



UNIDADE CURRICULAR: Compilação

CÓDIGO: 21018

DOCENTE: Professor Jorge Morais

A preencher pelo estudante

NOMES e N.º DE ESTUDANTE:

- Gonçalo Caraça, 2000130 (Alt + C ++ Elite)
- Inês Oliveira, 2001090 (Alt + C ++ Elite)
- Joana Martins, 2003351 (Alt + C ++ Elite)
- João Carvalho, 2103537 (Alt + C ++ Elite)
- Mário Carvalho, 2000563 (Alt + C ++ Elite)

CURSO: Licenciatura em Engenharia Informática

DATA DE ENTREGA: 02 de maio de 2023

TRABALHO / RESOLUÇÃO:

INTRODUÇÃO

De acordo com o enunciado, pretende-se, em grupo, construir um compilador para a linguagem YAIL, apresentando as análises léxica e sintática correspondentes.

Numa primeira fase escolhemos realizar um compilador em C++, mas ao longo do percurso, surgiram dificuldades de funcionalidade e devido às diversas incompatibilidades, alterou-se a linguagem para C, em consenso com todos os elementos do grupo.

Juntos, na nossa opinião, C, FLEX e BISON formam um conjunto de ferramentas poderosas e flexíveis para o desenvolvimento de um compilador, pois automatizam muitas das tarefas necessárias nesse processo, além disso, podem aumentar a eficiência e a precisão, embora num primeiro contacto a relação entre ambos apresentou dificuldades e exigiu empenho. Em suma, percebeu-se que a combinação dessas tecnologias permitia que o compilador fosse criado de forma mais rápida e eficiente, sem sacrificar o desempenho ou a qualidade do código.

No entanto, apesar de acreditarmos que a nossa escolha foi a mais adequada, o uso do FLEX e BISON apresentou algumas dificuldades. Foi necessário, conhecimento sólido em teoria da linguagem e gramática formal, para a estruturação e criação de gramáticas corretas e sem ambiguidade. Entender a correta diferença entre análise léxica e sintática, além de compreender a sintaxe dessas ferramentas.

Outra dificuldade foi o processo de depuração, identificar e corrigir erros no código, nem sempre foi tarefa fácil, no entanto, com dedicação, esforço e pesquisa, conseguimos ultrapassar essas dificuldades e aproveitar os benefícios destas ferramentas na construção do nosso compilador.

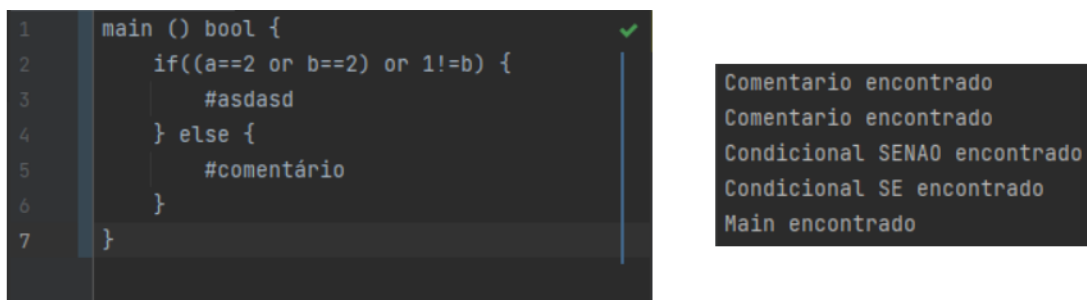
DESENVOLVIMENTO

Depois de optarmos pela escolha do FLEX, procedemos à criação da gramática necessária para a linguagem YAIL. Através do ficheiro “*lex.l*”, foram especificados os *tokens*, como por exemplo os tipos de variáveis (*INT*, *FLOAT*, *BOOL*), os nomes dos caracteres (*ABREPARENT*, *FECHAPARENT*, etc.), ou os nomes dos métodos *default* da linguagem (*write()*, *write_all()*, *write_string()*, etc), por exemplo.

