



Universidade do Porto

**FEUP** Faculdade de  
Engenharia

---

# Inteligência Artificial

## Compilador

**Relatório Final**

23 de Março de 2003

**Adriano Teixeira (adriano.teixeira@fe.up.pt)**

**Nelson Rodrigues (nelson@fe.up.pt)**

## **Objectivo**

O projecto desenvolvido teve como objectivo o desenvolvimento de um compilador para uma linguagem simples.

A partir de um conjunto de regras de uma linguagem proposta, foi desenvolvido um programa capaz de traduzir segmentos de código dessa, para uma linguagem posteriormente definida, semelhante ao assembly.

## **Motivação**

Apesar da grande aplicação dos programas deste género, o trabalho desenvolvido tem interesse maioritariamente académico. Trata-se de uma linguagem muito simplificada, e os mecanismos disponíveis e por nós utilizados, não serão, provavelmente, os mais utilizados na criação de compiladores para linguagens complexas. No entanto, o trabalho revelou-se um enorme desafio, uma vez que, se por um lado a geração de predicados de aceitação da linguagem não oferece grandes problemas, a tradução da linguagem e a detecção de erros, eleva imensamente o nível de complexidade. Assim, pudemos aperceber-nos das dificuldades da criação destes programas mas, também, da forma como são desenvolvidos e estruturados.

## Descrição

### Funcionalidades

O programa foi projectado para traduzir uma linguagem, cujas directivas sintácticas nos foram fornecidas, para uma outra, com estrutura por nós definida, que se aproximasse tanto quanto possível de uma linguagem máquina. Adicionalmente, tentou abordar-se a detecção de erros, identificando-os tal como à sua localização.

Assim, o programa, face a um ficheiro de entrada com segmentos de código, tenta validá-los e, se tal suceder, tradu-los para linguagem assembly. Se forem detectados erros, tenta detectar a origem desse erro, tal como a sua linha no ficheiro a compilar. No final, uma lista dos erros, é divulgada para o ficheiro: cada erro é acompanhado do código de erro que o gera.

### Estrutura do Programa

O programa consiste num módulo que faz a depuração e a compilação do código simultaneamente, detectando também os eventuais erros sintácticos.

### Esquema de Interpretação de Código

O programa foi desenvolvido com base nas regras de gramáticas nativas do prolog. A utilização das gramáticas possibilitou a manipulação das cadeias de caracteres de acordo com as estruturas de DFAs e NFAs. Optou-se também por fazer uma adaptação da forma normal de parsing, para evitar ciclos infinitos e para facilitar a detecção de erros no código bem como a sua identificação correcta: identificam-se os símbolos esperados e cada uma das cadeias anterior e posterior a esses, passando depois, cada uma dessas cadeias por um processo de divisão semelhante ao descrito anteriormente, até se atingirem os elementos mais simples que não podem ser divididos. Esta estrutura de divisão é, basicamente, uma forma de representar os dados do ficheiro numa árvore virtual em que os nós intermédios são as palavras reservadas (instruções, etc), e os nós folha são os dados indivisíveis inseridos pelo programador: variáveis e números. Aliás, no código desenvolvido, essa estrutura está presente (e claramente visível) no tratamento das expressões matemáticas (explicado na próxima sub-secção). Outra das opções esteve ligada à tentativa de obter o mínimo de predicados a falhar ao longo da execução do programa. Tal opção deveu-se essencialmente à necessidade de detectar e descrever os eventuais erros contidos num ficheiro de código a compilar sem que isso implique parar a validação quando um erro é encontrado. Ainda no sentido de detectar erros, optou-se por dividir as

expressões compostas nas suas várias partes: desta forma, a identificação da localização do erro, é mais precisa. As mensagens de erro são lançadas o mais próximo possível dos nós folha, para que se consiga apurar o melhor possível a causa de cada erro, dando informação mais clara ao programador.

Sobre a leitura do código do ficheiro original, tentou que fosse o mais flexível possível, permitindo mudanças de linha, tabs e espaços entre as várias instruções.

Para armazenar a informação são mantidas três listas:

- uma lista com a informação do ficheiro lido;
- uma lista com o código já traduzido;
- uma lista com os erros detectados.

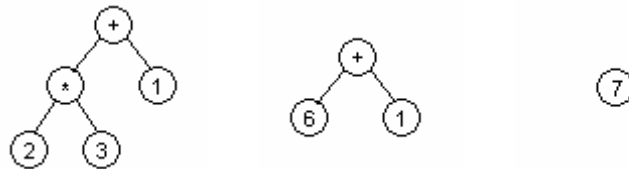
Adicionalmente são mantidas duas variáveis que acompanham a sequência de execução do programa, sendo decisivas para a compilação correcta:

- uma variável que contem valor 0 se ainda não foi encontrado nenhum erro ou 1, caso já tenha sido detectado pelo menos um erro;
- uma variável que contém um número que é incrementado em cada novo label adicionado ao código Asm, de forma a nunca surgirem repetições nos nomes (o que geraria ambiguidades no código traduzido).

### Implementação

Ao longo do desenvolvimento do programa, surgiram problemas que houve necessidade de ultrapassar:

- Transformação correcta de expressões matemáticas. Para transformar uma expressão matemática em código Asm que possa dar origem ao mesmo valor resultante, é necessário ter em conta a ordem de precedências das operações. Para tal, usou-se uma estrutura em árvore, em que os nós mais acima são as operações com menos prioridade na expressão, os mais abaixo os que têm maior prioridade e as folhas os operandos. Exemplo: à expressão  $2*3+1$  terá de



corresponder a árvore:

Desta forma, e sendo a árvore tratada de baixo para cima, o resultado será  $7 = 2*3+1$ .

