Perl语言及其在IC设计中的应用

许建超，2023/08/28

Perl 培训提纲

1. **Perl 简介**
    - Perl 的历史和背景
    - Perl 的主要应用领域
    - 第一个 Perl 程序
2. **Perl 的基础语法：变量**
    - 标量、数组和哈希
    - 变量声明和赋值
    - 基础的数据类型（字符串、数字、布尔值）
    - Perl 字符串详解
    - Perl 数组详解
    - Perl 哈希详解
    - 综合实例
3. **Perl 的基础语法：表达式**
    - 算术表达式
    - 字符串表达式
    - 比较表达式
    - 逻辑表达式
    - 赋值表达式
4. **Perl 的基础语法：控制结构**
    - 条件语句（`if`, `else`, `elsif`）
    - 循环语句（`for`, `foreach`, `while`）
    - 控制程序跳转语句（`next`, `last`, `return`）
5. **Perl 的基础语法：子函数**
    - 子函数的定义与调用
    - 参数传递和返回值
    - 递归
6. **Perl 的基础语法：文件操作与 Print/Printf 功能**
    - 打开和关闭文件
    - 读取和写入文件

    - print 和 printf

7. **Perl 的强大利器：正则表达式(入门)**
    - 基础的正则语法
    - 使用正则进行匹配和替换
    - 综合实例
8. **实践：应用于 IC 设计中的两个 Perl 脚本（片段）**
    - 例子1：从 spice 网表中提取指定电路的端口, 并自动生成激励
    - 例子2：深入和灵活分析 dspf 格式后仿真网表的寄生电容情况

第一节. Perl 简介

Perl 的历史和背景

- **创始**: Perl 由 Larry Wall 在 1987 年创立，最初是为了更好地处理文本报告。
- **发展**: Perl 经历了多次重要的版本更新，最为人知的是 Perl 5 (1994 年) 和 Perl 6 (现在被称为 Raku，独立于 Perl 5)。
- **特点**: Perl 被称为"程序员的瑞士军刀"，因为它功能强大而且灵活，尤其在文本处理方面。

Perl 的特色

- **正则表达式**: Perl 对正则表达式的支持非常完善，让文本处理变得非常简单。
- **CPAN (Comprehensive Perl Archive Network)**: 一个巨大的 Perl 模块库，拥有大量的模块和文档，为 Perl 社区提供了强大的支持。
- **灵活性**: Perl 的语法灵活，有时甚至被描述为"有多种方法做同一件事"。
- **类C语法**: Perl 的语法结构，与C语言非常相似。Perl程序可以写的非常复杂难懂，各种天书的符号，但也可以写的简洁优雅，跟C语言非常相似。不同程序语言之间的差异，用下面的例子可以看出：

分别用c语言，perl语言，python 语言，写一个求1-1000以内所有质数之和的程序。可以看到，perl语言与c语言非常相似。python 语言则有很大的不同。

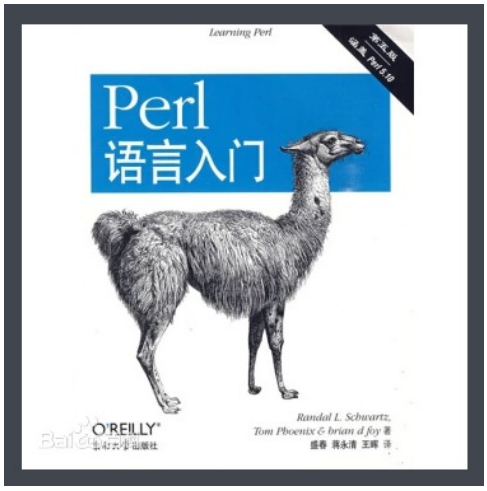
```
#include <stdio.h>
#include <stdbool.h>

bool is_prime(int num) {
    if (num <= 1) return false;
    if (num <= 3) return true;

    if (num % 2 == 0 || num % 3 == 0) return false;

    int i = 5;
    while (i * i <= num) {
        if (num % i == 0 || num % (i + 2) == 0)
            return false;
        i += 6;
    }
    return true;
}

int main() {
    int sum = 0;
    for (int i = 2; i <= 1000; i++) {
        if (is_prime(i)) {
            sum += i;
        }
    }
    printf("The sum of primes up to 1000 is: %d\n", sum);
    return 0;
}
```

```perl
#!/usr/bin/perl
use strict;
use warnings;

sub is_prime {
    my $num = shift;
    if ($num <= 1) {
        return 0;
    }
    if ($num <= 3) {
        return 1;
    }

    if ($num % 2 == 0 || $num % 3 == 0) {
        return 0;
    }

    my $i = 5;
    while ($i * $i <= $num) {
        if ($num % $i == 0 || $num % ($i + 2) == 0) {
            return 0;
        }
        $i += 6;
    }
    return 1;
}

my $sum = 0;
for (my $i = 2; $i <= 1000; $i++) {
    if(is_prime($i)){
        $sum += $i;
    }
}
print "The sum of primes up to 1000 is: $sum\n";
```

```python
def is_prime(num):
    if num <= 1:
        return False
    if num <= 3:
        return True

    if num % 2 == 0 or num % 3 == 0:
        return False

    i = 5
    while i * i <= num:
        if num % i == 0 or num % (i + 2) == 0:
            return False
        i += 6
    return True

sum_primes = sum(i for i in range(2, 1001) if is_prime(i))
print(f"The sum of primes up to 1000 is: {sum_primes}")
```

