



دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

گزارش پروژه

درس زبان‌های برنامه نویسی

## بررسی زبان R

سید محمدحسین هاشمی  
بردیا جوادی

استاد : دکتر آرش شفیعی

پاییز ۱۴۰۳

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

# فهرست مطالب

۹	مقدمه	۱
۹	تاریخچه زبان R	۱.۱
۹	ویژگی‌های زبان R	۲.۱
۱۰	هدف از ایجاد زبان و مشکلاتی که رفع کرد	۳.۱
۱۰	مشکلاتی که در ابتدای ظهور خود رفع می‌کرد	۴.۱
۱۰	بهبود نسبت به زبان‌های دیگر	۵.۱
۱۱	ویژگی‌های متمایز زبان	۶.۱
۱۱	ارزیابی بر اساس معیارهای مختلف	۷.۱
۱۲	مقایسه با زبان‌های مشابه	۸.۱
۱۲	خوانایی	۱.۸.۱
۱۲	سادگی	۲.۸.۱
۱۲	کتابخانه‌ها	۳.۸.۱
۱۲	کاربرد	۴.۸.۱
۱۳	کارایی	۵.۸.۱
۱۳	هزینه	۶.۸.۱
۱۳	قدرت آماری	۷.۸.۱
۱۳	قابلیت جابجایی	۸.۸.۱
۱۳	نتیجه‌گیری	۹.۸.۱
۱۴	نوع پیاده‌سازی	۹.۱
۱۴	مفسرها و کامپایلرهای موجود برای R	۱۰.۱
۱۶	نحو و معنا شناسی	۲
۱۶	نحو و معنا شناسی	۱.۲
۱۶	if - اجرای کد بر اساس شرط مشخص	۱.۱.۲
۱۶	else - بلوک جایگزین در صورت عدم تحقق شرط	۲.۱.۲
۱۶	repeat - اجرای حلقه بی‌نهایت تا زمان استفاده از break	۳.۱.۲
۱۷	while - اجرای حلقه تا زمانی که شرط برقرار باشد	۴.۱.۲
۱۷	function - تعریف توابع جدید	۵.۱.۲
۱۷	for - اجرای حلقه روی مجموعه‌ای از مقادیر	۶.۱.۲
۱۷	in - تعیین عضویت یا استفاده در حلقه	۷.۱.۲
۱۸	next - عبور از مرحله جاری و رفتن به مرحله بعدی حلقه	۸.۱.۲
۱۸	break - خروج از حلقه	۹.۱.۲
۱۸	TRUE و FALSE - مقادیر منطقی (Boolean) در R	۱۰.۱.۲

۱۸	.....	۱۱.۱.۲	NULL - نشان‌دهنده مقدار خالی یا بدون مقدار
۱۸	.....	۱۲.۱.۲	NA - مقدار غیرموجود یا ناشناخته (Not Available)
۱۹	.....	۱۳.۱.۲	Inf و -Inf - مقدار بی‌نهایت مثبت و منفی
۱۹	.....	۱۴.۱.۲	NaN - عدد غیرقابل تعریف (Not a Number)
	.....	۱۵.۱.۲	NA_character_، NA_complex_، NA_real_، NA_integer_ و NA_character_
۱۹	.....		- انواع خاص NA برای داده‌های مختلف
۱۹	.....	۲.۲	گرامر ساده برای بخشی از زبان
۲۰	.....	۳.۲	برنامه نمونه در زبان
۲۰	.....	۴.۲	درخت تجزیه برای برنامه نمونه
۲۰	.....	۵.۲	تقدم و وابستگی عملگرها
۲۱	.....	۶.۲	توصیف گرامر بر اساس تقدم عملگرها
۲۱	.....	۷.۲	معناشناسی عملیاتی و توصیف ساختارها
۲۱	.....	۱.۷.۲	تخصیص (Assignment)
۲۲	.....	۲.۷.۲	حلقه for
۲۲	.....	۳.۷.۲	چاپ (Print)
۲۴		۳	متغیرها و نوع‌های داده‌ای
۲۴	.....	۱.۳	انقیاد
۲۴	.....	۱.۱.۳	انقیاد نوع و مقدار
۲۴	.....	۲.۱.۳	انقیاد مقدار (Value Binding)
۲۵	.....	۳.۱.۳	تعریف متغیرها
۲۵	.....	۴.۱.۳	تاثیر پویا بودن انقیاد نوع و مقدار
۲۵	.....	۵.۱.۳	کاربرد و مزایا
۲۶	.....	۶.۱.۳	نتیجه‌گیری
۲۶	.....	۲.۳	بررسی انواع متغیرها
۲۶	.....	۱.۲.۳	متغیرهای ایستا (Static Variables)
۲۶	.....	۲.۲.۳	متغیرهای پویا در پشت (Dynamic Stack Variables)
۲۷	.....	۳.۲.۳	متغیرهای پویا در هیپ به طور صریح
۲۷	.....	۴.۲.۳	متغیرهای پویا در هیپ به طور ضمنی
۲۸	.....	۵.۲.۳	مقایسه انواع متغیر
۲۸	.....	۶.۲.۳	نتیجه‌گیری
۲۸	.....	۳.۳	حوزه‌های تعریف زبان
۲۸	.....	۱.۳.۳	حوزه تعریف ایستا (Static Scope)
۲۹	.....	۲.۳.۳	حوزه تعریف پویا (Dynamic Scope)
۲۹	.....	۳.۳.۳	افزودن حوزه پویا به زبان R
۳۰	.....	۴.۳.۳	پشتیبانی از هر دو نوع حوزه در R
۳۰	.....	۵.۳.۳	بلوک‌ها و کلمات کلیدی حوزه در R
۳۰	.....	۶.۳.۳	نتیجه‌گیری
۳۱	.....	۴.۳	نوع‌های داده‌ای
۳۱	.....	۱.۴.۳	نوع‌های داده‌ای پایه (Basic Data Types)
۳۳	.....	۲.۴.۳	ساختارهای داده‌ای پیچیده (Advanced Data Structures)
۳۴	.....	۳.۴.۳	اشاره‌گرها و متغیرهای مرجع
۳۴	.....	۴.۴.۳	بازیافت حافظه (Garbage Collection)

۳۴	رفع مشکلات ناشی حافظه و اشاره گر معلق	۵.۴.۳
۳۵	نتیجه گیری	۶.۴.۳
۳۶	برنامه نویسی تابعی	۴
۳۶	سازوکارهای برنامه نویسی تابعی در R	۱.۴
۳۷	پیاپی سازی توابع برنامه نویسی تابعی در R	۲.۴
۳۷	تأثیر توابع نگاشت، فیلتر و کاهش بر کارایی برنامه	۳.۴
۳۸	مقایسه با برنامه نویسی رویه ای (Procedural Programming)	۴.۴
۳۸	جمع بندی	۵.۴
۳۹	برنامه نویسی رویه ای	۵
۳۹	زیربرنامه ها (Subroutines)	۱.۵
۳۹	روش های ارسال متغیرها به توابع	۲.۵
۴۰	برنامه نویسی عمومی (Generic Programming)	۳.۵
۴۱	سایر ویژگی های برنامه نویسی رویه ای	۴.۵
۴۱	مقایسه برنامه نویسی رویه ای با تابعی	۵.۵
۴۲	جمع بندی	۶.۵
۴۳	برنامه نویسی شیء گرا	۶
۴۳	ساختارهای موجود برای برنامه نویسی شیء گرا	۱.۶
۴۵	چندریختی (Polymorphism)	۲.۶
۴۶	وراثت (Inheritance)	۳.۶
۴۶	مدیریت اشیاء در حافظه	۴.۶
۴۶	جمع بندی	۵.۶
۴۷	برنامه نویسی همروند	۷
۴۷	رشته ها (Threads)	۱.۷
۴۷	قفل ها (Locks) و سمافورها (Semaphores)	۲.۷
۴۸	ارسال پیام (Message Passing)	۳.۷
۴۸	مدیریت همروندی با بسته های خاص	۴.۷
۴۹	مقایسه سمافورها و قفل ها در R	۵.۷
۴۹	سایر سازوکارهای برنامه نویسی همروند	۶.۷
۴۹	مثالی کامل از برنامه نویسی همروند در R	۷.۷
۵۰	جمع بندی	۸.۷
۵۱	پیاپی سازی چند الگوریتم معروف با R	۸
۵۱	جستجوی دودویی (Binary Search)	۱.۸
۵۲	مرتب سازی حبابی (Bubble Sort)	۲.۸
۵۳	مرتب سازی سریع (Quick Sort)	۳.۸
۵۳	الگوریتم دایکسترا (Dijkstra's Algorithm)	۴.۸
۵۴	مرتب سازی ادغامی (Merge Sort)	۵.۸
۵۵	مسئله کوله پشتی (0/1 Knapsack Problem)	۶.۸

## فهرست جداول

۱۵	.....	۱۰۱	مقایسه پیاده‌سازی‌های مختلف زبان R
۲۸	.....	۱۰۳	مقایسه انواع تخصیص و سرعت آن‌ها

## فهرست کدها

۱۶	if	۱.۲
۱۶	else	۲.۲
۱۶	repeat	۳.۲
۱۷	while	۴.۲
۱۷	function	۵.۲
۱۷	for	۶.۲
۱۷	in	۷.۲
۱۸	next	۸.۲
۱۸	break	۹.۲
۱۸	TRUE, FALSE	۱۰.۲
۱۸	NULL	۱۱.۲
۱۸	NA	۱۲.۲
۱۹	Inf, -Inf	۱۳.۲
۱۹	NaN	۱۴.۲
۱۹	NA_integer_, NA_real_, NA_complex_, NA_character_	۱۵.۲
۱۹	Simple R grammar	۱۶.۲
۲۰	Simple R program	۱۷.۲
۲۱	Precedence of operators	۱۸.۲
۲۱	Operation grammar	۱۹.۲
۲۱	R Assignment	۲۰.۲
۲۱	C Assignment	۲۱.۲
۲۲	Assembly Assignment	۲۲.۲
۲۲	R for	۲۳.۲
۲۲	C for	۲۴.۲
۲۲	Assembly loop	۲۵.۲
۲۲	R Print	۲۶.۲
۲۲	C Print	۲۷.۲
۲۴	Type Binding	۱.۳
۲۴	Value Binding	۲.۳
۲۵	Explicit Variable	۳.۳
۲۵	Implicit Variable	۴.۳
۲۵	Type error	۵.۳
۲۶	Static variables	۶.۳
۲۷	Dynamic stack variables	۷.۳