- Garantir que uma a sequência de análise não entre em ciclo infinito, mesmo que a linguagem a implementar seja susceptível desse tipo de erros. Com o uso de gramáticas, determinadas construções gramaticais podem causar entradas

sucessivas no mesmo predicado, sem que ocorra “queda” de caracteres. Como exemplo, pode verificar-se a seguinte definição:

```
<declaracoes> ::= <declaraco> | <declaracao> ; <declaracoes>
```

De acordo com esta estrutura, uma passagem directa para prolog usando parsers, seria algo como:

```
declaracoes --> declaracao, “;”, declaracoes.  
declaracoes --> declaracao.
```

Uma expressão deste género, facilmente poderá dar origem a entradas em ciclo infinito, o que também não pode ser resolvido com ! (cut), uma vez que é também necessário captar os erros sintácticos, o que exige que os predicados sucedam sempre.

Para resolver esta situação, e recorrendo de novo ao exemplo anterior, optamos por captar o código antes e depois do “;”, fazendo chamadas aos predicados respectivos depois. Este procedimento tem custos a nível de performance quando analisamos soluções garantidamente correctas, mas em caso de existirem erros sintácticos ou haver sequências de código que produzam ciclos infinitos, os ganhos são imensos.

Para garantir a capacidade de captar os erros correctamente e com o mínimo de particularização, foi necessário alterar, mais uma vez, a implementação directa do uso das gramáticas do prolog. Tendo como exemplo a expressão: IF expTeste THEN expThen ELSE expElse, que seria directamente implementado como declaracao --> “IF”, expTeste, “ THEN “, expThen, “ ELSE “, expElse., um erro em qualquer das suas partes, ou mesmo a não existência do próprio If, originaria fail no predicado que testasse a referida expressão. Por um lado, não se pode optar por escrever um predicado alternativo (com o mesmo nome), que indicasse ter havido uma falha, já que nada garante que não tenha falhado logo no “IF” (pelo que a expressão analisada não seria um If, mas qualquer outra.

Ignorando essas falhas, perdemos a capacidade de captar erros. Assim, optamos por uma outra solução, que consiste em dividir o predicado acima, em vários predicados, que se vão chamando sequencialmente e nos quais já se pode verificar os erros. Tal pode ser visto entre as linhas 152 e 252 do código que segue em anexo.

### **Análise da complexidade dos algoritmos que usou**

Neste programa, baseamo-nos nas potencialidades das gramáticas do prolog, sendo o nível de backtracking por ele efectuado na pesquisa das combinações correctas, decisivo no nível de complexidade global. Não utilizamos algoritmos específicos já modelados e testados; apenas podemos estimar que os que usamos estejam algures entre o  $O(N \cdot \log N)$  e o  $O(N^2)$ .

### **Ambiente de desenvolvimento**

O trabalho foi desenvolvido em diversas máquinas, desde Pentium 800Mhz / 128MB até P4-m 2GHz / 512MB. Os sistemas operativos usados foram Win2000 e WinXP, sendo o ambiente de programação usado o Sicstus (pois o programa foi totalmente escrito em Prolog), devido à experiência que temos com este software já desde o semestre passado. Todo o código usado foi produzido por nós, não tendo sido usados pacotes ou predicados de outros.

### **Avaliação do programa**

A análise da potencialidade de um compilador, deverá estar sempre associada à linguagem que este processa. Assim sendo, consideramos que o mais importante seria a correcta compilação do código submetido. Seria também importante a geração e retorno de mensagens de erro o mais pormenorizadas possível. Seria também importante, num nível inferior, a rapidez da compilação.

Em termos de correcção do código gerado, pensamos, apoiados nos testes feitos, que a criação é correcta. Quanto a mensagens de erro, pensamos que a pormenorização está num bom nível, dada a estrutura que usamos. O facto de mostrar todos os erros existentes num ficheiro apenas com uma tentativa de compilação, será algo positivo, que sem dúvida apreciaríamos numa eventual análise a um qualquer compilador. O tempo de compilação foi um pouco penalizado pelas decisões tomadas, mas julgamos que os tempos são, ainda assim, aceitáveis.

## **Resultados Experimentais**

Ao longo do desenvolvimento do programa, foram feitos diversos testes, com vários inputs. Dado o tipo de programa e o seu propósito, a variedade de testes é fundamental para que a probabilidade de persistirem erros ‘ocultos’, seja minimizada. Devido ao tamanho dos inputs e recorrentes outputs, vamos apenas mostrar 2 exemplos, um de código gerado com sucesso, outro de um erro no ficheiro a compilar.

Caso 1: Código do ficheiro a compilar, correcto:

Caso 2: Código do ficheiro a compilar, com dois erros:

## **Conclusão**

Ao longo do desenvolvimento desta aplicação, sentiu-se um progressivo interesse pelo próximo passo a dar no desenvolvimento de cada funcionalidade nova. Sentimos ter sido um projecto com grandes exigências em termos de planificação.

Foi, contudo, gratificante o facto da aplicação estar a transformar código tal como esperado.

## **Melhoramentos**

Apesar de o resultado estar de acordo com o que esperávamos, poderiam ser feitas algumas alterações e adições que melhorariam a o programa. Por um lado, mensagens de erro ainda mais pormenorizadas, recorrendo a uma análise do segmento em que se encontra a falha (após concluir que realmente existe uma falha), procurando erros comuns, como inserção de um caracter acidental no meio de uma palavra reservadas, operadores esperados, operando em falta.. Por outro lado, algo que poderia de alguma forma melhorar o desempenho: subdividir o texto inicial em tokens, atribuindo possivelmente códigos às expressões e palavras reservadas.

