

Teste de Cobertura para C - -

Grupo 3

A67673 - André Geraldes

A61071 - Pedro Duarte

A67709 - Sandra Ferreira

21 de Janeiro de 2016

Resumo

Hoje em dia é crucial e totalmente indispensável fazer testes ao software produzido, pois, errar é Humano, como diz certo ditado. Como tal, realizou-se vários casos de testes sobre alguns programas para ser possível retirar conclusões sobre a sua qualidade. Mais uma vez, pondo a condição humana em causa, idealizou-se os casos de teste com inputs que devolvam os outputs esperados o que nada prova sobre a correcção do software. É, portanto, necessário analisar a cobertura dos testes que fazemos correr sobre os programas.

1 Introdução

No âmbito da unidade curricular de Análise e Teste de Software, foi dada uma escolha de entre vários projetos apresentados pelos docentes. O projeto escolhido a desenvolver recaiu no tema: Testes de Cobertura para a linguagem C-. Este projeto tem como objetivo desenvolver uma análise de cobertura de casos de teste para uma linguagem C-. A ideia consiste em estender o processador de C- de modo a que sempre que um conjunto de testes for executado seja possível analisar a sua cobertura, ou seja, num tom mais brejeiro se foi executada determinada instrução ou não. O resultado desta análise será a geração de um relatório que indica a cobertura desses casos de teste: por exemplo, indica se todas (ou que percentagem) as funções C- foram executadas/testadas, se todos os ramos das condições *if-then-else* foram testados, se todas as componentes de uma expressão lógica da condição de paragem de um ciclo foram testada, ou mesmo, se todos os blocos de código foram testados. O analisador de cobertura deve transformar/instrumentar o código C-, do programa a ser testado, com os casos de teste, de modo a este produzir informação sobre que funções/blocos de código/expressões lógicas foram usadas na execução do programa. Posteriormente, essa informação é usada para identificar as partes do código fonte que foram ou não testadas.

1.1 Estrutura do Relatório

O desenvolvimento deste projecto dividiu-se em três fases de entrega, pelo que neste relatório será explicado com detalhe cada etapa depois de uma breve referência ao modelo relacional usado mediante as ferramentas que se disponibilizaram, seguidamente passaremos para a demonstração de um caso de estudo feito. Por fim serão apresentadas umas notas finais sobre o projecto final apresentado e umas considerações sobre um possível trabalho futuro.

1.2 Modelo relacional

Para este trabalho foi fornecido ao grupo protótipos (desenvolvidos em Java e Haskell) com a implementação de processador de C- e de uma máquina virtual MSP. O grupo decidiu trabalhar sobre a versão desenvolvida em Java.