۲۷	Dynamic variables on heap explicitly	۸.۳
۲۸	Dynamic variables on heap implicitly	۹.۳
۲۹	Static scope	۱۰.۳
۲۹	Dynamic scope	۱۱.۳
۳۰	Adding a dynamic scope to R	۱۲.۳
۳۰	Blocks in R	۱۳.۳
۳۱	Logical data type	۱۴.۳
۳۱	Numeric data type	۱۵.۳
۳۲	Integer data type	۱۶.۳
۳۲	Character data type	۱۷.۳
۳۲	Complex data type	۱۸.۳
۳۳	Vector data type	۱۹.۳
۳۳	Matrix data type	۲۰.۳
۳۳	List data type	۲۱.۳
۳۴	Data Frame data type	۲۲.۳
۳۴	Array data type	۲۳.۳
۳۶	Lambda Functions	۱.۴
۳۶	Passing function to another function	۲.۴
۳۷	Return a function into function	۳.۴
۳۷	Map – Filter – Reduce	۴.۴
۳۸	Map – Filter – Reduce	۵.۴
۳۹	Subroutines	۱.۵
۳۹	Pass by Value	۲.۵
۴۰	Pass by Reference	۳.۵
۴۰	Generic Programming	۴.۵
۴۰	S3	۵.۵
۴۱	for	۶.۵
۴۱	Default Arguments	۷.۵
۴۱	Variable Scope	۸.۵
۴۳	S3 OOP	۱.۶
۴۴	S3 OOP	۲.۶
۴۴	Reference Classes	۳.۶
۴۵	S6 OOP	۴.۶
۴۵	Polymorphism	۵.۶
۴۶	Inheritance	۶.۶
۴۷	Multicore Processing	۱.۷
۴۸	Locks in parallel	۲.۷
۴۸	Message Passing in makeCluster	۳.۷
۴۸	future package	۴.۷
۴۹	foreach package	۵.۷
۴۹	Combination of concurrent execution and locks	۶.۷
۵۱	Binary Search	۱.۸
۵۲	Bubble Sort	۲.۸
۵۳	Quick Sort	۳.۸



۵۴	.....	Dijkstra's Algorithm	۴.۸
۵۵	.....	Merge Sort	۵.۸
۵۶	.....	0/1 Knapsack Problem	۶.۸

# فصل ۱

## مقدمه

### ۱.۱ تاریخچه زبان R

زبان برنامه‌نویسی R در ابتدا با هدف ساده‌سازی تحلیل‌های آماری و مدل‌سازی داده‌ها طراحی شد. این زبان که توسط راس ایهاکا و رابرت جنتلمن در دهه ۱۹۹۰ ابداع شد، به عنوان یک ابزار متن‌باز برای تحلیل داده‌های پیچیده توسعه یافت. در آن زمان، نیاز به زبانی که توانایی تحلیل آماری پیشرفته، مدل‌سازی ریاضی و تولید گراف‌های بصری را داشته باشد، به شدت احساس می‌شد. زبان R با الهام از زبان SAS طراحی شد و توانست مشکلاتی نظیر عدم انعطاف‌پذیری ابزارهای آماری موجود و محدودیت‌های گرافیکی آنها را برطرف کند. همچنین، متن‌باز بودن R موجب شد که جامعه‌ای پویا از کاربران و توسعه‌دهندگان حول آن شکل بگیرد، که این امر به گسترش سریع امکانات و کتابخانه‌های آن کمک کرد.

### ۲.۱ ویژگی‌های زبان R

R به دلیل قابلیت‌های منحصر به فرد خود در حوزه‌های مختلف کاربرد گسترده‌ای دارد. از جمله مهم‌ترین حوزه‌ها می‌توان به موارد زیر اشاره کرد:

- تحلیل داده‌ها و آمار پیشرفته: به‌طور خاص برای تحلیل‌های آماری پیچیده و مدل‌سازی داده‌ها طراحی شده است.
- یادگیری ماشین و هوش مصنوعی: بسیاری از الگوریتم‌های یادگیری ماشین و ابزارهای هوش مصنوعی در پیاده‌سازی شده‌اند.
- مصورسازی داده‌ها: قابلیت‌های گرافیکی پیشرفته آن را به ابزاری مناسب برای ایجاد نمودارهای حرفه‌ای و گزارش‌های بصری تبدیل کرده است.
- بیوانفورماتیک: در تحلیل داده‌های زیستی، از جمله داده‌های ژنومی و پروتئومی، کاربرد فراوان دارد.
- امور مالی و اقتصادی: بسیاری از مدل‌های مالی و پیش‌بینی‌های اقتصادی با استفاده از R پیاده‌سازی می‌شوند.

- تحقیقات علمی: به عنوان یک ابزار تحقیقاتی در علوم اجتماعی، روانشناسی و بسیاری از رشته‌های علمی مورد استفاده قرار می‌گیرد. علاوه بر این، R به دلیل پشتیبانی گسترده از کتابخانه‌های تخصصی، امکان تحلیل‌های پیشرفته در حوزه‌های خاص را فراهم می‌کند.
- بازاریابی و تحلیل مشتری: کسب‌وکارها از برای تحلیل رفتار مشتری، تقسیم‌بندی بازار و پیش‌بینی فروش استفاده می‌کنند.

### ۳.۱ هدف از ایجاد زبان و مشکلاتی که رفع کرد

R برای حل مشکلات خاصی طراحی شد که در آن زمان در زبان‌های موجود دیده می‌شد:

- رایگان و منبع‌باز بودن: در دهه ۹۰، بسیاری از ابزارهای آماری مثل SAS و MATLAB هزینه‌های بالایی داشتند. R به عنوان یک جایگزین رایگان و منبع‌باز برای این ابزارها توسعه یافت.
- سازگاری با زبان SAS: کاربران زبان SAS می‌توانستند کدهای خود را با تغییرات جزئی در R اجرا کنند. این ویژگی، پذیرش R را تسریع کرد.
- انعطاف‌پذیری بالا: R به طور خاص برای تحلیل آماری و مصورسازی داده‌ها بهینه‌سازی شده بود و نیازهای تحلیل‌گران را بهتر از زبان‌های عمومی مثل C یا Java برآورده می‌کرد.
- قابلیت توسعه‌پذیری: با ایجاد کتابخانه‌ها و بسته‌های متنوع، کاربران می‌توانستند قابلیت‌های R را برای نیازهای خاص خود توسعه دهند.

### ۴.۱ مشکلاتی که در ابتدای ظهور خود رفع می‌کرد

- عدم دسترسی به ابزارهای پیشرفته تحلیل داده: با ارائه ابزارهایی قدرتمند، نیاز به نرم‌افزارهای گران‌قیمت را کاهش داد.
- مصورسازی ضعیف داده‌ها: در آن زمان، بسیاری از زبان‌های برنامه‌نویسی، ابزارهای گرافیکی مناسبی نداشتند. R با کتابخانه‌هایی مثل ggplot2، انقلابی در مصورسازی داده‌ها ایجاد کرد.
- پیچیدگی کدنویسی در زبان‌های عمومی: تحلیل‌گران داده معمولاً نیاز داشتند تا با زبان‌هایی مثل Fortran یا C کار کنند که برای تحلیل داده بهینه نبودند. R این مشکل را رفع کرد و کار با داده‌ها را ساده‌تر نمود.

### ۵.۱ بهبود نسبت به زبان‌های دیگر

R بسیاری از قابلیت‌ها و ویژگی‌های موجود در زبان‌های دیگر را ارتقا داد و درون خود تجمیع کرد:

- مقایسه با SAS: رایگان و متن‌باز بود، در حالی که SAS یک نرم‌افزار تجاری بود.
- مقایسه با MATLAB: در تحلیل‌های آماری پیچیده کارآمدتر بود و به جامعه کاربری بزرگی در حوزه آمار متکی بود.

- مقایسه با Python (در زمان آغاز): Python در آن زمان بیشتر برای توسعه اسکریپت کاربرد داشت و قدرت تحلیل داده‌ای آن مثل امروز توسعه نیافته بود.

## ۶.۱ ویژگی‌های متمایز زبان

- تمرکز بر آمار: برخلاف زبان‌های عمومی مثل Python یا C++، R از ابتدا با هدف تحلیل آماری طراحی شده است.
- کتابخانه‌های تخصصی: R دارای بیش از ۱۸,۰۰۰ بسته تخصصی در حوزه‌های مختلف مانند ژنتیک، مالی، یادگیری ماشین و غیره است.
- مصورسازی پیشرفته: R ابزارهای قدرتمندی برای ایجاد نمودارهای پیچیده و تعاملی دارد که در مقایسه با زبان‌های مشابه، برجسته‌تر است.
- جامعه کاربری بزرگ: جامعه R بسیار فعال است و منابع یادگیری رایگان بسیاری فراهم کرده است.
- قابلیت‌های تعاملی: با ابزارهایی مثل RStudio و Shiny، R امکان ایجاد داشبوردها و برنامه‌های وب تعاملی را فراهم می‌کند.

## ۷.۱ ارزیابی بر اساس معیارهای مختلف

- خوانایی: برای کاربران با پس‌زمینه آمار و علوم داده خواناتر است. با این حال، نحو خاص آن ممکن است برای برنامه‌نویسان سنتی دشوار باشد.
- قابلیت اطمینان: در تحلیل‌های آماری و شبیه‌سازی‌ها بسیار قابل‌اعتماد است. با این حال، برای پروژه‌های بزرگ و داده‌های حجیم ممکن است نیاز به بهینه‌سازی داشته باشد.
- هزینه: رایگان و متن‌باز است، که آن را به انتخابی ایده‌آل برای دانشجویان و شرکت‌های نوپا تبدیل می‌کند.
- بهره‌وری و کارایی: برای تحلیل‌های آماری کوچک و متوسط، R بسیار سریع و کارآمد است. اما در مقایسه با زبان‌هایی مثل Python برای پروژه‌های چندمنظوره یا داده‌های حجیم ممکن است کارایی کمتری داشته باشد.
- هزینه یادگیری: برای کسانی که با آمار آشنایی دارند، آسان‌تر است. اما برای برنامه‌نویسان تازه‌کار، زبان‌هایی مثل Python ساده‌تر هستند.
- قابلیت جابجایی: بر روی تمامی سیستم‌عامل‌های رایج (ویندوز، مک، لینوکس) بدون مشکل اجرا می‌شود.
- ابزارهای توسعه: ابزارهایی مانند RStudio و Jupyter Notebook پشتیبانی قدرتمندی برای توسعه و تحلیل داده ارائه شده‌اند.

## ۸.۱ مقایسه با زبان‌های مشابه

### ۱.۸.۱ خوانایی

- R: طراحی شده با تمرکز بر آمار و تحلیل داده. کدهای R ممکن است برای تازه‌کاران کمی دشوار به نظر برسند، اما برای کسانی که در تحلیل آماری تجربه دارند، خوانا و ساده است.
- Python: خوانایی بالاتر به دلیل سینتکس ساده و عمومی. یادگیری و استفاده از آن برای افراد تازه‌کار آسان‌تر است.
- MATLAB: خوانا و ساده، مخصوصاً برای مهندسان. ساختار نوشتار آن شبیه به ریاضیات است.
- SAS: نسبتاً سخت‌تر به دلیل سینتکس خاص و قدیمی.

### ۲.۸.۱ سادگی

- R: برای تحلیل داده ساده و بهینه است، اما سینتکس آن برای کارهای غیرتحلیلی ممکن است پیچیده باشد.
- Python: ساده‌تر به دلیل طراحی چندمنظوره و استفاده گسترده در زمینه‌های مختلف.
- MATLAB: ساده اما گران است، زیرا نیاز به لایسنس دارد.
- SAS: پیچیده‌تر، به خصوص برای کاربران غیرمتخصص.