Compilador

## **Bibliografia**

Acetatos do Professor Eugénio Oliveira.

## **Manual do utilizador**

Para compilar um ficheiro, digitar 'compilar(fxIn, fxOut)', em que fxIn é o ficheiro a ser compilado (com respectiva path) e fxOut é o ficheiro onde o código traduzido ou os erros de compilação serão escritos (também com respectiva path).

### Exemplo de uma execução

Devido ao caracter da nossa aplicação, o exemplo de execução, não será uma sequência de ecrãs, mas o output resultante de um input dado, de acordo com a linguagem.

Input: Programa que lê um número e imprime a tabuada a ele relativo (se o valor for maior que 10, imprime -1).

```
READ Valor;

IF Valor < (10) THEN
(
    A := (1);

    WHILE A <= (10)
    DO
    (
        Resultado := A * Valor;
        A := A + (1);
        WRITE Resultado
    )
)
ELSE
(
    A := (0) - (1);
    WRITE A
)
)
```

Output: Código em Asm gerado pela aplicação.

```
read Valor
mov Acc, Valor
push Acc
mov Acc, #10
mov R0, Acc
pop Acc
add R1, R0, #1
subb R0, R1, Acc
jnc IfFalha1
mov Acc, #1
mov A, Acc
label WhileFalha2
mov Acc, A
push Acc
mov Acc, #10
mov R0, Acc
pop Acc
mov R1, R0
subb R0, R1, Acc
jnc WhileCerto2
mov Acc, A
push Acc
mov Acc, Valor
pop R0
mov R1, Acc
```

## Compilador

```
    mul Acc, R0, R1
    mov Resultado, Acc
    mov Acc, A
    push Acc
    mov Acc, #1
    pop R0
    mov R1, Acc
    add Acc, R0, R1
    mov A, Acc
    write Resultado
    jump WhileFalha2
label WhileCerto2
    jump IfCerto1
label IfFalha1
    mov Acc, #0
    push Acc
    mov Acc, #1
    pop R0
    mov R1, Acc
    sub Acc, R0, R1
    mov A, Acc
    write A
label IfCerto1
```

## Listagem do código

```

1
2     dic(":").
3     dic("=").
4     dic(">").
5     dic("<").
6     dic("(").
7     dic(")").
8     dic("/").
9     dic("*").
10    dic("+").
11    dic("-").
12    dic(";").
13    dic(" ").
14    dic("%").
15    dic("!").
16    dic("\n").
17    dic("\t").
18    dic(":").
19
20    dicPR("IF").
21    dicPR("THEN").
22    dicPR("ELSE").
23    dicPR("WHILE").
24    dicPR("DO").
25    dicPR("WRITE").
26    dicPR("READ").
27
28    op_comp(Falha, Final) -->
29        "<",
30        {
31            juntar("\tjnc ", Falha, Interml),
32            juntar(["\tadd R1, R0, #1", "\tsubb R0, R1,
Acc"], [Interml], Final)
33        }.
34    op_comp(Falha, Final) -->
35        "<=", %Acc <= R0
36        {
37            juntar("\tjnc ", Falha, Interml),
38            juntar(["\tmov R1, R0", "\tsubb R0, R1, Acc"],
[Interml], Final)
39        }.
40    op_comp(Falha, Final) -->
41        ">", %Acc > R0
42        {
43            juntar("\tjnc ", Falha, Interml),
44            juntar(["\tadd R1, R0, #1", "\tsubb R0, Acc,
R1"], [Interml], Final)
45        }.
46    op_comp(Falha, Final) -->
47        ">=", %Acc >= R0
48        {
49            juntar("\tjnc ", Falha, Interml),
50            juntar(["\tmov R1, Acc", "\tsubb Acc, R1, R0"],
[Interml], Final)
51        }.
52    op_comp(Falha, [Final]) -->
53        "=",
54        {
55            juntar("\tcjne Acc, R0, ", Falha, Final)
56        }.

```

## Compilador

```
57 op_comp(Falha, Final) -->
58     "<>" ,
59     {
60         juntar(["\tmov R1, Acc"], ["\tsub Acc, R1,
R0"], Interml),
61         juntar("\tjz ", Falha, Interml2),
62         juntar(Interml, [Interml2], Final)
63     }.
64
65 op2(["\tpop R0", "\tmov R1, Acc", "\tmul Acc, R0, R1"]) --
>
66     "*".
67 op2(["\tpop R0", "\tmov R1, Acc", "\tdiv Acc, R0, R1"]) --
>
68     "/".
69
70 op1(["\tpop R0", "\tmov R1, Acc", "\tadd Acc, R0, R1"]) --
>
71     "+".
72 op1(["\tpop R0", "\tmov R1, Acc", "\tsub Acc, R0, R1"]) --
>
73     "-".
74
75
76 compilar(FxEntrada, FxSaida):-
77     open(FxEntrada,read,StreamEntrada),
78     get0(StreamEntrada,Char),
79     leFicheiro(Char,Chars,StreamEntrada),
80     compilarC(Exp, ExpE, Erro, Chars, []), !,
81     (Erro == 0, escreveFx(FxSaida, Exp);
82     escreveFx(FxSaida, ExpE)),
83     close(StreamEntrada).
84
85 % Codigo dos Caracteres:
86 %     Return 10
87 %     Espaco 32
88 %     TAB 9
89 %     EOF -1
90
91
92 leFicheiro(-1, [], _) :- !. % Fim do Ficheiro
93 leFicheiro(end_of_file, [], _) :- !.
94 leFicheiro(Char, [Char|Chars], StreamEntrada) :-
95     get0(StreamEntrada, Prox),
96     leFicheiro(Prox, Chars, StreamEntrada).
97
98
99 compilarC(Exp, ExpE, Erro) -->
100     declaracoes(Exp, ExpE, 1, _, 0, Erro, 0, _).
101
102 declaracoes(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdF) -->
103     parteString(X1, Y2, ";"),
104     {
105
106         (
107             parentesis(X1,Z1), Z1==0
108             ;
109             parentesis(Y2,Z2), Z2==0
110         ),
111     nome(_, NlsI, X, _, NlsF, 0, X1, []),
```