Perl 的主要应用领域

- **文本处理**: 由于其强大的正则表达式支持，Perl 经常被用于日志分析、报告生成等。
- **网络编程**: Perl 也常用于网络相关的脚本，如 CGI 脚本。
- **系统管理**: Perl 经常被系统管理员用于自动化任务和脚本。
- **生物信息学**: Perl 在生物信息学领域有广泛的应用，用于处理和分析生物数据。
- **IC设计**: Perl 在IC设计有广泛的应用，用于构造各种脚本，进行仿真pattern的大批量自动构建，仿真结果的summary，从EDA工具的log文件中提取有用信息。



第一个 Perl 程序

```
#!/usr/bin/perl  # 声明使用 perl 解释器

use strict; # 引入 strict 模块，它强制声明所有变量，有助于避免某些常见的错误
use warnings; # 引入 warnings 模块，它使 Perl 输出有关潜在问题的警告

# 打印 "Hello, World!" 到标准输出，并附加一个换行符
print "Hello, World!\n";
```

第二节. Perl 的基础语法--变量

1. 基础的数据类型

- **字符串 (String)**: 文本数据，用双引号或单引号括起来。

  ```
  my $greeting = "Hello, World!";
  ```

- **数字 (Number)**: 整数或浮点数。

  ```
  my $integer = 10;
  my $float = 10.5;
  ```

- **布尔值 (Boolean)**: Perl 没有专门的布尔数据类型，但传统上，数字 0、字符串 "0"、空字符串、未定义的

值都被视为假（False），其他值被视为真（True）。

```
my $true_value = 1;
my $false_value = 0;
```

2. 标量、数组和哈希

- **标量 (Scalar)**: 用于存储单一值，如字符串、数字或引用。标量变量以 `$` 开头。

```
my $name = "Alice";
```

- **数组 (Array)**: 用于存储值的有序列表。数组变量以 `@` 开头。

```
my @fruits = ("apple", "banana", "cherry");
```

- **哈希 (Hash)**: 键值对的集合，也称为关联数组。哈希变量以 `%` 开头。

```
my %colors = ("apple" => "red", "banana" => "yellow");
```

3. Perl 字符串，详细讲解

字符串是 Perl 中最为核心和强大的部分之一，再结合Perl强大的正则表达式，形成了Perl的强大的文本处理能力。

3.1. 定义和初始化

在 Perl 中，可以使用单引号或双引号来定义字符串。

```
my $str1 = 'Hello, World!'; # 使用单引号
my $str2 = "Hello, Perl!";  # 使用双引号
```

两者之间的主要区别是：双引号内的字符串可以解析变量和特殊字符，而单引号则不行。

3.2. 字符串中的变量

在双引号字符串中，可以直接插入*变量。

```
my $name = "Alice";
print "Hello, $name!\n";  # 输出 "Hello, Alice!"
```

3.3. 特殊字符

在双引号字符串中，可以使用反斜线 `\` 来表示特殊字符：

- `\n`: 换行符
- `\t`: 制表符
- `\"`: 双引号
- `\\`: 反斜线

```
print "Line 1\nLine 2\n";
```

3.4. 连接字符串

使用 `.` 运算符连接字符串。

```
my $first = "Hello, ";
my $second = "World!";
print $first . $second;  # 输出 "Hello, World!"
```

3.5. 字符串长度

使用 `length` 函数获取字符串的长度。

```
my $str = "Hello";
print length $str;  # 输出 5
```

3.6. 字符串操作

- `substr`: 获取字符串的子串。

  ```
  my $part = substr("Hello, World!", 7, 5);  # "World"
  ```

- `index`: 在字符串中查找子串的位置。

  ```
  my $position = index("Hello, World!", "World");  # 返回 7
  ```

- `chop`: 移除字符串的最后一个字符。
- `chomp`: 移除字符串的尾部换行符（如果存在）。

3.7. 字符串比较

- `eq`: 检查两个字符串是否相等。
- `ne`: 检查两个字符串是否不相等。
- `lt`, `gt`, `le`, `ge`: 字符串的字典序比较。

```
if ($str1 eq $str2) {
    print "Both strings are equal.\n";
}
```

3.8. 多行字符串

```
my $text =qq{
This is a
multiline
string.
};
```

3.9. `split` 和 `join` 函数

`split` 函数是 Perl 中用于将字符串分割成数组的主要工具。

基本语法：

```
@array = split(/PATTERN/, $string);
```

- **PATTERN**：正则表达式，定义如何分割字符串。
- **$string**：要分割的字符串。

示例：

```
my $data = "apple,orange,banana";
my @fruits = split(/,/, $data);
print "@fruits\n";  # 输出 "apple orange banana"
```