### ۳.۸.۱ کتابخانه‌ها

- R: غنی‌ترین مجموعه کتابخانه‌های آماری و تحلیل داده را دارد (Bioconductor، CRAN).
- Python: دارای کتابخانه‌های عمومی قوی مانند NumPy، Pandas، و scikit-learn است. اما برای تحلیل‌های پیشرفته، به R نمی‌رسد.
- MATLAB: کتابخانه‌های ریاضی و شبیه‌سازی قوی دارد، اما برای تحلیل آماری گسترده نیست.
- SAS: ابزارهای تخصصی برای تحلیل داده دارد، اما کتابخانه‌های خارجی محدودند.

### ۴.۸.۱ کاربرد

- R: تحلیل داده، مدل‌سازی آماری، یادگیری ماشین.
- Python: همه‌منظوره، از تحلیل داده تا هوش مصنوعی و توسعه وب.
- MATLAB: محاسبات عددی، شبیه‌سازی و پردازش سیگنال.
- SAS: تحلیل داده‌های صنعتی و کسب‌وکاری.

## ۵.۸.۱ کارایی

- R: برای مجموعه داده‌های بزرگ ممکن است کند باشد مگر با استفاده از بسته‌های بهینه‌سازی شده.
- Python: سریع‌تر با کتابخانه‌هایی مانند NumPy و استفاده از Cython.
- MATLAB: کارایی بالا برای مسائل ریاضی و شبیه‌سازی.
- SAS: کارایی بالا در تحلیل داده‌های سازمانی.

## ۶.۸.۱ هزینه

- R: رایگان و متن‌باز.
- Python: رایگان و متن‌باز.
- MATLAB: بسیار گران و نیازمند لایسنس.
- SAS: گران‌قیمت و مناسب شرکت‌ها.

## ۷.۸.۱ قدرت آماری

- R: بالاترین قدرت آماری، با گسترده‌ترین ابزارها و مدل‌ها.
- Python: قدرت آماری متوسط. کتابخانه‌های آن برای تحلیل داده کافی هستند، اما به گستردگی R نیستند.
- MATLAB: قدرت آماری کمتر نسبت به R و Python.
- SAS: قدرت آماری بالا، اما بیشتر در حوزه‌های سازمانی کاربرد دارد.

## ۸.۸.۱ قابلیت جابجایی

- R: قابل نصب روی تمام سیستم‌عامل‌ها (Linux، macOS، Windows).
- Python: بسیار قابل جابجایی با پشتیبانی جهانی.
- MATLAB: نیاز به لایسنس برای هر سیستم‌عامل.
- SAS: معمولاً به صورت سروری استفاده می‌شود و به راحتی قابل انتقال نیست.

## ۹.۸.۱ نتیجه‌گیری

- اگر قدرت آماری و تحلیل داده اولویت است: R بهترین گزینه است.
- اگر نیاز به یک زبان همه‌منظوره با کاربری گسترده دارید: Python انتخاب بهتری است.
- اگر نیازمند به زبانی برای شبیه‌سازی هستید: MATLAB برتری دارد.
- اگر تحلیل‌های کسب‌وکاری سازمانی مدنظر است: SAS گزینه مناسبی است.

## ۹.۱ نوع پیاده‌سازی

- تفسیرشده: R به طور اصلی یک زبان تفسیرشده است، به این معنی که کدها به صورت خط به خط توسط مفسر اجرا می‌شوند. این قابلیت به تحلیل‌گران اجازه می‌دهد تا کدها را به صورت تعاملی اجرا کنند و سریعاً نتایج را ببینند.
- کامپایل جزئی (Just-in-Time Compilation): از نسخه ۱۳.۲ به بعد، R از کامپایلر JIT<sup>۱</sup> بهره می‌برد که با استفاده از بسته compiler ارائه شده است. JIT بهینه‌سازی‌هایی برای اجرای سریع‌تر کدها فراهم می‌کند.
- پیاده‌سازی ترکیبی: اگرچه R عمدتاً تفسیرشده است، برای برخی از عملیات‌های پیچیده‌تر، از کدهای C و Fortran استفاده می‌شود که کامپایل شده‌اند تا کارایی بیشتری داشته باشند.

## ۱۰.۱ مفسرها و کامپایلرهای موجود برای R

### ۱. GNU R (پایه):

- توسعه‌دهنده: تیم توسعه R که یک گروه جهانی از متخصصان و دانشمندان داده است.
- ویژگی‌ها:
  - مفسر اصلی زبان R.
  - پیاده‌سازی به زبان C و Fortran.
  - استفاده از بسته compiler برای کامپایل جزئی.
- مزایا:
  - منبع باز و رایگان.
  - پایدار و استاندارد، با پشتیبانی وسیع از جامعه کاربری.

### ۲. FastR:

- توسعه‌دهنده: تیم Oracle Labs به عنوان بخشی از پروژه GraalVM.
- ویژگی‌ها:
  - یک پیاده‌سازی جایگزین برای R با هدف افزایش سرعت اجرا.
  - از JIT برای اجرای سریع‌تر کدها بهره می‌برد.
- مزایا:
  - بهینه‌سازی برای کاربردهای داده‌های حجیم و عملکرد سریع‌تر نسبت به GNU R.
  - سازگاری با ابزارهای GraalVM.

### ۳. Renjin:

- توسعه‌دهنده: BeDataDriven، یک شرکت هلندی.
- ویژگی‌ها:
  - پیاده‌سازی R در JVM<sup>۲</sup>.

<sup>۱</sup>Just-in-Time

<sup>۲</sup>Java Virtual Machine

- مناسب برای ادغام با سیستم‌های مبتنی بر جاوا.
- مزایا:

- کارایی بالا در محیط‌های سازمانی جاوا.
- امکان ادغام مستقیم با زیرساخت‌های جاوا.

## ۴. pqR:

- توسعه‌دهنده: Radford Neal، یک آماردان برجسته.
- ویژگی‌ها:
- یک نسخه بهینه‌شده از GNU R با تمرکز بر اجرای سریع‌تر.
- بهره‌گیری از پردازش موازی برای بهبود کارایی.
- مزایا:

- سرعت بالاتر برای تحلیل داده‌ها.
- مناسب برای کاربران حرفه‌ای تر.

## ۵. Microsoft R Open (MRO):

- توسعه‌دهنده: میکروسافت (Microsoft).
- ویژگی‌ها:
- نسخه بهینه‌شده R با بهبود عملکرد.
- شامل بسته‌های از پیش کامپایل‌شده و بهینه‌سازی شده برای تحلیل داده.
- مزایا:
- یکپارچگی با ابزارهای میکروسافت مانند Azure Machine Learning.
- سرعت بیشتر در پردازش داده‌های حجیم.

مزایا	معایب	پیاده‌سازی
استاندارد و قابل اعتماد، منبع‌باز، جامعه کاربری گسترده	سرعت پایین‌تر نسبت به نسخه‌های بهینه‌شده	GNU R
سرعت بیشتر، سازگاری با GraalVM	نیاز به پیکربندی پیشرفته	FastR
سازگاری با جاوا، مناسب برای محیط‌های سازمانی	محدودیت در پشتیبانی برخی از بسته‌های R	Renjin
پردازش موازی، کارایی بالا	جامعه کاربری کوچک‌تر	pqR
سازگاری با ابزارهای میکروسافت، سرعت بیشتر	وابستگی به اکوسیستم میکروسافت	Microsoft R Open

جدول ۱۰۱: مقایسه پیاده‌سازی‌های مختلف زبان R



## فصل ۲

# نحو و معنا شناسی

### ۱.۲ نحو و معناشناسی

کلمات کلیدی زبان R شامل موارد زیر است و نمی‌توان از آن‌ها به عنوان نام متغیر یا تابع استفاده کرد: NA, NULL, FALSE, TRUE, break, next, in, for, function, while, repeat, else, if, NA\_character\_ و NA\_complex\_, NA\_real\_, NA\_integer\_, NaN, Inf  
توضیح کلمات کلیدی:

#### ۱.۱.۲ if - اجرای کد بر اساس شرط مشخص

Listing 2.1: if

```
if (x > 0) {  
  print("is positive")  
}
```

#### ۲.۱.۲ else - بلوک جایگزین در صورت عدم تحقق شرط

Listing 2.2: else

```
if (x > 0) {  
  print("is positive")  
} else {  
  print("is negative or zero")  
}
```

#### ۳.۱.۲ repeat - اجرای حلقه بی‌نهایت تا زمان استفاده از break

Listing 2.3: repeat

```
i <- 1
repeat {
  print(i)
  if (i == 5) break
  i <- i + 1
}
```

۴.۱.۲ while - اجرای حلقه تا زمانی که شرط برقرار باشد

Listing 2.4: while

```
i <- 1
while (i <= 5) {
  print(i)
  i <- i + 1
}
```

۵.۱.۲ function - تعریف توابع جدید

Listing 2.5: function

```
my_function <- function(a, b) {
  return(a + b)
}
print(my_function(3, 4))
```

۶.۱.۲ for - اجرای حلقه روی مجموعه‌ای از مقادیر

Listing 2.6: for

```
for (i in 1:5) {
  print(i)
}
```

۷.۱.۲ in - تعیین عضویت یا استفاده در حلقه

Listing 2.7: in

```
x <- 5
print(x %in% c(3, 5, 7)) # output: TRUE
```

۸.۱.۲ next - عبور از مرحله جاری و رفتن به مرحله بعدی حلقه

Listing 2.8: next

```
for (i in 1:5) {  
  if (i == 3) next  
  print(i)  
}
```

۹.۱.۲ break - خروج از حلقه

Listing 2.9: break

```
for (i in 1:5) {  
  if (i == 3) break  
  print(i)  
}
```

۱۰.۱.۲ TRUE و FALSE - مقادیر منطقی (Boolean) در R

Listing 2.10: TRUE, FALSE

```
x <- TRUE  
y <- FALSE  
print(x & y) # output: FALSE
```

۱۱.۱.۲ NULL - نشان‌دهنده مقدار خالی یا بدون مقدار

Listing 2.11: NULL

```
x <- NULL  
print(is.null(x)) # output: TRUE
```

۱۲.۱.۲ NA - مقدار غیرموجود یا ناشناخته (Not Available)

Listing 2.12: NA

```
x <- c(1, NA, 3)  
print(is.na(x)) # output: FALSE TRUE FALSE
```

## ۱۳.۱.۲ Inf و -Inf - مقدار بی‌نهایت مثبت و منفی

Listing 2.13: Inf, -Inf

---

```
print(1 / 0) # output: Inf
print(-1 / 0) # output: -Inf
```

---

## ۱۴.۱.۲ NaN - عدد غیرقابل تعریف (Not a Number)

Listing 2.14: NaN

---

```
print(0 / 0) # output: NaN
```

---

## ۱۵.۱.۲ NA\_integer\_, NA\_real\_, NA\_complex\_ و NA\_character\_ - انواع خاص NA برای داده‌های مختلف

Listing 2.15: NA\_integer\_, NA\_real\_, NA\_complex\_, NA\_character\_

---

```
x <- NA_integer_
print(typeof(x)) # output: integer
```

---

## ۲.۲ گرامر ساده برای بخشی از زبان

گرامر زیر یک زیرمجموعه کوچک از R را برای ساختارهای کنترلی و عملیات ریاضی توصیف می‌کند:

Listing 2.16: Simple R grammar

---

```
<program> ::= <statement> | <statement> ";" <program>
<statement> ::= <assignment> | <conditional> | <loop> |
  <expression>
<assignment> ::= <identifier> "<->" <expression>
<conditional> ::= "if" "(" <expression> ")" "{" <program> "}" [
  "else" "{" <program> "}" ]
<loop> ::= "for" "(" <identifier> "in" <expression> ")" "{"
  <program> "}"
<expression> ::= <term> | <term> <operator> <expression>
<term> ::= <number> | <identifier> | "(" <expression> ")"
<operator> ::= "+" | "-" | "*" | "/"
<number> ::= [0-9]+
<identifier> ::= [a-zA-Z_] [a-zA-Z0-9_]*
```

---

## ۳.۲ برنامه نمونه در زبان

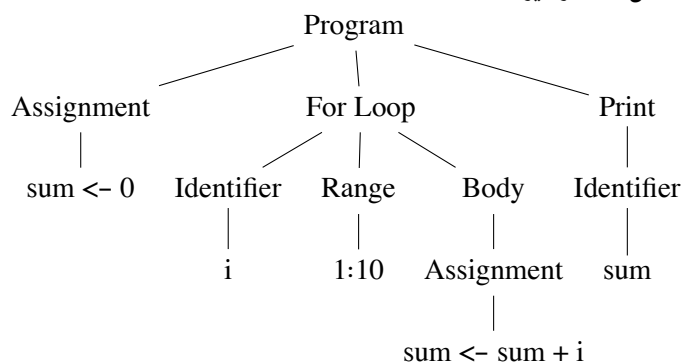
برنامه زیر یک حلقه برای محاسبه مجموع اعداد ۱ تا ۱۰ است:

Listing 2.17: Simple R program

```
sum <- 0
for (i in 1:10) {
  sum <- sum + i
}
print(sum)
```

## ۴.۲ درخت تجزیه برای برنامه نمونه

درخت تجزیه برنامه ۳.۲ شامل عناصر زیر است:



## ۵.۲ تقدم و وابستگی عملگرها

در R، تقدم عملگرها به صورت پیش فرض تعریف شده است. برخی از عملگرهای رایج با ترتیب تقدم:

۱. داخل پرانتز -  $()$ : بالاترین تقدم.
۲. توابع: توابع مانند  $\sin()$ ،  $\log()$  تقدم بیشتری از عملگرهای ریاضی دارند.
۳. عملگر توان: بیشترین تقدم میان عملگرهای ریاضی.
۴. ضرب و تقسیم -  $(*)$  و  $(/)$ : بالاتر از جمع و تفریق.
۵. جمع و تفریق -  $(+)$  و  $(-)$ : پایین تر از ضرب و تقسیم.

## وابستگی عملگرها:

- بیشتر عملگرها از چپ به راست ارزیابی می شوند (مثلاً  $+$ ،  $-$ ،  $*$ ،  $/$ ).

- عملگر توان از راست به چپ ارزیابی می‌شود.

مثال:

Listing 2.18: Precedence of operators

---

```
x <- 2 + 3 * 4^2
```

---

توضیح گام به گام:

۱. توان:  $4^2 = 16$ .

۲. ضرب:  $3 * 16 = 48$ .

۳. جمع:  $2 + 48 = 50$ .

## ۶.۲ توصیف گرامر بر اساس تقدم عملگرها

برای پیروی از تقدم عملگرها در گرامر، می‌توانیم گرامر را اصلاح کنیم:

Listing 2.19: Operation grammar

---

```
<expression> ::= <term> | <term> "+" <expression> | <term> "-"
  <expression>
<term>      ::= <factor> | <factor> "*" <term> | <factor> "/" <term>
<factor>    ::= <number> | "(" <expression> ")" | <factor> "^"
  <factor>
```

---

این گرامر تقدم توان را بالاتر از ضرب و تقسیم و تقدم ضرب و تقسیم را بالاتر از جمع و تفریق تضمین می‌کند.

## ۷.۲ معناشناسی عملیاتی و توصیف ساختارها

مقایسه برخی از ساختارهای پایه در زبان‌های برنامه نویسی در زبان‌های C، R، و اسمبلی:

### ۱.۷.۲ تخصیص (Assignment)

- کد R

Listing 2.20: R Assignment

---

```
sum <- 0
```

---

- معادل در C

Listing 2.21: C Assignment

---

```
int sum = 0;
```

---

- معادل در اسمبلی (فرض پردازنده x86)

Listing 2.22: Assembly Assignment

---

```
MOV sum, 0
```

---

۲.۷.۲ حلقه for

- کد R

Listing 2.23: R for

---

```
for (i in 1:10) {  
  sum <- sum + i  
}
```

---

- معادل در C

Listing 2.24: C for

---

```
for (int i = 1; i <= 10; i++) {  
  sum += i;  
}
```

---

- معادل در اسمبلی

Listing 2.25: Assembly loop

---

```
MOV i, 1  
MOV sum, 0  
LOOP_START:  
ADD sum, i  
INC i  
CMP i, 10  
JLE LOOP_START
```

---

۳.۷.۲ چاپ (Print)

- کد R

Listing 2.26: R Print

---

```
print(sum)
```

---

- معادل در C

Listing 2.27: C Print

---

```
printf("%d", sum);
```

---



## فصل ۳

# متغیرها و نوع های داده ای

### ۱.۳ انقیاد

#### ۱.۱.۳ انقیاد نوع و مقدار

انقیاد نوع (Type Binding)

نوع گذاری پویا (Dynamic Typing): در R، متغیرها نیازی به تعریف نوع در زمان اعلان ندارند. نوع متغیر در زمان اختصاص مقدار (assignment) تعیین می شود. این نوع گذاری به صورت زمان اجرا (Runtime) انجام می شود. نوع متغیر می تواند در طول برنامه تغییر کند.  
مثال:

Listing 3.1: Type Binding

```
x <- 42      # integer  
x <- "Hello" # character
```

در این مثال، متغیر x ابتدا یک عدد صحیح (integer) است، اما بعداً به یک مقدار رشته ای (character) تغییر می یابد. این نشان دهنده نوع گذاری پویا است.

### ۲.۱.۳ انقیاد مقدار (Value Binding)

مقداردهی متغیرها در زمان تخصیص مقدار به متغیر انجام می شود. از عملگر <- یا = برای اختصاص مقدار به متغیر استفاده می شود. انقیاد مقدار در R در زمان اجرا (Runtime) انجام می شود، زیرا مقادیر می توانند تغییر کنند.

Listing 3.2: Value Binding

```
y <- 10      # Initialization  
y <- y + 5    # New value
```

در این مثال، مقدار y ابتدا ۱۰ است و بعداً به ۱۵ تغییر می کند.

## ۳.۱.۳ تعریف متغیرها

صریح یا ضمنی بودن تعاریف: در زبان R، تعاریف متغیرها می‌توانند صریح (Explicit) یا ضمنی (Implicit) باشند.

## تعریف صریح

متغیرها به وضوح توسط کاربر تعریف می‌شوند. از عملگر تخصیص ( $\leftarrow$  یا  $=$ ) برای این کار استفاده می‌شود.

## Listing 3.3: Explicit Variable

---

```
my_var <- 20
```

---

## تعریف ضمنی

برخی اشیاء در R به صورت ضمنی در زمان استفاده از آنها تعریف می‌شوند، مثلاً در توابع.

## Listing 3.4: Implicit Variable

---

```
for (i in 1:5) {  
  print(i)  
}
```

---

در اینجا متغیر  $i$  بدون تعریف صریح، به طور خودکار در حلقه `for` تعریف می‌شود.

## ۴.۱.۳ تاثیر پویا بودن انقیاد نوع و مقدار

انعطاف‌پذیری زبان R به دلیل انقیاد پویا امکان تعریف متغیرها بدون در نظر گرفتن نوع را فراهم می‌کند، اما ممکن است خطاهای نوع (type error) را در زمان اجرا افزایش دهد.

## Listing 3.5: Type error

---

```
x <- 10  
x <- x + "5" # Error: Unable to add number and string.
```

---

## ۵.۱.۳ کاربرد و مزایا

- انعطاف‌پذیری: مناسب برای تحلیل داده و کدنویسی سریع.
- سادگی: نیاز به تعریف دقیق نوع و مقدار ندارد.
- کاهش پیچیدگی: مناسب برای کاربران غیر برنامه‌نویس (مانند تحلیل‌گران داده).

## ۶.۱.۳ نتیجه گیری

در زبان R:

۱. انقیاد نوع و مقدار به صورت پویا و در زمان اجرا (Runtime) انجام می شود.
۲. تعاریف متغیرها می توانند صریح (با تخصیص مقدار) یا ضمنی (در توابع یا ساختارهای کنترلی) باشند.
۳. انعطاف پذیری انقیاد نوع و مقدار، R را به زبانی ساده و مناسب برای تحلیل داده و آزمایش سریع ایده ها تبدیل کرده است.

## ۲.۳ بررسی انواع متغیرها

## ۱.۲.۳ متغیرهای ایستا (Static Variables)

در زبان R، متغیرهای ایستا به صورت صریح وجود ندارند. با این حال، متغیرهایی که خارج از بدنه توابع تعریف می شوند (مانند متغیرهای سراسری)، ممکن است به صورت ایستا در نظر گرفته شوند. این متغیرها در طول اجرای برنامه در حافظه باقی می مانند. مثال:

Listing 3.6: Static variables

---

```
global_var <- 100 # Global variable

my_function <- function() {
  print(global_var) # Access to a global variable
}

my_function()
```

---

## • پیاده سازی

این متغیرها در فضایی به نام محیط جهانی (Global Environment) ذخیره می شوند. محیط جهانی در طول اجرای برنامه فعال است و متغیرها تا زمانی که برنامه در حال اجرا باشد، باقی می مانند.

## • سرعت تخصیص

متغیرهای ایستا سریع تر از متغیرهای پویا تخصیص داده می شوند، زیرا در آغاز اجرا تخصیص می یابند.

## ۲.۲.۳ متغیرهای پویا در پشته (Dynamic Stack Variables)

متغیرهای تعریف شده داخل توابع، معمولاً در پشته ذخیره می شوند. این متغیرها فقط در طول اجرای تابع معتبر هستند. مثال:

Listing 3.7: Dynamic stack variables

---

```
my_function <- function() {
  local_var <- 10 # Local variable
  print(local_var)
}

my_function()
# Unreachable error outside the function
# print(local_var)
```

---

- پیاده سازی

این متغیرها در محیط تابع (Function Environment) ذخیره می شوند که در پشته مدیریت می شود. زمانی که تابع تمام می شود، محیط آن نیز از بین می رود.

- سرعت تخصیص

تخصیص در پشته سریع است، زیرا فقط به تغییر اشاره گر پشته نیاز دارد. آزادسازی حافظه نیز خودکار و سریع انجام می شود.

