Análise e Teste de Software
Framework de Teste Unitário para $C--$

Carlos Morais & José Pereira

20 de Janeiro de 2015

Contents

1 Introdução

Neste projeto pretende-se estender a framework $C--$ com a possibilidade de realizar testes unitários a la Java. Os testes unitários serão definidos com uma invocação a uma função $C--$ chamada assert.

Os argumentos de assert são: o primeiro argumento é o nome da função $C--$ a testar, os argumentos seguintes são valores com que essa funções será executada, e o último argumento será um valor (ou propriedade) que o resultado deverá produzir (obedecer). Este último argumento, poderá também ser um expressão que o resultado de a função a testar deverá verificar.

Dado um programa $C--$ com testes unitários, a framework de testes deverá permitir produzir dois programas executáveis. No primeiro caso deverá produzir um programa sem qualquer teste, que é a versão normal do programa.

No segundo caso, quando a framework é executada com a opção $-assert$, produz um relatório onde são apresentados mais informações dos resltados obtidos.

2 Assert

Na proposta do projeto são apresentados alguns exemplos de asserts para a $framework$. Seguindo esses exemplos tentamos idealizar diferentes tipos de asserts que a função deverá ter capacdade de testar.

Todos os asserts, deverão ter como argumento o nome da função a testar.

A função a testar, terá outros argumentos. Os primeiros serão os argumentos com que a função será testada, sendo que estes devem ser tantos e do mesmo tipo, que os argumentos declarados na função. Os tipos dos argumentos aceites são: inteiro(int), caracter(char) e boleano(boolean).

O último argumento é obrigatório e será o caso de teste do assert. Este poderá ser de um dos tipos já identificados, ou uma condição de comparação, em que o assert deverá comparar com a função a testar.

3 Exemplos de assert

Nesta secção serão apresentados três exemplos de prgramas $C--$ com asserts que no geral cobrem todos tipos de testes que deverão poder ser feitos com o assert.

3.1 Exemplo 1

```
char letraA(){
    return 'a';
}
```

```
assert( letraA('a') );
assert( letraA('u') );
```

3.2 Exemplo 2

```
boolean maior(int a, int b){
    if(a>b)
        return true;
    else
        return false;
}

assert( maior(3,4,false) );
assert( maior(5,4,true) );
```

3.3 Exemplo 3

```
int somaDez(int a){
    int res;
    res = a + 10;
    return res;
}

assert( somaDez(5,15) );
assert( somaDez(5,10) );
assert( somaDez(5,">=15") );
assert( somaDez(5,"<=15") );
assert( somaDez(5,"<20") );
assert( somaDez(5,"==22") );
assert( somaDez(5,">5") );
```

4 Gramática

A gramática utilizada foi alterada, para que a linguagem passasse a incluir os asssert. Por uma questão de legibilidade não está incluida a possibilidade de comentários, mas estes serão introduzidos com o ANTLR.

As alterações efetuadas foram as seguintes:

$$
\begin{aligned}
&\langle Programa\rangle \longrightarrow \langle Declaracao\rangle\ ';'\ \langle Programa\rangle \\
&\quad \mid \langle Funcao\rangle\ \langle Programa\rangle \\
&\quad \mid \langle Assert\rangle\ \langle Programa\rangle \\
&\quad \mid \varepsilon
\end{aligned}
$$

$$
\langle Assert\rangle \longrightarrow 'assert'\ '('\ ID\ '('\ ((\langle ArgsAssert\rangle\ ','\ \langle Expected\rangle) \mid \langle Expected\rangle\ )')\ ')'\ ';'
$$

$\langle ArgsAssert \rangle \longrightarrow INT$
. $| CHAR$
. $| bool\_assert$

$\langle Expected \rangle \longrightarrow INT$
. $| CHAR$
. $| bool\_assert$
. $| comp\_asser$

$\langle bool\_assert \rangle \longrightarrow 'true' | 'false'$

$\langle comp\_assert \rangle \longrightarrow '==' | '>=' | '<=' | '>' | '<'$

5 ANTLR

Depois de definida a gramática foi necessário proceder às alterações nas regras da gramática em ANTLR. Às regras foram também adicionados os construtores a serem aplicados pela estrutura do GOM.

