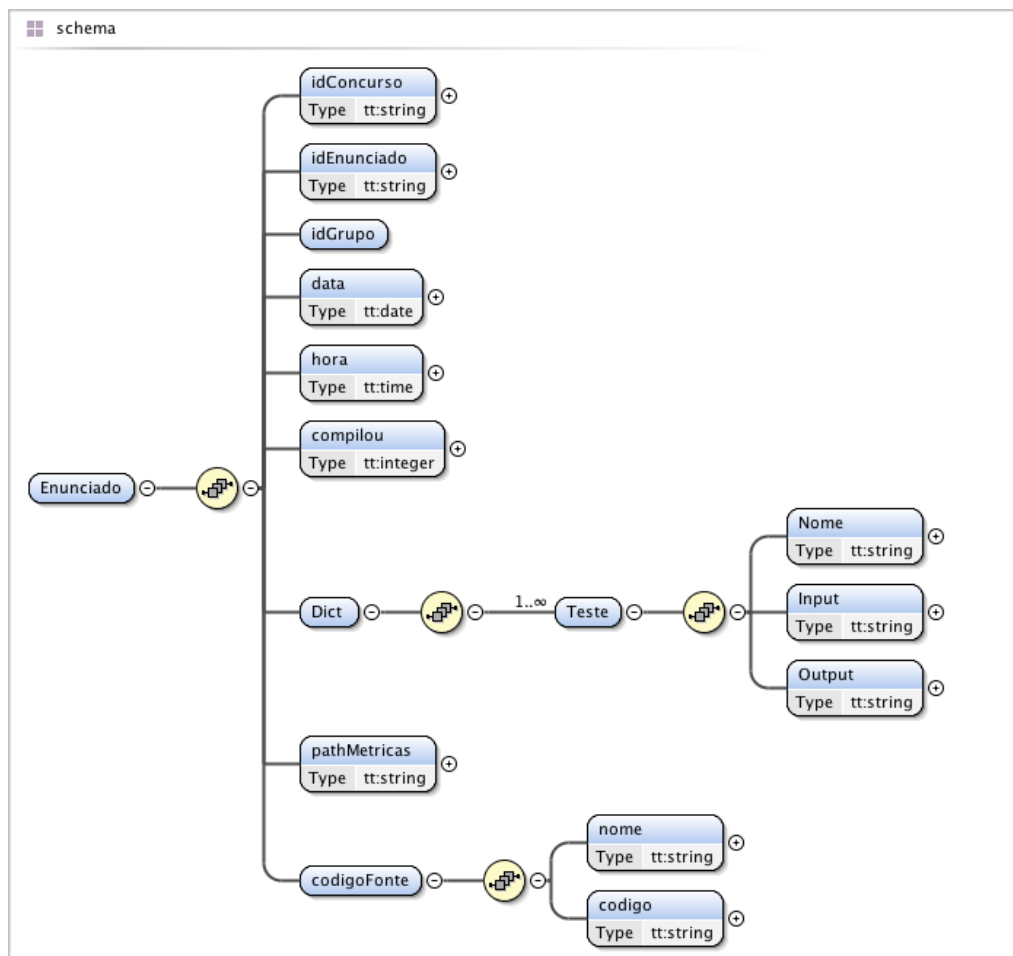


Figura 2.5: diagrama do schema para a tentativa

3

Métricas

Conteúdo

3.1	Análise Estática	18
3.2	Análise Dinâmica	19
3.3	Métricas de qualidade de <i>software</i>	20
3.3.1	Bug patterns	21
3.3.2	Source Lines of code (SLOC)	21
3.3.3	Métricas de Segurança	22
3.3.4	Cyclomatic complexity	23

Existem diversas métricas, com objectivos diferentes, para analisar um projecto de *software*. Essas métricas podem ser vistas como diferentes 'lentes' com as quais olhamos para um software. Neste capítulo, pretende-se mostrar a investigação que foi realizada relativamente a este tema, começando primeiro por definir alguns conceitos e depois dividir as métricas por categorias. Cada categoria será estruturada da mesma maneira, indicada mais à frente.

Assim, encarou-se a análise a um software como sendo uma área que se divide em dois ramos, a **Análise Estática** e a **Análise Dinâmica**.

3.1 Análise Estática

A Análise Estática é olhar para um programa sob o ponto de vista do seu código ou ficheiro já compilado, e retirar conclusões sobre as suas características, sem nunca recorrer à sua execução ou análise de resultados da execução. Ainda relativamente à Análise Estática, temos essencialmente duas maneiras de olhar para o software. Podemos ver este tendo em conta a qualidade do ficheiro objecto produzido (o código máquina que irá correr, p. ex: em java seria o *bytecode*) ou então tendo única e exclusivamente como objecto de observação os ficheiros de texto correspondentes ao código que compõe o programa. De notar que a Análise Estática é sempre relativa ao código do programa, ou seja, até mesmo uma análise que tenha em vista a qualidade do ficheiro objecto vai ser feita sobre o código do programa.

Assim sendo, no que diz respeito à qualidade do ficheiro objecto produzido, temos:

Syntax checking é um programa ou parte de um programa que tenta atestar a correcção da linguagem escrita.

Type checking é o processo de verificação dos tipos de dados num software que visa garantir a restrição no que diz respeito aos tipos, implicando assim maior qualidade do software produzido e menos probabilidade de acontecerem erros aquando da execução. Cada vez mais as linguagens recentes apresentam este tipo de sistemas, o que levam a que muitos dos erros ocorram em tempo de compilação, ou seja: completamente ainda a tempo de serem corrigidos pelos programadores, por exemplo: Haskell, C++0x, JAVA6. Estas linguagens apresentam um sistema de tipos forte o que garante este processo. ¹

Decompilation é o processo de pegar num ficheiro objecto e tentar inferir ou descobrir o seu código fonte que o originou. Designa-se assim porque é o inverso do processo de compilação. Com a ajuda deste tipo de análise consegue-se obter, entre outras coisas, os algoritmos alto nível do código máquina em questão. ²

No que diz respeito à análise da qualidade do código como produto final temos as seguintes metodologias:

Code metrics é uma vasta área que se dedica a análise do código em si para tirar conclusão acerca da sua qualidade, estabilidade e manutenção.

Style checking funciona como uma análise para verificar determinadas regras que à partida se acreditam como boas na produção de código. Estas regras podem ser relativas a indentação, existência de ficheiros README e de documentação. ³

Verification reverse engineering é o método que serve para verificar se a implementação de um determinado sistema cumpre a sua especificação. ⁴

O objectivo deste trabalho é puramente analisar estaticamente um programa, relativamente às métricas de código e eventualmente relativamente ao estilo também. Mesmo assim este tema tão vasto deixou-nos com motivação para conhecer o que é este mundo da análise de software.

3.2 Análise Dinâmica

Outros tipos de análises existentes são as chamadas análises dinâmicas, estas pegam numa peça de *software* e não tendo em conta, nem se preocupando com o código que a constitui, executam simplesmente o programa e analisam exhaustivamente sob vários prismas o seu comportamento. De seguida vamos dissertar sobre alguns destes métodos e práticas que existem para analisar a execução de um programa.

Log analysis é o método que consiste em pesquisar (automaticamente ou manualmente) os ficheiros de log produzidos por um determinado *software*, para perceber o que este está a fazer. Este tipo de análise muitas das vezes é feita a programas muito complexos e extensos que comunicam com o mundo real (rede, stdin, mundo IO). Um exemplo de quem faz este tipo de análise são os administradores de sistemas. ⁵

Testing é investigar o comportamento de um software através de uma bateria de testes que podem ter em consideração um determinado uso num caso que pode ser real. Geralmente, este tipo de análise simula os casos extremos a que o *software* pode ir, porque se acredita empiricamente que ao ter sucesso em situações extremas, há-de ter sucesso nos restantes casos. O que se pretende obter com este tipo

¹Mais informação em: http://en.wikipedia.org/wiki/Type_checking#Type_checking

²Mais informação em: <http://en.wikipedia.org/wiki/Decompiler>

³Mais informação em: http://en.wikipedia.org/wiki/Programming_style

⁴Exemplo [Aqui](#)

⁵Mais informação em: http://en.wikipedia.org/wiki/Log_analysis

de análise é o aumento na confiança de que o programa está a fazer o que é suposto, por parte de quem fabrica o produto.⁶

Debugging é um método que ajuda as pessoas a terem conhecimento do que determinado *software* está a fazer. Este método geralmente é usado ainda numa fase inicial do produto, quando está a ser desenvolvido pelos programadores. É um bom método de detectar defeitos, falhas ou pequenos *bugs* no *software*.⁷

Instrumentation é o método de monitorizar e medir o nível de performance de um determinado produto.⁸

Profiling é a investigação sobre o comportamento de um programa aquando a sua execução, usando para isso informações do género recursos computacionais. Este tipo de análise é útil para por exemplo efectuar gestão de memória.⁹

Benchmarking é o processo de comparar o processo do utilizador com os processos conhecidos de outros, de modo a obter conhecimento sobre as melhores práticas efectuadas na indústria.¹⁰

3.3 Métricas de qualidade de *software*

A presença de testes num determinado programa de *software*, leva a que esse artefacto ganhe pontos no que diz respeito à análise estática sob o ponto de vista da qualidade, porque, como dissemos anteriormente, a presença de testes num projecto de *software* leva a que tenhamos mais confiança neste. Existem algumas fórmulas que nos dão alguns indicadores numéricos sobre o nível desta confiança. De seguida são apresentadas algumas sobre a cobertura de testes.

$$\text{Line Coverage} = \frac{\text{Nr of test lines}}{\text{nr of tested lines}}$$

$$\text{Decision coverage} = \frac{\text{Nr of test methods}}{\text{Sum of McCabe complexity}}$$

$$\text{Test granularity} = \frac{\text{Nr of test lines}}{\text{nr of tests}}$$

$$\text{Test efficiency} = \frac{\text{Decision coverage}}{\text{line coverage}}$$

Podemos sempre aumentar a nossa precisão na análise se considerarmos apenas as linhas que contêm código e não as linhas em branco ou linhas com apenas um carácter, como por exemplo as aberturas de blocos em C, JAVA. Ainda podemos também considerar não apenas o numero total de linhas mas também o número de métodos/funções testados.

⁶Mais informação em: http://en.wikipedia.org/wiki/Software_testing

⁷Mais informação em: <http://en.wikipedia.org/wiki/Debugging>

⁸Mais informação em: [http://en.wikipedia.org/wiki/Instrumentation_\(computer_programming\)](http://en.wikipedia.org/wiki/Instrumentation_(computer_programming))

⁹Mais informação em: [http://en.wikipedia.org/wiki/Profiling_\(computer_programming\)](http://en.wikipedia.org/wiki/Profiling_(computer_programming))

¹⁰Mais informação em: <http://en.wikipedia.org/wiki/Benchmarking>

De notar que as formulas atrás descritas podem ser modificadas para obter outros tipos de métricas, não só referentes a testes, como por exemplo:

$$\text{Code granularity} = \frac{\text{Nr of lines}}{\text{Nr of (methods/functions)}}$$

Como já referimos anteriormente, a análise que se pretende, por agora, é essencialmente estática. Assim sendo, segue-se uma lista de métricas estudadas que se pretendem implementar no sistema.

3.3.1 Bug patterns

Um grave problema na utilização de linguagens de baixo nível como é o exemplo do C, é que tem de existir uma grande responsabilidade por parte do programador. Este tem de ter cuidado com pormenores que em linguagens de mais alto nível nem sequer pensa nisso. Um exemplo da falta de segurança do C, pode ser visto de seguida:

```
// foo.c file
#include <stdio.h>

int main() {
    char *a = "I like you";
    char *b = "I hate you";

    if(&a < &b) a = *(&a + 1);
    else      a = *(&a - 1);

    printf("%s\n", a);
}
```

Quando o utilizador executa o código acima descrito o que ele vai ter é o seguinte:

```
[ulissesaraujocosta@maclisses:c]-$ gcc -o foo foo.c
[ulissesaraujocosta@maclisses:c]-$ ./foo
I hate you
```

Isto demonstra que conseguimos aceder ao espaço de uma variável através de uma outra, simplesmente para isso tendo conhecimento de como funciona a stack de alocação de funções. Repare-se que o código prevê até o crescimento da stack, quer seja para cima ou para baixo. Assim conseguimos aceder ao valor de uma variável através de uma outra que esta declarada contiguamente.

À medida que subimos de nível nas linguagens vários problemas vão sendo melhorados e cada vez menos "poder" é dado ao programador. Por outra lado cada vez mais segurança também é dada. Mesmo assim há problemas sobre os quais nenhuma linguagem, por muito alto nível que ela seja, consegue resolver, um exemplo disso é:

- Null-dereferencing
- Lack of array bounds checking
- Buffer overflow

Como exemplo de uma ferramenta muito poderosa e que detecta vários padrões tidos como *bugs* é o *FindBugs* ¹¹ para JAVA que é gratuito. Todas essas ferramentas têm como objectivo principal ajudar o programador a ter alguma correcção do seu código.

3.3.2 Source Lines of code (SLOC)

Este tipo métricas diz respeito à informação que uma linha de código pode conter. Se pensarmos levemente no assunto podemos, erradamente, chegar a conclusão que contar o número de linhas é um problema fácil, mas mais uma vez aqui se mostra o quão empírico pode ser a tarefa de medir um

¹¹Ferramenta em <http://findbugs.sourceforge.net/>

software no que confere à sua qualidade.

Assim, podemos pensar em linhas de código, como linhas onde tenhamos alguma certeza de que se encontram partes fundamentais dos algoritmos e da implementação. Por exemplo, uma linha que esteja em branco ou que tenha comentários não deve ser contada, mais ainda, uma linha que apenas tenha (por motivos de indentação) um fechar bloco "}" também não é útil de contar para efeitos de ter uma noção total do SLOC.

Como podemos concluir, esta é uma métrica muito dependente da linguagem de programação que se está a usar. E é necessário arranjar um *tradeoff* entre a medição do esforço e a produtividade efectiva.

Como exemplo podemos ver ¹²:

```
for (i = 0; i < 100; i += 1) printf("hello"); /* How many lines of code is this? */
```

Neste exemplo temos:

- 1 Linha física de código (LOC)
- 2 Linhas lógicas de código (LLOC) (o for e o printf)
- 1 Linha de comentário

Neste caso, seria interessante contar este código como 2 linhas de código e não apenas como uma única.