### ۳.۲.۳ متغیرهای پویا در هیپ به طور صریح

در R، هرگاه متغیری مانند یک بردار، لیست یا داده ای بزرگ تعریف شود، این متغیرها در هیپ ذخیره می شوند. کاربر به طور مستقیم از تخصیص حافظه آگاه نیست، اما تخصیص هیپ توسط مفسر R انجام می شود. مثال:

Listing 3.8: Dynamic variables on heap explicitly

---

```
large_vector <- rep(1, 1e6) # Large vector
print(object.size(large_vector)) # Check the size at the heap
```

---

- پیاده سازی

هیپ در R برای تخصیص حافظه اشیاء بزرگ تر و پیچیده تر (مانند بردارها و لیست ها) استفاده می شود. مدیریت حافظه از طریق جمع آوری زباله (Garbage Collection) انجام می شود.

- سرعت تخصیص

تخصیص هیپ نسبت به پشته کندتر است، زیرا شامل درخواست های پیچیده تر از سیستم عامل و جمع آوری زباله است.

### ۴.۲.۳ متغیرهای پویا در هیپ به طور ضمنی

این نوع تخصیص زمانی رخ می دهد که R به صورت خودکار برای متغیرهای کوچک یا موقتی نیز از هیپ استفاده کند. برای مثال، مقادیری که در عملیات های موقتی ایجاد می شوند. مثال:

Listing 3.9: Dynamic variables on heap implicitly

```
result <- sum(1:100) # Create a temporary value for the range
print(result)
```

- پیاده سازی  
R ممکن است برای مقادیر موقتی نیز حافظه ای در هیپ تخصیص دهد، اما این حافظه به سرعت پس از استفاده آزاد می شود.
- سرعت تخصیص  
سرعت این تخصیص نسبت به تخصیص مستقیم در پشته کمتر است، اما R از بهینه سازی برای کاهش تاثیر عملکرد استفاده می کند.

### ۵.۲.۳ مقایسه انواع متغیر

نوع تخصیص	سرعت تخصیص	دلایل
ایستا (Static)	بسیار سریع	تخصیص در زمان شروع برنامه انجام می شود.
پویا در پشته (Stack)	سریع	مدیریت ساده با تغییر اشاره گر پشته.
پویا در هیپ (Heap)	متوسط تا کند	نیاز به درخواست های سیستم عامل و مدیریت زباله.
هیپ ضمنی (Implicit) (Heap)	کندتر	پیچیدگی در تخصیص موقت و آزادسازی سریع.

جدول ۱.۳: مقایسه انواع تخصیص و سرعت آن ها

### ۶.۲.۳ نتیجه گیری

زبان R بیشتر از تخصیص پویا در هیپ استفاده می کند، اما تخصیص پشته نیز برای متغیرهای محلی وجود دارد. سرعت تخصیص متغیرها به نوع تخصیص و پیچیدگی اشیاء بستگی دارد. برای داده های کوچک و ساده، پشته سریع ترین گزینه است، در حالی که برای داده های بزرگ یا پیچیده، هیپ استفاده می شود که کندتر است. R با استفاده از جمع آوری زباله و بهینه سازی داخلی، تاثیر کندی تخصیص در هیپ را کاهش می دهد.

## ۳.۳ حوزه های تعریف زبان

### ۱.۳.۳ حوزه تعریف ایستا (Static Scope)

در حوزه ایستا، متغیرها بر اساس محل تعریف شان در کد (نه محل فراخوانی) تعیین می شوند. R از این نوع حوزه تعریف پشتیبانی می کند. وقتی یک متغیر در داخل یک تابع تعریف نشده باشد، R به محیط

لغوی تابع نگاه می‌کند (محیطی که تابع در آن تعریف شده است) و متغیر را از آنجا می‌گیرد. مثال حوزه ایستا:

Listing 3.10: Static scope

```
x <- 10 # variable in the global environment

outer_function <- function() {
  inner_function <- function() {
    return(x) # x takes from the lexical environment
  }
  return(inner_function())
}

print(outer_function()) # output: 10
```

در این مثال، متغیر x در محیط جهانی تعریف شده است و تابع inner\_function آن را بر اساس محیط لغوی تابع پیدا می‌کند.

### ۲.۳.۳ حوزه تعریف پویا (Dynamic Scope)

در حوزه پویا، متغیرها بر اساس محل فراخوانی تابع (نه محل تعریف) تعیین می‌شوند. R به طور پیش فرض این نوع حوزه را پشتیبانی نمی‌کند، اما می‌توان آن را با استفاده از ویژگی‌های مدیریت محیط‌ها (مانند parent.frame() یا assign()) شبیه‌سازی کرد. مثال:

Listing 3.11: Dynamic scope

```
dynamic_function <- function() {
  print(x) # x takes from the calling environment
}

caller_function <- function() {
  x <- 20 # Define x in this environment
  dynamic_function()
}

caller_function() # output: 20
```

در این مثال، با استفاده از ویژگی‌های محیط، رفتار حوزه پویا شبیه‌سازی شده است.

### ۳.۳.۳ افزودن حوزه پویا به زبان R

اگر بخواهیم حوزه پویا را به R اضافه کنیم، باید تغییرات زیر اعمال شود:

#### ۱. مدیریت محیط اجرا (Execution Environment)

به جای استفاده از محیط لغوی، تابع باید به محیط فراخوانی دسترسی داشته باشد. این کار با تغییر نحوه ذخیره و جستجوی متغیرها در محیط‌ها ممکن می‌شود.

## ۲. پیاده سازی نمونه

برای شبیه سازی این تغییر، می توان از یک مفسر اصلاح شده یا تابعی کمکی استفاده کرد.

نمونه کد برای حوزه پویا:

Listing 3.12: Adding a dynamic scope to R

---

```
dynamic_scope <- function() {
  print(x) # Gets x from the calling environment
}

caller_function <- function() {
  x <- 30 # Define x in the calling environment
  evalq(dynamic_scope(), envir = environment()) # Function
    execution in the calling environment
}

caller_function() # output: 30
```

---

در اینجا، با استفاده از evalq() و انتقال محیط، رفتار حوزه پویا شبیه سازی شده است.

## ۴.۳.۳ پشتیبانی از هر دو نوع حوزه در R

R به طور مستقیم فقط حوزه ایستا دارد. حوزه پویا را می توان با استفاده از محیط های پویا (مانند parent.frame() یا evalq()) شبیه سازی کرد.

## ۵.۳.۳ بلوک ها و کلمات کلیدی حوزه در R

در R، بلوک ها با استفاده از تعریف می شوند. هیچ کلمه کلیدی خاصی برای تغییر مستقیم حوزه تعریف وجود ندارد، اما می توان با استفاده از توابع مدیریتی مانند assign()، environment()، parent.frame() و eval() به صورت ضمنی حوزه را تغییر داد. مثال بلوک:

Listing 3.13: Blocks in R

---

```
{
  y <- 50
  print(y)
}
# y is not available outside the block
# print(y) # Error: object 'y' not found
```

---

## ۶.۳.۳ نتیجه گیری

- حوزه ایستا: R به طور پیش فرض از این نوع استفاده می کند و متغیرها را از محیط تعریف پیدا می کند.
- حوزه پویا: پشتیبانی مستقیم وجود ندارد، اما می توان آن را شبیه سازی کرد.

- بلوک ها: R با تعریف می شوند و هیچ کلمه کلیدی خاصی برای مدیریت حوزه وجود ندارد، اما توابع محیطی می توانند حوزه را به طور غیرمستقیم تغییر دهند.

### ۴.۳ نوع های داده ای

#### ۱.۴.۳ نوع های داده ای پایه (Basic Data Types)

زبان R از انواع داده ای مختلفی پشتیبانی می کند که برای تحلیل داده ها، محاسبات آماری، و کار با داده های پیچیده طراحی شده اند. در اینجا، تمامی انواع داده ای در R، نحوه تخصیص آنها در حافظه، پیاده سازی، و ویژگی های هر کدام توضیح می دهیم:

##### منطقی (Logical)

- مقادیر: TRUE، FALSE، یا NA (عدم مقدار)
- کاربرد: برای شرط ها و عملگرهای بولی.
- عملگرها: & (AND)، | (OR)، ! (NOT).
- تخصیص در حافظه: معمولاً به صورت ۱ بیت یا بیشتر برای ذخیره مقدار منطقی.

مثال:

Listing 3.14: Logical data type

```
x <- TRUE
y <- FALSE
z <- x & y # FALSE
```

##### عددی (Numeric)

- مقادیر: مقادیر عددی (شامل مقادیر اعشاری)
- کاربرد: محاسبات ریاضی و آماری.
- عملگرها: +، -، \*، /، توان، %% (باقیمانده).
- تخصیص در حافظه: به صورت پیش فرض ۸ بایت (۶۴ بیت) برای هر مقدار.

مثال:

Listing 3.15: Numeric data type

```
a <- 3.14
b <- 2
result <- a + b # 5.14
```



## عدد صحیح (Integer)

- مقادیر: شامل مقادیر عددی صحیح.
- کاربرد: شمارنده ها و محاسبات عدد صحیح.
- ایجاد مقدار: با استفاده از L پس از عدد.
- تخصیص در حافظه: ۴ بایت (۳۲ بیت).

مثال:

Listing 3.16: Integer data type

---

```
int_val <- 5L
typeof(int_val) # "integer"
```

---

## رشته ای (Character)

- مقادیر: شامل مقادیر متنی.
- کاربرد: پردازش رشته ها، نام گذاری، و تحلیل متن.
- عملگرها: الحاق (paste و paste0).
- تخصیص در حافظه: به صورت آرایه ای از کاراکترها با طول متغیر.

مثال:

Listing 3.17: Character data type

---

```
str <- "Hello"
full_str <- paste(str, "World") # "Hello World"
```

---

## مختلط (Complex)

- مقادیر: شامل مقادیر مختلط  $a + bi$ .
- کاربرد: محاسبات ریاضی پیشرفته.
- عملگرها: جمع، تفریق، ضرب، تقسیم.
- تخصیص در حافظه: شامل دو مقدار عددی برای بخش حقیقی و موهومی.

مثال:

Listing 3.18: Complex data type

---

```
comp <- 3 + 2i
Im(comp) # 2
```

---

## ۲.۴.۳ ساختارهای داده ای پیچیده (Advanced Data Structures)

## برداری (Vector)

- مقادیر: مجموعه ای از مقادیر همگن.
- کاربرد: تحلیل داده و آرایه های یک بعدی.
- عملگرها: length و [ ] برای دسترسی.
- پیاده سازی: به صورت آرایه در حافظه.

مثال:

Listing 3.19: Vector data type

---

```
vec <- c(1, 2, 3)
vec[1] # 1
```

---

## ماتریس (Matrix)

- مقادیر: آرایه ای دوبعدی از مقادیر همگن.
- کاربرد: محاسبات ماتریسی.
- پیاده سازی: به صورت آرایه دوبعدی.

مثال:

Listing 3.20: Matrix data type

---

```
mat <- matrix(1:6, nrow=2)
mat[1, 2] # 3
```

---

## لیست (List)

- مقادیر: شامل مقادیر ناهمگن.
- کاربرد: ذخیره ساختارهای پیچیده.
- پیاده سازی: اشاره گرایی به مقادیر مختلف در حافظه.

