



**UNIDADE CURRICULAR:** Compilação

**CÓDIGO:** 21018

**DOCENTE:** Professor Jorge Morais

**NOMES e N.º DE ESTUDANTE:** (Alt + C ++ Elite)

- Gonçalo Caraça, 2000130
- Inês Oliveira, 2001090
- Joana Martins, 2003351
- João Carvalho, 2103537
- Mário Carvalho, 2000563

**CURSO:** Licenciatura em Engenharia Informática

**DATA DE ENTREGA:** 14 de junho de 2023

## TRABALHO / RESOLUÇÃO:

Tendo em conta que o relatório final trata o conjunto do trabalho realizado ao longo do semestre, procedeu-se a uma explicação mais detalhada do processo relativo às etapas de construção de um compilador para linguagem YAIL. Assim, após análise e avaliação das possibilidades de linguagem e preferências dos elementos do grupo, escolheu-se recorrer à **linguagem C** e às ferramentas **Flex** e **Bison**. Pois considerou-se ser um conjunto de ferramentas poderosas e flexíveis para o desenvolvimento de um compilador, dado que automatizam muitas das tarefas necessárias para este processo, facilitando a implementação do código executável final.

Para melhor esclarecimento do processo realizado, descreve-se a seguir, as etapas necessárias até à sua conclusão, começando pela definição do comando criado para executar o código final, passando pelas: Análise Léxica (AL), Análise Sintática (AS), Geração de Código Intermédio (GCI) - considerada como Análise Semântica, Otimização Simples (OS), Otimização Avançada (OA), Geração de Código Final (GCF) e os ficheiros de teste.

Através da execução do comando `./compileGlobal.sh`, foi possível gerar o ficheiro final que interpreta a sequência de *tokens* definida no ficheiro da análise sintática, que é interpretado à luz das regras atribuídas e identificadas no ficheiro `lexG.l`. Sendo este, posteriormente utilizado pela ferramenta **Bison**. Isto é, o ficheiro consequente do `lexG.l`, após execução do comando associado à ferramenta **Flex** permite obter um ficheiro `lex.yy.c` que é usado posteriormente pelo GCC.

O **Bison**, por si só, gera a função `yyparse()`, que chama automaticamente a função `yylex()`, que é gerada pelo **Flex**. A função `yyparse()` está definida no ficheiro `.tab.c` criado pelo ficheiro `.y`, enquanto que a função `yylex()` está no ficheiro `lex.yy.c` gerado anteriormente pelo ficheiro `.l`. Como esses dois ficheiros (`.l` e `.y`) são compilados independentemente e depois vinculados, o ficheiro `.y` inclui a função `yylex()` definida para evitar erros de compilação, através da chamada deste ficheiro diretamente na análise sintática.

No entanto, para garantir que o **Bison** seja executado antes do **Flex** (para gerar o arquivo ".tab.h" com os *%tokens* e outras definições), declarou-se essa ordem no ficheiro "compileGlobal.sh":

```
$ compileGlobal.sh
1  # para executar as novas melhorias
2  # o rm vai ajudar a detetar se existem erros no syntax ou no flex
3  # o syntax.c já tem o include do lex.yy.c já não precisa de incluir no gcc call
4
5  clear
6  rm syntaxG.c syntaxG.h lex.yy.c
7  bison -d syntaxG.y -o syntaxG.c
8  flex lexG.l
9  gcc syntaxG.c -o global
10 ./global YAILsemErros.txt
```

Figura 1 - Comando bash criado para automatização da compilação do código

Durante a construção da análise léxica recorreu-se à construção de caracteres literais e de palavras-chave imprescindíveis para a identificação de "*tokens*", após leitura do texto de entrada. Em parte, recorreu-se ao uso de regras estipuladas em "*Regular Expressions*" (ou "*RegEx*"), que permitissem produzir sequências de palavras (a partir de símbolos, números, ...) subsequentemente identificadas na Análise Sintática. Como exemplo, tem-se por objetivo que o carácter "#" indique constantemente que seja o início de um comentário. Outras discussões foram debatidas para melhor construir *RegEx* que identificassem, números inteiros ou reais, booleanos e nomes de funções:

```
paragrafo [\r\n]
comentario (#.*)
identificador [_a-zA-Z_]+([0-9]?|[_a-zA-Z_]?)*
inteiro [-]?[0-9]+
real [-]?[0-9]+(\.[0-9]+)?
booleano (false)|(true)
```