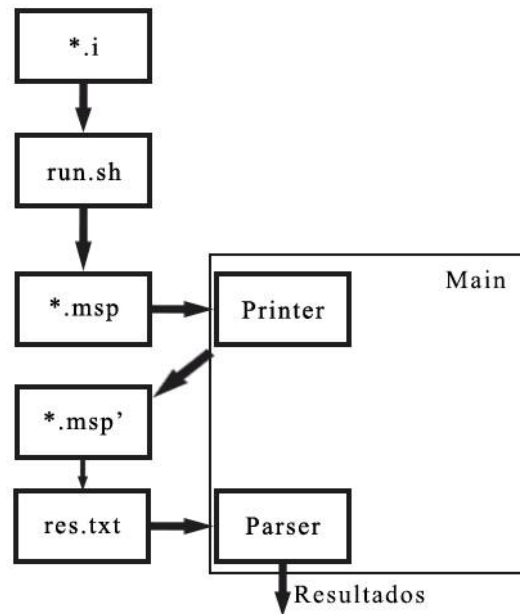


Figura 1: Modelo relacional

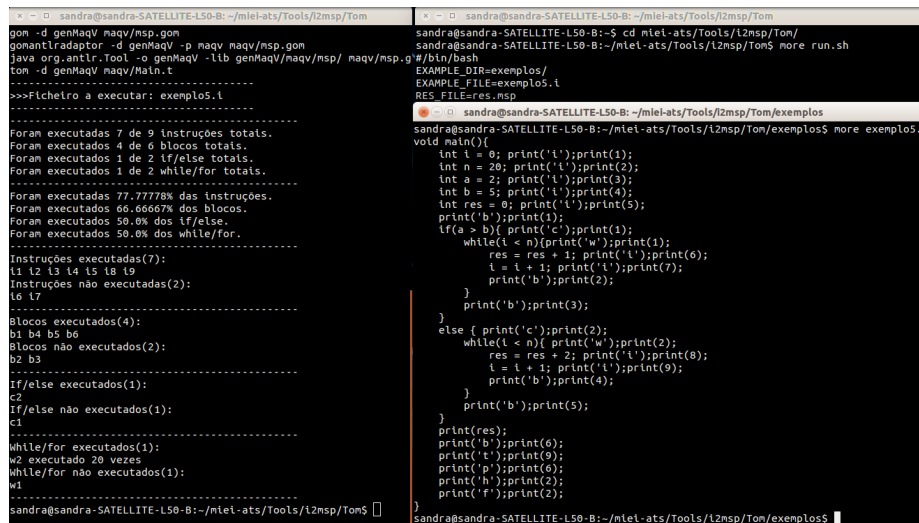
O percurso de toda a aplicação é feita seguindo estes passos:

- Criação de exemplos de programas em C-. Tais programas contém atribuições, condições, ciclos, invocação de outras funções e inserção de valores inputs do teclado na altura da compilação.
- A execução do nosso programa é controlada pelo script feito em bash **run.sh** que controla todo o processo de funcionamento do nosso programa. Este script será explicado ao longo do relatório e estará contido em anexo no final do mesmo.
- Será gerado código MSP do programa escolhido para teste;
- A Classe *Printer* foi desenvolvida para automaticamente fazer a cobertura do código obtido através de uma simples adição de prints.
- Será obtido um novo código MSP com a cobertura realizada;
- O código MSP será compilado através da máquina virtual gerando um resultado que é passado para um ficheiro `res.txt`;
- O ficheiro `res.txt` será tratado pela Classe *Parser* que o interpreta formando resultados finais sobre o teste de cobertura.

2 Primeira Fase de Desenvolvimento do Projecto

Para a primeira etapa de desenvolvimento do projecto delineou-se fazer vários programas em C-. Cada um destes programas seria anotado, com instruções de *print*, para ser possível observar os caminhos que o programa toma consoante os vários *inputs* que lhe são atribuídos. Para ser possível analisar o *tracing*, distinguimos quatro partes distintas num programa: as instruções, as condições, os ciclos e os blocos básicos (sequência consecutiva de instruções com uma única entrada e uma única saída). Assim sendo, para cada programa C-, são apresentadas as linhas de código seguidas de um "print('p');print(num);" onde "p" pode corresponder aos caracteres: i-instrução; c- condição (if/else); b- bloco ou w-while/for. O num, por sua vez, representa o número sequencial de cada uma destas partes do programa. No final dos ficheiros C- - encontram-se quatro *prints* que indicam o número total de instruções ('t'), blocos básicos ('p'), if/else ('h') e while/for ('f').

O *output* gerado por cada programa criado é direccionado para o ficheiro res.txt. Foi criado um *parser* com a finalidade de analisar este ficheiro e posteriormente apresentar as respectivas estatísticas. Estas estatísticas indicam quais foram as partes do código por onde o programa passou e a percentagem de instruções, condições blocos e ciclos que foram executados e ainda, no caso dos ciclos, indica ainda o número de vezes que este é executado.