`join` 函数是 `split` 的反操作，它将数组的元素组合成一个字符串。

基本语法：

```
$string = join(EXPR, LIST);
```

- **EXPR**：连接数组元素的字符串或字符。
- **LIST**：要连接的数组或列表。

示例：

```
my @names = ("Alice", "Bob", "Charlie");
my $joined_names = join(" & ", @names);
print "$joined_names\n";  # 输出 "Alice & Bob & Charlie"
```

综合示例：split 和 join 的组合使用

考虑一个任务，其中你需要将一个逗号分隔的字符串转换为一个由"&"连接的字符串：

```
my $data = "apple,orange,banana";
my @fruits = split(/,/, $data);
my $result = join(" & ", @fruits);
print "$result\n";  # 输出 "apple & orange & banana"
```

4. Perl 数组，详细讲解

4.1. 定义和初始化

在 Perl 中，数组是一个有序的元素列表。数组变量以 `@` 符号开头。

```
# 空数组
my @empty_array = ();

# 包含三个元素的数组
my @colors = ("red", "green", "blue");
```

4.2. 索引

Perl 数组的索引从 0 开始。您可以使用负数从数组的末尾开始反向索引。

```
my @fruits = ("apple", "banana", "cherry");

print $fruits[0]; # 输出 "apple"
print $fruits[-1]; # 输出 "cherry"
```

4.3. push 和 pop

- `push` 函数可以在数组的末尾添加一个或多个元素。
- `pop` 函数可以从数组的末尾移除并返回一个元素。

```
my @numbers = (1, 2, 3);

push @numbers, 4, 5; # @numbers 现在是 (1, 2, 3, 4, 5)
my $last = pop @numbers; # $last 的值为 5, @numbers 现在是 (1, 2, 3, 4)
```

4.4. shift 和 unshift

- `shift` 函数可以从数组的开始移除并返回一个元素。
- `unshift` 函数可以在数组的开始添加一个或多个元素。

```
my @animals = ("cat", "dog", "fish");

unshift @animals, "bird"; # @animals 现在是 ("bird", "cat", "dog", "fish")
my $first = shift @animals; # $first 的值为 "bird", @animals 现在是 ("cat", "dog", "fish")
```

4.5. 数组长度

使用 `scalar` 函数和数组上下文可以获得数组的长度。

```
my @items = ("a", "b", "c");
my $length = scalar @items; # $length 的值为 3
```

4.6. 切片

可以使用数组切片来获取数组的子集。

```
my @weekdays = ("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun");
my @weekend = @weekdays[5, 6]; # @weekend 是 ("Sat", "Sun")
```

4.7. 遍历数组

使用 `foreach` 循环遍历数组的每个元素。

```
my @languages = ("Perl", "Python", "Ruby");
foreach my $lang (@languages) {
    print "$lang\n";
}
```

5. Perl 哈希，详细讲解

5.1. 定义和初始化

哈希用于存储键和值之间的关系。哈希变量以 `%` 符号开头。

```
# 空哈希
my %empty_hash = ();

# 定义一个哈希
my %fruit_color = (
    "apple"  => "red",
    "banana" => "yellow",
    "grape"  => "purple"
);
```

5.2. 索引

可以使用 `{}` 来根据键访问哈希中的值。

```
my %prices = (
    "apple"  => 0.5,
    "orange" => 0.3
);

print $prices{"apple"}; # 输出 0.5
```

5.3. 添加和删除键值对

- 添加新的键值对很简单，只需为新键分配一个值。
- 使用 `delete` 函数删除键值对。

```
# 添加
$prices{"banana"} = 0.4;

# 删除
delete $prices{"apple"};
```

5.4. 检查键是否存在

使用 `exists` 函数检查键是否存在于哈希中。

```
if (exists $prices{"apple"}) {
    print "Apple price: $prices{'apple'}\n";
} else {
    print "Apple not found in prices hash.\n";
}
```

5.5. 获取所有键和值

- `keys` 函数返回哈希的所有键。
- `values` 函数返回哈希的所有值。

```
my @all_keys = keys %prices;
my @all_values = values %prices;
```

5.6. 打印哈希

可以使用循环来打印哈希的内容。

```
foreach my $key (keys %prices) {
    print "$key: $prices{$key}\n";
}
```

或者更简洁地使用 `while` 和 `each`：

```
while (my ($key, $value) = each %prices) {
    print "$key: $value\n";
}
```

5.7. 哈希大小

与数组一样，使用 `scalar` 函数可以获得哈希的大小（键的数量）。

```
my $size = scalar keys %prices;
```

6. 综合示例：个人信息管理

```perl
#!/usr/bin/perl
use strict;
use warnings;

# 标量变量
my $name = "Alice";
my $age = 30;

# 数组变量
my @hobbies = ("reading", "hiking", "coding");

# 哈希变量
my %contact_info = (
    "email" => "alice\@example.com",
    "phone" => "123-456-7890",
    "address" => "123 Elm St, Wonderland"
);

# 打印个人信息
print "Name: $name\n";
print "Age: $age\n";
print "Hobbies: ", join(", ", @hobbies), "\n";

# 打印联系信息
print "Contact Info:\n";
foreach my $key (keys %contact_info) {
    print "$key: $contact_info{$key}\n";
}

# 添加一个新的爱好
push @hobbies, "painting";

# 更新联系信息
$contact_info{"email"} = "new_email\@example.com";

# 再次打印个人信息
print "\nUpdated Information:\n";
print "Hobbies: ", join(", ", @hobbies), "\n";
print "Email: $contact_info{'email'}\n";
```