## Compilador

```

112         nome(_, NlsI2, Y, _, NlsF2, 0, Y2, []),
113         X\=[], Y\=[],
114         LinhaIT1 is LinhaI + NlsI,
115         declaracao(Decl, ExpE1, LinhaIT1, LinhaFT1,
ErroI, ErroF1, LabelIdI, LabelIdFT, X, []),
116         LinhaIT2 is LinhaFT1 + NlsF + NlsI2,
117         declaracoes(Decls, ExpE2, LinhaIT2, LinhaFT2,
ErroF1, ErroF, LabelIdFT, LabelIdF, Y, []),
118         LinhaF is LinhaFT2 + NlsF2,
119         (
120             ErroF = 0, juntar(Decl, Decls, Exp), ExpE
= []
121             ;
122             juntar(ExpE1, ExpE2, ExpE), Exp = []
123         )
124     }.
125     declaracoes(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdF) -->
126     nome(_, NlsI, X, _, NlsF, 0),
127     {
128         LinhaIT is LinhaI + NlsI,
129         declaracao(Exp, ExpE, LinhaIT, LinhaFT, ErroI,
ErroF, LabelIdI, LabelIdF, X, []),
130         LinhaF is LinhaFT + NlsF
131     }.
132
133     declaracao(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdI) -->
134     nome(fim, _, NlsF, X, 1),
135     ":",
136     nome(ini, _, NlsI, Y, 0),
137     {
138         NlsF=0, NlsI=0,
139         verifica(X), verificaNaoPR(X),
140         X\=[], Y\=[],
141         expressao(Z1, ExpE, LinhaI, LinhaF, ErroI,
ErroF, Y, []),
142         (
143             ErroF = 0,
144             juntar("\tmov ", X, EXP1),
145             juntar(EXP1, " , Acc", EXP2),
146             juntar(Z1, [EXP2], Exp)
147             ;
148             Exp = []
149         )
150     }.
151
152     declaracao(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdF) -->
153     "IF", nome(ini, EspI, NlsI, X, 0),
154     {
155         LabelIdIT is LabelIdI + 1,
156         EspI + NlsI > 0,
157         LinhaIT is LinhaI + NlsI,
158         if1(Exp, ExpE, LinhaIT, LinhaF, ErroI, ErroF,
LabelIdIT, LabelIdF, X,[])
159     }.
160     if1(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF, LabelIdI,
LabelIdF) -->
161     nome(fim, EspF, NlsF, X, 1),
162     {

```

## Compilador

```
163             EspF + NlsF > 0
164         },
165         "THEN",
166         nome(ini, EspI, NlsI, Y, 0),
167         {
168             cadeiaValida(X, []),
169
170             EspI + NlsI > 0,
171             juntar("If", "Certo", IfTrueT),
172             number_codes(LabelIdI, LabelIdIa),
173             juntar(IfTrueT, LabelIdIa, IfTrue),
174             juntar("label ", IfTrue, LblTrue),
175             LinhaIT is LinhaI,
176             if_teste(Teste, ExpE1, LinhaIT, LinhaFT1,
177 ErroI, ErroF1, LabelIdI, LabelIdFT, X, []),
178             LinhaIT2 is LinhaFT1 + NlsI + NlsF,
179             if2(Exp2, ExpE2, LinhaIT2, LinhaF, ErroF1,
180 ErroF, LabelIdFT, LabelIdF, Y, []),
181
182             (
183                 ErroF = 0,
184                 juntar(Teste, Exp2, Exp3),
185                 juntar(Exp3, [LblTrue], Exp),
186                 ExpE = []
187             ;
188             juntar(ExpE1, ExpE2, ExpE), Exp = []
189         )
190     }.
191 if1(_, ExpE, LinhaI, LinhaF, _, 1, LabelIdI, LabelIdI) -->
192     contaLinhas(X, Nls),
193     {
194         trataErros(X, Nls, "Erro: If sem Then!", ExpE,
195 LinhaI, LinhaF)
196     }.
197
198 if_teste(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF,
199 LabelIdI, LabelIdI) -->
200     {
201         juntar("If", "Falha", IfFalseT),
202         number_codes(LabelIdI, LabelIdIa),
203         juntar(IfFalseT, LabelIdIa, IfFalse)
204     },
205     teste(IfFalse, Exp, ExpE, LinhaI, LinhaF, ErroI,
206 ErroF).
207
208 if2(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF, LabelIdI,
209 LabelIdF) -->
210     nome(fim, EspF, NlsF, X, 1),
211     {
212         EspF + NlsF > 0
213     },
214     "ELSE",
215     nome(ini, EspI, NlsI, Y, 0),
216     {
217         cadeiaValida(X, []),
218         NlsI + EspI > 0,
219         juntar("If", "Falha", IfFalseT),
220         number_codes(LabelIdI, LabelIdIa),
221         juntar(IfFalseT, LabelIdIa, IfFalse),
222         juntar("label ", IfFalse, LblFalse),
223         juntar("If", "Certo", IfTrueT),
224         juntar(IfTrueT, LabelIdIa, IfTrue),
```