The image shows two terminal windows. The left window displays the execution of a program named 'exemplo5.i' using the 'genMaqV' tool. The output shows statistics for instructions, blocks, if/else statements, and while/for loops. The right window shows the source code of 'exemplo5.i' in C- syntax, which includes various print statements for tracing.

```
gcm -d genMaqV maqv/msp.gcm
gonantlrAdaptor -d genMaqV -p maqv maqv/msp.gcm
java org.antlr.Tool -o genMaqV -lib genMaqV/maqv/msp/ maqv/msp.g #/bin/bash
ton -d genMaqV maqv/Main.t
-----
>>>Ficheiro a executar: exemplo5.i
-----
Foram executadas 7 de 9 instruções totais.
Foram executados 4 de 6 blocos totais.
Foram executados 1 de 2 if/else totais.
Foram executados 1 de 2 while/for totais.
-----
Foram executadas 77.77778% das instruções.
Foram executados 66.66667% dos blocos.
Foram executados 50.0% dos if/else.
Foram executados 50.0% dos while/for.
-----
Instruções executadas(7):
t1 t2 t3 t4 t5 t8 t9
Instruções não executadas(2):
t6 t7
-----
Blocos executados(4):
b1 b4 b5 b6
Blocos não executados(2):
b2 b3
-----
If/else executados(1):
c2
If/else não executados(1):
c1
-----
While/for executados(1):
w2 executado 20 vezes
While/for não executados(1):
w1
-----
sandra@sandra-SATELLITE-L50-B:~/miel-ats/Tools/l2msp/Tom$
```

```
sandra@sandra-SATELLITE-L50-B:~/miel-ats/Tools/l2msp/Tom/
sandra@sandra-SATELLITE-L50-B:~/miel-ats/Tools/l2msp/Tom/
sandra@sandra-SATELLITE-L50-B:~/miel-ats/Tools/l2msp/Tom$ more run.sh
EXAMPLE_DIR=exemplos/
EXAMPLE_FILE=exemplo5.i
RES_FILE=res.msp
sandra@sandra-SATELLITE-L50-B:~/miel-ats/Tools/l2msp/Tom/exemplos$ more exemplo5.
void main(){
    int i = 0; print('t');print(1);
    int n = 20; print('l');print(2);
    int a = 2; print('t');print(3);
    int b = 5; print('t');print(4);
    int res = 0; print('l');print(5);
    print('b');print(1);
    if(a > b){ print('c');print(1);
        while(i < n){print('w');print(1);
            res = res + i; print('t');print(6);
            i = i + 1; print('t');print(7);
            print('b');print(2);
        }
        print('b');print(3);
    }
    else { print('c');print(2);
        while(i < n){ print('w');print(2);
            res = res + i; print('t');print(8);
            i = i + 1; print('t');print(9);
            print('b');print(4);
        }
        print('b');print(5);
    }
    print(res);
    print('b');print(6);
    print('t');print(9);
    print('p');print(6);
    print('h');print(2);
    print('f');print(2);
}
```

Figura 2: Estatísticas geradas pelo *parser* na análise do exemplo5.i em anexo

2.1 Considerações da Fase de Desenvolvimento

Para a primeira etapa houve um comprometimento em fazer o tracing dos programas de modo a ser possível averiguar os caminhos que os programas tomam consoante os diversos *inputs* que lhes são atribuídos. Adicionalmente, nesta fase

teríamos também de gerar estatísticas para saber que percentagem de código é executada.

Os prints que permitem verificar o *tracing* que é feito por um programa não são gerados automaticamente, isto é, para esta fase os prints foram colocados "à mão".

3 Segunda Fase de Desenvolvimento do Projecto

Nesta fase, foram trabalhados os objectivos de automatizar o processo de marcação de instruções em programas solicitados, não sendo, como na fase anterior de forma manual.

O seu desenvolvimento esteve dividido em cinco partes:

- Geração de código MSP: É compilado um exemplo de programa em C- (de realçar que tal programa encontrava-se sem os prints inseridos manualmente na primeira fase) de modo a gerar código MSP;
- Rastreio de instruções obtidas no código msp (atribuições, condições, ciclos e blocos) com adição automática de prints seguindo a nomenclatura atribuída na primeira fase.
- Tratamento do rastreio obtido para gerar uma cobertura do código pretendido, para tal apenas se aplicou o que já se tinha desenvolvido na primeira fase ao ficheiro gerado.
- Alterações no ficheiro run.sh para adicionar os passos da segunda fase.
- Alterações na main, para conforme o argumento recebido fazer as devidas instruções.

A base de desenvolvimento desta fase está no rastreamento das instruções contidas no código msp gerado. De realçar que o ficheiro que continha todo o código MSP foi dividido por parâmetros separados por vírgulas, cada um a corresponder a uma posição de um array criado "*params*" para guardar todas as instruções. Ao percorrer o array seriam adicionados as instruções de prints conforme as instruções presentes.

A função criada após a compreensão das instruções em MSP para realizar os prints conforme feitos na primeira fase insere duas instruções novas que realizam os prints necessários, "*Pushc umchar,IOut,Pushi inteiro,IOut*", por exemplo "*Pushc 'i',IOut,Pushi 1,IOut*" que imprimirá o identificador da instrução 1.

```
public void newPrint(char a, int i, int index){
    this.params.add(index+1, "Pushc '" + a + "',IOut");
    this.params.add(index+2, "Pushi "+ i +",IOut");
}
```

Figura 3: Função que insere um novo print na posição index.