此脚本首先声明了一个标量变量（名字和年龄）、一个数组变量（爱好）和一个哈希变量（联系信息）。然后，它打印出这些变量的值。接下来，我们向爱好列表中添加了一个新的爱好，并更新了电子邮件地址。最后，我们再次打印出更新后的爱好和电子邮件地址。

这个示例展示了如何声明、赋值、访问和更新 Perl 中的各种变量。希望这有助于您更全面地理解 Perl 的基础语法！

第三节. Perl 的基础语法--表达式

Perl 中的表达式是一种结构，它评估为一个值。表达式在 Perl 中是非常重要的，因为它们是构建复杂逻辑和操作的基础。以下是 Perl 中最常用的一些表达式及其详细说明：

1. 算术表达式

这些表达式用于执行基本的算术运算。

- 加法: `+`
- 减法: `-`
- 乘法: `*`
- 除法: `/`
- 取模: `%`
- 自增: `++`
- 自减: `--`

```
my $sum = 5 + 3;  # 结果为 8
my $product = 5 * 3;  # 结果为 15

my $sum=0;
for(my $i=0; $i<10; $i++){
    $sum+=$i*$i;
}
```

2. 字符串表达式

这些表达式用于执行字符串操作。

- 连接: `.`
- 复制: `x`

```
my $greeting = "Hello" . " World";  # "Hello World"
my $repeated = "Na" x 4;  # "NaNaNaNa"
```

3. 比较表达式

这些表达式用于比较值。

- 数字比较:
  - 等于: `==`
  - 不等于: `!=`
  - 小于: `<`
  - 大于: `>`
  - 小于或等于: `<=`
  - 大于或等于: `>=`
- 字符串比较:
  - 等于: `eq`
  - 不等于: `ne`
  - 小于: `lt`
  - 大于: `gt`
  - 小于或等于: `le`

- 大于或等于: `ge`

```
if (5 == 5) { ... }
if ("apple" eq "orange") { ... }
```

4. 逻辑表达式

这些表达式用于执行逻辑运算。

- 逻辑与: `&&` 或 `and`
- 逻辑或: `||` 或 `or`
- 逻辑非: `!` 或 `not`
- 逻辑异或: `xor`

```
if ($a > 5 && $b < 10) { ... }
if ($str eq "apple" || $str eq "orange") { ... }
```

5. 赋值表达式

这些表达式用于给变量赋值。

- 基本赋值: `=`
- 复合赋值: `+=`, `-=`, `*=`, `/=`, `.=` 等

```
my $x = 5;
$x += 3;  # $x 的值现在为 8
```

6. 条件表达式 (三元操作符)

这是一个简单的 "if-else" 表达式，格式为 `条件 ? 值1 : 值2`。

```
my $max = $a > $b ? $a : $b;
```

7. 范围表达式

这是一个产生数字或字符范围的表达式。

```
my @numbers = (1..10);  # (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
my @letters = ('a'..'f');  # (a, b, c, d, e, f)
```

当然可以，控制结构是编程语言中的基石，它们用于确定代码的执行顺序。以下是 Perl 控制结构的详细讲解：

第四节. Perl 的基础语法--控制结构

条件语句 (if, else, elsif)
循环语句 (for, foreach, while, until)

控制程序跳转语句（next ,last, return)

1. 条件语句

a. `if`

这是最基本的条件语句，用于根据一个条件执行一段代码。

```
my $x = 10;
if ($x > 5) {
    print "$x is greater than 5\n";
}
```

b. `else`

与 `if` 配合使用，当 `if` 的条件不成立时执行。

```
if ($x > 20) {
    print "$x is greater than 20\n";
} else {
    print "$x is not greater than 20\n";
}
```

c. `elsif`

用于在同一个 `if` 语句中检查多个条件。

```
if ($x > 20) {
    print "$x is greater than 20\n";
} elsif ($x > 10) {
    print "$x is greater than 10 but not greater than 20\n";
} else {
    print "$x is not greater than 10\n";
}
```

2. 循环语句

a. `for`

与 C 语言中的 `for` 循环相似，用于指定次数的循环。

```
for (my $i = 0; $i < 5; $i++) {
    print "This is loop iteration $i\n";
}
```

b. `foreach`

用于遍历数组或哈希的元素。
注：for 和 foreach 没啥区别，我习惯用 for 。

```
my @fruits = ("apple", "banana", "cherry");
foreach my $fruit (@fruits) {
    print "Fruit: $fruit\n";
}
```

或者更简短地：

```
foreach (@fruits) {
    print "Fruit: $_\n";  # $_ 是默认的遍历变量
}
```

c. `while`

当给定条件为真时，重复执行代码块。

```
my $count = 0;
while ($count < 5) {
    print "Count: $count\n";
    $count++;
}
```