Além disso, neste ficheiro também estão presentes os lexemas aceites através de expressões *regex*, para que seja possível o compilador interpretar parágrafos, ou até mesmo aceitar os caracteres definidos para os nomes das variáveis (identificadores).

Através do comando “*flex lex.y*”, é criado um ficheiro em linguagem C, “*lex.yy.c*”, que representa o analisador léxico para a gramática que o nosso grupo especificou em “*lex.l*”.

No ficheiro “*syntax.y*” especificamos a gramática a ser analisada. Começamos por incluir todos os *tokens* aceites gerados pelo analisador léxico, como os tipos e características específicas da linguagem YAIL. A construção da análise léxica foi elaborada por camadas, de modo a facilitar o que o analisador pode aceitar para a linguagem nos diferentes níveis definidos. Na primeira camada é aceite o método *main()*, *structs*{}, *global*, ou comentários, por exemplo, mas nunca poderão ser aceites instruções como condições *if-else*, ou *while*, dado que estes apenas são válidos dentro de funções, como é possível visualizar no exemplo infra.

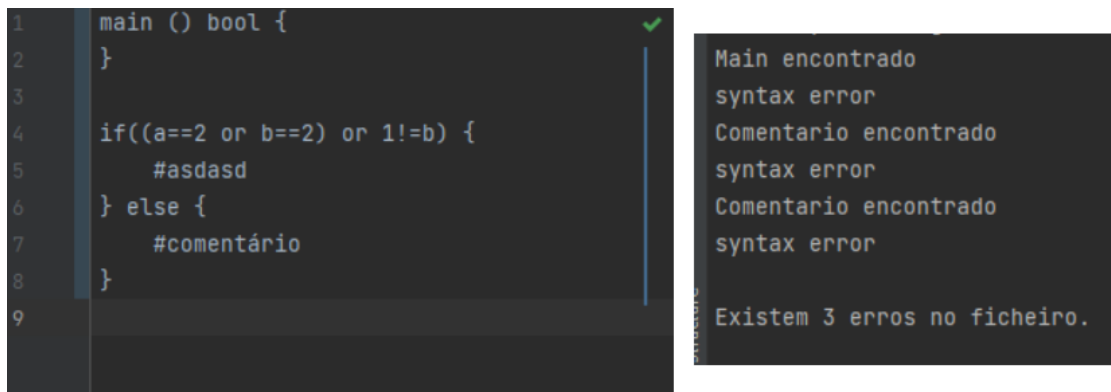


The image shows a code editor with a dark theme. On the left, a code snippet is displayed with line numbers 1 through 7. The code is written in YAIL and includes a function `main () bool {`, an `if` statement with a complex condition, a comment `#asdasd`, an `else` block with another comment `#comentário`, and closing braces. A green checkmark is visible in the top right corner of the code editor. To the right of the code editor, a separate window displays the output of the lexical analysis, listing the tokens found: `Comentario encontrado`, `Comentario encontrado`, `Condicional SENA0 encontrado`, `Condicional SE encontrado`, and `Main encontrado`.

```
1 main () bool {
2     if((a==2 or b==2) or 1!=b) {
3         #asdasd
4     } else {
5         #comentário
6     }
7 }
```

Comentario encontrado
Comentario encontrado
Condicional SENA0 encontrado
Condicional SE encontrado
Main encontrado

Figura 1 - Programa YAIL sem erros

The image shows a code editor on the left and a terminal window on the right. The code editor contains a YAIL program with several syntax errors. The terminal window displays the output of the program, which identifies the errors and counts them.

```
1 main () bool {
2 }
3
4 if((a==2 or b==2) or 1!=b) {
5     #asdasd
6 } else {
7     #comentário
8 }
9
```

```
Main encontrado
syntax error
Comentario encontrado
syntax error
Comentario encontrado
syntax error

Existem 3 erros no ficheiro.
```