3.3.3 Métricas de Segurança

No que diz respeito à avaliação de software relativamente a métricas de segurança, é necessário compreender os tipos de ataques possíveis que uma aplicação que comunica online ou com serviços terceiros pode sofrer. Existem várias técnicas de tentativa de corrupção ou alteração de um sistema online, entre elas as que se podem identificar automaticamente são:

SQL injection attack é um tipo de ameaça que ocorre quando são pedidos dados ao utilizador e este, aproveitando-se de falhas de segurança, injecta as suas próprias *queries* à base de dados. Segue-se um pequeno exemplo de como este ataque é efectuado, imagine-se a seguinte *query*:

```
SELECT descricao FROM enunciados WHERE titulo = 'exemplo1';
```

O campo *titulo* é dado por um utilizador do sistema. Com falta de controlo sobre esse campo, um utilizador com más intenções poderia escrever `exmplo1'; DROP TABLE users ;--`, o que resultaria na seguinte *query*:

```
SELECT descricao FROM enunciados WHERE titulo = 'exemplo1'; DROP TABLE users ; --'
```

Assim, um simples campo a pedir um dado a um utilizador poderia significar na destruição total da base de dados. ¹³

Storing plaintext and sending passwords são falhas de segurança relacionadas com a falta de protecção sobre as *passwords*. Imagine-se o seguinte caso, um sistema de login em que as *passwords*, em vez de estarem guardadas numa base de dados e protegidas por funções de *hash* (como no nosso sistema), estão guardadas num ficheiro de texto simples, ou na própria aplicação. Seria então trivial para um utilizador mal intencionado descobrir as *passwords*.

XSS - Cross-site scripting é um tipo de vulnerabilidade na segurança muitas vezes encontradas em páginas *Web*. Consiste em um cliente inserir *scripts* nessas páginas para que seja vista por outros utilizadores. Assim, um simples tarefa como descobrir uma *password* de um determinado sujeito seria bastante simples, bastando para isso que esse sujeito entrasse na página aquando o ataque. ¹⁴

¹²Informação e exemplo: http://en.wikipedia.org/wiki/Source_lines_of_code

¹³Mais informação em: http://en.wikipedia.org/wiki/SQL_injection

¹⁴Mais informação em: http://en.wikipedia.org/wiki/Cross-site_scripting

Existem várias ferramentas que procuram por estes padrões e tentam avisar o programador que o seu código pode ser vulnerável a algum destes tipos de ataques, entre elas sobressaem-se as seguintes aplicações proprietárias:

- Fortify para Java ¹⁵
- Coverity para C C++ C# ¹⁶

3.3.4 Cyclomatic complexity

Esta métrica é bastante completa e tem como objectivo indicar a complexidade de um programa [McC76]. Para isso representa-se um programa como um grafo de controlo de fluxo (um grafo orientado), onde os nodos são as instruções e uma aresta que liga dois nodos significa que aquelas duas instruções são executadas uma seguida da outra.

A definição é bastante simples, mas este tipo de representação de um programa pode ter grandes vantagens, por exemplo para medir a quantidade de *test cases* necessários desenvolver para testar todo um programa [WMW96]. Para atingir este objectivo apenas teríamos de contar o número de caminhos linearmente independentes que compõe este grafo.

A definição matemática deste conceito é:

$$M = E - N + 2P$$

onde:

M cyclomatic complexity

E o número de arestas no grafo

N o número de nodos no grafo

P o número de componentes ligados no grafo

Assim, depois de calcular para cada programa (que pode ser um conjunto de ficheiro) a sua Cyclomatic Complexity, temos na seguinte tabela uma relação directa entre este valor calculado e a taxa de risco do programa em questão.

Cyclomatic Complexity	Risco
1-10	um programa simples sem muito risco
11-20	programa mais complexo de risco moderado
21-50	programa complexo, alto risco
>50	programa instável (muito alto risco)

Tabela 3.1: Avaliação da cyclomatic complexity

¹⁵Ferramenta pode ser encontrada em <https://www.fortify.com/>

¹⁶Ferramenta pode ser encontrada em <https://www.coverity.com/>

Parte II

WebApp e interface pelo terminal & Scripts de Instalação e Avaliação

4

Web Application

Conteúdo

4.1 Criação de grupos, docentes e concorrentes	25
4.2 Linguagens de programação	27
4.3 Submissão de programas	28
4.4 Compilação	28
4.5 Execução	29
4.6 Guardar resultados	29

Neste capítulo vamos expor alguns pormenores relacionados com a implementação do problema proposto. Será explicada a maneira que encontramos para que seja cumprida a arquitectura que escolhemos e modelamos inicialmente.

4.1 Criação de grupos, docentes e concorrentes

O sistema permite que qualquer utilizador não registado se registe como grupo [4.1](#), e associe a si um ou mais concorrentes. Para efectuar o registo terá de preencher o nome do grupo/docente, um e-mail válido e uma password que tenha entre 6 a 40 caracteres.

Este login é utilizado por todo o grupo, para participar nos mais variados concursos.

Sign up

Name

Nome do docente

Email

docente@di.uminho.pt

Password

.....

Confirmation

.....

Sign up

Figura 4.1: Página de registo

Um concorrente é caracterizado por um nome, um número de aluno e um e-mail 4.2.



Grupo Os maiores

E-mail: manuel_s@gmail.com

Criado em: 12/03/2011 15:45:14

Concorrentes

nome: pedro antonio

e-mail: pedro_antonio@gmail.com

número de aluno pg19999

criado em: 12/03/2011 15:45:49

[delete](#)

[editar](#)

nome: josé silva

e-mail: jsilva@gmail.com

número de aluno pg11111

criado em: 12/03/2011 15:46:14

[delete](#)

[editar](#)

Figura 4.2: Dados de um grupo

As contas de docente só podem ser criados pelo administrador. Para simplificar o trabalho do administrador, o docente pode criar uma conta de grupo, à qual mais tarde será concedida privilégios de docente 4.3.

```

Terminal — ruby — 111x11
ruby-1.9.2-p136 :020 > user = User.last
=> #<User id: 101, name: "Nome do docente", email: "docente@di.uminho.pt", created_at: "2011-03-12 14:52:09",
updated_at: "2011-03-12 14:52:09", encrypted_password: "019a6cdafd5181e36628aa9a53dcf6c2c9ff600695286aa5669...",
, salt: "f39d5ee9e78d20d66257b89c0996ffcc1b675d3528af7b646f2...", admin: false>
ruby-1.9.2-p136 :021 > user.toggle!(:admin)
=> true
ruby-1.9.2-p136 :022 > user
=> #<User id: 101, name: "Nome do docente", email: "docente@di.uminho.pt", created_at: "2011-03-12 14:52:09",
updated_at: "2011-03-12 14:54:27", encrypted_password: "63e12baa9d2efc751de5957fb9cbec4fac34b8ed73f9b57bc6e...",
, salt: "f39d5ee9e78d20d66257b89c0996ffcc1b675d3528af7b646f2...", admin: true>
ruby-1.9.2-p136 :023 >

```

Figura 4.3: Comandos necessários para tornar um utilizador sem privilégios num docente.

4.2 Linguagens de programação

O nosso sistema é multilingue, ou seja, é possível submeter código fonte em várias linguagens de programação diferentes, desde que a linguagem tenha sido correctamente configurada por um docente. Cada linguagem é caracterizada por uma série de campos 4.4, os quais serão explicados de seguida:

- string de compilação: string que será executada quando se pretender compilar determinado código fonte. Esta string tem a particularidade de no lugar em que é suposto conter o nome do ficheiro a compilar, contém `#{file}`.
Desta forma a string de compilação torna-se genérica, e independente do nome do ficheiro a compilar. exemplo: `gcc -O2 -Wall #{file}`
- string simples de execução: string utilizada para executar quando o código fonte foi compilado pela string de compilação.
exemplo 1:
- string de compilação: `gcc -O2 -Wall #{file}`
- string de execução respectiva: `./a.out`
exemplo 2:
- string de compilação: `gcc -O2 -Wall #{file} -o exec`
- string de execução respectiva: `./exec`
- string complexa de execução: a necessidade de uma segunda string de execução surgiu quando tentamos preparar o sistema para receber makefiles (inicialmente apenas para C). Nestes casos, o nome do executável gerado pela compilação não é conhecido à partida. Desta forma é necessário analisar o makefile, e só depois executar, tendo em conta a informação que retiramos do makefile. Assim, e para a linguagem C, a string complexa de execução seria:
- `./#{file}`
em que `#{file}` representa o nome do executável.

Nova linguagem

Nome da linguagem

String usada para compilar

(coloque #{file} aonde colocaria o nome do ficheiro a compilar)

String usada para executar

(no lugar do nome do executavel coloque o nome default do ficheiro

(ex:a.out))

String usada para executar

(coloque #{file} aonde colocaria o nome do ficheiro a executar)

Figura 4.4: Configuração de uma nova linguagem de programação no sistema

4.3 Submissão de programas

Sempre que acharem adequado, os grupos podem submeter os seus programas para serem avaliados. Para tal têm de escolher a linguagem de programação na qual resolveram o problema, de entre as disponíveis. E de seguida basta escolherem o ficheiro que pretendem submeter e carregar no botão de submissão.

No caso do administrador, pode ainda submeter tentativas no formato XML.

Enunciado enunciado 1

Descricao: a

Peso: 28%

função de avaliação: diff -wiB

Linguagem:

Submeter tentativa: No file chosen

Figura 4.5: Página de submissão de programas (vista de administrador)

4.4 Compilação

Estando as linguagens de programação correctamente configuradas, a compilação torna-se bastante simples. Quando uma tentativa é submetida no sistema, começamos por verificar se foi submetida apenas um ficheiro de código, ou um ficheiro comprimido.

Caso seja apenas um ficheiro, o sistema tenta compilar o código submetido, com a string de compilação da linguagem de programação em causa.

No caso de se tratar de um ficheiro comprimido, depois de o descomprimir, o sistema verifica se existe um makefile entre os ficheiros extraídos. Caso se verifique, é corrido o comando *make*, e tenta retirar o nome do executável gerado, de modo a poder ser usado na execução.

4.5 Execução

No fim da compilação, o sistema vai executar o programa uma vez para cada input. O processo de execução no caso de a compilação ter sido feita à custa do makefile, é feita usando a string complexa de execução (sendo o nome do executável aquele que foi retirado do makefile) . Se tal não tiver acontecido, é usada a string simples.

A execução pode ser abortada se ultrapassar o tempo máximo de execução, que é definido aquando da criação do enunciado em questão. A título de demonstração, de seguida apresentamos a porção de código que "mata" o processo relativo à execução do programa, caso ele demore mais de que o tempo máximo.

```
#thread que executa o a.out
out = "default";i=0
exec = Thread.new do
  out = '#{execString} #{input}'
end

#thread que conta x segundos e dps termina a execucao do programa
timer = Thread.new do
  sleep 5
  if exec.alive?
    Thread.kill(exec)
    i=1
    if params[:tentativa][:execStop] == false
      @erros += "Time out! Pelo menos a execucao de um dos inputs foi terminada por demorar
demasiado tempo!"
    end
    params[:tentativa][:execStop] = true
  end
end

exec.join
if timer.alive?
  Thread.kill(timer)
end
timer.join
```

4.6 Guardar resultados

Para cada input do enunciado em questão, o programa é executado uma vez. O seu output é comparado com o output esperado e é guardada uma entrada na base de dados com a percentagem de testes nos quais o programa teve sucesso.

No caso de o código não compilar, ou da execução do programa demorar mais tempo do que o máximo previsto pelo docente quando criou o enunciado, estas informações são também guardadas na base de dados.

Além de se guardarem todas as tentativas, a melhor é também guardada numa tabela à parte, para que o melhor resultado para cada enunciado seja de fácil acesso.

A qualquer momento o grupo pode consultar os dados relativos às suas últimas tentativas ou às últimas tentativas de todos os participantes [4.6](#), e também os seus melhores resultados.

Num Tentativa	Grupo	Concurso	Enunciado	Resultado	Compilou	Terminou a execução
62	Example User	concurso 2 (Activo)	enunciado 1	0.0	true	false
61	Example User	test contest (Activo)	teste2	100.0	true	false
60	Example User	test contest (Activo)	teste2	100.0	true	false
59	Example User	test contest (Activo)	teste2	100.0	true	false
58	Example User	test contest (Activo)	teste2	0.0	true	false
57	Example User	test contest (Activo)	teste2		false	false
56	Example User	test contest (Activo)	teste2	0.0	false	false
55	Example User	test contest (Activo)	teste2	0.0	false	false

Figura 4.6: Página onde pode consultar as tentativas (vista das tentativas de todos os utilizadores)

5

Interface pelo terminal

Conteúdo

5.1	Perl	31
5.2	Milestone II	32
5.3	Milestone IV	33

Tendo em vista a facilidade, para alguns utilizadores, em manusear um sistema por um terminal, decidiu-se criar uma interface para a aplicação desenvolvida. Esta interface, ainda em fase de desenvolvimento, vai permitir, essencialmente, trabalhar com a base de dados do sistema. Os objectivos passam por consultar listas de determinadas entidades, desde enunciados a utilizadores do sistema. De realçar que este modo de comunicação com o sistema apenas é utilizado pelos **administradores**

5.1 Perl

Como linguagem de desenvolvimento para esta interface, decidiu-se usar Perl devido à rapidez de implementação (visto que a criação de uma interface pelo terminal não constituía um dos principais objectivos) e a vasta diversificação de módulos existentes para auxílio ao desenvolvimento. Desses módulos, destaca-se o uso do módulo `DBIx::Class`, um módulo de comunicação a base de dados (apresentado durante as aulas de EL::PLN), que basicamente representa em classes cada tabela existente na base de dados, transformando também simples queries em métodos sobre as tabelas.

Tem-se em vista também a utilização dos seguintes módulos:

Digest::SHA2 módulo necessário para ser possível criar um utilizador. A utilização deste módulo servirá para cifrar a password para posteriormente guardar na base de dados. Escolheu-se este módulo por ser compatível com uma *gem*, com o mesmo nome, da ferramenta Rails

Term::ReadLine módulo que se pretende usar para substituir os menus clássicos, como se pode ver na secção seguinte, por um sistema actual por comandos e histórico.

XML::DT módulo que se irá usar para a transformação de enunciados em XML para Latex e para a inserção na base de dados.

5.2 Milestone II

Na fase actual desta interface, o utilizador desta interface terá pela frente um menu principal (Figura 5.1).

