



**Ciências
ULisboa**

Relatório do Projeto Aguda Compiler

Milestone 3

Ricardo Miguel Neto Sobral 56332

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
```

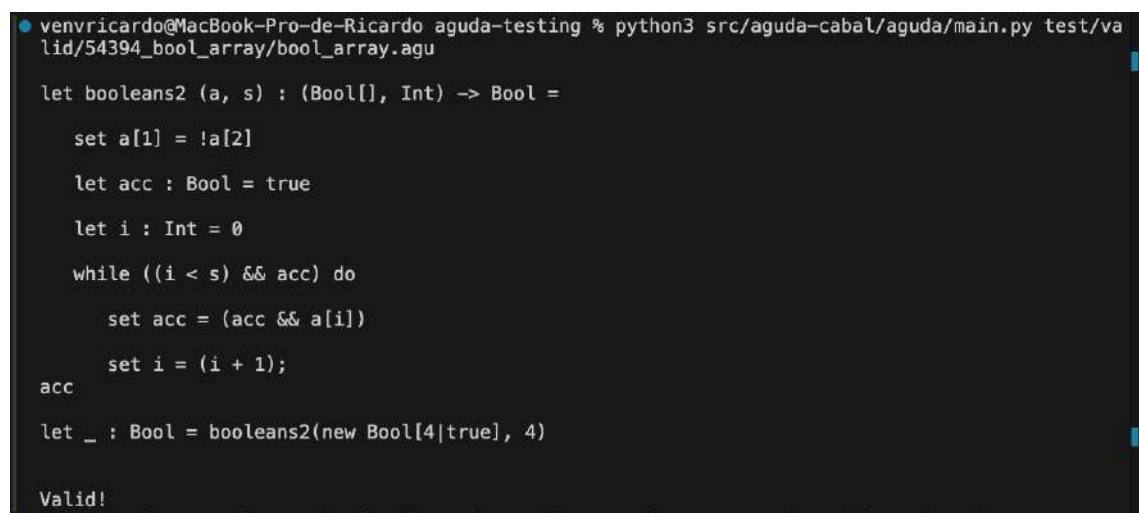
E agora está tudo pronto para correr o compilador!

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:



```
venvricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 src/aguda-cabal/aguda/main.py test/valid/54394_bool_array/bool_array.agu

let booleans2 (a, s) : (Bool[], Int) -> Bool =

  set a[1] = !a[2]

  let acc : Bool = true

  let i : Int = 0

  while ((i < s) && acc) do
    set acc = (acc && a[i])
    set i = (i + 1);
  acc

let _ : Bool = booleans2(new Bool[4|true], 4)

Valid!
```

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

Para correr vários testes temos várias opções:

- 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:

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

- 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:

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

- 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:

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

-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:

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

- 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_aguda/factorial_in_aguda.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.

E no 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 indica se o ficheiro é válido. Vejamos o exemplo:

```
venvricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 src/aguda-cabal/aguda/main.py test/valid/54394_bool_array/bool_array.agu

let booleans2 (a, s) : (Bool[], Int) -> Bool =
  set a[1] = !a[2]

  let acc : Bool = true

  let i : Int = 0

  while ((i < s) && acc) do
    set acc = (acc && a[i])
    set i = (i + 1);
  acc

let _ : Bool = booleans2(new Bool[4|true], 4)



Valid!
```

O conteúdo do ficheiro .agu em questão:

```
4 let booleans2 (a, s) : (Bool[], Int) -> Bool =
5
6     set a[1] = !a[2]; -- change one value for testing
7
8     let acc : Bool = true;
9     let i : Int = 0;
10    while i < s && acc do (
11        set acc = acc && a[i];
12        set i = i+1
13    );
14    acc
15
16 let _ : Bool = booleans2(new Bool[4 | true], 4)
17
```

Para além, disso também foi criado outros dois ficheiros Python para auxiliar na comparação dos resultados do compilador criado pelo professor. Esses ficheiros são:


-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:


```
(venv) venvricardo@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:

```
(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.

- *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:

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

E ele retorna algo deste tipo:

```
64371_two_sum/two_sum.agu: ✓ OK (esperado ✓, obtido ✓)
64854_add/add.agu: ✓ OK (esperado ✓, obtido ✓)
64854_chess_matrix/chess_matrix.agu: ✓ OK (esperado ✓, obtido ✓)
64854_coin_flip/coin_flip.agu: ✓ OK (esperado ✓, obtido ✓)
64854_heart/heart.agu: ✓ OK (esperado ✗, obtido ✗)
64854_inc/inc.agu: ✓ OK (esperado ✓, obtido ✓)
64854_is_pair/is_pair.agu: ✓ OK (esperado ✓, obtido ✓)
64854_positive_power/positive_power.agu: ✓ OK (esperado ✓, obtido ✓)
64854_printA/printA.agu: ✓ OK (esperado ✓, obtido ✓)
64854_produce_letter/produce_letter.agu: ✓ OK (esperado ✓, obtido ✓)
64854_switch/switch.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_arrayOfUnit/arrayOfUnit.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_assoc_eq/assoc_eq.agu: ✗ ERROR (esperado ✓, obtido ✗)
tcomp000_collatz/collatz.agu: ✗ ERROR (esperado ✓, obtido ✗)
tcomp000_diagonal/diagonal.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_double_wild/tcomp000_double_wild.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_evenOdd/evenOdd.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_expIf/expIf.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_if_then_if/if_then_if.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_letOfLet/letOfLet.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_lookup/lookup.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_lookupRec/lookupRec.agu: ✓ OK (esperado ✓, obtido ✓)
tcomp000_powers/powers.agu: ✓ OK (esperado ✓, obtido ✓)

Resumo Final:
✓ 166 testes corretos
✗ 16 testes incorretos
```

De acordo com a figura acima, conseguimos perceber que o compilador é capaz de avaliar 166 ficheiros .agu, da mesma maneira que o compilador do professor), no entanto, não avaliou 16 ficheiros da mesma maneira, e é através destes 16 ficheiros que fui capaz de entender várias coisas interessantes.

Em 15 desses 16 ficheiros .agu, que o meu compilador não obteve o mesmo resultado do professor deve-se a um problema de parsing deliberado. Vejamos um exemplo:

```
└─ Author: fc64361, Gustavo Henriques

let fact (n) : Int -> Int =
  if n == 0 then 1 else n * fact(n - 1)

let _ : Unit =
  print(fact(5)); -- Should print 120
  print(fact(0)); -- Should print 1
  print(fact(7))  -- Should print 5040
```

Neste ficheiro (*test/valid/64361_factorial/factorial.agu*), o meu compilador lança o seguinte erro de sintático:

```
venvricardo@MacBook-Pro-de-Ricardo aguda-testing % python3 src/aguda-cabal/aguda/main.py test/valid/64361_factorial/factorial.agu
[DEBUG] Unexpected token: LexToken(SEMICOLON,',';',7,143)
[DEBUG] Expected something like: declaration
Syntax Error: Syntax error at line 7, column 19 near ';'
venvricardo@MacBook-Pro-de-Ricardo aguda-testing %
```

Este erro não devia acontecer, mas dada a estrutura do meu parser para tratar de casos, como sequencias de instruções dentro de um while ou if, ele não é capaz de lidar com ; em casos como de *let x : unit = print("A");*

```
5
6 let _ : Unit =
7   print(fact(5)); -- Should print 120
8   print(fact(0)); -- Should print 1
9   print(fact(7))  -- Should print 5040
```

Este erro sintático ocorre nestes 15 ficheiros:

```
tcomp000_collatz/collatz.agu
test/valid/56311_empty_array/empty_array.agu
test/valid/64361_greatest_common_divisor/greatest_common_divisor.agu
test/valid/64361_fibonacci/fibonacci.agu
test/valid/64361_factorial/factorial.agu
test/valid/58185_string_concatenation/string_concatenation.agu
test/valid/58185_simple_arithmetic/simple_arithmetic.agu
test/valid/58185_sequencing/sequencing.agu
test/valid/58185_power_operator/power_operator.agu
```

```
test/valid/58185_mutual_recursion/mutual_recursion.agu  
  
test/valid/58185_boolean_logic/boolean_logic.agu  
  
test/valid/58170_sameNameVar/sameNameVar.agu  
  
test/valid/58170_pyramid/pyramid.agu  
  
test/valid/58170_categorizeNumbers/categorizeNumbers.agu  
  
test/valid/58166_pingPong/pingPong.agu
```

Até foi possível corrigir este erro sintático, no entanto ao corrigir este erro, era levantado erros semânticos de variáveis não declaradas, quando entrávamos em blocos while e if.