3. 控制程序跳转语句

a. `next`

在循环中，`next` 关键字用于立即跳到下一次循环迭代，跳过循环体中的任何后续代码。

```
for my $i (1..5) {
    if ($i == 3) {
        next;
    }
    print "$i\n"; # 将不会打印数字 3
}
```

b. `last`

`last` 关键字用于立即退出循环，不再执行任何后续的循环迭代。

```
my $j = 1;
while ($j <= 5) {
    if ($j == 4) {
        last;
    }
    print "$j\n"; # 将只打印 1, 2, 和 3
    $j++;
}
```

c. `return`

虽然 `return` 通常与函数和子例程相关，但它也可以用于控制结构中，尤其是当您希望从一个深层嵌套的控制结构中返回值时。

```
sub find_number {
    my @numbers = @_;
    foreach my $num (@numbers) {
        if ($num == 5) {
            return "Found!";
        }
    }
    return "Not Found!";
}

print find_number(1, 2, 3, 4, 5, 6);  # 输出 "Found!"
```

4. 综合示例：查找数组中的奇数，并跳过特定的数

```
#!/usr/bin/perl
use strict;
use warnings;

sub find_odds {
    my @numbers = @_;
    my @odds = ();

    for my $num (@numbers) {
        # 跳过 5 和 7
        next if $num == 5 or $num == 7;

        # 如果数字大于 20，则停止查找并返回结果
        last if $num > 20;

        # 如果数字是 11，打印
        if ($num == 11) {
            print "Number is 11\n";
        }

        # 添加奇数到 @odds 数组
        if ($num % 2 != 0) {
            push @odds, $num;
        }
    }

    return @odds;
}

my @nums = (1, 2, 3, 4, 5, 11, 13, 7, 19, 22);
my @result = find_odds(@nums);
print "Odd numbers found: ", join(", ", @result), "\n";
```

此代码定义了一个 `find_odds` 函数，该函数接收一个数字数组，并返回数组中的所有奇数，但跳过数字 5 和 7。如果遇到大于 20 的数字，函数会停止处理并返回。此外，每次遇到数字 11 时，执行打印任务。

第五节. Perl 的基础语法--子函数

Perl 的子函数（也称为子程序或简单地称为函数）允许您将代码组织成独立的、可重用的逻辑块。子函数在 Perl 中非常有用，因为它们可以帮助您模块化代码、提高代码的可读性和重用性。

1. 定义子函数

子函数是使用 `sub` 关键字定义的：

```
sub function_name {
    # function body
}
```

例如：

```
sub greet {
    print "Hello, World!\n";
}
```

2. 调用子函数

定义了子函数后，可以通过其名称来调用它：

```
greet();  # 输出 "Hello, World!"
&greet();  # 输出 "Hello, World!"
```

3. 局部变量定义

使用 `my` 关键字可以在子函数内部声明局部变量，这些变量的作用域仅限于该子函数。

```
sub example_scope {
    my $local_var = "I'm local!";
    print $local_var;  # 输出 "I'm local!"
}

example_scope();
# print $local_var;  # 这会导致错误，因为 $local_var 在这里是不可见的
```

4. 传递参数

当调用子函数时，可以传递参数给它。在子函数内部，这些参数存储在特殊数组 `@_` 中。

```
sub greet_person {
    my ($name) = @_;
    print "Hello, $name!\n";
}

greet_person('Alice');  # 输出 "Hello, Alice!"
```

传递多个参数，并在函数内部按顺序接收它们。

```
Copy code
sub add {
    my ($a, $b) = @_;
    return $a + $b;
}

print add(3, 4);  # 输出 7
```

传递数组或哈希时，通常会将其展开为列表。

```
Copy code
sub print_array {
    my @arr = @_;
    print "Array: @arr\n";
}

my @example = (1, 2, 3, 4);
print_array(@example);  # 输出 "Array: 1 2 3 4"
```

5. 返回值

子函数可以使用 `return` 关键字返回一个值。如果没有明确使用 `return`，子函数将返回其最后一条语句的值。

返回标量

```
sub add {
    my ($a, $b) = @_;
    return $a + $b;
}

my $sum = add(3, 4);  # $sum 的值为 7
```

返回数组

```
sub get_numbers {
    my @array= (1, 2, 3, 4, 5);
    return @array;
}

my @nums = get_numbers();
print "@nums";  # 输出 "1 2 3 4 5"
```

返回哈希

```
sub get_hash {
    my %fruit=(apple => "fruit", carrot => "vegetable");
    return %fruit;
}

my %data = get_hash();
print "$data{apple} is a $data{apple}\n";  # 输出 "apple is a fruit"
```

返回多个值

```
sub get_details {
    return ("Alice", 30, "Engineer");
}

my ($name, $age, $job) = get_details();
print "$name is a $job and is $age years old.\n";  # 输出 "Alice is a Engineer and is 30
years old."
```

6. 递归

像其他编程语言一样，Perl 子函数也可以递归调用自身。

```
sub factorial {
    my ($n) = @_;
    return 1 if $n <= 1;
    return $n * factorial($n - 1);
}

print factorial(5);  # 输出 120
```