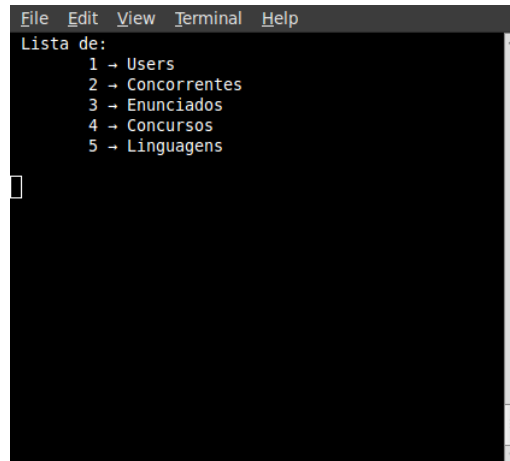


Figura 5.1: Menu Principal

Cada escolha desse menu representa as principais acções que se podem efectuar com uma base de dados:

- A listagem de elementos
- A procura de certos elementos e posterior actualização dos mesmos
- A inserção de novos elementos

Assim, caso o utilizador escolha a primeira opção, será levado para um novo menu contendo as 5 principais entidades deste sistema: Os Users; os Enunciados; as Concorrentes; os Concursos; e as Linguagens disponíveis para responder em cada concurso. A título de exemplo, caso o utilizador quisesse saber as linguagens disponíveis, a informação seria apresentada da seguinte forma(Figura 5.2):

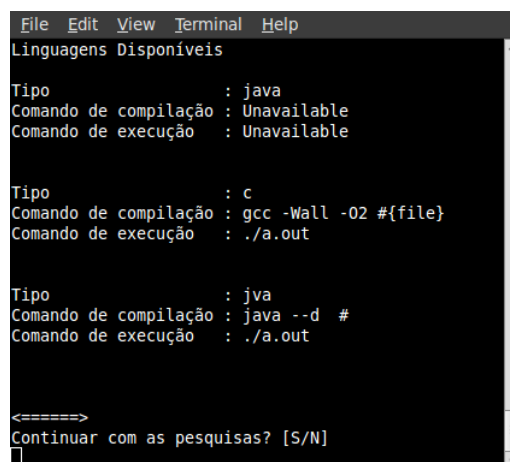
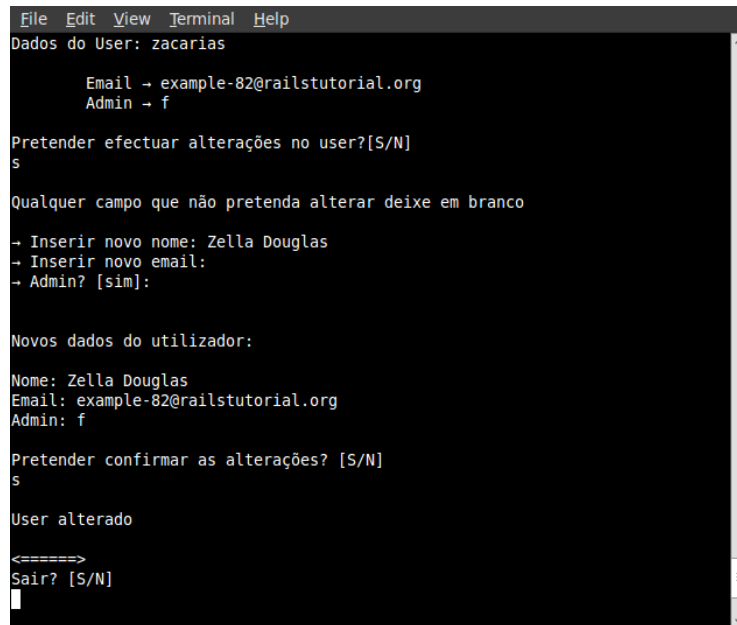


Figura 5.2: Linguagens disponíveis

O utilizador também poderia procurar por um único elemento. Como se pode ver na Figura 5.3, dando um nome de utilizador, seria retornada a informação sobre esse sujeito. Caso pretendesse, o utilizador poderia posteriormente proceder à alteração do mesmo.



```
File Edit View Terminal Help
Dados do User: zacarias

      Email → example-82@railstutorial.org
      Admin → f

Pretender efectuar alterações no user?[S/N]
s

Qualquer campo que não pretenda alterar deixe em branco

→ Inserir novo nome: Zella Douglas
→ Inserir novo email:
→ Admin? [sim]:

Novos dados do utilizador:
Nome: Zella Douglas
Email: example-82@railstutorial.org
Admin: f

Pretender confirmar as alterações? [S/N]
s

User alterado

<=====>
Sair? [S/N]
```

Figura 5.3: Procura e alteração do user Zella Douglas

5.3 Milestone IV

A implementação da interface pelo terminal foi adiada para trabalho futuro. Justifica-se esta decisão por esta interface não acrescentar nada de muito relevante à utilização deste sistema, pois todas as operações, que se encontravam disponíveis, poderiam-se realizar através da aplicação *web*. Também houve diversos problemas de compatibilização de versões de *Perl* usadas, que dificultaram em muito o seu processo de desenvolvimento.

Ainda assim, houve um melhoramento relativamente à disponibilização dos menus, encontrando-se agora num modo mais clássico aos sistemas *Unix*, dispondo de históricos, da capacidade de se auto-completar através de *emph*tabs. Também se encontrava implementada a adição de diversos elementos à base de dados, onde a adição de utilizadores foi o ponto de maior interesse devido ao uso de módulos criptográficos de modo a ser compatível com a aplicação *web*.

Finaliza-se este capítulo com a intenção de no futuro se proceder à continuação do desenvolvimento desta interface, possivelmente quando a análise ao primeiro caso de estudo se encontrar terminada.

6

Scripts de avaliação

Conteúdo

6.1	Geração de imagens	34
6.2	Makefile	36
6.3	Cloning	37

A ideia de usar scripts auxiliares prende-se com o facto de muitas pequenas operações conseguem ser muito simplificadas com a rápida implementação de um script que ajue a resolver o problema em questão. Como linguagem principal usamos essencialmente o Perl por ser uma linguagem fundamental para esta UCE, mas porque também estarmos satisfeitos com as potencialidades únicas que apresenta. Algumas vezes deparámo-nos com pequenos problemas onde não se justifica o uso de Ruby ou Haskell por trazer mais complexidade e baixar o ritmo ao desenvolvimento desta aplicação.

6.1 Geração de imagens

Foi desenvolvido um script que usa essencialmente o módulo GD do Perl para gerar estatísticas relativas ao número de linhas do projecto submetido. Este script (count.pl) encontra-se muito bem documentado, acompanhado de um README onde é explicado o seu funcionamento:

```
perl count.pl -open <dirPath> [-verbose] [-separated | -allTogether]
               [-percent] [-bars | -pie] -out <fileNamePrefix>
```

Este script na sua utilização mínima pode ser executado da seguinte maneira:

```
perl count.pl -open ~/projecto -out projecto
```

Isto irá pesquisar recursivamente na pasta ~/projecto todos os ficheiros de várias linguagens de programação e produzir 3 imagens, todas elas com o prefixo projecto.

```
projecto_LinesPerLanguage.png
projecto_FilesPerLanguage.png
projecto_RatioFilesLines.png
```

Cada uma destas 3 imagens contem informação relativa à totalidade do projecto, a primeira imagem (6.1) mostra a quantidade de linhas de código e de comentários relativamente a todas as linguagens presentes no projecto. Desta forma conseguimos ter uma noção do impacto que cada linguagem tem para o projecto final e a quantidade de documentação que existe relativamente a cada linguagem.

Outra imagem que a execução do comando anterior gera é a quantidade de ficheiros por linguagem, assim sendo temos uma noção da modularidade que existe em cada utilização de cada uma das linguagens utilizadas.

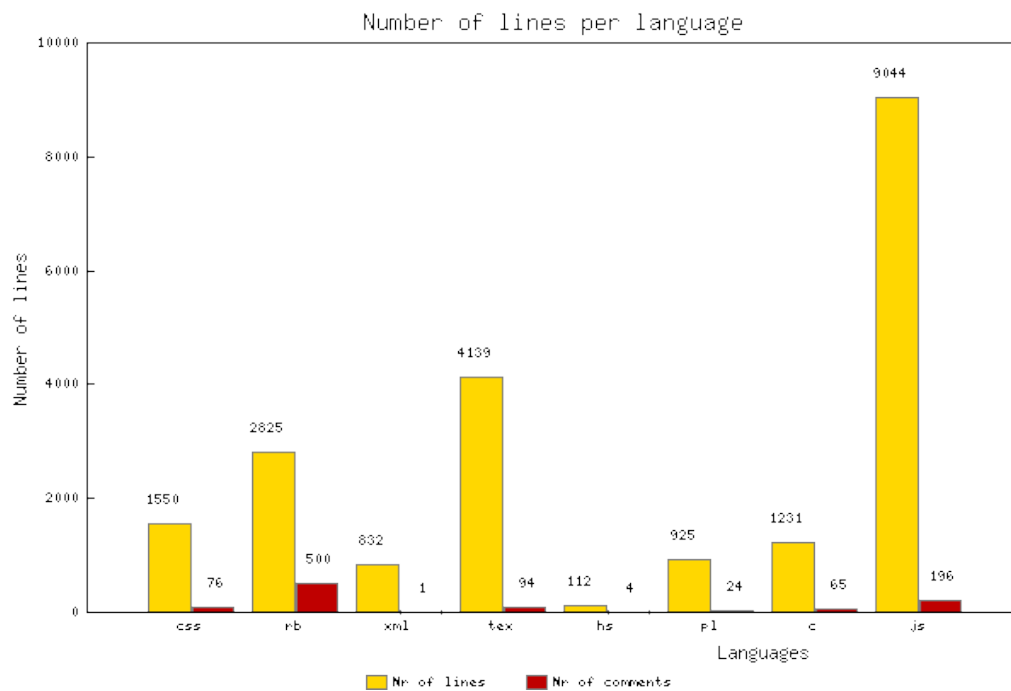


Figura 6.1: Linhas por Linguagem

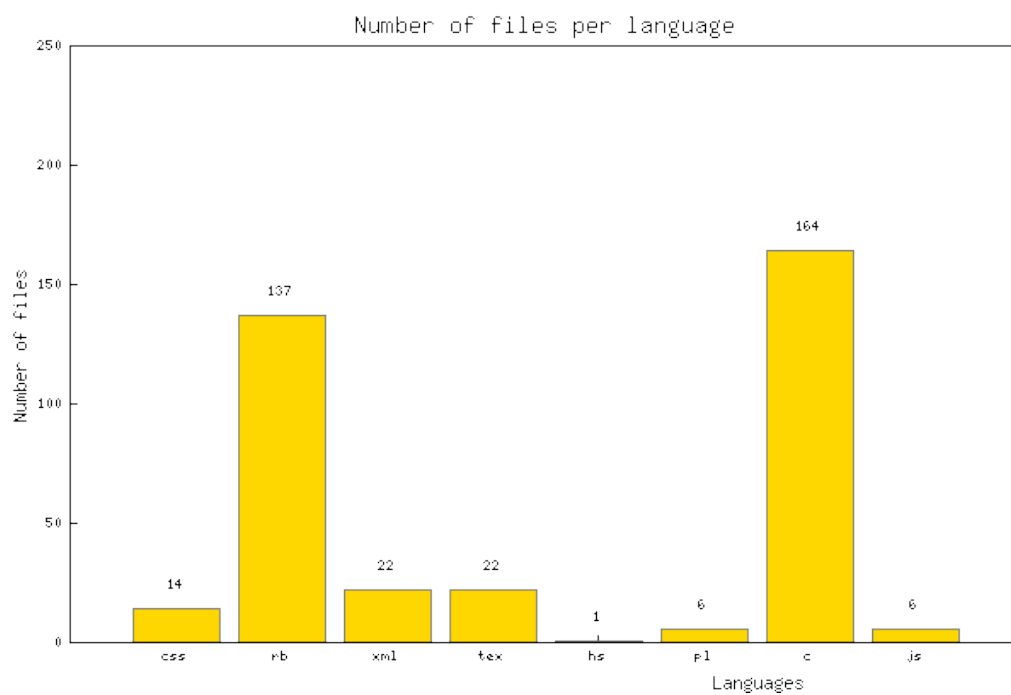


Figura 6.2: Ficheiros por Linguagem

A última imagem gerada é um rácio entre o número de ficheiros e o número de linhas para cada linguagem. Assim conseguimos ter uma noção do número médio de linhas que constitui cada um dos ficheiros do projecto.

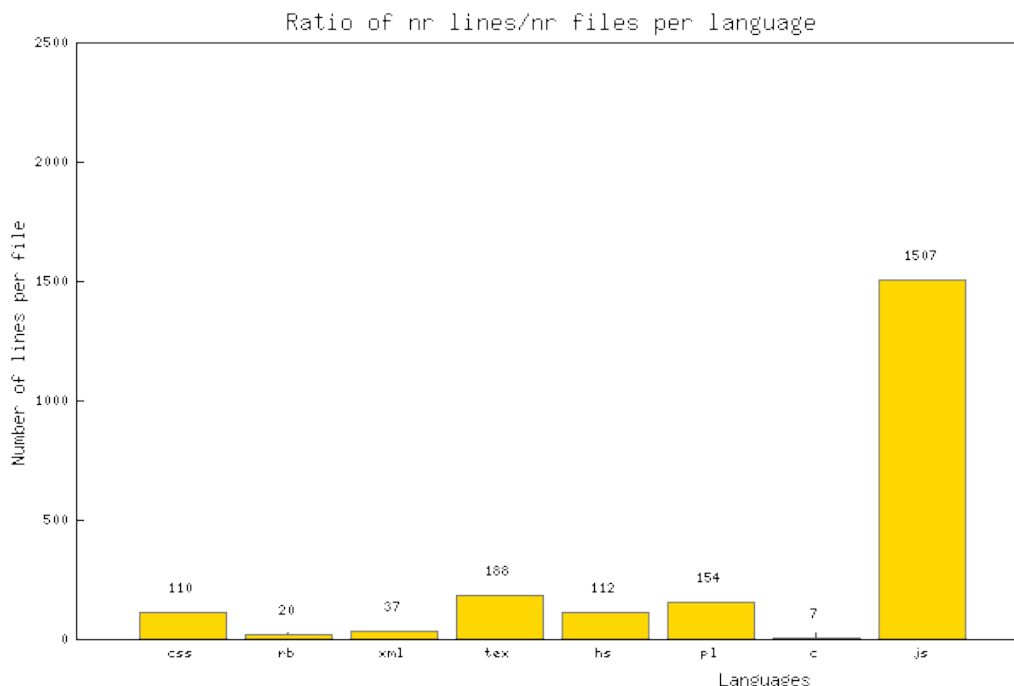


Figura 6.3: Linhas por Linguagem

Como se pode verificar, todas estas estatísticas são meramente um indicador e nada de maior se pode concluir, sem ser apenas ter uma noção global da quantidade de linhas e o uso de quais linguagens relativamente a um projecto.