```
3  let arraySum (a) : Int[] -> Int =  
4      let i : Int = 0;  
5      let acc : Int = 0;  
6      while i < length(a) do {  
7          set acc = acc + a[i];  
8          set i = i + 1  
9      };  
10     acc
```

Por exemplo neste ficheiro, o compilador passava a indicar que a variável *i*, que tinha sido inicializada na linha 4, não existia, quando chegávamos à linha 6, a linha onde o while começava. Isto devia-se a um problema de scope. O que acontecia, é ao chegar a um bloco while ou if, a variável “*i*” já não pertencia ao scope. Isto resultava de dois possíveis problemas, o primeiro que tem a ver com a adição indevida do novo scope, que em vez de criar um scope fazendo uma cópia do antigo, criava apenas um scope vazio, ou seja {}. E o segundo e mais relevante, tratava-se de um problema relacionado com o ast que o parser passava ao validator, o que fazia com que as primeiras variáveis de algumas funções desaparecessem do scope da função. Infelizmente não consegui resolver esse problema de comunicação entre o parser e o validator, porque gastei tempo em vão, a tentar encontrar uma sofisticada de como encontrar o final do corpo da função através da indentação do código. Posto isto, e apesar de estar ciente do problema, decidi deixar o parser como estava (erro do let .. unit = .. ;), mesmo sabendo que iria causar problemas com aqueles 15 ficheiros de teste, que foram descritos acima, pois ainda tinha outros aspetos dos projetos para tratar.

No ficheiro *valid/tcomp000_assoc_eq/assoc_eq.agu*, o ficheiro log indica que teste é um teste válido, no entanto se observarmos bem o ficheiro, e lermos as instruções dadas na primeira fase do projeto, conseguimos perceber que existe um problema com o ficheiro.

```
-- Author: tcomp000, Vasco T. Vasconcelos  
  
let _ : Unit =  
    let b : Bool = true ;  
    print (true == b == (false || true))
```


Regras de associatividade apresentadas no ficheiro *aguda.pdf*:

Operator precedence and associativity

- For `+` `-` `*` `/` `%` `==` `!=` `<` `<=` `>` `>=` `!` `||` `&&` take the precedence and associativity of the Java programming language

Em Java não é possível fazer operações lógicas como `a == b == c`, pois o sinal “`==`” não é associativo em java, e por isso precisamos fazer `(a == b) == (b == c)`. Posto isto, não devia ser possível fazer `true == b == (false || true)`), e assim como em Java devia ser marcado com um erro de sintaxe. Repare no que o meu compilador retorna:

```
venhricardo@macbook-pro-de-ricardo-aguda-testing % python3 src
/aguda/main.py test/valid/tcomp000_assoc_eq/assoc_eq.agu
[DEBUG] Unexpected token: LexToken(EQ,'==',5,101)
[DEBUG] Expected something like: expression
Syntax Error: Syntax error at line 5, column 20 near '=='
```

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 “;”, dos outros 15 ficheiros válidos descritos acima.

```
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_in
t.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

```

How to change the max number of errors to be printed

Para alterar o número máximo de erros apresentados por ficheiro, pelo compilador é simples. No ficheiro `src/aguda-cabal/aguda/main.py`, basta alterar o valor atribuído a `max_erros` na linha 36. É importante frisar, que esse valor nunca deve ser 0, pois assim, o validator não faz a sua avaliação e todos os ficheiros com erros semânticos são considerados válidos.

```

35  # Validação semântica
36  validator = Validator(max_errors=5) # valor ajustável, nunca deve ser 0!
37  validator.source_lines = source_code.splitlines()
38  validator.validate(ast_root)

```

A brief description of how you implemented the symbol table

A *symbol table* foi implementada como uma *pilha de dicionários*.

Cada dicionário representa um *scope* (por exemplo, o corpo de uma função, um bloco dentro de um `if` ou `while`, etc.).

A pilha de scopes está presente na classe `Validator` como `self.ctx`, e é inicializada como `[{}]`. Para entrar num novo *scope* (por exemplo, dentro de uma função) adiciona-se um novo dicionário no topo da pilha através de `self.ctx.append({})`. E quando saímos do *scope* o dicionário é removido com `self.ctx.pop()`.

Quando é feita a procura de uma variável, a pesquisa é feita de baixo para cima, ou seja, do *scope* “mais específico” para o mais “global”.

As funções são todas geridas através de uma tabela global, denominada `self.functions`, que associa o nome da função aos seus tipos (de parâmetros e retorno).

A brief description of how you implemented bidirectional type checking (or why you did not follow this approach)

No ficheiro `validator.py` foi implementada, *bidirectional type checking*, tal como foi sugerido no enunciado.

Neste `validator.py` podemos encontrar:

- `typeof(expr)` que infere o tipo de uma expressão
- `checkAgainst(expr, expected_type)` que verifica se uma expressão tem um determinado tipo

O `typeof` é responsável percorrer a AST e “calcular” o tipo de cada expressão. Já o `checkAgainst` chama `typeof(expr)` para obter o tipo calculado, para compará-lo com o tipo esperado. Caso os sejam diferentes deve ser reportado um erro, e como é obvio existem inúmeros casos de erros possíveis, portanto foram implementados vários casos, que permitiam retornar mensagens de erro claras e precisas. Por exemplo para operações aritméticas, o validator (em `BinOp`), verifica se os dois operados são do tipo `Int`. Caso não sejam é retornada uma mensagem deste tipo: `"Operator '+' needs Int operands"`.

A maneira como o validator foi implementado respeita o *bidirectional type checking*, o que permite obter mensagens de erro mais compreensíveis e detalhadas para o utilizador.