## Compilador

```
218         LinhaIT is LinhaI,
219         if_then(Exp1, ExpE1, LinhaIT, LinhaFT1, ErroI,
ErroF1, LabelIdI, LabelIdFT, X, []),
220         LinhaIT2 is LinhaFT1 + NlsI + NlsF,
221         if_else(Exp2, ExpE2, LinhaIT2, LinhaF, ErroF1,
ErroF, LabelIdFT, LabelIdF, Y, []),
222         (
223             ErroF = 0,
224             juntar("\tjump ", IfTrue, ExpI),
225             juntar(Exp1, [ExpI, LblFalse], Interm),
226             juntar(Interm, Exp2, Exp), ExpE = []
227             ;
228             juntar(ExpE1, ExpE2, ExpE), Exp = []
229         )
230     }.
231 if2(_, ExpE, LinhaI, LinhaF, _, 1, LabelIdI, LabelIdI) -->
232     contaLinhas(X, Nls),
233     {
234         trataErros(X, Nls, "Erro: If sem Else!", ExpE,
LinhaI, LinhaF)
235     }.
236
237 if_then(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF, LabelIdI,
LabelIdF) -->
238     declaracao(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdF).
239
240 if_else(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF, LabelIdI,
LabelIdF) -->
241     declaracao(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdF).
242
243 declaracao(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdF) -->
244     "WHILE", nome(ini, EspI, NlsI, X, 0),
245     {
246         LabelIdIT is LabelIdI + 1,
247         EspI + NlsI > 0,
248         LinhaIT is LinhaI + NlsI,
249         while1(Exp, ExpE, LinhaIT, LinhaF, ErroI,
ErroF, LabelIdIT, LabelIdF, X, [])
250     }.
251 while1(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF, LabelIdI,
LabelIdF) -->
252     nome(fim, EspF, NlsF, X, 1),
253     {
254         EspF + NlsF > 0
255     },
256     "DO",
257     nome(ini, EspI, NlsI, Y, 0),
258     {
259         cadeiaValida(X, []),
260         EspI + NlsI > 0,
261         juntar("While", "Certo", WhileTrueT),
262         number_codes(LabelIdI, LabelIdIa),
263         juntar(WhileTrueT, LabelIdIa, WhileTrue),
264         juntar("label ", WhileTrue, LblTrue),
265         juntar("While", "Falha", WhileFalseT),
266         juntar(WhileFalseT, LabelIdIa, WhileFalse),
267         juntar("label ", WhileFalse, LblFalse),
268         juntar("\tjump ", WhileFalse, Exp1),
```

## Compilador

```
269         LinhaIT1 is LinhaI,
270         while_teste(Teste, ExpE1, LinhaIT1, LinhaFT1,
ErroI, ErroF1, LabelIdI, LabelIdFT, X, []),
271         LinhaIT2 is LinhaFT1 + NlsI + NlsF,
272         while2(Decl1, ExpE2, LinhaIT2, LinhaF, ErroF1,
ErroF, LabelIdFT, LabelIdF, Y, []),
273         (
274             ErroF = 0,
275             juntar([LblFalse], Teste, Interm1),
276             juntar(Interm1, Decl1, Interm2),
277             juntar(Interm2, [Exp1, LblTrue], Exp),
ExpE = []
278             ;
279             juntar(ExpE1, ExpE2, ExpE), Exp = []
280         )
281     }.
282 while1(_, ExpE, LinhaI, LinhaF, _, 1, LabelIdI, LabelIdI)
-->
283     contaLinhas(X, Nls),
284     {
285         trataErros(X, Nls, "Erro: While sem Do!", ExpE,
LinhaI, LinhaF)
286     }.
287
288 while_teste(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdI) -->
289     {
290         juntar("While", "Certo", WhileTrueT),
291         number_codes(LabelIdI, LabelIdIa),
292         juntar(WhileTrueT, LabelIdIa, WhileTrue)
293     },
294     teste(WhileTrue, Exp, ExpE, LinhaI, LinhaF, ErroI,
ErroF).
295
296 while2(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF, LabelIdI,
LabelIdF) -->
297     declaracao(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdF).
298
299 declaracao([Exp], _, LinhaI, LinhaI, ErroI, ErroI,
LabelIdI, LabelIdI) -->
300     "READ", nome(ini, EspI, _, X, 0),
301     {
302         EspI>0,
303         X\=[],
304         verifica(X),
305         not(inteiro(_, X, [])),
306         juntar("\tread ", X, Exp)
307     }.
308 declaracao([Exp], _, LinhaI, LinhaI, ErroI, ErroI,
LabelIdI, LabelIdI) -->
309     "WRITE", nome(ini, EspI, _, X, 0),
310     {
311         EspI>0,
312         X\=[],
313         verifica(X),
314         not(inteiro(_, X, [])),
315         juntar("\twrite ", X, Exp)
316     }.
317 declaracao(Decls, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdF) -->
```