第六节. Perl 的基础语法--文件读/写，以及print/printf功能

我们通过一个完整的例子展示这些基本操作。

示例任务：

我们有一个文件 `input.txt`，其中包含商品及其价格。我们将：

1. 读取该文件的内容。
2. 为每个商品计算总价（假设数量为5）。
3. 将格式化的结果写入 `output.txt`。

文件 `input.txt` 的内容：

```
Apple,0.65
Orange,0.85
Banana,0.55
```

Perl 代码：

```
#!/usr/bin/perl
use strict;
use warnings;

# 1. 读取文件内容
open(my $in_fh, '<', 'input.txt') or die "Could not open file 'input.txt' $!";
my @items;

while (my $line = <$in_fh>) {
    chomp $line;
    my ($item, $price) = split(',', $line);
    push @items, {item => $item, price => $price};
}
close $in_fh;

# 2. 计算总价
foreach my $record (@items) {
    $record->{total} = $record->{price} * 5;  # 假设数量为5
}

# 3. 写入格式化的结果到 output.txt
open(my $out_fh, '>', 'output.txt') or die "Could not open file 'output.txt' $!";
printf $out_fh "%-10s | %6s | %6s\n", "Item", "Price", "Total";
print $out_fh "-" x 30 . "\n";

foreach my $record (@items) {
    printf $out_fh "%-10s | $%6.2f | $%6.2f\n", $record->{item}, $record->{price}, $record->
{total};
}
close $out_fh;

print "Processing complete. Check 'output.txt' for formatted results.\n";
```

执行上述代码后，`output.txt` 文件的内容应该是：

```
Item       |  Price |  Total
------------------------------
Apple      | $  0.65 | $  3.25
Orange     | $  0.85 | $  4.25
Banana     | $  0.55 | $  2.75
```

这个示例演示了如何从文件读取数据、处理数据以及如何使用 `print` 和 `printf` 将格式化的结果写入文件。

在提供的示例中，`@items` 是一个数组，但它不是一个普通的数组，它是一个数组的数组引用。每个数组元素都是一个哈希引用，代表一个商品记录。这是它的数据结构：

```
@items = (
    { item => "Apple",  price => 0.65, total => 3.25 },
    { item => "Orange", price => 0.85, total => 4.25 },
    { item => "Banana", price => 0.55, total => 2.75 },
    ...
)
```

- **数组** `@items`：这是一个标准的 Perl 数组，其中每个元素都是一个哈希引用。
- **哈希引用**：每个哈希引用代表一个商品记录，包含三个键：`item`、`price` 和 `total`。`item` 键的值是商品的名称（一个字符串），`price` 键的值是商品的单价（一个浮点数），`total` 键的值是商品的总价（一个浮点数）。

这种数据结构常用于存储复杂的数据集，例如数据库查询的结果或 CSV 文件的内容。通过组合数组和哈希，你可以创建非常灵活和强大的数据结构来满足各种需要。

关于 printf 的格式：

以下是一些常见的 `printf` 格式占位符：

- `%d`：表示整数
- `%f`：表示浮点数
- `%s`：表示字符串
- `%.2f`：表示保留两位小数的浮点数
- `%x`：表示十六进制数
- `%o`：表示八进制数

第七节. Perl 的强大利器--正则表达式（入门）

正则表达式是 Perl 的核心功能之一，Perl 被广泛认为是文本处理和模式匹配的王者。Perl 的正则表达式功能既强大又灵活，使得文本搜索、提取和替换变得非常简单。

1. 基础匹配

使用 `=~` 操作符与 `m//` 进行匹配。如果省略 `m`，可以只使用 `//`：

```
my $string = "Hello, World!";
if ($string =~ /World/) {
    print "Matched!\n";  # 会输出这一行
}
```

2. 字符集

- `[...]`：匹配方括号内的任何字符
- `[^...]`：匹配不在方括号内的任何字符

```
if ("apple" =~ /[aeiou]/) {
    print "Has a vowel!\n";  # 会输出这一行
}
```

3. 量词

- `*`：匹配前一个字符0次或多次
- `+`：匹配前一个字符1次或多次
- `?`：匹配前一个字符0次或1次
- `{n}`：匹配前一个字符恰好n次
- `{n,}`：匹配前一个字符至少n次
- `{n,m}`：匹配前一个字符至少n次，但不超过m次

```
if ("1000" =~ /\d+/) {
    print "Has one or more digits!\n";  # 会输出这一行
}
```

4. 预定义字符集

- `\d`：匹配一个数字，等同于`[0-9]`
- `\D`：匹配一个非数字
- `\w`：匹配一个单词字符（字母、数字或下划线）
- `\W`：匹配一个非单词字符
- `\s`：匹配一个空白字符（如空格、制表符或换行符）
- `\S`：匹配一个非空白字符

5. 查找

使用 `=~` 进行匹配，并用括号捕获匹配的内容：

```
my $info = "Name: Alice Age: 30";
if ($info =~ /Name: (\w+) Age: (\d+)/) {
    print "Name: $1, Age: $2\n";  # 输出 "Name: Alice, Age: 30"
}
```

6. 替换

使用 `s///` 进行替换：

```
my $text = "cats and dogs";
$text =~ s/cats/kittens/;
print $text;  # 输出 "kittens and dogs"
```

