Capítulo 2
Estrutura léxica

A estrutura léxica de uma linguagem de programação é o conjunto de regras elementares que especificam o modo de escrever programas nessa linguagem. É a sintaxe de nível mais baixo de uma linguagem; especifica detalhes de como são os nomes de variáveis, os caracteres delimitadores para comentários e como uma instrução do programa é separada da seguinte. Este breve capítulo documenta a estrutura léxica de JavaScript.

2.1 Conjunto de caracteres

Os programas JavaScript são escritos com o conjunto de caracteres Unicode. Unicode é um superconjunto de ASCII e Latin-1 e suporta praticamente todos os idiomas escritos usados hoje no planeta. A ECMAScript 3 exige que as implementações de JavaScript suportem Unicode versão 2.1 ou posterior e a ECMAScript 5 exige que as implementações suportem Unicode 3 ou posterior. Consulte o quadro na Seção 3.2 para mais informações sobre Unicode e JavaScript.

2.1.1 Diferenciação de maiúsculas e minúsculas

JavaScript é uma linguagem que diferencia letras maiúsculas de minúsculas. Isso significa que palavras-chave, variáveis, nomes de função e outros *identificadores* da linguagem sempre devem ser digitados com a composição compatível de letras. A palavra-chave `while`, por exemplo, deve ser digitada como "while" e não como "While" ou "WHILE." Da mesma forma, `online`, `Online`, `OnLine` e `ONLINE` são quatro nomes de variável distintos.

Note, entretanto, que HTML não diferencia letras maiúsculas e minúsculas (embora a XHTML diferencie). Por causa de sua forte associação com JavaScript do lado do cliente, essa diferença pode ser confusa. Muitos objetos e propriedades de JavaScript do lado do cliente têm os mesmos nomes das marcas e atributos HTML que representam. Ao passo que essas marcas e nomes de atributo podem ser digitados com letras maiúsculas ou minúsculas na HTML, em JavaScript elas normalmente devem ser todas minúsculas. Por exemplo, o atributo de rotina de tratamento de evento `onclick` da HTML às vezes é especificado como `onClick` em HTML, mas deve ser especificado como `onclick` no código JavaScript (ou em documentos XHTML).

2.1.2 Espaço em branco, quebras de linha e caracteres de controle de formato

JavaScript ignora os espaços que aparecem entre sinais em programas. De modo geral, JavaScript também ignora quebras de linha (mas consulte a Seção 2.5 para ver uma exceção). Como é possível usar espaços e novas linhas livremente em seus programas, você pode formatar e endentar os programas de um modo organizado e harmonioso, que torne o código fácil de ler e entender.

Além do caractere de espaço normal (`\u0020`), JavaScript também reconhece os seguintes caracteres como espaço em branco: tabulação (`\u0009`), tabulação vertical (`\u000B`), avanço de página (`\u000C`), espaço não separável (`\u00A0`), marca de ordem de byte (`\uFEFF`) e qualquer caractere unicode da categoria Zs. JavaScript reconhece os seguintes caracteres como términos de linha: avanço de linha (`\u000A`), retorno de carro (`\u000D`), separador de linha (`\u2028`) e separador de parágrafo (`\u2029`). Uma sequência retorno de carro, avanço de linha é tratada como um único término de linha.

Os caracteres de controle de formato Unicode (categoria Cf), como RIGHT-TO-LEFT MARK (`\u200F`) e LEFT-TO-RIGHT MARK (`\u200E`), controlam a apresentação visual do texto em que ocorrem. Eles são importantes para a exibição correta de alguns idiomas e são permitidos em comentários, strings literais e expressões regulares literais de JavaScript, mas não nos identificadores (por exemplo, nomes de variável) de um programa JavaScript. Como casos especiais, ZERO WIDTH JOINER (`\u200D`) e ZERO WIDTH NON-JOINER (`\u200C`) são permitidos em identificadores, mas não como o primeiro caractere. Conforme observado anteriormente, o caractere de controle de formato de marca de ordem de byte (`\uFEFF`) é tratado como caractere de espaço.