مثال:

Listing 3.21: List data type

---

```
lst <- list(num=1, txt="Hello", vec=c(1,2,3))
lst$num # 1
```

---

## چارچوب داده (Data Frame)

- مقادیر: ساختاری جدولی برای داده.
- کاربرد: ساختاری جدولی برای داده.
- پیاده سازی: به صورت لیست با ستون های هم طول.

مثال:

Listing 3.22: Data Frame data type

```
df <- data.frame(A=c(1,2), B=c("X", "Y"))
df$A # A
```

## آرایه (Array)

- مقادیر: آرایه چندبعدی از مقادیر همگن.
- کاربرد: محاسبات چندبعدی.

مثال:

Listing 3.23: Array data type

```
arr <- array(1:8, dim=c(2,2,2))
arr[1,1,1] # 1
```

## ۳.۴.۳ اشاره گر ها و متغیرهای مرجع

R به طور مستقیم اشاره گر ها را در اختیار کاربر قرار نمی دهد. تمامی اشیاء در R به صورت مرجع محور (Reference-Based) مدیریت می شوند. عملگر address() برای بررسی مکان حافظه استفاده می شود.

## ۴.۴.۳ بازیافت حافظه (Garbage Collection)

R از یک جمع آوری زباله (Garbage Collector) استفاده می کند.

- پیاده سازی: از روش های ردیابی و شمارش مرجع برای آزادسازی حافظه اشیاء استفاده می شود.
- عملگر: gc() برای اجرای دستی جمع آوری زباله.

## ۵.۴.۳ رفع مشکلات نشی حافظه و اشاره گر معلق

R به دلیل مدیریت خودکار حافظه، از مشکلاتی مانند اشاره گر معلق (Dangling Pointer) جلوگیری می کند. مدیریت: اشیاء استفاده نشده توسط Garbage Collector شناسایی و آزاد می شوند.

## ۶.۴.۳ نتیجه گیری

- R طیف وسیعی از نوع های داده ای ساده و پیچیده را ارائه می دهد.
- حافظه به صورت خودکار و بهینه تخصیص می یابد.
- بازیافت حافظه به جلوگیری از مشکلات ناشی حافظه کمک می کند، و این ویژگی R را برای تحلیل داده های پیچیده ایده آل می کند.

## فصل ۴

# برنامه نویسی تابعی

زبان R از برنامه نویسی تابعی (Functional Programming) به طور کامل پشتیبانی می کند. R طراحی شده است تا با داده ها به صورت برداری (vectorized) کار کند و همین امر آن را برای عملیات تابعی بسیار مناسب می سازد. در ادامه به جزئیات سؤالات شما پاسخ داده می شود.

### ۱.۴ سازوکارهای برنامه نویسی تابعی در R

R شامل ویژگی ها و توابع متعددی برای پشتیبانی از برنامه نویسی تابعی است:

#### ۱. توابع لامبدا (Lambda Functions)

در R، می توانید به صورت مستقیم توابع ناشناس (inline functions) تعریف کنید. مثال:

Listing 4.1: Lambda Functions

```
squared <- function(x) x^2
result <- lapply(1:5, function(x) x^2) # Using a lambda
function
print(result)
```

#### ۲. ارسال تابع به تابع

توابع در R اشیاء درجه اول هستند، یعنی می توان آن ها را به عنوان آرگومان به دیگر توابع ارسال کرد. مثال:

Listing 4.2: Passing function to another function

```
apply_function <- function(f, x) {
  f(x) # Execute the input function
}
apply_function(sin, pi / 2) # Passing sin function to another
function
```

## ۳. بازگرداندن تابع از تابع

R اجازه می‌دهد که یک تابع از درون تابع دیگر بازگردانده شود. مثال:

Listing 4.3: Return a function into function

```
generate_multiplier <- function(n) {
  function(x) x * n # Return a new function
}
double <- generate_multiplier(2)
double(5) # result: 10
```

## ۴. توابع نگاشت (Map)، فیلتر (Filter) و کاهش (Reduce)

- map: توابعی مانند lapply و sapply برای اعمال یک تابع بر هر عنصر یک لیست یا بردار استفاده می‌شوند.
- filter: تابع Filter برای انتخاب عناصر مطابق با شرط استفاده می‌شود.
- reduce: تابع Reduce برای کاهش یک بردار به یک مقدار استفاده می‌شود.

مثال:

Listing 4.4: Map – Filter – Reduce

```
# Map
squared <- sapply(1:5, function(x) x^2)

# Filter
filtered <- Filter(function(x) x > 2, 1:5)

# Reduce
sum_result <- Reduce(`+`, 1:5)
```

## ۲.۴ پیاده‌سازی توابع برنامه‌نویسی تابعی در R

این توابع با استفاده از مفاهیم بردارسازی (Vectorization) و ارث‌بری از توابع پایه‌ای (Base functions) در R اجرا می‌شوند. R در پشت صحنه، این عملیات را بهینه می‌کند، مثلاً با استفاده از کدهای نوشته‌شده در زبان C برای عملکرد بهتر.

## ۳.۴ تأثیر توابع نگاشت، فیلتر و کاهش بر کارایی برنامه

استفاده از توابع تابعی (مانند sapply یا Reduce) معمولاً باعث افزایش کارایی می‌شود، زیرا این توابع در سطح پایین بهینه شده‌اند. این روش‌ها معمولاً کد خواناتر و کوتاه‌تری ایجاد می‌کنند که نگهداری آن‌ها آسان‌تر است. با این حال، برای مجموعه داده‌های کوچک ممکن است اختلاف کارایی قابل ملاحظه‌ای وجود نداشته باشد.

## ۴.۴ مقایسه با برنامه نویسی رویه ای (Procedural Programming)

برنامه نویسی تابعی می تواند از نظر کارایی و سادگی کد برتری داشته باشد، اما این به اندازه داده ها و کاربرد بستگی دارد. مثال:

Listing 4.5: Map - Filter - Reduce

---

```
# Functional (Using Reduce)
system.time({
  result_func <- Reduce(`+`, 1:1e6)
})

# Procedural (Using Loop)
system.time({
  result_loop <- 0
  for (i in 1:1e6) {
    result_loop <- result_loop + i
  }
})
```

---

نتایج نشان می دهد که نسخه تابعی معمولاً سریع تر یا مشابه است، زیرا از بهینه سازی داخلی R بهره می برد.

## ۵.۴ جمع بندی

R به طور گسترده برنامه نویسی تابعی را پشتیبانی می کند و این روش می تواند به سادگی کد و گاهی افزایش کارایی منجر شود. با این حال، برای تصمیم گیری درباره بهترین روش، حجم داده ها و کاربرد خاص پروژه را باید در نظر گرفت.

## فصل ۵

# برنامه نویسی رویه‌ای

زبان R از برنامه‌نویسی رویه‌ای (Procedural Programming) به خوبی پشتیبانی می‌کند. در این سبک برنامه‌نویسی، کد به صورت ترتیبی و از طریق توابع و زیربرنامه‌ها (subroutines) سازماندهی می‌شود. در ادامه به جزئیات سازوکارهای برنامه‌نویسی رویه‌ای در R پرداخته می‌شود:

### ۱.۵ زیربرنامه‌ها (Subroutines)

در R، زیربرنامه‌ها همان توابع هستند که می‌توانند کد تکراری یا پیچیده را درون خود مدیریت کنند. این توابع با استفاده از کلمه کلیدی function تعریف می‌شوند. مثال:

Listing 5.1: Subroutines

```
calculate_area <- function(length, width) {  
  return(length * width) # return value  
}  
  
area <- calculate_area(5, 3) # Subroutine call  
print(area) # output: 15
```

### ۲.۵ روش‌های ارسال متغیرها به توابع

در R، متغیرها می‌توانند به دو روش به توابع ارسال شوند:

#### ۱. ارسال بر اساس مقدار (Pass by Value)

R به طور پیش‌فرض، متغیرها را به صورت "بر اساس مقدار" به توابع ارسال می‌کند. این بدان معناست که مقدار کپی می‌شود و تغییرات داخل تابع بر متغیر اصلی تأثیر نمی‌گذارد. مثال:

Listing 5.2: Pass by Value

```
modify_value <- function(x) {  
  x <- x + 1
```



```

    return(x)
}

a <- 10
result <- modify_value(a)
print(a) # Original value does not change (output: 10)

```

## ۲. ارسال بر اساس مرجع (Pass by Reference)

برای ارسال متغیرها بر اساس مرجع، از اشیاء خاصی مانند لیست‌ها (lists) یا محیط‌ها (environments) استفاده می‌شود. مثال:

Listing 5.3: Pass by Reference

```

modify_reference <- function(env) {
  env$a <- env$a + 1 # Changing the value in the environment
}

e <- new.env()
e$a <- 10
modify_reference(e)
print(e$a) # value changes (output: 11)

```

## ۳.۵ برنامه‌نویسی عمومی (Generic Programming)

R از توابع عمومی پشتیبانی می‌کند، به این معنا که توابع می‌توانند با انواع مختلف داده‌ها (اعداد، رشته‌ها، بردارها، و ...) کار کنند. R به صورت پویا نوع داده را تشخیص می‌دهد و نیازی به تعریف نوع ورودی وجود ندارد. مثال:

Listing 5.4: Generic Programming

```

print_value <- function(x) {
  print(x) # The function works for any data type
}

print_value(42)      # integer
print_value("Hello") # string
print_value(TRUE)    # logical

```

علاوه بر این، R از سیستم‌های عمومی مانند S3، S4 و Reference Classes برای مدیریت رفتار توابع برای انواع مختلف داده‌ها استفاده می‌کند. مثال از روش S3:

Listing 5.5: S3

```

print.custom <- function(x) {
  cat("Custom value:", x, "\n")
}

```

---

```
x <- 100
class(x) <- "custom"
print(x) # Generic function call based on data type
```

---

## ۴.۵ سایر ویژگی‌های برنامه‌نویسی رویه‌ای

- شرط‌ها و حلقه‌ها
- R شامل ساختارهای شرطی (if, else) و حلقه‌های تکرار (for, while, repeat) برای اجرای منطقی و ترتیبی کد است.

Listing 5.6: for

---

```
for (i in 1:5) {
  print(i)
}
```

---

- توابع پیش‌فرض (Default Arguments)
- در R می‌توان آرگومان‌های پیش‌فرض برای توابع تعریف کرد. مثال:

Listing 5.7: Default Arguments

---

```
greet <- function(name = "World") {
  cat("Hello,", name, "!\n")
}

greet()           # output: Hello, World!
greet("Alice")   # output: Hello, Alice!
```

---

- بازه‌های متغیر (Variable Scope)
- متغیرها در R دارای Local Scope یا Global Scope هستند. مثال:

Listing 5.8: Variable Scope

---

```
x <- 10 #
scope_example <- function() {
  x <- 5 #
  return(x)
}
print(scope_example()) # output: 5
print(x)               # output: 10
```

---

## ۵.۵ مقایسه برنامه‌نویسی رویه‌ای با تابعی

- برنامه‌نویسی رویه‌ای برای وظایف ترتیبی و ساده‌تر مناسب است.