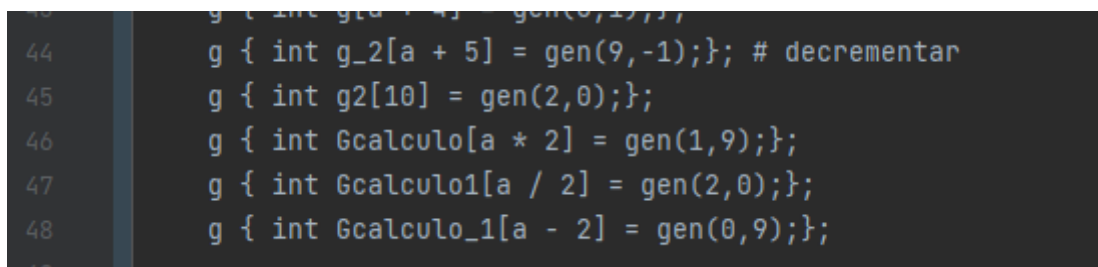
Figura 2 - Programa YAIL com erros

Para melhor compreensão do que se realiza durante a leitura do próprio, adaptaram-se as mensagens de alerta de erros, tais como identificação da linha e o tipo de *token* onde se encontra potencialmente um erro. Indicando também a quantidade de erros detetados.

Adicionalmente, também optamos por incluir *printf*'s na consola com a gramática detetada, de modo a facilitar a visualização da interpretação do analisador.

Para compilar o analisador sintático através do analisador lexical, criamos um ficheiro *bash* “*compileEfolioA.sh*” que agrupa os comandos necessários para gerar os analisadores e fazer a análise léxica a partir um ficheiro *.txt* que contém o código da linguagem YAIL. Para os nossos testes, optamos por criar dois ficheiros, “*YAILComErros.txt*” e “*YAILSemErros.txt*” e fomos efetuando testes à medida que fizemos o desenvolvimento, com 100% de sucesso para os exemplos de “*YAILsemErros.txt*”. Foram realizados testes para todos os exemplos encontrados ao longo do enunciado, para abranger uma maior análise sobre todos os casos possíveis.

Embora esteja quase completo, avalia-se com satisfação a globalidade do resultado obtido e com grande contentamento por se ter ultrapassado o desafio de incluir funções e cálculos mais complexos tais como, potências dentro de raízes pertencentes às funções “*defaults*” de YAIL, tais como:

The image shows a code editor with a YAIL program. The code defines several functions for generating values, including 'g', 'g2', 'Gcalculo', 'Gcalculo1', and 'Gcalculo_1'. The code is syntactically correct.

```
44 g { int g_2[a + 5] = gen(9,-1);}; # decrementar
45 g { int g2[10] = gen(2,0);};
46 g { int Gcalculo[a * 2] = gen(1,9);};
47 g { int Gcalculo1[a / 2] = gen(2,0);};
48 g { int Gcalculo_1[a - 2] = gen(0,9);};
```

Figura 3 - Programa YAIL sem erros para “gen”

```

102 global { float def = pow(a,1);}
103 global { float def = pow(a + 2,1);}
104 global { float def = square_root(8);}
105 global { float def = square_root(pow(2,2) + 1);}
106 global { float def = square_root(pow(2,2+1) + 1);}
107 global { float def = square_root(pow(q.x - p.x, 2) + 1);}
108 global { float def = square_root(pow(q.x-p.x,2) + pow(q.y-p.y,2) + 1);}

```

Figura 4 - Programa YAIL sem erros para “global” com métodos especiais e operações matemáticas

MELHORIAS FUTURAS

Como melhorias futuras pretendemos remover os “warnings” do código implementado, assim como código redundante.

Além disso, pretendemos corrigir a problemática dos operadores aritméticos invertidos (Exemplo: “+=” em vez de “=+”).

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.