2.1.3 Sequências de escape Unicode

Alguns componentes de hardware e software de computador não conseguem exibir ou introduzir o conjunto completo de caracteres Unicode. Para ajudar os programadores que utilizam essa tecnologia mais antiga, JavaScript define sequências especiais de seis caracteres ASCII para representar qualquer código Unicode de 16 bits. Esses escapes Unicode começam com os caracteres `\u` e são seguidos por exatamente quatro dígitos hexadecimais (usando as letras A–F maiúsculas ou minúsculas). Os escapes Unicode podem aparecer em strings literais, expressões regulares literais e em identificadores JavaScript (mas não em palavras-chave da linguagem). O escape Unicode para o caractere "é", por exemplo, é `\u00E9`, e as duas strings JavaScript a seguir são idênticas:

```
"café" === "caf\u00e9"          // => true
```

Os escapes Unicode também podem aparecer em comentários, mas como os comentários são ignorados, eles são tratados como caracteres ASCII nesse contexto e não são interpretados como Unicode.

2.1.4 Normalização

O Unicode permite mais de uma maneira de codificar o mesmo caractere. A string "é", por exemplo, pode ser codificada como o caractere Unicode `\u00E9` ou como um caractere ASCII "e" normal, seguido da marca de combinação de acento agudo `\u0301`. Essas duas codificações podem parecer exatamente a mesma quando exibidas por um editor de textos, mas têm diferentes codificações binárias e são consideradas diferentes pelo computador. O padrão Unicode define a codificação preferida para todos os caracteres e especifica um procedimento de normalização para converter texto

em uma forma canônica conveniente para comparações. JavaScript presume que o código-fonte que está interpretando já foi normalizado e não tenta normalizar identificadores, strings nem expressões regulares.

2.2 Comentários

JavaScript aceita dois estilos de comentários. Qualquer texto entre // e o final de uma linha é tratado como comentário e é ignorado por JavaScript. Qualquer texto entre os caracteres /* e */ também é tratado como comentário; esses comentários podem abranger várias linhas, mas não podem ser aninhados. As linhas de código a seguir são todas comentários JavaScript válidos:

```
// Este é um comentário de uma linha.
/* Este também é um comentário */ // e aqui está outro comentário.
/*
* Este é ainda outro comentário.
* Ele tem várias linhas.
*/
```

2.3 Literais

Um *literal* é um valor de dados que aparece diretamente em um programa. Os valores seguintes são todos literais:

```
12                // O número doze
1.2               // O número um ponto dois
"hello world"     // Uma string de texto
'Hi'              // Outra string
true              // Um valor booleano
false             // O outro valor booleano
/javascript/gi    // Uma "expressão regular" literal (para comparação de padrões)
null              // Ausência de um objeto
```

Os detalhes completos sobre literais numéricos e string aparecem no Capítulo 3. As expressões regulares literais são abordadas no Capítulo 10. Expressões mais complexas (consulte a Seção 4.2) podem servir como array e objeto literais. Por exemplo:

```
{ x:1, y:2 }       // Um inicializador de objeto
[1,2,3,4,5]        // Um inicializador de array
```

2.4 Identificadores e palavras reservadas

Um *identificador* é simplesmente um nome. Em JavaScript, os identificadores são usados para dar nomes a variáveis e funções e para fornecer rótulos para certos laços no código JavaScript. Um identificador JavaScript deve começar com uma letra, um sublinhado (_) ou um cifrão ($). Os caracteres subsequentes podem ser letras, dígitos, sublinhados ou cifrões. (Os dígitos não são permitidos como primeiro caractere, para que JavaScript possa distinguir identificadores de números facilmente.) Todos estes são identificadores válidos:

```
i
my_variable_name
v13
```

```
_dummy
$str
```

Por portabilidade e facilidade de edição, é comum usar apenas letras e dígitos ASCII em identificadores. Note, entretanto, que JavaScript permite que os identificadores contenham letras e dígitos do conjunto de caracteres Unicode inteiro. (Tecnicamente, o padrão ECMAScript também permite que caracteres Unicode das categorias obscuras Mn, Mc e Pc apareçam em identificadores, após o primeiro caractere.) Isso permite que os programadores utilizem nomes de variável em outros idiomas e também usem símbolos matemáticos:

```
var sí = true;
var π = 3.14;
```

Assim como qualquer linguagem, JavaScript reserva certos identificadores para uso próprio. Essas "palavras reservadas" não podem ser usadas como identificadores normais. Elas estão listadas a seguir.

