Capítulo 3

O que é um compilador?

Um compilador é um programa de sistema que traduz um programa descrito em uma linguagem de alto nível para um programa equivalente em código de máquina para um processador. Em geral, um compilador não produz diretamente o código de máquina mas sim um programa em linguagem simbólica *(assembly)* semanticamente equivalente ao programa em linguagem de alto nível. O programa em linguagem simbólica é então traduzido para o programa em linguagem de máquina através de montadores, descritos no Capítulo 10.

Para desempenhar suas tarefas, um compilador deve executar duas atividades básicas. A primeira atividade é a **análise** do código fonte, onde a estrutura e o significado do programa de alto nível são reconhecidos. A segunda atividade é a **síntese**, que traduz o programa a seu equivalente em linguagem simbólica. Embora conceitualmente seja possível executar toda a análise e apenas então iniciar a síntese, em geral as duas atividades ocorrem praticamente em paralelo.

Considere o exemplo do clássico programa em C++*(hello, world)* em versão tupiniquim:

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4    cout << "Oi, gente!" << endl;
5  }
```

Para executar este programa e obter na tela do console a saída *Oi, gente*, os seguintes passos são necessários:

1. Criar o arquivo com o texto do código-fonte.

2. Gerar o arquivo executável a partir do código-fonte criado no passo anterior.

3. Executar o programa gerado.

Para o primeiro passo, é importante que o arquivo criado tenha como conteúdo apenas a seqüência de caracteres que compõe o programa. Para tanto, é preciso

utilizar um editor de programas e não um editor de documentos, pois este inclui instruções de formatação que, embora não visíveis para o autor, estão presentes no arquivo. Basta comparar o tamanho de dois arquivos criados exatamente com o mesmo conteúdo mas com os diferentes tipos de editores[1]:

```
[Exemplos] ls -l
-rw-r--r--        88 Nov 13 08:19 hello.cpp
-rw-r--r--      5246 Nov 13 08:38 helloDoc.cpp
```

Neste exemplo, o primeiro arquivo foi criado com um editor de programas e o outro com um conhecido processador de textos.

Ao usar um editor de documentos para criar um arquivo `helloDoc.cpp` com o conteúdo do programa, o compilador não consegue processar o código-fonte em função dos caracteres estranhos:

```
[Exemplos] g++ helloDoc.cpp
helloDoc.cpp:1: stray ’\3’ in program
helloDoc.cpp:1: stray ’\4’ in program
helloDoc.cpp:1: stray ’\24’ in program
helloDoc.cpp:1:6: warning: null character(s) ignored
[...mais 1000 linhas (literalmente) de mensagens de erro.]
```

Já com o arquivo criado com o editor de programas, nenhuma mensagem de erro é gerada e um arquivo executável é obtido:

```
[Exemplos] g++ hello.cpp
[Exemplos] ls -l
-rwxr-xr-x     13403 Nov 13 08:58 a.out
[...]
```

O compilador assumiu por omissão o nome `a.out` para o arquivo executável. Caso o nome desejado fosse outro, este poderia ser indicado para o compilador através da utilização da chave `-o`:

```
[Exemplos] g++ hello.cpp -o hello
[Exemplos] ls -l
-rwxr-xr-x     13403 Nov 13 09:03 hello
[...]
[Exemplos] ./hello
Oi, gente!
```

Para que o compilador possa produzir o arquivo executável, é preciso que o código-fonte esteja de acordo com as regras da linguagem de programação. Obviamente, o compilador não reconhece programas escritos em outras linguagens de programação, mesmo que parecidas:

[1] Para estes exemplos, foram utilizados os aplicativos do sistema operacional linux.