Caso queiramos podermos ainda gerar as mesmas imagens, mas em percentagem. Como é obvio não faz sentido gerar percentagens sobre rácios, assim quando o utilizador emite o comando:

```
perl count.pl -open ~/projecto -out projecto -percent
```

geramos apenas a percentagem para as duas primeiras imagens.

Uma outra utilização interessante deste script permite a geraçã de uma única imagem com o intuito de ter um resumo de todo o projecto submetido numa única imagem, como se a imagem que sumariza a utilização de linhas do projecto.

Para gerar este ficheiro precisamos de fazer:

```
perl count.pl -open ~/projecto -out projecto -all
```

Por defeito quando utilizamos esta flag, todos os resultados que vão aparecer irão ser em percentagem, visto que podemos ter uma grande disparidade de números de linhas entre várias linguagens. Obtemos assim a imagem 6.4:

Este script é ainda capaz de gerar pie charts em vez de gráficos de barras.

6.2 Makefile

O nosso sistema de submissão permite ao utilizador submeter um `makefile` afim de facilitar a compilação do código do seu projecto, caso a compilação deste seja mais exigente do que uma trivial passagem pelo `gcc`. Assim é necessário extrair do `makefile` a informação sobre o ficheiro `objecto` que este vai gerar aquando da execução do comando `make`.

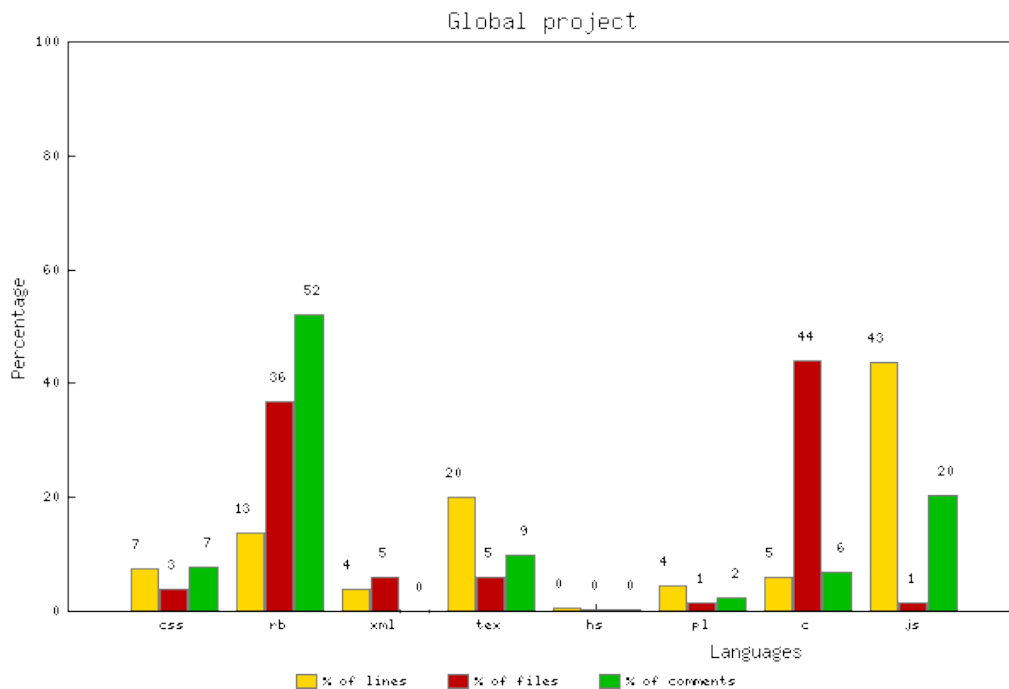


Figura 6.4: Visão global do projecto

Servindo-nos do módulo `Perl Makefile::Parser` conseguimos, com poucas linhas extrair o nome do binário que irá ser gerado pelo makefile.

Muitas das vezes, são scripts com a simplicidade que este apresenta que fazem a diferença e que mostram o verdadeiro poder de estar à vontade numa linguagem de utilização rápida como são as de scripting e nomeadamente o Perl.

Pretendemos melhorar este script e não o damos como terminado.

6.3 Cloning

Um dos objectivos finais para este projecto é que o sistema tenha um mecanismo de detecção de clones. Apesar de ainda não implementado no sistema, o nosso trabalho nesse sentido já começou. Criamos um script na linguagem *Perl*, a linguagem escolhida prende-se com o facto de grande parte das tarefas realizadas pelo script serem de processamento de texto.

De seguida vamos descrever passo a passo o que o script faz actualmente, tendo em conta que ainda não está preparado para ser utilizado no sistema:

- recebe um ficheiro por parâmetro
- executa o comando `ctags -x *.c`, cujo output contém entre outras coisas, o nome das funções que existem nos ficheiros `c` encontrados, e as linhas referentes a cada um
- a partir do output do `ctags` retira as linhas que identificam o início de cada função
- lê o ficheiro recebido como parâmetro, divide-o por funções, guardando as linhas de cada uma, numa posição da hash
- analisa cada posição da hash e altera-a, retirando todos os comentários e espaços, substituindo todas as variáveis por *var*, todas as strings por *S* e todos os números por 1

- de seguida lê um segundo ficheiro e trata-o da mesma forma que o primeiro
- por fim compara cada posição da primeira hash com todas as posições da segunda. Se houver casos em que as duas são iguais, imprime para o STDOUT um aviso

Temos noção de que o script precisa de algumas reformulações para a sua integração com o sistema. Para começar, em vez de receber apenas um ficheiro por parâmetro deverá receber o path de dois ficheiros. Além de prepararmos o script para ser integrado no sistema também é necessário afiná-lo de modo a que não existam muitos falsos positivos.

Estas entre outras, serão algumas das alterações ao script feitas para a próxima fase, que garantirá ao sistema a ferramenta necessária para a detecção de clones.

7

Detecção de clones

Conteúdo

7.1	Modo de utilização	39
7.2	Processamento do código	39
7.3	Output	40
7.4	Integração com a aplicação	40

Os primeiros passos para a implementação da detecção de clones na aplicação já tinha sido dados anteriormente, quando se começou por escrever um script em Perl para o efeito. O script já foi descrito em secções anteriores, no entanto apenas agora foi integrado na aplicação. Como nesse processo sofreu algumas alterações, vamos voltar a explicá-lo ao pormenor.

7.1 Modo de utilização

O script tem como finalidade comparar dois ficheiros, e verificar se foram detectadas possíveis cópias, entre os dois. Desta forma o script ao ser chamado, recebe dois ficheiros como input:

- `perl cloneDt -file path1 -comp path2`

sendo que *path1* representa o path do ficheiro que se está a testar e *path2* representa o path do ficheiro com o qual se pretende comparar.

7.2 Processamento do código

Para que seja possível detectar possíveis clones, mesmo depois de o grupo que está a submeter a cópia ter alterado por exemplo o nome a algumas variáveis ou trocado algumas funções de ordem, é necessário fazer um processamento do código fonte, generalizando ou eliminando alguns dos componentes de um programa.

Para tal, o script efectua uma série de procedimentos que irão ser descritos de seguida:

- começa por executar o comando `ctags -x path1`, de forma a obter a linha em que começa cada função do ficheiro C em questão;
- guarda num array o código desse ficheiro, dividindo cada função por uma posição diferente do array;

- elimina comentários uni e multi-linha, da linguagem C;
- substituí todas as variáveis e tipos por *var* ;
- substituí todos os números por *1*;
- elimina todos os espaços;
- de seguida efectua os mesmos passos para o segundo ficheiro;
- compara cada elemento do primeiro array gerado, com os elementos do segundo;

7.3 Output

O script no fim de comparar os dois arrays, no caso de ter detectado uma possível cópia em pelo menos uma função, imprime a percentagem de funções nas quais foram detectadas cópias.

7.4 Integração com a aplicação

Sempre que uma tentativa é submetida no sistema, o script é executado, comparando os ficheiros C que acabou de submeter com os ficheiros C das tentativas mais recentes, referentes ao mesmo enunciado, dos restantes grupos.

No caso de se detectar um possível clone, é inserida uma nova entrada na base de dados, com a informação referente a este evento.

A qualquer momento o administrador pode consultar esta lista de warnings, e se achar que foi um falso positivo, pode eliminar a entrada da lista. Ainda na listagem pode aceder ao código dos ficheiros em que se detetou o possível clone.

Warnings					
Percentagem de funções em que se detectou	Ficheiro	Comparado com	ID concurso	ID enunciado	Eliminar warning
100.0	contaParams.c	contaParams.c	11	8	delete
100.0	somaParams.c	somaParams.c	11	8	delete

Figura 7.1: Listagem dos warnings

8

Instalação

Conteúdo

8.1	User check	41
8.2	Identificação da máquina	42
8.3	Instalação MacOSX	43
8.3.1	Bibliotecas C	43
8.3.2	Módulos Perl	43
8.3.3	Instalação de software essencial para a Partell	43
8.3.4	Ruby, Rails e Gems	44
8.4	Instalação Ubuntu	44
8.4.1	Bibliotecas C	45
8.5	Fase actual	45

Após um dos elementos do grupo ter sofrido um grande infortúnio com a sua máquina de trabalho, surgiu a ideia, que deveria ter sido lembrada no início do desenvolvimento deste projecto, de implementar uma *script* de instalação (em *bash*) que cuidasse de pôr em funcionamento todo o tipo de aplicações que o projecto oferece.

Assim, começou-se por identificar as diversas ferramentas usadas ao longo deste projecto e que serão essenciais para o seu funcionamento final. Desde da aplicação web, com *Ruby* e *Rails* (e diversas *gems* de cada), ao *parser*, com *Haskell*, *Strafunski* e *Language.C*, a passar pela interface pelo terminal implementada em *Perl* (e os seus inúmeros módulos), vasto será o leque de preocupações que se terá que ter para que a partir de uma simples *script* se ponha o projecto pronto a funcionar.

Neste capítulo será explicada o funcionamento da *script*, desde da identificação inicial da máquina em que se encontra, à instalação do mais pequeno módulo de *Perl* utilizado. Vale a pena também referir a importância desta *script*, pois além de ser bastante útil para a gestão e manutenção deste projecto, e apesar de ser uma tarefa árdua (tendo em conta que se começou a meio do projecto), torna-se muito gratificante pelos ganhos de *skill* em administração de sistemas.

8.1 User check

Antes de se dar início à instalação da máquina é chamada uma função `check_user_id` para verificar se o utilizador é administrador da sua máquina. O seguinte pedaço de código diz respeito a essa função.


```
1 function check_user_id {
2     if [ ! "$(whoami)" = "root" ]; then
3         echo "Not running as root. Yes this is an installation file..."
4         exit 1 ;
5     fi
6 }
```

8.2 Identificação da máquina

A *script* começa a funcionar pela operação mais básica possível, a identificação da máquina em que corre. Esta poderá ter estar instalada com diversos sistemas operativos, o que só por si, diferenciaria muito o comportamento da *script*. No momento em que corre o projecto, apenas foi possível ter em consideração máquinas com sistemas operativos Linux e MacOSX.

De seguida encontra-se o pedaço de código para o funcionamento em questão:

```
1 function install_package {
2     case `uname -s` in
3         "Darwin")
4             install_macosx
5             ;;
6         "Linux")
7     case `uname -v` in
8         *"Ubuntu"*)
9             install_ubuntu
10            ;;
11            *)
12            echo "Your Linux is not supported yet. If it does have a packet manager please
13                send an email to $admin_email"
14            exit 1;
15            ;;
16        esac
17        ;;
18        *)
19        echo "Your operative system is not supported yet. Please send an email to $
20            admin_email"
21        exit 1;
22        ;;
23    esac
24 }
```

Como se pode ver, existem duas hipóteses principais na entrada desta função. Caso o output da função `uname` (função essa que imprime dados sobre o sistema operativo instalado) seja igual a *"Darwin"*, dá-se início à instalação em MacOSX pela invocação da função `install_macosx` (linha 4). Caso o output seja igual a *"Linux"* procura-se então mais informações sobre o sistema operativo instalado, com a *flag* `-v`. Das distribuições que o Linux oferece, apenas está suportada a distribuição Ubuntu.

Dependendo agora do sistema conhecido, passa-se à invocação de uma das duas seguintes funções:

```
1 function install_macosx {
2     echo "Working on a MacOSX machine" | $andlogfile
3     build_macosx
4     $portins gd2
5     install_perl_mac
6     install_perl_modules
7 }
8
9 function install_ubuntu {
10    echo "Working on a Ubuntu machine" | $andlogfile
11    build_ubuntu
12    $aptiins libgd-dev
13    install_perl_ub
14    install_perl_modules
15 }
```

8.3 Instalação MacOSX

Seguindo o raciocínio da secção anterior, a instalação em MacOSX parte da invocação da função `install_macosx` que contém um conjunto de operações, geralmente dedicadas a uma parte do projecto.

8.3.1 Bibliotecas C

O primeiro ponto de instalação refere-se à biblioteca `gd2`, necessária para o uso de um módulo Perl chamado `GD`. Essa biblioteca será indispensável para se gerar os gráficos descritos na secção 6.1.

8.3.2 Módulos Perl

Para se proceder à instalação dos módulos Perl usados neste projecto, é necessário definir a lista de módulos a instalar, como se pode ver na linha 3 da função seguinte.

```
1 function install_perl_modules {
2     perl -MCPAN -e '$CPAN->{prerequisites_policy}=follow'
3     local packages=(Makefile::Parser Parse::Yapp GD GD::Graph GD::Graph::bars GD::Graph::
4         pie Path::Class
5         Moose Term::ReadLine Term::ReadLine::Gnu Digest::SHA DBIx::Class Data::Dumper
6         )
7     if_not_exist_install_perl_modules ${packages[@]}
```

A partir desse passo, é chamada a função `if_not_exist_install_perl_modules ${packages[@]}` (linha 6) que procede à instalação dos módulos definidos.

```
1 function if_not_exist_install_perl_modules {
2     local packages=()
3
4     for module in $@; do
5         is_perl_module_installed $module
6         if [ $? -eq 1 ]; then
7             echo "$module installed";
8         else
9             echo "$module not installed, installing...";
10            packages[${#packages[@]}+1]=$module;
11        fi
12    done;
13
14    EXEC="cpan -if ${packages[@]}"
15    $EXEC
16 }
17
18 function is_perl_module_installed {
19     if [ -z "$(perl -M$1 -e 1 2>&1)" ]; then
20         return 1;
21     else
22         return 0;
23     fi
24 }
```

Para cada módulo que a função recebe, será testada para ver se se encontra já instalada, pela função `is_perl_module_installed`. Caso se encontre instalada, será removida da lista (linha 10). No final será contruído o comando de instalação e posteriormente executado (linha 15).