2.4.1 Palavras reservadas

JavaScript reserva vários identificadores como palavras-chave da própria linguagem. Você não pode usar essas palavras como identificadores em seus programas:

```
break       delete      function    return      typeof
case        do          if          switch      var
catch       else        in          this        void
continue    false       instanceof  throw       while
debugger    finally     new         true        with
default     for         null        try
```

JavaScript também reserva certas palavras-chave não utilizadas atualmente na linguagem, mas que poderão ser usadas em futuras versões. A ECMAScript 5 reserva as seguintes palavras:

```
class   const   enum   export   extends   import   super
```

Além disso, as seguintes palavras, que são válidas em código JavaScript normal, são reservadas no modo restrito:

```
implements  let     private     public      yield
interface   package   protected   static
```

O modo restrito também impõe restrições sobre o uso dos identificadores a seguir. Eles não são totalmente reservados, mas não são permitidos como nomes de variável, função ou parâmetro:

```
arguments   eval
```

ECMAScript 3 reservou todas as palavras-chave da linguagem Java e, embora tenham sido consentidos em ECMAScript 5, você ainda deve evitar todos esses identificadores, caso pretenda executar seu código em uma implementação ECMAScript 3 de JavaScript:

```
abstract    double      goto        native      static
boolean     enum        implements  package     super
byte        export      import      private     synchronized
char        extends     int         protected   throws
class       final       interface   public      transient
const       float       long        short       volatile
```

JavaScript predefine diversas variáveis e funções globais e você deve evitar o uso de seus nomes em suas próprias variáveis e funções:

```
arguments              encodeURI              Infinity    Number           RegExp
Array                  encodeURIComponent     isFinite    Object           String
Boolean                Error                  isNaN       parseFloat       SyntaxError
Date                   eval                   JSON        parseInt         TypeError
decodeURI              EvalError              Math        RangeError       undefined
decodeURIComponent     Function               NaN         ReferenceError   URIError
```

Lembre-se de que as implementações de JavaScript podem definir outras variáveis e funções globais, sendo que cada incorporação de JavaScript específica (lado do cliente, lado do servidor, etc.) terá sua própria lista de propriedades globais. Consulte o objeto Window na Parte IV para ver uma lista das variáveis e funções globais definidas por JavaScript do lado do cliente.

2.5 Pontos e vírgulas opcionais

Assim como muitas linguagens de programação, JavaScript usa o ponto e vírgula (;) para separar instruções (consulte o Capítulo 5). Isso é importante para tornar claro o significado de seu código: sem um separador, o fim de uma instrução pode parecer ser o início da seguinte ou vice-versa. Em JavaScript, você normalmente pode omitir o ponto e vírgula entre duas instruções, caso essas instruções sejam escritas em linhas separadas. (Você também pode omitir um ponto e vírgula no final de um programa ou se o próximo sinal do programa for uma chave de fechamento }.) Muitos programadores JavaScript (e o código deste livro) utilizam pontos e vírgulas para marcar explicitamente os finais de instruções, mesmo onde eles não são obrigatórios. Outro estilo é omitir os pontos e vírgulas quando possível, utilizando-os nas poucas situações que os exigem. Qualquer que seja o estilo escolhido, existem alguns detalhes que você deve entender sobre os pontos e vírgulas opcionais em JavaScript.