```
[Exemplos] cat helloJava.cpp
class helloJava {
 public static void main(String[] args) {
  System.out.println("Oi, gente!");
 }
}
[Exemplos] g++ helloJava.cpp
helloJava.cpp:2: parse error before ‘static’
```

Já um pequeno engano no código-fonte é suficiente para impedir a criação do arquivo executável. Por exemplo, ao omitir o caráter terminador de comando (`;`) ao final da linha 4 do arquivo, o compilador gera uma mensagem de erro:

```
hello.cpp: In function ‘int main()’:
hello.cpp:5: parse error before ‘}’ token
```

Observe que a mensagem indica que o erro ocorre na linha 5, que é quando o compilador detectou a ausência do terminador de comando. De fato, esse terminador poderia estar na linha seguinte que o programa compilaria corretamente.

Outras vezes, um pequeno engano no código pode gerar muitas mensagens de erro. É o que ocorre quando o compilador não consegue identificar claramente a causa do erro, o que faz com que haja uma propagação de erros a partir daquele ponto do código. Por exemplo, por omitir apenas o caráter de abertura da seqüência de caracteres (`"`) da linha 4, o compilador gera as seguintes mensagens:

```
[Exemplos] g++ hello.cpp
hello.cpp: In function ‘int main()’:
hello.cpp:4: ‘Oi’ undeclared (first use this function)
hello.cpp:4: (Each undeclared identifier is reported only
   once for each function it appears in.)
hello.cpp:4: ‘gente’ undeclared (first use this function)
hello.cpp:4: parse error before ‘!’ token
hello.cpp:4:21: warning: multi-line string literals are
   deprecated
hello.cpp:4:21: missing terminating " character
hello.cpp:4:21: possible start of unterminated string
   literal
```

Observe que o compilador, ao encontrar os símbolos `Oi` e `gente` na linha 4 do código-fonte, assumiu que estes seriam variáveis do programa e que não haviam sido declaradas. Como todas as variáveis em C++ devem ser declaradas antes do uso, a mensagem indicou esse erro e não a ausência do caráter de abertura da cadeia de caracteres. Nas três últimas linhas, o compilador indica a presença do caráter `"` na linha 4, coluna 21, mas assume que este inicia uma seqüência de caracteres e reclama que não encontrou o final correspondente.

Outro tipo de erro detectado pelo compilador considera apenas os símbolos isoladamente. Por exemplo, a compilação do seguinte programa:

```
1  #include <iostream>
2  using namespace std;
3  int main() {
4    int 1var;
5    cout << 1var << endl;
6  }
```

apresenta a mensagem de erro associada à má formação do nome da variável — neste caso o compilador, ao identificar como primeiro caráter do símbolo um dígito, assumiu que o programador estaria escrevendo uma constante numérica:

```
tkerr.cpp: In function ‘int main()’:
tkerr.cpp:4: invalid suffix on integer constant
tkerr.cpp:4: parse error before numeric constant
tkerr.cpp:5: invalid suffix on integer constant
```

Observe que neste caso o compilador nem chegou a analisar a expressão da linha 4 como um todo, pois neste caso a mensagem seria provavelmente diferente — algo no sentido de que o nome atribuído à variável é inválido. No entanto, o erro foi sinalizado numa etapa em que o compilador simplesmente analisou os símbolos do programa isoladamente.

O programa compilador deve conhecer todas as regras associadas à formação de um programa correto na sua linguagem-alvo e, dado um código-fonte nessa linguagem, verificar se essas regras estão sendo obedecidas. As regras que indicam como deve ser um programa correto são expressas através de gramáticas. A etapa de análise de um programa verifica a aplicação dessas regras e apenas para programas cuja análise tenha sido corretamente concluída é possível concluir com sucesso a compilação, através da síntese do programa equivalente em outro formato.

3.1 Exercícios

3.1 Reproduza no computador no qual você irá desenvolver suas atividades práticas os passos necessários para criar e executar o programa do exemplo `hello.cpp`. Indique o editor e o compilador utilizado, assim como o tamanho do código-fonte e do programa executável. Verifique também que mensagens são geradas para as situações de erro aqui apresentadas.

3.2 Há vários ambientes integrados de desenvolvimento (na sigla de origem inglesa, IDE) que combinam a edição, a compilação e até a execução de programas em C++. Verifique se algum está disponível no computador que você utiliza e descreva-o. Avalie e indique quais as vantagens em utilizar um ambiente desse tipo em relação à abordagem básica de linha de comando que foi apresentada nos exemplos deste capítulo.