As alterações implementadas são as seguintes. (Na regra *myassert* não apresenta a possibilidade de comentários para melhorar a sua legibilidade, contudo estes estão incluidos no código original.)

```
programa :
  declaracao ';' -> declaracao
  | funcao -> funcao
  | myassert -> myassert
  ;


myassert  :
  'assert' '(' ID '(' ((argsAssert ',' expected) | expected) ')' ')' ';'
       -> ^(Assert ID ^(ListaArgsAssert argsAssert?) ^(ExpAssert expected))
  ;

argsAssert :
  argumentoAssert ( ',' argumentoAssert )* -> argumentoAssert*
  ;

argumentoAssert :
  comentarios INT comentarios -> ^(ArgumentoAssertInt INT)
  | comentarios CHAR comentarios -> ^(ArgumentoAssertChar CHAR)
  | comentarios boolean_assert comentarios
      -> ^(ArgumentoAssertBool boolean_assert)
  ;
```

```
expected  :
  expectedVal
  | expectedBool
  | expectedComp
  ;

expectedVal  :
  comentarios INT comentarios -> ^(ExpectedArgInt INT)
  | comentarios CHAR comentarios -> ^(ExpectedArgChar CHAR)
  ;

expectedBool : comentarios boolean_assert comentarios
      -> ^(ExpectedArgBool boolean_assert)
  ;

expectedComp  :
  comentarios '"' comentarios compassert comentarios INT
  comentarios '"' comentarios
      -> ^(ExpectedArgComp compassert INT)
  ;


boolean_assert  : 'true' | 'false';

compassert  :( '==' | '>=' | '< | '>' | '<' )  ;
```

6 TOM

A estrutura do GOM sofreu também as alterações de acordo com os novos construtores que vão ser utilizados pelo ANTLR. As alterações implementadas foram as seguintes.

```
Instrucao =
  (...)
  | Assert(Nome:String,ArgumentosAssert:ArgumentosAssert,
      ExpectedAssert:ExpectedAssert)
  (...)

ArgumentosAssert = ListaArgsAssert(ArgumentosAssert*)
  | ArgumentoAssertInt(Int:int)
  | ArgumentoAssertChar(Char:String)
  | ArgumentoAssertBool(BoolAssert:String)

ExpectedAssert = ExpAssert(ExpectedAssert*)
  | ExpectedArgInt(Int:int)
```

```
  | ExpectedArgChar(Char:String)
  | ExpectedArgBool(BoolAssert:String)
  | ExpectedArgComp(CompAssert:String,Int:int)
```

7 Dificuldades Encontradas

No recorrer da segunda fase foram encontradas algumas dificuldades devido à simplicidade da máquina vrtual MSP. Algumas instruções aceites pela linguagem $C--$ não são correctamente executadas com a MSP.

7.1 Sequência de IFs

No primeiro casso não é possível declarar if's seguidos no main de um programa $C--$.

```
void main(){
    if(1==1)
        print('T');
    else
        print('F');

    if(2 > 3)
        print('T');
    else
        print('F');
}
```

Para contornar esta situação, basta que em vez de declarar estas instruções no main, estas sejam executadas por funções auxiliares.

```
void main(){
    f1();
    f2();
}

void f1(){
    if(1==1)
        print('T');
    else
        print('F');
}

void f2(){
    if(2 > 3)
        print('T');
    else
```

```
        print('F');
}
```

7.2 boolean e char

Neste caso funções em que o valor de retorno é um char ou um Boolean não funcionam corretamente na MSP. As funções cujo o valor de retorno é um Char não produzem qualquer resultado, quanto às funções cujo valor de retorno é um Boolean dão um erro ao executas a MSP. Este problema estende-se às instruções em que é feito a comparação destes tipos.
Foram efetuadas várias tentativas no sentido de contornar esta limitação. A melhor solução para que esta limitação fosse resolvida, seria corrigir a MSP, contudo não se encontra no âmbito deste projeto.
De modo a que este tipo de funções funcionasse na MSP, fizemos algumas alterações no front-end $C--$. Os valores boolean e char são convertidos para um inteiro. Um boolean com valor true, é convertido para um inteiro com o valor 1 sendo um false convertido para o valor 0. Quanto aos char, são convertidos para o inteiro que é o seu código ASCII correspondente.
Desta forma, é possivel escrever em código $C--$ funções com boolean e char, mas na verdade estes são tratados como valores inteiros na MSP.
As instruções obtidas do parser, passam pela seguinte função, que é responsável por converter os respetivos valores.