- **Atribuições:** Para as este tipo de instruções foi procurado a string "Store", pois esta está associada a guardar valores de atribuições, onde após a devida correspondência se invoca a função anterior com o char 'i' e this.ni.

```
public void addPrints(){
    boolean main = true;
    int tam = this.params.size();
    int i;

    //Instruções que sao atribuiçoes
    for(i = 0; i < tam; i++){
        String h = this.params.get(i);
        if(h.contains("Store")){
            this.ni++;
            this.newPrint('i', this.ni, i);
            i += 2;
            tam += 2;
        }
    }
}
```

Figura 4: Adição dos prints de instruções.

- **Condições (If-else) e ciclos ("while" e "for").** Este tipo mais complexo de instruções/conjunto de instruções foi rastreado através da string "Jump"seguida de outra string que permitiria distinguir quais os ciclos e quais as condições.

```
//If/else/while/for
for(i = 0; i < tam; i++){
    String h = this.params.get(i);
    if(h.contains("Jumpf \\"senao\\")){
        this.nif++;
        this.newPrint('c', this.nif, i);
        i += 2;
        tam += 2;
    }
    else if (h.contains("Jump \\"fse\\")){
        if(!this.params.get(i+2).contains("ALabel \\"fse\\")){
            this.nif++;
            this.newPrint('c', this.nif, i+1);
            i += 3;
            tam += 2;
        }
    }
}
```

Figura 5: Exemplo do que foi feito para condições e ciclos.

- **Blocos** Para um tipo mais geral de rastrear como os blocos de código foram identificados, também através da string "Jump"seguida de outra string identificadora, parâmetros de blocos anteriores e interiores a ciclos e if-else. Outros blocos de código também foram identificados pela string "Halt"ou "Ret"que correspondem ao fim de uma função.

```

//Blocos
for(i = 0; i < tam; i++){
    String h = this.params.get(i);
    if(h.equals("Halt") || h.equals("Ret")){
        this.nb++;
        this.newPrint('b', this.nb, i-1);
        i += 2;
        tam += 2;
    }
    //Bloco anterior a um ciclo for
    if(h.contains("Jumpf \ "ffor")){
        this.nb++;
        this.newPrint('b', this.nb, i-1);
        i += 2;
        tam += 2;
    }
    //Bloco interior de um for
    if(h.contains("Jump \ "for")){
        this.nb++;
        this.newPrint('b', this.nb, i-1);
        i += 2;
        tam += 2;
    }
    //Bloco anterior a um if
    if(h.contains("Jumpf \ "senao")){
        this.nb++;
        this.newPrint('b', this.nb, i-1);
        i += 2;
        tam += 2;
    }
}

```

Figura 6: Exemplo do que foi feito para os blocos de código.

- **Totais** Após a colocação dos prints de cada instrução seria necessário colocar no fim da main os valores totais de cada um. A instrução "Halt" indica o fim da main, então os prints são colocados antes desta.

```

//Apos colocar todos os prints, inserir os valores totais no fim da main
for(i = 0; i < tam && main; i++){
    String h = this.params.get(i);
    if(h.equals("Halt") && main){
        main = false;
        this.params.add(i, "Pushi " + this.ni + ",I0ut");
        this.params.add(i, "Pushc 't',I0ut");

        this.params.add(i, "Pushi " + this.nb + ",I0ut");
        this.params.add(i, "Pushc 'p',I0ut");

        this.params.add(i, "Pushi " + this.nif + ",I0ut");
        this.params.add(i, "Pushc 'h',I0ut");

        this.params.add(i, "Pushi " + this.nc + ",I0ut");
        this.params.add(i, "Pushc 'f',I0ut");
    }
}

```

Figura 7: Adição dos prints com os valores totais.