8.3.3 Instalação de software essencial para a ParteIII

No que diz respeito a todas as aplicações e módulos essenciais para o desenvolvimento da Parte III deste relatório, sendo eles:

- `GHCi` - Compilador de *Haskell*, juntamente com um interpretador da mesma linguagem.
- `Happy` - Um gerador de *parsers* escrito em *Haskell*.
- `Alex` - Um gerador lexical escrito em *Haskell*.
- `Language.C` - Uma biblioteca do *Haskell* para análise e geração de código C.

O seguinte pedaço de código diz respeito à sua compilação e instalação:

```

1 function build_macosx {
2     is_ghc_installed
3     if [ $? -eq 1 ]; then
4         echo "GHC is installed, I will continue..."
5         is_ghc_package_installed "happy"
6         if [ $? -eq 0 ]; then
7             echo "Happy is not installed, I will install"
8             $portins hs-happy
9         fi
10        is_ghc_package_installed "alex"
11        if [ $? -eq 0 ]; then
12            echo "Alex is not installed, I will install"
13            $portins hs-alex
14        fi
15        is_ghc_package_installed "language"
16        if [ $? -eq 0 ]; then
17            echo "Language.C is not installed, I will install"
18            build_language_c
19            cd Parser/language-c-0.3.2.1/
20            runhaskell Setup.hs install
21            cd -
22        fi
23    else
24        is_ghc_package_installed "happy"
25        if [ $? -eq 0 ]; then
26            echo "Happy is not installed, I will install"
27            $portins hs-happy
28        fi
29        is_ghc_package_installed "alex"
30        if [ $? -eq 0 ]; then
31            echo "Alex is not installed, I will install"
32            $portins hs-alex
33        fi
34        echo "GHC is not installed, I will do it for you."
35        $portins ghc
36        build_language_c
37    fi
38 }

```

Começa-se por verificar a instalação do compilador de Haskell, caso existe, verifica se existe cada uma das bibliotecas e/ou aplicações. Caso não existe, procede para a sua instalação.

8.3.4 Ruby, Rails e Gems

A presente secção encontra-se ainda inacabada...

```

1 function install_rvm_and_ruby {
2     # Install RVM
3     #bash < <(curl -s https://rvm.beginrescueend.com/install/rvm)
4     curl -s https://rvm.beginrescueend.com/install/rvm | bash
5     # Install some rubies
6     source "$HOME/.rvm/scripts/rvm"
7     rvm get head
8     rvm reload
9     rvm install 1.8.7
10    rvm --default 1.8.7
11 }
12
13 function install_rails {
14     #sudo apt-get install rubygems
15     #sudo apt-get install libxslt-dev libxml2-dev libsqlite3-dev
16     gem install rails --version 3.0.3
17     cd sample_app
18     bundle install
19     cd -
20 }

```

8.4 Instalação Ubuntu

Esta secção segue o modelo da anterior mas agora em relação ao sistema operativo Ubuntu. Percorre todas as operações que estão descritas na secção 8.2, na função `install_ubuntu`.

De realçar que a instalação de Perl e seus módulos (ver secção 8.3.2), a instalação de *software* essencial para a Partelll (ver secção 8.3.3) e a instalação de Ruby, Rails e Gems (ver secção 8.3.4), ocorre de maneira bastante semelhante à descrita nas secções anteriores, mudando apenas uma ou outra função.

8.4.1 Bibliotecas C

A instalação de bibliotecas C na *script* são chamadas à função `install_aptitude_modules`. Essa função recebe uma lista como argumentos, e para cada elemento verifica se se encontra instalado. Caso não esteja corre o comando da linha 8 com a biblioteca em questão.

```
1 function install_aptitude_modules {
2     for pkg in $@; do
3         dpkg -s $pkg
4         if [ $? -eq 0 ]; then
5             echo "module $pkg installed"
6         else
7             echo "module $pkg not installed, installing..."
8             aptitude --assume-yes install $pkg
9         fi
10    done;
11 }
```

8.5 Fase actual

Devido ao número elevado de tecnologias diferentes usadas neste sistema (decisão tomada pelo grupo de modo a aumentar o conhecimento das ferramentas existentes), decidiu-se interromper, por esta fase, a implementação da *script* de instalação, visto que teste após teste (em que cada teste demorava muito tempo), haveria sempre preciosismos essenciais para o sucesso desta *script*.

Ainda assim, não se pretende desistir da ideia de disponibilizar e facilitar o uso do sistema. Recorreu-se então ao uso da *wiki* do repositório *github* onde se encontra este projecto¹. Na *wiki* encontra-se diversa documentação à cerca do sistema, tais como:

- Listagem do conteúdo da documentação
- Instalação
- Execução de diversas componentes
- Apontadores para diversos documentos sobre métricas usados na fase de investigação.
- Algumas fórmulas de métricas implementadas
- Autores e respectivos contactos

No que diz respeito a este capítulo, o ponto Instalação, fornece-se informação sobre ferramentas essenciais ao funcionamento deste sistema, modo de instalação, *packages*, *módulos*, *gems* e *bibliotecas* necessárias.

Para finalizar este capítulo, realça-se que o processo de desenvolvimento da *script* apenas se encontra interrompido, e que a *wiki* usada é apenas um modo temporário de ultrapassar este problema e fornecer informação ao utilizador de como usar este sistema. No momento em que a equipe de desenvolvimento reflectir sobre o sistema e o uso das suas componentes, o processo de desenvolvimento volta ao activo.

¹Repositório em: <https://github.com/ulisses/Static-Code-Analyzer>

Parte III

Parser e Aplicação das métricas

9

Strafunski

Conteúdo

9.1	Explicação	47
9.1.1	Estratégias	48
9.2	Exemplos	49
9.2.1	McCabe Index—	49
9.2.2	Extracção das assinaturas das funções	50

Strafunski é uma biblioteca de programação genérica que implementa estratégias aplicadas ao paradigma funcional [LV02, LV03].

Esta biblioteca segue a infraestrutura de *Scrap your boilerplate* [LP05] do Haskell¹ e com isto pretende-se que o programador escreva código mais genérico que pode funcionar sobre vários tipos de dados e ao mesmo tempo código mais curto.

Esta abordagem requer muita investigação e nem tanto começar de imediato a programar, visto haver bastante teoria por trás destas filosofias.

9.1 Explicação

Nesta secção iremos explicar com mais algum detalhe as funções disponíveis na API do Strafunski e o que conseguimos fazer com elas.

O pacote Strafunski está dividido nos seguintes módulos:

```
Data
  Generics
    Strafunski
      StrategyLib
        Data.Generics.Strafunski.StrategyLib.ChaseImports
        Data.Generics.Strafunski.StrategyLib.ContainerTheme
        Data.Generics.Strafunski.StrategyLib.EffectTheme
        Data.Generics.Strafunski.StrategyLib.FixpointTheme
        Data.Generics.Strafunski.StrategyLib.FlowTheme
        Data.Generics.Strafunski.StrategyLib.KeyholeTheme
        Data.Generics.Strafunski.StrategyLib.MetricsTheme
        Data.Generics.Strafunski.StrategyLib.MonadFunctions
        Data.Generics.Strafunski.StrategyLib.NameTheme
        Data.Generics.Strafunski.StrategyLib.OverloadingTheme
        Data.Generics.Strafunski.StrategyLib.PathTheme
        Data.Generics.Strafunski.StrategyLib.RefactoringTheme
        Data.Generics.Strafunski.StrategyLib.StrategyInfix
        Data.Generics.Strafunski.StrategyLib.StrategyPrelude
        Data.Generics.Strafunski.StrategyLib.StrategyPrimitives
```

¹Todos os módulos que estão debaixo do Data.Generics e cujos tipos implementam as classes Data e Typeable

```
Data.Generics.Strafunski.StrategyLib.TermRep
Data.Generics.Strafunski.StrategyLib.TraversalTheme
```

É interessante explicar apenas alguns destes módulos, os que contêm as funções principais e nem tanto explicar em grande detalhe cada um deles.

Tipos de dados

Como já foi referenciado atrás para o Strafunski conseguir processar um tipo de dados criado por nós, este tem de ser instância de `Data` e de `Typeable`. Porque por exemplo a classe `Data` obriga à implementação de um catamorfismo para o nosso tipo de dados, chamado de *gfoldl* e um anamorfismo *gunfold*, entre outras funções para travessias na nossa árvore de tipos. Como nota é importante dizer que o utilizador (a pessoa que desenhou os tipos de dados) não necessita fazer a instânciação manual desta classe. Podemos recorrer ao confortável **deriving Data** que o Haskell suporta para qualquer declaração de tipos².

No que diz respeito aos tipos de dados utilizados pelo Strafunski temos dois, o **TP** para estratégias de preservação dos tipos (o tipo do output e input coincidem), e **TU** para estratégias de tipo unificados (o output é sempre de um determinado tipo independentemente do tipo de entrada), como é explicado em [LV02].

Estes tipo aos olhos do utilizador são abstractos, porque o lote de funções que nos são dadas para os trabalhar não implica o conhecimento destes.

9.1.1 Estratégias

De seguida iremos explicar no que consistem estas estratégias e como é que elas ganham forma no Strafunski, para efeitos de documentação iremos separar a explicação das funções pelo pacote a que pertencem, dentro do Strafunski.

É de notar que para cada função que existe para o tipo de dados **TU** também existe a mesma para o **TP**.

Data.Generics.Strafunski.StrategyLib.StrategyPrimitives

Para a aplicação de uma estratégia a uma instância do nosso tipo de dados *t* iremos usar:

$$\text{applyTP} :: (\text{Monad } m, \text{Term } t) \Rightarrow \text{TP } m \rightarrow t \rightarrow m \ t$$

$$\text{applyTU} :: (\text{Monad } m, \text{Term } t) \Rightarrow \text{TU } u \ m \rightarrow t \rightarrow m \ u$$

Permite adicionar estratégias:

$$\text{adhocTP} :: (\text{Monad } m, \text{Term } t) \Rightarrow \text{TP } m \rightarrow (t \rightarrow m \ t) \rightarrow \text{TP } m$$

$$\text{adhocTU} :: (\text{Monad } m, \text{Term } t) \Rightarrow \text{TU } u \ m \rightarrow (t \rightarrow m \ u) \rightarrow \text{TU } u \ m$$

Permite executar estratégias em sequência:

$$\text{seqTP} :: \text{Monad } m \Rightarrow \text{TP } m \rightarrow \text{TP } m \rightarrow \text{TP } m$$

$$\text{seqTU} :: \text{Monad } m \Rightarrow \text{TP } m \rightarrow \text{TU } u \ m \rightarrow \text{TU } u \ m$$

Para tentar usar estratégias diferentes:

$$\text{choiceTP} :: \text{MonadPlus } m \Rightarrow \text{TP } m \rightarrow \text{TP } m \rightarrow \text{TP } m$$

$$\text{choiceTU} :: \text{MonadPlus } m \Rightarrow \text{TU } u \ m \rightarrow \text{TU } u \ m \rightarrow \text{TU } u \ m$$

Aplica esta estratégia a todos os subtermos imediatos, para o **TU** os resultados são reduzidos com a função do monoid `+`:

$$\text{allTP} :: \text{Monad } m \Rightarrow \text{TP } m \rightarrow \text{TP } m$$

$$\text{allTU} :: (\text{Monad } m, \text{Monoid } u) \Rightarrow \text{TU } u \ m \rightarrow \text{TU } u \ m$$

²Internamente o GHC trata de chamar a biblioteca do DrIFT. Esta faz uma análise ao código e infere as instâncias para essas classes, mais informações em: <http://repetae.net/computer/haskell/DrIFT/>

Data.Generics.Strafunski.StrategyLib.TraversalTheme

Esta função permite aplicar a estratégia ao primeiro filho que aparecer da esquerda para a direita:

$$\text{once_tdTP}, \text{once_buTP} :: \text{MonadPlus } m \Rightarrow TP\ m \rightarrow TP\ m$$

$$\text{once_tdTU}, \text{once_buTU} :: \text{MonadPlus } m \Rightarrow TU\ u\ m \rightarrow TU\ u\ m$$

Data.Generics.Strafunski.StrategyLib.StrategyPrelude

Função identidade:

$$\text{idTP} :: \text{Monad } m \Rightarrow TP\ m$$

Função constante:

$$\text{constTU} :: \text{Monad } m \Rightarrow u \rightarrow TU\ u\ m$$

A seguinte função é uma estratégia em si que falha sempre:

$$\text{failTP} :: \text{MonadPlus } m \Rightarrow TP\ m$$

$$\text{failTU} :: \text{MonadPlus } m \Rightarrow TU\ u\ m$$

9.2 Exemplos

Agora iremos mostrar alguns exemplos de como usar a biblioteca Strafunski para conseguir processar a nossa árvore de parsing.

9.2.1 McCabe Index—

O índice de McCabe consiste, como já referimos anteriormente, num grafo orientado onde os nodos são as computações. Para se chegar a este número temos de aplicar a fórmula já descrita. Uma forma incipiente para calcular o índice de McCabe poderia ser contar o número de ifs, switches, fors e mais algumas expressões de controlo. Assim conseguimos fazer isto com o Strafunski da seguinte forma:

```
testMcCabe :: IO Int
testMcCabe = parr >= mcCabeIndex . fromRight

instance Num a => Monoid a where
    mappend = (+)
    mempty  = 0

mcCabeIndex :: Data a => a -> IO Int
mcCabeIndex = applyTU (full_tdTU isConditional)

isConditional = constTU 0 'ad hocTU' (return . test)

test (CIf _ _ _ _) = 1
test (CSwitch _ _ _ _) = 1
test (CWhile _ _ _ _) = 1
test (CFor _ _ _ _ _) = 1
test _ = 0
```

Como já explicamos anteriormente o tipo **TU** quando usado com determinados operadores exige que tenhamos uma instância de **Monoid** para o tipo de retorno. Assim fomos obrigados a definir a instância de **Monoid** para os tipos Numéricos do Haskell como a função soma. Temos consciência de que isto por si só é uma grande restrição e também temos consciência que existem alternativas para resolver este problema, como é o caso das classes **Sum** e **Mul**. No entanto não conseguimos pôr a funcionar a tempo de entrega deste relatório essas instâncias com o tipo **TU** do Strafunski.

