Processamento de Linguagens

Parte I - B

FLINT

Flex Inline Template

Bruno Cancelinha (A75428)

José Bastos (A74696)

Marcelo Miranda (A74817)

23 de Abril de 2017

Universidade do Minho

Conteúdo

1 Introdução **1**

1.1 Estrutura do relatório 1

2 Análise e especificação **2**

2.1 Análise do problema 2

2.2 Objetivos 2

3 Ferramenta FLINT **3**

3.1 Notação 4

3.1.1 Funções 4

3.1.2 Declaração de variáveis 4

3.1.3 Variáveis em linha 5

3.1.4 Mapeamentos 5

3.1.5 Expressões condicionais 5

3.1.6 Chamamento de funções 6

3.2 Utilização 7

3.3 Output 7

3.3.1 Mapeamentos 8

3.3.2 Expressões condicionais 8

3.3.3 Chamamento de funções 8

4 Implementação **9**

4.1 Criação de funções C 9

4.1.1 Estrutura de dados 9

4.1.2 Funções 10

4.2 Funções Auxiliares 10

4.3 Flex 11

5 Conclusão **12**

A Exemplos **13**

Resumo

FLINT - Uma ferramenta simples que visa a tornar mais fácil a criação de templates para strings em C.

Capítulo 1

Introdução

Vamos agora apresentar um trabalho desenvolvido no âmbito de enraizar a matéria lecionada na unidade curricular ***Processamento de Linguagens***, mais especificamente, o uso da ferramenta ***Flex***.

Foi nos dada a escolher uma de entre 6 opções, viemos a optar pelo enunciado 2.6 *Inline Templates em C* e dai surgiu a ferramenta que estamos agora a apresentar, o ***Flint*** (Flex Inline Template).

1.1 Estrutura do relatório

Neste relatório, começamos por analisar o problema em questão e delinear os objetivos. Seguidamente apresentamos a ferramenta ***Flint*** e todas as suas funcionalidades bem como a utilização e o *output* que produz. Continuamos a demonstrar como foi implementada a ferramenta começando por explicar as estruturas e funções mais básicas até à sua implementação concreta em ***Flex***. Terminamos com uma conclusão breve do trabalho. Em apêndice vêm algums exemplos das capacidades do FLINT.

Capítulo 2

Análise e especificação

2.1 Análise do problema

Em 1972 estava Dennis Ritchie a terminar o que se veio a tornar uma das mais importantes linguagens de programação, a linguagem C. Devido à sua idade e filosofia como linguagem imperativa de baixo nível, *strings* não são suportadas naturalmente ou, melhor dizendo, não são suficientemente flexíveis para tratar, como é muitas vezes o caso, linguagens naturais.

É muito comum querer ter uma *string* com conteúdo variável, um simples cumprimento como `"Olá NOME"` em que `NOME` representa uma outra *string* com o nome do recetor. Para este efeito é frequentemente usada a função `sprintf` da biblioteca `<string.h>` que permite escrever vários dados numa string com um formato específico. Esta solução funciona e de facto é eficaz, mas é tediosa quando se pretende fazer a um conjunto significativo de linhas. E, devido à natureza estática de *arrays* em C, não é fácil garantir que não se excede o limite máximo da *string*.

2.2 Objetivos

Seria ideal imaginarmos uma ferramenta que simplificasse todo este processo de modelação e que garantisse uma maneira eficaz de construir um numeroso conjunto de *strings* formatadas com dados variáveis. Essa ferramenta teria de ser capaz de transformar um modelo, em código C compilável e eficiente. Teria que usar ***strings dinâmicas***, mantendo sempre atenção para não permitir *leaks* de memória. Estamos portanto a falar de um ***processador de modelos***, como será chamado em Português, ou *template engine* na língua anglo-saxónica, em par com uma ***linguagem de modelação***, ou *template language*.

É com este objetivo que desenvolvemos uma *linguagem de modelação* e o seu correspondente *processador de modelos*, que recebe o modelo especificado nessa linguagem e o converte em código C válido.