$$Instrucao\ recFix(Instrucao);$$

8 Compilação em modo Assert

Para que seja produzido um programa que teste os asserts declarados no programa $C--$, é executada um função que recebe como argumento as instruções lidas pelo parser. Esta é responsável por criar um main() para o programa ser executado no modo assert, sendo substituido um main() que tenha sido declarado no programa $C--$.

$$Instrucao\ createAssertMain(Instrucao);$$

Para cada instrução Assert é executada uma função auxiliar responsável por adicionar uma função ao programa $C--$, que será responsável pelo teste do assert. É adicionada também uma instrução à função main() para que a função seja executada na MSP.

$$Instrucao\ createFuncaoAssert(Assert);$$

É utilizada a API fornecida pela framework para a inclusão destas instruções no programa $C--$ a ser produzido.
Durante esta fase também é produzido um ficheiro com informações sobre os asserts produzidos, que será utilizado pela MSP para a criação de um relatório

com os resultados dos testes produzidos.

$$void\ prepareReport(Instrucao);$$

9 MSP em modo Assert

A máquina virtual MSP ao receber como argumento o programa compilado em modo assert terá como output os resultados de cada função criada para o teste de um assert. Uma vez que a framework $C--$ não permite a utilização de Strings, para cada teste apenas é produzido um 'T' caso tenha sucesso, ou um 'F' em caso de insucesso.
De forma obter um output mais agradável, a MSP deverá ser executada com a opção $-assert$, para que seja lido o ficheiro com as informações sobre os asserts produzido pelo compilador $C--$.
Desta forma é feito um cruzamento do output da MSP com os asserts que estão a ser testados de forma a produzir um output que indique para cada função gerada para um assert, se este passou ou falhou o teste.
É também produzido um relatório em HTML, onde para cada função testada em modo assert são indicados em quantos testes a respetiva função passou e o número total de testes efetuados. São também apresentados os testes em que a função não falhou o teste.

10 Utilização da framework em modo Assert

A Framework foi estendida para que quando esta fosse executada com a opção $-assert$, em vez de produzir o programa normal, produza um programa executável pela máquina virtual MSP, em que o output deste será o resultado dos testes dos asserts declarados.

java gram/Main -assert ⟨ teste.i ⟩ teste.msp

É também criado um ficheiro "reportAssert.txt" que deverá acompanhar o ficheiro".msp" para ser executado na MSP. Este ficheiro contém algumas informações que são utilizadas para a produção do relatório de testes pela MSP. Para que seja produzido o relatório pela MSP, esta também deverá ser invocada com a opção $-assert$, produzindo assim um ficheiro"report.html", para além do output do respetivo programa.

java maqv/Main teste.msp -assert

11 Exemplo de teste

Criado um programa $C--$ com os exemplos apresentados na secção 3, são apresentados os outputs gerados pela MSP.

11.1 output terminal

```
[Carloss-MBP-2:genMaqV carlosmorais$ java maqv/Main report.msp -assert
 somaDez passa
 somaDez FALHA
 somaDez passa
 somaDez passa
 somaDez passa
 somaDez FALHA
 somaDez passa
 letraA passa
 letraA FALHA
 maior passa
 maior passa
```

11.2 Relatório html

Relatorio de testes unitarios

- maior
  passou em 2 de 2 testes

- somaDez
  passou em 5 de 7 testes
  testes em que falhou:
  - somaDez(5)==10
  - somaDez(5)==22

- letraA
  passou em 1 de 2 testes
  testes em que falhou:
  - letraA()==u

12 Conclusão e trabalho futuro

Apesar de algumas dificuldades encontradas ao logo do desenvolvimento, infligidas pela simplicidade da MSP, conseguimos cumprir com o trabalho inicialmente proposto, acrescentando ainda a produção de um relatório em html com os detalhes dos testes efetuados.

De notar que embora a MSP não consiga executar funções que manipulam valores booleanos ou caracretes, é possível obter o resultado certo de um assert para estas funções, uma vez que os valores são previamente convertidos para inteiros.
Embora todos os objetivos tenham sido realizados, existe ainda um conjunto de tarefas que podem ser feitas em contunuidade deste projeto.
Em primeiro lugar, após uma eventual correção da MSP, a framework deverá ser atualizada, para que passe a trabalhar realmente com valores booleanos e caracteres e não com inteiros.
O relatório html produzido é bastante simples, pelo que podem ser adicionadas novas informações relevantes sobre os testes efetuados.
Alguns dos trabalhos realizados nesta disciplina para a framework poderão também ser integrados com este, como a localização de falhas e o teste de cobertura.