- برنامه‌نویسی تابعی برای عملیات پیچیده‌تر و داده‌های بزرگ کارایی بیشتری دارد.
- در بسیاری از موارد، برنامه‌نویسی تابعی در R به دلیل استفاده از بهینه‌سازی داخلی و بردارها سریع‌تر از حلقه‌های تکرار عمل می‌کند.

## ۶.۵ جمع‌بندی

R یک زبان انعطاف‌پذیر است که برنامه‌نویسی رویه‌ای را به طور کامل پشتیبانی می‌کند. این زبان ابزارهایی مانند توابع، روش‌های ارسال متغیر، توابع عمومی، و ساختارهای کنترلی را فراهم می‌کند که به توسعه کدهای خوانا و قابل نگهداری کمک می‌کنند.

## فصل ۶

# برنامه‌نویسی شیء‌گرا

زبان R از برنامه‌نویسی شیء‌گرا (Object-Oriented Programming یا OOP) پشتیبانی می‌کند. R از چندین سیستم برای مدیریت برنامه‌نویسی شیء‌گرا استفاده می‌کند، از جمله S3، S4، Reference، و Classes. این سیستم‌ها هر کدام دارای ویژگی‌ها و سازوکارهای خاصی هستند که در ادامه توضیح داده می‌شوند.

### ۱.۶ ساختارهای موجود برای برنامه‌نویسی شیء‌گرا

#### • سیستم S3

S3 یک سیستم ساده و انعطاف‌پذیر برای شیء‌گرایی در R است.

– S3 از کلاس‌های غیررسمی و توابع عمومی (Generic Functions) استفاده می‌کند.

– ساختار یک شیء S3 معمولاً لیستی است که با یک کلاس مشخص شده است.

مثال:

Listing 6.1: S3 OOP

```
# Defining an S3 object
person <- list(name = "Ali", age = 25)
class(person) <- "Person"

# Define a public function
print.Person <- function(x) {
  cat("Name:", x$name, "\nAge:", x$age, "\n")
}

print(person)
```

#### • سیستم S4

S4 سیستم رسمی‌تر و قدرتمندتری است که از تعریف دقیق کلاس‌ها و متدها پشتیبانی می‌کند.

- نیاز به تعریف صریح کلاس‌ها با استفاده از `setClass`.
- متدها با استفاده از `setMethod` و `setGeneric` تعریف می‌شوند.

مثال:

Listing 6.2: S3 OOP

```
# Defining a class S4
setClass("Person",
slots = list(name = "character", age = "numeric"))

# Create an object of class S4
person <- new("Person", name = "Ali", age = 25)

# Access to slots
print(person@name)
```

#### • Reference Classes

این سیستم از OOP کلاسیک (مشابه زبان‌هایی مانند Java یا Python) پشتیبانی می‌کند.

- امکان تعریف متغیرهای خصوصی و متدها.
- اشیاء به صورت ارجاعی مدیریت می‌شوند (یعنی تغییرات در متغیرها روی شیء اصلی اعمال می‌شوند).

مثال:

Listing 6.3: Reference Classes

```
Person <- setRefClass("Person",
fields = list(name = "character", age = "numeric"),
methods = list(
greet = function() {
  cat("Hello, my name is", name, "and I am", age, "years
    old.\n")
}
))

# Create object
person <- Person$new(name = "Ali", age = 25)
person$greet()
```

#### • سیستم R6

R6 یک سیستم مدرن‌تر است که برای برنامه‌نویسی شیء‌گرا در R طراحی شده است.

- متدهای R6 به طور مستقیم به اشیاء متصل هستند.
- برخلاف S3 و S4، نیازی به تابع عمومی نیست.

مثال:

Listing 6.4: S6 OOP

---

```
library(R6)

Person <- R6Class("Person",
  public = list(
    name = NULL,
    age = NULL,
    initialize = function(name, age) {
      self$name <- name
      self$age <- age
    },
    greet = function() {
      cat("Hello, my name is", self$name, "and I am", self$age,
        "years old.\n")
    }
  )
))

# Create object
person <- Person$new(name = "Ali", age = 25)
person$greet()
```

---

## ۲.۶ چندریختی (Polymorphism)

R از چندریختی به خوبی پشتیبانی می‌کند، به ویژه در سیستم‌های S3 و S4. در S3 و S4، توابع عمومی مانند print بر اساس کلاس شیء رفتار متفاوتی از خود نشان می‌دهند.

Listing 6.5: Polymorphism

---

```
# Definition of two different classes
setClass("Rectangle", slots = list(length = "numeric", width =
  "numeric"))
setClass("Circle", slots = list(radius = "numeric"))

# General method definition for area calculation
setGeneric("area", function(obj) standardGeneric("area"))
setMethod("area", "Rectangle", function(obj) obj@length * obj@width)
setMethod("area", "Circle", function(obj) pi * obj@radius^2)

# use
rect <- new("Rectangle", length = 5, width = 3)
circle <- new("Circle", radius = 2)
area(rect) # output: 15
area(circle) # output: 12.566
```

---

### ۳.۶ وراثت (Inheritance)

- در S3، وراثت از طریق لیست کلاس‌ها (کلاس والد) انجام می‌شود.
- در S4، وراثت به طور رسمی با استفاده از contains تعریف می‌شود.
- در Reference Classes و R6، وراثت به صورت سلسله‌مراتبی تعریف می‌شود.

مثال در S4:

Listing 6.6: Inheritance

---

```
setClass("Shape", slots = list(color = "character"))
setClass("Rectangle", contains = "Shape", slots = list(length =
  "numeric", width = "numeric"))

# Create a rectangle
rect <- new("Rectangle", color = "blue", length = 5, width = 3)
rect@color # Inherit the Color property from the Shape class
```

---

### ۴.۶ مدیریت اشیاء در حافظه

- در سیستم‌های S3 و S4، اشیاء به صورت کپی مدیریت می‌شوند، بنابراین تغییرات بر روی یک شیء معمولاً به نسخه کپی اعمال می‌شود.
- در سیستم‌های Reference Classes و R6، اشیاء به صورت ارجاعی (Reference) مدیریت می‌شوند، یعنی تغییرات مستقیماً روی شیء اصلی اعمال می‌شود.

### ۵.۶ جمع‌بندی

زبان R از برنامه‌نویسی شی‌گرا با چندین سیستم (S3، S4، Reference Classes، و R6) پشتیبانی می‌کند.

- S3 برای کاربردهای ساده و انعطاف‌پذیر مناسب است.
  - S4 برای مواردی که به دقت بیشتر در تعریف کلاس‌ها و متدها نیاز است.
  - Reference Classes و R6 برای برنامه‌های پیچیده‌تر و نیازمند مدیریت دقیق‌تر طراحی شده‌اند.
- این ویژگی‌ها R را به زبانی قدرتمند برای توسعه پروژه‌های شی‌گرا تبدیل می‌کند.

## فصل ۷

# برنامه‌نویسی همروند

زبان R از برنامه‌نویسی همروند (Concurrent Programming) پشتیبانی می‌کند. این قابلیت در R از طریق بسته‌های مختلف و سازوکارهای گوناگونی مانند رشته‌ها (Threads)، ارسال پیام (Message Passing)، قفل‌ها (Locks)، و سمافورها (Semaphores) پیاده‌سازی شده است. در ادامه به توضیحات مربوط به این سازوکارها پرداخته می‌شود.

### ۱.۷ رشته‌ها (Threads)

R به صورت پیش‌فرض از رشته‌های چندگانه (Multithreading) پشتیبانی نمی‌کند، زیرا برای اجرای تک‌رشته‌ای (Single Threaded) طراحی شده است. با این حال، برای کارهای موازی و همروند از بسته‌های خارجی مانند parallel و future استفاده می‌شود.  
مثال: Multicore Processing

Listing 7.1: Multicore Processing

```
library(parallel)

# Define a function to process
task <- function(x) {
  Sys.sleep(2) # Simulation of a long operation
  return(x^2)
}

# Concurrent execution using processes
result <- mclapply(1:4, task, mc.cores = 4)
print(result)
```

### ۲.۷ قفل‌ها (Locks) و سمافورها (Semaphores)

برای مدیریت همزمانی و جلوگیری از شرایط رقابتی (Race Conditions)، می‌توان از قفل‌ها یا سمافورها استفاده کرد. در R این قابلیت از طریق بسته‌هایی مانند parallel و future قابل دسترسی است.  
مثال: استفاده از قفل‌ها با parallel



Listing 7.2: Locks in parallel

---

```
library(parallel)

# Create a lock
lock <- makeCluster(1) #
clusterEvalQ(lock, { counter <- 0 }) #

# Increase shared variable value in safe mode
clusterExport(lock, "counter")
clusterEvalQ(lock, { counter <- counter + 1 })
stopCluster(lock)
```

---

### ۳.۷ ارسال پیام (Message Passing)

R از مکانیزم ارسال پیام برای ارتباط بین پردازنده‌ها پشتیبانی می‌کند. این قابلیت عمدتاً با بسته‌هایی مانند `parallel` و `future` فراهم می‌شود.  
مثال: ارسال پیام با استفاده از `makeCluster`

Listing 7.3: Message Passing in makeCluster

---

```
library(parallel)

# Create a cluster
cl <- makeCluster(2)

# Send and receive data between nodes
clusterExport(cl, "task")
result <- parLapply(cl, 1:2, function(x) x * 2)
print(result)

# Stop the cluster
stopCluster(cl)
```

---

### ۴.۷ مدیریت همروندی با بسته‌های خاص

#### • بسته future

بسته `future` یک رابط ساده برای همروندی و محاسبات موازی فراهم می‌کند و از چندین backend (مانند `Multicore` و `Multisession`) پشتیبانی می‌کند.

Listing 7.4: future package

---

```
library(future)

plan(multicore, workers = 4) # Set for concurrent processing
result <- future_lapply(1:4, function(x) x^2)
```

---

---

```
print(result)
```

---

- بسته foreach

بسته foreach برای اجرای حلقه‌های همروند استفاده می‌شود.

Listing 7.5: foreach package

---

```
library(foreach)
library(doParallel)

cl <- makeCluster(4)
registerDoParallel(cl)

# Execution of the parallel loop
result <- foreach(i = 1:4, .combine = c) %dopar% {
  i^2
}
print(result)

stopCluster(cl)
```

---

## ۵.۷ مقایسه سمافورها و قفل‌ها در R

- قفل‌ها (Locks): برای کنترل دسترسی به منابع مشترک استفاده می‌شوند. بسته‌های parallel و future این قابلیت را فراهم می‌کنند.
- سمافورها (Semaphores): به طور مستقیم در R پشتیبانی نمی‌شوند، اما می‌توان از قفل‌ها برای پیاده‌سازی رفتار مشابه استفاده کرد.

## ۶.۷ سایر سازوکارهای برنامه‌نویسی همروند

- اجرای غیرهمزمان (Asynchronous Execution): از طریق بسته‌هایی مانند async و future امکان‌پذیر است.
- مدیریت حافظه مشترک: R به صورت پیش‌فرض از حافظه مشترک (Shared Memory) پشتیبانی نمی‌کند، اما بسته‌هایی مانند bigmemory این قابلیت را فراهم می‌کنند.