Capítulo 3

Ferramenta FLINT

A ferramenta ***FLINT*** (**Fl**ex **In**line **T**emplate), vem como resposta ao problema de tratamento de *strings* em C, facilitando a criação e uso de modelos na linguagem. Traz consigo um processador de modelos, escrito usando ***Flex***, que converte a linguagem de modelação **FLINT** numa função em C que recebe os dados usados durante o modelo, e retorna a *string* já formatada. FLINT tem a capacidade de reconhecer, na maior parte dos casos, as variáveis usadas no modelo, essas variáveis aparecerão como argumentos da função gerada pela ordem que aparecerem no código, ignorando casos em que aparecem repetidas.

Um documento começa com a primeira linha `//FLINT` para não deixar dúvidas de que se trata de facto de um ficheiro FLINT. De seguida vem as funções FLINT que será explicada em mais detalhe da secção seguinte mas, basicamente, funções FLINT são um *template*, são a parte de texto que será convertida em código C. O FLINT é suficientemente inteligente para distinguir o código C dos modelos, portanto mantém intacto qualquer código que o ficheiro original contenha.

As *strings* em C, como já foi referido neste relatório, não estão pensadas para lidar com tamanhos variáveis, isto é, não são dinâmicas. Não passam de uma sequência fixa de bytes terminada com um byte a NULL, o chamado *NULL terminator*. Embora este comportamento seja compreensível e, muitas vezes adequado, para um processador de modelos é esperado que seja capaz de suportar qualquer que seja o tamanho dos dados que lhe desejamos passar, e não podemos prever qual o tamanho dos dados que serão utilizados em conjunto com o modelo. Por esta mesma razão, decidimos que seria necessário o uso das ***GStrings*** do ***GLib***. Assim, o código gerado pelo **FLINT** usará `GString`s. O FLINT certifica-se que o código de *output* tem o `#include <glib.h>`, colocando-o caso este não esteja presente no ficheiro de *input* e não fazendo nada se estiver.

É possível converter algo como:

```
1 Greeting={{Bem-vindo [% nome %].}}
```

No código C seguinte:

```
char* Greeting( char* nome) {
    GString *str = g_string_new(NULL);

    g_string_append_printf(str, "Bem-vindo %s.", nome);

    return g_string_free(str, FALSE);
}
```

3.1 Notação

Nesta secção vamos ver como é a notação da linguagem de modelação FLINT e a sua robustez. Para distinguir caracteres pertencentes ao modelo de variáveis e outras características da linguagem FLINT, estas palavras especiais são colocadas entre `[% %]`, sendo que qualquer caracter que não pertença a esta notação FLINT, incluindo \n e outros caracteres especiais, serão considerados na string a retornar.

3.1.1 Funções

Como já foi dito acima, FLINT baseia-se em funções que são nada mais que modelos a serem convertidos em C.

Uma função FLINT é definida entre chavetas duplas e é-lhe atribuida um nome com o simbolo de igual. Assim, `Func={{ MODELO }}` representa uma função de nome `Func`, neste caso `MODELO` é só um marcador de posição sem qualquer valor semântico. Devido ao facto de qualquer caracter dentro das duplas chavetas será considerado na ***string*** resultante, a posição das duplas chavetas }} é bastante relevante.

3.1.2 Declaração de variáveis

Apesar da capacidade do FLINT, de automaticamente detetar variáveis no modelo e as coloca como atributos da função gerada, é por vezes necessário explicitar essas variáveis. Assim, para declarar uma varíavel com o nome `idade` do tipo `int`:

```
[% VAR int idade %]
```

Se o nome da variável já estiver a ser utilizado antes da declaração, o FLINT já a terá registado, embora é provável que com um tipo diferente do pretendido e esta declaração é simplesmente ignorada. De qualquer maneira, esta funcionalidade é útil em `IF`s e em ***variáveis em linha***. como veremos à frente.