3.1 Considerações da segunda fase de desenvolvimento

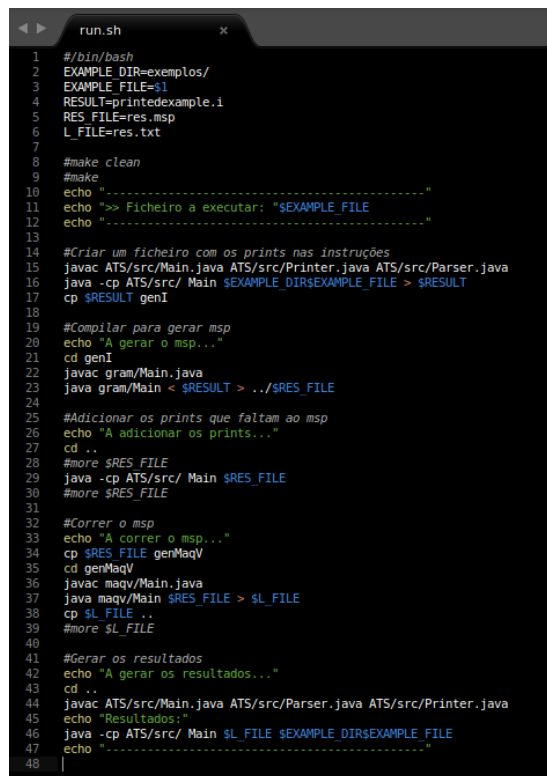
O objectivo de automatização do processo de cobertura num programa em C- foi alcançado, com a análise do código MSP gerado. Feita a sua leitura e reescrita foi possível criar um novo programa em C- para ser usado na nomenclatura implementada na primeira fase, de forma automática a fazer a cobertura de código em C-.

4 Última fase de desenvolvimento

Para alcançar o objectivo final ficava por desenvolver a possibilidade de colocar graficamente na consola, indicações da cobertura do código, mais especificamente as partes não executadas. Para isso adicionamos à classe `Parser.java` uma função `insertNotExec` com a capacidade de indicar quais as linhas que não correram. O seu modo de funcionamento começa por fazer uma leitura do ficheiro C- e colocar cada linha numa posição de um *arraylist*. Utiliza um *arraylist* que contém tudo que não foi executado e faz o trace identificando essas linhas. No final imprime o código C- já com as identificações das linhas com cores (a vermelho o que não foi executado; a verde o que foi executado).

4.1 Alterações no script run.sh

Fizemos algumas alterações no ficheiro script usado durante as fases anteriores



```
run.sh
1 #/bin/bash
2 EXAMPLE_DIR=exemplos/
3 EXAMPLE_FILE=s1
4 RESULT=printedexample.i
5 RES_FILE=res.msp
6 L_FILE=res.txt
7
8 #make clean
9 #make
10 echo "-----"
11 echo ">> Ficheiro a executar: '$EXAMPLE_FILE'"
12 echo "-----"
13
14 #Criar um ficheiro com os prints nas instruções
15 javac ATS/src/Main.java ATS/src/Printer.java ATS/src/Parser.java
16 java -cp ATS/src/ Main $EXAMPLE_DIR$EXAMPLE_FILE > $RESULT
17 cp $RESULT genI
18
19 #Compilar para gerar msp
20 echo "A gerar o msp..."
21 cd genI
22 javac gram/Main.java
23 java gram/Main < $RESULT > ../$RES_FILE
24
25 #Adicionar os prints que faltam ao msp
26 echo "A adicionar os prints..."
27 cd ..
28 #more $RES_FILE
29 java -cp ATS/src/ Main $RES_FILE
30 #more $RES_FILE
31
32 #Correr o msp
33 echo "A correr o msp..."
34 cp $RES_FILE genMaqV
35 cd genMaqV
36 javac maqv/Main.java
37 java maqv/Main $RES_FILE > $L_FILE
38 cp $L_FILE ..
39 #more $L_FILE
40
41 #Gerar os resultados
42 echo "A gerar os resultados..."
43 cd ..
44 javac ATS/src/Main.java ATS/src/Parser.java ATS/src/Printer.java
45 echo "Resultados:"
46 java -cp ATS/src/ Main $L_FILE $EXAMPLE_DIR$EXAMPLE_FILE
47 echo "-----"
48
```