Considere o código a seguir. Como as duas instruções aparecem em linhas separadas, o primeiro ponto e vírgula poderia ser omitido:

```
a = 3;
b = 4;
```

Contudo, escrito como a seguir, o primeiro ponto e vírgula é obrigatório:

```
a = 3; b = 4;
```

Note que JavaScript não trata toda quebra de linha como ponto e vírgula: ela normalmente trata as quebras de linha como pontos e vírgulas somente se não consegue analisar o código sem os pontos e vírgulas. Mais formalmente (e com as duas exceções, descritas a seguir), JavaScript trata uma quebra de linha como ponto e vírgula caso o próximo caractere que não seja espaço não possa ser interpretado como a continuação da instrução corrente. Considere o código a seguir:

```
var a
a
=
3
console.log(a)
```

JavaScript interpreta esse código como segue:

```
var a; a = 3; console.log(a);
```

JavaScript trata a primeira quebra de linha como ponto e vírgula porque não pode analisar o código `var a a` sem um ponto e vírgula. O segundo a poderia aparecer sozinho como a instrução `a;`, mas JavaScript não trata a segunda quebra de linha como ponto e vírgula porque pode continuar analisando a instrução mais longa `a = 3;`.

Essas regras de término de instrução levam a alguns casos surpreendentes. O código a seguir parece ser duas instruções distintas, separadas por uma nova linha:

```
var y = x + f
(a+b).toString()
```

Porém, os parênteses na segunda linha de código podem ser interpretados como uma chamada de função de `f` da primeira linha, sendo que JavaScript interpreta o código como segue:

```
var y = x + f(a+b).toString();
```

Muito provavelmente, essa não é a interpretação pretendida pelo autor do código. Para funcionarem como duas instruções distintas, é necessário um ponto e vírgula explícito nesse caso.

Em geral, se uma instrução começa com (, [, /, + ou -, há a possibilidade de que possa ser interpretada como uma continuação da instrução anterior. Na prática, instruções começando com /, + e - são muito raras, mas instruções começando com (e [não são incomuns, pelo menos em alguns estilos de programação com JavaScript. Alguns programadores gostam de colocar um ponto e vírgula protetor no início de uma instrução assim, para que continue a funcionar corretamente mesmo que a instrução anterior seja modificada e um ponto e vírgula, anteriormente de término, removido:

```
var x = 0                            //Ponto e vírgula omitido aqui
;[x,x+1,x+2].forEach(console.log)    // 0 ; protetor mantém esta instrução
                                     //separada
```

Existem duas exceções à regra geral de que JavaScript interpreta quebras de linha como pontos e vírgulas quando não consegue analisar a segunda linha como uma continuação da instrução da primeira linha. A primeira exceção envolve as instruções `return`, `break` e `continue` (consulte o Capítulo 5). Essas instruções frequentemente aparecem sozinhas, mas às vezes são seguidas por um identificador ou por uma expressão. Se uma quebra de linha aparece depois de qualquer uma dessas palavras (antes de qualquer outro sinal), JavaScript sempre interpreta essa quebra de linha como um ponto e vírgula. Por exemplo, se você escreve:

```
return
true;
```

JavaScript presume que você quis dizer:

```
return; true;
```

Contudo, você provavelmente quis dizer:

```
return true;
```

Isso significa que não se deve inserir uma quebra de linha entre `return`, `break` ou `continue` e a expressão que vem após a palavra-chave. Se você inserir uma quebra de linha, seu código provavelmente vai falhar de uma maneira inesperada, difícil de depurar.

A segunda exceção envolve os operadores `++` e `--` (Seção 4.8). Esses podem ser operadores prefixados, que aparecem antes de uma expressão, ou operadores pós-fixados, que aparecem depois de uma expressão. Se quiser usar um desses operadores como pós-fixados, eles devem aparecer na mesma linha da expressão em que são aplicados. Caso contrário, a quebra de linha vai ser tratada como ponto e vírgula e o operador `++` ou `--` vai ser analisado como um operador prefixado aplicado ao código que vem a seguir. Considere este código, por exemplo:

```
x
++
y
```

Ele é analisado como `x; ++y;` e não como `x++; y`.