



**Ciências
ULisboa**

Relatório do Projeto Aguda Compiler

Milestone 3

Ricardo Miguel Neto Sobral 56332

How to update your tests

Para este projeto foi feito Fork do repositório git aguda-testing. Nesse fork estão presentes dois branches, o “main” e o “patch-limpo”. O branch “main” contem o código do compilador (escrito em python) enquanto o branch “patch-limpo”, contem apenas os ficheiros presentes no repositório aguda-testing (do professor). Este último branch serve apenas para atualizar os ficheiros de testes e para fazer os pedidos de merge request.

Para atualizar os testes, temos de garantir que estamos no branch “path-limpo”, e podemos fazer isso com o seguinte comando:

```
git checkout patch-limpo
```

Posto isto, basta executar o comando *git pull* para atualizar os ficheiros de testes.

How to build your compiler

Para esta fase do projeto foi usado Python e um ambiente virtual (venv) para isolamento.

Para além disso foi utilizado Ply (versão 3.11), para tratar de ficheiros essenciais como o lexer e o parser.

É importante indicar que o ficheiro main encontra-se em *src/aguda-cabal/aguda/main.py*.

Posto isto, é necessário correr dois comandos para correr o container Docker:

```
docker-compose build
```

e de seguida:

```
docker-compose run aguda_app bash
```

E agora está tudo pronto para correr o compilador!

How to run a particular test

Para correr um único ficheiro de teste (.agu) precisamos correr o seguinte comando no terminal:

```
python3 src/aguda-cabal/aguda/main.py <path-to-test>.agu
```

Onde path-to-test, é o caminho relativo para o teste específico, vejamos o seguinte exemplo:

```

let newMatrix : String[] = new String[15];
(venv) ricardo@MacBook-Pro-de-Ricardo aguda-testing-m4-local % python3 src/aguda-cabal/aguda/main.py test/valid/tcomp000_power-iterative/power-iterative.agu
Validation successful!

let power (base, exponent) : (Int, Int) -> Int =
    let result : Int = 1
    while (exponent > 0) do
        set result = (result * base)
        set exponent = (exponent - 1);
    result

let main (_) : (Unit) -> Unit =
    print(power(2, 6))

Código LLVM gerado em: test/valid/tcomp000_power-iterative/power-iterative.ll

```

How to run the whole test suite (valid and invalid programs)

Para esta fase do projeto os ficheiros python importantes para correr são abordados na próxima secção, denominada “How to interpret the testing output (how many tests passed, which failed)”, no entanto

Para correr vários testes temos várias opções:

1 run_all_valid_tests.py

Para correr todos os testes validos, isto é, todos os testes dentro da pasta *test/valid*. Para correr este ficheiro usamos o seguinte comando:

python3 run_all_valid_tests.py

```

(venv) ven@ricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 run_all_valid_tests.py

```

2 run_invalid_tests.py

Para correr todos os testes invalidos, isto é, todos os testes dentro das pastas *test/invalid-syntax* e *test/inavlid-semantic*. Para correr este ficheiro usamos o seguinte comando:

python3 run_invalid_tests.py

```
(venv) venvricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 run_invalid_tests.py
```

3 run_invalid_syntax_tests.py

Este ficheiro, corre todos os ficheiros (.agu) dentro da pasta test/invalid-syntax que devia conter todos os testes inválidos, que apresentassem apenas erros léxicos e de sintaxe.

Para correr este ficheiro use este comando:

```
python3 run_invalid_syntax_tests.py
```

```
Lexical Error: Lexical Error at line 3, column 11: illegal character.
(venv) venvricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 run_invalid_syntax_tests.py
```

4 run_invalid_semantic_tests.py

Este ficheiro, corre todos os ficheiros (.agu) dentro da pasta test/invalid-syntax que devia conter todos os testes inválidos, que apresentassem apenas erros léxicos e de sintaxe.