3.1.3 Variáveis em linha

O mais simples dos processadores de modelos tem que incluir, pelo menos, a funcionalidade de substituição simples de variáveis no meio do *template*. Assim, sendo este um característica tão simples, será simplesmente o nome da variável entre a notação do FLINT `[% %]`, como será possível confirmar no exemplo dado no início deste capítulo. Quando o FLINT encontra ***variáveis em linha***, ou *inline variables*, regista automaticamente o seu nome para ser colocada nos argumentos da função gerada, por omissão regista a variável com tipo `char*`, mas o FLINT suporta o caso da variável já ter sido declarada com o tipo `float` ou `int`. Assim é possível fazer algo como:

```
1 Greeting={{Olá [% nome %]!
2 [% VAR int idade %]
3 Tu tens [% idade %], certo? }}
```

Neste exemplo, FLINT regista a varíavel `char* nome` e `int idade`, por esta ordem.

3.1.4 Mapeamentos

Para além da trivial variável em linha, o FLINT suporta ***mapeamentos***, que são representados da seguinte maneira:

```
1 [% MAP Func n l %]
```

Mapeamentos permitem mapear a função `Func` a uma lista `l` com `n` elementos. Chamando a função `Func` para cada elemento de `l`, havendo a restrição de que a função apenas pode receber um argumento do tipo `char*`.

Quando o FLINT encontra um mapeamento, regista a variável `n` do tipo `int` e `l` do tipo `char**`.

3.1.5 Expressões condicionais

Nem sempre tudo o que está no modelo corresponde ao que desejamos apresentar. Podemos querer ter um modelo que varia consoante os dados. Trata-se aqui a questão de controlo de fluxo, permitir que certas partes do modelo sejam apresentadas ou não, consoante os dados perante a condição imposta.

O FLINT também permite ***expressões condicionais*** do tipo ***if***, ***else if*** e ***else***. Um bloco `IF` termina obrigatoriamente com `ENDIF` e pode, ou não, conter blocos `ELIF`, que representa o *else if*, e `ELSE`. Tanto o `IF` como o `ELIF` recebem uma condição como argumentos, qualquer variável encontrada nessa condição é registada como sendo do tipo `int`, a não ser que já tenha sido registada anteriormente.

```
[% IF sexo == FEMININO %]
Estás muito bonita hoje.
[% ELIF sexo == MASCULINO %]
Estás muito bonito hoje.
[% ELSE %]
Bom dia.
[% ENDIF %]
```

No exemplo acima, caso `sexo` seja `FEMININO`, é apresentada a *string* `"Estás muito bonita hoje"`, `"Estás muito bonito hoje"` caso `sexo` seja `MASCULINO` ou `"Bom dia."` em qualquer outro caso. O FLINT automaticamente deteta a palavra `sexo` como uma variável do tipo `int`, e ignora as palavras `FEMININO` e `MASCULINO` assumindo que se tratam de constantes pelo facto de estarem escritas todas em maiúsculas. O FLINT vai detetar qualquer palavra minúscula como uma variável a não ser que esteja protegida pelo carácter `:`, muito útil quando a condição chama funções em C.

```
Olá [% nome %]!
[% IF (:strcmp:(nome, "Maria") == 0) || (:strcmp:(nome, "João") == 0) %]
Tu outra vez!?
[% ENDIF %]
```

Aproveitamos o exemplo de cima para demonstrar, para além do uso do *escape character* `:`, o uso de uma disjunção, embora conjunções também sejam possíveis, aliás, qualquer condição que funcionaria em C, irá funcionar no FLINT. Também é possível ter ***if aninhados***.

3.1.6 Chamamento de funções

É possível chamar funções FLINT dentro dum modelo usando a expressão `CALL`, seguida do nome da função e os seus argumentos. Estes argumentos são variáveis que constarão nos argumentos da função gerada, se não tiverem já sido referidos antes da expressão `CALL`, o FLINT vai assumi-los como `char*`. Para além da função `CALL`, o FLINT permite `BREAK` que sai do modelo.

```
Nome={{
Nome: [% nome %]}}

Idade={{
[% IF i < 18 %]
[% BREAK %]
[% ENDIF %]
Idade: [% i %]}}

Info={{
[% CALL Nome nome %]
[% VAR int idade %]
[% CALL Idade idade %]
}}
```

No exemplo acima, a função `Idade` termina logo se o valor de `i` for inferior a 18. Podemos também observar que a função `Info` chama as duas funções anteriores, tendo o cuidado de declarar a variável `idade` como um inteiro.