你也可以使用 `g` 标志进行全局替换：

```
$text =~ s/[aeiou]/X/g;
print $text;  # 输出 "kXttXns Xnd dXgs"
```

7. 非贪婪匹配

默认情况下，正则表达式的匹配是"贪婪"的，意味着它们会尽可能多地匹配。使用 `?` 使匹配变得"非贪婪"：

```
my $text = "The quick <b>brown</b> fox <i>jumps</i> over";
if ($text =~ /<.*?>/g) {
    print "Matched: $&\n";  # 输出 "<b>" 和 "<i>"
}
```

8. 模式修饰符

- `i`：使匹配不区分大小写
- `g`：全局匹配
- `m`：多行模式，允许 `^` 和 `$` 匹配内部行
- `s`：单行模式，使 `.` 匹配包括换行符在内的所有字符
- `x`：扩展模式，允许在模式中使用空白和注释

```
my $mix = "Hello WORLD";
if ($mix =~ /world/i) {  # 不区分大小写
    print "Matched!\n";
}
```

9. 一个综合性例子，展示 Perl 正则表达式用法

场景描述：

你有一个日志文件，其中记录了网站的访问记录。每条记录都按以下格式存储：

```
[YYYY-MM-DD HH:MM:SS] IP_ADDRESS - REQUEST_METHOD ENDPOINT STATUS_CODE
```

例如:

```
[2023-08-23 10:45:12] 192.168.1.1 - GET /home 200
[2023-08-23 10:46:15] 192.168.1.2 - POST /login 401
[2023-08-23 10:47:17] 192.168.1.3 - GET /dashboard 404
```

你的任务是解析这个日志，提取所有返回 404 状态码的记录，并打印相关的 IP 地址和请求端点。

Perl 代码示例：

```
#!/usr/bin/perl
use strict;
use warnings;

my $log = <<'END';
[2023-08-23 10:45:12] 192.168.1.1 - GET /home 200
[2023-08-23 10:46:15] 192.168.1.2 - POST /login 401
[2023-08-23 10:47:17] 192.168.1.3 - GET /dashboard 404
[2023-08-23 10:48:19] 192.168.1.4 - GET /profile 200
[2023-08-23 10:49:20] 192.168.1.5 - PUT /update 404
END

while ($log =~ /\[(.*?)\] (\d+\.\d+\.\d+\.\d+) - (\w+) (\/\w+) (\d+)/g) {
    my ($timestamp, $ip, $method, $endpoint, $status) = ($1, $2, $3, $4, $5);

    if ($status == 404) {
        print "Error 404 detected from IP $ip on endpoint $endpoint at $timestamp\n";
    }
}
```

输出：

```
Error 404 detected from IP 192.168.1.3 on endpoint /dashboard at 2023-08-23 10:47:17
Error 404 detected from IP 192.168.1.5 on endpoint /update at 2023-08-23 10:49:20
```

在此示例中，我们使用了 Perl 的正则表达式来解析日志记录。每条记录都被捕获为五个组：时间戳、IP 地址、请求方法、端点和状态码。然后，我们检查状态码是否为 404，并相应地打印消息。

第八节. 实践：应用于 IC 设计中的两个 Perl 脚本（片段）

1. 从 spice 网表提取指定模块的端口，并生成默认激励

```
#!/usr/bin/perl
use strict;
use warnings;

# 内置的SPICE网表
my $spice_content = <<'END_SPICE';
.subckt latch_4bit_cb d<3> d<2> d<1>
+ d<0> le q<3> q<2> q<1>
+ q<0> rn vh vl
xi0 q<0> vh vl d<0> rn cb dlatchrb_hvt
xi1 q<1> vh vl d<1> rn cb dlatchrb_hvt
xi2 q<2> vh vl d<2> rn cb dlatchrb_hvt
xi3 q<3> vh vl d<3> rn cb dlatchrb_hvt
.ends latch_4bit_cb

.subckt or2_mv y vdd vss a b
xi1 y vdd vss net13 inv_x2_mv
xi0 net13 vdd vss a b nor2_mv
.ends or2_mv
```

```
.subckt inv_x2_mv y vdd vss a
m0 y a vss vss nmos_6v w=2e-6 l=600e-9
m1 y a vdd vdd pmos_6v w=5e-6 l=600e-9
.ends inv_x2
END_SPICE

my $sim_cell = "latch_4bit_cb"; # 指定的子电路

&main();
sub main {
    my $port_list_ptr = gen_port_list_spice();
    my @port_list_xinst = @{$port_list_ptr};
 print "\n$sim_cell port list:\n";
    print_array(@port_list_xinst);

 print "\nVoltage Sources:\n";
    generate_port_stims(@port_list_xinst);
}

sub gen_port_list_spice {
    # 开始解析SPICE网表
    my $capture = 0; # 用于判断是否在目标子电路中
    my @ports; # 用于存储端口
    my $port_string = ""; # 用于存储可能跨多行的端口字符串

    for my $row (split("\n", $spice_content)) {
        chomp $row;

        # 检查是否是目标子电路开始的行
        if ($row =~ /\.subckt $sim_cell\s+(.*)/) {
            $capture = 1;
            $port_string .= $1;
        }
        # 如果是在目标子电路中，并且行以+开始，则继续添加端口
        elsif ($capture and $row =~ /^\+\s*(.*)/) {
            $port_string .= " " . $1;
        }
        # 检查是否是子电路结束的行
        elsif ($row =~ /\.ends $sim_cell/) {
            $capture = 0;
            @ports = split(/\s+/, $port_string);
            last; # 结束循环
        }
    }

    return \@ports;
}


sub generate_port_stims {
    my @ports = @_;
    for my $port (@ports) {
        my $source = sprintf("V_%-20s%-20s%-20s", $port, $port, "0  dc=dvdd");
        print "$source\n";
    }
}
```

```
sub print_array {
    my @list = @_;
    for (@list) {
        print "$_\n";
    }
}
```