## Compilador

```
318
319     parente(Decls, ExpE, LinhaI, LinhaF, ErroI, ErroF,
LabelIdI, LabelIdF).
320
321 parente(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF, LabelIdI,
LabelIdF) -->
322
323     "(", nome(ini, _, NlsI, X, 0),
324     {
325         LinhaIT is LinhaI + NlsI,
326         parent(Exp, ExpE, LinhaIT, LinhaF, ErroI,
ErroF, LabelIdI, LabelIdF, X, [])
327     }.
328
329 parent(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF, LabelIdI,
LabelIdF) -->
330     nome(fim, _, NlsI, X, 1), ")",
331     {
332         parentesis(X,0),
333         declaracoes(Exp, ExpE, LinhaI, LinhaFT, ErroI,
ErroF, LabelIdI, LabelIdF, X, []),
334         LinhaF is LinhaFT + NlsI
335     }.
336 parent(_, ExpE, LinhaI, LinhaF, _, 1, LabelIdI, LabelIdI)
-->
337     contaLinhas(X, Nls),
338     {
339         trataErros(X, Nls, "Erro nos parentesis!",
ExpE, LinhaI, LinhaF)
340     }.
341
342 declaracao(_, ExpE, LinhaI, LinhaF, _, 1, LabelIdI,
LabelIdI) -->
343     contaLinhas(X, Nls),
344     {
345         trataErros(X, Nls, "Erro!", ExpE, LinhaI,
LinhaF)
346     }.
347
348 teste(Cab, Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF) -->
349     nome(fim, _, NlsF, X, 1), op_comp(Cab, Z3), nome(ini,
_, NlsI, Y, 0),
350     {
351         X\=[], Y\=[],
352
353         LinhaIT is LinhaI + NlsF,
354         expressao(Z1, ExpE1, LinhaIT, LinhaFT1, ErroI,
ErroF1, X, []),
355         LinhaIT2 is LinhaFT1 + NlsI,
356         expressao(Z2, ExpE2, LinhaIT2, LinhaF, ErroF1,
ErroF, Y, []),
357         (
358             ErroF = 0,
359             juntar(Z1, ["\tpush Acc"], Z4),
360             juntar(Z4, Z2, Z5),
361             juntar(Z5, ["\tmov R0, Acc", "\tpop
Acc"], Z6),
362             juntar(Z6, Z3, Exp), ExpE = []
363         );
364         Exp= [],
365         juntar(ExpE1, ExpE2, ExpE)
```

## Compilador

```

366         )
367     }.
368
369 teste(_, _, ExpE, LinhaI, LinhaF, _, 1) -->
370     contaLinhas(X, Nls),
371     {
372         trataErros(X, Nls, "Erro na expressao de
teste!", ExpE, LinhaI, LinhaF)
373     }.
374
375 expressao(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF) -->
376     nome(fim, _, NlsF, X, 1), op2(Z), nome(ini, _, NlsI,
Y, 0),
377     {
378         X\=[], Y\=[],
379         LinhaIT is LinhaI + NlsF,
380         expressao(A, ExpE1, LinhaIT, LinhaFT1, ErroI,
ErroF1, X, []),
381         LinhaIT2 is LinhaFT1 + NlsI,
382         expressao1(B, ExpE2, LinhaIT2, LinhaF, ErroF1,
ErroF, Y, []),
383         (
384             ErroF=0,
385             juntar(A, ["\tpush Acc"], A1),
386             juntar(A1, B, B1),
387             juntar(B1, Z, Exp), ExpE = []
388             ;
389             juntar(ExpE1, ExpE2, ExpE),
390             Exp=[]
391         )
392     }.
393 expressao(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF) -->
394     nome(X),
395     {
396         !,
397         X\=[],
398         expressao1(Exp, ExpE, LinhaI, LinhaF, ErroI,
ErroF, X, [])
399     }.
400
401 expressao1(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF) -->
402     nome(fim, _, NlsF, X, 1), op1(Z), nome(ini, _, NlsI,
Y, 0), % nome(Y),
403     {
404         X\=[], Y\=[],
405         LinhaIT is LinhaI + NlsF,
406         expressao1(A, ExpE1, LinhaIT, LinhaFT1, ErroI,
ErroF1, X, []),
407         LinhaIT2 is LinhaFT1 + NlsI,
408         expressao0(B, ExpE2, LinhaIT2, LinhaF, ErroF1,
ErroF, Y, []),
409         (
410             ErroF = 0,
411             juntar(A, ["\tpush Acc"], A1),
412             juntar(A1, B, B1),
413             juntar(B1, Z, Exp), ExpE = []
414             ;
415             juntar(ExpE1, ExpE2, ExpE), Exp=[]
416         )
417     }.
418 expressao1(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF) -->

```

## Compilador

```

419         nome(X),
420         {
421             !, X\=[],
422             expressao0(Exp, ExpE, LinhaI, LinhaF, ErroI,
ErroF, X, [])
423         }.
424
425     expressao0(Exp, ExpE, LinhaI, LinhaF, ErroI, ErroF) -->
426         "(", nome(ini, _, NlsI, Y, 0),
427         {
428             LinhaIT is LinhaI + NlsI,
429             parente2(Exp, ExpE, LinhaIT, LinhaF, ErroI,
ErroF, Y, [])
430         }.
431
432     parente2([Exp], ExpE, LinhaI, LinhaF, ErroI, ErroF) -->
433         nome(fim, _, NlsF, X, 1), ")",
434         {
435             X\=[],
436             parentesis(X, Z), Z==0,
437             (
438                 LinhaF is LinhaI + NlsF,
439                 inteiro(XT, X, []),
440                 juntar("\tmov Acc, #", XT, Exp),
441                 ErroF is ErroI,
442                 ExpE = []
443             );
444             (
445                 expressao(Exp, ExpE, LinhaI, LinhaFT,
ErroI, ErroF, X, []),
446                 LinhaF is LinhaFT + NlsF
447             )
448         }.
449     parente2(_, ExpE, LinhaI, LinhaF, _, 1) -->
450         contaLinhas(X, Nls),
451         {
452             trataErros(X, Nls, "Erro nos parentesis!",
ErroF, LinhaI, LinhaF)
453         }.
454
455
456     expressao0([Exp], _, LinhaI, LinhaI, ErroI, ErroI) -->
457         nome(X),
458         {
459             verifica(X), verificaNaoPR(X), X\=[],
460             not(inteiro(_, X, [])),
461             juntar("\tmov Acc, ", X, Exp)
462         }.
463
464     nome([X|Y]) -->
465         [X], nome(Y).
466     nome([]) -->
467         [].
468
469     inteiro([X|Y]) -->
470         [X], inteiro(Y), {X>="0",X<="9"}.
471     inteiro([]) -->
472         [].
473
474     verifica([]).
475     verifica([X|Y]):-