## ۷.۷ مثالی کامل از برنامه‌نویسی همروند در R

ترکیب اجرای همروند و قفل‌ها:

Listing 7.6: Combination of concurrent execution and locks

---

```
library(parallel)
```

```

# Define a shared variable
shared_counter <- 0

# Create a lock to manage access
cl <- makeCluster(2)

clusterExport(cl, c("shared_counter"))
clusterEvalQ(cl, {
  library(parallel)
  increment <- function(x) {
    Sys.sleep(1)
    x + 1
  }
})

# Run concurrently
result <- parLapply(cl, 1:4, function(x) increment(shared_counter))
print(result)

stopCluster(cl)

```

## ۸.۷ جمع‌بندی

زبان R از طریق بسته‌های داخلی و خارجی مانند parallel، future و foreach از برنامه‌نویسی همروند پشتیبانی می‌کند. این ابزارها قابلیت‌هایی مانند قفل‌ها، ارسال پیام و مدیریت پردازش‌های موازی را فراهم می‌کنند. با این حال، برای پروژه‌های بسیار پیچیده، زبان‌هایی مانند Python یا Java ممکن است امکانات بیشتری برای مدیریت پیشرفته همروندی داشته باشند.

## فصل ۸

# پیاده سازی چند الگوریتم معروف با R

### ۱.۸ جستجوی دودویی (Binary Search)

- هدف: پیدا کردن موقعیت یک عنصر در یک آرایه مرتب شده.
- روش کار:
  - ابتدا وسط آرایه را بررسی می کند.
  - اگر مقدار وسط همان مقدار هدف باشد، موقعیت را برمی گرداند.
  - اگر مقدار وسط کمتر از مقدار هدف باشد، جستجو را به نیمه ی راست ادامه می دهد.
  - اگر مقدار وسط بیشتر باشد، جستجو در نیمه ی چپ انجام می شود.
- پیچیدگی زمانی:  $O(\log n)$
- کاربرد: مناسب برای داده های مرتب شده.

Listing 8.1: Binary Search

```
# 1. Binary Search
binary_search <- function(vec, target) {
  left <- 1
  right <- length(vec)

  while (left <= right) {
    mid <- floor((left + right) / 2)

    if (vec[mid] == target) {
      return(mid)
    } else if (vec[mid] < target) {
      left <- mid + 1
    } else {
      right <- mid - 1
    }
  }
}
```

```

    }

    return(-1) # Element not found
}

# Test Binary Search
binary_search(c(1, 3, 5, 7, 9), 5)

```

## ۲.۸ مرتب سازی حبابی (Bubble Sort)

- هدف: مرتب کردن یک آرایه به صورت صعودی.
- روش کار:
- هر عنصر را با عنصر مجاور مقایسه می کند و اگر لازم باشد، جای آن ها را عوض می کند.
- این کار را تا زمانی ادامه می دهد که آرایه کاملاً مرتب شود.
- پیچیدگی زمانی:  $O(n^2)$  در بدترین حالت.
- کاربرد: به دلیل سادگی برای آموزش مرتب سازی مناسب است، اما در عمل کند است.

Listing 8.2: Bubble Sort

```

# 2. Bubble Sort
bubble_sort <- function(vec) {
  n <- length(vec)

  for (i in 1:(n - 1)) {
    for (j in 1:(n - i)) {
      if (vec[j] > vec[j + 1]) {
        temp <- vec[j]
        vec[j] <- vec[j + 1]
        vec[j + 1] <- temp
      }
    }
  }

  return(vec)
}

# Test Bubble Sort
bubble_sort(c(64, 34, 25, 12, 22, 11, 90))

```

### ۳.۸ مرتب سازی سریع (Quick Sort)

- هدف: مرتب کردن آرایه با استفاده از تقسیم و غلبه (Divide and Conquer).
- روش کار:

- یک عنصر به عنوان محور (Pivot) انتخاب می شود.
- عناصر کوچکتر از محور به یک بخش و بزرگترها به بخش دیگر تقسیم می شوند.
- هر بخش به صورت بازگشتی مرتب می شود.

- پیچیدگی زمانی

- بهترین حالت:  $O(n \log n)$
- بدترین حالت:  $O(n^2)$  (وقتی آرایه تقریباً مرتب باشد).
- کاربرد: یکی از سریع ترین روش ها برای مرتب سازی در عمل.

Listing 8.3: Quick Sort

---

```
# 3. Quick Sort
quick_sort <- function(vec) {
  if (length(vec) <= 1) {
    return(vec)
  }

  pivot <- vec[1]
  less <- vec[vec < pivot]
  equal <- vec[vec == pivot]
  greater <- vec[vec > pivot]

  return(c(quick_sort(less), equal, quick_sort(greater)))
}

# Test Quick Sort
quick_sort(c(10, 7, 8, 9, 1, 5))
```

---

### ۴.۸ الگوریتم دایکسترا (Dijkstra's Algorithm)

- هدف: پیدا کردن کوتاه ترین مسیر از یک گره مبدأ به تمام گره های دیگر در یک گراف وزندار.
- روش کار:

- فاصله مبدأ از خودش صفر و فاصله از سایر گره ها را بی نهایت فرض می کند.

- گره‌هایی که هنوز بازدید نشده‌اند، بررسی می‌شوند و فاصله کوتاه‌تر به‌روزرسانی می‌شود.
- این فرآیند تا زمانی ادامه می‌یابد که تمام گره‌ها بازدید شوند.
- پیچیدگی زمانی:  $O(V^2)$  (با ماتریس مجاورت)، یا  $O((E + V) \log V)$  (با استفاده از صف اولویت).
- کاربرد: برای مسائل مسیر کوتاه، مانند مسیریابی در نقشه.

Listing 8.4: Dijkstra's Algorithm

```
# 4. Dijkstra's Algorithm
library(igraph)

dijkstra <- function(graph, source) {
  distances <- rep(Inf, vcount(graph))
  distances[source] <- 0
  visited <- rep(FALSE, vcount(graph))

  while (any(!visited)) {
    current <- which.min(ifelse(visited, Inf, distances))
    visited[current] <- TRUE

    neighbors <- neighbors(graph, current, mode = "out")
    for (neighbor in neighbors) {
      edge_weight <- E(graph)[get.edge.ids(graph, c(current,
        neighbor))][weight]
      distances[neighbor] <- min(distances[neighbor],
        distances[current] + edge_weight)
    }
  }

  return(distances)
}

# Create a test graph
g <- graph(edges = c(1, 2, 1, 2, 3, 2, 1, 3, 4), n = 3, directed =
  TRUE)
E(g)$weight <- c(1, 2, 4)

# Test Dijkstra's Algorithm
dijkstra(g, 1)
```

## ۵.۸ مرتب‌سازی ادغامی (Merge Sort)

- هدف: مرتب کردن آرایه با استفاده از رویکرد تقسیم و غلبه.
- روش کار:

- آرایه را به دو نیمه تقسیم می‌کند.
- هر نیمه را به صورت بازگشتی مرتب می‌کند.
- دو نیمه مرتب‌شده را با هم ادغام می‌کند.

- پیچیدگی زمانی:  $O(n \log n)$
- کاربرد: بسیار کارآمد برای داده‌های بزرگ.

## Listing 8.5: Merge Sort

```
# 5. Merge Sort
merge_sort <- function(vec) {
  if (length(vec) <= 1) {
    return(vec)
  }

  mid <- floor(length(vec) / 2)
  left <- merge_sort(vec[1:mid])
  right <- merge_sort(vec[(mid + 1):length(vec)])

  merge <- function(left, right) {
    result <- c()
    while (length(left) > 0 && length(right) > 0) {
      if (left[1] <= right[1]) {
        result <- c(result, left[1])
        left <- left[-1]
      } else {
        result <- c(result, right[1])
        right <- right[-1]
      }
    }
    return(c(result, left, right))
  }

  return(merge(left, right))
}

# Test Merge Sort
merge_sort(c(38, 27, 43, 3, 9, 82, 10))
```

## ۶.۸ مسئله کوله‌پشتی (0/1 Knapsack Problem)

- هدف: پیدا کردن ترکیبی از آیتم‌ها که بیشترین ارزش را در محدودیت وزن ارائه دهد.
- روش کار:



- از رویکرد برنامه‌ریزی پویا (Dynamic Programming) استفاده می‌کند.
- یک ماتریس برای ذخیره بهترین مقدار ممکن در هر وزن و آیتم تا لحظه کنونی ایجاد می‌کند.
- پیچیدگی زمانی:  $O(n * W)$ ، که  $n$  تعداد آیتم‌ها و  $W$  ظرفیت کوله‌پشتی است.
- کاربرد: در مسائلی مانند بهینه‌سازی سرمایه‌گذاری یا مدیریت منابع.

---

#### Listing 8.6: 0/1 Knapsack Problem

---

```
# 6. Knapsack Problem (0/1)
knapsack <- function(weights, values, capacity) {
  n <- length(weights)
  dp <- matrix(0, n + 1, capacity + 1)

  for (i in 1:n) {
    for (w in 1:capacity) {
      if (weights[i] <= w) {
        dp[i + 1, w + 1] <- max(dp[i, w + 1], dp[i, w -
          weights[i] + 1] + values[i])
      } else {
        dp[i + 1, w + 1] <- dp[i, w + 1]
      }
    }
  }

  return(dp[n + 1, capacity + 1])
}

# Test Knapsack
knapsack(c(1, 2, 3), c(6, 10, 12), 5)
```

---

## منابع

- [۱] The R Project for Statistical Computing: وبسایت رسمی زبان R که مستندات رسمی و جامع در مورد زبان، ساختارها، و پیاده‌سازی آن را ارائه می‌دهد.  
<https://www.r-project.org>
- [۲] R Documentation: پایگاه داده‌ای گسترده از مستندات R برای توابع و کتابخانه‌های مختلف.  
<https://www.rdocumentation.org>
- [۳] Advanced R by Hadley Wickham: کتابی معتبر درباره مفاهیم پیشرفته در R، شامل مدیریت حافظه، پیاده‌سازی داده‌ها، و مفاهیم مرتبط با حوزه تعریف.  
<https://adv-r.hadley.nz>
- [۴] R for Data Science: کتابی از Hadley Wickham و Garrett Golemund که اصول برنامه‌نویسی و تحلیل داده با R را پوشش می‌دهد.  
<https://r4ds.had.co.nz>
- [۵] R: in Collection Garbage: مستنداتی درباره جمع‌آوری زباله و مدیریت حافظه در R.  
<https://stat.ethz.ch/R-manual/R-devel/library/base/html/gc.html>
- [۶] R Programming for Data Science by Roger D. Peng: کتابی با تمرکز بر اصول برنامه‌نویسی R برای دانش داده و تحلیل آماری.  
<https://leanpub.com/rprogramming>
- [۷] Stack Overflow (R Tag): انجمنی فعال برای پرسش و پاسخ در مورد مسائل برنامه‌نویسی R.  
<https://stackoverflow.com/questions/tagged/r>
- [۸] CRAN (Comprehensive R Archive Network): مخزن رسمی R برای دانلود بسته‌ها و مستندات مرتبط.  
<https://cran.r-project.org>
- [۹] W3Schools: وبسایتی آموزشی برای یادگیری زبان R و دیگر تکنولوژی‌های برنامه‌نویسی.  
<https://www.w3schools.com/r/>