9.2.2 Extracção das assinaturas das funções

Um outro exemplo muito interessante e bastante útil é a extracção do cabeçalho das funções em C que constam no nosso código. Isto é bastante útil para se ter uma noção do tamanho da implementação e da sua estruturação.

```
getFunctionsSign :: IO [CExtDecl]
getFunctionsSign = parr >>= return . getFunSign . fromRight

getFunSign = applyTU (once_tdTU names1)
names1 = failTU 'ad hoc TU' fromFunctionToSign

fromFunctionToSign (CFDefExt (CFunDef lCDeclSpec cDeclr _ _ nInfo))
  = [CDeclExt (CDecl lCDeclSpec [(Just $ cDeclr, Nothing, Nothing)] internalNode)]
fromFunctionToSign _ = []
```

Como podemos ver na função *fromFunctionToSign* apenas estamos a converter de um construtor do tipo **CExtDecl**, que é o **CFDefExt** para o mesmo tipo, mas com o construtor **CDeclExt**, ou seja estamos a passar o nodo da árvore de uma definição para uma declaração.

Para o caso do seguinte código:

```
void main(int argc, char **argv) {
  if(1) {}
  switch (1) {
    case 1: {
      lala(1,1);
      break;
    }
    case 1: {
      break;
    }
  }
  return 0;
}

void lala(int a, int b) {
  if(a) {}
}
```

iríamos ter o seguinte output:

```
*Main> getFunctionsSign >>= putStrLn . unlines . map (show . pretty)
void main(int argc, char * * argv);
void lala(int a, int b);
```

10

Front-end

Conteúdo

10.1 Estudo do Front-End	51
10.2 Bug no Language.C em MacOSX	54
10.3 Exemplos da árvore gerada pelo Language.C	56

Como já tínhamos explicado na primeira milestone, decidimos que o nosso sistema vai tentar suportar ao máximo avaliação de métricas sobre código C. Seria muito interessante suportar outras, mas acreditamos nesta altura que se o nosso sistema for extendido para suportar a avaliação de outras linguagens que não o C então deveríamos recorrer a ferramentas externas que fizessem algum trabalho por nós.

Relativamente ao Front-end que utilizamos, ele está feito em Haskell e foi um GSoc (Google Summer of Code) feito em 2008, chama-se Language.C¹. Este pacote de software apresenta um completo e bem testado parser e pretty printer para a definição da linguagem C99 e ainda muitas das GNU extensions.

A nossa ideia é pegar em toda a investigação e trabalho dedicado à análise e descoberta de métricas, que estão descritas no Capítulo 3, e implementa-las utilizando este Front-end.

Inicialmente decidimos partir para a exploração da linguagem (dos tipos de dados) que estavam definidos neste parser. Rápidamente encontramos a AST da linguagem C99 e assim descobrimos que a linguagem C não é assim tão grande como estaríamos à espera, como podemos ver no Apêndice A.

10.1 Estudo do Front-End

Todos os tipos deste parser estão munidos de um `NodeInfo`, que nada mais é do que a informação relativa ao ficheiro, número de linha e coluna onde apareceu esta derivação.

Um ficheiro em linguagem C99 é representado como uma lista de declarações externas que pode ser uma declaração ou uma definição de função como podemos ver na secção A.2.4.

```
data CTranslUnit = CTranslUnit [CExtDecl] NodeInfo
data CExtDecl = CDeclExt CDecl
              | CFDefExt CFunDef
              | CAsmExt CStrLit
```

Depois podemos ver que uma declaração externa pode ser então uma declaração como podemos ver na secção A.2.2, vejamos o que é isto de declaração no Haskell:

```
data CDecl = CDecl [CDeclSpec] [(Maybe CDeclr, Maybe CInit, Maybe CExpr)] NodeInfo
```

¹Mais informação em: http://trac.sivity.net/language_c

Este tipo de declarações é bastante abrangente e inclui declarações de estruturas de dados, declaração de parâmetros e tipos de dados.

Tal como a definição é uma lista de declarações C e qualificadores:

```
data CDeclSpec = CStorageSpec CStorageSpec -- ^ storage-class specifier or typedef
               | CTypeSpec    CTypeSpec    -- ^ type name
               | CTypeQual    CTypeQual    -- ^ type qualifier

data CStorageSpec = CAuto      NodeInfo    -- ^ auto
                  | CRegister NodeInfo    -- ^ register
                  | CStatic   NodeInfo    -- ^ static
                  | CExtern   NodeInfo    -- ^ extern
                  | CTypeDef   NodeInfo    -- ^ typedef
                  | CThread   NodeInfo    -- ^ GNUC thread local storage

data CTypeSpec = CVoidType    NodeInfo
                | CCharType   NodeInfo
                | CShortType   NodeInfo
                | CIntType     NodeInfo
                | CLongType    NodeInfo
                | CFloatType   NodeInfo
                | CDoubleType  NodeInfo
                | CSignedType  NodeInfo
                | CUnsigType   NodeInfo
                | CBoolType    NodeInfo
                | CComplexType NodeInfo
                | CSUType      CStructUnion NodeInfo -- ^ Struct or Union specifier
                | CEnumType    CEnum         NodeInfo -- ^ Enumeration specifier
                | CTypeDef     Ident         NodeInfo -- ^ Typedef name
                | CTypeOfExpr  CExpr         NodeInfo -- ^ typeof(expr)@
                | CTypeOfType  CDecl         NodeInfo -- ^ typeof(type)@
```

CTypeQual diz respeito a qualificadores de tipos, por exemplo:

- const
- volatile
- restrict
- inline
- __attribute__

```
data CTypeQual = CConstQual NodeInfo
               | CVolatQual NodeInfo
               | CRestrQual NodeInfo
               | CInlineQual NodeInfo
               | CAttrQual   CAttr
```

E ainda uma lista de triplos de Maybes (Maybe CDeclr, Maybe CInit, Maybe CExpr). Esta lista como tem 3 Maybes pode apresentar várias formas, estas formas definem o tipo de declaração que temos:

Toplevel declarations Uma declaração na forma (Just declr, init?, Nothing)

Structure declarations Declarações na forma (Just declr, Nothing, size?)

Parameter declarations Declarações na forma (Just declr, Nothing, Nothing) ou (Nothing, Nothing, Just size)

Toplevel declarations Declarações na forma (Just declr, Nothing, Nothing)

Tempo agora para mostrar o que são estas definições CDeclr, CInit e CExpr.

```
data CDeclr = CDeclr (Maybe Ident) [CDerivedDeclr] (Maybe CStrLit) [CAttr] NodeInfo
```

CDeclr corresponde a declarações de funções ou tipos em C. Por exemplo:

```
int x;
CDeclr "x" []

const int * const * restrict x;
CDeclr "x" [CPtrDeclr [restrict], CPtrDeclr [const]]
```

```
int* const f();
CDeclr "f" [CFunDeclr [],CPtrDeclr [const]]
```

Ou seja, primeiro temos um possível identificador, seguido de uma lista de declaração derivadas seguido de um string literal.

Vejamos agora a definição de CDerivedDeclr:

```
data CDerivedDeclr = CPtrDeclr [CTypeQual] NodeInfo
  -- ^ Pointer declarator @CPtrDeclr tyquals declr@
  | CArrDeclr [CTypeQual] (CArrSize) NodeInfo
  -- ^ Array declarator @CArrDeclr declr tyquals size - expr?@
  | CFunDeclr (Either [Ident] ([CDecl],Bool)) [CAttr] NodeInfo
  -- ^ Function declarator @CFunDeclr declr (old-style - params | new-style -
  -- params) c - attrs@
```

Isto basicamente corresponde a uma atribuição de um pointer (CPtrDeclr attrs), uma atribuição do resultado da computação de uma função (CFunDeclr attrs) e atribuição de um array de elementos (CArrDeclr attrs).

CAttr é simplesmente a utilização da anotação `__attribute__`

Voltando ao triplo de Maybes, temos ainda como segundo elemento CInit que correspondem a inicializações em C.

```
data CInit = CInitExpr CExpr
  NodeInfo      -- ^ assignment expression
  | CInitList CInitList
  NodeInfo      -- ^ initialization list (see 'CInitList')
```

Estes inicializadores podem ser uma de duas coisas, ou um assignment a uma expressão ou então uma inicialização de uma lista, rodeada de parêntices de bloco.

Usemos então o caso das inicializações para introduzir as CExpr.

```
data CExpr = CComma [CExpr] -- comma expression list, n >= 2
  NodeInfo
  | CAssign CAssignOp -- assignment operator
  CExpr -- l-value
  CExpr -- r-value
  NodeInfo
  | CCond CExpr -- conditional
  NodeInfo
  | CBinary CBinaryOp -- binary operator
  CExpr -- lhs
  CExpr -- rhs
  NodeInfo
  | CCast CDecl -- type name
  CExpr
  NodeInfo
  | CUnary CUnaryOp -- unary operator
  CExpr
  NodeInfo
  | CSizeofExpr CExpr
  NodeInfo
  | CSizeofType CDecl -- type name
  NodeInfo
  | CAlignofExpr CExpr
  NodeInfo
  | CAlignofType CDecl -- type name
  NodeInfo
  | CComplexReal CExpr -- real part of complex number
  NodeInfo
  | CComplexImag CExpr -- imaginary part of complex number
  NodeInfo
  | CIndex CExpr -- array
  CExpr -- index
  NodeInfo
  | CCall CExpr -- function
  [CExpr] -- arguments
  NodeInfo
  | CMember CExpr -- structure
  Ident -- member name
  Bool -- deref structure? (True for '->')
  NodeInfo
  | CVar Ident -- identifier (incl. enumeration const)
  NodeInfo
  | CConst CConst -- ^ integer, character, floating point and string
  constants
  | CCompoundLit CDecl
  CInitList -- type name & initialiser list
  NodeInfo -- ^ C99 compound literal
  | CStatExpr CStat NodeInfo -- ^ GNU C compound statement as expr
```

```
| CLabAddrExpr Ident NodeInfo -- ^ GNU C address of label
| CBuiltinExpr CBuiltin      -- ^ builtin expressions, see 'CBuiltin'
```

Esta definição é tão completa que abrange também algumas extensões feitas à linguagem C pelo, conhecidas como GCC Extensions, como por exemplo as: `alignof`, `__real`, `__imag`, (`{ stmt-expr }`).

Terminada a explicação extensa de `CDecl`, continuamos a explicação de `CExtDecl` e passamos à definição de `CFunDef`, ou seja uma definição de função como podemos ver em `function-definition` na secção A.2.4.

```
data CFunDef = CFunDef [CDeclSpec]      -- type specifier and qualifier
               CDeclr                  -- declarator
               [CDecl]                  -- optional declaration list
               CStat                    -- compound statement
               NodeInfo
```

Esta definição usa uma lista de especificadores de tipo e de quantificação, uma declaração, uma lista adicional de declarações e um C Statement. Como já explicamos todos, passamos para a explicação do `CStat`.

Esta definição é capaz de ser a mais familiar e fácil de perceber na medida em que é aqui que estão definidos os statements C que caracterizam esta linguagem:

```
data CStat = CLabel Ident CStat [CAttr] NodeInfo -- ^ An (attributed) label followed by a
statement
| CCase CExpr CStat NodeInfo                    -- ^ A statement of the form @case expr :
  stmt@
| CCases CExpr CExpr CStat NodeInfo             -- ^ A case range of the form @case lower
  ... upper : stmt@
| CDefault CStat NodeInfo                       -- ^ The default case @default : stmt@
| CExpr (Maybe CExpr) NodeInfo                 -- ^ A simple statement, that is in C: evaluating an expression with side-effects
-- and discarding the result.
| CCompound [Ident] [CBlockItem] NodeInfo        -- ^ compound statement @CCompound
  localLabels blockItems at@
| CIf CExpr CStat (Maybe CStat) NodeInfo        -- ^ conditional statement @CIf ifExpr
  thenStmt maybeElseStmt at@
| CSwitch CExpr CStat NodeInfo                  -- ^ switch statement @CSwitch selectorExpr switchStmt@, where @switchStmt@
  usually includes
  -- /case/, /break/ and /default/ statements
| CWhile CExpr CStat Bool NodeInfo              -- ^ while or do-while statement @CWhile
  guard stmt isDoWhile at@
| CFor (Either (Maybe CExpr) CDecl)
  (Maybe CExpr)
  (Maybe CExpr)
  CStat
  NodeInfo
  -- ^ for statement @CFor init expr -2 expr -3 stmt@, where @init@ is either a
  declaration or
  -- initializing expression
| CGoto Ident NodeInfo                         -- ^ goto statement @CGoto label@
| CGotoPtr CExpr NodeInfo                     -- ^ computed goto @CGotoPtr labelExpr@
| CCont NodeInfo                              -- ^ continue statement
| CBreak NodeInfo                             -- ^ break statement
| CReturn (Maybe CExpr) NodeInfo              -- ^ return statement @CReturn returnExpr@
| CAsm CAsmStmt NodeInfo                      -- ^ assembly statement
```

Aqui podemos então ver as Labels, Cases, statements de default dos Cases, expressões, ifs, switches, whiles, fors, gotos, continues, breaks, retuns e instruções assembly.

Grande parte deste Front-End, Language.C, fica assim explicado. Achamos que explicar os detalhes sobre o `CStat` não seria necessário, visto já termos detalhado os mais importantes.

10.2 Bug no Language.C em MacOSX

Depois de explorar e usar, sempre com exemplos pequenos este front-end descobrimos que se os ficheiros C submetidos fizerem uso de bibliotecas externas teríamos um problema ao correr o avaliador de software em MacOSX. Este problema já é conhecido pela equipa que desenvolve o Language.C².

Embora o nosso sistema final corra numa máquina com Linux, o desenvolvimento desta aplicação de avaliação será feito em MacOSX e assim isto confere um problema nesta fase. Mesmo assim também

²Conforme se pode ver neste ticket: http://trac.sivity.net/language_c/ticket/2

foi interessante descobrir este bug porque pretendemos que a nossa aplicação corra no máximo de plataformas possíveis. Imagine-se o caso em que queremos rapidamente instalar numa máquina o nosso sistema para demonstração ou montar um concurso rápido, sem que seja necessário um servidor.

Este bug prende-se com o facto do Language.C na sua versão mais recente, 3.2.1, não suportar a notação de `__BLOCKS__` que as bibliotecas do MacOSX têm.