Figura 2 - Exemplo de parte de código correspondente à análise léxica

De seguida, procedeu-se à construção de regras gramaticais que obedecem a critérios de escrita do YAIL. Recorrendo a leitura da sequência dos "*tokens*" gerados anteriormente pela análise léxica, verifica-se assim, se a escrita de código segue a gramática da linguagem pedida, recorrendo à ferramenta **Bison**, para obter o ficheiro executável (".y"), e ou identificação de erros.

Nesta primeira etapa (e para entrega do E-Fólio A), os ficheiros finais obtidos (executáveis e compilados) apresentavam-se como promissores, até se começar

a implementar a parte responsável pela Análise Semântica (tópico abordado de seguida). Durante este tempo de espera de feedback da avaliação, como primeira etapa corrigiu-se: o sinal “menos” que não funcionava como negativo para a realização de todos o tipo de cálculos e as condições para a construção dos vetores dentro de constantes. De seguida avançou-se para a análise semântica.

A Análise Semântica é responsável por verificar se o programa segue as regras semânticas da linguagem no que respeita à sua estrutura, e a uma lógica gramatical. É graças a esta parte do código que se consegue identificar possíveis erros, compatibilidades ou o uso repetido de variáveis. Para isso, ficando à responsabilidade do **Bison**, procedeu-se à inclusão de código em C diretamente no mesmo ficheiro (“.y”) para que a chamada às funções respetivas de avaliação, fosse estabelecida. Deste modo, foi criada uma Árvore Sintática Abstrata (AST) que representa a estrutura do programa, que será utilizada no passo seguinte.

O avanço nesta fase foi mais moroso, que o início da construção da Análise Léxica, pois o nível de compreensão global de todas as etapas necessárias seguintes para a construção de um compilador, não se encontravam claras. Este foi um dos pontos de dificuldade sentido pelo grupo, dado faltar ainda conhecimento sobre que direção tomar. Se até aqui, foi possível distribuir trabalho por cada elemento (e cruzar ideias e dados), a partir daqui a estratégia de trabalho mudou para um encontro regular em pequenos grupos, na tentativa de recuperar eficiência na escrita e compreensão da forma como cada parte comunica, entre os diferentes componentes.

Ainda que até este ponto, o avanço não tenha sido o desejado, já se tinha, apesar de tudo, conseguido alcançar o suficiente para que, após feedback do E-Fólio A, o impacto de uma nova reestruturação da linguagem YAIL provocasse um atraso na construção final da Análise Semântica. Este tornou-se então o novo momento desafiante no processo de construção da árvore, dado que implicou investimento de tempo e energias, numa nova maneira de pensar e uma implementação mais adaptada à linguagem, que não fora até então tida em conta. Como tal, deu-se início a uma nova Análise Lexical, Sintática e Semântica, que foi entregue no E-Fólio B, apresentando algumas faltas na Análise Lexical, algumas incoerências na Análise Sintática, e, conseqüentemente, uma Análise Semântica incompleta.