Para correr este ficheiro use este comando:

```
python3 run_invalid_semantic_tests.py
```

```
venvricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 run_invalid_semantic_tests.py
```

5 run_test_of_student.py

Para um melhor entendimento dos resultados, e para auxiliar a correção de erros no compilador, também foi criado um ficheiro Python, que corre os testes válidos de um aluno, passando o número de aluno. É de notar que também funciona para os testes do professor, que começam com tcomp000.

Para correr este ficheiro use este comando:

```
Python3 run_test_of_student.py <numero_de_aluno>
```

```
(venv) venvricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 run_test_of_student.py 56332
== Running test/valid/56332_factorial.in aguuda/factorial.in aguuda.agu ==
```

How to interpret the testing output (how many tests passed, which failed)

Para interpretar os resultados dos testes é simples. Para os casos em que os testes não passam na avaliação do compilador, é lançado um erro, indicado se o erro é um erro léxico

sintático ou semântico, como podemos ver nos casos abaixo:

```
=== Running test/invalid-syntax/64854_non_letter_id/non_letter_id.agu ===  
Syntax Error: Syntax error at line 4, column 5 near '1'  
=== Running test/invalid-syntax/tcomp000_wrong_comment/wrong_comment.agu ===  
Lexical Error: Lexical error at line 3, column 1: illegal character '#'  
=== Running test/invalid-semantic/tcomp000-branches-diff-types/branches-diff-types.agu ===  
Semantic Error: Error: (2,3) Branches must have same type, got Unit and Int  
(if true then unit else 5)
```

Para além disso, o compilador também indica a linha e a coluna onde o erro ocorre.



Nos casos dos erros semânticos, o compilador também imprime a expressão onde ocorre o erro e em alguns casos a linha em que ocorre o erro.

Para os ficheiros .agu que passam na avaliação do compilador, é impresso no terminal o AST do respetivo programa, graças ao ficheiro auxiliar pretty_printer.py e também é feita a geração de código LLVM, mas essa questão é aprofundada mais à frente com outro ficheiro de correr os ficheiros .ll.

```
(venv) ricardo@MacBook-Pro-de-Ricardo aguda-testing-m4-local % python3 src/aguda-caba  
l/aguda/main.py test/valid/tcomp000_power-iterative/power-iterative.agu  
Validation successful!  
  
let power (base, exponent) : (Int, Int) -> Int =  
  let result : Int = 1  
  while (exponent > 0) do  
    set result = (result * base)  
    set exponent = (exponent - 1);  
  result  
  
let main (__) : (Unit) -> Unit =  
  print(power(2, 6))  
  
Código LLVM gerado em: test/valid/tcomp000_power-iterative/power-iterative.ll  
(venv) ricardo@MacBook-Pro-de-Ricardo aguda-testing-m4-local %
```

Para além, disso também foram criados outros três ficheiros Python para auxiliar na comparação dos resultados do compilador do professor. Esses ficheiros são:

1 run_valid.py

Que corre os testes validos, tal como o ficheiro run_all_valid_tests.py, no entanto em vez de apresentar o tipo de erro ou imprimir o AST do programa, limita-se a indicar se o ficheiro .agu passa na verificação do compilador, imprimindo  OK, para os testes que passam na verificação, e  ERROR para os testes que não passam na verificação.

Para correr este ficheiro pode usar este comando:

```
python3 run_valid.py
```

```
Syntax Error: Syntax error at line 7, column 22 near '  
(venv) venricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 run_valid.py
```

Resultando em algo deste tipo:

```
=== Running 54394_identifier/identifier.agu ===  
✓ OK  
  
=== Running 54394_string_printing/string_printing.agu ===  
✓ OK  
  
=== Running 54394_combinations/combinations.agu ===  
✗ ERROR  
  
=== Running 54394_bool_array/bool_array.agu ===  
✓ OK  
  
=== Running 54394_zip/zip.agu ===  
✗ ERROR  
  
=== Running 54394_array_operations/array_operations.agu ===  
✓ OK
```