3.2 Utilização

A ferramenta FLINT deve ser chamada pelo terminal da seguinte maneira:
`$ flint [file] [-o output]`
Caso nenhum ficheiro de output seja especificado, o FLINT imprime para o *standard output*.

3.3 Output

A maneira sistemática com que o FLINT converte os modelos em funções C, pode ser generalizada da seguinte maneira:

```
char* FUNCAO( ... ) {
        GString *str = g_string_new(NULL);

        (...)
        g_string_append_printf(str,  ... );
        (...)

        return g_string_free(str, FALSE);
}
```

Tal como já foi referido, usamos as ***GStrings*** do ***GLib*** pelas suas vantagens como *strings* dinâmicas. Assim temos a primeira linha da função dedicada a inicializar a *string* e a última a libertar toda a informação auxiliar, deixando apenas o `char*` correspondente aos dados que fazem a *string*. Como é possível notar, as *strings* estão alocadas em memória dinâmica, sendo

portanto aconselhado que sejam libertadas depois de ter sido chamada a função.

Vamos agora analisar uma generalização do código gerado algumas das funcionalidades vistas acima.

3.3.1 Mapeamentos

Um mapeamento do tipo `[% MAP Func n l %]` é desdobrado num `for` em C, tendo cuidado para libertar a memória alocada por cada chamamento da função `Func`.

```
for (int i = 0 ; i < n; i++){
        char* aux = Func(l[i]);
        g_string_append_printf(str, "%s", aux);
        free(aux);
}
```

3.3.2 Expressões condicionais

Os `IF`s como o do primeiro exemplo da secção anterior é transformado no seguinte código em C:

```
if ( sexo == FEMININO) {
        g_string_append_printf(str, "Estás muito bonita hoje.\n");
} else if ( sexo == MASCULINO) {
        g_string_append_printf(str, "Estás muito bonito hoje.\n");
} else {
        g_string_append_printf(str, "Bom dia.\n");
}
```

3.3.3 Chamamento de funções

Um chamamento como `[% CALL Func x y z %]` é convertido em C tendo, novamente, atenção para libertar a memória alocada pelo chamamento da função `Func`.

```
char* Func_aux = Func(x, y, z);
g_string_append_printf(str, "%s", Func_aux);
free(Func_aux);
```

Como podemos observar, é declarada a variável `Func_aux`, ou seja, o nome da função seguido de aux, para depois ser libertada.

Capítulo 4

Implementação

Vamos agora seguir o processo pelo qual as funcionalidades que vimos acima forma implementadas. Em primeiro lugar, decidimos usar as funcionalidades da biblioteca ***GLib*** que dispunha de uma coletânea de estruturas de dados que nos foram úteis, sendo a ***GQueue*** a mais proeminente usada.

Este trabalho impunha manipular strings em C, para facilitar esse processo, dispomos da biblioteca ***Simple Dynamic Strings*** (`https://github.com/antirez/sds`), que traz consigo as `sds`.

Vamos agora descrever como está implementada a ferramenta FLINT começando no código mais básico.

4.1 Criação de funções C

Para transformar o código FLINT em funções C, desenvolvemos uma estrutura de dados com funções adequadas a facilitar este mesmo trabalho. Assim rapidamente foi possível implementar todas as funcionalidades descritas no capítulo anterior.

4.1.1 Estrutura de dados

A estrutura de dados tem é definida da seguinte maneira:

```
1 struct buffer {
2     GQueue *var_name, *var_type;
3     GQueue *lines;
4     sds hdr, line, att;
5     int indent;
6 }
```

Tomando partido das *Queues* do ***GLib***, guardamos o nome de uma variável FLINT bem como o tipo a que ela corresponde, quer seja `char*`, `int` ou `float`, apesar de estar em duas estruturas separadas, o tipo que corresponde à variável guardada na posição `n` de `var_name` está guardada na posição `n` de `var_type`. Guardamos também as linhas prontas a serem escritas no ecrã.

Usamos três ***Simple Dynamic Strings***. `hdr` para escrever o *header* da função gerada. `line` que corresponde à linha formatada a ser impressa, isto é, pode conter `%s`, `%d` ou `%f`, entre caracteres comuns. `att` corresponde à linha de atributos que segue a *string* formatada no `g_string_append_printf`.