Na reestruturação do E-Fólio B, reavaliou-se o que já tinha sido construído. Pois durante o E-Fólio A tentou-se construir uma linguagem YAIL que fosse mais abrangente, à semelhança de outras linguagens de programação. Onde por exemplo se destacam a construção de ciclos dentro de ciclos (*for/while*). Mas, segundo a correção do enunciado para o E-Fólio B, YAIL acabou por se revelar uma linguagem mais restrita e menos dinâmica. Isto, exigiu um esforço redobrado, o que obrigou a um aumento de realização de testes específicos complementares para garantir a sua viabilidade, além da alteração do que já fora construído. Como exemplo tem-se os dois ficheiros para verificação da escrita de YAIL.

```

1      structs {
2          alasd{int x y
3              # geradores
4              int g[10] = gen(1,9);
5              int g[a + 1] = gen(1,9);
6              int g_1[a + 2] = gen(0,9);
7              int g1[a + 3] = gen(2,0);
8              int g[a + 4] = gen(0,1);
9              int g_2[a + 5] = gen(9,-1); # decrementar
10             int g2[10] = gen(2,0);
11             int Gcalculo[a * 2] = gen(1,9);
12             int Gcalculo1[a / 2] = gen(2,0);
13             int Gcalculo_1[a - 2] = gen(0,9);
14             a1 { y, t, r; };
15         }
16     }
17     int g[10] = gen(1,9);
18     int g[a + 1] = gen(1,9);
19
20     structs #inicio
21     point2D { x,y; }; # point2D é composta por 2 reais
22     point2D { x;y, }; # point2D é composta por 2 reais
23     point3D { float }; # point3D por 3 reais x, y e z
24     point3D { tni }; # point3D por 3 reais x, y e z
25     pointND {[]} ##### pointND é um vetor de reais
26 }
27 struct{
28     # definição # # # # estruturas
29 }
30
31 # constantes

```

```

1      int a = 20;
2
3      structs {
4          # structs
5          a { int q ;};
6          a { float w ;};
7          a { bool e ;};
8          a1 { bool y, t, r; };
9
10         b { int v[]; };
11         b { int v[2]; };
12         b { int v[2+a]; };
13         b { int v[2+1]; };
14         b { int v[2+1*2]; };
15
16         c { int vetor [] = {1};};
17         c { int vetor [] = {1,2};};
18         c { int vetor [] = {1};};
19         c { int v_2[10] = {1,2};};
20         c { int v_b[a] = {1,2,3};};
21         c { int v_bb[a + 2] = {1,2,3};};
22         c { int v_B1[a + 2] = {1,2};};
23         c { int v_b1[a + 2] = {2,3};};
24         c { int v_b_3[a + 2] = {1,2,3};};
25         c { int v_b_4[a + 2] = {1,2,3};};
26         #c { int v_bb[a - 2] = {1,2,3};};
27         #c { int v_B1[a * 2] = {1,2};};
28         #c { int v_b1[a / 2] = {2,3};};
29         #c { int v_b_3[a % 2] = {1,2,3};};
30         #c { int v_b_3[a * 2 - 2] = {1,2,3};};
31         #c { int v_b_3[a / 2 + 2] = {1,2,3};};

```

Figura 3 - Ficheiros de testes criados (com erro e sem erro) para verificar as análises léxica e sintática

Ainda durante a fase de implementação do código para o E-Fólio B, foram considerados os detalhes indicados como melhorias a ter em conta, provenientes do E-Fólio A. Recorrendo-se para isso, também à utilização do comando “win\_bison --report=all --warninfgs=all --debug -o %(Filename.cpp)”, que permitiu identificação de erros e tratamentos de mensagens de aviso e ambiguidades. De entre estas, destaca-se a correção:

- Da forma “*RegEx*” para atribuir nome às funções, dada por “IDENT”;
- De cálculos de números negativos, em que o sinal matemático não era interpretado como subtração, quando escrito como número negativo (exemplo:  $15 * (-3)$ );
- Da forma de construir constantes;
- Das ambiguidades entre “*declara\_variavel*” e “*declaracao\_atribuicao*”, e de todas as condições de “*while*”, e “*if/else*”;
- Da possibilidade de ciclos “*for*” ou “*while*”, dentro uns dos outros;
- Da forma de usar os operadores lógicos e os mesmos quando repetidos para indicar incrementação “++”, por exemplo;
- Da implementação ordenada dos comparativos algébricos, para funcionar na ordem correta;
- Retirada do ponto final, utilizado para separar estruturas dentro de estruturas, (“q.p”, como exemplo);
- Da sequência lógica de limitar as possibilidades de construção do código: abertura do ficheiro; início de estruturas; início de constantes; início de global; início de main;