No entanto, o intuito da criação deste ficheiro Python, era usar os resultados (expressos de um modo mais simples) para comparar com os resultados do ficheiro log do compilador do professor (*test/test.log*). Sendo assim, sempre que este *run_valid.py* foi usado, utilizou-se este comando:

```
python3 run_valid.py > meu_teste_output.log
```

```
(venv) venvricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 run_valid.py > meu_teste_output.log
```

Para que o resultado do ficheiro *run_valid.py*, fosse impresso num ficheiro log, denominado *meu_teste_output.log*, que posteriormente é usado para fazer a comparação.

2 compare_with_log.py

Este ficheiro é que é responsável por fazer a comparação entre o ficheiro de log *meu_teste_output.log* e o *test.log* (o ficheiro de log do professor), indicando o número de testes .agu que o meu compilador e o compilador do professor obtêm o mesmo resultado. Para correr este ficheiro basta usar este comando:

```
python3 compare_with_log.py
```

```
✗ 10 testes incorretos  
(venv) venvricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 compare_with_log.py
```

E ele retorna algo deste tipo:

```
tcomp000_evenOddResult/evenOddResult.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_expIf/expif.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_if_then_if/if_then_if.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_intFuns/intFuns.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_letOfLet/letOfLet.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_letOfLetFun/letOfLetFun.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_letPlusLet/letPlusLet.agu: ✗ ERROR (esperado ✓, obtido ✗)
tcomp000_notNotNot/notNotNot.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_power-iterative/power-iterative.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_power-of-power/power-of-power.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_sumThree/sumThree.agu: ✓ OK (esperado ✓, obtido ✓)

Resumo Final:
✓ 116 testes corretos
✗ 20 testes incorretos
```

De acordo com a figura acima, conseguimos perceber que o compilador é capaz de avaliar 171 ficheiros .agu, da mesma maneira que o compilador do professor), no entanto, não avaliou 24 ficheiros corretamente. Estes erros estão relacionados com problemas das fases passadas.

Falando dos testes inválidos, apenas não consegui obter o resultado esperado no ficheiro *test/invalid-semantic/64854_adding_string_int/adding_string_int.agu*, pois trata-se do mesmo problema do “;”, que tive na terceira fase do projeto.

```
let a : String = "a";
let x : Int = 1;
-- type checking problem
let res : String = a + x
```

O resultado obtido:

```
=== Running test/invalid-semantic/64854_adding_string_int/adding_string_int.agu ===
[DEBUG] Unexpected token: LexToken(SEMICOLON,';',3,53)
[DEBUG] Expected something like: declaration
Syntax Error: Syntax error at line 3, column 21 near ';'

```

Neste ficheiro era esperado que o compilador retornasse algo como:

Operator '+' needs Int operands

Expected type Int, got String

E no ficheiro *test/invalid-semantic/64371_variable_not_in_scope/variable_not_in_scope.agu*

```

1  -- Author: 64371, Ricardo Costa
2
3  let main: Unit =
4      let n: Int = 13;
5      let odd: Bool = (
6          let even: Bool = n % 2 == 0;
7          !even
8      );
9      print(odd);
10     print(even) -- semantic error: 'even' is not declared in this scope

```

Devido ao problema do scope descrito anteriormente, não foi possível retornar o resultado esperado.

O resultado obtido:

```

=== Running test/invalid-semantic/64371_variable_not_in_scope/variable_not_in_scope.ag
u ===

let main : Unit =
let n : Int = 13
;
let odd : Bool =
let even : Bool = ((n % 2) == 0)
; !even
; print(odd); print(even)

Valid!