Imagine-se o seguinte código Haskell a usar Language.C:

```
module Main where

import Language.C
import Language.C.System.GCC
import Language.C.Data.Ident
import System.Environment

process :: String -> IO ()
process file = do
    stream <- parseCFile (newGCC "gcc") Nothing [] file
    case stream of
        ( Left error ) -> print error
        ( Right cprog ) -> print "OK"

main :: IO ()
main = do
    files <- getArgs
    mapM_ process files
```

O comportamento deste programa seria receber n ficheiros e a cada um deles passar ao pre-processor do GCC, seguidamente é construída a árvore de parsing que fica guardada em *stream* e posteriormente verificamos se tivemos algum erro, se sim reportamos o erro para o *stdout* se não então imprimimos "OK" para o *stdout*.

Ao alimentar este programa com o seguinte pedaço de código:

```
#include <stdlib.h>

void main(int argc, char **argv) {
    malloc(10);
    return 0;
}
```

acontece o seguinte erro:

```
[ulissesaraujocosta@macclisses:Parser]-$ ./main main.c
/usr/include/stdlib.h:272: (column 20) [ERROR] >>> Syntax Error !
Syntax error !
The symbol '^' does not fit here.
```

Ao inspecionar a linha 272 do `stdlib.h` verificamos a presença do seguinte código:

```
#ifdef __BLOCKS__
int atexit_b(void (^)(void));
void *bsearch_b(const void *, const void *, size_t,
size_t, int (^)(const void *, const void *));
#endif /* __BLOCKS__ */
```

O problema reside exactamente aqui, ou seja o Language.C não consegue descobrir esta notação. Este problema parece já ter sido aberto há alguns meses e está previsto para correcção na próxima versão 4.0 do Language.C.

Afim de tentarmos mesmo assim conseguir usar este front-end lembramos-nos de proibir o pre-processor do GCC para definir a variável `__BLOCKS__`. Assim sendo temos que incluir no nosso código a injeção de uma linha de código que fará com que todos os ficheiros C processados pelo nosso programa apresentem a desactivação desta variável:

```
#undef __BLOCKS__
#include <stdlib.h>

void main(int argc) {
    malloc(10);
    return 0;
}
```

10.3 Exemplos da árvore gerada pelo Language.C

A árvore de parsing que é criada pelo Language.C tem 84 tipos de nodos (constructores de tipos), o que faz dela uma árvore pequena, fácil de ler, de compreender e de começar rapidamente a trabalhar a informação contida.

Infelizmente o pacote Language.C não vem com as intâncias *Show* para os tipos, o que faz com que para vermos a nossa árvore tenhamos que estar constantemente a ler os tipos de dados e assim guiar às escuras.

Assim sendo, editamos os ficheiros todos e derivamos a classe *Show* para todos os tipos³.

Agora tentamos explicar com mais detalhe relativamente à secção 10.1 os tipos de dados gerados pelo Language.C.

Imaginemos um ficheiro com o nome `main.c` cujo conteúdo é o seguinte:

```
void lala(int a, int b);
```

Ou seja, apenas a declaração de uma função, depois de executar a função de parsing obtemos a seguinte árvore:

```
CTranslUnit
  [CDeclExt
    (CDecl
      [CTypeSpec (CVoidType (NodeInfo("main.c", 2, 1) (Name { nameId = 1 }))))]
      [ (Just
        (CDeclr (Just ' lala ')
          [CFunDeclr (Right
            [CDecl [CTypeSpec (CIntType (NodeInfo("main.c",2,11) (Name {nameId = 4}))))]
            [(Just (CDeclr (Just 'a') [] Nothing [] (NodeInfo("main.c", 2, 15)
              (Name { nameId = 5 }))))]
              ,Nothing, Nothing)] (NodeInfo("main.c", 2, 11) (Name { nameId = 6 })))
            ,CDecl [CTypeSpec (CIntType (NodeInfo("main.c", 2, 18) (Name { nameId = 8 }))))]
            [(Just (CDeclr (Just ' b ') [] Nothing [] (NodeInfo ("main.c",2,22)
              (Name {nameId = 9}))))]
              ,Nothing,Nothing)]
              (NodeInfo ("main.c",2,18) (Name {nameId = 10})),False))
            [] (NodeInfo ("main.c",2,10) (Name {nameId = 11}))
            ]
            Nothing [] (NodeInfo ("main.c",2,6) (Name {nameId = 2}))
          )
        ,Nothing
        ,Nothing
      )
    ] (NodeInfo ("main.c",2,1) (Name {nameId = 12}))))]
```

Como podemos ver, embora a árvore de parsing seja bastante pequena, uma simples declaração de uma função pode tomar um tamanho grande. Mesmo assim achamos bastante simples, principalmente depois de ler a especificação da AST do C.

Agora imaginemos que temos a seguinte função vazia em C:

```
void lala(int a, int b) { }
```

A árvore que iríamos obter seria:

```
CFDefExt
  (CFunDef
    [CTypeSpec
      (CVoidType (NodeInfo ("main.c",28,1) (Name { nameId = 71 })))
    ]
    (CDeclr
      (Just 'lala')
      [CFunDeclr
        (Right
          [CDecl
            [CTypeSpec (CIntType (NodeInfo("main.c", 28, 11) (Name { nameId = 74 }))))]
            [(Just (CDeclr (Just ' a ') [] Nothing
              [] (NodeInfo ("main.c",28,15) (Name {nameId = 75})))
              ,Nothing
              ,Nothing)
            ] (NodeInfo ("main.c",28,11) (Name {nameId = 76}))
          ,CDecl

```

³A nossa versão do Language.C está disponível no nosso pacote de software

10.3. EXEMPLOS DA ÁRVORE GERADA PELO LANGUAGE.C

```
[CTypeSpec (CIntType (NodeInfo ("main.c",28,18) (Name {nameId = 78})))][
  [(Just (CDeclr (Just 'b')[] Nothing[]
    (NodeInfo("main.c", 28, 22) (Name { nameId = 79 })))
    ,Nothing
    ,Nothing)] (NodeInfo("main.c", 28, 18) (Name { nameId = 80 }))), False
  )
  )[] (NodeInfo("main.c", 28, 10) (Name { nameId = 81 })))
  Nothing[] (NodeInfo("main.c", 28, 6) (Name { nameId = 72 })))[]
(CCompound[] [] (NodeInfo("main.c", 28, 25) (Name { nameId = 82 })))
(NodeInfo("main.c", 28, 1) (Name { nameId = 83 })))
```


11

Métricas Implementadas

Conteúdo

11.1 Métricas calculadas	58
11.2 Tipo de dados das métricas	59
11.3 API de métricas	59
11.4 Exemplos de cálculo de métricas	60

Neste capítulo iremos falar sobre as métricas que implementamos, descrevê-las e explicar em detalhe como estas podem ser utilizadas em conjunto com a API que desenvolvemos para no futuro se poder extrair mais informação a partir do trabalho que já foi realizado até aqui.

11.1 Métricas calculadas

As métricas que o nosso sistema actual permite calcular estão divididas por grupos conforme o seu nível de actuação, assim sendo temos métricas relativas à complexidade, métricas relativas às linhas de ficheiros, linhas de comentários, funções e includes.

Mostramos agora todas as métricas que o nosso sistema consegue calcular até agora:

- Grafo de includes do sistema e de cada ficheiro
- Nr linhas de comentários (que não são pedaços de código comentados)
- Densidade de comentários
- Index de McCabe
- NLOC (nr de linhas do pretty print)
- Nr de linhas físicas
- Clones por bloco
- Assinaturas de funções e nomes de funções

Todas estas métricas podem ser aplicadas tanto ao projecto total, como a um ficheiro ou até a apenas uma única função, dependendo se faz sentido ou não calcular essa métrica para vários níveis de detalhe.

Por exemplo, o index de mccabe é possível e faz sentido calcular para os três níveis: projecto, ficheiro e

função.

Demos especial enfoque á contagem de número de linhas de comentários, sua densidade porque achamos que estes traduzem um bom software ou um sistema útil para o desenvolvimento por pessoas externas.

Por clone por bloco temos em consideração várias linhas contíguas e verificamos se essas linhas ocorrem em mais algum ficheiro. Isto é uma forma mais precisa para dizer se um pedaço de código foi clonado.

11.2 Tipo de dados das métricas

De seguida mostramos as declarações em Haskell dos tipos de dados que suportam o conjunto a que chamamos de *Metrics*.

```
type Metrics = M.Map MetricName MetricValue
type MetricName = (String, Maybe FileSrc, Maybe FunctionName)
data MetricValue =
  | Num Double
  | Clone (M.Map FileDst [(Ocurrency, LineSrc, LineDst)])
  | Includes ([SystemIncludes],[Includes])
  | FunSig [FunSignature]
  | Graphviz DotFile
  | GraphvizProject DotFile
```

Escolhemos um *Map* como tipo principal para guardar todas as nossas métricas por este apresentar a flexibilidade e rapidez desejada num sistema deste tipo.

Como foi dito na secção anterior há métricas que faz sentido aplicar ao projecto todo, enquanto que outras apenas a ficheiros ou at e funções, conseguimos essa expressividade através de um triplo cujo primeiro membro é obrigatório e os seguintes não o são. Deixamos assim ao critério de quem implementa as métricas se pretende aplicar essa métrica apenas ao projecto ou a mais alguma coisa em concreto. Este tipo será a chave n nosso *Map*, sendo que o valor é o valor da métrica em si.

11.3 API de métricas

Mostramos agora algumas das funções mais interessantes que compõe a API para manipulação de métricas que desenvolvemos.

Para criar uma métrica apartir do zero, podemos usar a seguinte função:

```
(>.>) :: Metrics -> (MetricName, MetricValue) -> Metrics
m >.> (mn, mv) =
  case M.lookup mn m of
    Nothing -> m'
    (Just mv') -> if mv' == mv then m else m'
  where m' = M.insert mn mv m
```

Esta função é quase sempre usada aquando da criação de métricas novas, um exemplo do seu uso poderia ser: *emptyMetrics >>> (("mccbaIndex", Nothing, Nothing), Num 10)*. Estamos assim a dizer para criar uma métrica nova cujo seu nome é *mccbaIndex* e aplicamos esta métrica ao pacote de software todo e ainda dizemos que o seu valor é o número 10.

Uma outra função muito interessante é a função de união de métricas.

```
(>+>) :: Metrics -> Metrics -> Metrics
m1 >+> m2 = M.union m1 m2
```

Esta função permite-nos por exemplo, pegar numa lista de métricas e facilmente converter tudo numa única métrica.

```
concatMetrics :: [Metrics] -> Metrics
concatMetrics = foldl1 (>+>) emptyMetrics
```

Uma outra função é o típico *foldr* sobre o nosso conjunto de métricas.

```
foldrM :: (MetricName -> MetricValue -> c -> c) -> c -> Metrics -> c
foldrM f s = M.foldrWithKey f s
```

Esta função ajudou-nos bastante no caso concreto de produzir um documento \LaTeX , visto que muito facilmente dizíamos o que pretendíamos fazer a cada uma das métricas que estavam dentro do nosso conjunto. Mostramos agora um pequeno exemplo extraído desse mesmo código onde extraímos a informação da métrica que cria um ficheiro graphviz apartir dos includes.

```
...
foldrM step noop m
  where step k v r = "\\begin{dot2tex}[]"
    >> (fromString $ fromGraphvizP v)
    >> "\\end{dot2tex}"
    // r
    fromGraphvizP (GraphvizProject 1) = 1
```

Uma outra função muito útil para nós foi a seguinte:

```
getMetricsFrom :: (a -> IO Metrics) -> [a] -> IO Metrics
getMetricsFrom f l =
  forkMapM f l >>=
    return . concatMetrics . map (either (const emptyMetrics) id)
```

Graças a esta função conseguimos tornar o nosso calculador muito escalável. Esta função pode ser vista como um *map* monadico que paraleliza as várias computações. Ou seja, pegamos numa função e numa lista e executamos esta função em paralelo para os vários elementos da lista. Conseguimos ganhos de performance graças a esta função.

Um exemplo do seu uso, para extrair a métrica index de mccabe de uma lista de ficheiros.

```
getListOfCFiles :: FilePath -> IO [FilePath]
getTreeFromFile :: FilePath -> [FilePath] -> IO [(FilePath, CTranslUnit)]
mccabePerFun :: (FilePath, CTranslUnit) -> IO Metrics

getListOfCFiles fp >>= getTreeFromFile fp >>= getMetricsFrom mccabePerFun
```

11.4 Exemplos de cálculo de métricas

Agora mostramos alguns exemplos de métricas calculadas por nós.

Para calcular o index de mccabe, decidimos abordar a questão não pela sua definição formal, mas sim por outras definições alternativas que encontramos. Assim, fomos por uma solução muito simples de implementar com o *Strafunski* que apenas conta o número de ocorrências para determinados construtores de tipos.

Estamos a dizer para encontrar todas as ocorrências de *Ifs*, *Switches*, *Whiles*, *Fors* e ainda operadores *and* e *or* e adicionar o valor 1 ao resultado.

Quando usamos esta métrica para funções apenas passamos como input uma única função de cada vez e não o programa todo. Isso é muito fácil de obter, visto que o Language.C representa um ficheiro como uma lista de funções.

```
mccabeIndex :: Data a => a -> IO Int
mccabeIndex = applyTU (full_tdtU typesOfInstr)

typesOfInstr = constTU 0
  'ad hocTU' loop
  'ad hocTU' binaryOp
loop :: CStat -> IO Int
loop = return . loop_
  where loop_ (CIf _ _ _) = 1
    loop_ (CSwitch _ _ _) = 1
    loop_ (CWhile _ _ _) = 1
    loop_ (CFor _ _ _ _) = 1
    loop_ _ = 0
binaryOp :: CBinaryOp -> IO Int
binaryOp = return . binaryOp_
  where binaryOp_ CLndOp = 1
    binaryOp_ CLorOp = 1
    binaryOp_ _ = 0
```

Mostramos ainda a implementação de uma métrica bastante simples, que conta o número de linhas (pretty print) que uma determinada função tem. Esta métrica é importante para evitar contar o número de linhas directamente do ficheiro e evitamos assim o estilo de programação de cada programador.

```
ncloc :: (FilePath,CTranslUnit) -> IO Metrics
ncloc (file,tree) =
  let len = (length . filter (not . null) . lines . show . pretty) tree
  in return $ emptyMetrics
    >.> (("ncloc",Just file,Nothing),Num $ fromIntegral len)
```

12

Conclusão e Trabalho Futuro

Ao longo de toda primeira fase modelamos os vários aspectos do nosso sistema. Na segunda fase partimos para a implementação do sistema e focamos também a nossa atenção na exploração de um frontend para a linguagem C. Na terceira fase deu-se continuidade à implementação de uma interface em linha de comandos que fizesse algumas das tarefas que se podem fazer através da interface Web. Também se aprofundou conhecimentos sobre o frontend *Language.C*.