A partir desta reformulação, tentou-se ter em conta a quantidade de “*warnings*” do código redundante, para evitar possíveis “*loops*” ou incoerências que obrigassem a uma revisão posterior. Desde este momento, até à entrega do código final, priorizou-se a Geração de Código Intermediário (com Análise Semântica) e Otimização do Código Simples.

A Geração de Código Intermediário, consiste numa representação de nível mais baixo do programa, podendo ser uma representação em forma de árvore. Tendo por objetivo ser um facilitador da otimização e geração de código final, por ser nesta fase que se permite a tradução do código-fonte para uma forma intermediária, que à “*priori*”, é mais fácil para ser otimizada e traduzida posteriormente para código-máquina.

A Otimização de Código Simples representa uma etapa fundamental, visando melhorar o desempenho e eficiência do programa. Durante esta etapa são aplicadas várias estratégias com o intuito de eliminar possíveis falhas, por meio de técnicas como a análise de fluxo de dados. Na qual foram identificadas dependências entre as variáveis e as instruções, e posteriormente reduzida a

necessidade de acessos desnecessários à memória, assim como remoção de cálculos redundantes. Através da eliminação de código morto, são identificadas porções de código que não são utilizadas na execução, durante a etapa de compilação. A propagação de constantes, também se trata de uma técnica que visa substituir os valores constantes pelos respectivos resultados durante a compilação, o que permite evitar as operações de leitura de variáveis desnecessárias. Teve-se como atenção, encontrar o equilíbrio entre o resultado final pretendido, o desempenho e a legibilidade do código.

Na fase de Geração de Código Final, o código é gerado a partir da Otimização Avançada, onde as técnicas de geração, podem variar dependendo da complexidade do compilador e do alvo. O código gerado será específico para a arquitetura do processador de destino. Na fase seguinte, é necessário vincular o código-fonte com bibliotecas e outros módulos externos, para que o código compilado seja executado na máquina de destino, isto é, em *assembly*. Estas duas últimas fases, foram retiradas da avaliação final, tal como indicado na última observação efetuada no dia 2 de junho de 2023.