Figura 8: Script run.sh da última fase de desenvolvimento

4.2 Resultado Final

```
▶ ./run.sh exemplo5.i
-----
>> Ficheiro a executar: exemplo5.i
-----
A gerar o msp...
A adicionar os prints...
A correr o msp...
A gerar os resultados...
Resultados:
-----
Foram executadas 7 de 9 instruções totais.
Foram executados 4 de 6 blocos totais.
Foram executados 1 de 2 if/else totais.
Foram executados 1 de 2 while/for totais.
-----
Foram executadas 77.77778% das instruções.
Foram executados 66.66667% dos blocos.
Foram executados 50.0% dos if/else.
Foram executados 50.0% dos while/for.
-----
Instruções executadas(7):
i1 i2 i3 i4 i5 i8 i9
Instruções não executadas(2):
i6 i7
-----
Blocos executados(4):
b1 b4 b5 b6
Blocos não executados(2):
b2 b3
-----
If/else executados(1):
c2
If/else não executados(1):
c1
-----
```

Figura 9: Resultado Final

```

-----
While/for executados(1):
w2 executado 20 vezes
While/for não executados(1):
w1
-----

void main(){
    int i = 0;
    int n = 20;
    int a = 2;
    int b = 5;
    int res = 0;

>    if(a > b){
>        while(i < n){
>            res = res + 1;
>            i = i + 1;
>        }
    }
    else {
        while(i < n){
            res = res + 2;
            i = i + 1;
        }
    }
    print(res);
}
0 código não executado está marcado com '>'
-----

```

Figura 10: Resultado Final

4.3 Conclusões Finais e trabalho futuro

Chegando à fase final de apresentação do projecto e juntando todas as considerações descritas em cada parte consideramos que o objectivo do trabalho foi alcançado. Mediante a criação de um programa em C— é possível obter estatísticas que abordam a cobertura do mesmo, estando presentes as percentagens das instruções executadas como também blocos, atribuições, condições e ciclos.

Também é possível visualizar quais foram as instruções não executadas como pretendido. Como possível trabalho futuro consideramos que caso se desenvolva mais este projecto, ficam a faltar a possibilidade de inserção de um conjunto de inputs para formar um teste mais completo ao código. Como nota final achamos que foi desafiante trabalhar com uma linguagem criada por um grupo de alunos deste curso.

5 Anexos

Exemplo5b.i - Exemplo de Programa com prints da primeira fase

```
void main(){
    int i = 0; print('i');print(1);
    int n = 20; print('i');print(2);
    int a = 2; print('i');print(3);
    int b = 5; print('i');print(4);
    int res = 0; print('i');print(5);
    print('b');print(1);
    if(a > b){ print('c');print(1);
        while(i < n){print('w');print(1);
            res = res + 1; print('i');print(6);
            i = i + 1; print('i');print(7);
            print('b');print(2);
        }
        print('b');print(3);
    }
    else { print('c');print(2);
        while(i < n){ print('w');print(2);
            res = res + 2; print('i');print(8);
            i = i + 1; print('i');print(9);
            print('b');print(4);
        }
        print('b');print(5);
    }
    print(res);
    print('b');print(6);
    print('t');print(9);
    print('p');print(6);
    print('h');print(2);
    print('f');print(2);
}
```

Exemplo5.i

```
void main(){
    int i = 0;
    int n = 20;
    int a = 2;
    int b = 5;
    int res = 0;

    if(a > b){
        while(i < n){
            res = res + 1;
            i = i + 1;
        }
    }
    else {
```

```

        while(i < n){
            res = res + 2;
            i = i + 1;
        }
    }
    print(res);
}

```

```

#!/bin/bash
EXAMPLE_DIR=exemplos/
EXAMPLE_FILE=exemplo6.i
RES_FILE=res.msp
L_FILE=res.txt

#make clean
#make
echo "-----"
echo ">> Ficheiro a executar: \"$EXAMPLE_FILE\""
echo "-----"
cp $EXAMPLE_DIR$EXAMPLE_FILE genI
cd genI
javac gram/Main.java
java gram/Main < $EXAMPLE_FILE > ../$RES_FILE

cd ..
#more $RES_FILE
javac ATS/src/Main.java ATS/src/Printer.java ATS/src/Parser.java
java -cp ATS/src/ Main $RES_FILE
#more $RES_FILE

cp $RES_FILE genMaqV
cd genMaqV
javac maqv/Main.java
java maqv/Main $RES_FILE > $L_FILE
cp $L_FILE ..
#more $L_FILE

cd ..
javac ATS/src/Main.java ATS/src/Parser.java ATS/src/Printer.java
java -cp ATS/src/ Main $L_FILE

```

Figura 11: Script run.sh da segunda fase de desenvolvimento