2. 详细分析 dspf 格式后仿真网表的耦合电容情况

dspf 格式的网表，除了可以直接用于 hspice 仿真以外，它还具有完整的寄生信息，这个信息可以用于对寄生的反标分析。不过可惜的是，对于1个节点来说，它的寄生电容是分散在网表中的，只有一部分电容紧挨着这个节点的定义，这对于分析来说很不方便。

这个脚本，对 dspf 网表进行了二次处理，使得每个节点的所有相关的寄生电容，都与该节点的定义紧挨着。而且，还可以指定节点分析其电容之和（还可以定义排除选项），也就是说，符合某些条件的电容被排除掉（例如SX之间的电容，因为全驱抵消了，因此不需要分析）。

```
#!/usr/bin/perl
use strict;
use warnings;

# 示例网表内容
my $dspf_netlist = <<'END';
*|NET AA 7e-15
cc_1 AA BB 2e-15
cc_3 AA CC 1e-15
cc_2 AA
+DD 4e-15

*|NET BB 7e-15
cc_23 BB CC 1e-15
cc_24 BB DD
+4e-15
cc_23
+BB CC 1e-15

*|NET CC 9e-15
cc_25
+CC DD 4e-15
cc_26 CC
+EE 3e-15

*|NET xA/xB/EE<0> 12e-15
cc_27 xA/xB/EE<0> EE<1> 4e-15
cc_28 xA/xB/EE<0> F 4e-15
cc_29 xA/xB/EE<0> EE<2> 3e-15
cc_30 xA/xB/EE<0> G 4e-15

m_25 CC DD 4e-15
x_26 CC
+EE 3e-15
END

# 定义用于存储网络名和其对应的电容值的哈希
my %nets;
```

```
# 定义用于存储每个节点及其关联的电容的哈希
my %capacitors;

# 处理断行连接的内容，将换行符后的'+'替换为空格，将多行内容合并为一行
$dspf_netlist =~ s/\n\+/ /g;

# 遍历每一行来解析数据
for my $line (split /\n/, $dspf_netlist) {
    # 匹配网络定义行，提取网络名和对应的电容值
    if ($line =~ /\*\|NET (\S+)\s+(\S+)/) {
        my ($net, $value) = ($1, $2);
        $nets{$net} = $value;
    }
    # 匹配电容定义行，提取电容名、两个节点和电容值
    elsif ($line =~ /^(cc_\w+)\s+(\S+)\s+(\S+)\s+(\S+)/) {
        my ($cap_name, $node1, $node2, $value) = ($1, $2, $3, $4);
        $value =~ s/f/e-15/; # 将 'f' 替换为 'e-15'
        # 将电容信息存储到两个节点的数组中
        push @{$capacitors{$node1}}, { name => $cap_name, node1 => $node1, node2 => $node2,
value => $value };
        push @{$capacitors{$node2}}, { name => $cap_name, node1 => $node2, node2 => $node1,
value => $value };
    }
}

# 按要求的格式打印输出
# 按电容值从大到小的顺序排序网络
for my $net (sort { $nets{$b} <=> $nets{$a} } keys %nets) {
    print "*|NET $net $nets{$net}\n";
    if (exists $capacitors{$net}) {
        # 按电容值从大到小排序电容
        my @sorted_capacitors = sort { $b->{value} <=> $a->{value} } @{$capacitors{$net}};
        my %printed_capacitors; # 用于跟踪已打印的电容
        for my $cap (@sorted_capacitors) {
            # 跳过已经打印的电容
            next if $printed_capacitors{$cap->{name}};
            print "$cap->{name} $cap->{node1} $cap->{node2} $cap->{value}\n";
            $printed_capacitors{$cap->{name}} = 1; # 标记电容为已打印
        }
    }
    print "\n";
}

# 指定查询的网络和排除模式
my @nets_to_query=qw(AA CC xA/xB/EE<0>);
my $exclude_pattern="EE*";

# 根据指定的网络和排除模式计算电容总和
for my $net (@nets_to_query) {
    my $total_capacitance = 0;
    if (exists $capacitors{$net}) {
        for my $cap (@{$capacitors{$net}}) {
            # 只排除匹配指定模式的node2
            next if defined $exclude_pattern && $cap->{node2} =~ /$exclude_pattern/;
            $total_capacitance += $cap->{value};
        }
    }
    $total_capacitance =~ s/e-15/f/; # 将 'e-15' 替换为 'f'
```

```
    print "$net => $total_capacitance\n";
}
```