O *buffer* é dotado da capacidade de mudar a identação alterando aquela variável `int indent`.

4.1.2 Funções

A estrutura vem munida de várias funções, apresentamos aqui as mais interessantes.

reg_variable
Regista uma variável, ou seja, adiciona essa variável aos parametros a receber na função gerada.

build_strapp_line
Constrói uma linha do tipo "*string append*", isto é, linhas que seguem a estrutura `g_string_append_printf(str, line, att);`.

build_header
A partir das variáveis registadas em `var_name` e `var_type`, gera o cabeçalho da função.

begin_function
Prepara o *buffer* para começar a desenvolver a função. Escreve o cabeçalho da função na `GQueue *lines`.

push_line
Coloca uma dada linha na `GQUeue *lines`.

4.2 Funções Auxiliares

A estrutura demonstrada anteriormente na secção 4.1 é utilizada for funções simples, que podem ou não receber a linha do ficheiro de *input* e a processam. As funções responsáveis pelas funcionalidades demonstradas na secção 3.1 do capítulo ***Ferramenta FLINT***, são processadas por funções auxiliares com o prefixo `flint_`, assim temos, pela ordem da secção:

flint_var
Usado para declarações de variáveis `[% VAR <tipo> <nome> %]`.

flint_invar
Para processar variáveis em linha `[% [variavel] %]`.

flint_map
Mapeamentos `[% MAP <funcao> <tamano> <lista> %]`.

flint_if e **flint_elif**
Expressões condicionais `[% IF <condicao> %]` e `[% ELIF <condicao> %]` .

flint_call
Chamamento de funções `[% CALL <funcao> <argumentos...> %]`.

4.3 Flex

Flex é usado para ler o *input* e chamar as funções que acabamos de ver. Dado que o *input* é divido em dois contextos diferentes, o código C chamado `CODE` e o modelo, `TEMPLATE`. Qualquer caracter encontrado no código C é simplesmente enviado para o *output*, expetando se corresponder à expressão `.+=\{\{`, caso esse em que entra no contexto do modelo e volta para o código C ao encontrar }}.

Capítulo 5

Conclusão

Terminado o trabalho, fomos capazes de implementar uma ferramenta de processamento de modelos usando o ***Flex***. É capaz de suportar, para além das funções de variáveis em linha e de mapeamentos sugeridos pelo professor, expressões condicionais e chamamento de funções também muito úteis para modelação.

Embora o FLINT seja capaz, conforme previsto, de transformar um modelo em código C legível e compilável sem *leaks* de memória, há certas funcionalidades que queríamos ter visto implementadas embora não tivesse sido possível.

Em primeiro lugar, o FLINT só suporta uma única *flag*, `-o` para redefinir o output, embora interessante gostaríamos de ter implementado flags que alterariam diretamente o output como forma a dar mais customização ao código gerado. Podíamos imaginar uma *flag* que definia a posição da chaveta {, se diretamente à frente do cabeça ou na linha seguinte. Ou até escolher entre *tabs* ou espaços, podando mudar a quantidade de espaços para cada identação.

Outra funcionalidade que seria interessante e útil para a ferramenta seria controlo de erros, permitir que dado um ficheiro FLINT mal formatado, o programa fosse capaz de detetar a linha onde a infração foi registada e devolver uma mensagem de erro. Infelizmente da maneira como foi utilizado, o ***Flex*** não facilitava esta tarefa.

Por último lugar, o chamamento de funções assume que por omissão que os argumentos são do tipo `char*` se não tiverem sido explicitados anteriormente. Visto que a funcionalidade `CALL` foi pensada para chamar funções FLINT, ou seja, podia assumir *a priori* o tipo correto dos argumentos.

Apesar desta reflexão com um peso algo que negativo, ficamos bastante satisfeitos com o resultado do trabalho e com a facilidade que o ***Flex*** permitiu para a sua implementação. Resta-nos apresentar agora em anexo alguns exemplos de utilização que achamos interessantes.

Apêndice A

Exemplos

```
Comprimento={{ Olá, }}
```