## DIFICULDADES GLOBAIS

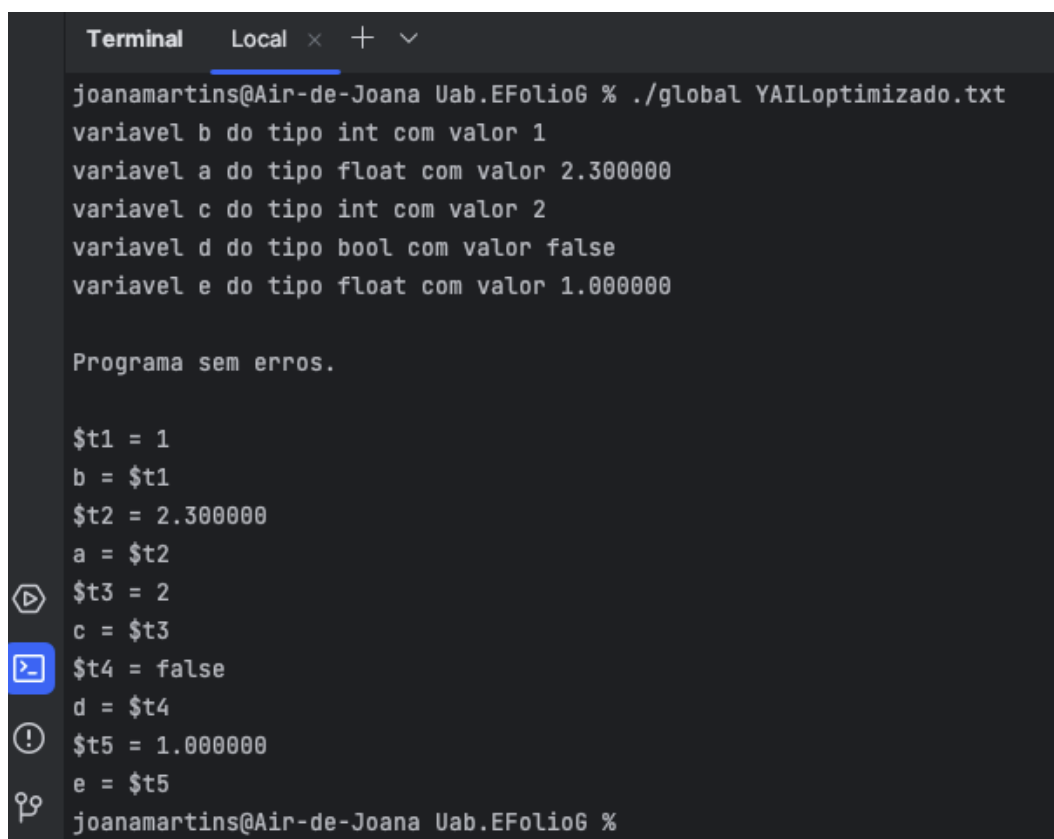
Como maneira de indicar as dificuldades sentidas ao longo da evolução da construção do compilador, estabeleceu-se o conjunto de problemas sentidos na sua globalidade. Assim, se destaca, que o iniciar da implementação do código para a análise lexical e análise sintática foi demorado, pois, não fora óbvia a maneira como se construíam os dois ficheiros (***Flex*** e o ***Bison***). Tal como a compreensão dos seus erros.

Ainda que inicialmente se tenha considerado escrever em linguagem C++, por simplicidade de utilização e de chamada de ficheiros, a escolha direccionou-se para linguagem C. O desenvolvimento do código em C, permitiu que, a sua construção no ficheiro de Análise Sintática, facilitou na compreensão sobre como o ***Bison*** lê e imprime os resultados ao chamar a função respetiva. Assim, se conseguiu adaptar as mensagens de alerta e de erros, tais como a identificação das linhas e o tipo de *token* a corrigir.

Durante os corretivos, entre o E-Fólio A e o E-Fólio B, para se fazer distinção das duas novas partes corrigidas e melhoradas (ficheiros antigos e ficheiros novos corrigidos), iniciou-se a construção de um novo ficheiro diretamente em “lex.l” e “syntax.y”. Assim, deu-se um novo nome aos documentos adicionando o nome “Otimizado.”.

## POR IMPLEMENTAR

Como indicado, procedeu-se ao início da árvore, e durante a reformulação e continuação do E-Fólio B, alterou-se mais uma vez a construção de algumas funções. Tal como, em vez de se tratar cada tipo de variável individualmente, começou-se por tratar todas no seu conjunto (exemplo: INT, FLOAT, BOOL) ao se chamar uma função global que coloca na pilha, por ordem de chamada da variável correspondente.



```
Terminal Local x + v
joanamartins@Air-de-Joana Uab.EFolio6 % ./global YAILOptimizado.txt
variavel b do tipo int com valor 1
variavel a do tipo float com valor 2.300000
variavel c do tipo int com valor 2
variavel d do tipo bool com valor false
variavel e do tipo float com valor 1.000000

Programa sem erros.

$t1 = 1
b = $t1
$t2 = 2.300000
a = $t2
$t3 = 2
c = $t3
$t4 = false
d = $t4
$t5 = 1.000000
e = $t5
joanamartins@Air-de-Joana Uab.EFolio6 %
```

Figura 4 – Resultado final do compilador ./global YAILOptimizado.txt em macOS

Ficando a faltar também a verificação das condições de construção necessárias para que as dimensões das “gen(tamanho)”, como para as condições dos vetores.