```

Resultado esperado:

```

=== Running test/invalid-semantic/64371_variable_not_in_scope/variable_not_in_scope.ag
u ===
Semantic Error: Error: (9,11) Undefined variable 'odd'
odd
Error: (10,11) Undefined variable 'even'
even

```

3 run_valid_tests_expected.py

Com este ficheiro é possível correr os testes válidos e comparar o resultado gerado a partir dos ficheiros .ll (os quais são gerados pelo compilador), e o valor esperado que estão presentes nos ficheiros .expect. Para correr este ficheiro python basta correr este comando:

```
python3 run_valid_tests_expected.py
```

```
(venv) ricardo@MacBook-Pro-de-Ricardo aguda-testing-m4-local % python3 run_valid_tests_expected.py
```

É de notar que este ficheiro percorre o ficheiro test.log fornecido pelo professor, e verifica para quais testes válidos foi possível gerar código LLVM. Isto foi necessário pois muitos colegas ainda não tinham movido os ficheiros válidos (que continha Strings, arrays, etc) para a respetiva pasta valid-not-implemented, o que acabava por comprometer a análise dos resultados.

Este ficheiro python retornar o resultado da comparação, assinalando como “Pass” caso tenha obtido o mesmo valor que está no ficheiro .expect do respetivo ficheiro teste, e “Fail” caso tenha obtido um valor diferente do esperado (neste caso apresenta o resultado obtido e

o esperado), e ainda no caso de não conseguir correr o ficheiro aguda, devido a algum erro (sintático, semântico ou até do próprio codegen.py), apresenta a mensagem “erro ao compilar”. Posto isto, ao correr o ficheiro run_valid_tests_expected.py, devemos obter algo assim:

```
[PASS] evenOddCond.agu -> true
[PASS] evenOddResult.agu -> true
[PASS] expif.agu -> false
[PASS] if_then_if.agu -> unit
[PASS] intFuns.agu -> 30
[FAIL] letOfLet.agu
  Expected: unit
  Got      : 5
[FAIL] letOfLetFun.agu
  Expected: unit
  Got      : 5
[FAIL] letPlusLet.agu - erro ao compilar
[PASS] notNotNot.agu -> false
[PASS] power-iterative.agu -> 64
[PASS] power-of-power.agu -> -1712324607
[PASS] sumThree.agu -> 18

Resumo dos Resultados:
✓ Passaram: 106
✗ Falharam: 8
Total      : 114
```

É também interessante apontar que, segundo o ficheiro test.log, o compilador do professor validava e gerava código LLVM para o ficheiro test/valid/64854_heart/heart.agu, que contém um array de strings, e, portanto, não devia gerar código LLVM. O ficheiro em questão:

```
-- Author: 64854, Diogo Almeida

let giantHeart(_) : Unit -> Unit =
  let heart_matrix : String[] = new String[13 | ""]; -- 17 chars
  let i : Int = 0 ;
  while i < length(heart_matrix) do
    if i == 0 then print("  _ _ _ _ _ \n") ;
    if i == 1 then print(" / \ / \ \n") ;
    if i == 2 then print(" / \ / \ \n") ;
    if i == 3 then print("|   -   |\n") ;
    if i == 4 then print("|       |\n") ;
    if i == 5 then print(" \       / \n") ;
    if i == 6 then print(" \       / \n") ;
    if i == 7 then print(" \       / \n") ;
    if i == 8 then print(" \       / \n") ;
    if i == 9 then print(" \       / \n") ;
    if i == 10 then print(" \       / \n") ;
    if i == 11 then print(" \       / \n") ;
    if i == 12 then print(" \       / \n") ;
    set i = i + 1
  )

let main(_) : Unit -> Unit =
  print("Love Aguda!\n");
  giantHeart(unit)
```

O resultado do meu compilador ao tentar correr este ficheiro:

```
False NotImplementedError(msg)
NotImplementedError:
Error: (4,5) Not implemented: Generating code for type 'String[]'
let heart_matrix : String[] = new String[13 | ""]; -- 17 chars
o (venv) ricardo@MacBook-Pro-de-Ricardo aguda-testing-m4-local %
```