```

## Compilador

```
476         not(dic([X])), verifica(Y).
477
478     verifica([], _).
479     verifica([X|Y], Car):-
480         X\=Car, verifica(Y).
481
482     imprime([]).
483     imprime([X|Xs]):-
484         atom_chars(XF,X),
485         write(XF), nl,
486         imprime(Xs).
487
488     juntar([], L, L).
489     juntar([H|T], L, [H|NT]):-
490         juntar(T, L, NT).
491
492     espacos(Ret) --> " ", espacos(X), {Ret is X + 1}.
493     espacos(Ret) --> "", {Ret is 0}.
494
495     nls(Ret) --> "\n", nls(X), {Ret is X + 1}.
496     nls(Ret) --> " ", nls(Ret).
497     nls(Ret) --> "", {Ret is 0}.
498
499     nls(Ret, Ret2) --> "\n", nls(X, Ret2), {Ret is X + 1}.
500     nls(Ret, Ret2) --> " ", nls(Ret, Y), {Ret2 is Y + 1}.
501     nls(Ret, Ret2) --> "\t", nls(Ret, Y), {Ret2 is Y + 1}.
502     nls(0, 0) --> "".
503
504     espNlsTabs(Sum, Sum) -->
505         nls(Sum, 0).
506
507     espNlsTabs(Sum, 0) -->
508         nls(0, Sum).
509     espNlsTabs(Sum, X) -->
510         nls(X, Y), {Sum is X + Y}.
511
512     esp(X, Y) -->
513         espNlsTabs(X, Y).
514
515     not(X):-
516         X, !, fail.
517     not(_).
518
519
520     parentesis([40|Y], Z):- !,
521         parentesis(Y, Z1),
522         Z1<0,
523         Z is Z1 + 1.
524     parentesis([41|Y], Z):- !,
525         parentesis(Y, Z1),
526         Z1<0,
527         Z is Z1 - 1.
528     parentesis([_|Y], Z1):- !,
529         parentesis(Y, Z1).
530     parentesis([], 0).
531
532     %Le uma das partes da cadeia, eliminando e contando as
533     %ocorrencias d espacos, \n e \t no inicio e no fim
534     nome(0,0,[],0,0,_) --> [].
535     nome(EspI, NlsI, Letras, EspF, NlsF, A) -->
536         delt(A, F, Esp, Nl, _), {F=0},
```

## Compilador

```
536     nome(EspIT, NlsIT, Letras, EspFT, NlsFT, F),
537     {
538         EspI is EspIT + Esp,
539         NlsI is NlsIT + Nl,
540         EspF is EspFT,
541         NlsF is NlsFT
542     }.
543
544 nome(EspI, NlsI, Letras, EspF, NlsF, A) -->
545     delt(A, F, Esp, Nl, _), {F=2},
546     nome(EspIT, NlsIT, Letras, EspFT, NlsFT, F),
547     {
548         EspI is EspIT,
549         NlsI is NlsIT,
550         EspF is EspFT + Esp,
551         NlsF is NlsFT + Nl
552     }.
553
554 nome(EspI, NlsI, [LetrasI|LetrasF], EspF, NlsF, A) -->
555     delt(A, F, _, _, LetrasI), {F=1},
556     nome(EspIT, NlsIT, LetrasF, EspFT, NlsFT, F),
557     {
558         EspI is EspIT,
559         NlsI is NlsIT,
560         EspF is EspFT,
561         NlsF is NlsFT
562     }.
563
564 %Le uma das partes da cadeia, eliminando e contando as
565 %ocorrencias d espacos, \n e \t no inicio
566 nome(ini,0,0,[],_) --> [].
567 nome(ini,EspI, NlsI, Letras, 0) -->
568     delt(0, F, Esp, Nl, _), {F=0}, {!},
569     nome(ini,EspIT, NlsIT, Letras, F),
570     {
571         EspI is EspIT + Esp,
572         NlsI is NlsIT + Nl
573     }.
574
575 nome(ini,EspI, NlsI, [LetrasI|LetrasF], _) -->
576     {!},
577     [LetrasI],
578     %delt(A, F, Esp, Nl, LetrasI), {F=1},
579     nome(ini,EspIT, NlsIT, LetrasF, 1),
580     {
581         EspI is EspIT,
582         NlsI is NlsIT
583     }.
584
585 %Le uma das partes da cadeia, eliminando e contando as
586 %ocorrencias d espacos, \n e \t no fim
587 nome(fim, 0,0,[],_) --> [].
588 nome(fim, EspF, NlsF, Letras, A) -->
589     delt(A, F, Esp, Nl, _), {F=2},
590     nome(fim, EspFT, NlsFT, Letras, F),
591     {
592         EspF is EspFT + Esp,
593         NlsF is NlsFT + Nl
594     }.
595
596 nome(fim, EspF, NlsF, [LetrasI|LetrasF], A) -->
```