## CONTRIBUTOS

De modo geral, cada elemento do grupo teve o seu impacto e importância para a concretização deste projeto. Destaca-se como uma força global do grupo, o facto de cada um se manter disponível para o avançar do projeto, independentemente da facilidade ou dificuldade da tarefa. Pois o importante, foi avançar ao ritmo do projeto, dando liberdade para cada um explicar o que já tinha feito e o que faltava por fazer. Se inicialmente, foi colocada mais energia no avançar da tarefa, a partir do E-Fólio B, o importante foi avançar em grupo, dado que a revisão de erros, diagnóstico de problemas e interpretação dos erros, era mais rápida a dois ou três, do que cada um em separado. Foi importante que, neste caso, todas as pessoas estivessem em total confiança, dada a dificuldade do projeto. Assim, como todos se conheciam, foi fácil por a render em prol do grupo, as capacidades de cada um, para se obter a maior colaboração.

Assim, cada um foi sendo mais responsável por alguns parâmetros específicos, dedicando o seu tempo a algum ponto que necessitasse de mais atenção, para outro colega poder dar mais atenção a outra responsabilidade. Não tendo sido atribuída uma responsabilidade única por pessoa, mas, por confiança e conhecimento do modo de funcionamento noutros trabalhos, ficou fácil todos assumirem todas as responsabilidades circunstancialmente. Entre estas, destacam-se as responsabilidades seguintes:

- Atenção à evolução que o código tinha e a ordem do repositório no GitHub (repositório);
- Estar atento aos novos critérios de avaliação, e o que era pedido em cada momento;
- Compreensão do que consistia cada parte do compilador e como se interligava com a teoria;
- Análise e resolução de problemas e diagnósticos de erros;
- Criação de testes válidos e testes de erros, e a sua manutenção;
- Exercícios das atividades formativas e como cada parte se correlacionava com a fase seguinte a implementar no compilador;
- Correção de bugs e testes.

O repositório encontra-se disponível, mas apenas aberto depois da hora do E-Fólio global terminar no dia 14 de junho de 2023.

📁 .idea	Adding ReadMe for all "AF's"
📁 Uab.AF1	Adding ReadMe for all "AF's"
📁 Uab.AF2	Criação do compile.sh para correr os comandos
📁 Uab.AF3	AF3
📁 Uab.AF4	Correctivos On Going
📁 Uab.AF5	YAILSemErros => tem 119 erros
📁 Uab.AF6	YAILSemErros => tem 119 erros
📁 Uab.EFolioA	novo comando compile melhorias
📁 Uab.EFolioB	adicionado syntaxG.y e lexG.l
📁 Uab.EFolioG	adicionado syntaxG.y e lexG.l
📄 .gitignore	yail new version com novos vetores
📄 LICENSE	Initial commit
📄 README.md	Update README.md

Figura 5 – Exemplo do repositório usado

## DOCUMENTOS

O ficheiro `efolioG_Grupo_Alt_C_Elite.zip` contém ficheiros correspondentes à evolução para o E-Fólio Global, com o executável: `compileGlobal.sh`, `lexG.l`, `syntaxG.y`, `syntaxG.c`, `syntaxG.h`, que utiliza o ficheiro `testeYAILOtimizado.txt`. Semelhante a entrega dos restantes E-Fólios o trabalho pode ser executado através do ficheiro `compileGlobal.sh`.

Além disso, encontram-se também os mesmos ficheiros que se optou por manter para representar a evolução do E-Fólio B até ao momento, que pode ser testada através do executável, `compileEFolioB.sh`.

## BIBLIOGRAFIA

Aho, A., S. Lam, M., Ravi , S., & D. Ullman, J. (2008). *Compilers: principles, techniques and tools* (2ª ed.). (A. Wesley, Ed., & D. Vieira, Trad.) Universidade Federal de Minas Gerais, Brasil: Pearson.

Reis Santos, P., & Thibault, L. (2015). *Compiladores*. FCA.