Valid not implemented

Para estes ficheiros de teste válidos, o compilador também apresenta mensagens ao corrê-los. Para estes casos, é apresentada uma mensagem indicando que algo não foi implementado. Vejamos o seguinte caso do ficheiro `test/valid-not-implemented/56334_bubbleSort/bubbleSort.agu`:

```
An unexpected error occurred:
Error: (3,1) Not implemented: Generating code for type 'Int[]'
let buildArray(_) : Unit -> Int[] =
Traceback (most recent call last):
  File "/Users/ricardo/Desktop/fcul/Técnicas de Compilação/aguda-testing-m4-local/src/aguda-cabal/aguda
e 43, in main
    generate_llvm_code(ast_root, validator, llvm_output_filename)
    ~~~~~
  File "/Users/ricardo/Desktop/fcul/Técnicas de Compilação/aguda-testing-m4-local/src/aguda-cabal/aguda
line 1022, in generate_llvm_code
    llvm_return_type = code_generator.get_llvm_type(aguda_return_type, node=decl)
  File "/Users/ricardo/Desktop/fcul/Técnicas de Compilação/aguda-testing-m4-local/src/aguda-cabal/aguda
line 48, in get_llvm_type
    raise NotImplementedError(msg)
NotImplementedError:
Error: (3,1) Not implemented: Generating code for type 'Int[]'
let buildArray(_) : Unit -> Int[] =
(von) ricardo@MacBook-Pro-de-Ricardo aguda-testing-m4-local % cd src/aguda-cabal/aguda/; ls -la
```

Em alguns destes ficheiros de testes, são apresentados mensagens de erros de sintaxe e semântica, pois como já indiquei anteriormente o projeto apresenta alguns problemas relacionados às fases anteriores.

Short-circuit boolean expressions

Nesta fase do compilador AGUDA, os operadores booleanos `&&` e `||` foram implementados com comportamento de short-circuit, dado que o LLVM não possui operadores lógicos com short-circuit, a estratégia adotada baseou-se na geração explícita de blocos de controlo e instruções phi.

Operador &&

A expressão ‘a && b’ é transformada da seguinte forma:

1. Avalia-se ‘a’ (LHS).
2. Se ‘a’ for falso, salta-se diretamente para o bloco de merge com valor false.
3. Se ‘a’ for verdadeiro, avalia-se ‘b’, e o seu valor será o resultado final.

Isto é realizado em LLVM com:

- um bloco ‘and_rhs’ onde se avalia o RHS,
- um bloco ‘and_merge’ onde se combina o resultado com phi.

Operador ||

De forma simétrica, para ‘a || b’:

1. Avalia-se 'a'.
2. Se 'a' for verdadeiro, salta-se diretamente para o merge com valor true.
3. Caso contrário, avalia-se 'b'.

Também são gerados blocos 'or_rhs' e 'or_merge' com o respetivo phi.

Erros e problemas do projeto

A maioria se não todos os testes que falham na avaliação semântica e na geração de código deve-se ao facto de não ter implementado o validator com o padrão ABC e consequentemente o codegen (o qual trata da geração de código LLVM), que acabou por ser afetado.

Esse erro mostrou-se fatal, pois não fui capaz de corrigir certos problemas de outras fases. Caso tivesse mais tempo, teria feito um refactor do meu validator antes de prosseguir para o codegen, mas como não foi possível, decidi deixar o código desta maneira.

Alguns dos problemas do meu código:

Erros no parser (possivelmente) e validator

Problema de escopos (muito provavelmente porque não foi implementado o padrão ABC)

Avaliação incorreta do tipo dos branches em IF's

Estes problemas não permitiram que o meu compilador aguda executa-se com êxito todos os testes, quer válidos, não-válidos e validos-não-implementados.