## Compilador

```
595     delt(A, F, _, _, LetrasI), {F=1},
596     nome(fim, EspFT, NlsFT, LetrasF, F),
597     {
598         EspF is EspFT,
599         NlsF is NlsFT
600     }.
601
602     delt(0, 0, 1, 0, _) --> " ".
603     delt(0, 0, 1, 0, _) --> "\t".
604     delt(0, 0, 0, 1, _) --> "\n".
605     delt(0, 1, _, _, X) --> [X], {[X]\=" " , [X]\="\t",
[X]\="\n"}.
606     delt(1, 2, 1, 0, _) --> " ".
607     delt(1, 2, 1, 0, _) --> "\t".
608     delt(1, 2, 0, 1, _) --> "\n".
609     delt(1, 1, _, _, X) --> [X].
610     delt(2, 2, 1, 0, _) --> " ".
611     delt(2, 2, 1, 0, _) --> "\t".
612     delt(2, 2, 0, 1, _) --> "\n".
613
614
615     contaLinhas([X|Y], LF) -->
616     [X], contaLinhas(Y, LI), {[X]=="\n", LF is LI + 1};
LF is LI}.
617     contaLinhas([], 0) -->
618     [].
619
620
621     escreveFx(NomeFx, Lista):-
622     tell(NomeFx),
623     escreveLinhas(Lista),
624     told.
625
626     escreveLinhas([]).
627     escreveLinhas([LinhaI|LinhaS]):-
628     escreveCaract(LinhaI),
629     put("\n"),
630     escreveLinhas(LinhaS).
631
632     escreveCaract([]).
633     escreveCaract([CaractI|CaractS]):-
634     put(CaractI),
635     escreveCaract(CaractS).
636
637     trataErros(Cadeia, Nls, ExpreErro, [ExpE], LinhaI,
LinhaF):-
638     number_codes(LinhaI, LinhaIa),
639     (
640         Nls > 0,
641         LinhaF is LinhaI + Nls,
642         number_codes(LinhaF, LinhaFa),
643         juntar(ExpreErro, "\n\tLinhas ", Interml),
644         juntar(Interml, LinhaIa, Interml2),
645         juntar(" a ", LinhaFa, Interml3),
646         juntar(Interml2, Interml3, ExpEt)
647     );
648     LinhaF is LinhaI,
649     juntar(ExpreErro, "\n\tLinha ", Interml1),
650     juntar(Interml1, LinhaIa, ExpEt)
651     ),
652     juntar("\n\tCodigo: ", Cadeia, Interml),
```

## Compilador

```
653         juntar(ExpEt, InterM, ExpE).
654
655     cadeiaValida --> cadeiaValida(X, P), {X==1, P==0}.
656     cadeiaValida(Ret, P) --> "IF", cadeiaValida(X, P), {Ret is
X // 4, !}.
657     cadeiaValida(Ret, P) --> "THEN", cadeiaValida(X, P), {Ret
is X * 2, !}.
658     cadeiaValida(Ret, P) --> "ELSE", cadeiaValida(X, P), {Ret
is X * 2, !}.
659     cadeiaValida(Ret, P) --> "WHILE", cadeiaValida(X, P), {Ret
is X // 3, !}.
660     cadeiaValida(Ret, P) --> "DO", cadeiaValida(X, P), {Ret is
X * 3, !}.
661     cadeiaValida(Ret, P) --> "(", cadeiaValida(X, P1), {Ret is
X // 5, P is P1 + 1, !}.
662     cadeiaValida(Ret, P) --> ")", cadeiaValida(X, P1), {Ret is
X * 5, P is P1 - 1, !}.
663     cadeiaValida(Ret, P) --> [_], cadeiaValida(Ret, P), {!}.
664     cadeiaValida(Ret, 0) --> "", {Ret is 1, !}.
665
666     cadeiaValida2 --> cadeiaValida2(X), {X==1}.
667     cadeiaValida2(Ret) --> "IF", cadeiaValida2(X), {Ret is X
// 4, !}.
668     cadeiaValida2(Ret) --> "THEN", cadeiaValida2(X), {Ret is X
* 2, !}.
669     cadeiaValida2(Ret) --> "ELSE", cadeiaValida2(X), {Ret is X
* 2, !}.
670     cadeiaValida2(Ret) --> "WHILE", cadeiaValida2(X), {Ret is
X // 3, !}.
671     cadeiaValida2(Ret) --> "DO", cadeiaValida2(X), {Ret is X *
3, !}.
672     cadeiaValida2(Ret) --> [_], cadeiaValida2(Ret), {!}.
673     cadeiaValida2(Ret) --> "", {Ret is 1, !}.
674
675     separador -->
676         separador(_, _).
677     separador(X, Y) -->
678         nome(_, _, X, _, _, 0),
679         ";",
680         nome(_, _, Y, _, _, 0),
681         {cadeiaValida(Xv, X, []), Xv=1, cadeiaValida(Yv, Y,
[]), Yv=1}.
682
683     parteString([], X, Z) -->
684         Z, nome(X).
685     parteString([K|Ys], X, Z) -->
686         [K], parteString(Ys, X, Z).
687
688     verificaNaoPR(X):-                               not(dicPR(X)).
```

Compilador

**Nome dos elementos do grupo**

Adriano Teixeira ([adriano.teixeira@fe.up.pt](mailto:adriano.teixeira@fe.up.pt))

Nelson Rodrigues ([nelson@fe.up.pt](mailto:nelson@fe.up.pt))