Toda a modelação do sistema realizada, foi importante, devido à visão mais alargada que nos deu do problema e da sua resolução. A implementação correu de forma tranquila, onde apenas pequenos ajustes se realizaram, em relação ao pensado primeira fase. O trabalho relativo à importação de enunciados e tentativas em formato XML, que tinha ficado incompleto na fase I, foi na fase II completado.

Na fase II foi também dado início ao desenvolvimento de uma interface em linha de comandos escrita em Perl, que alarga a acessibilidade à aplicação, deixando do acesso à mesma estar dependente de um browser.

Ainda nesta fase a exploração do frontend *Language.C* também deu os primeiros passos.

No fim da terceira fase, o frontend escolhido foi completamente dominado e partiu-se para a exploração de técnicas de programação genérica que implementasse estratégias de percorrer árvores de parsing, para que se conseguisse extrair informação sobre o código que recebemos de input.

O sistema, agora na última fase, está apto a receber as soluções dos utilizadores, tratá-las, apresentar resultados e fazer detecção de clonning. Preparou-se também o sistema para gerar e permitir o download de um relatório em pdf sobre os resultados obtidos nas tentativas submetidas pelos utilizadores, e também um relatório sobre as métricas extraídas pelo analisador criado. A aplicação Web ficou assim terminada.

O analisador de métricas desenvolvido implementou todas as métricas pretendidas. Além disso foi criada uma API de metrics e um detector de clones. Este analisador gera um relatório em pdf, que além das métricas também apresenta os possíveis clones encontrados.

O output do analisador deixa assim ao dispor do utizador tudo o que precisas para extrair uma noção de qualidade do software submetido.

Bibliografia

- [Ker88] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.
- [LP05] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press, September 2005.
- [LV02] Ralf Lämmel and Joost Visser. Design patterns for functional strategic programming. In *Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, RULE '02, pages 1–14, New York, NY, USA, 2002. ACM.
- [LV03] R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proc. of Practical Aspects of Declarative Programming (PADL'03)*, volume 2562 of *LNCS*, pages 357–375. Springer-Verlag, January 2003.
- [McC76] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.
- [WMW96] Arthur H. Watson, Thomas J. McCabe, and Dolores R. Wallace. Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric. In *U.S. Department of Commerce/National Institute of Standards and Technology*, 1996.



AST da Linguagem C99

Esta definição de AST é parte da original descrita no Apendix A do [Ker88], excluimos algumas definições lexicas como por exemplo a definição de digito ou de alfanumerico por ser fácil perceber o que se quer dizer com isso e para não alongar muito mais este apêndice.

A.1 Gramática Lexica

A.1.1 Elementos Lexicos

```
token:
    keyword
    identifier
    constant
    string-literal
    punctuator
preprocessing-token:
    header-name
    identifier
    pp-number
    character-constant
    string-literal
    punctuator
    each non-white-space character that cannot be one of the above
```

A.1.2 Keywords

```
keyword:
    one of, auto, break, case, char, const
    continue, default, do, double, else
    enum, extern, float, for, goto, if
    inline, int, long, register, restrict
    return, short, signed, sizeof, static
    struct, switch, typedef, union
    unsigned, void, volatile, while
    _Bool, _Complex, _Imaginary
```

A.1.3 Identificadores

```
identifier:
    identifier-nondigit
    identifier identifier-nondigit
    identifier digit
identifier-nondigit:
    nondigit
    universal-character-name
    other implementation-defined characters
```

A.1.4 Universal character names

```

universal-character-name:
    \u hex-quad
    \U hex-quad hex-quad
hex-quad:
    hexadecimal-digit hexadecimal-digit
    hexadecimal-digit hexadecimal-digit

```

A.1.5 Constantes

```

constant:
    integer-constant
    floating-constant
    enumeration-constant
    character-constant
integer-constant:
    decimal-constant integer-suffixopt
    octal-constant integer-suffixopt
    hexadecimal-constant integer-suffixopt
decimal-constant:
    nonzero-digit
    decimal-constant digit
octal-constant:
    0
    octal-constant octal-digit
hexadecimal-constant:
    hexadecimal-prefix hexadecimal-digit
    hexadecimal-constant hexadecimal-digit
hexadecimal-prefix: one of
    0x 0X
integer-suffix:
    unsigned-suffix long-suffixopt
    unsigned-suffix long-long-suffix
    long-suffix unsigned-suffixopt
    long-long-suffix unsigned-suffixopt
unsigned-suffix: one of
    u U
long-suffix: one of
    l L
long-long-suffix: one of
    ll LL
floating-constant:
    decimal-floating-constant
    hexadecimal-floating-constant
decimal-floating-constant:
    fractional-constant exponent-partopt floating-suffixopt
    digit-sequence exponent-part floating-suffixopt
hexadecimal-floating-constant:
    hexadecimal-prefix hexadecimal-fractional-constant
    binary-exponent-part floating-suffixopt
    hexadecimal-prefix hexadecimal-digit-sequence
    binary-exponent-part floating-suffixopt
fractional-constant:
    digit-sequenceopt . digit-sequence
    digit-sequence .
exponent-part:
    e signopt digit-sequence
    E signopt digit-sequence
sign: one of
    +
digit-sequence:
    digit
    digit-sequence digit
hexadecimal-fractional-constant:
    hexadecimal-digit-sequenceopt .
    hexadecimal-digit-sequence
    hexadecimal-digit-sequence .
binary-exponent-part:
    p signopt digit-sequence
    P signopt digit-sequence
hexadecimal-digit-sequence:
    hexadecimal-digit
    hexadecimal-digit-sequence hexadecimal-digit
floating-suffix: one of
    f l F L
enumeration-constant:
    identifier
character-constant:
    ' c-char-sequence '
    L' c-char-sequence '
c-char-sequence:
    c-char

```



```
c-char-sequence c-char
c-char:
    any member of the source character set except
    the single-quote ', backslash \, or new-line character
    escape-sequence
escape-sequence:
    simple-escape-sequence
    octal-escape-sequence
    hexadecimal-escape-sequence
    universal-character-name
simple-escape-sequence: one of
    \' \" \? \\
    \a \b \f \n \r \t
    \v
octal-escape-sequence:
    \ octal-digit
    \ octal-digit octal-digit
    \ octal-digit octal-digit octal-digit
hexadecimal-escape-sequence:
    \x hexadecimal-digit
    hexadecimal-escape-sequence hexadecimal-digit
```

A.1.6 String literals

```
string-literal:
    " s-char-sequenceopt "
    L" s-char-sequenceopt "
s-char-sequence:
    s-char
    s-char-sequence s-char
s-char:
    any member of the source character set except
    the double-quote ", backslash \, or new-line character
    escape-sequence
```

A.1.7 Header names

```
header-name:
    < h-char-sequence >
    " q-char-sequence "
h-char-sequence:
    h-char
    h-char-sequence h-char
h-char:
    any member of the source character set except
    the new-line character and >
q-char-sequence:
    q-char
    q-char-sequence q-char
q-char:
    any member of the source character set except
    the new-line character and "
```

A.2 Phrase structure grammar

A.2.1 Expressions

```
primary-expression:
    identifier
    constant
    string-literal
    ( expression )
postfix-expression:
    primary-expression
    postfix-expression [ expression ]
    postfix-expression ( argument-expression-listopt )
    postfix-expression . identifier
    postfix-expression -> identifier
    postfix-expression ++
    postfix-expression -( type-name ) { initializer-list }
    ( type-name ) { initializer-list , }
argument-expression-list:
    assignment-expression
    argument-expression-list , assignment-expression
unary-expression:
    postfix-expression
    ++ unary-expression
    -- unary-expression
    unary-operator cast-expression
```

```

        sizeof unary-expression
        sizeof ( type-name )
unary-operator: one of
    & * + - ~
    !
cast-expression:
    unary-expression
    ( type-name ) cast-expression
multiplicative-expression:
    cast-expression
    multiplicative-expression * cast-expression
    multiplicative-expression / cast-expression
    multiplicative-expression % cast-expression

additive-expression:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression
shift-expression:
    additive-expression
    shift-expression << additive-expression
    shift-expression >> additive-expression
relational-expression:
    shift-expression
    relational-expression
    relational-expression
    relational-expression
    relational-expression
    <
    >
    <=
    >=
    shift-expression

equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression
AND-expression:
    equality-expression
    AND-expression & equality-expression
exclusive-OR-expression:
    AND-expression
    exclusive-OR-expression ^ AND-expression
inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression | exclusive-OR-expression
logical-AND-expression:
    inclusive-OR-expression
    logical-AND-expression && inclusive-OR-expression
logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || logical-AND-expression
conditional-expression:
    logical-OR-expression
    logical-OR-expression ? expression : conditional-expression
assignment-expression:
    conditional-expression
    unary-expression assignment-operator assignment-expression
assignment-operator: one of
    = *= /= %= += -=
    <<= >>= &= ^= |=

expression:
    assignment-expression
    expression , assignment-expression
constant-expression:
    conditional-expression

```

A.2.2 Declarations

```

declaration:
    declaration-specifiers init-declarator-listopt ;
declaration-specifiers:
    storage-class-specifier declaration-specifiersopt
    type-specifier declaration-specifiersopt
    type-qualifier declaration-specifiersopt
    function-specifier declaration-specifiersopt
init-declarator-list:
    init-declarator
    init-declarator-list , init-declarator
init-declarator:
    declarator

```

```

        declarator = initializer
storage-class-specifier:
    typedef
    extern
    static
    auto
    register

type-specifier:
    void, char, short, int, long
    float, double, signed, unsigned
    _Bool, _Complex, struct-or-union-specifier
    enum-specifier, typedef-name, *

struct-or-union-specifier:
    struct-or-union identifieropt { struct-declaration-list }
    struct-or-union identifier
struct-or-union:
    struct
    union
struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration
struct-declaration:
    specifier-qualifier-list struct-declarator-list ;
specifier-qualifier-list:
    type-specifier specifier-qualifier-listopt
    type-qualifier specifier-qualifier-listopt
struct-declarator-list:
    struct-declarator
    struct-declarator-list , struct-declarator
struct-declarator:
    declarator
    declaratoropt : constant-expression

enum-specifier:
    enum identifieropt { enumerator-list }
    enum identifieropt { enumerator-list , }
    enum identifier
enumerator-list:
    enumerator
    enumerator-list , enumerator
enumerator:
    enumeration-constant
    enumeration-constant = constant-expression
type-qualifier:
    const
    restrict
    volatile
function-specifier:
    inline
declarator:
    pointeropt direct-declarator
direct-declarator:
    identifier
    ( declarator )
    direct-declarator [ type-qualifier-listopt assignment-expressionopt ]
    direct-declarator [ static type-qualifier-listopt assignment-expression ]
    direct-declarator [ type-qualifier-list static assignment-expression ]
    direct-declarator [ type-qualifier-listopt * ]
    direct-declarator ( parameter-type-list )
    direct-declarator ( identifier-listopt )
pointer:
    * type-qualifier-listopt
    * type-qualifier-listopt pointer
type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier
parameter-type-list:
    parameter-list
    parameter-list , ...

parameter-list:
    parameter-declaration
    parameter-list , parameter-declaration
parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declaratoropt
identifier-list:
    identifier
    identifier-list , identifier
type-name:
    specifier-qualifier-list abstract-declaratoropt
abstract-declarator:

```

```

        pointer
        pointeropt direct-abstract-declarator
direct-abstract-declarator:
    ( abstract-declarator )
    direct-abstract-declaratoropt [ type-qualifier-listopt
    assignment-expressionopt ]
    direct-abstract-declaratoropt [ static type-qualifier-listopt
    assignment-expression ]
    direct-abstract-declaratoropt [ type-qualifier-list static
    assignment-expression ]
    direct-abstract-declaratoropt [ * ]
    direct-abstract-declaratoropt ( parameter-type-listopt )

typedef-name:
    identifier
initializer:
    assignment-expression
    { initializer-list }
    { initializer-list , }
initializer-list:
    designationopt initializer
    initializer-list , designationopt initializer
designation:
    designator-list =

designator-list:
    designator
    designator-list designator
designator:
    [ constant-expression ]
    . identifier

```

A.2.3 Statements

```

statement:
    labeled-statement
    compound-statement
    expression-statement
    selection-statement
    iteration-statement
    jump-statement
labeled-statement:
    identifier : statement
    case constant-expression : statement
    default : statement
compound-statement:
    { block-item-listopt }
block-item-list:
    block-item
    block-item-list block-item
block-item:
    declaration
    statement
expression-statement:
    expressionopt ;
selection-statement:
    if ( expression ) statement
    if ( expression ) statement else statement
    switch ( expression ) statement

iteration-statement:
    while ( expression ) statement
    do statement while ( expression ) ;
    for ( expressionopt ; expressionopt ; expressionopt ) statement
    for ( declaration expressionopt ; expressionopt ) statement
jump-statement:
    goto identifier ;
    continue ;
    break ;
    return expressionopt ;

```

A.2.4 External definitions

```

translation-unit:
    external-declaration
    translation-unit external-declaration
external-declaration:
    function-definition
    declaration
function-definition:
    declaration-specifiers declarator declaration-listopt compound-statement
declaration-list:

```

```
declaration
declaration-list declaration
```

A.3 Preprocessing directives

```
preprocessing-file:
    groupopt
group:
    group-part
    group group-part
group-part:
    if-section
    control-line
    text-line
    # non-directive
if-section:
    if-group elif-groupsopt else-groupopt endif-line
if-group:
    # if
    constant-expression new-line groupopt
    # ifdef identifier new-line groupopt
    # ifndef identifier new-line groupopt
elif-groups:
    elif-group
    elif-groups elif-group
elif-group:
    # elif constant-expression new-line groupopt

else-group:
    # else new-line groupopt

endif-line:
    # endif new-line

control-line:
    # include pp-tokens new-line
    # define identifier replacement-list new-line
    # define identifier lparen identifier-listopt )
    replacement-list new-line
    # define identifier lparen ... ) replacement-list new-line
    # define identifier lparen identifier-list , ... )
    replacement-list new-line
    # undef identifier new-line
    # line pp-tokens new-line
    # error pp-tokensopt new-line
    # pragma pp-tokensopt new-line
    # new-line
text-line:
    pp-tokensopt new-line
non-directive:
    pp-tokens new-line
lparen:
    a ( character not immediately preceded by white-space
replacement-list:
    pp-tokensopt

pp-tokens:
    preprocessing-token
    pp-tokens preprocessing-token
new-line:
    the new-line character
```